Search and Conquer (SaC) — Java Library for Searching Graphs and Game Trees

User Guide

Przemysław Klęsk and Marcin Korzeń

Faculty of Computer Science and Information Technology
West Pomeranian University of Technology
Żołnierska 49, 71-210 Szczecin, Poland
Preface

Search algorithms constitute a significant part in studies on artificial intelligence. From a historical 
perspective, *pathfinding* and *chess* come to mind as probably two most natural examples where 
search algorithms have been applicable with success. These examples are also good representatives 
for the two general groups of search problems that the SaC library attempts to solve, namely:

1. **discrete (combinatorial) optimization problems**, where a problem can be represented as a 
   *graph of states*, and finding its solution brings down to searching;

2. **two-person games problems**, where one looks for the best move or decision in a game or a 
   situation of conflict, which can be represented as a *game tree*.

Fig. 1 illustratively depicts the two groups.

As regards the first group, on one hand it includes various types of geographical or physical 
graphs where some object can move from one location to another. One may think here of examples

---

1Images acquired from the Google Images search engine.
like: shortest path finding (for different means of transport), routing problems (internet routing, vehicle routing), traversing mazes, etc. On the other hand, this group includes also a variety of abstract graphs related to puzzles or riddles, combinatorial by nature, where by certain manipulations we can change the state of some object. We would like to discover a sequence of such manipulations which brings the object to a solution state with some desired properties. One may think here of many recreation examples like: Rubik’s cube, sliding puzzle, sudoku, solitaires; but also of more practical problems in technology like: spatial packing problems (2D or 3D), optimal material cutting, arranging schedules, resource planning, etc.

As regards the second group, the straightforward examples are mind games e.g.: chess, checkers, GO, Hex, Nim, tic-tac-toe, and many others. Beside those, one may indicate some examples taken from the game theory, like the famous prisoner’s dilemma, but also: bargaining problems, political conflicts, competition wars between companies, etc.

In games, a player at a given game position (a state) typically has some number of possible moves at disposal which transfer the game to new positions. In each of these positions the opponent has a number of counter-moves, and this scheme continues. A tree structure representing the game arises in a natural manner. Due to the exponential growth of game trees, computer programs are in practice limited to going over only a small portion of a whole tree. Usually, the analysis must be ended up after just several levels of depth. Then, by assigning some numerical evaluations to terminal positions — being the consequences of moves made at the root of the tree — the algorithm is able to indicate the most promising move. It is worth to remark that not only mind games can be analyzed in this manner. Many computer games, usually strategy games, but possibly even arcade ones or shooters, can be subjected to that scheme if only rules of a game are well defined and a finite set of moves can be formulated at decision moments.

Aim and purposes

In the programmer’s eye search algorithms share many common elements, despite some differences (usually minor ones). Therefore, the main idea behind the Search and Conquer (SaC) library was to propose a set of interfaces and classes — the API — which would unify and facilitate the way search algorithms are carried out. Java was the selected language for the library. Although is it easy to look up on the Internet examples of implementations (also in Java) of some search algorithms, these examples are typically disconnected from one another, and coded with the focus on a specific problem, so with little generality.

The intention of authors was to design the SaC library in such a manner that its future users, not necessarily skilled programmers, would be able to ‘hook up’ with ease their own specific problems to the library. By that we mainly mean that to formulate and to solve a particular search problem requires possibly very little of the programming effort on the user’s part.

Currently, in SaC there are nine implementations, ready to be executed, of well known search algorithms — six for graphs (Breadth-first search, Depth-first search, Dijkstra’s algorithm, Best-first search, A*, IDA*) and three for games (Min-Max, alpha-beta pruning, Scout). At disposal are also some extras (configuration options, selection of data structures involved) that lead to variations of the default search performance. Additionally, the library comes with a set of examples included in
the sac.examples package. Among them there are console solvers for: sliding puzzle, traveling salesman problem, sudoku; and two-person game applications (with a simple GUI) for: checkers and Nim.

The central ‘actor’ in SaC is the sac.State interface. It represents an abstract state in some graph or some game tree. The rest of the object oriented API is built up around this interface.

From the user’s perspective, in order to execute a search algorithm it is sufficient to describe the particular search problem by defining — i.e. providing implementations for — the following elements.

1. **Generation of descendants** — What new states (direct descendants) can be generated from a given state?

2. **Identification** — What identifiers (string or integer representations) can be assigned to states, so that the same state is not visited multiple times unnecessarily?

3. **Termination** — Is given state a terminal? I.e. a solution state (graphs) or a win state (game trees).

3’. **Heuristics (optional)** — An estimation how far a state is from the solution (graphs), or an evaluation whether the state represents some advantage for the maximizing or the minimizing player (game trees).

In SaC, suitable places are prepared for the above purposes. We have assigned the number 3’ to the last element — heuristics, since it can be viewed as an extension of the element no. 3 — termination.

We see three potential groups of users that might be interested in SaC. Firstly — people, not necessarily computer scientists, needing to solve some engineering or technological optimization problem. Secondly — computer games programmers, in particular programmers of games for mobile devices (the choice of Java may be a friendly factor). Thirdly — academic communities, using the mentioned algorithms in research or didactics.

**Acknowledgements**

The SaC library has been developed as a subproject within a larger academic project named TEWI (Polish abbreviation for Telekomunikacja Edukacja Wiedza Innowacje, translatable as: Telecommunication Education Knowledge Innovations), financed by the European Fund for Regional Development (POIG.02.03.00-00-028/09), see [http://www.wi.pb.edu.pl/index.php/projekty-ue/tewi](http://www.wi.pb.edu.pl/index.php/projekty-ue/tewi) and [http://tewi.p.lodz.pl](http://tewi.p.lodz.pl).

The goal of TEWI was to create a web platform gathering open source software projects, from various domains, developed by Polish academic communities. The project has been conducted by a consortium managed by the Łódź University of Technology, and involved the following partners: Białystok Technical University, Warsaw University of Technology, Gdańsk University of Technology, Łódź Univeristy of Computer Sciences and Skills, Cracow University of Technology, West Pomeranian University of Technology in Szczecin, Polish-Japanese Institute of Information Technology.
As regards the SaC project itself, it has been programmed in the Java language, using the Java SE 7 (1.7.0) version. Java is a product and a trademark of the Oracle company (http://www.oracle.com).

The authors would like to thank the creators of the following projects and libraries that contributed to SaC:

- **Graphviz** — a free software for automatic graph drawing, see [http://www.graphviz.org](http://www.graphviz.org). In SaC, we enable a possibility to generate output text files, compliant with the Graphviz language. The files contain representations of graphs or trees that were searched by a particular algorithm. Having such a file and the Graphviz software installed, the user can produce visualizations using one of Graphviz engines like dot, neato, etc. We used Graphviz in its 2.38 version. All graph / tree illustrations presented throughout this user guide have been produced using Graphviz;

- **jFreeChart** — a free library which allows to generate and display charts from the Java code, see [http://www.jfree.org/jfreechart](http://www.jfree.org/jfreechart). In SaC, we take advantage of jFreeChart in two contexts: (1) when displaying graphical monitors showing the progress of an ongoing graph search procedure, (2) when producing bar charts or xy plots based on statistics collected from some large experiment (e.g. a comparison of multiple algorithms or heuristics on multiple instances of some search problem). We used jFreeChart in its 1.0.14 version (jfreechart-1.0.14.jar);

- **jCommons** — a project offering generally reusable Java components [http://www.jcommons.sourceforge.net](http://www.jcommons.sourceforge.net). In SaC, the jCommons library is needed as a prerequisite for jFreeChart. We used jCommons in its 1.0.17 version (jcommon-1.0.14.jar).

- **SWT (Standard Widget Toolkit)** — an open source widget toolkit for Java designed to provide efficient, portable access to the user-interface facilities of the operating systems on which it is implemented, see [http://www.eclipse.org/swt](http://www.eclipse.org/swt). In SaC, the SWT is used in simple demo-AIs built for the games of checkers and Nim.
Chapter 1

Getting started

1.1 Installation and requirements

Since SaC is a Java library, placed in a .jar file, there is no explicit installation procedure whatsoever. In other words, the user is supposed to simply download the library and attach it to his Java project, where he intends to use SaC and its searching capabilities.

This user guide document was written along with the first version of SaC distribution, numbered as 1.0.0. The sac-1.0.0-dist.zip file (to be downloaded) encapsulates the whole distribution, and contains the following elements.

The library files:

- sac-1.0.0.jar — the main SaC library file,
- jfreechart-1.0.14.jar — a prerequisite library,
- jcommon-1.0.14.jar — a prerequisite library,
- swt.jar — a prerequisite library (64 bit version),

the documentation (javadoc) and source files:

- sac_doc.zip,
- sac_src.zip,

the UML class diagram — a high-level view on SaC’s main elements:

- sac.uml.pdf,

examples (templates) of configuration files:

- graph_configurator_example.properties,
- game_configurator_example.properties,
and finally, a few .bat files (for Windows) executing examples of SaC ready-made solvers / programmes:

- run_sudoku.bat,
- run_slidingpuzzle.bat,
- run_tsp.bat,
- run_rectpacking.bat,
- run_checkers.bat,
- run_nim.bat.
1.2 “Hello world” for a graph

Suppose we have at disposal the following Java class to represent directed graphs with weights (the class is not related to SaC itself).

```java
public class DirectedGraph {
    private double[][] costs;
    private int goal;

    public DirectedGraph(int howManyNodes, int goal) {
        costs = new double[howManyNodes][howManyNodes];
        for (int i = 0; i < howManyNodes; i++)
            for (int j = 0; j < howManyNodes; j++)
                costs[i][j] = Double.POSITIVE_INFINITY;
        this.goal = goal;
    }

    public void addEdge(int i, int j, double cost) {
        costs[i][j] = cost;
    }

    public double[][] getCosts() {
        return costs;
    }

    public int getGoal() {
        return goal;
    }
}
```

It uses a two-dimensional array (double[][] costs) to store costs of transitions between particular nodes — in other words to model the edges. The symbolic infinity (Double.POSITIVE_INFINITY) indicates the lack of an edge. The index of the target node to be reached is kept in the field named goal. Obviously, the presented implementation is just an example and many other possibilities may come to mind. Up to now SaC is not involved anyhow.

Now, consider a graph created by the code below and depicted in Fig. 1.1. Suppose we would like to be able to search this graph (or other graphs alike, stored as DirectedGraph) using SaC.

```java
DirectedGraph myGraph = new DirectedGraph(7, 6);
myGraph.addEdge(0, 1, 3.0);
myGraph.addEdge(0, 2, 1.0);
myGraph.addEdge(0, 5, 2.5);
myGraph.addEdge(1, 3, 2.0);
myGraph.addEdge(1, 4, 1.5);
myGraph.addEdge(2, 1, 1.0);
myGraph.addEdge(2, 4, 3.0);
myGraph.addEdge(3, 7, 1.0);
myGraph.addEdge(4, 7, 2.0);
myGraph.addEdge(5, 6, 4.0);
myGraph.addEdge(6, 7, 0.5);
```
Figure 1.1: “Hello world” graph to be travelled from node 0 (initial) to node 7 (goal).

To apply SaC and its searching capabilities we prepare the following class.

```java
public class HelloWorldGraphState extends GraphStateImpl {
    public static DirectedGraph dg = null;

    private int i; // current node

    public HelloWorldGraphState(int i) {
        this.i = i;
    }

    @Override
    public List<GraphState> generateChildren() {
        List<GraphState> children = new ArrayList<GraphState>();
        double[][] costs = dg.getCosts();
        for (int j = 0; j < costs.length; j++)
            if (costs[i][j] < Double.POSITIVE_INFINITY)
                children.add(new HelloWorldGraphState(j));
        return children;
    }

    @Override
    public int hashCode() {
        return i;
    }

    @Override
    public boolean isSolution() {
        return (i == dg.getGoal());
    }

    @Override
    public String toString() {
        return Integer.toString(i);
    }
}
```
A reader’s implementation could differ in details but in general the following rules have to be followed. Firstly, the class should extend the sac.game.GraphStateImpl class which is a suitable abstraction for states taking part in graph searches. Secondly, the three vital elements should be described: (1) generation of descendants — by overriding the generateChildren() method, (2) identification of states — in our example this has been done by an override of the hashCode() method (it comes from the java.lang.Object class), (3) termination — by overriding the isSolution() method.

Additionally, a function object called StateFunction is attached to our class by means of the setGFunction(...) static method. In fact, this fragment is not a must in general, but in our example it is needed to take the costs of edges into account. Without this fragment all transitions would be by default treated as being of cost 1 (one manipulation / move). The attached object is equipped with a calculate(...) method. It is meant to return the cost paid by travelling from the initial state up to the current state. The name setGFunction(...) is driven by a naming convention known from graph search algorithms, in particular the A∗ algorithm. Commonly, the travelled cost from the initial state up to a state \( s \) is denoted by \( g(s) \), and the estimated cost from \( s \) to the goal state by \( h(s) \). The understanding of these details is not important right now and will be discussed later in section [4.1](#).

Now, by executing the following main(...) method

```java
HelloWorldGraphState.dg = myGraph; // static reference to graph from previous listing
GraphSearchAlgorithm algorithm = new Dijkstra(new HelloWorldGraphState(0));
algorithm.execute();
HelloWorldGraphState solution = (HelloWorldGraphState) algorithm.getSolutions().get(0);
System.out.println("SOLUTION: " + solution);
System.out.println("PATH: " + solution.getPath());
System.out.println("PATH COST: " + solution.getG());
```

one can see the search results related to the Dijkstra’s algorithm:

SOLUTION: 7
PATH: [0, 2, 1, 3, 7]
PATH COST: 5.0

The reader can check that the path shown is in fact the path of lowest cost (the shortest path).
The search graph generated by SaC during our exemplary execution is presented in Fig. 1.2 (the visualization was produced by Graphviz from an input file generated from SaC’s API). Throughout this user guide, the coloring of our illustrations for graph searches is as follows: yellow denotes depth = 0.0
g = 0.0
h = 0.0
f = 0.0
0

depth = 1.0
g = 1.0
h = 0.0
f = 1.0
2

depth = 1.0
g = 2.5
h = 0.0
f = 2.5
4

depth = 2.0
g = 2.0
h = 0.0
f = 2.0
1

depth = 2.0
g = 6.5
h = 0.0
f = 6.5
6

depth = 3.0
g = 4.0
h = 0.0
f = 4.0
3

depth = 3.0
g = 3.5
h = 0.0
f = 3.5
4

depth = 4.0
g = 5.0
h = 0.0
f = 5.0
7

Figure 1.2: Search graph produced by SaC using Dijkstra’s algorithm for the “hello world” exemplary graph from Fig. 1.1.
the initial state, red denotes states that were generated but not visited, green denotes visited states
lying on the path to the solution, light gray denotes visited states not on the path, blue indicates
the goal (solution) state. In every box, displayed is an information about: the depth of a state, its
traveled cost \( g \), estimation of its remaining cost \( h \) (0 by default when no heuristics is defined), and
its representation in the inner white rectangle.

The reader might have noticed the lack of the \( 0 \rightarrow 1 \) link in the search graph from Fig. 1.2
although such a transition is possible according to our directed graph under search (Fig. 1.1). The
explanation is as follows — as the search procedure was progressing it did at first generate the
state 1 as one of descendants of state 0, but later it discovered that it is cheaper to reach the state
1 going via 2 (path \( 0 \rightarrow 2 \rightarrow 1 \)), therefore finally, the state 2 was memorized as the ‘best parent’
for state 1. Probably, this is a good place to point out the distinction between a graph subject to
searching and a search graph. The first pertains to the structure, possibly with cycles and possibly
infinite, describing some real-world object or phenomenon that we analyze by searching. The
latter describes the way the actual search procedure was progressing. Our illustrations of search
graphs in this user guide can be regarded as images or snapshots of the search procedure at its
stoppage moment.

How would a breadth-first search procedure do on our example (instead of Dijkstra’s algo-
rithm)? The necessary modification is in the line instantiating the algorithm (line 2):

```java
GraphSearchAlgorithm algorithm = new BreadthFirstSearch(new HelloWorldGraphState(0));
```

with the rest of the code unchanged. It results in the following output:

```
SOLUTION: 7
PATH: [0, 1, 3, 7]
PATH COST: 6.0
```

and the search graph illustrated in Fig. 1.3. One can recognize that the breadth-first search
procedure neglects the transition costs and is guided only by depths of states (i.e. the number of
hops from the initial state). The resulting path is therefore of greater cost than before.

As the reader might have noted our exemplary HelloWorldGraphState class kept a static
reference to the directed graph object. The reference was static because it was common to all
HelloWorldGraphState objects and could be shared. This reference was needed for the purpose of
generation of descendants — we accessed it in the generateChildren() method to know what are
the edges outgoing from the current node (encapsulated by the search state). We should explain
that such a feature — memorization of the whole graph under search — will not be always present
when using SaC. Very often the programmer will be able to construct the descendants solely on
the basis of the information present in the current state (e.g. possible manipulations for the current
arrangement of a sliding puzzle or Rubik’s cube, etc.). The reader should understand that graphs
under search can be very large, often too large to be memorized or even infinite.

\[^1\]Since the procedure suitably avoids the cycles a search graph is also often referred to as a search tree.
Let us now play with a larger scale example, depicted in Fig. 1.4. We generate a graph with 100 nodes placed on a plane within a square of side 100. The initial node is placed in the top-left corner, the goal node in the bottom-right corner. The rest of nodes is placed randomly according to the uniform distribution within the square. Edges between nodes are also generated on random but satisfy the following constraints. The number of edges is approximately 10% of the total number of edges in a completely connected graph. Two nodes can be connected only if the distance between them along a straight line is not greater than 20% of the square diagonal. In other words we allow only fairly close nodes to be connected — the graph can be regarded as a simulation of a map. The cost of an edge is equal to the aforementioned distance plus some random \( \epsilon > 0 \) (uniformly...
distributed) proportional to that distance but not greater than 10% of it. Fig. 1.4 shows the graph together with the shortest path. The reader may zoom in the figure to see more details.

Figure 1.4: Graph with 100 nodes generated on random: initial node 0 (yellow), goal node 99 (blue). Shortest path marked as bolder black.
To store our graph we have introduced a new class `DirectedGraphWith2DCoordinates`, shown below, as an extension of `DirectedGraph`. Again, we remark this class is not related to SaC itself.

```java
public class DirectedGraphWith2DCoordinates extends DirectedGraph {
    private double[][] coordinates;

    public DirectedGraphWith2DCoordinates(int howManyNodes, int goal) {
        super(howManyNodes, goal);
        coordinates = new double[howManyNodes][2];
    }

    public void addCoordinates(int i, double x, double y) {
        coordinates[i][0] = x;
        coordinates[i][1] = y;
    }

    public double[][] getCoordinates() {
        return coordinates;
    }
}
```

The code to generate the actual instance of our graph (from Fig. 1.4) is listed below. In the code we use a `java.util.Random` object as the random generator and we force the seed of randomization (1234). This allows to generate exactly the same graph on every run of the program.

```java
int n = 100;
DirectedGraphWith2DCoordinates myGraph = new DirectedGraphWith2DCoordinates(n, n - 1);
double side = 100.0;
myGraph.addCoordinates(0, 0.0, side); // initial
myGraph.addCoordinates(n - 1, side, 0.0); // goal
Random random = new Random(1234); // java.util.Random, imposed randomization seed: 1234
for (int i = 1; i < n - 1; i++)
    myGraph.addCoordinates(i, side * random.nextDouble(), side * random.nextDouble());
int e = (int) Math.round(0.1 * n * (n - 1)); // 10 % of a complete graph
double maxDistance = 0.2 * side * Math.sqrt(2.0); // max distance between two nodes to be connected
for (int k = 0; k < e; k++) {
    int i, j;
    double distance;
    do {
        i = random.nextInt(n);
        j = random.nextInt(n);
        distance = Math.sqrt(
            Math.pow(myGraph.coordinates[i][0] - myGraph.coordinates[j][0], 2)
            + Math.pow(myGraph.coordinates[i][1] - myGraph.coordinates[j][1], 2)
        );
    } while ((i == j) || (myGraph.getCosts()[i][j] < Double.POSITIVE_INFINITY) || (distance > maxDistance));
    double epsilon = random.nextDouble() * 0.1 * distance;
    myGraph.addEdge(i, j, distance + epsilon);
}
```
To search our graph with Dijkstra's algorithm we prepare the following program and we display a bit more information to the console than before. Note that we again use the same state class — `HelloWorldGraphState`.

```java
HelloWorldGraphState dg = myGraph; // our random graph (map) with 100 nodes
GraphSearchAlgorithm algorithm = new Dijkstra(new HelloWorldGraphState(0));
algorithm.execute();
HelloWorldGraphState solution = (HelloWorldGraphState) algorithm.getSolutions().get(0);

System.out.println("SOLUTION: "+ solution);
System.out.println("PATH: "+ solution.getPath());
System.out.println("PATH COST: "+ solution.getG());
System.out.println("DURATION [ms]: "+ algorithm.getDurationTime());
System.out.println("CLOSED: "+ algorithm.getClosedStatesCount());
System.out.println("OPEN: "+ algorithm.getOpenSet().size());
```

We execute the program and obtain the following output in the console:

```
SOLUTION: 99
PATH: [0, 18, 14, 64, 60, 10, 5, 99]
PATH COST: 149.51694106202785
DURATION [ms]: 11
CLOSED: 100
OPEN: 0
```

Without going too much into details, let us explain that `algorithm.getClosedStatesCount()` returns the number of states visited during the search, whereas `algorithm.getOpenSet().size()` returns the number of states that were generated but not visited because the algorithm was stopped when reaching a solution.

Perhaps some readers may know that Dijkstra’s algorithm belongs to so called uninformed search methods. The algorithm takes into account only the travelled (experienced) cost $g(s)$ while traversing a graph and does not use any apriori information or heuristics to guide the ongoing search more economically to the goal. Another famous algorithm called $A^*$ performs an informed search. It uses the sum $g(s) + h(s)$ to decide about the order in which states are visited. We remind that the heuristic summand $h(s)$ is an estimate on the cost remaining to the goal (in fact a lower bound on exact remaining cost). Let us try out $A^*$ on our last example.

The necessary modification (line 2):

```java
GraphSearchAlgorithm algorithm = new AStar(new HelloWorldGraphState(0));
```

leads to the following output

```
SOLUTION: 99
PATH: [0, 18, 14, 64, 60, 10, 5, 99]
PATH COST: 149.51694106202785
DURATION [ms]: 10
CLOSED: 100
OPEN: 0
```
We note that beside a tiny change in duration time nothing else changed. The number of visited states is still the same — 100. Our algorithm worked exactly as Dijkstra’s algorithm. What is wrong? The heuristic function (the additional information) has not been defined. We need to go back to our HelloWorldGraphState class and add the following fragment to the static block:

```java
public class HelloWorldGraphState extends GraphStateImpl {
    static {
        setHFunction(new StateFunction() {
            @Override
            public double calculate(State state) {
                HelloWorldGraphState hwgs = (HelloWorldGraphState) state;
                DirectedGraphWith2DCoordinates dg2D = (DirectedGraphWith2DCoordinates) HelloWorldGraphState.dg;
                return Math.sqrt(Math.pow(dg2D.getCoordinates()[hwgs.i][0] - dg2D.getGoal()[0], 2) + Math.pow(dg2D.getCoordinates()[hwgs.i][1] - dg2D.getGoal()[1], 2));
            }
        });
    }
}
```

By doing so we define the heuristics which estimates the cost remaining to the goal as the distance along a straight line (Euclidean distance). We are able to calculate such distance because we know the coordinates of the goal. Clearly, some obstacles might occur along the way, but still this guiding information is better than none. Below we show the new output for $A^*$ (after the heuristic function has been attached to the class):

```
SOLUTION: 99
PATH: [0, 18, 14, 64, 60, 10, 5, 99]
PATH COST: 149.51694106202785
DURATION [ms]: 3
CLOSED: 18
OPEN: 38
```

As one can note the algorithm finds the same optimal path but the number of visited (closed) states is now significantly smaller. Search graphs produced by both algorithms are shown in Fig. 1.5.
Figure 1.5: Search graphs produced by Dijkstra’s (a) and $A^*$ (b) algorithms for the random graph with 100 nodes from Fig. 1.4
1.3 "Hello world" for a game

Imagine the following simple game. There are \( n \) stones on a table. Players make their moves interchangeably by taking away either one or two stones. A player is not allowed to take two stones when exactly two are left. The player left with the last stone loses.

![Figure 1.6: Exemplary game — “take one or two stones away”. Player left with the last stone loses.](image)

Some readers probably know that this kind of game can be easily solved via dynamic programming. When \( n = 1 \) the value of the game for the current player is a loss (\( L \)). Therefore, for \( n = 2 \) and \( n = 3 \) the game value is a win (\( W \)), since the player to move can transfer both these cases to \( n = 1 \) by taking one or two stones respectively. For \( n = 4 \) the game value is a loss again, and the pattern \( LWWLWWLWW \ldots \) emerges. Therefore, it is easy to tell the game value and the correct move for any \( n \). If \( n \) can be expressed as \( n = 3m \) or \( n = 3m - 1 \) (for some natural \( m \)), then it is a win for the player to move and he must simply transfer the game into \( n = 3m - 2 \).

Suppose however, that we do not know the dynamic programming approach, and we would like to find the solution of our game for some \( n \) using SaC. Here is how.
public class HelloWorldGameState extends GameStateImpl {

    private int n;

    public HelloWorldGameState(int n) {
        this.n = n;
    }

    @Override
    public List<GameState> generateChildren() {
        List<GameState> children = new LinkedList<GameState>();
        for (int i = 1; i < 2 && i < n; i++) {
            HelloWorldGameState child = new HelloWorldGameState(n - i);
            child.setMoveName(Integer.toString(i));
            child.setMaximizingTurnNow(!isMaximizingTurnNow());
            children.add(child);
        }
        return children;
    }

    @Override
    public int hashCode() {
        int[] pair = {n, (isMaximizingTurnNow() ? 1 : -1)};
        return Arrays.hashCode(pair);
    }

    static {
        setHFunction(new StateFunction() {
            @Override
            public double calculate(State state) {
                HelloWorldGameState hwgs = (HelloWorldGameState) state;
                if (hwgs.n == 1)
                    return (hwgs.isMaximizingTurnNow()) ? Double.NEGATIVE_INFINITY : Double.POSITIVE_INFINITY;
                return 0.0;
            }
        });
    }
}

Firstly, our class should extend the sac.game.GameStateImpl class which is a suitable abstraction for states taking part in game tree searches. Secondly, the three vital elements should be described: (1) generation of descendants — by overriding the generateChildren() method, (2) identification of states — in our example this has been done by an override of the hashCode() method, which comes from the java.lang.Object class, and we produce the hashcode from a pair of information (number of stones n, whose turn it is) uniquely describing our states, (3) termination (or heuristics) — this has been done by attaching via the setHFunction(...) static method a suitable function object to our class. This function is equipped with a calculate(...) method, and in our example it returns either ±∞ when the last stone is reached or 0 (no evaluation) otherwise.
Now, by executing the following `main(...)` method:

```java
GameSearchAlgorithm algorithm = new MinMax(new HelloWorldGameState(6));
algorithm.execute();
System.out.println("MOVES_SCORES: " + algorithm.getMovesScores());
```

one obtains the answer for \( n = 6 \), being:

MOVES SCORES: {2=1.498077612385263E308, 1=-1.3482698511467367E308}

The large positive value \( \approx 1.498 \cdot 10^{308} \) is the SaC’s way to indicate a win for the maximizing (initial) player if he starts by taking two stones. The large negative value \( \approx -1.348 \cdot 10^{308} \) indicates a loss for the player if he starts by taking one stone.

We should shortly explain that in game searches, in order to represent wins or losses SaC returns numbers of order \( 10^{308} \) (close to the `Double.MAX_VALUE` value) rather than symbolic infinities — `Double.NEGATIVE_INFINITY` or `Double.POSITIVE_INFINITY` (as the code from the example might suggest). The returned numbers show how quickly a win or a loss is attained after a particular move. A larger absolute value indicates a win or a loss after fewer moves. This can be regarded as ‘grading of infinities’, which SaC calculates on the fly, and is explained in more detail in chapter 4 — “Searching game trees” (in the API section).

The search tree generated by SaC during the example from above is presented in Fig. [1.7] (the visualization was produced by Graphviz from an input file generated from SaC’s API). Throughout this user guide, the coloring of our illustrations for game searches is as follows: yellow denotes the initial state, light gray denotes visited regular states, dark gray indicates non-win terminal states, blue indicates win terminal states (a win for either player), dark red indicates states for which the game value or a bound on that value was read as a ready result (because these states had occurred before) from the so called transposition table\(^3\), light red indicates the so called cutoff states which could not affect the game value, green indicates states residing along the principal variation\(^4\). In every box, displayed is an information about: depth of a state, its heuristic value \( h \), its game value \( v \) (also known as the minimax value) assigned by the search procedure, and its representation.

The reader might be tempted now to check if the search would also work correctly for some larger \( n \), e.g. \( n = 100 \). The answer unfortunately is no. At least not without some intervention by the user. The reason is that for games, SaC, like all search engines, has some depth limit (search horizon), by default set to 3.5 i.e. seven half-moves\(^5\). Therefore, win states located beyond the search horizon are not seen by the algorithm. Be reminded that in our exemplary implementation we return zero for non-win states (line no. 34 of the `HelloWorldGameState` class). Actually, with depth limit set to seven half-moves, already for \( n = 9 \) there will exist one win state beyond the search horizon. To demonstrate more of such states, in Fig. [1.8] we depict the game tree produced by SaC for \( n = 12 \). In the bottom left corner one can see several states, marked with dark gray, that are

\(^2\)To be precise, SaC actually may return `Double.NEGATIVE_INFINITY` or `Double.POSITIVE_INFINITY`. This happens only in the case when a win or a loss is predicted to come immediately — that is in one half-move.

\(^3\)The notion and mechanisms of the transposition table will explained later in section 4.2.5.

\(^4\)Principal variation is a line of play during which both players make optimal moves with respect to the current search horizon.

\(^5\)In the game related nomenclature, a single move made by one player is called a half-move or a ply, and is counted as 1/2. A full move is considered to be done after both players have made their half-moves.
CHAPTER 1. GETTING STARTED

Figure 1.7: "Hello world" example for $n = 6$ — game tree searched by SaC.

terms due to depth limit reached, and were assigned a zero by our implementation as their heuristic game evaluation. We mention that the program response in that case would be

MOVES SCORES: {2=1.1556598724114885E308, 1=0.0}
and the score 0.0 is a consequence of a too shallow search subtree.

Figure 1.8: “Hello world” example for \( n = 12 \) — game tree searched by SaC. Terminal non-win states (dark gray) visible in the bottom left corner due to a default depth limit set to 3.5.

To solve the case of \( n = 100 \), a more ‘aware’ user can encourage SaC to search deeper. A sample code to do so is as follows.

```java
public static void main(String[] args) {
    GameSearchConfigurator configurator = new GameSearchConfigurator();
    configurator.setDepthLimit(49.5);
    GameSearchAlgorithm algorithm = new MinMax(new HelloWorldGameState(6), configurator);
    algorithm.execute();
    System.out.println("MOVES.SCORES:..." + algorithm.getMovesScores());
    System.out.println("VISITED:..." + algorithm.getClosedStatesCount());
    System.out.println("DURATION...[ms]:..." + algorithm.getDurationTime());
}
```
And, the information displayed to the console is

```
VISITED: 2550
DURATION TIME [ms]: 70
```

demonstrating the correct response that the initial player loses regardless of his moves after the optimal play by his opponent. Fig. 1.9 illustrates the tree associated to that search in a simpler format (we encourage the reader to zoom in the figure multiple times to see the coloring of states better).

Figure 1.9: “Hello world” example for $n = 100$ — game tree searched by SaC with the depth limit explicitly set to 49.5 (99 half-moves).
Chapter 2

State abstraction

In this section we present some of the API related to the state as a top-level abstraction for both graph states and game states.

In SaC, introduced is an interface sac.State and its default implementation sac.StateImpl. To give the reader an overview we start by showing the source code\(^1\) of the interface, then we make some comments on particular elements. Since most of methods in the sac.StateImpl implementation are straightforward, hereby we show only the more important excerpts of it. For full code the reader is addressed to Appendix 6.1.

```java
package sac;
import java.util.List;
import sac.graphviz.Graphvizable;

public interface State extends Comparable<State>, Graphvizable {
    public Identifier getIdentifier();
    public void refreshIdentifier();
    public State getParent();
    public void setParent(State parent);
    public List<? extends State> getChildren();
    public double getDepth();
    public void setDepth(double depth);
    public List<String> getMovesAlongPath();
    public List<? extends State> generateChildren();
    public double getH();
    public void setH(Double h);
    public void refreshH();
    public String getMoveName();
    public void setMoveName(String moveName);
    public void refresh();
}
```

\(^1\)The listing is in its full, but without javadocs. The javadocs can be found within the library both in the actual source files and in the generated HTML format.
2.1 Identifiers

The first element of interest is the identifier (sac.Identifier). Identifiers are meant to be representations of states within search procedures. If possible, they should be unique representations, but more importantly they should be fast representations (i.e. quickly computable). In SaC, two scenarios are allowed: (1) identifiers are integers calculated via the int hashCode() method — the default scenario, (2) identifiers are strings calculated via the toString() method.

As the reader may know, both hashCode() and toString() are standard Java methods included in the most elementary class java.lang.Object and are meant to be overridden by the user if needed. In other words, in SaC we let the user decide which scenario he prefers and we expect him to implement at least one of the above two methods. The sac.Identifier type should be viewed as a wrapper for this mechanism, regardless of the chosen scenario.

We remark, that typically in practice we decide for string identifiers when we absolutely need to guarantee the uniqueness of identifiers in given problem. Yet, one should be aware that comparisons of long strings may be computationally expensive. On the other hand, we decide for integer identifiers (hashcodes) when we care more about the speed of execution, and simultaneously we agree to accept a tiny risk of a situation when conflicting hashcodes might occur, i.e. the same hashcodes for two different states.

The source code of the sac.Identifier is as follows.

```java
package sac;

public final class Identifier implements Comparable<Identifier> {

    private static IdentifierType type = IdentifierType.HASH_CODE;
    private Object id = null;

    public Identifier(State state) {
        id = (type == IdentifierType.HASH_CODE) ? Integer.valueOf(state.hashCode()) : state.toString();
    }

    public static final IdentifierType getType() {
        return type;
    }

    public static final void setType(IdentifierType type) {
        Identifier.type = type;
    }

    public int compareTo(Identifier otherIdentifier) {
        if (type == IdentifierType.HASH_CODE)
            return ((Integer) id).compareTo((Integer) otherIdentifier.id);
        else
            return ((String) id).compareTo((String) otherIdentifier.id);
    }

    public boolean equals(Object otherIdentifier) {
        Identifier otherIdentifier2 = (Identifier) otherIdentifier;
        if (type == IdentifierType.HASH_CODE)
            return ((Integer) id).equals((Integer) otherIdentifier2.id);
        else
```
As one may note the `sac.Identifier` class is basically a wrapper around its inner field: private `Object id`. It is interpreted either as an integer or a string depending on a global static setting (private static `IdentifierType type`). On the usage level, this setting can be changed either directly from the code via the static `setType(...)` method, or by means of so called `configurators`, which are discussed in further sections.

### 2.2 Parent – children bindings

The `sac.State` interface contains a getter `getParent()` and a setter `setParent(...)` to respectively retrieve or impose a reference to the parent state for a given state. These two methods are meant mainly for the purposes of the core of SaC, and in practice are called seldom by an end user. In fact, we discourage the user to call the setter on his own because a careless usage might potentially result in an incorrect behaviour of a search procedure. As a convention, we use the `null` result of a getter when a state is the root state.

As regards descendant states (children) there is only a getter present — `getChildren()`. It is meant to return a list of references to children. We should remark however that by default SaC does not memorize these references for memory saving reasons. Hence, by default an empty list is returned. We explain that a link from a child to its parent (accessible via `getParent()`) is always memorized, but the opposite direction links from a parent to its children are redundant. In practice, children objects during a search procedure are created by the SaC itself via the `generateChildren()` callback method executed on a parent. The children are suitably visited later on. Again, this default SaC behaviour (related to not-memorizing references from parents to children) can be explicitly changed by a `configurator` object, but in practice such a change is needed only in the case when the user is interested in drawing graphs by means of Graphviz, and is not needed for the search itself.

A vital element is the mentioned `generateChildren()` callback method. This method must inevitably be implemented by the user and should return a list of direct descendants (children) for given state. In such an implementation the user is expected to:

- create a local list,

---

2In graph searches, children states (non-visited so far) are placed in a queue at this stage and polled from it later. In game searches, a recursive call on a child state is made by the core of SaC. In either case, to correctly carry out the search procedure there is no need to memorize references from a parent to all its children.
- build the children according to the rules / nature of the specific problem,
- give names to the moves (or manipulations) which lead to the creation of particular children (via `setMoveName(...)`,
- add the children to the local list,
- return the list.

Such proceedings have been already demonstrated in the “hello world” examples.

On the other hand, the user is not expected to attach anyhow his local list to the parent object (and in fact should not do it, having in mind the remarks from the previous paragraph). The core of SaC awaits just for the result of the `generateChildren()` callback method and proceeds with the result appropriately further on.

### 2.3 Depth of state, path from root to given state

The `sac.State` interface allows the user to ask for the depth of a state by the `getDepth()` getter. Depth is understood as the number of moves performed to transfer the root state into the given state. In particular, the depth of the root state is zero. Although the API makes also the setter available — `setDepth(...)` — the user is discouraged to use it explicitly.

Additionally, the user is allowed to ask for the *path* (sequence of states or moves) from the root state up to the given state. Depending on the need, this can be done either by the `getPath()` method (returns the sequence of states) or by the `getMovesAlongPath()` method (returns the sequence of move names). We should remark that the path as a sequence of states includes the root state in the front and therefore is longer by one element comparing to the sequence of move names.

### 2.4 Heuristics

The interpretation of a heuristic value assigned to a state is different for graphs and different for game trees. When searching graphs, the heuristics is a non-negative estimation of the distance remaining to the solution. When searching game trees, the heuristics is an evaluation of the game position represented by a given state. In that case positive values indicate some advantage for the maximizing player and negative values indicate some advantage for the minimizing player. By default, the player who starts the game is regarded as the maximizing player in SaC.

---

3. For analogical reasons as in the case of `setParent(...)`.  
4. In fact it should be either an exact value or a lower bound on that distance, otherwise the heuristic is *inadmissible* and might lead to non-optimal paths to the solution.  
5. Such an evaluation is often imprecise and reflects human experience and knowledge about the game. For example in chess, the heuristics may involve materialistic elements (number and quality of pieces remaining on the board), but also positional elements like: advantage in the board center, structure of pawns, activeness of pieces, pass pawns, king safety, etc.
The API provides both the getter `getH()` and the setter `setH(...)` for the heuristics. Again (as in former contexts), we discourage the user to use the setter explicitly, it is meant only for the purposes of the core of SaC.

To define how the heuristics should be calculated for given search problem, the user is supposed to provide an implementation of the `sac.StateFunction` class. Its default form in SaC is presented below and it returns zero via the `calculate(...)` method regardless of the state passed as an argument. The user should therefore appropriately populate the body of this method.

```java
package sac;

public class StateFunction {
    public StateFunction() {
    }

    public double calculate(State state) {
        return 0.0;
    }
}
```

In particular, in the context of game trees we encourage the user to return `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY` constants as a representation of a win respectively for the maximizing and the minimizing player.

When the user-defined function is ready, it should be instantiated as an object — the calculator of heuristics — and attached to the state class by means of the static method `setHFunction(StateFunction hFunction)` (it comes from the `sac.StateImpl` class). Note that `setHFunction(...)` is static, therefore the imposed function object will also be static — common to all the states.

There are two programmistic ways the user may create and attach his function object. The first way is to create a separate named class (thus a separate source file) being an extension of the `sac.StateFunction`, e.g. in the following manner

```java
public class MyHeuristics extends StateFunction {
    public double calculate(State state) {
        // suitable calculations
    }
}
```

and then to attach an instance of this class by placing the following static block in the state class.

```java
public class MyState extends GraphStateImpl { // or extends GameStateImpl
    static {
        setHFunction(new MyHeuristics());
    }
}
```

---

6 A side remark: in the context of graph searching, the `Double.POSITIVE_INFINITY` constant might theoretically be also of some use, e.g. for generated states that do not satisfy certain constraints for given problem. Such states with infinite heuristics would be then placed at the very end of the queue. However, one may argue about the sense of keeping them in the queue at all. Usually, it is better to simply discard (ignore) such states i.e. not to return them among results of the `generateChildren()` method.
The second way is to do it a bit simpler, namely by the Java mechanism of an anonymous class (so without a separate source file), as shown below.

```java
public class MyState extends GraphStateImpl { // or extends GameStateImpl

    ... static {
        setHFunction(new StateFunction() {
            @Override
            public double calculate(State state) {
                // suitable calculations
            }
        });
    }
}
```

### 2.5 Refresh methods

In the listing of the `sac.State` interface from the beginning of this chapter one can notice three refresh methods: `refreshIdentifier()`, `refreshH()`, `refresh()`. When and why are they needed?

Firstly, we need to explain that both the identifier and the value of heuristics are designed in SaC in such a manner so that they are computed possibly only once. By that, SaC tries avoid multiple and often expensive computations of the same results. The typical scenario is the following. The computation of either the identifier or the heuristics is postponed until the first call to the suitable getter is made, i.e. `getIdentifier()` or `getH()` respectively. More precisely, after a state object is constructed (instantiated) its internal variables meant to keep the identifier and the heuristics remain set to `null` for some time. When the first call to a particular getter is made, the suitable value is computed according to the recipe provided by the programmer and is stored internally (in the place of the `null`). After that, for each successive call those memorized values are served immediately, not recomputed.

As already said, the scenario described above is typical and covers the most common use cases in SaC. However, there exist certain situations when explicit refreshes (forcing of recomputation) of identifier or heuristic value are necessary. Such situations usually occur e.g. when the same (working) reference in RAM memory pointing to some state is used multiple times to initiate the search procedure, but the content of this reference gets changed from one search to another. A good example of that can be a chess playing program. Within its main loop, meant to handle the ongoing game and to display it to the screen, the program may use the same working reference to a state object representing the current position on the board. After a move by

---

7 As an example one may think of the Traveling Salesman Problem, described in the section 3.3 in which the heuristics can be based on the concept of Minimum Spanning Tree (MST). To generate a MST one needs to perform an additional algorithm (e.g. Kruskal’s or Prim’s algorithm). Therefore, it would be a clear waste to do it multiple times unnecessarily within a search procedure.
the human player is made, the program applies the move to that state by some suitable method, e.g. `currentState.makeMove("e2:e4")`, and we would like to use the modified state as the starting point for a SaC search procedure. Since the state may now contain old values of the identifier and the heuristics (from before the move) the programmer ought to refresh such a state. Otherwise, a search might work incorrectly. The aforementioned methods are therefore meant to explicitly force the recomputation of: the identifier (refreshIdentifier()), the heuristics (refreshH()), or both (refresh()).
Chapter 3

Searching graphs

This chapter is devoted to the part of SaC library responsible for searching graphs. We begin the chapter with a section reminding several well known graph search algorithms. The reader familiar with these algorithms can move over to subsequent sections. The second section discusses SaC’s API related to graph searching. The last one presents four graph-related examples in which we have applied SaC for demonstration, namely: sliding puzzle, Traveling Salaseman Problem, and sudoku.

3.1 Algorithms

Most of graph search algorithms can be conveniently formulated with the use of two sets (collections) of states — named by convention as the Open set and the Closed set. At a given stage of a running algorithm, the Closed set contains states that have already been visited (and turned out not to be a solution), whereas the Open set contains states waiting to be potentially visited. Awaiting states have been generated as descendants (graph neighbours) from the previously visited ones.

Depending on the type and purpose of the search algorithm, Open and Closed sets are implemented via different data structures, affecting their behaviour and performance. In particular, one can note that decisive about the algorithm type is the order according to which generated states are taken (and removed) from the Open set for further processing.

3.1.1 Breadth-first search, Depth-first search

It is difficult to point out original authors of breadth or depth-first searching techniques. In fact, they are commonly treated as simple non-informative techniques to traverse a graph, rather than actual search methods which should be informative (guided by some useful information). Historically, it is likely that the very first version of a Depth-first search (DFS) was investigated in the 19th century by a French mathematician Charles Pierre Trémaux as a strategy for solving mazes.

As the naming suggests, in the breadth-first approach the algorithm must first vist all states with depth $d$ before it can proceed with states at depth $d + 1$. In a sense contrarily, in the depth-first
approach the algorithm must not visit any unvisited state at depth $d$ if there exist some generated and unvisited states with depth $d + 1$. We present both these algorithms in the subsequent pseudocodes. In fact, the majority of the code remains the same, the only difference is in the order in which successive states are removed (poll'd) from the Open queue (line no. 6).

**Alg. 1 Breadth-first search**

1. `procedure BreadthFirstSearch(s_0)`  
2. $\text{Closed} := \emptyset$  
3. set reference from $s_0$ to its parent to null  
4. $\text{Open} := \{s_0\}$  
5. `while Open $\neq \emptyset$ do`  
6. remove from Open the state $s$ with the smallest depth  
7. `if s is the goal state then return s`  
8. generate descendants $\{t\}$ of $s$  
9. `for all t do`  
10. `if t $\notin$ Closed and t $\notin$ Open then add t to Open`  
11. add $s$ to Closed  
12. `return null`  

**Alg. 2 Depth-first search**

1. `procedure DepthFirstSearch(s_0)`  
2. $\text{Closed} := \emptyset$  
3. set reference from $s_0$ to its parent to null  
4. $\text{Open} := \{s_0\}$  
5. `while Open $\neq \emptyset$ do`  
6. remove from Open the state $s$ with the largest depth  
7. `if s is the goal state then return s`  
8. generate descendants $\{t\}$ of $s$  
9. `for all t do`  
10. `if t $\notin$ Closed and t $\notin$ Open then add t to Open`  
11. add $s$ to Closed  
12. `return null`  

In the pseudocodes we implicitly assume that all states are aware of their depths (on the programmistic level, each state object is equipped with an integer depth field). Once a descendant $t$ is generated from $s$, the depth of $t$ becomes equal to the depth of $s$ plus one.

Due to the wanted order, the Open set can in practice be implemented as a simple FIFO (First In First Out) collection (an ordinary queue) for the breadth-first case, and as a LIFO (Last In First Out) collection (a stack) for the depth-first case. Yet, for large graphs both FIFO and LIFO data structures may become inefficient. We discuss more efficient data structures, better suited for more advanced algorithms and larger graphs to be searched, in the successive sections.
3.1.2 Dijkstra’s algorithm

The famous shortest path finding algorithm, known popularly as Dijkstra’s algorithm, was conceived by Edsgar Dijkstra in 1956 and published in the work (Dijkstra, 1959). Originally, the algorithm has been formulated so that it allows to find all shortest paths between an initial graph node (state) and all the remaining nodes (a.k.a. single source all shortest paths). This explains why the algorithm does not use any heuristics that would reduce the search time by estimating the remaining distance to the goal, since there exist no single goal.

On the other hand, the algorithm can easily be modified so that it stops sooner (before shortest paths to all nodes are established) — at the moment it reaches a particular node (state) distinguished as the goal. We use this idea in our presentation of the algorithm. In fact, the reader shall see later that Dijkstra’s algorithm, formulated as the shortest path finder for a single goal, can be viewed as a special case of the $A^\ast$ algorithm, described later\textsuperscript{1}.

Let $g(s)$ denote the exact cost (or distance) for traveling from the initial state to the state $s$. Let $\Delta(s \rightarrow t)$ denote the cost of transition from $s$ to $t$, where $t$ is a direct descendant (neighbour) of $s$. Then, the Dijkstra’s algorithm can be formulated as follows.

\begin{algorithm}
1: procedure Dijkstra($s_0$) \hfill $\triangleright$ initial state $s_0$
2: \hspace{1em} $\text{Closed} := \emptyset$ \hfill $\triangleright$ empty set of visited states
3: \hspace{1em} $g(s_0) := 0$ \hfill $\triangleright$ distance covered from start
4: \hspace{1em} set reference from $s_0$ to its parent to null
5: \hspace{1em} $\text{Open} := \{s_0\}$ \hfill $\triangleright$ queue of states to be visited
6: \hspace{1em} while $\text{Open} \neq \emptyset$ do
7: \hspace{2em} remove from $\text{Open}$ the state $s$ with the smallest $g(s)$ \hfill $\triangleright$ ‘poll’ operation
8: \hspace{2em} if $s$ is the goal state then return $s$ \hfill $\triangleright$ solution found
9: \hspace{2em} generate descendants $\{t\}$ of $s$
10: \hspace{2em} for all $t$ do
11: \hspace{3em} if $t \in \text{Closed}$ then continue \hfill $\triangleright t$ already visited
12: \hspace{3em} $g(t) := g(s) + \Delta(s \rightarrow t)$
13: \hspace{3em} set reference from $t$ to its parent to $s$
14: \hspace{3em} if $t \notin \text{Open}$ then
15: \hspace{4em} add $t$ to $\text{Open}$
16: \hspace{3em} else
17: \hspace{4em} if new $g(t)$ is smaller than value known so far then
18: \hspace{5em} replace $t$ in $\text{Open}$ with the new one
19: \hspace{5em} update position of $t$ in $\text{Open}$
20: \hspace{3em} add $s$ to $\text{Closed}$
21: return null \hfill $\triangleright$ no solution found
\end{algorithm}

\textsuperscript{1}If the heuristic summand of the ordering function used in the $A^\ast$ algorithm is forced to be always zero, the $A^\ast$ algorithm becomes the Dijkstra’s algorithm.
It is worth to remark that by means of update steps — \( g(t) := g(s) + \Delta(s \rightarrow t) \) — the \( g \) cost can be regarded as a cost observed along the way, which makes it exact.

In practice on the programmatic level, the Open set in Dijkstra’s algorithm is commonly implemented as a priority queue. It is a data structure typically based on a binary heap\(^2\) which helps to efficiently retrieve the successive states from the queue preserving the order induced by the \( g \) function. Addition of an element (state) into the priority queue is of logarithmic complexity — \( O(\log_2 n) \), where \( n \) stands for the number of elements in the queue. Removal of the minimal element from the head of the queue (poll operation) is also \( O(\log_2 n) \) complex. The latter can be explained by the fact that although the minimal element can be quickly peeked at in \( O(1) \) time, as it is the first element in the heap, after this element is removed the heap must be rearranged\(^3\) which requires logarithmic time. As regards the ‘replace’ or ‘contains’ operation performed on a priority queue (needed in lines no. 14–19 in the algorithm), they are unfortunately of linear time — \( O(n) \) — at least for typical priority queue implementations in existing programming languages. In SaC, we offer some improvements with respect to these operations by applying auxiliary data structures. This topic is discussed further in the API-related section 3.2 for graphs.

As regards implementations of the Closed set, one should mostly care about fast performance of ‘contains’ and ‘add’ operations. Therefore, typically Closed sets are implemented as hash maps having constant or amortized-constant complexity — \( O(1) \). The hash maps are fast at the cost of RAM memory consumption. This might become a problem when very large graphs are to be searched. In SaC, we offer an alternative possibility to use a sorted tree as a Closed set, which allows for some memory savings but implies logarithmic complexity — \( O(\log_2 n) \).

### 3.1.3 Best-first search (BFS)

Pearl (1984) proposed a search approach in which the algorithm expands always the most promising (the ‘best’) state in the first order. The evaluation how promising a state \( s \) is, is performed by a so called heuristic function \( h(s) \). This function can be constructed differently, and in general may depend on: the information contained in \( s \) itself, the information collected along the path from the initial state up to \( s \), some general knowledge about the problem and the desired properties of the solution state. By convention \( h(s) \) is defined as non-negative function and its smaller values suggest closeness of \( s \) to the goal state. Informally, \( h(s) \) can be therefore treated as a distance function.

In general, Best-first search algorithms and heuristics applied in them are designed to quickly reach the goal state by any path. Therefore, they do not focus on optimizing the path anyhow. In

\(^2\)Binary heaps can be regarded as binary trees, commonly implemented using dynamically extensible arrays. An element at index \( i \) in the array has its children under indices \( 2i + 1 \) and \( 2i + 2 \) (with indexing starting from zero). Oppositely, the parent of the \( i \)-th element is under the index \( \lfloor (i - 1)/2 \rfloor \). There are two types of heaps: max-heaps and min-heaps depending on the wanted order to be imposed. For Dijkstra’s algorithm a min-heap is used, satisfying the following condition: \( g(s_1) \leq g(s_2) \) and \( g(s_1) \leq g(s_3) \), for all \( s_1, s_2, s_3 \) where \( s_2, s_3 \) are children elements (in the heap’s array) for the element \( s_1 \).

\(^3\)The rearrangement of the heap is carried out by placing temporarily the last element from the underlying array in the place of the first element, and by moving this element downwards (swapping it with one of its children) as long as the heap condition is not met. The number of swaps is at most equal to the height of the tree represented by the heap, thus approximately \( O(\log_2 n) \).
other words by using the best-first approach we do not care how long the path is, neither in terms of the number of states (manipulations) along it, nor its cost. In fact, the notion of cost of the path (e.g. as it was defined by the g function for Dijkstra’s algorithm) does not exist in the BFS.

Alg. 4 Best-first search

1: procedure **BestFirstSearch**(s₀) ▷ initial state s₀
2:  
3:  
4:  
5:  
6:  
7:  
8:  
9:  
10:  
11:  
12:  
13:  
14:  
15:  
16:  
17:  
18:  
19:  
20:  
21: return null ▷ no solution found

A comment should be made on the fragment of the algorithm (lines 16–19), where it discovers, for a descendant state t, that another instance of that state already exists in the Open set, and then checks if the newly calculated h(t) is smaller (better) than the value known so far. A question can be posed: can the heuristic function return different values for two instances of the same state? In the general case, the answer is ‘yes’ — e.g. if the h function is not solely the function of a state, but also depends on some information collected along the path. Nevertheless, most commonly in practice the BFS applications use heuristic functions being constant for multiple instances of the same state.

3.1.4 A∗

The A∗ algorithm was proposed by Haart, Nilsson and Raphael in (Hart, Nilsson and Raphael, 1968; Hart, Nilsson and Raphael, 1972). Informally, the algorithm can be regarded as a combination of Dijkstra’s and BFS algorithms (or a more general variant of them). It is because the algorithm uses both the exact cost function g and the heuristic cost function h to decide about the order in
which states are visited.

More strictly, let the evaluation function (deciding about the retrieval order from the Open set) for a state \( s \) be of form:

\[
f(s) = g(s) + h(s),
\]

where \( g(s) \) is the exact cost or distance (observed along the way) for traveling from the initial state to the state \( s \), and \( h(s) \) is a heuristic estimation of the cost remaining to the goal state. Because the summand \( h(s) \) is a heuristic function, the whole \( f(s) \) function can also be regarded as a heuristics.

Beneath, we present the pseudocode of the algorithm.

Alg. 5 \( A^* \)

1: procedure AStar(\( s_0 \)) \( \triangleright \) initial state \( s_0 \)
2: \( \text{Closed} := \emptyset \) \( \triangleright \) empty set of visited states
3: \( g(\( s_0 \)) := 0 \) \( \triangleright \) distance covered from start
4: calculate \( h(\( s_0 \)) \) \( \triangleright \) heuristics according to provided recipe
5: \( f(\( s_0 \)) := g(\( s_0 \)) + h(\( s_0 \)) \) \( \triangleright \) sum deciding about order of Open queue
6: set reference from \( s_0 \) to its parent to null
7: \( \text{Open} := \{\( s_0 \}\} \) \( \triangleright \) queue of states to be visited
8: while \( \text{Open} \neq \emptyset \) do
9: remove from Open the state \( s \) with the smallest \( f(s) \) \( \triangleright \) ‘poll’ operation
10: if \( s \) is the goal state then return \( s \) \( \triangleright \) solution found
11: generate descendents \( \{t\} \) of \( s \)
12: for all \( t \) do
13: if \( t \in \text{Closed} \) then continue \( \triangleright \) \( t \) already visited
14: \( g(t) := g(s) + \Delta(s \rightarrow t) \)
15: calculate \( h(t) \)
16: \( f(t) := g(t) + h(t) \)
17: set reference from \( t \) to its parent to \( s \)
18: if \( t \notin \text{Open} \) then
19: add \( t \) to Open
20: else
21: if new \( f(t) \) is smaller than value known so far then
22: replace \( t \) in Open with the new one
23: update position of \( t \) in Open
24: add \( s \) to Closed
25: return null \( \triangleright \) no solution found

For the shortest (smallest cost) path finding applications, it is crucial that the \( h \) function is a so called \textit{admissible heuristics}. This means that \( h \) must not overestimate the unknown true cost remaining to the goal state. In other words \( h \) should be a lower bound on the remaining cost.

If the heuristics is admissible and the \( A^* \) algorithm finds a goal state than it is guaranteed that this state is the optimal solution — has the shortest path. \textit{Proof}: Imagine the \( A^* \) algorithm stops at a certain point (line no. 10) and returns a state \( s' \) with the cost value \( g(s') \). Obviously, \( h(s') = 0, \)
since $s^*$ satisfied the stop condition. We know that the Open queue preserves the non-decreasing order of removal with respect to $f$ function for successive states. Therefore, it is known that all states $s \in$ Open, remaining in the queue at the stop moment, satisfy the condition $f(s) \geq f(s^*)$. Two cases should now be considered. (1) If some state $s$ satisfies the stop condition, then $h(s) = 0$ but $g(s) \geq g(s^*)$, since $f(s) \geq f(s^*)$. In other words $s$ is also the goal state but with the cost of its path not cheaper than the cost of path for $s^*$. (2) If some state $s$ does not satisfy the stop condition, but can potentially lead to the goal state later and we have that $g(s) \leq g(s^*)$, then the final path will for sure be not cheaper than the path for $s^*$, because the lower bound $h(s) > 0$ on the remaining true cost indicates that $g(s) + h(s) \geq g(s^*)$, again because $f(s) \geq f(s^*)$. ■

An auxiliary notion of monotnous heuristics is also useful in this context. A heuristics is said to be monotnous if for all pairs $s, t$, where $t$ is a descendant of $s$, the following condition is satisfied:

\[ f(s) \leq f(t), \]  

(3.2)

which can also be rewritten as

\[
\begin{align*}
g(s) + h(s) & \leq g(t) + h(t), \\
h(s) & \leq g(t) - g(s) + h(t), \\
h(s) & \leq \Delta(s \rightarrow t) + h(t).
\end{align*}
\]  

(3.3)

The last inequality can be restated as follows: the heuristic value at $s$ must not be greater than the cost of transition from $s$ to $t$ plus the heuristic value at $t$. The inequality can become equality only in cases when one travels towards the goal along a straight line (in the sense of metrics associated with the given graph). Moreover, if a heuristics is monotonous then it is also admissible.

Historically, the algorithm was first named as $A$ algorithm. In the notational sense, this was related to the fact that if we consider two heuristics $h$ and $h^*$, where $h^*$ is marked with a star to denote an optimal heuristics (returning exact costs remaining to the goal), then the algorithm using $h^*$ can also be called optimal and denoted as $A^*$. Moreover, the optimality of such an algorithm is twofold. Firstly, it performs best (visits the fewest states) among all $A$ algorithms. Secondly, it performs not worse than all other graph search algorithms (not necessarily from the family of $A$ algorithms) which are worse or equally well informed in the sense of information stored in $h^*$.

### 3.1.5 IDA*

For some problems where the search graph is very large, the $A^*$ algorithm may run into RAM memory consumption troubles. The number of states in both Open and Closed sets might potentially exhaust the whole RAM.

Iterative Deepening $A^*$ (IDA*) proposed by Korf (1985) can be viewed as a variant of $A^*$ with low memory consumption. The IDA* does not keep record of visited states, i.e. does not use the Closed set. Also, it typically keeps in memory only the states residing on the path it currently pursues. Depending on the algorithm formulation, it either uses only a small Open set (non recursive formulation with a main while loop) or does not use the Open set at all (recursive formulation). Obviously, the low memory usage is not for free — the IDA* is slower than $A^*$ because it visits many states multiple times.
The idea behind the IDA* can be sketched as follows. First, the algorithm uses the heuristic value calculated for the initial state $h(s_0)$ to establish the search horizon $H = f(s_0) = 0 + h(s_0)$. Then, the algorithm follows different search paths outgoing from the initial state in a depth-first manner. If it reaches a goal state within the search horizon (i.e. if the observed cost $g$ for the reached state is less or equal to $H$) then it returns the goal and the associated path. Whenever the algorithm reaches a state outside the search horizon, then the state is not pursued further (its descendants are not generated), but the algorithm may use the observed cost for this state and its heuristics to establish a new search horizon $H'$. More precisely the new search horizon is derived as:

$$H' = \min_{|s: g(s) > H|} f(s).$$  \hspace{1cm} (3.4)

Finally, when all paths reaching outside the current horizon $H$ are exhausted then the algorithm deepens the search horizon by the substitution $H := H'$, and the whole process is repeated.

We now present the pseudocodes for two IDA* formulations — a recursive one and a non-recursive (with a main while loop).

**Alg. 6 Recursive IDA***

1. **procedure** **RecursiveIterativeDeepeningAStar**($s_0$)  \hspace{1cm} ➔ initial state $s_0$
   2. $g(s_0) := 0$  \hspace{1cm} ➔ distance covered from start
   3. calculate $h(s_0)$  \hspace{1cm} ➔ heuristics according to provided recipe
   4. $f(s_0) := g(s_0) + h(s_0)$  \hspace{1cm} ➔ sum deciding about order of Open queue
   5. set reference from $s_0$ to its parent to null
   6. $H := f(s_0)$  \hspace{1cm} ➔ first search horizon
   7. **while** true **do**
   8. $(s, H') := \text{Search}(s_0, H)$
   9. **if** $s \neq \text{null} \text{ then return } s$  \hspace{1cm} ➔ solution found
   10. **if** $H' = \infty \text{ then return } \text{null}$  \hspace{1cm} ➔ no solution found
   11. $H := H'$
   12. **procedure** **Search**($s, H$)  \hspace{1cm} ➔ initial state $s_0$
   13. **if** $f(s) > H \text{ then return } \left(\text{null}, f(s)\right)$
   14. **if** $s$ is the goal state **then return** $(s, g(s))$  \hspace{1cm} ➔ solution found
   15. $H' := \infty$
   16. generate descendents $\{t\}$ of $s$
   17. **for all** $t$ **do**
   18. $g(t) := g(s) + \Delta(s \rightarrow t)$
   19. $f(t) := g(t) + h(t)$
   20. $(u, H'') := \text{Search}(t, H)$
   21. **if** $u \neq \text{null} \text{ then return } (u, g(u))$  \hspace{1cm} ➔ solution found
   22. $H' := \min\{H', H''\}$  \hspace{1cm} ➔ deepening the horizon
   **return** $(\text{null}, H')$
Alg. 7 IDA*

1: procedure IterativeDeepeningAStar(s₀)  
   ▷ initial state s₀
2:   g(s₀) := 0  
   ▷ distance covered from start
3:   calculate h(s₀)  
   ▷ heuristics according to provided recipe
4:   f(s₀) := g(s₀) + h(s₀)  
   ▷ sum deciding about order of Open queue
5:   set reference from s₀ to its parent to null
6:   Open := {s₀}  
   ▷ queue of states to be visited
7:   H := f(s₀)  
   ▷ first search horizon
8:   H′ := ∞  
   ▷ next search horizon
9:   while Open ≠ ∅ do
10:      remove from Open the state s with the smallest f(s)  
      ▷ ‘poll’ operation
11:         if g(s) > H then
12:            H′ := min(H′, f(s))
13:            if Open = ∅ then
14:               H := H′  
               ▷ deepening the horizon
15:               H′ := ∞
16:               Open := {s₀}
17:               continue
18:         if s is the goal state then return s  
      ▷ solution found
19:      generate descandants {t} of s
20:      for all t do
21:         g(t) := g(s) + Δ(s → t)
22:         calculate h(t)
23:         f(t) := g(t) + h(t)
24:         set reference from t to its parent to s
25:         if t ∉ Open then
26:            add t to Open
27:        else
28:            if new f(t) is smaller than value known so far then
29:               replace t in Open with the new one
30:               update position of t in Open
31:      return null  
      ▷ no solution found

3.2 API

3.2.1 Graph state abstraction

In SaC’s API for searching graphs, the main ‘actor’ is the sac.graph.GraphState interface being an extension of the sac.State discussed earlier in Chapter 2. Below, we present the code listing for the sac.graph.GraphState interface and then some excerpts from its default implementation — sac.graph.GraphStateImpl. For full code of the implementation the reader is addressed to Appendix 6.2.
The first thing to note is that the GraphState interface reformulates the signatures of four basic methods related to the parent – children binding and path tracking. The difference (of cosmetic nature) is in the types returned, which now become GraphState or List<GraphState>, as the resulting objects should be treated as graph states (rather than general states).

The new elements in the interface (with respect to its ancestor sac.State) are cost related methods: getG(), getF(), refreshCosts(), and the isSolution() method. As regards the cost related methods the relevant excerpts from their implementation are listed below.

```java
public abstract class GraphStateImpl extends StateImpl implements GraphState {
    
    protected Double g = 0.0;
    protected Double f = null;
    protected static StateFunction gFunction;

    public static class GFunction extends StateFunction {
        @Override
        public double calculate(State state) {
            return (state.getParent() == null) ? 0.0 : ((GraphState) state.getParent()).getG() + 1.0;
        }
    }
}
```
As one may note, graph states are by default equipped with a function of observed cost (g function) which treats each transition from a parent to its child as being of cost 1. This default behavior is often valid for many graph problems where our objective is to minimize the number of moves or manipulations leading from the initial state to the goal state (e.g. sliding puzzle, Rubik’s cube, simple non-weighed graphs). This assumption makes the g function behave identically as depth. If it is user’s wish to change this default behaviour for a given problem, he must provide an implementation of a StateFunction and attach statically it to his class (being an extension of the GraphState class) by the setGFunction(...) method.

It is worth to remark that the calculations of g and f functions are postponed until the very first calls of respective getters — getG() and getF(). It is the same design idea as was discussed earlier in the context of identifiers and heuristics (h function). And its purpose is to minimize the number of actual calculations. The refreshCosts() method is meant to be called by search algorithms, typically just after the moment where descendant states are generated and then processed one after another. Explicit user calls to this method are in practice not needed (exceptions are the presentation-related or online game playing situations discussed already in Section 2.5).

As regards the isSolution() method, the user should provide its implementation in compliance with the nature of his search problem. When a state represents the goal state (satisfies the desired terminal properties) then the isSolution() method should naturally return true (otherwise — false).

### 3.2.2 General graph search algorithm

In this section we discuss SaC’s internal mechanisms related to the general graph searching procedure. This procedure is represented by an abstract class: sac.graph.GraphSearchAlgorithm. Obviously, the end user of SaC library does not have to be aware of core-level intricacies. Instead, in practice he can limit himself to instantiating a specific search algorithm (e.g.: new Dijkstra(...) or new AStar(...), etc.) and running it for his particular problem. Therefore, we recommend the reading of subsequent contents only to readers really interested in low level details of SaC,
perhaps the readers intending to extend the library in the future with new algorithms or data structures.

The listing below demonstrates the most important parts of the GraphSearchAlgorithm class. For clarity, some less interesting parts have been skipped. The class is designed to work as a general and common (model) procedure for searching graphs. As the reader shall see later, the subclasses representing specific search algorithms (e.g. BreadthFirstSearch, DepthFirstSearch, Dijkstra, etc.) are, in the programmistic sense, very light extensions of the GraphSearchAlgorithm class. In fact, those light extensions only redefine the order according to which states are removed from the Open set.

```java
package sac.graph;

... public abstract class GraphSearchAlgorithm extends SearchAlgorithm {

protected GraphState initial = null;
protected OpenSet openSet = null;
protected ClosedSet closedSet = null;
protected GraphSearchConfigurator configurator = null;
protected List<GraphState> solutions = null;
protected int step = 0;
protected GraphState current = null;
protected GraphState bestSoFar = null;

public GraphSearchAlgorithm(GraphState initial, GraphSearchConfigurator configurator) {
    ...
}

protected void setupOpenAndClosedSets(Comparator<GraphState> openSetComparator) {
    ...
}

protected void reset() {
    ...
}

... // getters, setters

@Override
public void execute() {
    ... // initialization of data structures and monitoring thread
    doExecute(); // actual search start
    ... // stoppage of monitoring threads
}

protected void doExecute() {
    ... // the body of the main search procedure, presented later on
}

4 Exception to this rule is the IterativeDeepeningAStar class, which additionally overrides the main search procedure — execute().
```
We now concentrate on the `doExecute()` method. It can be viewed as a counterpart of algorithmic pseudocodes discussed earlier in Section 4.1. Here is the full listing of the method.

```java
protected void doExecute() {
    startTime = System.currentTimeMillis();
    if (initial == null)
        return;
    openSet.add(initial);
    step = 0;
    while (!openSet.isEmpty()) {
        step++;
        // time limit check
        if (configurator.getTimeLimit() < Long.MAX_VALUE) {
            long currentTime = System.currentTimeMillis();
            if (currentTime - startTime > configurator.getTimeLimit()) {
                endTime = System.currentTimeMillis();
                break;
            }
        }
        // poll current best from queue
        current = openSet.poll();
        // putting current to closed set
        if (configurator.isClosedSetOn())
            closedSet.put(current);
        // keeping best so far
        if ((initial.getH() > 0) && ((bestSoFar == null) || (current.getH() < bestSoFar.getH())))
            bestSoFar = current;
        // solution check
        boolean isSolution = current.isSolution();
        // registering solution
        if (isSolution) {
            if (!solutions.isEmpty())
                bestSoFar = current;
            solutions.add(current);
            if (configurator.getWantedNumberOfSolutions() == solutions.size())
                break;
        }
        // generating children
        List<GraphState> children = current.generateChildren();
        // iterating over children
        for (GraphState child : children) {
            // check if child was closed
            boolean closedSetContains = (configurator.isClosedSetOn()) ? closedSet.contains(child) : false;
            if (!closedSetContains) { // child not in closed set
                // set child -> parent link and depth
                child.setParent(current);
                child.setDepth(current.getDepth() + 1);
```
Apart from several auxiliary operations (time limit checking, memorizing the solutions, memorizing the best state so far) the important steps involved in the above procedure proceed as in most of algorithms discussed earlier.

The two crucial places to note are invocations of callback methods: `current.isSolution()` and `current.generateChildren()`. Their implementations must be provided by the user. Note also that found solutions are not returned directly from the procedure, but memorized in the internal field (List<GraphState> solutions) of the class. Once the wanted number of solutions is reached (by default it is 1), the procedure is stopped.

The `doExecute()` method is not called directly by the user (it is under protected scope). Instead, the user calls a wrapper method `execute()`. Its full listing is as follows.

```java
public void execute() {
    reset();
    GraphSearchMonitor monitor = null;
    Thread monitorThread = null;
    if (configurator.isMonitorOn()) {
        // starting monitor thread
        monitor = new GraphSearchMonitor();
        monitorThread = new Thread(monitor);
        monitorThread.start();
    }
    // time limit checking
    long endTime = System.currentTimeMillis();
    // updating g, h, f
    child.refreshCosts();
    // checking if child is in open set
    boolean openSetContains = openSet.contains(child);
    if (!openSetContains) {
        // adding child reference to parent
        if (configurator.isParentsMemorizingChildren())
            current.getChildren().add(child);
        // adding child to open set
        openSet.add(child);
    } else {
        // getting reference to child existing in open set
        GraphState existingChild = openSet.get(child);
        // replacing, if new child better than existing
        if (openSet.getComparator().compare(child, existingChild) < 0) {
            openSet.replace(existingChild, child);
            // adding child reference to parent (better child)
            if (configurator.isParentsMemorizingChildren())
                current.getChildren().add(child);
            // removing from some other parent reference to worse existing child
            existingChild.getParent().getChildren().remove(existingChild);
        }
    }
    endTime = System.currentTimeMillis();
}
```
As one can note, the overhead around the actual `doExecute()` call boils down to: resetting the algorithm and operations on the monitoring thread. As regards the latter, the monitoring thread allows the user to watch the progress of a graph search, e.g.: number of states in `Open` and `Closed` sets, number of solutions found so far, RAM usage, etc. There are two monitor classes provided with SaC: `sac.graph.ConsoleGraphSearchMonitor` and `sac.graph.GraphicalGraphSearchMonitor`. Their common abstraction is a superclass named: `sac.graph.ConsoleGraphSearchMonitor` — the type actually used in the code. More details on monitors is given later in Section 5.3. Before starting a graph search, the suitable monitor is brought to life (using information from the configurator object) and triggered as a side thread. After the `doExecute()` method is finished, the monitor thread is stopped.

As regards the `reset()` method, it involves the following operations: (1) setting up the wanted type for state identifiers (hash codes or strings) from the configurator object\(^5\); (2) refreshing the initial object (in particular: detaching the list of its descendants present after the previous search), (3) clearing the solutions list, (4) nulling auxiliary references to current and best states, (5) setting up data structures for `Open` and `Closed` sets. Below we show the listing of the `reset()` method.

```java
protected void reset() {
    Identifier.setType(this.configurator.getIdentifierType());
    if (initial != null) {
```

\(^5\)Its content might have changed since the last search execution.
The setupOpenAndClosedSets(...) method inspects the information stored in the configurator object and instantiates, by means of the Java reflection mechanism, suitable implementations for Open and Closed sets (before the search is started). The listing of the method is as follows.

```java
protected void setupOpenAndClosedSets(Comparator<GraphState> openSetComparator) {
    // open set
    try {
        Constructor<OpenSet> constructor = (Constructor<OpenSet>) Class.forName(configurator.
            getOpenSetClassName()).getConstructor(Comparator.class);
        this.openSet = (OpenSet) constructor.newInstance(openSetComparator);
    } catch (Exception e) {
        this.openSet = new OpenSetAsPriorityQueueFastContainsFastReplace(openSetComparator);
        e.printStackTrace();
    }

    // closed set
    if (configurator.isClosedSetOn())
        try {
            Constructor<ClosedSet> constructor = (Constructor<ClosedSet>) Class.forName(
                configurator.getClosedSetClassName()).getConstructor();
            this.closedSet = (ClosedSet) constructor.newInstance();
        } catch (Exception e) {
            this.closedSet = new ClosedSetAsHashMap();
            e.printStackTrace();
        }
}
```

This method is also of importance in the context of defining specific search algorithms (subclasses of the general and abstract GraphSearchAlgorithm), which is discussed in the subsequent section.

### 3.2.3 Specific graph search algorithms

As one may have noted, the argument passed to the setupOpenAndClosedSets(...) method is a comparator Comparator<GraphState> openSetComparator. It decides about the states retrieval order associated with the Open set. Both the setupOpenAndClosedSets(...) method and the mentioned comparator play an important role in defining the specific search algorithms. This can be best depicted by the following example.

```java
package sac.graph;
import java.util.Comparator;
```
The presented class is an actual definition of the $A^*$ algorithm in SaC. As one can see the code is very short. The class extends the `GraphSearchAlgorithm` abstraction, and in its main constructor it instantiates a suitable comparator for the `Open` set. For convenience, the comparator is defined as an inner class (bottom part of the code). In the example, the comparator works as it should be working for the case of $A^*$ algorithm — $f$ function values are used for comparing states. In the case of a tie, the comparison of identifiers decides. In other words, states with equal $f$ values are sorted lexicographically.

The proceedings described above are common for all specific graph search algorithms. To demonstrate it, we present another example, this time of the `BestFirstSearch` class. The programmistic scheme is repeated, and the resulting code is very short again.
3.2.4 Variants of Open and Closed sets

In SaC, actual implementations of Open or Closed sets are separated, in the programmistic sense, from their abstractions — sac.graph.OpenSet and sac.graph.ClosedSet interfaces, respectively. The listings below present the methods offered by these interfaces.

```java
package sac.graph;
import java.util.Comparator;

public interface OpenSet {
    public void add(GraphState graphState);
    public GraphState poll();
    public GraphState peek();
    public boolean contains(GraphState graphState);
    public void replace(GraphState graphState, GraphState replacer);
    public GraphState get(GraphState graphState);
    public int size();
    public boolean isEmpty();
    public void clear();
    public Comparator<GraphState> getComparator();
}

package sac.graph;
public interface ClosedSet {
    public boolean contains(GraphState graphState);
    public GraphState get(GraphState graphState);
    public void put(GraphState graphState);
    public void remove(GraphState graphState);
    public int size();
    public boolean isEmpty();
    public void clear();
}
```

Currently in SaC, available are three variants (implementations) of the Open set and two variants of the Closed set, as listed in tables 3.1 and 3.2. The variants differ with respect to the
computational complexity of operations involved. Faster variants are naturally more memory consuming.

Table 3.1: Available variants (implementations) of the Open set and their computational complexities.

<table>
<thead>
<tr>
<th>class name (in sac.graph package)</th>
<th>poll</th>
<th>contains/get</th>
<th>add</th>
<th>replace</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSetAsPriorityQueue</td>
<td>$O(\log_2 n)$</td>
<td>$O(n)$</td>
<td>$O(\log_2 n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>OpenSetAsPriorityQueueFastContains</td>
<td>$O(\log_2 n)$</td>
<td>$O(1)$</td>
<td>$O(\log_2 n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>OpenSetAsPriorityQueueFastContainsFastReplace</td>
<td>$O(\log_2 n)$</td>
<td>$O(1)$</td>
<td>$O(\log_2 n)$</td>
<td>$O(\log_2 n)$</td>
</tr>
</tbody>
</table>

Table 3.2: Available variants (implementations) of the Closed set and their computational complexities.

<table>
<thead>
<tr>
<th>nazwa klasy (w sac.graph)</th>
<th>contains/get</th>
<th>put</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClosedSetAsTreeMap</td>
<td>$O(\log_2 n)$</td>
<td>$O(\log_2 n)$</td>
</tr>
<tr>
<td>ClosedSetAsHashMap</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

As regards the Open set, the first implementation listed in Table 3.1 is OpenSetAsPriorityQueue. It is based on a standard priority queue class — java.util.PriorityQueue — offered within J2SE. This class in turn is underneath based on a binary heap data structure. It is very likely that the java.util.PriorityQueue class would be a programmer’s choice if a search algorithm was to be implemented in Java from scratch. Although poll and add operations are of logarithmic complexity — $O(\log_2 n)$ — for this class, the contains operation is linear — $O(n)$. The reason is that to look for a particular object within a binary heap (contains operation), one must sequentially traverse all the objects it contains. As regards the replace operation, it is worth to remark that a suitable method for that purpose is not offered directly by J2SE in the java.util.PriorityQueue class. In order to do a replacement one must combine several methods. First, find the object by iterating over all objects in the queue (linear time), then remove it (logarithmic time), and finally insert its replacement into the heap (logarithmic time). Therefore, the finding part is the slowest, which makes the complexity of the whole operation $O(n)$.

The second variant of the Open set is the OpenSetAsPriorityQueueFastContains class. It is faster from the first variant with respect to the contains operation. On the low level, apart from a standard priority queue we have equipped this class with an auxiliary hash map (java.util.HashMap). The hash map memorizes the pairs <Identifier, GraphState>, where the Identifier works as a key and the reference to GraphState works as a value in the map. Every time a state is being added to the Open set, it is added both the priority queue and the hash map. Owing to the latter, one is able later to quickly check, in $O(1)$ time, if a state with a particular identifier is present in Open. Moreover, since identifiers are coupled with references to states, one can also quickly check if the value of a suitable comparing function ($g, f, h$, depth, etc.) in the existing state is better than for the state currently processed. By that one knows if a replace operation is necessary.

The third variant of the Open set is the class: OpenSetAsPriorityQueueFastContainsFastReplace. This is the variant turned on by default. The class uses a custom implementation of the binary heap, provided by the authors of SaC. The heap is implemented as a dynamic array data structure.
As an auxiliary structure, a hash map is again involved. Firstly, this hash map makes the contains operation $O(1)$ fast (as in the former variant). But secondly (and more importantly), this hash map allows also to have a quick access to array indices of states contained in the heap. The following code excerpt shows how this is achieved.

```java
package sac.graph;

public class OpenSetAsPriorityQueueFastContainsFastReplace extends OpenSetImpl {

    private ArrayList<GraphState> binaryHeap;
    private Map<Identifier, MapEntry> map;

    private class MapEntry {
        private GraphState graphState;
        private int binaryHeapIndex;
    }

    public OpenSetAsPriorityQueueFastContainsFastReplace(Comparator<GraphState> comparator) {
        super(comparator);
        binaryHeap = new ArrayList<GraphState>(1024 * 1024);
        map = new HashMap<Identifier, MapEntry>(1024 * 1024, (float) 0.75);
    }

    @Override
    public void replace(GraphState graphState, GraphState replacer) {
        Identifier identifier = graphState.getIdentifier();
        Integer index = map.get(identifier).getBinaryHeapIndex();
        if (index == null)
            return;
        binaryHeap.set(index.intValue(), replacer);
        map.get(identifier).setGraphState(replacer);
        reheapUp(index);
    }

    protected void reheapUp(int childIndex) {
        if (childIndex == 0)
            return; // stop of recursion
        int parentIndex = (childIndex - 1) / 2;
        GraphState parent = binaryHeap.get(parentIndex);
        GraphState child = binaryHeap.get(childIndex);
        if (comparator.compare(parent, child) > 0) { // comparator comes from OpenSetImpl
            binaryHeap.set(parentIndex, child);
            binaryHeap.set(childIndex, parent);
            map.get(child.getIdentifier()).setBinaryHeapIndex(parentIndex);
            map.get(parent.getIdentifier()).setBinaryHeapIndex(childIndex);
            reheapUp(parentIndex);
        }
    }
}
```

When a replacement is to be done, the suitable index is read from the map and the replacing (better) state is inserted into the heap related array under that index. Then, the heap has to be
reorganized upwards — \texttt{reheapUp(...)} method — starting on from the new state, so that the heap condition is satisfied. This is done in $O(\log_2 n)$ time.

We now move on to discuss \textit{Closed} set variants.

Currently in \textit{SaC}, available are two \textit{Closed} set variants. The default one is represented by the class: \texttt{ClosedSetAsHashMap}. As the name suggests, it is based on a standard (J2SE) hash map — \texttt{java.util.HashMap} — allowing for $O(1)$ complexity of all operations one needs in graph searching: \texttt{put}, \texttt{get}, \texttt{contains}. The trade-off is obviously in high memory consumption. By default, an instantiation of the \texttt{ClosedSetAsHashMap} class, creates underneath a map with an initial capacity of $\approx 8 \cdot 10^6$ entries, and the \textit{load factor} of 0.75 (as shown in the listing below).

```java
package sac.graph;

...

public class ClosedSetAsHashMap implements ClosedSet {

    private Map<Identifier, GraphState> map;

    public ClosedSetAsHashMap() {
        this.map = new HashMap<Identifier, GraphState>(8 * 1024 * 1024, (float) 0.75);
    }

    ...

}
```

The second \textit{Closed} set variant — \texttt{ClosedSetAsTreeMap} — has been provided in \textit{SaC} with an intention to mitigate some of memory consumption problems one can come across more quickly when using a slightly faster hash map based variant. The \texttt{ClosedSetAsTreeMap} implementation is based on the J2SE’s \texttt{java.util.TreeMap} class, which in turn underneath is based on the \textit{red–black tree} data structure\footnote{Red–black trees are self-balancing binary search trees. The idea is due to Bayer (1972) who originally named the structure as \textit{symmetric binary B-Tree}. The contemporary modern name is due to the paper (Guibas and Sedgewick, 1978). Balance of the tree is preserved by painting each tree node with one of two colors (by convention red and black) and imposing certain properties on occurrence of the colors. The properties are designed in such a manner that tree rearrangements and repaintings are carried out efficiently — in $O(\log_2 n)$ time. The tree is typically not perfectly balanced, but guarantees that the distance from the root to the furthest leaf is at maximum twice as long as the distance to the closest leaf. This makes the complexity of all crucial operations proportional to the height of the tree, thus also $O(\log_2 n)$}. The red-black tree guarantees the complexity of all important (for graph searching) operations — \texttt{put}, \texttt{get}, \texttt{contains} — to be $O(\log_2 n)$.
3.2.5 Configuration options for searching graphs

On several occasions we have mentioned the usage of a GraphSearchConfigurator object. Below, we present a brief code listing of this class, with all configuration options and their default values. We purposely leave the javadocs present in the listing to make the meaning of options more clear. Every configuration option can be accessed by a suitable getter or setter (that are skipped in the listing).

```java
public class GraphSearchConfigurator {

    /**
     * Identifier type for states. By default: HASH_CODE.
     */
    private IdentifierType identifierType = IdentifierType.HASH_CODE;

    /**
     */
    private String openSetClassName = OpenSetAsPriorityQueueFastContainsFastReplace.class.getName();

    /**
     * Is closed set on. By default: true. Closed set can be off when the search space is a tree 
     * (not a graph with cycles).
     */
    private boolean closedSetOn = true;

    /**
     * Class name for closed set. By default: sac.graph.ClosedSetAsHashMap.
     */
    private String closedSetClassName = ClosedSetAsHashMap.class.getName();

    /**
     * Do parents memorize references to their children. Set to false for lower memory usage ( 
     * WARNING: in that case drawing graph via GraphViz is impossible). By default: false.
     */
    private boolean parentsMemorizingChildren = false;

    /**
     * Wanted number of solutions. By default: 1.
     */
    private int wantedNumberOfSolutions = 1;

    /**
     */
    private long timeLimit = Long.MAX_VALUE;

    /**
     * Is monitor on. By default: false;
     */
    private boolean monitorOn = false;

    /**
     * Class name for monitor. By default: sac.graph.DefaultConsoleMonitor.
     */
    private String monitorClassName = ConsoleGraphSearchMonitor.class.getName();
}
```
A configurator can be instantiated either by a default constructor with no arguments — 
GraphSearchConfigurator(), or from a properties text file — 
GraphSearchConfigurator(String propertiesFilePath). The names of options in the properties file are uppercased versions of field names, and with underscores separating successive words. Below, we show a possible exemplary contents of a configuration .properties file with some non-default values.

```java
//Example of graph configurator settings
identifierType=STRING
openSetClassName=sac.graph.OpenSetAsPriorityQueueFastContains
closedSetOn=true
closedSetClassName=sac.graph.ClosedSetAsHashMap
parentsMemorizingChildren=false
wantedNumberOfSolutions=5
timeLimit=Long.MAX_VALUE
monitorOn=false
monitorClassName=sac.graph.GraphicalGraphSearchMonitor
monitorRefreshTime=250
```

### 3.3 Examples

#### 3.3.1 Sliding puzzle

The sliding puzzle is a one-player recreation invented by Noyes Chapman in 1880. The player is given a board containing flat pieces — tiles — commonly with a picture or numbers drawn on them. In the initial state the pieces are shuffled. There is an empty space in the place of one of the tiles and the player can slide adjacent tiles into the empty space. The goal is to bring the board back to its original arrangement. In contrary to human players, from a computer solver we typically require also that the found sequence of moves is minimal.

Most commonly the boards with sliding puzzles are square grids, in general of size \( n \times n \), see Fig. 3.1\footnotemark[7] Due to one tile taken away, the puzzle is also known as \( (n^2 - 1) \)-puzzle. Eight-puzzles are quite easily solvable by children, but some fifteen-puzzle instances can already be out of reach for adults, especially if the minimal path to solution should be found.

\footnotetext[7]{Images acquired from Google Images search engine.}
Implementation of sliding puzzle state

Below we show an implementation of a sliding puzzle state using SaC. We represent the puzzle as a one-dimensional array of bytes and implement the required routines: generation of descendents — `generateChildren()`, identification — `hashCode()`, termination — `isSolution()`. For convenience, we introduce a copying constructor and some helper methods: to get possible moves, to make a move, and to shuffle the puzzle. In the ending static block we attach the heuristics to the class — `HFunctionLinearConflicts`, which is discussed a bit later on.

```
public class SlidingPuzzle extends GraphStateImpl {
    protected static byte n;
    protected static byte N; // N = n * n
    protected byte emptyIndex;
    protected byte[] board;

    public SlidingPuzzle(int n) {
        SlidingPuzzle.n = (byte) n;
        SlidingPuzzle.N = (byte) (n * n);
        board = new byte[N];
        for (byte i = 0; i < N; i++)
            board[i] = i;
        emptyIndex = 0;
    }

    public SlidingPuzzle(SlidingPuzzle parent) {
        board = new byte[N];
        for (byte i = 0; i < N; i++)
            board[i] = parent.board[i];
        emptyIndex = parent.emptyIndex;
    }
}
```
public LinkedList<Byte> getPossibleMoves() {
    LinkedList<Byte> list = new LinkedList<Byte>();
    if ((emptyIndex % n) + 1 < n)
        list.add((byte) (emptyIndex + 1));
    if ((emptyIndex % n) - 1 >= 0)
        list.add((byte) (emptyIndex - 1));
    if (emptyIndex + n < N)
        list.add((byte) (emptyIndex + n));
    if (emptyIndex - n >= 0)
        list.add((byte) (emptyIndex - n));
    return list;
}

private void makeMove(byte newEmptyIndex) {
    board[emptyIndex] = board[newEmptyIndex];
    board[newEmptyIndex] = 0;
    emptyIndex = newEmptyIndex;
}

public void shuffle(int numberOfMoves) {
    Random randi = new Random();
    for (int i = 0; i < numberOfMoves; i++) {
        List<Byte> moves = getPossibleMoves();
        int index = randi.nextInt(moves.size());
        byte mov = moves.get(index);
        makeMove(mov);
    }
}

@Override
public List<GraphState> generateChildren() {
    List<GraphState> list = new LinkedList<GraphState>();
    Iterator<Byte> it = getPossibleMoves().listIterator();
    while (it.hasNext()) {
        SlidingPuzzle child = new SlidingPuzzle(this);
        child.makeMove(it.next());
        String moveName = "D";
        if (this.emptyIndex - 1 == child.emptyIndex)
            moveName = "L";
        else if (this.emptyIndex + 1 == child.emptyIndex)
            moveName = "R";
        else if (this.emptyIndex - n == child.emptyIndex)
            moveName = "U";
        child.setMoveName(moveName);
        list.add(child);
    }
    return list;
}

@Override
public int hashCode() {
    return Arrays.hashCode(board);
}

public String toString() {
    StringBuilder stringBuilder = new StringBuilder();  
    StringBuilder line = new StringBuilder();
    final int cellSize = 5;
    for (int i = 0; i < n * cellSize + 1; i++)
        line.append("-");
    return line.toString();
}
Example of an easy eight-puzzle

The program beneath solves the following puzzle: (0, 1, 5, 8, 2, 7, 4, 3, 6). The puzzle was created by shuffling (starting from a correctly arranged board) with one hundred random moves, but turns out to be an easy one to solve.
CHAPTER 3. SEARCHING GRAPHS

<table>
<thead>
<tr>
<th>8</th>
<th>2</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

SOLUTION:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
</table>

PATH LENGTH: 17
CLOSED STATES: 26
OPEN STATES: 17
DURATION TIME: 0 ms
Figure 3.2: Graph searched by SaC using “linear conflicts” heuristics for an eight-puzzle (0, 1, 5, 8, 2, 7, 4, 3, 6).
Heuristics for sliding puzzle

An insightful reading on how to construct heuristics in general can be found in (Hansson, Mayer and Yung, 1985). The authors analyze in particular the sliding puzzle problem and describe three heuristics for it: “Misplaced tiles”, “Manhattan”, “Manhattan + linear conflicts”, together with proofs of their admissibility. Below, we explain them in brief and show their implementations in SaC.

“Misplaced tiles”  This heuristics is a primitive one. It boils down to counting cells (tiles) that are in a wrong place, except for the empty cell. This can be regarded as a relaxed model of the sliding puzzle in which we ignore the actual rules and we are able to freely pick up cells and put them down into the right location, as if each such manipulation — “move” — was of cost 1. Clearly the maximum value of this heuristics is $n^2 - 1$, which can be a significant underestimation of the true minimum cost. Here is an implementation.

```java
public class HFunctionMisplacedTiles extends StateFunction {
    @Override
    public double calculate(State state) {
        SlidingPuzzle slidingPuzzle = (SlidingPuzzle) state;
        double h = 0.0;
        for (int i = 0; i < slidingPuzzle.board.length; i++) {
            if ((i != slidingPuzzle.emptyIndex) && (slidingPuzzle.board[i] != i))
                h += 1.0;
        }
        return h;
    }
}
```

“Manhattan”  This is a popularly known heuristics. It calculates the sum of cell distances, in Manhattan metrics, to their correct location (the distance for the empty cell is excluded). In terms of relaxed models, this can be regarded as a variant of the sliding puzzle in which we do not have to make moves using the empty cell, but rather each cell can move independently. While traveling the tiles may freely stack on one another.

```java
public class HFunctionManhattan extends StateFunction {
    @Override
    public double calculate(State state) {
        SlidingPuzzle slidingPuzzle = (SlidingPuzzle) state;
        double h = 0.0;
        for (int i = 0; i < slidingPuzzle.board.length; i++) {
            if (i != slidingPuzzle.emptyIndex)
                h += manhattan(slidingPuzzle, i);
        }
        return h;
    }

    protected int manhattan(SlidingPuzzle slidingPuzzle, int index) {
        int n = slidingPuzzle.n;
        int index = i;
        return h;
    }
```
**Manhattan + linear conflicts**  This is the most advanced heuristic of the three. Apart from the Manhattan distances summand it also includes the second summand being the number of so-called linear conflicts multiplied by two. What is a linear conflict? Imagine that a first row of a fifteen-puzzle is as follows: 1, 2, 3, 0. In this row the tiles are not in their correct locations, the sum of Manhattan distances is 3, but there is no linear conflict, since by shifting the 0 element to the left three times the row gets correctly arranged. Now consider the following row: 2, 1, 3, 0. The sum of Manhattan distances is again 3, but there exist a linear conflict, since 1 is after 2. In consequence, one of these elements at some point during the actual solving will have to be brought down to the second row, and after some time, brought back up. That is why each linear conflict requires at least two additional moves. Linear conflicts should be detected both in rows and columns. Here is an implementation.

```java
public class HFunctionLinearConflicts extends HFunctionManhattan {
    @Override
    public double calculate(State state) {
        SlidingPuzzle slidingPuzzle = (SlidingPuzzle) state;
        return super.calculate(state) + linearConflicts(slidingPuzzle);
    }

    protected int linearConflicts(SlidingPuzzle slidingPuzzle) {
        int n = SlidingPuzzle.n;
        byte[] table = slidingPuzzle.board;
        int h = 0;
        int[] group = new int[n];
        int[] conflicts = new int[n];

        // rows
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++)
                group[j] = table[i * n + j];

            for (int j = 0; j < n - 1; j++) {
                if ((group[j] / n != i) && (group[j] > 0)) // is this row the goal row for group[j]
                    conflicts[j] = 0;
                else {
                    for (int k = j + 1; k < n; k++) {
                        if ((group[k] / n == i) && (group[k] > 0) && (group[j] > group[k]))
                            conflicts[j]++;
                    }
                }
            }
        }
    }
}
```
// while there remain some positive conflicts[j]
while (true) {
    int max = Integer.MIN_VALUE;
    int jMax = -1;
    for (int j = 0; j < n - 1; j++)
        if (conflicts[j] > max) {
            max = conflicts[j];
            jMax = j;
        }
    if (max <= 0)
        break;
    conflicts[jMax] = 0;
    for (int k = jMax + 1; k < n; k++)
        if ((group[k] / n == i) && (group[jMax] > group[k])) {
            h += 2.0;
            conflicts[k]--;
        }
}

// columns
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
        group[j] = table[j * n + i];
    for (int j = 0; j < n - 1; j++)
        if ((group[j] % n != i) && (group[j] > 0)) // is this column the goal row for
            conflicts[j] = 0;
        else {
            for (int k = j + 1; k < n; k++)
                if ((group[k] % n == i) && (group[k] > 0) && (group[j] > group[k]))
                    conflicts[j]++;
        }
}

// while there remain some positive conflicts[j]
while (true) {
    int max = Integer.MIN_VALUE;
    int jMax = -1;
    for (int j = 0; j < n - 1; j++)
        if (conflicts[j] > max) {
            max = conflicts[j];
            jMax = j;
        }
    if (max <= 0)
        break;
    conflicts[jMax] = 0;
    for (int k = jMax + 1; k < n; k++)
        if ((group[k] / n == i) && (group[jMax] > group[k])) {
            h += 2.0;
            conflicts[k]--;
        }
}

return h;
Comparison of heuristics — an experiment

In SaC, one can quite conveniently carry out a comparison of heuristics for the same fixed problem, iterating over them in a loop. Here is a sample code.

```java
StateFunction[] heuristics = {new HFunctionMisplacedTiles(), new HFunctionManhattan(), new HFunctionLinearConflicts();

for (StateFunction h : heuristics) {
    SlidingPuzzle.setHFunction(h);
    GraphSearchAlgorithm algorithm = new AStar(slidingPuzzle); // slidingPuzzle object defined earlier
    algorithm.execute();
    SlidingPuzzle solution = (SlidingPuzzle) algorithm.getSolutions().get(0);
    // printing information to the console
}
```

Beneath, we show two outputs of such comparisons. The first is for the original example with \( n = 3 \) — puzzle \((0,1,5,8,2,7,4,3,6)\), and the second is for the \( n = 4 \) case — puzzle \((0,1,7,10,6,5,3,2,12,4,14,11,9,13,8,15)\).

**SLIDING PUZZLE TO SOLVE:**
----------------------
| 0 | 1 | 5 |
|----------------------
| 4 | 3 | 6 |

***
HEURISTICS: sac.examples.slidingpuzzle.HFunctionMisplacedTiles
PATH LENGTH: 17
CLOSED STATES: 440
OPEN STATES: 268
DURATION TIME: 30 ms
***

HEURISTICS: sac.examples.slidingpuzzle.HFunctionManhattan
PATH LENGTH: 17
CLOSED STATES: 38
OPEN STATES: 26
DURATION TIME: 0 ms
***

HEURISTICS: sac.examples.slidingpuzzle.HFunctionLinearConflicts
PATH LENGTH: 17
CLOSED STATES: 26
OPEN STATES: 17
DURATION TIME: 0 ms

**SLIDING PUZZLE TO SOLVE:**
-----------------------------
| 0 | 1 | 7 | 10 |
Obviously, regardless of the heuristics applied we obtain solution paths of the same length. But, one can clearly note that better heuristics, i.e. more tight lower bounds on the true cost, lead to smaller graphs being searched — fewer closed and open states. The graphs from both examples are depicted in figures 3.3 (tree-like layout) and 3.4 (star-like layout).
Figure 3.3: Graphs searched by SaC using different heuristics for the same initial eight-puzzle: (0, 1, 5, 8, 2, 7, 4, 3, 6).
"Manhattan" (4229 states in total)

"Manhattan + linear conflicts" (1861 states in total)

Figure 3.4: Graphs searched by SaC using different heuristics for the same initial fifteen-puzzle: (0, 1, 7, 10, 6, 5, 3, 2, 12, 4, 14, 11, 9, 13, 8, 15). “Misplaced tiles” heuristics was omitted due to too large graph. Visualizations were rendered by the neato layouter in Graphviz.
Some harder fifteen-puzzles

In this section we show some results obtained using SaC for harder sliding puzzle examples. These examples are taken as a short excerpt from the work (Hansson et al., 1985). We put away the “Misplaced tiles” heuristics from considerations as being practically useless. Experiments are carried out using two algorithms A* and IDA*.

In the table below presented are five selected instances of a fifteen-puzzle. The numbering shown in the left-most column corresponds to the numbering in (Hansson et al., 1985). The instances are ordered top down from the easiest one (i.e. requiring the least of time to be solved) towards the hardest one. Before executing the search we explicitly imposed the RAM memory limit on the JVM to be 2 GB. This was meant to demonstrate that only two first of the examples allow us to solve them by the A* algorithm being more memory consuming than IDA*, which does not keep record of closed set. We should mention that we have used default SaC configuration settings which make the Open set to be implemented by sac.graph.OpenSetAsPriorityQueueFastContainsFastReplace and the closed set by sac.graph.ClosedSetAsHashMap — both fastest but the most memory consuming structures. Obviously the duration time of A* for the examples it managed to solve was shorter than in the case of IDA*.

<table>
<thead>
<tr>
<th>no.</th>
<th>initial state</th>
<th>path length</th>
<th>IDA* closed</th>
<th>IDA* time [s]</th>
<th>A* closed and open</th>
<th>A* time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>85</td>
<td>4,7,13,10,1,2,9,6,12,8,14,5,3,0,11,15</td>
<td>44</td>
<td>1.5 \times 10^7</td>
<td>46.7</td>
<td>1.7 \times 10^5, 1.6 \times 10^5</td>
<td>2.8</td>
</tr>
<tr>
<td>5</td>
<td>4,7,14,13,10,3,9,12,11,5,6,15,1,2,8,0</td>
<td>56</td>
<td>2.6 \times 10^7</td>
<td>60.5</td>
<td>1.6 \times 10^6, 1.4 \times 10^6</td>
<td>34.4</td>
</tr>
<tr>
<td>2</td>
<td>13,5,4,10,9,12,8,14,2,3,7,1,0,15,11,6</td>
<td>55</td>
<td>3.8 \times 10^7</td>
<td>85.6</td>
<td>2.6 \times 10^6, 2.1 \times 10^6</td>
<td>77.1</td>
</tr>
<tr>
<td>54</td>
<td>12,11,0,8,10,2,13,15,5,4,7,3,6,9,14,1</td>
<td>56</td>
<td>1.9 \times 10^8</td>
<td>442.5</td>
<td>out of RAM (2 GB) at: 2.7 \times 10^6, 2.3 \times 10^6</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>14,13,15,7,11,12,9,5,6,0,2,1,4,8,10,3</td>
<td>57</td>
<td>2.5 \times 10^8</td>
<td>634.8</td>
<td>out of RAM (2 GB) at: 2.3 \times 10^6, 2.0 \times 10^6</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 3.3: Performance of A* and IDA* algorithms for five selected instances of fifteen-puzzle taken from (Hansson et al., 1985).

Sliding puzzle console solver

Along with the distribution of the SaC library comes a console solver dedicated for the sliding puzzle. The solver can be accessed by the included run_slidingpuzzle.bat file (or directly by the sac.examples.slidingpuzzle.ConsoleSolver class). A default execution triggered via

```java
java -Xmx2048M -cp "sac-1.0.0.jar;jfreechart-1.0.14.jar;jcommon-1.0.17.jar" sac.examples.slidingpuzzle.ConsoleSolver
```

(the line from the .bat file) produces the following output to the screen with help information and a default eight puzzle solved:

```
SLIDING PUZZLE SOLVER
PARAMETERS:
```
CHAPTER 3. SEARCHING GRAPHS

- sp - input path to text file (one line, comma-separated) with sliding puzzle to be solved ('0' assumed as an empty tile)
- a - full class name of graph search algorithm to be used (default: sac.graph.AStar)
- h - full class name of heuristic function to be used (default: sac.examples.slidingpuzzle.HFunctionLinearConflicts)
- c - input path to .properties file with configuration settings for search process
- g - output path to .dot file in Graphviz format, representing graph that was searched
- gWithContent - true/false flag stating if points in Graphviz graph should be drawn with a content or no

DEFAULT SLIDING PUZZLE: '0,3,2,4,7,8,1,5,6'.

SLIDING PUZZLE TO SOLVE:

```
| 0 | 3 | 2 |
| 4 | 7 | 8 |
| 1 | 5 | 6 |
```

ALGORITHM: sac.graph.AStar.
HEURISTICS: sac.examples.slidingpuzzle.HFunctionLinearConflicts.

SOLVING...

DURATION TIME: 16 ms.
CLOSED STATES: 45.
OPEN STATES: 31.
SOLUTION:
```
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |
```

PATH LENGTH (INCLUDING TERMINAL STATES): 17.
ALL DONE.

3.3.2 Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is one of the most recognizable problems within computer science. The problem can be formulated as follows: Given is a set of places with connections (roads) between them. A salesman departs from one of the places (distinguished as the starting point), must visit all the places and return to the place of origin. The salesman is allowed to visit each place only once. The goal is to find the shortest route for the salesman. In graph terms, the TSP can be reformulated as the problem of finding a Hamiltonian cycle with the smallest cost for the given graph.

In general, the TSP is NP-hard, although some of its large instances have been successfully solved (e.g. a TSP tour covering all 24,978 towns in Sweden\(^8\)). One may note that for \( n \) places and a complete graph of connections, a naive approach checking exhaustively all possible routes must perform \((n - 1)!\) iterations, which is exponential with respect to \( n \). It should also be noted that a greedy approach, consisting in visiting the nearest place at every step, leads in general to non-optimal solutions. In particular, for TSPs with places distributed randomly and uniformly on a plane (Euclidean TSP) the greedy approach returns on average routes longer by approximately 25% than an optimal route (Johnson and McGeoch, 1997). On the other hand, it is also possible to construct such ‘malicious’ examples for which the greedy approach leads to the worst routes (Gutin, Yeo and Zverovich, 2002).

The exemplary implementation of a TSP solver in SaC, described in the following section, is based on a well known and simple heuristics using the concept of a minimum spanning tree (MST). For a survey on more advanced and recent methods related to the TSP the reader is addressed e.g. to the following works: (Rego, Gamboa, Glover and Osterman, 2011; Kaplan, Shafrir, Lewenstein\(^8\)).

\(^8\)The project was carried out by a research team directed by David Applegate from AT&T Labs, [http://www.math.uwaterloo.ca/tsp/sweden/](http://www.math.uwaterloo.ca/tsp/sweden/)
CHAPTER 3. SEARCHING GRAPHS

and Sviridenko, 2003).

Implementation of TSP state

Our solver is restricted to TSPs related to symmetric graphs with complete sets of connections. Before presenting the actual state implementation, we first show some code excerpts of auxiliary base classes, namely: sac.examples.tsp.Place, sac.examples.tsp.Connection and sac.examples.tsp.Map. For full versions the reader is addressed to source codes of the library.

A place is represented by its $x$, $y$ coordinates and an id number (for convenience). It also contains a set of references to the connections outgoing to other places. In the excerpt below we omit getters and setters (marked by ...).

```java
public class Place implements Comparable<Place> {
    private int id;
    private double x;
    private double y;
    private SortedSet<Connection> connections = null;

    public Place(int id, double x, double y) {
        this.id = id;
        this.x = x;
        this.y = y;
        connections = new TreeSet<Connection>();
    }

    @Override
    public int compareTo(Place otherPlace) {
        return id - otherPlace.id;
    }

    @Override
    public boolean equals(Object otherPlace) {
        Place otherPlace2 = (Place) otherPlace;
        return (compareTo(otherPlace2) == 0);
    }

    ...
}
```

A connection — an edge in our graph — is defined by a pair of places and is equipped with its cost precalculated prior to the instantiation. We keep the two references to places always sorted according to their ids. Again, in the excerpt we omit getters and setters.

```java
public class Connection implements Comparable<Connection> {
    private Place place1 = null;
    private Place place2 = null;
    private double cost;

    public Connection(Place place1, Place place2, double cost) {
        if (place1.compareTo(place2) < 0) {
            this.place1 = place1;
            this.place2 = place2;
```
A map object represents the world (the graph) for our salesman. It stores references to places and connections in data structures convenient for the future search. One of the class constructors creates a set of \( n \) places, distributed randomly within a unit square, together with all their connections. The place with id 1 is treated as the starting point. In the code excerpt below we omit: getters, setters, a constructor and auxiliary methods for populating the map with a specific TSP from a text file, and the code related to drawing operations (visualization purposes).
private void calculateCosts() {
    int n = places.size();
    for (int i = 1; i <= n; i++)
        for (int j = i + 1; j <= n; j++) {
            Place place1 = places.get(i);
            Place place2 = places.get(j);
            double cost = Math.sqrt((place1.getX() - place2.getX()) * (place1.getX() - place2.getX())
            + (place1.getY() - place2.getY()) * (place1.getY() - place2.getY()));
            Connection connection = new Connection(place1, place2, cost);
            connections.add(connection);
            place1.getConnections().add(connection);
            place2.getConnections().add(connection);
        }
    }
}

@Override
public String toString() {
    StringBuilder result = new StringBuilder(" ");
    int i = 0;
    for (java.util.Map.Entry<Integer, Place> entry : places.entrySet()) {
        Integer id = entry.getKey();
        Place place = entry.getValue();
        result.append(id + ":( " + place.getX() + "," + place.getY() + ") ");
        if (((++i) < places.size())
            result.append("\n");
    }
    return result.toString();
}

...
We leave some of the comments present in the code, shown for clarification of some variables or operations.

```java
public class TravelingSalesmanProblem extends GraphStateImpl {

    /**
     * Reference to the map, common for all states.
     */
    public static Map map = null;

    private List<Connection> route = null;
    private List<Integer> routeAsIds = null;
    private Place currentPlace = null;
    private SortedSet<Place> remaining = null;

    /**
     * Minimum spanning tree built on remaining places.
     */
    private MinimumSpanningTree mst = null;

    /**
     * Best (cheapest) connection from the current place to some place in the minimum spanning tree of remaining places.
     */
    private Connection bestConnectionToMST = null;

    public TravelingSalesmanProblem(Map map) {
        TravelingSalesmanProblem.map = map;
        route = new ArrayList<Connection>();
        currentPlace = map.getStartPlace();
        routeAsIds = new ArrayList<Integer>();
        routeAsIds.add(currentPlace.getId());
        remaining = new TreeSet<Place>();
        remaining.addAll(map.getPlaces().values());
    }

    public TravelingSalesmanProblem(TravelingSalesmanProblem parent) {
        route = new ArrayList<Connection>();
        for (Connection connection : parent.route)
            route.add(connection);
        routeAsIds = new ArrayList<Integer>();
        for (Integer id : parent.routeAsIds)
            routeAsIds.add(id);
        currentPlace = parent.currentPlace;
        remaining = new TreeSet<Place>();
        remaining.addAll(parent.remaining);
    }

    @Override
    public List<GraphState> generateChildren() {
        List<GraphState> children = new LinkedList<GraphState>();
        for (Connection connection : currentPlace.getConnections()) {
            Place otherPlace = (connection.getPlace1().equals(currentPlace)) ? connection.getPlace2() : connection.getPlace1();
            if (remaining.contains(otherPlace)) {
                if ((otherPlace.equals(map.getStartPlace())) && (remaining.size() > 1))
                    continue;
                TravelingSalesmanProblem child = new TravelingSalesmanProblem(this);
                child.route.add(connection);
                child.routeAsIds.add(otherPlace.getId());
            }
        }
        return children;
    }
}
```
child.currentPlace = otherPlace;
child.remaining.remove(otherPlace);
child.setMoveName(String.valueOf(otherPlace.getId()));
children.add(child);
}
return children;

@Override
public boolean isSolution() {
    return remaining.isEmpty();
}

@Override
public String toString() {
    return routeAsIds.toString();
}

@Override
public int hashCode() {
    return routeAsIds.hashCode();
}

static {
    setHFunction(new StateFunction() {
        @Override
        public double calculate(State state) {
            TravelingSalesmanProblem tsp = (TravelingSalesmanProblem) state;
            // checking if parent’s MST can be used (specifically: if the place to
            // which parent’s bestConnectionToMST leads does not fork further)
            if (tsp.parent != null) {
                TravelingSalesmanProblem tspParent = (TravelingSalesmanProblem) tsp.parent;
                Connection singleConnection = tspParent.mst.isPlaceWithSingleConnection(tsp.
                                                                                     currentPlace);
                if (singleConnection != null) {
                    tsp.mst = new MinimumSpanningTree(tspParent.mst);
                    tsp.mst.getConnections().remove(singleConnection);
                    tsp.mst.setCost(tsp.mst.getCost() - singleConnection.getCost());
                    tsp.bestConnectionToMST = singleConnection;
                    return tsp.mst.getCost() + tsp.bestConnectionToMST.getCost();
                }
            }
            // construction of new MST required
            tsp.mst = new MinimumSpanningTree(tsp.remaining);
            tsp.bestConnectionToMST = null;
            double bestCost = Double.POSITIVE_INFINITY;
            for (Connection connection : tsp.currentPlace.getConnections()) {
                Place otherPlace = (connection.getPlace1() == tsp.currentPlace) ? connection.
                                                                                   getPlace2() : connection.getPlace1();
                if ((tsp.remaining.contains(otherPlace)) && (connection.getCost() < bestCost) ) {
                    bestCost = connection.getCost();
                    tsp.bestConnectionToMST = connection;
                }
            }
            double cost = tsp.mst.getCost();
            if (tsp.bestConnectionToMST != null)
                cost += bestCost;
            return cost;
        }
    });
}
As regards the fields in our TravelingSalesmanProblem class, it contains: a static reference to the map object, a collection of non-static fields describing the particular state, a minimum spanning tree object meant for the heuristics calculation purposes, and a separate reference to the best (cheapest) connection from the current place to some place within the MST.

On can notice that there is a certain redundancy in the description of the state, since we represent the route as a list of successive connections (List<Connection route) and simultaneously also as a list of ids of visited places (List<Integer> routeAsIds). Moreover, we keep a sorted set of remaining places (SortedSet<Place> remaining), which is also redundant, since such a set could be always uniquely derived from the route and the set of all places (kept inside the map). Nevertheless, such redundancy in terms of memory consumption is later compensated by speed gains in operations like: descendants generations, construction of MST, or identification of a state via toString() or hashCode() methods.

As regards the generation of descendants — the generateChildren() method — it boils down to: copying the parent state, iterating over connections outgoing from the current place and appending a new place to the route provided that this place has not been already visited. After that, the reference to the current place is set to the new place.

As for the cost functions g and h, we implement both of them as anonymous functions and attach them statically to the class via setGFunction(...) and setHFunction(...) respectively. The g function is straightforward and boils down to adding up the cost of the last connection to the g cost of the parent. The h function is a bit more complex. First of all we should remark that the MST reference in a descendant state is null at the start, and the calculation of MST is postponed until the first call to the getH() method is made. Then, we check whether the MST of the parent can be fairly cheaply reused or if a new MST should be derived. In doing so, we simply check whether the last place in the route so far (current place in our descendant) has only one edge within the parent’s MST. In other words we need to know if there is no fork from that place within the MST. If there is no such fork, we can reuse this MST and modify it only slightly i.e. the single edge is deleted from the new MST and it (this edge) becomes the best connection from the current state to the modified MST. Otherwise, if a fork exists, we rebuild the MST from the scratch.

The implementation of the MST related class (sac.examples.tsp.MinimumSpanningTree) applies Kruskal’s algorithm (Kruskal, 1956) and is included in Appendix 6.5.

111Hence, underneath it is also the first call to the hFunction.calculate() method.
Examples of small and easy TSPs

The following code example generates and solves an easy Euclidean TSP with \( n = 5 \) places distributed randomly within a unit square.

```java
Map map = new Map(5);
TravelingSalesmanProblem tsp = new TravelingSalesmanProblem(map);
System.out.println("TSP TO SOLVE: \n" + map);
GraphSearchAlgorithm algorithm = new AStar(tsp);
algorithm.execute();

if (algorithm.getSolutions().isEmpty()) {
    System.out.println("NO SOLUTIONS FOUND.");
    System.out.println("BEST STATE SO FAR: ");
    System.out.println(algorithm.getBestSoFar());
} else {
    TravelingSalesmanProblem solution = (TravelingSalesmanProblem) algorithm.getSolutions().get(0);
    System.out.println("SOLUTION: "+ solution);
    System.out.println("PATH COST (LENGTH): "+ solution.getG());
}
```

The program outputs the following result to the console and the graph searched is depicted in Fig. 3.5. In the figure, every graph state contains a visualization of the partial route travelled so far (marked in black) and the minimum spanning tree (marked in gray) related to the calculation of heuristics. It is possible to note that the algorithm reaches in fact two solutions, because in a non-directed graph a reversal of an optimal route leads to an equivalent optimal route as well.

TSP TO SOLVE:
1: (0.12251284631678094, 0.380209913097437)
2: (0.5639280517556917, 0.5560186588425798)
3: (0.6089976670947047, 0.17022778070498457)
4: (0.7890152412835078, 0.9022516218739999)
5: (0.2753596011637117, 0.5962434229024552)
CLOSED STATES: 12
OPEN STATES: 12
DURATION TIME: 4 ms
SOLUTION: [1, 3, 2, 4, 5, 1]
PATH COST (LENGTH): 2.1937848431352442

In Fig. 3.6 we present three TSP examples for \( n = 5 \), \( n = 10 \), and \( n = 15 \). The figure shows the console output of a TSP solving program and the resulting routes. Search graphs related to these examples are shown on subsequent pages, in figures 3.7-3.9.
Figure 3.5: Graph searched by SaC using MST-based heuristics for a Euclidean Traveling Salesman Problem with $n = 5$ places distributed randomly within a unit square.
Example 1 \((n = 10)\)

TSP TO SOLVE:

1: \((0.12721669119887158, 0.37344395814939046)\)
2: \((0.121603630813373, 0.6769233524617521)\)
3: \((0.6967743394616167, 0.6538204515701251)\)
4: \((0.6567922366222397, 0.87314753237464)\)
5: \((0.837799934790782, 0.13271896685690432)\)
6: \((0.8792651552573485, 0.27602108108406176)\)
7: \((0.5705158967610642, 0.10468983588670522)\)
8: \((0.9198623796845153, 0.536743717725981)\)
9: \((0.9368773693192925245, 0.1636900529887899)\)
10: \((0.88809627387394, 0.912468246489994)\)

CLOSED STATES: 30
OPEN STATES: 107
DURATION TIME: 43 ms
SOLUTION: \([1, 9, 7, 5, 6, 8, 3, 10, 4, 2, 1]\)
PATH COST (LENGTH): 3.1485171705107526

Example 2 \((n = 15)\)

TSP TO SOLVE:

1: \((0.6465487963847636, 0.21821665354693065)\)
2: \((0.12410081933969885, 0.8205066551015285)\)
3: \((0.7636609394148224, 0.3769891636455803)\)
4: \((0.2196694214954909, 0.7210071301565283)\)
5: \((0.3581164219226575, 0.982959069895216)\)
6: \((0.1948154559287916, 0.32576292427763125)\)
7: \((0.9778397669379531, 0.971245640779875)\)
8: \((0.7873474438251418, 0.87126978719532189)\)
9: \((0.48764622796626023, 0.9568681493382324)\)
10: \((0.829342562969854255, 0.6286381616959926)\)
11: \((0.1794181666639963, 0.83981621810772)\)
12: \((0.1843725152726462, 0.9219106679999714)\)
13: \((0.7488761819366391, 0.5957264194749598)\)
14: \((0.2898782153839893, 0.9667669199121948)\)
15: \((0.528666683692958, 0.84729149324681696)\)

CLOSED STATES: 299
OPEN STATES: 1146
DURATION TIME: 200 ms
SOLUTION: \([1, 8, 15, 6, 10, 4, 2, 11, 12, 14, 5, 9, 7, 13, 3, 1]\)
PATH COST (LENGTH): 3.4907876232208066

Example 3 \((n = 20)\)

TSP TO SOLVE:

1: \((0.08186119552601445, 0.49089277702354817)\)
2: \((0.05155073744417993, 0.08268872589515641)\)
3: \((0.2852609246313431, 0.08012634694163899)\)
4: \((0.866223694909227, 0.5731671341375282)\)
5: \((0.5688821507850508, 0.684445021174342)\)
6: \((0.839879069360267, 0.139861137729462)\)
7: \((0.17436467726205545, 0.86464464254908844)\)
8: \((0.029867615369098, 0.08574765069812)\)
9: \((0.2893602559697667, 0.265953977656689)\)
10: \((0.9856471449994421, 0.896562129948886)\)
11: \((0.8228879882715065, 0.6699187892826845)\)
12: \((0.9391576678180453, 0.2955146353444262)\)
13: \((0.3174935387616885, 0.0985738515517892)\)
14: \((0.3963134217759185, 0.2217979615638157)\)
15: \((0.98609242582995, 0.51737923165219)\)
16: \((0.8743126353859299, 0.879976761997406)\)
17: \((0.321394892952197, 0.8778741549117)\)
18: \((0.224386263366414, 0.8179912632336629)\)
19: \((0.911127380899347, 0.952184778429923)\)
20: \((0.394877377734747)\)

CLOSED STATES: 148
OPEN STATES: 1751
DURATION TIME: 305 ms
SOLUTION: \([1, 8, 9, 6, 5, 4, 17, 29, 11, 16, 13, 7, 15, 19, 14, 3, 19, 8, 12, 2, 1]\)
PATH COST (LENGTH): 3.794877377734747

Figure 3.6: Several TSP examples solved by SaC.
Figure 3.7: Solution and search graph for the TSP problem from example 1 in Fig. 3.6
Figure 3.8: Solution and search graph for the TSP problem from example 2 in Fig. 3.6
Figure 3.9: Solution and search graph for the TSP problem from example 3 in Fig. 3.6.
Simple statistics for random Euclidean TSPs

Examples from the previous subsection were fairly simple. In fact, we selected them out because the $A^*$ algorithm generated quite small search graphs for them — the number of states present at the stop moment (in both Open and Closed sets) was smaller than 2,000 in all cases. These graphs were convenient for us for the purpose of visualization.

In this subsection we would like to give the reader a feeling on the actual difficulty that particular initial values of $n$ create for the search algorithm. For that purpose we arranged a batch experiment consisting of multiple random TSPs. 20 repetitions were done for each $n = 5, 10, 15, 20$. The code below represents this experiment. Additionally, it demonstrates the usage of sac.stats.Stats class, convenient for batch experiments of that type. Results produced by the presented program are gathered in Table 3.4.

More information on how to register statistics and monitor search procedures within SaC can be found in Chapter 5.

1 Stats stats = new Stats();
2 for (int n = 5; n <= 20; n += 5) {
3   for (int r = 0; r < 20; r++) {
4     Map map = new Map(n);
5     TravelingSalesmanProblem tsp = new TravelingSalesmanProblem(map);
6     GraphSearchAlgorithm algorithm = new AStar(tsp);
7     algorithm.execute();
8     stats.addEntries(algorithm, n, r);
9   }
10 }
11 }
12 for (int n = 5; n <= 20; n += 5) {
13   System.out.println("***");
14   System.out.println("n = " + n);
15   System.out.println("MEAN_CLOSED_STATES: " + stats.mean(StatsCategory.GRAPH_SEARCH_CLOSED_STATES.toString(), n, null));
16   System.out.println("MEAN_OPEN_STATES: " + stats.mean(StatsCategory.GRAPH_SEARCH_OPEN_STATES.toString(), n, null));
17   System.out.println("MEAN_DURATION_TIME: " + stats.mean(StatsCategory.GRAPH_SEARCH_DURATION_TIME.toString(), n, null));
18   System.out.println("MEAN_PATH_G: " + stats.mean(StatsCategory.GRAPH_SEARCH_PATH_G.toString(), n, null));
19 }

Table 3.4: Simple statistics observed in 20 repetitions for random Euclidean TSPs (within a unit square).

<table>
<thead>
<tr>
<th>n</th>
<th>repetitions</th>
<th>mean closed states</th>
<th>mean open states</th>
<th>mean duration time</th>
<th>mean path cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>20</td>
<td>18.7</td>
<td>16.4</td>
<td>3.1 ms</td>
<td>1.98</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>194.4</td>
<td>721.4</td>
<td>24.5 ms</td>
<td>2.79</td>
</tr>
<tr>
<td>15</td>
<td>20</td>
<td>3,893.7</td>
<td>24,756.0</td>
<td>472.6 ms</td>
<td>3.37</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>25,643.9</td>
<td>247,673.7</td>
<td>9,742.3 ms</td>
<td>3.91</td>
</tr>
</tbody>
</table>
TSP console solver

Along with the distribution of the SaC library comes a console solver dedicated for the Traveling Salesman Problem. The solver can be accessed by the included run_tsp.bat file (or directly by the sac.examples.tsp.ConsoleSolver class). A default execution triggered via

```
java -Xmx2048M -cp "sac-1.0.0.jar;jfreechart-1.0.14.jar;jcommon-1.0.17.jar" sac.examples.tsp.ConsoleSolver
```

(the line from the .bat file) generates a random TSP for \( n = 10 \) and produces the following output to the screen with help information available:

```
TRAVELING SALESMAN PROBLEM (TSP) SOLVER

PARAMETERS:
-tsp - input path to text file (places in successive lines written as: x, y) with a TSP to be solved
-a - full class name of graph search algorithm to be used (default: sac.graph.AStar)
-tspImage - output path to .gif file representing the TSP to be solved
-tspSolutionImage - output path to .gif file representing the solution of TSP
-c - input path to .properties file with configuration settings for search process
-g - output path to .dot file in Graphviz format representing graph that was searched
-gWithContent - true/false flag stating if points in Graphviz graph should be drawn with a content or no

PREPARING A RANDOM TSP
TSP TO SOLVE:

1: (0.198759212864678, 0.42856952786747304)
2: (0.8456388466645194, 0.84516997375973016)
3: (0.848110983862101, 0.8478836286901716)
4: (0.513848049526595, 0.8885151748281576)
5: (0.93389673286441, 0.8803986086584886)
6: (0.991278715326988, 0.551266244991797)
7: (0.588183939471224, 0.4997779862892434)
8: (0.8615968949459115, 0.4739872794866)
9: (0.456822385653671, 0.873989658612715)
10: (0.9215007375726422, 0.84934196019090054)

ALGORITHM: sac.graph.AStar
SOLVING...
DURATION TIME: 110 ns.
CLOSED STATES: 222.
OPEN STATES: 717.
SOLUTION:
[1, 8, 2, 4, 5, 10, 6, 3, 9, 7, 1]
PATH COST (LENGTH): 2.8254791884954213
ALL DONE.
```

A specific (non-random) TSP problem can be given as input by providing a text file via -tsp parameter. In such a file the coordinates of places should be specified in successive lines with \( x \) and \( y \) values separated by a comma. Note that the console solver rescales original coordinates to fit the places in the unit square.

### 3.3.3 Sudoku

Sudoku is a number-placement puzzle. In the most common setup, the sudoku board is a 9 × 9 grid of cells with distinguished 9 subsquares inside, each being a 3 × 3 grid. Initially, the board is partially filled with numbers. The goal of the solver is to fill in the empty cells in such a manner that all rows, all columns, and all subsquares contain all the numbers from the set \{1, 2, ..., 9\}. It is commonly agreed that a well-posed initial sudoku should lead to a unique solution. Sudoku can be regarded as a special case of a latin square\[12\] with an additional constraint imposed on the subsquares.

Fig. 3.10 shows an example of the sudoku puzzle.

---

\[12\]Latin square is a \( n \times n \) grid filled with \( n \) symbols. Each symbol must occur exactly once in each row and each column.
Sudoku was popularized by a Japanese company Nikoli in late 1980s. Some sources (Shortz, 2005; Wikipedia, 2014) suggest that the puzzle was originally and anonymously proposed by Howard Garns, a freelance puzzle designer from Indiana, and published first in 1979 in Dell Magazines under the name *Number Place*.

A generalized sudoku can be defined as follows. Given is an initial $n^2 \times n^2$ grid of cells, containing $n^2$ subsquares (each of size $n \times n$). The grid is partially filled with numbers. The goal is to fill the missing cells in such a manner that all rows, all columns, and all subsquares contain all the numbers from the set $\{1, 2, \ldots, n^2\}$.

**Implementation of sudoku state**

Below we show a sudoku implementation using SaC. The code is almost in its full, omitted are import statements, and the `toGraphvizLabel()` method. The sudoku board is represented as a two-dimensional array of bytes. The value of zero represents and empty cell to be filled in. The subsquare side length $n$ is denoted in the code by `n`, and the whole board side length $n^2$ is denoted by `N`. Each sudoku state `knows` its number of empty cells (`emptyCells`). It also keeps some auxiliary fields with information about: the sum of remaining possibilities over all cells, and the cell with the minimum number of remaining possibilities (this cell is used later on to generate descendants). The remaining possibilities for an $(i, j)$ cell are the ones that remain after excluding the numbers occurring in the $i$-th row, the $j$-th column, and the subsquare that $(i, j)$ cell belongs to.

```java
public class Sudoku extends GraphStateImpl {
    protected static byte n;
    protected static byte N; // n * n
    protected byte[][] board = null;

    protected int emptyCells; // number of empty cells
    protected int sumRemainingPossibilities; // sum of remaining possibilities in all cells.
    protected int minRemainingPossibilities; // minimum number of remaining possibilities in some cell
    protected int minI = -1; // row position of the cell with minimum number of possibilities
```
protected int minJ = -1; // column position of the cell with minimum number of possibilities
protected SortedSet<Byte> possibilities = null; // remaining possibilities in the 'minimum cell'

public Sudoku(Sudoku parent) {
    board = new byte[N][N];
    emptyCells = parent.emptyCells;
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            board[i][j] = parent.board[i][j];
        }
    possibilities = new TreeSet<Byte>();
}

public Sudoku(int n) {
    Sudoku.n = (byte) n;
    N = (byte) (n * n);
    board = new byte[N][N];
    emptyCells = N * N;
    possibilities = new TreeSet<Byte>();
}

public Sudoku(int n, String sudokuAsCommaSeparatedString) {
    Sudoku.n = (byte) n;
    N = (byte) (n * n);
    board = new byte[N][N];
    emptyCells = N * N;
    Stringtokenizer tokenizer = new StringTokenizer(sudokuAsCommaSeparatedString, ",");
    int z = 0;
    while (tokenizer.hasMoreElements()) {
        byte number = Byte.valueOf((String) tokenizer.nextElement());
        int i = z / N;
        int j = z % N;
        board[i][j] = number;
        if (number > 0) {
            emptyCells--;
        }
        z++;
    }
    possibilities = new TreeSet<Byte>();
}

@Override
public List<GraphState> generateChildren() {
    List<GraphState> children = new LinkedList<GraphState>();
    double theH = getH(); // call made in order to do pre-calculation for heuristics and to
    // discover minI, minJ
    // (if not present so far)
    if (minRemainingPossibilities == 0)
        return children; // discrepancy or solution
    if (theH > 0) {
        for (byte possibility : possibilities) {
            Sudoku child = new Sudoku(this);
            child.board[minI][minJ] = possibility;
            if (child.isAdmissible()) {
                child.emptyCells = emptyCells - 1;
                child.setMoveName("(" + (minI + 1) + "," + (minJ + 1) + "):=" + possibility);
                children.add(child);
            }
        }
    }
}
public boolean isSolution() {
    return (emptyCells == 0);
}

@Override
public int hashCode() {
    byte[] linearBoard = new byte[N * N];
    for (int i = 0; i < N; i++) {
        System.arraycopy(board[i], 0, linearBoard, i * N, N);
    }
    return Arrays.hashCode(linearBoard);
}

@Override
public String toString() {
    StringBuilder builder = new StringBuilder();
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (board[i][j] > 0)
                builder.append(board[i][j] + 1);
        }
        if (i < N - 1)
            builder.append("\n");
    }
    return builder.toString();
}

protected boolean isAdmissible() {
    List<Byte> groupUnderCheck = new ArrayList<Byte>();
    // checking each square
    for (int k = 0; k < N; k++) {
        int minI = n * (k % n);
        int minJ = n * (k / n);
        for (int i = minI; i < minI + n; i++)
            for (int j = minJ; j < minJ + n; j++)
                if (board[i][j] > 0)
                    groupUnderCheck.add(Byte.valueOf(board[i][j]));
        if (!isGroupAdmissible(groupUnderCheck))
            return false;
        groupUnderCheck.clear();
    }
    // checking each row
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (board[i][j] > 0)
                groupUnderCheck.add(Byte.valueOf(board[i][j]));
            if (!isGroupAdmissible(groupUnderCheck))
                return false;
        }
        groupUnderCheck.clear();
    }
}
CHAPTER 3. SEARCHING GRAPHS

133
134 // checking each column
135 for (int j = 0; j < N; j++) {
136   for (int i = 0; i < N; i++) {
137     if (board[i][j] > 0)
138       groupUnderCheck.add(Byte.valueOf(board[i][j]));
139     if (!isGroupAdmissible(groupUnderCheck))
140       return false;
141     groupUnderCheck.clear();
142   }
143 }
144 return true;
145 }
146
protected boolean isGroupAdmissible(List<Byte> group) {
147   if (group.size() == 0)
148     return true; // empty group is implied by all zeros in it
149   for (int i = 0; i < group.size() - 1; i++) {
150     byte tempByte = group.get(i).byteValue();
151     for (int j = i + 1; j < group.size(); j++) {
152       byte toBeCompared = group.get(j).byteValue();
153       if (tempByte == toBeCompared)
154         return false;
155     }
156   }
157   return true;
158 }
159
protected void precalculateForHeuristics() {
160   minRemainingPossibilities = Integer.MAX_VALUE;
161   sumRemainingPossibilities = 0;
162   for (int i = 0; i < N; i++)
163     for (int j = 0; j < N; j++) {
164       if (board[i][j] > 0)
165         continue;
166       SortedSet<Byte> remaining = remainingPossibilities(i, j);
167       int remainingSize = remaining.size();
168       sumRemainingPossibilities += remainingSize;
169       if (remainingSize < minRemainingPossibilities) {
170         minRemainingPossibilities = remainingSize;
171         minI = i;
172         minJ = j;
173       possibilities = remaining;
174       if (minRemainingPossibilities == 0)
175         return;
176     }
177   }
178 }
179
private SortedSet<Byte> remainingPossibilities(int i, int j) {
180   SortedSet<Byte> remaining = new TreeSet<Byte>();
181   for (int k = 1; k <= N; k++)
182     remaining.add((byte) k);
183   // removing from remaining numbers existing in i-th row and j-th column
184   for (int k = 0; k < N; k++) {
185     remaining.remove(board[i][k]);
186     remaining.remove(board[k][j]);
187   }
The implementation presented contains three constructors — a copying constructor, a constructor of an empty board (populated with zeros), and a constructor from a comma separated string with numbers. It also contains suitable routines like: `hashCode()`, `toString()`, `isSolution()`, `generateChildren()`.

As regards the generation of descendants, it is always carried out with respect to some cell containing the minimum number of remaining possibilities. If there is more than one such cell in the board, the most top-left one is used. Information about the ‘minimum cell’ (its position, the actual remaining possibilities, and their number) are precalculated in the private `precalculateForHeuristics()` method.

In our implementation a sudoku can have at most $n^2$ descendants if all \{1,2,...,n^2\} remaining possibilities are valid. We remark that our process of descendants generation is in fact a guessing process. Although we do not introduce immediate conflicts (duplications of some number in a row, a column, or a subsquare) by excluding the numbers present in the row, the column, and the subsquare the chosen ‘minimum cell’ belongs to, it is possible that such conflicts may occur later. In other words we make the search procedure follow the paths for all guesses that seem currently valid. We implement the `isAdmissible()` method to check if a sudoku is legal or not. If there exist a duplication this method returns a `false`. Each potential descendant is first checked using the `isAdmissible()` method before being added to the actual list of descendants. Therefore, it is possible that a sudoku state (reached within the search procedure) may have as few as zero descendants. This means that in fact we have reached a contradictory state and the search algorithm shall abandon the current search path and move to another one by polling the next state from the `Open` queue.

By default, our sudoku implementation is equipped with a heuristic function represented by the class `HFunctionSumRemainingPossibilities`. It is attached by the `setHFunction(…) static call`. This heuristics boils down to calculating the number of remaining possibilities in every cell of the board and returning the sum. In other words, we say that a sudoku is closer to the goal state the smaller that sum is. On the other hand, we shall show that it is also possible to successfully use a different and trivial heuristic function — `HFunctionEmptyCells` — yielding the number of empty cells in the board. In that case, we say a sudoku is closer to the goal state the fewer empty cells it contains. We should mention that since sudoku is a placement puzzle we do not have to
care about minimizing the path (the number of moves / manipulations leading to the solution). In fact, from the very start we know what number of cells must be populated. From the perspective of search algorithms, this means we do not have to take the travelled cost $g$ into account, and this type of puzzles is typically well tackled by Best-first search algorithms or Depth-first-search\textsuperscript{13}. Moreover, we do not have to worry about the admissibility of the heuristics (since the $g$ summand is neglected).

Below, we include the codes of the two heuristics described above.

```java
public class HFunctionSumRemainingPossibilities extends StateFunction {
    @Override
    public double calculate(State state) {
        Sudoku sudoku = (Sudoku) state;
        sudoku.precalculateForHeuristics();
        return ((sudoku.minRemainingPossibilities == 0) && (sudoku.emptyCells > 0)) ? Double.POSITIVE_INFINITY : sudoku.sumRemainingPossibilities;
    }
}
```

```java
public class HFunctionEmptyCells extends StateFunction {
    @Override
    public double calculate(State state) {
        Sudoku sudoku = (Sudoku) state;
        sudoku.precalculateForHeuristics();
        return ((sudoku.minRemainingPossibilities == 0) && (sudoku.emptyCells > 0)) ? Double.POSITIVE_INFINITY : sudoku.emptyCells;
    }
}
```

**Example of a ‘not so easy’ sudoku**

We start with an example of a seemingly easy sudoku. It contains 54 givens and therefore merely 27 empty cells (typical sudokus have fewer than 30 givens). In particular, the top three rows in the example are already filled in completely. Yet, our SaC solver (based on a Best-first search algorithm) follows several stray leads before finding the correct path leading to the solution. Our run took 70 ms. Beneath, we present the source code and the program output.

```
Sudoku sudoku = new Sudoku(3,
    "8,5,4,2,1,9,7,6,3, +
    "3,9,7,8,6,5,4,2,1, +
    "2,6,1,4,7,3,9,8,5, +
    "0,8,0,0,0,0,0,9,0, +
    "0,4,0,5,3,8,0,7,0, +
    "0,3,0,0,0,0,5,0, +
    "9,2,6,3,8,4,5,1,7, +
    "5,0,3,7,0,0,0,4,8, +
```

\textsuperscript{13}A* or Dijkstra's algorithm would in this case degenerate in a sense to Breadth-first search and would only slow down the performance.
CHAPTER 3. SEARCHING GRAPHS

10 "4,7,8,0,0,1,0,0,0");
11 System.out.println("SUDOKU TO SOLVE:\n" + sudoku);
12 GraphSearchAlgorithm algorithm = new BestFirstSearch(sudoku);
13 algorithm.execute();
14 System.out.println("SOLUTION:\n" + algorithm.getSolutions().get(0));
15 System.out.println("CLOSED STATES:");
16 System.out.println("OPEN STATES:");
17 System.out.println("DURATION TIME");

SUDOKU TO SOLVE:
8,5,4,2,1,9,7,6,3
3,9,7,8,6,5,4,2,1
2,6,1,4,7,3,9,8,5
0,8,0,0,0,0,9,0
0,4,0,5,3,8,0,7,0
0,3,0,0,0,0,5,0
9,2,6,3,8,4,5,1,7
5,0,3,7,0,0,0,4,8
4,7,8,0,0,1,0,0,0

SOLUTION:
8,5,4,2,1,9,7,6,3
3,9,7,8,6,5,4,2,1
2,6,1,4,7,3,9,8,5
7,8,5,1,2,6,3,9,4
6,4,9,5,3,8,1,7,2
1,3,2,9,4,7,8,5,6
9,2,6,3,8,4,5,1,7
5,1,3,7,9,2,6,4,8
4,7,8,6,5,1,2,3,9
---
CLOSED STATES: 54.
OPEN STATES: 5.
DURATION TIME 70 ms.

Fig. 3.11 shows the graph searched for the sudoku from our first example. The heuristic function used in the search was “sum of remaining possibilities” (HFunctionSumRemainingPossibilities class). As usual, we encourage the reader to zoom in the figure to see more details. One can note that the first two passes were driven by forced moves — there existed cells where exactly one remaining possibility was left. After that, the search procedure reached a state where the cell (4,3) had two remaining possibilities {2,5} and it was the ‘minimum cell’. Two descendant states with each of these two possibilities inserted to the board were generated and added to the Open queue. Interestingly, the descendant related to the first possibility was superficially more attractive since its heuristic score was \( h = 57 \), as opposed to \( h = 63 \) for the second possibility. In other words, placing the 2 in the cell (4,3) (first possibility) eliminated more remaining possibilities from the whole board than if the 5 was placed (second possibility). Therefore, the algorithm followed the first possibility in the first order. This later turned out to be leading to a contradiction \( (h = \infty) \). As one can note from the figure, there were more such forking points where the guesswork driven by our heuristics lead the procedure astray.

How would the second heuristic function — “empty cells” (HFunctionEmptyCells) — cope with the same example? Below, we first show the code excerpt demonstrating how one can
switch to another heuristics (the rest of the code is as before), then we show the program output.
Obviously, the procedure reaches the same solution but one can note that now a few more states
are visited. We remark that this fact should not be treated as a proof that the “empty cell” heuristics
is worse in general. Later, we shall show a counter example.

```
GraphSearchAlgorithm algorithm = new BestFirstSearch(sudoku);
Sudoku.setHFunction(new HFunctionEmptyCells());
algorithm.execute();
```

SUDOKU TO SOLVE:
8,5,4,2,1,9,7,6,3
3,9,7,8,6,5,4,2,1
2,6,1,4,7,3,9,8,5
0,8,0,0,0,0,0,9,0
0,4,0,5,3,8,0,7,0
0,3,0,0,0,0,0,0,5,0
9,2,6,3,8,4,5,1,7
5,0,3,7,0,0,0,4,8
4,7,8,0,0,1,0,0,0

SOLUTION:
8,5,4,2,1,9,7,6,3
3,9,7,8,6,5,4,2,1
2,6,1,4,7,3,9,8,5
7,8,5,1,2,6,3,9,4
6,4,9,5,3,8,1,7,2
1,3,2,9,4,7,8,5,6
9,2,6,3,8,4,5,1,7
5,1,3,7,9,2,6,4,8
4,7,8,6,5,1,2,3,9
---
CLOSED STATES: 65.
OPEN STATES: 6.
DURATION TIME 70 ms.
Figure 3.11: Graph searched by SaC using “sum of remaining possibilities” heuristics for a “not so easy sudoku”.
CHAPTER 3. SEARCHING GRAPHS

Easy sudoku

Let us now switch to a truly easy sudoku that one might have come across in a newspaper. It has 36 givens, so fewer than in the previous example, but the search procedure leads directly (along one path) to the solution — there are no false leads. Below, we show the output of the related program and the graph search in Fig. 3.12 using “sum of remaining possibilities” as heuristics. Not to consume too much space the figure was rotated by 90° because it represents a single path.

SUDOKU TO SOLVE:
0,2,0,8,1,0,7,4,0
7,0,0,0,0,3,1,0,0
0,0,0,0,2,8,0,5
0,0,9,0,4,0,8,7
4,0,0,2,0,8,0,0,3
1,6,9,0,3,9,2,0,0
3,0,2,7,0,0,0,6,0
0,0,5,6,0,0,8,0,8
0,7,6,0,5,1,6,9,0

SOLUTION:
5,2,3,8,1,6,7,4,9
7,8,4,5,9,3,1,2,6
6,9,1,4,7,2,8,3,5
2,3,9,1,4,5,6,8,7
4,5,7,2,6,8,9,1,3
1,6,8,9,3,7,2,5,4
3,4,2,7,8,9,5,6,1
9,1,5,6,2,4,3,7,8
8,7,6,3,5,1,4,9,2

CLOSED STATES: 46.
OPEN STATES: 0.
DURATION TIME 63 ms.

Figure 3.12: Graph searched by SaC using “sum of remaining possibilities” heuristics for an easy sudoku.

Now, we again try the second heuristics (“empty cells”) on the same initial sudoku. Below is the program output. As on can see the number of states residing in Closed and Open sets at the stoppage moment are exactly the same. Therefore, both heuristics perform equally for this particular example.

SOLUTION:
5,2,3,8,1,6,7,4,9
7,8,4,5,9,3,1,2,6
6,9,1,4,7,2,8,3,5
2,3,9,1,4,5,6,8,7
4,5,7,2,6,8,9,1,3
1,6,8,9,3,7,2,5,4
3,4,2,7,8,9,5,6,1
9,1,5,6,2,4,3,7,8
8,7,6,3,5,1,4,9,2
CHAPTER 3. SEARCHING GRAPHS

---
CLOSED STATES: 46.
OPEN STATES: 0.
DURATION TIME 62 ms.

‘Qassim Hamza’ sudoku

The following sudoku, known as ‘Qassim Hamza’, is regarded as a very hard for human. There are 22 givens arranged diagonally in subsquares to make things harder. The solving process requires several guesses to be made. Beneath, we show the outputs of two runs of our solver, respectively for “sum of remaining possibilities” and “empty cells” heuristics. Both executions last less than a second, but this time the trivial “empty cells” heuristics turns out to be cheaper. Moreover, the difference in the number of visited and generated states is significant. Fig. 3.13 demonstrates the smaller of the two graphs searched (this time we display the nodes with no contents).

SUDOKU TO SOLVE:

\[
\begin{array}{ccc}
0 & 0 & 0,7,0 & 0,8,0,0 \\
0,0,0,0,4,0,0,3,0 \\
0,0,0,0,0,0,9,0,0,1 \\
6,0,0,5,0,0,0,0,0 \\
0,1,0,0,3,0,0,4,0 \\
0,0,5,0,0,1,0,0,7 \\
5,0,0,2,0,0,6,0,0 \\
0,0,0,0,8,0,0,9,0 \\
0,0,7,0,0,0,0,0,2 \\
\end{array}
\]

HEURISTICS: sac.examples.sudoku.HFunctionSumRemainingPossibilities.
SOLUTION:

\[
\begin{array}{ccc}
3,2,9,7,1,6,8,5,4 \\
1,7,6,8,4,5,2,3,9 \\
4,5,8,3,2,9,7,6,1 \\
6,4,3,5,7,2,9,1,8 \\
7,1,2,9,3,8,5,4,6 \\
8,9,5,4,6,1,3,2,7 \\
5,8,1,2,9,4,6,7,3 \\
2,3,4,6,8,7,1,9,5 \\
9,6,7,1,5,3,4,8,2 \\
\end{array}
\]

CLOSED STATES: 5267.
OPEN STATES: 452.
DURATION TIME 593 ms.
---
HEURISTICS: sac.examples.sudoku.HFunctionEmptyCells.
SOLUTION:

\[
\begin{array}{ccc}
3,2,9,7,1,6,8,5,4 \\
1,7,6,8,4,5,2,3,9 \\
4,5,8,3,2,9,7,6,1 \\
6,4,3,5,7,2,9,1,8 \\
7,1,2,9,3,8,5,4,6 \\
8,9,5,4,6,1,3,2,7 \\
5,8,1,2,9,4,6,7,3 \\
2,3,4,6,8,7,1,9,5 \\
9,6,7,1,5,3,4,8,2 \\
\end{array}
\]

CLOSED STATES: 525.
OPEN STATES: 40.
DURATION TIME 171 ms.
Figure 3.13: Graph searched by SaC using “empty cells” heuristics for the ‘Qassim Hamza’ sudoku.
CHAPTER 3. SEARCHING GRAPHS

Minimum sudoku

The next puzzle is an example of a so called minimum sudoku. Minimum sudokus are such initial arrangements which lead to a unique (single) solution and the number of givens is minimal. Up to recently it was not known what is the size of minimum sudokus for the $9 \times 9$ board. In 2012 a group of researches from Dublin University College proved that 17 is the size\[14\]. In other words all $9 \times 9$ sudokus with 16 or fewer givens lead to more than one solution. For strictness we also remark that obviously not all sudokus with 17 givens are necessarily minimal.

Below, we show a particular minimum sudoku and results of two executions of our SaC solver using two heuristics. This time the results are comparable (with a slight advantage of the ‘sum of remaining possibilities’ function).

SUDOKU TO SOLVE:
0,0,0,0,0,2,7,5,0
0,1,8,0,9,0,0,0,0
0,0,0,0,0,0,0,0,0
4,9,0,0,0,0,0,0,0
0,3,0,0,0,0,0,0,8
0,0,0,7,0,0,2,0,0
0,0,0,0,3,0,0,0,9
7,0,0,0,0,0,0,0,0
5,0,0,0,0,0,0,0,0
---
HEURISTICS: sac.examples.sudoku.HFunctionSumRemainingPossibilities.
SOLUTION:
9,4,6,1,8,2,7,5,3
3,1,8,5,9,7,4,2,6
2,7,5,6,4,3,8,9,1
4,9,2,3,1,8,5,6,7
6,3,7,2,5,4,9,1,8
8,5,1,7,6,9,2,3,4
1,2,4,8,3,5,6,7,9
7,8,3,9,2,6,1,4,5
5,6,9,4,7,1,3,8,2
CLOSED STATES: 5348.
OPEN STATES: 717.
DURATION TIME 530 ms.

HEURISTICS: sac.examples.sudoku.HFunctionEmptyCells.
SOLUTION:
9,4,6,1,8,2,7,5,3
3,1,8,5,9,7,4,2,6
2,7,5,6,4,3,8,9,1
4,9,2,3,1,8,5,6,7
6,3,7,2,5,4,9,1,8
8,5,1,7,6,9,2,3,4
1,2,4,8,3,5,6,7,9
7,8,3,9,2,6,1,4,5
5,6,9,4,7,1,3,8,2
CLOSED STATES: 5430.
OPEN STATES: 723.
DURATION TIME 561 ms.

\[14\]See: https://en.wikipedia.org/wiki/Mathematics_of_Sudoku
Suppose we want to check that the given sudoku is in fact minimal. Up to now our search procedure was stopped at the moment it reached the first goal state. From the status of the Open set at the stoppage moment one can see there are still potential leads to be followed. Let us force SaC to pursue those leads. The code excerpt below shows how this can be achieved.

```java
GraphSearchConfigurator configurator = new GraphSearchConfigurator();
configurator.setWantedNumberOfSolutions(Integer.MAX_VALUE);

GraphSearchAlgorithm algorithm = new BestFirstSearch(sudoku, configurator);
algorithm.execute();
```

The crucial point is to explicitly instantiate a GraphSearchConfigurator object and to tell it we want to find all solutions via setWantedNumberOfSolutions(Integer.MAX_VALUE). In fact, setWantedNumberOfSolutions(2) would work too if we were only interested in checking the uniqueness of the solution.

The execution of our program leads now to the output shown below. It confirms the fact our particular sudoku was minimal — the number of found solutions is one and we can see that the Open set is now emptied (all valid leads are exhausted).

FOUND SOLUTIONS: 1.
CLOSED STATES: 9730.
OPEN STATES: 0.
DURATION TIME 764 ms.

Now, suppose we want to play a bit and we remove one given number (the left most given in the first row) from our initial board. Again, we run our solver and we tell it (by means of a configurator) to find the first three solutions. Here is the full code, slightly modified, and the program output.

```java
Sudoku sudoku = new Sudoku(3,
"0,0,0,0,0,0,7,5,0," +
"0,1,8,0,9,0,0,0,0," +
"0,0,0,0,0,0,0,0,0," +
"4,9,0,0,0,0,0,0,0," +
"0,3,0,0,0,0,0,0,8," +
"0,0,0,7,0,2,0,0,0," +
"0,0,0,0,3,0,0,0,9," +
"7,0,0,0,0,0,0,0,0," +
"5,0,0,0,0,0,0,8,0");
System.out.println("SUDOKU TO SOLVE: \n" + sudoku);

GraphSearchConfigurator configurator = new GraphSearchConfigurator();
configurator.setWantedNumberOfSolutions(3);

GraphSearchAlgorithm algorithm = new BestFirstSearch(sudoku, configurator);
algorithm.execute();
```

---

15If RAM allows and if the set of all solutions is smaller than Integer.MAX_VALUE.
System.out.println("FOUND SOLUTIONS: "+algorithm.getSolutions().size() + ".");
System.out.println("CLOSED STATES: "+algorithm.getClosedStatesCount() + ".");
System.out.println("OPEN STATES: "+algorithm.getOpenSet().size() + ".");
System.out.println("DURATION TIME: "+algorithm.getDurationTime() + "ms.");

int i = 0;
for (GraphState solution : algorithm.getSolutions()) {
    System.out.println("−−−");
    System.out.println("SOLUTION "+(++i) + ":");
    System.out.println(solution);
    System.out.println("−−−");
}

SUDOKU TO SOLVE:
0,0,0,0,0,0,7,5,0
0,1,8,0,9,0,0,0,0
0,0,0,0,0,9,0,8,0
4,9,0,0,0,0,0,0,0
0,3,0,0,0,0,0,8,0
0,0,6,7,0,2,0,0,0
0,0,0,3,0,0,0,0,9
7,0,0,0,0,0,0,0,0
5,0,0,0,0,0,8,0

FOUND SOLUTIONS: 3.
CLOSED STATES: 202.
OPEN STATES: 36.
DURATION TIME 109 ms.
---
SOLUTION 1:
9,6,4,8,2,3,7,5,1
2,1,8,5,9,7,6,5,3
3,7,5,6,4,1,8,9,2
4,9,2,3,6,8,5,1,7
1,3,7,4,5,2,9,6,8
8,5,6,7,1,9,2,4,3
6,8,1,2,3,5,4,7,9
7,4,9,1,8,6,3,2,5
5,2,3,9,7,4,1,8,6
---
SOLUTION 2:
9,6,4,8,2,3,7,5,1
2,1,8,5,9,7,6,5,3
3,7,5,6,4,1,8,9,2
4,9,2,3,6,8,5,1,7
1,3,7,4,5,2,9,6,8
8,5,6,7,1,9,2,4,3
6,8,1,2,3,5,4,7,9
7,4,3,9,8,6,1,2,5
5,2,9,1,7,4,3,8,6
---
SOLUTION 3:
9,6,4,8,2,3,7,5,1
2,1,8,5,9,7,6,4,3
3,7,5,6,4,1,8,9,2
4,9,2,3,6,8,5,1,7
1,3,7,4,5,2,9,6,8
8,5,6,7,1,9,2,4,3
6,8,1,2,3,5,4,7,9
7,4,3,9,8,6,1,2,5
5,2,9,1,7,4,3,8,6
---
5,2,9,1,7,4,3,8,6

We now know that our removal of one given caused that at least three solutions exist. How many solutions are there in total now? We run the code again changing `setWantedNumberOfSolutions(3)` to `setWantedNumberOfSolutions(Integer.MAX_VALUE)`. Below we show the program output. For shortness we do not display the particular solutions, just their multiplicity.

FOUND SOLUTIONS: 1414.
CLOSED STATES: 90594.
OPEN STATES: 0.
DURATION TIME 3619 ms.

The output may be a bit surprising — there are now 1414 solutions in total and our search algorithm had to visit 90594 states before stopping. The duration time was now noticeable — 3619 ms.

All $4 \times 4$ sudokus

We use the ideas from the previous example to discover all distinct sudokus for the $4 \times 4$ board. It is sufficient to create an empty $4 \times 4$ sudoku and to ask SaC for all the solutions. The program code and its output are as follows.

```java
1 Sudoku sudoku = new Sudoku(2);
2 System.out.println("SUDOKU TO SOLVE:
3   0,0,0,0
4   0,0,0,0
5   0,0,0,0
6   0,0,0,0
7 FOUND SOLUTIONS: 288.
8 CLOSED STATES: 2273.
9 OPEN STATES: 0.
10 DURATION TIME 188 ms.
```

It turns out there are 288 different $4 \times 4$ sudokus. Fig. 3.14 illustrates the graph involved in that search.

One might be tempted to do the same thing for a traditional $9 \times 9$ sudoku. Yet, the trial must be abandoned, as it has been demonstrated combinatorically that the number of different sudoku puzzles is approximately $6.7 \cdot 10^{21}$. Thus, it is impossible to collect them within RAM memory.
CHAPTER 3. SEARCHING GRAPHS

Figure 3.14: Graph searched by SaC in order to discover all 4 × 4 sudokus.

Sudoku console solver

Along with the distribution of the SaC library comes a console solver dedicated for the sudoku puzzle. The solver can be accessed by the included run_sudoku.bat file (or directly by the sac.examples.sudoku.ConsoleSolver class). A default execution triggered via

```java
java -Xmx2048M -cp "sac-1.0.0.jar;jfreechart-1.0.14.jar;jcommon-1.0.17.jar" sac.examples.sudoku.ConsoleSolver
```

(the line from the .bat file) produces the following output to the screen with help information and a default ‘Qassim Hamza’ sudoku solved:

```
SUDOKU SOLVER
---------------------------------------------------------------------------------------------------------------------------
PARAMETERS:
- s - input path to text file (one line, comma-separated) with sudoku to be solved (possible sudokus: 4x4, 9x9, 16x16, etc.)
- h - full class name of heuristic function be used (default: sac.examples.sudoku.HFunctionSumRemainingPossibilities)
- c - input path to .properties file with configuration settings for search process
- g - output path to .dot file in Graphviz format representing graph that was searched
- gWithContent - true/false flag stating if points in Graphviz graph should be drawn with a content or no
---------------------------------------------------------------------------------------------------------------------------
DEFAULT SUDOKU ‘QASSIM HAMZA’.
SUDOKU TO SOLVE:22
6,0,0,7,0,0,8,0,0
6,0,0,0,4,0,0,3,0
0,0,0,0,0,9,0,0,1
6,0,0,5,0,0,0,0,0
0,1,0,0,3,0,0,4,0
0,0,5,0,0,1,0,0,7
5,0,0,2,0,0,6,0,0
0,3,0,0,8,0,0,9,0
0,0,7,0,0,0,0,0,2
HEURISTICS: sac.examples.sudoku.HFunctionSumRemainingPossibilities.
SOLVING....
DURATION TIME 655 ms.
CLOSED STATES: 5267.
OPEN STATES: 452.
FOUND 1 SOLUTION(S):
---
SOLUTION 1:
3,2,9,7,1,6,8,5,4
1,7,6,8,4,5,2,3,9
4,1,3,2,9,7,6,1
6,4,1,5,2,9,1,8
7,1,2,9,3,8,5,4,6
8,9,5,6,1,3,2,7
3,8,1,2,9,4,6,7,3
2,3,4,6,8,7,1,9,5
9,6,7,1,5,3,4,8,2
ALL DONE.
```

As an additional example we would like to show here a 16 × 16 sudoku solved by the console solver. We prepared the string representing such a sudoku in a text file d:/sudoku16.txt and we passed the file path to the console solver via -s parameter. The following command:

```java
java -Xmx2048M -cp "sac-1.0.0.jar;jfreechart-1.0.14.jar;jcommon-1.0.17.jar" sac.examples.sudoku.ConsoleSolver -s d:/sudoku16.txt
```

produced the output beneath to the console. As one can note the search procedure took over 90s to finish.
chapter 3. searching graphs

sudoku solver

parameters:
-s - input path to text file (one line, comma-separated) with sudoku to be solved (possible sudokus: 4x4, 9x9, 16x16, etc.)
-h - full class name of heuristic function be used (default: sac.examples.sudoku.HFunctionSumRemainingPossibilities)
-c - input path to .properties file with configuration settings for search process
-g - output path to .dot file in Graphviz format representing graph that was searched
-gWithContent - true/false flag stating if points in Graphviz graph should be drawn with a content or no

sudoku to solve:

7,0,0,0,5,1,6,1,10,9,0,8,8,0,6
12,5,0,0,8,15,14,0,4,0,9,0,11,8,16,2
6,5,13,2,13,6,9,7,0,5,0,8,1,8,0,6,3
1,0,0,0,8,6,0,5,0,9,0,0,8,0,0,0,6
8,0,1,0,9,2,8,15,0,0,0,5,0,6,0,11
15,0,9,3,0,8,8,6,6,0,7,14,6,8,1,0
14,0,16,8,0,6,0,5,1,6,0,9,8,18,2,6,0
8,0,12,8,0,8,0,9,1,10,13,16,8,0,3
6,0,0,0,8,8,6,6,11,0,9,3,8,0,6
6,6,4,0,9,10,8,7,0,14,0,9,8,11,0
6,16,0,12,8,3,5,0,10,9,8,0,8,7,0,5,6
3,0,0,15,8,0,6,3,11,4,9,14,0,6,0,16
6,0,0,9,14,15,13,0,19,8,0,4,0,8,6
9,0,7,8,6,8,0,6,0,1,0,9,8,8
5,13,8,0,3,1,8,16,0,9,11,6,8,8,15,0
16,1,15,9,0,5,16,14,9,0,8,6,6,6,0
heuristics: sac.examples.sudoku.HFunctionSumRemainingPossibilities.
solving...
duration time 96879 ms.
closed states: 133585.
open states: 33769.
found 1 solution(s):

solution 1:

7,14,6,9,8,5,1,12,3,11,2,16,4,10,13,15
12,8,13,5,7,15,14,1,4,6,9,18,11,1,16,2
4,15,10,2,13,16,11,6,1,7,12,5,8,9,3,14
1,3,11,16,2,9,4,16,8,14,13,15,7,5,6,12
6,9,13,1,3,14,2,9,15,4,16,12,5,7,8,11
15,5,9,3,18,13,16,11,2,8,7,14,6,12,1,4
14,16,6,1,12,7,15,11,9,5,13,18,2,9,13
2,13,12,7,5,6,8,9,4,18,10,13,16,15,14,3
13,7,5,12,6,2,14,16,15,3,11,9,4,8,10
8,6,4,15,16,18,13,5,7,12,14,9,2,3,11,1
11,16,14,12,4,3,9,7,16,2,14,8,15,13,5,6
3,9,2,10,15,11,8,1,6,13,5,4,12,14,7,16
16,12,2,6,11,14,15,13,5,18,8,7,3,4,2,9
9,2,7,14,6,12,4,13,16,15,1,3,11,10,5
5,13,8,4,1,18,2,12,9,11,6,14,16,15,7
16,1,15,11,3,7,5,16,14,1,4,2,13,6,12,8
all done.
Chapter 4

Searching game trees

This chapter is devoted to the part of SaC library responsible for searching game trees. We begin the chapter with a section reminding several well known game search algorithms. The reader familiar with these algorithms can move over to subsequent sections. The second section discusses SaC’s API related to game tree searching. The last one presents two game-related examples in which we have applied SaC for demonstration, namely: checkers and Nim.

4.1 Algorithms

Algorithms for game trees are based on the concept of minimax. Historically, this concept is due to von Neumann who formulated and proved the minimax theorem (von Neumann, 1928; von Neumann and Morgenstern, 1944). The theorem itself has a bit different and more general setting than typically can be met in algorithms meant for games like for example chess. The theorem pertains to zero-sum two-person games, covers both cases where players make simultaneous or alternate moves, and implies the existence of an optimal, so called, mixed strategy for each player. If both players apply their optimal strategies the game will be driven to a saddle point (a.k.a. minimax point) resulting in a game value (minimax value) that none of the players can improve by changing his strategy. Putting it more simply, the concept of minimax can be explained as a decision rule which tells a player to minimize the maximum possible payoff for the other player.

SaC provides three well known algorithms dedicated for games where players make alternate moves. The algorithms are recursive and it is convenient to express each of them in a form of two twin procedures that call one another interchangeably. In subsequent three sections we present these algorithms. We try to give them in a compact form, limiting ourselves to the most important backbone. In later sections, we describe some possible refinements and tricks that the algorithms can be augmented with to boost up their performance.

4.1.1 Min-Max

The subsequent pseudocode shows the simplest algorithm for game trees — the Min-Max algorithm. It demonstrates the basic mechanics underlying game tree searching. In the code $s$ denotes
the current state (for which the current recursive call is made), \( d \) denotes the depth of \( s \), and \( D \) is the maximum depth limit.

 Alg. 8 Min-Max

1. **procedure** MMEvaluateMaxState\((s, d, D)\)
2. \( \text{if IsTerminal}(s, d, D) \text{ then return } h(s) \) \hspace{1cm} \( \triangleright h(s) — \text{heuristic evaluation of position} \)
3. \( v := -\infty \)
4. generate descendants \( \{t\} \) of \( s \)
5. for all \( t \) do
6. \( w := \text{MMEvaluateMinState}(t, d + \frac{1}{2}, D) \)
7. if \( s \) is the root state then memorize \( w \) as the score for \( s \rightarrow t \) move
8. \( v := \max\{v, w\} \)
9. return \( v \)

 Alg. 8 Min-Max

10. **procedure** MMEvaluateMinState\((s, d, D)\)
11. \( \text{if IsTerminal}(s, d, D) \text{ then return } h(s) \) \hspace{1cm} \( \triangleright h(s) — \text{heuristic evaluation of position} \)
12. \( v := \infty \)
13. generate descendants \( \{t\} \) of \( s \)
14. for all \( t \) do
15. \( w := \text{MMEvaluateMaxState}(t, d + \frac{1}{2}, D) \)
16. if \( s \) is the root state then memorize \( w \) as the score for \( s \rightarrow t \) move
17. \( v := \min\{v, w\} \)
18. return \( v \)

The outermost call of the recursion should be made with respect to an initial state from which we want to start the analysis, in particular this can be the initial position of the game, and that state is treated as the root of game tree.

In lines 7 and 16 the algorithm memorizes a score for some move. In an actual implementation such scores are typically memorized in a data structure that resides in the global scope outside the recursion (for example this could be a hash map).

We distinguished the condition checking if given state is a terminal as a separate method IsTerminal. It can be regarded as an additional routine method. We do not want to define it here explicitly as it may depend on the rules of the game. Most commonly though, in IsTerminal one has to check if either: maximum depth has been reached i.e. \( d = D \); or \( s \) is a win state, i.e. \( h(s) = \pm \infty \); or \( s \) is a non-win but terminal state due to some special rule, for example a stalemate or a perpetual check in chess.

We should remark that in some situations it may be reasonable to pursue consequences of a state deeper although it is a terminal according to the maximum depth reached \((d = D)\). For example, in chess or checkers when a series of captures by both players is involved, one should analyze the series to the very end and return its final evaluation. An algorithmic gadget for that purpose, called **Quiescence**, is described in further sections.

As regards the computational complexity, it is easy to note that it is exponential with respect to the imposed depth limit \( D \) — the search horizon. If the given game has a constant branching factor
b (or if it is possible to estimate that factor to be \( b \) on average) then the computational complexity of Min-Max is of \( O(b^D) \) order. In that sense the algorithm can be treated as an exhaustive search up to the imposed horizon. It does not discard any subtrees.

### 4.1.2 Alpha-beta pruning

There are several people regarded as independent and almost simultaneous discoverers of the alpha-beta pruning algorithm (a.k.a. alpha-beta cut-offs) around late 1950s and early 1960s. In particular, the discoverers or contributors were: Daniel J. Edwards, Allen Newell, Hebert A. Simon, John McCarthy, Arthur Samuel, Alexander Brudno; see e.g. (Edwards and Hart, 1963; Brudno, 1963; Newell and Simon, 1976). Later, Knuth and Moore (1975) refined the algorithm and gave its detailed analysis. Pearl (1982) proved its optimality.

The pseudocode of the \( \alpha-\beta \) pruning algorithm is presented below.

<table>
<thead>
<tr>
<th>Alg. 9 Alpha-beta pruning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: procedure alpha-beta-evaluate-max-state(s, d, D, ( \alpha ), ( \beta ))</td>
</tr>
<tr>
<td>2: if is-terminal(s, d, D) then return ( h(s) ) ( \triangleright ) ( h(s) ) — heuristic evaluation of position</td>
</tr>
<tr>
<td>3: generate descendents ( {t} ) of ( s )</td>
</tr>
<tr>
<td>4: for all ( t ) do</td>
</tr>
<tr>
<td>5: ( v := ) alpha-beta-evaluate-min-state(t, ( d + \frac{1}{2} ), D, ( \alpha ), ( \beta ))</td>
</tr>
<tr>
<td>6: if ( s ) is the root state then memorize ( v ) as the score for ( s \rightarrow t ) move</td>
</tr>
<tr>
<td>7: ( \alpha := \max{\alpha, v} )</td>
</tr>
<tr>
<td>8: if ( \alpha \geq \beta ) then return ( \alpha ) ( \triangleright ) cut off — no further child ( t ) will be checked</td>
</tr>
<tr>
<td>9: return ( \alpha )</td>
</tr>
<tr>
<td>10: procedure alpha-beta-evaluate-max-state(s, d, D, ( \alpha ), ( \beta ))</td>
</tr>
<tr>
<td>11: if is-terminal(s, d, D) then return ( h(s) ) ( \triangleright ) ( h(s) ) — heuristic evaluation of position</td>
</tr>
<tr>
<td>12: generate descendents ( {t} ) of ( s )</td>
</tr>
<tr>
<td>13: for all ( t ) do</td>
</tr>
<tr>
<td>14: ( v := ) alpha-beta-evaluate-max-state(t, ( d + \frac{1}{2} ), D, ( \alpha ), ( \beta ))</td>
</tr>
<tr>
<td>15: if ( s ) is the root state then memorize ( v ) as the score for ( s \rightarrow t ) move</td>
</tr>
<tr>
<td>16: ( \beta := \min{\beta, v} )</td>
</tr>
<tr>
<td>17: if ( \alpha \geq \beta ) then return ( \beta ) ( \triangleright ) cut off — no further child ( t ) will be checked</td>
</tr>
<tr>
<td>18: return ( \beta )</td>
</tr>
</tbody>
</table>

Alpha-beta pruning belongs to the class of “branch and bound” algorithms. It works by tracking two bounds on the game value along the recursion: a lower bound \( \alpha \) and an upper bound \( \beta \). At any given point of the recursion one can understand \( \alpha \) as the guaranteed payoff for the maximizing player and \( \beta \) as the guaranteed payoff for the minimizing player. The outermost call of the recursion is made with the settings \( \alpha = -\infty \) and \( \beta = \infty \), i.e. the most pessimistic values for the maximizing and the minimizing player respectively.

The legal situation for the bounds is that they remain satisfying the \( \alpha < \beta \) inequality. Clearly, \( \alpha > \beta \) is a contradiction. For example, a maximizing player cannot be guaranteed with a payoff of
5, while the minimizing player is simultaneously guaranteed a payoff of −2. Therefore, if at some stage of tree analysis one comes across a state for which the initial condition to run the subrecursion is \( \alpha \geq \beta \), then, it means there is no point to do so because the subtree rooted by that state will not affect the game value. All such subtrees can be discarded. Note that the equality \( \alpha = \beta \) is not a contradiction, but it also means that neither player can improve his score within a given subtree. In other words, entering subtrees with \( \alpha \geq \beta \) would not be an effect of optimal behaviour of either player. On the other hand, one must understand that \( \alpha-\beta \) is an exact algorithm, not an approximation. Its results are identical with results of Min-Max. It means that even though we do not analyze some subtrees we do not miss anything relevant in the game. If an actual game being played entered in its course of actions a discarded subtree that would mean one of the players made a mistake and the opponent can only gain from that — he can improve or sustain his result, not deteriorate it.

It can be shown that an optimistic complexity of the alpha-beta pruning is \( O(b^{1/2D}) \). It arises when the ordering of children nodes is optimal, meaning that the best moves are always analyzed in first order, which allows for early cut-offs. The pessimistic complexity is \( O(b^D) \) — the same as for Min-Max. With random ordering of moves one obtains an average complexity of \( O(b^{3/4D}) \)

It is also worth mentioning that the shown pseudocode represents the alpha-beta pruning algorithm in the so called fail-soft version. It means that outcomes returned from particular recursion calls do not have to fall into the starting \([\alpha, \beta] \) interval the call was initially made with. There also exist the fail-hard version, in which each outcome must be bounded to the initial \([\alpha, \beta] \) interval\(^1\). The difference between the two versions does not affect neither the final result of the outermost call nor the collection of memorized move scores. One can check that even if some subrecursion returns and outcome outside the initial \([\alpha, \beta] \) interval, then, at the recursive stage one level up, this outcome will be ignored because it does not improve the suitable \( \alpha \) or \( \beta \) payoff. On the other hand the fail-soft version is important because it helps to construct more advanced algorithms (like Scout or MTD-f) that are based on the notion of so called zero windows. We discuss that concept in the next section.

### 4.1.3 Scout

The main idea behind the Scout algorithm is driven from the following Knuth-Moore theorem.

**Theorem 1** (Knuth and Moore, 1975, “theorem about \( \alpha-\beta \) window”) Let \( v^* \) denote the true, exact game value returned by the Min-Max procedure executed with respect to some state. Let \( v \) denote the outcome of a fail-soft alpha-beta pruning procedure executed for that state and with a search window defined by \( \alpha \) and \( \beta \) numbers imposed arbitrarily. Then, the following three cases are possible.

1. \( \alpha < v < \beta \Rightarrow v = v^* \),
2. \( v \leq \alpha \Rightarrow v^* \leq v \),
3. \( \beta \leq v \Rightarrow v \leq v^* \).

\(^1\)In the fail-hard version, lines 8 and 17 would return: \( \beta \) and \( \alpha \) respectively.
The meaning of the theorem for the design of new search algorithm is the following. We may try to guess a more narrow $\alpha$-$\beta$ window than in the traditional pruning procedure. Note that the more narrow the window is the more cutoffs are likely to be produced within the recursion. If the outcome value $v$, being the consequence of our guess, falls inside the $(\alpha, \beta)$ interval then we are lucky, because $v$ is equal to the true game value $v^*$, and we can experience a computational gain (if more cutoffs occur). On the other hand if the outcome falls outside the interval, then one obtains a lower or an upper bound on true game value. Now, two cases are possible: either we have to repeat the execution with a wider window (and we experience a computational loss), or we can make use of the bound (and we experience a gain again). For example, if the maximizing player is guaranteed with a payoff $\alpha$ and we obtained an outcome $v < \alpha$, then, despite the fact that $v$ is not an exact game value, we do not have to repeat the computations because $v$ upperbounds $v^*$, and the computations (if repeated) would not improve the $\alpha$.

Historically, the first one to study this idea was Pearl (1980). He postulated that one can more cheaply test whether a payoff can be improved. He introduced two recursive procedures \texttt{test(...)} and \texttt{eval(...)}, where the first one returned a boolean value stating if an improvement was possible, while the other calculated the exact game value if needed.

The idea was then refined by Reinefeld (1983) who introduced the concept of zero search windows (a.k.a. null windows or scout windows). If the payoffs in a game are integer numbers then a zero window is the one satisfying the condition

$$\alpha + 1 = \beta. \quad (4.1)$$

We first give some initial definitions and remarks that help to understand the Scout algorithm. Then, we demonstrate the actual pseudocode (Alg. 10).

1. We shall say that an imposed $\alpha$-$\beta$ window succeeded if the value $v$ returned by the \texttt{fail-soft} procedure satisfies: $\alpha < v < \beta$. This implies that the true game value $v^*$ equals $v$.

2. We shall say that an imposed $\alpha$-$\beta$ window failed low if the value $v$ returned by the \texttt{fail-soft} procedure satisfies: $v \leq \alpha$. This implies that $v$ is an upper bound on the true game value, i.e.: $v^* \leq v$.

3. We shall say that an imposed $\alpha$-$\beta$ window failed high if the value $v$ returned by the \texttt{fail-soft} procedure satisfies: $\beta \leq v$. This implies that $v$ is a lower bound on the true game value, i.e.: $v \leq v^*$.

4. A zero window (with $\alpha + 1 = \beta$) must fail either low or high.

5. In the algorithm, only the first descendant of every state is analyzed with a full $\alpha$-$\beta$ window, the second and further descendants are analyzed with zero windows, i.e. $\alpha-(\alpha+1)$ or $(\beta+1)$-$\beta$, respectively for the states of type MAX and MIN.

6. If a zero window, imposed for a descendant of a MAX state, fails low then we do not care — the payoff of the maximizing player could not have been improved within that descendant’s subtree. This usually carries a computational gain because more cutoffs are likely to occur with a zero window imposed.
7. If a zero window, imposed for a descendant of a MAX state, fails high then we have to repeat
the search with respect to that descendant with a wider window: $v-\beta$, to obtain an exact value
for the subtree. The repetition carries a computational loss, but note that the new window
$v-\beta$ is still more narrow than a full initial $\alpha-\beta$ window.

8. The above two remarks are symmetrically opposite for the MIN states.

9. The algorithm contains a condition checking if the current state is two half-moves away from
the maximum depth or deeper ($D - d \leq 2 \cdot 1/2$). If so, the repetition of search is not needed
despite a fail-situation because it can be shown the algorithm works correctly at those depths.

10. Experimental studies, in particular from (Reinefeld, 1983), indicate that computational gains
consequencing from zero windows and thus more frequent cutoffs are typically greater than
computational losses consequencing from search repetitions.

11. Reinefeld (1983) experimented with random games imposing branching factors within
[20, 60] (similar to chess) and depths of {4, 5} half-moves. He experienced that his Scout
algorithm visited about 20% less of tree terminals than in the case of a traditional alpha-beta
pruning.
Alg. 10 Scout
1: \textbf{procedure} \textsc{ScoutEvaluateMaxState}(s, d, D, \alpha, \beta) \\
2: \hspace{1em} \textbf{if} IsTerminal(s, d, D) \textbf{then return} h(s) \hfill \triangleright h(s) \text{ — heuristic evaluation of position} \\
3: \hspace{1em} b := \beta \\
4: \hspace{1em} \text{generate descendents \{}t\\} of \ s \\
5: \hspace{1em} \textbf{for all} t \textbf{do} \\
6: \hspace{2em} v := \textsc{ScoutEvaluateMinState}(t, d + \frac{1}{2}, D, \alpha, b) \\
7: \hspace{2em} \textbf{if} t \text{ is not the first descendant and } D - d > 2 \cdot 1/2 \text{ and } b \leq v \textbf{then} \hfill \triangleright \text{failing high} \\
8: \hspace{2em} v := \textsc{ScoutEvaluateMinState}(t, d + \frac{1}{2}, D, v, \beta) \\
9: \hspace{2em} \textbf{if} s \text{ is the root state then memorize } v \text{ as the score for } s \rightarrow t \text{ move} \\
10: \hspace{2em} \alpha := \max\{\alpha, v\} \\
11: \hspace{2em} \textbf{if} \alpha \geq \beta \textbf{then return } \alpha \hfill \triangleright \text{cut off — no further child } t \text{ will be checked} \\
12: \hspace{2em} b := \alpha + 1 \hfill \triangleright \text{forcing a zero window} \\
13: \hspace{1em} \textbf{return} \alpha \\
14: \textbf{procedure} \textsc{ScoutEvaluateMinState}(s, d, D, \alpha, \beta) \\
15: \hspace{1em} \textbf{if} IsTerminal(s, d, D) \textbf{then return} h(s) \hfill \triangleright h(s) \text{ — heuristic evaluation of position} \\
16: \hspace{1em} a := \alpha \\
17: \hspace{1em} \text{generate descendents \{}t\\} of \ s \\
18: \hspace{1em} \textbf{for all} t \textbf{do} \\
19: \hspace{2em} v := \textsc{ScoutEvaluateMaxState}(t, d + \frac{1}{2}, D, a, \beta) \\
20: \hspace{2em} \textbf{if} t \text{ is not the first descendant and } D - d > 2 \cdot 1/2 \text{ and } v \leq a \textbf{then} \hfill \triangleright \text{failing low} \\
21: \hspace{2em} v := \textsc{ScoutEvaluateMaxState}(t, d + \frac{1}{2}, D, a, v) \\
22: \hspace{2em} \textbf{if} s \text{ is the root state then memorize } v \text{ as the score for } s \rightarrow t \text{ move} \\
23: \hspace{2em} \beta := \min\{\beta, v\} \\
24: \hspace{2em} \textbf{if} \alpha \geq \beta \textbf{then return } \beta \hfill \triangleright \text{cut off — no further child } t \text{ will be checked} \\
25: \hspace{2em} a := \beta - 1 \hfill \triangleright \text{forcing a zero window} \\
26: \hspace{1em} \textbf{return} \beta
CHAPTER 4. SEARCHING GAME TREES

4.2 API

4.2.1 Game state abstraction

In SaC’s API for searching games, the main ‘actor’ is the \texttt{sac.graph.GameState} interface being an extension of the \texttt{sac.State} discussed earlier in Chapter 2. Below, we present the code listing for the \texttt{sac.graph.GameState} interface and then some excerpts from its default implementation — \texttt{sac.graph.GameStateImpl}. For full code of the implementation the reader is addressed to Appendix 6.3.

```java
package sac.game;
import java.util.List;
import sac.State;

public interface GameState extends State {
    public static final double H_SMALLEST_INFINITY = 0.5 * Double.MAX_VALUE;

    @Override
    public GameState getParent();
    @Override
    public List<GameState> getChildren();
    @Override
    public List<GameState> getPath();
    @Override
    public List<GameState> generateChildren();

    public boolean isMaximizingTurnNow();
    public void setMaximizingTurnNow(boolean maximizingTurnNow);

    public boolean isWinTerminal();
    public boolean isNonWinTerminal();

    public List<String> getMovesAlongPrincipalVariation();

    public boolean isQuiet();
    public boolean isVisited();
    public void setVisited(boolean visited);
    public boolean isReadFromTranspositionTable();
    public void setReadFromTranspositionTable(boolean readFromTranspositionTable);
}
```

The first thing to note is that the \texttt{GameState} interface reformulates the signatures of four basic methods related to the parent – children binding and path tracking. The difference (of cosmetic nature) is in the types returned, which now become \texttt{GameState} or \texttt{List<GameState>}, as the resulting objects should be treated as game states (rather than general states).

Apart from the above, there are several new elements in the interface (with respect to its ancestor \texttt{sac.State}):

- a constant \texttt{H_SMALLEST_INFINITY} storing the smallest value of the heuristic function (position evaluation) that already represents a win for a player — all values larger than the constant also represent wins, but achieved faster (at smaller depths),

- a pair of methods to check or set whose turn it is now — \texttt{isMaximizingTurnNow()},

...
setMaximizingTurnNow(...),
- a method designed to indicate whether the current state is a win terminal state — isWinTerminal(),
- a method designed to indicate whether the current state is a terminal but non-win state due to some special rule of the game (e.g. perpetual check or a stalemate in chess) — isNonWinTerminal(),
- a method returning the list of move names along the principal variation from the current state downwards the tree — getMovesAlongPrincipalVariation(),
- a method designed to indicate whether the current state is a so called quiet state — isQuiet(),
- a pair of methods to check or set whether the given state was actually visited during the search procedure; it may be so that a state was generated but in fact not visited due to a cutoff — isVisited(), setVisited(...),
- a pair of methods to check or set whether the evaluation for given state was read from the transposition table — isReadFromTranspositionTable(), setReadFromTranspositionTable(...).

The last two pairs of methods are not recommended to be called explicitly by the user, they are meant for internal purposes of SaC’s API.

Below, we show some excerpts from the default game state implementation in SaC.

```java
public abstract class GameStateImpl extends StateImpl implements GameState {

    protected boolean maximizingTurnNow = true;
    protected List<String> movesAlongPrincipalVariation = null;
    protected boolean visited = false;
    protected boolean readFromTranspositionTable = false;

    ...

    @Override
    public final boolean isMaximizingTurnNow() {
        return maximizingTurnNow;
    }

    @Override
    public final voidsetMaximizingTurnNow(boolean maximizingTurnNow) {
        this.maximizingTurnNow = maximizingTurnNow;
    }

    @Override
    public boolean isQuiet() {
        return true; // default implementation
    }
}
```

\[\text{2} \]E.g. in chess or checkers typically a state is regarded as quiet if no immediate captures are possible. If a state is not quiet then it is advised it should be analyzed further (to have a good evaluation of its consequences) even though it may be a state at the maximum depth of deeper.
CHAPTER 4. SEARCHING GAME TREES

4.2.2 General (abstract) game search algorithm

In this section we discuss SaC’s internal mechanisms related to the general game searching procedure. This procedure is represented by an abstract class: `sac.game.GameSearchAlgorithm`. Obviously, the end user of SaC library does not have to be aware of core-level intricacies. Instead, in practice he can limit himself to instantiating a specific search algorithm (e.g.: `new MinMax()`, `new AlphaBetaPruning(...)` or `new Scout(...)`) and running it for his particular game state. Therefore, we recommend the reading of subsequent contents only to readers really interested in low level details of SaC, perhaps the readers intending to extend the library in the future with new algorithms or data structures.

The listing below demonstrates the most important parts of the `GameSearchAlgorithm` class. For clarity, some less interesting parts have been skipped. The full code is available in the Appendix 6.4. The class is designed to work as a general and common (model) procedure for searching games and provides a set of helper methods that the specific algorithms — the subclasses — can rely on.

```java
package sac.game;
...

public abstract class GameSearchAlgorithm extends SearchAlgorithm {
  protected GameState initial = null;
  protected GameState current = null;
  protected Map<String, Double> movesScores = null;
  protected TranspositionTable transpositionTable = null;
  protected RefutationTable refutationTable = null;
  protected GameSearchConfigurator configurator = null;
  protected int closedCount = 0; // number of closed states (= number of calls of methods
                                 // evaluateMaxState(), evaluateMinState) since last reset().
  protected double depthReached = 0.0; // maximum depth that was reached in the search (owing
                                        // to quiescence) since last reset().
```
protected boolean stopForced = false; // boolean flag stating if stop was forced (e.g. from an outer thread)

public GameSearchAlgorithm(GameState initial, GameSearchConfigurator configurator) {
...
}

@Override
public final void execute() {
reset();
startTime = System.currentTimeMillis();
doExecute();
endTime = System.currentTimeMillis();
}

protected void doExecute() {
Double gameValue = null;
if (initial.isMaximizingTurnNow())
gameValue = evaluateMaxState(initial, Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY, 0.0, configurator.getDepthLimit());
else
gameValue = evaluateMinState(initial, Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY, 0.0, configurator.getDepthLimit());
if (configurator.isTranspositionTableOn())
transpositionTable.putOrUpdate(initial, gameValue, Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY);
current = null;
}

protected final Double evaluateMaxState(GameState gameState, double alpha, double beta, double depth, double depthLimit) {
// time limit check
if ((stopForced) || (configurator.getTimeLimit() < Long.MAX_VALUE)) {
long currentTime = System.currentTimeMillis();
if (currentTime - startTime > configurator.getTimeLimit()) {
endTime = System.currentTimeMillis();
return null;
}
}
closedCount++;
current = gameState;
gameState.setVisited(true);
return doEvaluateMaxState(gameState, alpha, beta, depth, depthLimit);
}

protected final Double evaluateMinState(GameState gameState, double alpha, double beta, double depth, double depthLimit) {
// time limit check
if ((stopForced) || (configurator.getTimeLimit() < Long.MAX_VALUE)) {
long currentTime = System.currentTimeMillis();
if (currentTime - startTime > configurator.getTimeLimit()) {
endTime = System.currentTimeMillis();
return null;
}
}
closedCount++;
current = gameState;
gameState.setVisited(true);
return doEvaluateMinState(gameState, alpha, beta, depth, depthLimit);
}
public abstract Double doEvaluateMaxState(GameState gameState, double alpha, double beta, double depth, double depthLimit);

public abstract Double doEvaluateMinState(GameState gameState, double alpha, double beta, double depth, double depthLimit);

protected void reset() {
    ...
}

public Map<String, Double> getMovesScores() {
    return movesScores;
}

public final String getFirstBestMove() {
    ...
}

public final List<String> getBestMoves() {
    ...
}

... // getters, setters

protected final List<GameState> generateChildrenWrapper(GameState parent) {
    List<GameState> children = parent.generateChildren();
    for (GameState child : children) {
        child.setParent(parent);
        child.setDepth(parent.getDepth() + 0.5);
        if (configurator.isParentsMemorizingChildren())
            parent.getChildren().add(child);
        recalculateHIfLarge(child);}
    return children;
}

protected final static void recalculateHIfLarge(GameState state) {
    double h = state.getH();
    if (Math.abs(h) > GameState.MAX_H_VALUE) {
        h = Math.signum(h) * GameState.MAX_H_VALUE * (1.0 + 1.0 / state.getDepth());
        state.setH(h);
    }
}

public final boolean isGameStateTerminal(GameState gameState, double depth, double depthLimit) {
    depthReached = Math.max(depthReached, depth);
    if (((gameState.isWinTerminal()) || (gameState.isNonWinTerminal()))
        return true;
    if (depth > depthLimit)
        return (configurator.isQuiescenceOn()) ? gameState.isQuiet() : true;
    return false;
}

public final static boolean isExactGameValue(double childValue, double alpha, double beta) {
    return ((alpha < childValue) && (childValue < beta)) || ((childValue == Double.NEGATIVE_INFINITY) && (alpha == Double.NEGATIVE_INFINITY))
        || ((childValue == Double.POSITIVE_INFINITY) && (beta == Double.POSITIVE_INFINITY));
}
The main functional goal of a GameSearchAlgorithm object (regardless of its actual type) is to assign numeric scores to available moves in the current game position. The method exposed to the user for that purpose is the execute() method. Once it is finished, the move scores are memorized in a protected field: Map<String, Double> movesScores. It is a map type data structure storing (key, value) pairs, where the key (String) represents the name of a move and the value (Double) represents the heuristic evaluation for the move. Once the game tree analysis is finished the move scores are accessible to the user by one of the following methods: getMovesScores(), getFirstBestMove(), getBestMoves(). The first one is the most general getter returning all the scores. The second method returns only one name of a move which is the best or belongs to the group of several best moves (when equal highest scores occur). The third method returns a list of all best moves (again, when equal scores occur). It is worth to remind that in order for the scores map to be populated, the user must not forget to impose names on the descendant states, via the setMoveName(...) method, while they are being generated inside the generateChildren(). Both these methods come from the more general sac.State interface.

The next elements to note in the GameSearchAlgorithm class are references to transposition and refutation tables. These structures are discussed in detail in later sections, but it can be explained now, in short, that they allow for a quicker search either by avoiding multiple analysis of the same subtrees reached by different paths (transposition table), or by encouraging more cutoffs (refutation table).

As mentioned before the main operational method of the GameSearchAlgorithm class is the execute(). Underneath, the method calls a hierarchy of wrapping submethods, namely: it first calls the doExecute(), which in turn calls a suitable wrapper evaluateMaxState(...) or evaluateMinState(...) depending on whose turn it is now, then these wrappers call the actual evaluation methods — doEvaluateMaxState(...) or doEvaluateMinState(...). The wrapping mechanism allows to suitably handle the following elements:

- resetting initial state and some data structures before a new analysis of the tree,
- time measurements — execution and an early stoppage under time control,
- checking if an outer stop on the procedure was forced,
- counting and flagging the visited states.

One can note that the most low-level methods doEvaluateMaxState(...) and doEvaluateMinState(...)

```java
public final static void updateMovesAlongPrincipalVariation(GameState parent, GameState child) {
    List<String> movesAlongPrincipalVariation = parent.getMovesAlongPrincipalVariation();
    movesAlongPrincipalVariation.clear();
    movesAlongPrincipalVariation.add(child.getMoveName());
    movesAlongPrincipalVariation.addAll(child.getMovesAlongPrincipalVariation());
}

public final void forceStop() {
    stopForced = true;
}
```
are left as abstract and their implementations are supposed to be filled in by the specific search algorithms — subclasses of the GameSearchAlgorithm class.

We now comment on a few auxiliary methods of the GameSearchAlgorithm class that are used within search procedures.

The generateChildrenWrapper(...) is a method designed to be called from within the implementations of doEvaluateMaxState(...) or doEvaluateMinState(...) methods. That wrapper is meant to handle some operations accompanying the generation of descendants. Firstly, the wrapper invokes the generateChildren() call back method on the parent state. The implementation of that method is supposed to be provided by the user. Then, the wrapper iterates over the created descendants and for each of them it updates the depth and parent–child binding information. Finally, for each descendants its heuristic evaluation is updated, if it represents a win state, via the recalculateHIfLarge(...) method. The details of this operation are discussed in the next subsection.

The isGameStateTerminal(...) helper returns a boolean stating whether one of the following events occurred: the maximum depth was reached and the given state is quiet, the given state is a win terminal, the given state is a non-win terminal i.e. a terminal implied by some special rule of the game (e.g. perpetual check in chess).

The isExactGameValue(...) helper should be considered in the light of the Knuth-Moore theorem. The method returns the flag true if the tested game value falls inside the $\alpha - \beta$ window. Infinities are also suitably handled in the method. The method is useful in two places. Firstly, it is a condition when an actual algorithm is about to memorize a score for a move. If a game value is not exact, but rather a bound, then the algorithm should not memorize it as a score. Secondly, a similar condition occurs when a transposition table considers a state and its score to be stored.

The updateMovesAlongPrincipalVariation(...) forces an update of the principal variation, stored as a list of move names, for given parent object. It is done by using such a descendant of the parent that have just led to an improvement of the game value (or bound). This method is called in suitable places by MinMax, AlphaBetaBruning, and Scout classes inside their doEvaluateMaxState(...) and doEvaluateMinState(...) methods when the aforementioned improvement takes place.

### 4.2.3 Recalculation of heuristics for win states

The recalculateHIfLarge(...) method maps suitably the infinities (Double.POSITIVE_INFINITY or Double.NEGATIVE_INFINITY) to large values, taking into account the depth at which the winning position was achieved. This allows to ‘differentiate infinities’ in case several winning lines are possible. The algorithm assigns the highest score to the move which leads to a win in the fewest moves.

Let $h$ denote the heuristic evaluation assigned originally to a given state by the user and let $d$ be the depth of that state ($d \in \{0.0, 0.5, 1.0, 1.5, \ldots\}$). Also, let $h_{\infty}$ denote the $H_{\text{SMALLEST\_INFINITY}}$ constant (equal to $8.988465674311579E307$) from the GameState class, i.e. the smallest value representing a win. Then, provided that $h \geq h_{\infty}$ the recalculation of the evaluation is carried out
according to the following formula:

\[ h := \text{sgn}(h) \cdot h_{\infty} \cdot (1 + 1/d). \] (4.2)

The table 4.1 shows some recalculation results in the Double type (ignoring the sign) as the depth parameter grows. As one can see the successive values approach \( H_{\text{SMALLEST_INFINITY}} \) as \( d \to \infty \). It is worth to comment that the recalculation value for \( d = 0.0 \) was not reported in the table because such case does not occur in the typical usage of SaC — the analysis of a game tree is typically ordered for some state while the game is still ongoing, not finished. Therefore, the earliest moment a win can be reached is for \( d = 0.5 \). Yet, it can be checked programmistically that for \( d = 0.0 \) the formula (4.2) responds with \text{Double.POSITIVE_INFINITY}, so with the same result as for \( d = 0.5 \).

The recalculation mechanism is a facilitation for the user (the programmer of some game). He does not have to worry about ‘differentiating infinities’ himself. His heuristic evaluation function, attached via the \text{setHFunction(...)} functionality, is required only to return \text{Double.POSITIVE_INFINITY} or \text{Double.NEGATIVE_INFINITY} in case of a win and regardless of the depth. Those values shall later be recalculated automatically on the fly by SaC.

<table>
<thead>
<tr>
<th>( d )</th>
<th>recalculated unsigned ( h )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>\text{Double.POSITIVE_INFINITY}</td>
</tr>
<tr>
<td>1.0</td>
<td>1.7976931348623157E308</td>
</tr>
<tr>
<td>1.5</td>
<td>1.498077612385263E308</td>
</tr>
<tr>
<td>2.0</td>
<td>1.3482698511467367E308</td>
</tr>
<tr>
<td>2.5</td>
<td>1.2583851944036209E308</td>
</tr>
<tr>
<td>3.0</td>
<td>1.1984620899082103E308</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 4.1: Initial values of recalculated evaluations for win states depending on their depths.

### 4.2.4 Specific game search algorithms

In this section presented are code excerpts for the three game search algorithms included in SaC. Let us start with the MinMax class.

```java
package sac.game;
import java.util.List;
public class MinMax extends GameSearchAlgorithm {
    public MinMax(GameState initial, GameSearchConfigurator configurator) {
        super(initial, configurator);
    }
    public MinMax(GameState initial) {
        super(initial, null);
    }
}```
CHAPTER 4. SEARCHING GAME TREES

117

public MinMax() {
    super(null, null);
}

@Override
public Double doEvaluateMaxState(GameState gameState, double alpha, double beta, double depth , double depthLimit) {
    if (isGameStateTerminal(gameState, depth, depthLimit)) {
        if (configurator.isTranspositionTableOn())
            transpositionTable.putOrUpdate(gameState, gameState.getH(), alpha, beta);
        return gameState.getH();
    }
    List<GameState> children = generateChildrenWrapper(gameState);
    double value = Double.NEGATIVE_INFINITY;
    for (GameState child : children) {
        double childValue = null;
        if (configurator.isTranspositionTableOn())
            childValue = transpositionTable.get(child, alpha, beta);
        if (childValue == null) {
            childValue = (child.isMaximizingTurnNow()) ? evaluateMaxState(child, alpha, beta, depth + 0.5, depthLimit) : evaluateMinState(child, alpha, beta, depth + 0.5, depthLimit);
            if (childValue == null)
                return null;
            if (configurator.isTranspositionTableOn())
                transpositionTable.putOrUpdate(child, childValue, alpha, beta);
        }
        if (childValue > value) {
            value = childValue;
            updateMovesAlongPrincipalVariation(gameState, child);
        }
    }
    return value;
}

@Override
public Double doEvaluateMinState(GameState gameState, double alpha, double beta, double depth , double depthLimit) {
    if (isGameStateTerminal(gameState, depth, depthLimit)) {
        if (configurator.isTranspositionTableOn())
            transpositionTable.putOrUpdate(gameState, gameState.getH(), alpha, beta);
        return gameState.getH();
    }
    List<GameState> children = generateChildrenWrapper(gameState);
    double value = Double.POSITIVE_INFINITY;
    for (GameState child : children) {
        double childValue = null;
        if (configurator.isTranspositionTableOn())
            childValue = transpositionTable.get(child, alpha, beta);
        if (childValue == null) {
            childValue = (child.isMaximizingTurnNow()) ? evaluateMaxState(child, alpha, beta, depth + 0.5, depthLimit) : evaluateMinState(child, alpha, beta, depth + 0.5, depthLimit);
            if (childValue == null)
                return null;
            if (configurator.isTranspositionTableOn())
                transpositionTable.putOrUpdate(child, childValue, alpha, beta);
        }
    }
    return value;
}
transpositionTable.putOrUpdate(child, childValue, alpha, beta);

if (childValue < value) {
    value = childValue;
    updateMovesAlongPrincipalVariation(gameState, child);
}

if (depth == 0.0)
    movesScores.put(child.getMoveName(), childValue);
}

return value;
}
}

Apart from constructors there are only two methods — the implementations of doEvaluateMaxState(...) and doEvaluateMinState(...), that were undefined (abstract) in the general GameSearchAlgorithm class. As one can see these implementations correspond well to the algorithmic pseudocode 8 of Min-Max presented before. A notable deviation can be seen in the way the subrecursions are being called, namely, the doEvaluateMaxState(...) does not necessarily have to call the ‘opposite’ wrapper evaluateMinState(...) but may again call a wrapper to itself — evaluateMaxState(...) (and vice-versa). It all depends on whose turn it is to play and this condition is checked on every child state — child.isMaximizingTurnNow(). What is the point of this generalization? When can players deviate from moves being made interchangeably all the time? A good example, to answer that, are card games. Think of any card game where a player is allowed either to take a trick e.g. by playing a high card or to duck it (refuse it) by playing a small card, whatever he thinks is better for him. If he decides to take the trick then it is (commonly) his turn again to play on. This means that a sequence of two max operations (or two min operations) occurs. The algorithmic pseudocode 8 does not include such a scenario just for the sake of notational conciseness. Therefore, it is worth to remember that it is programmer’s responsibility to suitably change the turn flags when the children states are being generated — inside the generateChildren(...) implementation.

A certain extra in the presented code is also the usage of the transposition table. The usage is conditional, i.e. it takes place only when the suitable configuration option is on, and accounts for two cases: (1) a game value for a child already exists in the transposition table, so it can be read from there without executing the search recursion, (2) a game value for a child was just calculated and should be stored or updated in the transposition table. More technical details about the transposition table are described in the subsequent section 4.2.5. It can also be noticed that alpha and beta variables, passed as arguments, are never used. These variables are foreseen for the more advanced algorithms, which we now move to.

Let us have a look at the AlphaBetaPruning class.

```java
package sac.game;
import java.util.List;
public class AlphaBetaPruning extends GameSearchAlgorithm {
    public AlphaBetaPruning(GameState initial, GameSearchConfigurator configurator) {
        super(initial, configurator);
    }
```
public AlphaBetaPruning(GameState initial) {
    super(initial, null);
}

public AlphaBetaPruning() {
    super(null, null);
}

@Override
public Double doEvaluateMaxState(GameState gameState, double alpha, double beta, double depth, double depthLimit) {
    if (isGameStateTerminal(gameState, depth, depthLimit)) {
        if (configurator.isTranspositionTableOn())
            transpositionTable.putOrUpdate(gameState, gameState.getH(), alpha, beta);
        return gameState.getH();
    }
    List<GameState> children = generateChildrenWrapper(gameState);
    if (configurator.isRefutationTableOn())
        refutationTable.reorder(gameState, children);
    for (GameState child : children) {
        Double childValue = null;
        if (configurator.isTranspositionTableOn())
            childValue = transpositionTable.get(child, alpha, beta);
        if (childValue == null) {
            childValue = (child.isMaximizingTurnNow()) ? evaluateMaxState(child, alpha, beta, depth + 0.5, depthLimit) :
                evaluateMinState(child, alpha, beta, depth + 0.5, depthLimit);
            if (childValue == null)
                return null;
        }
        if (configurator.isTranspositionTableOn())
            transpositionTable.putOrUpdate(child, childValue, alpha, beta);
        if ((depth == 0.0) && (isExactGameValue(childValue, alpha, beta)))
            movesScores.put(child.getMoveName(), childValue);
        if (childValue > alpha) {
            alpha = childValue;
            updateMovesAlongPrincipalVariation(gameState, child);
        }
        if (configurator.isRefutationTableOn())
            refutationTable.put(gameState, child);
    }
    if (alpha >= beta)
        return alpha;
}

@Override
public Double doEvaluateMinState(GameState gameState, double alpha, double beta, double depth, double depthLimit) {
    if (isGameStateTerminal(gameState, depth, depthLimit)) {
        if (configurator.isTranspositionTableOn())
            transpositionTable.putOrUpdate(gameState, gameState.getH(), alpha, beta);
        return gameState.getH();
    }
    List<GameState> children = generateChildrenWrapper(gameState);
    if (configurator.isRefutationTableOn())
        refutationTable.reorder(gameState, children);
    for (GameState child : children) {
        Double childValue = null;
        if (configurator.isTranspositionTableOn())
            childValue = transpositionTable.get(child, alpha, beta);
        if (childValue == null) {
            childValue = (child.isMaximizingTurnNow()) ? evaluateMaxState(child, alpha, beta, depth + 0.5, depthLimit) :
                evaluateMinState(child, alpha, beta, depth + 0.5, depthLimit);
            if (childValue == null)
                return null;
        }
        if (configurator.isTranspositionTableOn())
            transpositionTable.putOrUpdate(child, childValue, alpha, beta);
        if ((depth == 0.0) && (isExactGameValue(childValue, alpha, beta)))
            movesScores.put(child.getMoveName(), childValue);
        if (childValue > alpha) {
            alpha = childValue;
            updateMovesAlongPrincipalVariation(gameState, child);
        }
        if (configurator.isRefutationTableOn())
            refutationTable.put(gameState, child);
    }
    if (alpha >= beta)
        return alpha;
}
This time, the alpha and beta variables are in use. They are updated if an evaluation returned by a child state leads to an improvement of the payoff for given player. In particular, the last ‘if’ statement — if (alpha >= beta) {...} — leads to a cut off of a subtree, in compliance with the algorithm 9. One more new element is a possible usage of the so-called refutation table. More details about it are given in section 4.2.5. Hereby, it suffices to say that the refutation table may potentially reorder the children states — refutationTable.reorder(...) — in such a way that the most promising child is considered as the first one in the loop, and by that it may lead to an early cut off. On the other hand, each time an actual cut off is met (alpha >= beta) the information about the parent–child pair causing that cut off is stored in the refutation table for a plausible usage in the future. It is also worth to explain that an application of the refutation table was not possible in the previous MinMax class because the MinMax algorithm always analyzes all the children states, it cannot induce a cut off.

Let us now move to the Scout class — representing the most advanced algorithm for game trees within SaC.

```
package sac.game;
import java.util.List;

public class Scout extends GameSearchAlgorithm {
    public Scout(GameState initial, GameSearchConfigurator configurator) {
        super(initial, configurator);
    }
```
public Scout(GameState initial) {
    super(initial, null);
}

public Scout() {
    super(null, null);
}

@Override
public Double doEvaluateMaxState(GameState gameState, double alpha, double beta, double depth,
        double depthLimit) {
    if (isGameStateTerminal(gameState, depth, depthLimit)) {
        if (configurator.isTranspositionTableOn())
            transpositionTable.putOrUpdate(gameState, gameState.getH(), alpha, beta);
        return gameState.getH();
    }
    List<GameState> children = generateChildrenWrapper(gameState);
    if (configurator.isRefutationTableOn())
        refutationTable.reorder(gameState, children);
    double b = beta;
    for (int i = 0; i < children.size(); i++) {
        GameState child = children.get(i);
        Double childValue = null;
        boolean researchNeeded = false;
        double bound = alpha;
        if (configurator.isTranspositionTableOn())
            childValue = transpositionTable.get(child, alpha, b);
        if (childValue == null) {
            if (child.isMaximizingTurnNow()) {
                // scout search with zero window
                childValue = evaluateMaxState(child, alpha, b, depth + 0.5, depthLimit);
                if (childValue == null)
                    return null; // time limit reached
                // checking if window fails high, if so, research with broader window
                if ((i > 0) && (b <= childValue) && (childValue < beta) && ((configurator.
                    isQuiescenceOn()) || (depthLimit - depth > 0.5))) {
                    researchNeeded = true;
                    bound = childValue;
                    childValue = evaluateMaxState(child, bound, beta, depth + 0.5, depthLimit );
                    if (childValue == null)
                        return null; // time limit reached
                }
            }
        } else {
            // scout search with zero window
            childValue = evaluateMinState(child, alpha, b, depth + 0.5, depthLimit);
            if (childValue == null)
                return null; // time limit reached
            // checking if window fails high, if so, research with broader window
            if ((i > 0) && (b <= childValue) && (childValue < beta) && ((configurator.
                isQuiescenceOn()) || (depthLimit - depth > 0.5))) {
                researchNeeded = true;
                bound = childValue;
                childValue = evaluateMinState(child, bound, beta, depth + 0.5, depthLimit );
                if (childValue == null)
                    return null; // time limit reached
            }
        }
    }
}
 CHAPTER 4. SEARCHING GAME TREES

65
66 if (configurator.isTranspositionTableOn()) {
67 if (!researchNeeded)
68 transpositionTable.putOrUpdate(child, childValue, alpha, b);
69 else
70 transpositionTable.putOrUpdate(child, childValue, bound, beta);
71 }
72 if ((depth == 0.0) && (isExactGameValue(childValue, alpha, beta)))
73 movesScores.put(child.getMoveName(), childValue);
74 if (childValue > alpha) {
75 alpha = childValue;
76 updateMovesAlongPrincipalVariation(gameState, child);
77 if (configurator.isRefutationTableOn())
78 refutationTable.put(gameState, child);
79 }
80 if (alpha >= beta)
81 return alpha;
82 if (Math.abs(alpha) < GameState.H_SMALLEST_INFINITY)
83 b = alpha + 1.0;
84 }
85
86 return alpha;
87 }
88
89 @Override
90 public Double doEvaluateMinState(GameState gameState, double alpha, double beta, double depth,
91 double depthLimit) {
92 if (isGameStateTerminal(gameState, depth, depthLimit)) {
93 if (configurator.isTranspositionTableOn())
94 transpositionTable.putOrUpdate(gameState, gameState.getH(), alpha, beta);
95 return gameState.getH();
96 } List<GameState> children = generateChildrenWrapper(gameState);
97 if (configurator.isRefutationTableOn())
98 refutationTable.reorder(gameState, children);
99 double a = alpha;
100 for (int i = 0; i < children.size(); i++) {
101 GameState child = children.get(i);
102 Double childValue = null;
103 boolean researchNeeded = false;
104 double bound = beta;
105 if (configurator.isTranspositionTableOn())
106 childValue = transpositionTable.get(child, a, beta);
107 if (childValue == null) {
108 if (child.isMaximizingTurnNow()) {
109 // scout search with zero window
110 childValue = evaluateMaxState(child, a, beta, depth + 0.5, depthLimit);
111 if (childValue == null)
112 return null; // time limit reached
113 // checking if window fails low, if so, research with broader window
114 if ((i > 0) && (childValue <= a) && (alpha < childValue) && ((configurator.
115 isQuiescenceOn()) || (depthLimit - depth > 0.5))) {
116 researchNeeded = true;
117 bound = childValue;
118 childValue = evaluateMaxState(child, alpha, bound, depth + 0.5,
119 depthLimit);
120 if (childValue == null)
121 return null; // time limit reached
122 } else {
123 // scout search with zero window

childValue = evaluateMinState(child, a, beta, depth + 0.5, depthLimit);
if (childValue == null)
    return null; // time limit reached
// checking if window fails low, if so, research with broader window
if ((i > 0) && (childValue <= a) && (alpha < childValue) &&
    ((configurator.isQuiescenceOn()) || (depthLimit - depth > 0.5))) {
    researchNeeded = true;
    bound = childValue;
    childValue = evaluateMinState(child, alpha, bound, depth + 0.5,
    depthLimit);
    if (childValue == null)
        return null; // time limit reached
}
if (configurator.isTranspositionTableOn()) {
    if (!researchNeeded)
        transpositionTable.putOrUpdate(child, childValue, a, beta);
    else
        transpositionTable.putOrUpdate(child, childValue, alpha, bound);
} if ((depth == 0.0) && (isExactGameValue(childValue, alpha, beta))
    movesScores.put(child.getMoveName(), childValue);
if (childValue < beta) {
    beta = childValue;
    updateMovesAlongPrincipalVariation(gameState, child);
    if (configurator.isRefutationTableOn())
        refutationTable.put(gameState, child);
}
if (alpha >= beta)
    return beta;
if (Math.abs(beta) < GameState.H_SMALLEST_INFINITY)
    a = beta - 1.0;
}
return beta;
}

The code corresponds well to the algorithm yet, it is quite space consuming. Some overhead is related to the checks if a repeated search has to be done due to the zero window failing high or low. Also, a slightly different information is being put to the transposition table depending on whether the evaluation came from the first search or the second search.

4.2.5 Transposition table

In the context of games, the name transposition historically comes from chess and denotes the possibility of reaching the same position (the same state) by different sequences of moves. If the in-depth recurrence for such a position has already been performed before, then one can save time by taking a ready-made result provided that it is memorized somewhere — in a transposition table. For example, note that in chess the state reached after the initial sequence e2-e4, e7-e5, Ng1-f3, Ng8-f6 has 4! = 24 transpositions, accounting for all possible moves permutations.

For fast performance, typically a transposition table is implemented as a hash map, so that one can look up its entries in constant time — O(1). Yet, it is possible to implement transposition
tables as binary search trees, less memory-consuming but leading to the logarithmic look-up time — $O(\log_2 n)$, $n$ being the number of stored entries.

The keys to a transposition table can be state identifiers or their hash codes. For example in chess, a key should take into account the positions of remaining pieces plus some additional information about: castling possibilities, en passant captures, repetitions of moves. Sometimes, apart from the main transposition table, one can use additional transposition tables as libraries of openings or endgames.

In SaC, the overall implementation of the transposition table mechanisms is constituted by the following five elements:

1. the `TranspositionTableKey` class — it uses the (identifier, depth) pair as the key to represent a state stored in the transposition table,

2. the `TranspositionTableEntry` class — it represents an entry in the table associated to some key; the entry stores three numbers related to the state evaluation: its lower bound, its exact value, its upper bound (any of them can also be a `null` value),

3. the `TranspositionTable` interface — it defines a general set of methods that all transposition table containers should provide,

4. the `TranspositionTableImpl` class being the default (but still abstract) implementation of the above interface,

5. the `TranspositionTableAsHashMap` or `TranspositionTableAsTreeMap` class — the ready to use extensions of the default implementation, with the actual transposition table stored as a particular data structure: a hash map or a tree map (red-black tree), respectively.

Provided that the transposition table is turned on, there are two main situations when search algorithms interact with it. The first situation is when an algorithm iterates over the generated descendants, and it checks for each of them if there is a ready evaluation stored in the transposition table — this is done via the `get(...)` method. If so, the algorithm uses such an evaluation and does not trigger the searching recurrence downward the tree (from that descendant on). The second situation is when an algorithm reaches a terminal state or finishes processing a descendant state, then, it is the right time to put the calculated evaluation into the transposition table — this is done via the `putOrUpdate(...)` method. Both mentioned situations can be observed in the listings of `AlphaBetaPruning` and `Scout` classes presented back in section 4.1. Below, we show a code excerpt from the `TranspositionTableImpl` class that demonstrates the mentioned `get(...)` and `putOrUpdate(...)` methods and we then comment on it.

```java
package sac.game;
import java.util.AbstractMap;
public abstract class TranspositionTableImpl implements TranspositionTable {
    protected AbstractMap<TranspositionTableKey, TranspositionTableEntry> map;
    protected int usesCount = 0;
```
@Override
public Double get(GameState gameState, double alpha, double beta) {
    Double valueOrBoundOrNull = doGet(gameState, alpha, beta);
    if (valueOrBoundOrNull != null) gameState.setReadFromTranspositionTable(true);
    return valueOrBoundOrNull;
}

protected Double doGet(GameState gameState, double alpha, double beta) {
    TranspositionTableEntry entry = map.get(new TranspositionTableKey(gameState));
    if (entry == null) return null;
    if (entry.getExactGameValue() != null) {
        usesCount++;
        return entry.getExactGameValue();
    } else if ((entry.getUpperBoundOnGameValue() != null) && (entry.getUpperBoundOnGameValue() < alpha)) {
        usesCount++;
        return entry.getUpperBoundOnGameValue();
    } else if ((entry.getLowerBoundOnGameValue() != null) && (beta <= entry.getLowerBoundOnGameValue())) {
        usesCount++;
        return entry.getLowerBoundOnGameValue();
    }

    return null;
}

@Override
public TranspositionTableEntry get(GameState gameState) {
    return doGet(gameState);
}

protected TranspositionTableEntry doGet(GameState gameState) {
    return map.get(new TranspositionTableKey(gameState));
}

@Override
public void putOrUpdate(GameState gameState, Double value, double alpha, double beta) {
    TranspositionTableKey key = new TranspositionTableKey(gameState);
    TranspositionTableEntry entry = map.get(key);
    if (entry == null) {
        // put
        if (GameSearchAlgorithm.isExactGameValue(value, alpha, beta))
            entry = new TranspositionTableEntry(null, new Double(value), null);
        else {
            if (value <= alpha) // alpha-beta window fails low - value is an upper bound
                entry = new TranspositionTableEntry(null, null, new Double(value));
            else // alpha-beta window fails high - value is a lower bound
                entry = new TranspositionTableEntry(new Double(value), null, null);
        }
        map.put(new TranspositionTableKey(gameState), entry);
    } else {
        // update
        if (GameSearchAlgorithm.isExactGameValue(value, alpha, beta)) {
            entry.setExactGameValue(value);
            entry.setUpperBound(value);
            entry.setLowerBound(value);
        } else {
            if (value <= alpha) // alpha-beta window fails low - value is an upper bound
                entry.setUpperBound(value);
            else // alpha-beta window fails high - value is a lower bound
                entry.setLowerBound(value);
        }
    }
}
entry.setLowerBoundOnGameValue(null);
entry.setUpperBoundOnGameValue(null);
}
else {
    if ((value <= alpha) && ((entry.getUpperBoundOnGameValue() == null) || (value <
        entry.getUpperBoundOnGameValue()))) {
        entry.setUpperBoundOnGameValue(value); // tighter upper bound
    }
    if ((entry.getLowerBoundOnGameValue() != null) && (entry.
        getUpperBoundOnGameValue() != null)
        && (entry.getLowerBoundOnGameValue().doubleValue() == entry.
            getUpperBoundOnGameValue().doubleValue()) {
        entry.setExactGameValue(new Double(entry.getLowerBoundOnGameValue().
            doubleValue()));
        entry.setLowerBoundOnGameValue(null);
        entry.setUpperBoundOnGameValue(null);
    }
}
} else if ((beta <= value) && ((entry.getLowerBoundOnGameValue() == null) || (value <
(entry.getLowerBoundOnGameValue()) < value)) {
    entry.setLowerBoundOnGameValue(value); // tighter lower bound
}
if ((entry.getLowerBoundOnGameValue() != null) && (entry.
    getUpperBoundOnGameValue() != null)
    && (entry.getLowerBoundOnGameValue().doubleValue() == entry.
        getUpperBoundOnGameValue().doubleValue()) {
    entry.setExactGameValue(new Double(entry.getLowerBoundOnGameValue().
        doubleValue()));
    entry.setLowerBoundOnGameValue(null);
    entry.setUpperBoundOnGameValue(null);
}

The presented excerpts take into account the Knuth-Moore theorem, cited in section 4.1.3, and operate accordingly.

Consider first the get(GameState gameState, double alpha, double beta) method. It starts by looking up a transposition table entry for given state and if such an entry exists it checks whether the entry contains an exact game value or a bound. The exact value is returned unconditionally. A bound is returned only if it is useful. More precisely, an upper bound can be taken advantage of if the current \( \alpha \) is greater or equal to the bound, because it means that the maximizing player cannot improve his payoff by following the given state. Conversely, a lower bound can be taken advantage of if the current \( \beta \) is lower or equal to the bound, because it means that the minimizing player cannot improve his payoff by following the given state. If no entry or a useful bound exists in the table, the null value is returned. We should remark that apart from the mentioned getter to which the current \( \alpha-\beta \) window is passed on, there also exist another getter — get(GameState gameState) — in the TranspositionTable interface, which is supposed to return the whole entry (of TranspositionTableEntry type) that is the triplet of values: a lower bound, an exact value, an upper bound. Yet, this is only an auxiliary getter, and all game searching algorithms in SaC use the former variant, always passing on the current \( \alpha \) and \( \beta \) values.

Consider now the putOrUpdate(...) method. In the case of ‘put’ (when no entry exists so far), the method creates an entry and suitably places the given evaluation as an exact value or a bound
depending on its relation with respect to the $\alpha$-$\beta$ window. The ‘update’ case is more expanded. When the new value can be immediately recognized as an exact value, because it falls strictly in between $\alpha$ and $\beta$, then it is simply stored that way (lines 67–70). Otherwise, the code tries to make an update of the suitable bound, but only in the case the new bound is tighter than the previous one (lines 71-89). In a certain peculiar case, it is possible that after a bound update is made, both upper and lower bounds meet (become equal) — then, the entry object is rearranged to represent the exact value from now on.

Please note that the default TranspositionTableImpl implementation is an abstract class as it does not specify the data structure underlying the transposition table:

```java
protected AbstractMap<TranspositionTableKey, TranspositionTableEntry> map;
```

The aforementioned extensions existing in SaC — TranspositionTableAsHashMap and TranspositionTableAsTreeMap — are very light classes, consisting solely of constructors that assign a suitable data structure to the map object, as shown below.

```java
package sac.game;
import java.util.HashMap;

public class TranspositionTableAsHashMap extends TranspositionTableImpl {
    public TranspositionTableAsHashMap() {
        this.map = new HashMap<TranspositionTableKey, TranspositionTableEntry>(512 * 1024, (float) 0.75);
    }
}
```

```java
package sac.game;
import java.util.TreeMap;

public class TranspositionTableAsTreeMap extends TranspositionTableImpl {
    public TranspositionTableAsTreeMap() {
        this.map = new TreeMap<TranspositionTableKey, TranspositionTableEntry>();
    }
}
```

4.2.6 Refutation table

Refutation table is a valuable addition in the setting of progressive search (a.k.a. iterative deepening) i.e. when multiple searches are executed successively for new positions arising along an ongoing game. It is a typical scenario under a main game loop. Then, it is possible that some information from the previous searches can be used with a benefit in a new search. More precisely, the purpose of a refutation table is to memorize the information about best continuations at shallow depths of the tree. By best continuations meant are such moves — (parent, child) pairs — that led to the best
payoff for given parent or even caused a cut-off in the previous search iteration (refutation moves or cutoff moves). This information can be taken advantage of in the future, i.e. it can be used to reorder children and try best continuations as first in the next iteration hoping to obtain the cut-offs sooner.

The scenario described above is often referred to as the principal variation search (PSV), although the notion of ‘principal variation’ itself has a slightly different meaning than the notions of ‘refutation’ or ‘refutation table’. Yet, the two are conceptually very close together (Marsland, 1983).

How large are refutation tables? Obviously, it depends on the game branching factor and on how deep do we want to track the refutation information. For a moment consider for simplicity a game with alternate moves and a constant branching factor $b$. After both players make their moves, the root for the next analysis shifts somewhere two plies down the tree to one of $b^2$ possible states. Each of those states has some best descendant underneath (possibly a cut-off causer). Therefore, there are at most $b^2$ entries be kept in the refutation table for that particular level. If the programmer wants to track things (refutations) deeper, then the following quantities of new entries occur (at most) for successive levels: $b^2$, $b^3$, $b^4$, …. By default, SaC stores the refutation information for 4 plies. It is also worth to add that for typical games there are much more entries in a transposition table than in a refutation table.

In SaC, a refutation table keeps two collections inside: a ‘save collection’ meant for saving entries usable by the next search iteration, and a ‘read collection’ meant for reading entries from the previous iteration. Once the current iteration is completed, a reference to the ‘read collection’ is replaced by the ‘save collection’, whereas the reference to the ‘save collection’ is emptified (becomes a null). These operations are performed in the reset() method (provided by SaC’s API) and a particular search algorithm is supposed to call this method in-between the iterations.

The overall SaC’s implementation of refutation tables is represented by the following three elements:

1. the RefutationTable interface — it defines a general set of methods that all refutation table containers should provide,

2. the RefutationTableImpl class being the default (but still abstract) implementation of the above interface,

3. the RefutationTableAsHashMap or RefutationTableAsTreeMap class — the ready to use extensions of the default implementation, with the actual refutation table stored as a particular data structure: a hashmap or a tree map (red-black tree), respectively.

Similarly as it was in the case for transposition tables, the realizations of a refutation table as a hashmap or a red-black tree leads to constant or logarithmic look-up times respectively.

Provided that the refutation table is turned on, there are two main situations when search algorithms interact with it. The first is when an algorithm tries to reorder the descendants — this is done via the reorder(...) method being called immediately after the descendants were generated. If no suitable entry is found the refutation table (or there is just one descendant) then no reordering takes place. The second situation is when an algorithm comes across a payoff

---

3 The ‘principal variation’ represents a complete sequence of best moves starting from the given root up to a given maximum depth that leads to the minimax value of the game.
improvement. Then, the search algorithm should register or update an entry in the refutation table — this is done via the put(...) method. Both mentioned situations can be observed in the listings of AlphaBetaPruning and Scout classes presented back in section 4.1. Below, shown are the most important excerpts from the default RefutationTableImpl class together with short full listings of RefutationTableAsHashMap or RefutationTableAsTreeMap classes.

```java
tenpackage sac.game;
import java.util.AbstractMap;
import java.util.List;
import sac.Identifier;

public abstract class RefutationTableImpl implements RefutationTable {
    protected final static double DEFAULT_DEPTH_LIMIT = 2.0;
    protected double depthLimit;
    protected int usesCount = 0;

    protected AbstractMap<Identifier, Identifier> tableToSave = null;
    protected AbstractMap<Identifier, Identifier> tableToRead = null;

    @Override
    public void put(GameState parent, GameState child) {
        double depth = parent.getDepth() - 0.5; // in progressive search, we assume that next
                                                // iteration (possibly reading from refutation
                                                // table) will start from level +0.5, so moves at
                                                // level 0.0 are not memorized in refutation
                                                // table
        if ((depth >= 0) && (depth <= depthLimit))
            tableToSave.put(parent.getIdentifier(), child.getIdentifier());
    }

    @Override
    public void reorder(GameState parent, List<GameState> children) {
        if ((children == null) || (children.size() <= 1))
            return; // no reorder done
        double depth = parent.getDepth();
        if (depth > depthLimit)
            return; // no reorder done
        Identifier bestChildIdentifier = tableToRead.get(parent.getIdentifier());
        if (bestChildIdentifier == null)
            return; // no reorder done
        GameState bestChild = null;
        int i = 0;
        for (GameState child : children) {
            if (child.getIdentifier().equals(bestChildIdentifier)) {
                bestChild = child;
                break;
            }
            i++;
        }
        if (i == 0)
            return; // no reorder done
        if (bestChild != null) {
```
package sac.game;
import java.util.HashMap;
import sac.Identifier;
public class RefutationTableAsHashMap extends RefutationTableImpl {
    public RefutationTableAsHashMap() {
        super();
        tableToSave = new HashMap<Identifier, Identifier>();
        tableToRead = new HashMap<Identifier, Identifier>();
    }
    public RefutationTableAsHashMap(double depthLimit) {
        super(depthLimit);
        tableToSave = new HashMap<Identifier, Identifier>();
        tableToRead = new HashMap<Identifier, Identifier>();
    }
    @Override
    public void reset() {
        tableToRead = tableToSave;
        tableToSave = new HashMap<Identifier, Identifier>();
    }
}

package sac.game;
import java.util.TreeMap;
import sac.Identifier;
public class RefutationTableAsTreeMap extends RefutationTableImpl {
    public RefutationTableAsTreeMap() {
        super();
        tableToSave = new TreeMap<Identifier, Identifier>();
        tableToRead = new TreeMap<Identifier, Identifier>();
    }
    public RefutationTableAsTreeMap(double depthLimit) {
        super(depthLimit);
    }
}
tableToSave = new TreeMap<Identifier, Identifier>();
tableToRead = new TreeMap<Identifier, Identifier>();

@Override
public void reset() {
tableToRead = tableToSave;
tableToSave = new TreeMap<Identifier, Identifier>();
}
4.2.7 Configuration options for searching games

On several occasions we have mentioned the usage of a GameSearchConfigurator object. Below, we present a brief code listing of this class, with all configuration options and their default values. We purposely leave the javadocs present in the listing to make the meaning of options more clear. Every configuration option can be accessed by a suitable getter or setter (that are skipped in the listing).

```java
public class GameSearchConfigurator {

    /**
     * Identifier type for states. By default: STRING.
     */
    private IdentifierType identifierType = IdentifierType.HASH_CODE;

    /**
     * Depth limit expressed in full moves (not plies). By default: 3.5 (i.e. 7 plies).
     */
    private double depthLimit = 3.5;

    /**
     * Is transposition table on. By default: true.
     */
    private boolean transpositionTableOn = true;

    /**
     * Class name for transposition table. By default: sac.game.TranspositionTableAsHashMap.
     */
    private String transpositionTableClassName = "sac.game.TranspositionTableAsHashMap";

    /**
     * Is quiescence on. By default: true.
     */
    private boolean quiescenceOn = true;

    /**
     * Is refutation table on. By default: true.
     */
    private boolean refutationTableOn = true;

    /**
     * Class name for refutation table. By default: sac.game.RefutationTableAsHashMap.
     */
    private String refutationTableClassName = "sac.game.RefutationTableAsHashMap";

    /**
     * Depth limit for refutation table. By default: 2.0 (RefutationTableImpl.DEFAULT_DEPTH_LIMIT).
     */
    private double refutationTableDepthLimit = RefutationTableImpl.DEFAULT_DEPTH_LIMIT;

    /**
     * Do parents memorize references to their children. Set to false for lower memory usage (WARNING: in that case drawing game tree via GraphViz is impossible). By default: false.
     */
    private boolean parentsMemorizingChildren = false;
}
```
/**
*/
private long timeLimit = Long.MAX_VALUE;

public GameSearchConfigurator() {
}

public GameSearchConfigurator(String propertiesFilePath) throws Exception {
    ...
}

... // getters, setters

A configurator can be instatiated either by a default constructor with no arguments —
GameSearchConfigurator(), or from a properties text file — GameSearchConfigurator(String
propertiesFilePath). The names of options in the properties file are uppercased versions of field
names, and with underscores separating successive words. Below, we show a possible exemplary
contents of a configuration .properties file with some non-default values.

#Example of game configurator settings
identifierType=STRING
depthLimit=3.5
transpositionTableOn=true
transpositionTableClassName=sac.game.TranspositionTableAsHashMap
quiescenceOn=true
refutationTableOn=true
refutationTableClassName=sac.game.RefutationTableAsHashMap
refutationTableDepthLimit=2.0
parentsMemorizingChildren=false
timeLimit=Long.MAX_VALUE
4.3 Examples

4.3.1 Checkers

The game of checkers as such does not require an introduction. As regards the computer programs playing checkers, one of the historically important ones was a program written by Samuel in early 1950s, see (Samuel, 1983). The program was capable of self training, equipped with a weighted evaluation function. The weights (parameters) of that function were exposed to updates and improvements as two instances of the program were playing many games against each other. The program evolved this way was able to beat quite advanced players, but not the best ones of that time.

The exemplary implementation of checkers under SaC is not meant to introduce a strong program, but simply to demonstrate the API guidelines needed to be fulfilled by a programmer to implement similar mind games using SaC. We start by showing the code excerpts from the sac.examples.checkers.Checkers class representing a checkers state (for full source code the reader is addressed to the library), we then show several experiments and illustrations related to checkers prepared using SaC.

It is worth to remark that the SaC distribution is equipped with a simple graphical interface designed for experimenting with checkers and with different search algorithms — the program is represented by the sac.examples.checkers.CheckersGame class and a screenshot of it is shown in Fig. 4.1. It allows for a ‘human vs computer’ play or ‘computer vs computer’, using different algorithms and configurations. Also, it logs many informations to the screen. The program can be triggered for example by the following line:

```java
java -Xmx2048M -cp "sac-1.0.8.jar;jfreechart-1.0.14.jar;jcommon-1.0.17.jar;swt.jar" sac.examples.checkers.CheckersGame
```
Figure 4.1: A screenshot from the exemplary checkers program distributed with SaC — sac.examples.checkers.CheckersGame.
Implementation of checkers state

Our SaC representation of a checkers state — the `sac.examples.checkers.Checkers` class — stores four lists of pieces locations accounting for: white pawns, white kings, black pawns, and black kings. The locations themselves are encapsulated in the `BoardLocation` class, representing a pair of two-dimensional coordinates. Besides, the state is equipped with necessary SaC routines for: (1) identification of states — by either the `toString()` or the `hashCode()` method, (2) generation of descendants — `generateChildren()`, and (3) heuristic evaluation of the position — appended using the `setHFunction(new StateFunction() {...})` static method. The code excerpts presented below skip large fragments containing auxiliary constants and helper methods (e.g. for copying states, exporting states as images or files, checking the presence of a piece at particular location, etc.). Also a large part, rather technical, related to the generation of moves and captures was skipped. We only marked the signatures of the suitable methods involved in this aspect, in particular: `populatePossiblePawnKills(...)`, `populatePossibleKingKills(...)`, `populatePossiblePawnMoves(...)`, `populatePossibleKingMoves(...)`. We can mention that methods for captures (kills) work recursively by calling themselves after each partial capture is discovered, in order to look for further capture possibilities (within the same move) from a given intermediate location. When a legal move or capture is discovered, its string representation (e.g. “A1:C3”) is appended to a certain global list of moves passed as a parameter.

We now ask the reader to have a brief look at the code, we then give a few additional comments.

```java
package sac.examples.checkers;

public class Checkers extends GameStateImpl {

    public static int n;

    private static final double PAWN_HEURISTIC_FACTOR = 100.0;
    private static final double KING_HEURISTIC_FACTOR = 10.0 * PAWN_HEURISTIC_FACTOR;

    private List<BoardLocation> whitePawns = null;
    private List<BoardLocation> whiteKings = null;
    private List<BoardLocation> blackPawns = null;
    private List<BoardLocation> blackKings = null;

    private List<String> possibleMoves = null;
    private boolean hasSomeKillMoves = false;

    public Checkers(Checkers parent) {
        super();
        if (parent == null) {
            whitePawns = new ArrayList<BoardLocation>();
            whiteKings = new LinkedList<BoardLocation>();
```

4 We do not present that class hereby for brevity — it is a straightforward implementation, keeping a pair of integers to indicate a two dimensional location and providing the necessary routines to convert forth and back from chess-like notation (e.g.: A1, B4, etc.) to integer coordinates.
blackPawns = new LinkedList<BoardLocation>();
blackKings = new LinkedList<BoardLocation>();
} else {
    whitePawns = copyLocationList(parent.whitePawns);
    whiteKings = copyLocationList(parent.whiteKings);
    blackPawns = copyLocationList(parent.blackPawns);
    blackKings = copyLocationList(parent.blackKings);
}
setMaximizingTurnNow(parent.isMaximizingTurnNow());
...

@Override
public String toString() {
    return toShortString();
}

public String toShortString() {
    StringBuilder builder = new StringBuilder(""");
    builder.append(locationsToString(whitePawns));
    builder.append(DESCRIPTION_SEPARATOR);
    builder.append(locationsToString(whiteKings));
    builder.append(DESCRIPTION_SEPARATOR);
    builder.append(locationsToString(blackPawns));
    builder.append(DESCRIPTION_SEPARATOR);
    builder.append(locationsToString(blackKings));
    builder.append(DESCRIPTION_SEPARATOR);
    builder.append(Boolean.valueOf(isWhiteTurnNow()).toString());
    return builder.toString();
}

@Override
public int hashCode() {
    List<Integer> locationsAsXY = new ArrayList<Integer>();
    for (BoardLocation location : whitePawns) {
        locationsAsXY.add(location.getX());
        locationsAsXY.add(location.getY());
    }
    locationsAsXY.add(0); // serving as separator
    for (BoardLocation location : whiteKings) {
        locationsAsXY.add(location.getX());
        locationsAsXY.add(location.getY());
    }
    locationsAsXY.add(0); // serving as separator
    for (BoardLocation location : blackPawns) {
        locationsAsXY.add(location.getX());
        locationsAsXY.add(location.getY());
    }
    locationsAsXY.add(0); // serving as separator
    for (BoardLocation location : blackKings) {
        locationsAsXY.add(location.getX());
        locationsAsXY.add(location.getY());
    }
    locationsAsXY.add(0); // serving as separator
    locationsAsXY.add(isWhiteTurnNow() ? 1 : -1); // serving as 'whos turn' identifier
return locationsAsXY.hashCode();
}

public BoardLocation makeMove(String moveString, boolean changeTurn) {
    moveString = moveString.toUpperCase();
    possibleMoves = null;
    StringTokenizer tokenizer = new StringTokenizer(moveString, MOVE_SEPARATOR);

    List<BoardLocation> pawns = null;
    List<BoardLocation> kings = null;
    List<BoardLocation> opponentPawns = null;
    List<BoardLocation> opponentKings = null;

    if (isWhiteTurnNow()) {
        pawns = whitePawns;
        kings = whiteKings;
        opponentPawns = blackPawns;
        opponentKings = blackKings;
    } else {
        pawns = blackPawns;
        kings = blackKings;
        opponentPawns = whitePawns;
        opponentKings = whiteKings;
    }

    BoardLocation location = BoardLocation.stringToLocation(tokenizer.nextToken());
    BoardLocation newLocation = null;
    piecesInPlay = null;
    isPawnInPlay = false;
    if (pawns.contains(location)) {
        piecesInPlay = pawns;
        isPawnInPlay = true;
    } else {
        piecesInPlay = kings;
    }

    while (tokenizer.hasMoreTokens()) {
        newLocation = BoardLocation.stringToLocation(tokenizer.nextToken());
        int dx = (newLocation.getX() - location.getX() > 0) ? 1 : -1;
        int dy = (newLocation.getY() - location.getY() > 0) ? 1 : -1;
        piecesInPlay.remove(location);
        while (!location.equals(newLocation)) {
            location.setX(location.getX() + dx);
            location.setY(location.getY() + dy);
            if (opponentPawns.contains(location))
                opponentPawns.remove(location);
            else if (opponentKings.contains(location))
                opponentKings.remove(location);
        }
        piecesInPlay.add(newLocation);
        Collections.sort(piecesInPlay);
    }

    // promotion check
    if (isPawnInPlay) {
        if ((isMaximizingTurnNow()) && (newLocation.getY() == n)) {
            whitePawns.remove(newLocation);
            whiteKings.add(newLocation);
        } else if (isMaximizingTurnNow()) && (newLocation.getY() == 1)) {
            blackPawns.remove(newLocation);
            blackKings.add(newLocation);
        } else if (isMaximizingTurnNow()) && (newLocation.getY() == n) {
            whitePawns.remove(newLocation);
            whiteKings.add(newLocation);
        } else if (isMaximizingTurnNow()) && (newLocation.getY() == 1) {
            blackPawns.remove(newLocation);
            blackKings.add(newLocation);
        }
    }
}
blackPawns.remove(newLocation);
blackKings.add(newLocation);

// ply
if (changeTurn) {
    setMaximizingTurnNow(!isMaximizingTurnNow());
}

refresh();
return newLocation;

...
while (kingsIterator.hasNext()) {
    BoardLocation king = kingsIterator.next();
    populatePossibleKingMoves(this, king, possibleMoves);
}

// pawn regular moves
pawnsIterator = (isWhiteTurnNow()) ? whitePawns.iterator() : blackPawns.iterator();
while (pawnsIterator.hasNext()) {
    BoardLocation pawn = pawnsIterator.next();
    populatePossiblePawnMoves(this, isWhiteTurnNow(), pawn, possibleMoves);
}

return possibleMoves;

private static void populatePossiblePawnKills(Checkers checkers, String prefix, boolean isWhitesTurnNow, BoardLocation pawn, List<String> globalList) {
    ...
}

private static void checkKingKillsAlongDirection(Checkers checkers, BoardLocation king, int dx, int dy, List<BoardLocation> opponentPawns, List<BoardLocation> opponentKings, List<BoardLocation> ownPawns, List<BoardLocation> ownKings, List<String> moves) {
    ...
}

private static void populatePossibleKingKills(Checkers checkers, String prefix, boolean isWhitesTurnNow, BoardLocation king, List<String> globalList, int forbiddenDirection) {
    ...
}

private static void populatePossibleKingMoves(Checkers checkers, BoardLocation king, List<String> globalList) {
    ...
}

private static void populatePossiblePawnMoves(Checkers checkers, boolean isWhitesTurnNow, BoardLocation pawn, List<String> globalList) {
    ...
}

private void eliminateNonMaximalKills(List<String> killMoves) {
    ...
}

@Override
public List<GameState> generateChildren() {
    List<String> moves = getPossibleMoves();
    List<GameState> children = new LinkedList<GameState>();
    for (String move : moves) {
        Checkers child = new Checkers(this);
        child.makeMove(move, true);
        child.setMoveName(move);
        children.add(child);
    }
    return children;
}
The first important element to note is that our Checkers class extends the default game state implementation — sac.game.GameStateImpl, a prerequisite of SaC.

As regards the states identification functionality, both toString(...) and hashCode(...) possibilities work by building up suitable concatenations (accordingly to the type returned) of pieces locations, consecutively for: white pawns, white kings, black pawns, black kings, using some recognizable separators in-between. Finally, the concatenation is appended with an information about whose turn it is now to play.

As regards the move generation functionality, noted should be the getPossibleMoves() method, doing the main work and calling several submethods underneath. It first collects the capture moves for kings and pawns and then, if no capture moves exist, it collects the regular displacement moves. If some capture moves do exist, the player is obliged to perform one of them (instead of a non-capture move) according to the rules of checkers. Also, it is worth to
add that when many capture moves are possible, the player must choose one of such moves that
take the largest possible number of opponent’s pieces. This rule is fulfilled in the code by the
**eliminateNonMaximalKills(...)** method, which removes from the collected list the moves with
too few captures. Finally the **getPossibleMoves()** method returns the list of strings, representing
the moves in a chess-like notation. The method directly responsible for the generation of desc-
cendants, provided by SaC’s interace, is the **generateChildren(...)** method. In our example,
the **generateChildren(...)** implementation calls first the aforementioned **getPossibleMoves()**
method and then iterates over the moves, applying each of them to a copy of the parent state via
a **makeMove(String)** call. Please note also that each descendant gets labeled with the name of the
move it was caused by, using the **child.setMoveName(...)** call — this is also a must within SaC
that was already described in the “Hello world” section 1.3 for games.

As regards the heuristic evaluation of states, we implemented a very simple function for the
purpose of example. As it is required by SaC, the heuristics is attached statically to the class
via the **setHFunction(...)** method. In this case, we pass to it a **StateFunction** object defined
anonymously. Looking at its **calculate(...)** method, two parts can be seen. The first part
pertain to win positions. It returns infinity (with a suitable sign) in case some player is left with
no pieces or no moves (a blockage). The second part, calculates the actual evaluation for the non-
win positions. The evaluation takes into account only two components: (1) the material, where
kings are weighted as being worth ten times a pawn\(^5\) \(2\) the pawn advancement, where each
pawn adds \(1, 2, \ldots, \text{or } n - 1\) points depending on the board row (rank) it is located at, counting
rows towards the promotion. To be more precise, the materialistic evaluation treats each pawn
as being worth 100 points (in chess such points are often referred to as *centipawns*), and the king
is worth 1000 points. Relatively to it, the rewards for pawn advancements are fairly small. Each
next row is being counted for 1 more point (a single centipawn). Clearly, the function described
above does not translate onto a strong program playing checkers. For the sake of simplification all
positional aspects\(^6\) were neglected on purpose.

One more element worth noticing is the implementation of the **isQuiet(...)** method. The
method is provided by the **sac.GameState** interface. In our example, a state of checkers is regarded
as quiet if there are no immediate captures possible from it. We remind that this functionality
is related to the algorithmic gadget known as *Quiescence*, which leads to locally deeper search
horizons when a series of captures occurs; see also page 110 in section 4.2.1.

**Some trees from initial checkers position**

Hereby, we present some illustrations representing SaC’s checkers searches with trees generated
from the initial checkers position (i.e. with white pieces to make the first move). We remark that
by default SaC does not memorize parent-child links in order to save memory. Therefore, in order

\(^5\)This relative value is chosen as a pure guess in our exemplary checkers. Also, owing to that choice, a user
experimenting with the **sac.examples.checkers.CheckersGame** program can easily observe differences in reported
evaluations due to the fact that: pawn advancements change the evaluation by 1 point, pawn captures by 100 points,
and promotions to kings (or king captures) change it by 1000 points.

\(^6\)E.g.: pawns structure / patterns, ‘side vs center’ placement, defended pieces, occupation of main diagonal by kings,
etc.
to produce Graphviz illustrations one has to explicitly reconfigure the algorithm to be executed. The listing below is an example how this reconfiguration can be done.

```java
GameSearchConfigurator configurator = new GameSearchConfigurator();
configurator.setParentsMemorizingChildren(true);

GameSearchAlgorithm algorithm = new AlphaBetaPruning(someCheckersState, configurator);
algorithm.execute();
GameSearchGraphviz.go(algorithm, "d:/output.dot", true, true);
```

We remind the coloring scheme in the figures: yellow denotes the initial state, light gray denotes visited regular states, dark gray indicates non-win terminal states, blue indicates win terminal states (a win for either player), dark red indicates states for which the game value or a bound on that value was read as a ready result (because these states had occurred before) from the so called transposition table3, light red indicates the so called cutoff states which could not affect the game value, green indicates states residing along the principal variation. In every box, displayed is an information about: depth of a state, its heuristic evaluation \( h \), its game value \( v \) (also known as the minimax value) assigned by the search procedure, and its representation (if turned on). We also encourage the reader to zoom-in the figures.

Fig. 4.2 shows a tree generated using alpha-beta pruning algorithm with depth set to 1.0 (two half-moves). Fig. 4.4 is meant to depict the difference in trees generated using the Min-Max algorithm, without and with the Quiescence option. This time the depth is set to 1.5 (so for simplification the states are displayed with no contents inside). Fig. 4.4 compares the trees produced by alpha-beta pruning and Scout, both using 1.5 as the general search horizon, whereas Fig. 4.5 shows an analogical comparison with deeper general horizon — 2.0.

Figure 4.2: Checkers search tree generated by SaC from the initial position (root), using alpha-beta pruning algorithm and search depth set to 1.0 (two half-moves). The Quiescence option is turned on and makes the algorithm look beyond the 1.0 depth in several places.
Figure 4.3: Checkers search trees generated by SaC from the initial position (root), using Min-Max algorithm and with Quiescence turned off (a) and on (b), respectively. General search depth set to 1.5 (three half-moves), but Quiescence feature makes the algorithm visit deeper trees sometimes (when pawn captures take place).
(a) maximum depth 1.5 (unless state not quiet), alpha-beta pruning

(b) maximum depth 1.5 (unless state not quiet), Scout

Figure 4.4: Checkers search trees generated by SaC from the initial position (root), using alpha-beta pruning (a) and Scout (b) algorithms. General search depth set to 1.5 (three half-moves), but Quiescence feature makes the algorithm visit deeper trees sometimes (when pawn captures take place).
(a) maximum depth 2.0 (unless state not quiet), *alpha-beta pruning*

(b) maximum depth 2.0 (unless state not quiet), *Scout*

Figure 4.5: Checkers search trees for initial position as root, generated by SaC with *alpha-beta pruning* (a) and *Scout* (b) algorithms. General search depth set to 2.0 (four half-moves), but *quiescence* feature makes the algorithm visit deeper trees sometimes (when pawn captures take place).
Endgame examples

In this subsection we show some examples of SaC results obtained when searching endgames of checkers.

Fig. 4.6 pertains to a position with 3 white pawns against 2 black pawns, and white pieces to play now. The gap between opposing pieces is just one row, therefore the winning move for white — G5:H6 — is discovered quickly. SaC visited 89 states in about 50 ms. The global search horizon was narrowed to 2.5 and the Quiescence option was on, which made the algorithm reach the 3.5 depth in several places. The displayed payoff associated with the G5:H6 move is \(1.1556598724114885 \times 10^{308}\), representing a win in 7 half-moves.

Fig. 4.7 also pertains to an endgame with 3 pawns against 2, but with two rows of a gap in-between. This ‘little’ difference causes that the algorithm is unable to discover (with certainty) a winning move within the default search horizon of seven half-moves (3.5). Therefore, we forced the algorithm to see deeper, setting the horizon to 5.0. After visiting 2095 and about half a second of time the right move D2:E3 was discovered with its payoff \(1.0622732160550047 \times 10^{308}\). It represents a win in 12 half-moves, as the algorithm reached positions that deep owing to the Quiescence. The reader should also note the difference between the number of states drawn in the figure — 3506 — and the number of states actually visited by the algorithm — 2095. The difference is caused by cutoff states and states for which the evaluation was read from the transposition table.

Fig. 4.8 depicts a “3 kings vs 1 king” ending. It shows how white can trap opponent’s king in 9 half-moves at most. Since the number of visited states — 65438 — is now considerably large, we prepared a special visualization where the right sequence of moves (the principal variation) is exposed whereas subtrees generated by other moves (side moves) are only marked by their roots.
Endgame example 1

(a) endgame position

(b) SaC result

Searching with sac.game.AlphaBetaPruning started...
Searching with sac.game.AlphaBetaPruning done in 46 ms.
Closed states: 89
General depth limit: 2.5
Maximum depth reached (Quiescence): 3.5
Transposition table size: 89
Transposition table uses: 2
Refutation table size: 41
Refutation table uses: 0
Moves scores: {G5:H6=1.1556598724114885E308, F4:E5=999.0}
Best moves: [G5:H6]

(c) search tree for depth 2.5 using alpha-beta pruning (100 states drawn)

Figure 4.6: SaC results for an endgame checkers position. Win for white found within search horizon of depth 3.5.
Endgame example 2

(a) endgame position

(b) SaC result

Searching with sac.game.Scout...
Searching done. Time: 577 ms.
Closed states: 2095
General depth limit: 5.0
Maximum depth reached (Quiescence): 6.0
Transposition table size: 1632
Transposition table uses: 498
Refutation table size: 127
Refutation table uses: 0
Scores: {D2:E3=1.0622732160550047E308, C3:B4=204.0, B2:A3=101.0}
Best move: D2:E3

(c) search tree for depth 5.0 using Scout (3 506 states drawn, star-like layout)

Figure 4.7: SaC results for an endgame checkers position. Win for white found within search horizon of depth 6.0.
Endgame example 3 — “3 kings vs 1 king”

(a) endgame position

(b) SaC result
Searching with sac.game.Scout started...
Searching with sac.game.Scout done in 4164 ms.
Closed states: 65438
General depth limit: 3.5
Maximum depth reached (Quiescence): 4.5
Transposition table size: 64106
Transposition table uses: 82549
Refutation table size: 5734
Refutation table uses: 0
Moves scores: {B2:D4=1.0985902490825263E308, B2:A3=3000.0}
Best moves: [B2:D4]

(c) fragment of search tree with principal variation using for depth 3.5 using Scout

Figure 4.8: SaC results for an endgame checkers position. Win for white found within search horizon of depth 4.5.
4.3.2 Nim

Nim is a two person mind game, where players interchangeably remove objects from a given number of piles. A player must take at least one object and can take as many objects as he wants provided that they come from the same pile. Depending on the convention being played, the player to take the last remaining object is either the winner (normal variant) or the loser (misère variant). A common initial setup is that given are three piles, consisting of 3, 4 and 5 objects; but, other setups are possible.

Nim has been solved mathematically (for any number of piles and objects). There exist a simple calculation scheme telling which player will win at a given position with an optimal line of play. Moreover, it is possible to tell the right move. The key notion of the scheme is a so called nim-sum of pile sizes, coded in the binary system. The sum is taken using the exclusive-or operation (or modulo 2 sum). For example for piles of sizes 3, 4, 5, their binary representations and the nim-sum are as follows:

\[
\begin{align*}
3 &= (0, 1, 1)_2, \\
4 &= (1, 0, 0)_2, \\
5 &= (1, 0, 1)_2, \\
nim-sum: &= (0, 1, 0)_2.
\end{align*}
\]

It turns out that the optimal strategy boils down to making moves which keep the nim-sum of the remaining piles equal to zero. Interestingly, this strategy is in general valid for both conventions of the play (normal or misère), and only the very endgames require slight changes depending on the convention. In the normal variant, when the game is reduced to two piles only, the player should make such moves to keep both piles equal. In the misère variant, when there remain piles with no more than two objects, the correct move is to leave an odd number of piles of size one. More details on Nim, its variants, history and strategy can be found at: [https://en.wikipedia.org/wiki/Nim](https://en.wikipedia.org/wiki/Nim).

We now move to the implementation issues and SaC’s API. As in former examples, we first show code excerpts from the state representation class — `sac.examples.nim.NimState`, then we comment on it with the focus on vital elements: generation of descendants, identification, heuristics.

```java
package sac.examples.nim;

public class NimState extends GameStateImpl {

    private List<Integer> piles;

    public NimState() {
        piles = new LinkedList<Integer>();
        piles.add(3);
        piles.add(4);
        piles.add(5);
        setMaximizingTurnNow(true);
    }
}
```
public NimState(List<Integer> initial, boolean isWhiteTurnNow) {
    piles = new ArrayList<Integer>(initial);
    Collections.copy(piles, initial);
    setMaximizingTurnNow(isWhiteTurnNow);
}

public NimState(NimState parent) {
    super();
    piles = new ArrayList<Integer>();
    for (int item : parent.getPiles()) {
        piles.add(item);
    }
    setMaximizingTurnNow(parent.isMaximizingTurnNow());
}

public List<Integer> getPiles() {
    return piles;
}

public List<List<Integer>> getPossibleMoves() {
    List<List<Integer>> list = new ArrayList<List<Integer>>();
    int qi;
    for (int i = 0; i < piles.size(); i++) {
        qi = piles.get(i);
        for (int j = 1; j <= qi; j++) {
            ArrayList<Integer> move = new ArrayList<Integer>(
                Collections.nCopies(piles.size() - 1, 0));
            move.add(i, j);
            list.add(move);
        }
    }
    return list;
}

public void makeMove(List<Integer> move) {
    assert (checkMove(move));
    int newValue;
    for (int i = 0; i < piles.size(); i++) {
        newValue = piles.get(i) - move.get(i);
        piles.remove(i);
        piles.add(i, newValue);
    }
    setMaximizingTurnNow(!maximizingTurnNow);
    refresh();
}

public void makeMove(String stringMove) {
    String[] si = stringMove.substring(1, stringMove.length() - 1).split(", ");
    List<Integer> move = new LinkedList<Integer>();
    for (String s : si) {
        move.add(Integer.parseInt(s.trim()));
    }
    makeMove(move);
}

public boolean checkMove(List<Integer> move) {
    if (piles.size() != move.size()) {
        return false;
    }
for (int i = 0; i < piles.size(); i++) {
    if (!(move.get(i) <= piles.get(i)) && (move.get(i) > 0)) {
        return false;
    }
}
return true;}

@Override
public List<GameState> generateChildren() {
    List<GameState> children = new ArrayList<GameState>();
    for (List<Integer> move : getPossibleMoves()) {
        NimState child = new NimState(this);
        child.makeMove(move);
        child.setMoveName(move.toString());
        children.add(child);
    }
    return children;
}

@Override
public String toString() {
    return piles.toString() + "|" + maximizingTurnNow;
}

@Override
public int hashCode() {
    String string = toString();
    return string.hashCode();
}

public boolean isTerminal() {
    for (Integer item : piles) {
        if (item > 0)
            return false;
    }
    return true;
}

static {
    setHFunction(new StateFunction() {
        @Override
        public double calculate(State state) {
            NimState nimState = (NimState) state;
            if (nimState.isTerminal())
                return Double.POSITIVE_INFINITY * (nimState.isMaximizingTurnNow() ? -1 : 1);
            return 0.0;
        }
    });
}

The piles of objects are represented by a list of integers — List<Integer> piles — so that piles.get(i) returns the number of objects remaining on the i-th pile.

There are two methods present in the class related to making a move in a Nim game — makeMove(List<Integer> move) and makeMove(String stringMove). They differ only in the
type of the argument and work the same way underneath. As described in the introduction on Nim, typically a player is allowed to remove objects only from one pile. In the implementation though we decided to go for more generality, and a move is a sequence of length equal to the number of piles, where numbers in the sequence represent how many objects should be subtracted from successive piles. Hence, potentially one could write for example: `nimState.makeMove(“2,0,1”)`, which indicates that two objects should be removed from the first pile, no objects from the second pile, and one object from the third. Despite the more general code at this place, there is a method named `getPossibleMoves(...)`, which in fact generates only the moves where objects are subtracted from a single pile. Therefore, the generated sequences, representing legal moves, consist of a single non-zero value somewhere and zeros elsewhere.

As regards the identification of states, both `toString()` and `hashCode()` methods have been implemented. Yet, in fact the `hashCode()` method first calls the `toString()` and then converts its result to an integer, using the Java built-in mechanism to calculate a hash code for a string.7

The position evaluation function is, as always, attached via the `setHFunction(...)` static method. In our Nim implementation we distinguish only three possible evaluations: ±∞ (for win positions) or 0 (for non-win positions). As it was mentioned before in the introduction, Nim is not a kind of game (contrarily e.g. to chess or checkers) where a player is capable of building slowly and incrementally an advantage for himself. A move is either correct or not at any stage of a game. That is why the code shown represents only such a primitive heuristic function.8 In the presented form of the code, the heuristics reflects the normal game variant, i.e. the player to take the last object is the winner.

7 see [http://docs.oracle.com/javase/7/docs/api/java/lang/String.html#hashCode%28%29](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html#hashCode%28%29)
8 It is possible to comment that the remarks above (about a move being either correct or not) are in fact also correct for chess. Chess is not a solved game. And although its game tree is astronomically huge, it is a finite tree. Rules like: triple repetition of a position, perpetual check, etc. cause that the game cannot be played for ever. Since the chess tree is finite, it is easy to prove that there exist an objective answer about the outcome of the game under an optimal play. We, people, do not know that outcome (chess is still unsolved), but it certainly does exist.
Chapter 5

Tools

This chapter is devoted to additional tools provided with SaC. They are not indispensable in a typical usage, but provide some auxiliary functionalities that might make the work with SaC more convenient. First, discussed is SaC’s interface to Graphviz — a language for automatic graph drawing. Secondly, we describe how to carry out batch experiments using SaC and how to collect and plot statistics from such experiments. The plots are generated with the help of jFreeChart library. Finally, we present the possibility of monitoring the progress of graph searching (for heavy problems).

5.1 Graphviz

Graphviz (short for Graph Visualization Software) is a free software and an underlying language for automatic graph drawing. The reader can jump ahead to look at figures 5.1 and 5.2 to have an outlook on the Graphviz language. Yet, we should stress that a SaC user is not required to be familiar with that language and still can generate visualizations for his search experiments. As one can see, for the graph from Fig. 5.1 there is no information in the code about the formatting style of the nodes and edges. The code describes just the structure of the graph. The graph from Fig. 5.2 (or more precisely the search tree) has been generated automatically by SaC for a simple sliding puzzle problem. Apart from the structure, the code now describes also the formatting of its elements, partially via the HTML language.

In SaC, we enable a possibility to generate output text files, compliant with the language. The files contain representations of graphs or trees searched by a particular algorithm. Having such a file and the Graphviz software installed, the user can produce visualizations using one of Graphviz engines like dot, neato, etc. For more details about the Graphviz and its language we address the reader to the Graphviz official website: [http://www.graphviz.org](http://www.graphviz.org). We shall now discuss some elements of SaC’s API related to Graphviz.

Any search algorithm from SaC that has stopped successfully can potentially be ‘flushed out’ to a .dot file compliant with the Graphviz language. There is one necessary condition — the user has to explicitly reconfigure SaC, before an algorithm is executed, and make parent states memorize their children states. By default, only the opposite way link is memorized — from a
Figure 5.1: A simple directed graph coded in the Graphviz language (a) and rendered out (b).

child to its parent. The mentioned reconfiguration facilitates the generation of the Graphviz code, but obviously is more memory consuming.

The reconfiguration is done either by the `sac.graph.GraphSearchConfigurator` class or the `sac.game.GraphSearchConfigurator` class, depending on the type of search. Let us remind here an example from the checkers related section.

```java
GameSearchConfigurator configurator = new GameSearchConfigurator();
configurator.setParentsMemorizingChildren(true);
GameSearchAlgorithm algorithm = new AlphaBetaPruning(someCheckersState, configurator);
algorithm.execute();
GameSearchGraphvizer.go(algorithm, "d:/output.dot", true, true);
```

In the example, the algorithm to be executed is, at its instantiation, equipped with a an initial state and additionally with a configurator. The configurator’s option to memorize parent-children links is set to true. Once the algorithm is finished, an output .dot file compliant with Graphviz is produced by an invocation of the `sac.graphviz.GameSearchGraphvizer.go(…)` method. The arguments to that method are as follows: the algorithm object, a path to the target output file, a boolean flag stating whether the nodes should be rendered out with their contents (i.e. state representations) or just as empty circles, and finally another boolean flag stating wheter names of moves (or manipulations) should be printed near the edges.

On the Windows operating system, the following exemplary command-line invocation would produce an output .pdf file from the given .dot file (provided that Graphviz is installed in the system and that the dot command is visible).

```
D:\>dot -Tpdf output.dot -o d:\output.pdf
```
Figure 5.2: Graphviz code generated automatically by SaC for a simple sliding puzzle search (above) and its visualization rendered out (below).
CHAPTER 5. TOOLS

By default, the Graphviz engine renders its outputs according to the so called dot layouter. For search trees, this results in natural top-down visualizations where successive rows correspond to successive depth levels and the initial state is at the top. For star-like visualizations, with the initial state in the middle, the neato layouter should be used. The suitable command-line invocation for that purpose could look like this:

```
D:\>dot -Tpdf output.dot -Kneato -o d:\output.pdf
```

For more information about available options, switches and command-line Graphviz invocations the reader is addressed to the documentation, in particular to: [http://www.graphviz.org/doc/info/command.html](http://www.graphviz.org/doc/info/command.html).

As regards the displayed contents of nodes, a SaC user can affect those by providing his state class with an implementation of the toGraphvizLabel() method. This method is visible and can be overridden because the base interface in SaC — sac.State — extends a sac.graphviz.Graphvizable interface with that particular method. By default, the toGraphvizLabel() method returns the identifier of a state object. Therefore, when elegant visualizations are intended, a more meaningful state representation should be implemented. Please note, that Graphviz allows to use HTML language for that purpose. Thus, a programmer can construct a suitable HTML string within his implementation of the toGraphvizLabel() method. As an example, below we show a code excerpt for that purpose taken from the sliding puzzle class.

```java
@Override
public String toGraphvizLabel() {
    StringBuilder builder = new StringBuilder();
    builder.append("<\FONT FACE='monospace' POINT-SIZE=8'><TABLE BORDER='0' CELLBORDER='1'
    CELLSPACING=0'>");

    int k = 0;
    for (int i = 0; i < n; i++) {
        builder.append("<TR>");
        for (int j = 0; j < n; j++) {
            builder.append("<TD>");
            ((board[k] == 0) ? "" : board[k]) + "</TD>");
            k++;
        }
        builder.append("</TR>");
    }
    builder.append("</TABLE></FONT>");
    return builder.toString();
}
```

5.2 Statistics and charts for batch experiments

SaC allows to conveniently carry out batch experiments. What do we mean by that? Imagine a compact piece of code, containing a number of loops, that iterate for example over: multiple random instances of a certain problem, multiple search algorithms, multiple heuristic functions, and different data structures involved; and in each iteration an execution of the search procedure for given settings is performed. Once such a collection of experiments is registered, one would
like to easily extract some statistics out of it — for example: an average solution path length per an algorithm, a total of visited states per a heuristic, an average duration time per particular combinations of an algorithm and a data structure, a variance in the size of the transposition table, and so forth. The object-oriented design of SaC allows to do so, see the example below.

```java
public static void main(String[] args) throws Exception {
    System.out.println("Starting....");
    long t1 = System.currentTimeMillis();
    Stats stats = new Stats();
    
    // loop over random sliding puzzle problems
    for (int i = 0; i < 100; i++) {
        SlidingPuzzle puzzle = new SlidingPuzzle((byte) 3);
        puzzle.shuffle((Math.random() > 0.5) ? 1000 : 1001); // shuffling
        
        // initial solution by A* so that optimal path length is known for further statistics
        AStar astar = new AStar(new SlidingPuzzle(puzzle));
        astar.execute();
        int optimalPathLength = astar.getSolutions().get(0).getPath().size();
        
        // loop over algorithms
        GraphSearchAlgorithm[] algorithms = { new BestFirstSearch(), new AStar() };
        for (GraphSearchAlgorithm algorithm : algorithms) {
            
            // loop over heuristics
            StateFunction[] heurs = { new HFunctionManhattan(), new HFunctionLinearConflicts() };
            for (StateFunction h : heurs) {
                
                // loop over different open set implementations
                Class[] openSetClasses = { OpenSetAsPriorityQueue.class, OpenSetAsPriorityQueueFastContains.class, OpenSetAsPriorityQueueFastContainsFastReplace.class };
                for (Class openSetClass : openSetClasses) {
                    algorithm.setInitial(new SlidingPuzzle(puzzle));
                    SlidingPuzzle.setHFunction(h);
                    GraphSearchConfigurator configurator = new GraphSearchConfigurator();
                    configurator.setOpenSetClassName(openSetClass.getName());
                    algorithm.setConfigurator(configurator);
                    
                    // search
                    algorithm.execute();
                    
                    // register current single run in stats object
                    stats.addEntries(algorithm, i, algorithm.getClass(), h.getClass(), openSetClass, optimalPathLength);
                    
                }
            }
        }
    
    long t2 = System.currentTimeMillis();
    System.out.println("Experiment total time [s]: " + (0.001 * (t2 - t1)));
    
    // charts based on collected statistics to be produced here ...
}
```
The example represents experimentations on random sliding puzzle problems, in the $3 \times 3 - 1$ variant. It involves four nested loops in which varied are: problems, algorithms, heuristics, and data structures implementing the Open set. An important element to notice in the code is the sac.stats.Stats object. The role of this object is to register singular experiments and then to produce the wanted statistics out of it.

As far as registration of experiments is concerned, the usage is simple and convenient. One instantiates a sac.stats.Stats object at the very start and then in the middle of all loops one registers results of a particular execution by a single line of code, calling the addEntries(...) method. The full signature of that method is as follows.

```java
public void addEntries(SearchAlgorithm algorithm, Object... multiIndex) {
    ...
}
```

As one can see, the method requires a reference to the search algorithm (that finished its execution) as the first argument. The second argument, named multiIndex, is passed by the ellipsis programming mechanism — so in fact it can by an arbitrary number of arguments that together describe the settings for a given experiment. It can be explained that on the low level SaC uses that multiIndex as a key to a large hash map that stores results of experiments inside the sac.stats.Stats object. If a reader is interested in such low level details we address him to the source code of the library.

The exemplary lines of code shown below are the sequel to the previous batch experiment. They demonstrate how one can extract some wanted statistics (i.e. their numeric values) once all loops are finished. We encourage the reader to try to guess what the lines are meant to calculate.

```java
1  stats.mean(StatsCategory.GRAPH_SEARCH_PATH_LENGTH.toString(), null, AStar.class, null, null, null);
2  stats.mean(StatsCategory.GRAPH_SEARCH_PATH_LENGTH.toString(), null, BestFirstSearch.class, null, null, null);
3  stats.mean(StatsCategory.GRAPH_SEARCH_CLOSED_STATES.toString(), null, null, HFunctionManhattan.class, null, null);
4  stats.variance(StatsCategory.GRAPH_SEARCH_CLOSED_STATES.toString(), null, null, HFunctionManhattan.class, null, null);
5  stats.mean(StatsCategory.GRAPH_SEARCH_CLOSED_STATES.toString(), null, null, HFunctionLinearConflicts.class, null, null);
6  stats.variance(StatsCategory.GRAPH_SEARCH_CLOSED_STATES.toString(), null, null, HFunctionLinearConflicts.class, null, null);
7  stats.max(StatsCategory.GRAPH_SEARCH_DURATION_TIME.toString(), null, AStar.class, null, OpenSetAsPriorityQueue.class);
8  stats.max(StatsCategory.GRAPH_SEARCH_DURATION_TIME.toString(), null, AStar.class, null, OpenSetAsPriorityQueueFastContainsFastReplace.class);
```

Well, the lines from above, if augmented with suitable System.out.println(...) instructions and descriptions, could result in the following output to the console:

```
Mean solution path length for A*: 23.2
Mean solution path length for Best-first-search: 57.45
Mean number of closed states for Manhattan heuristics: 616.125
Variance of number of closed states for Manhattan heuristics: 533885.259375
Mean number of closed states for Manhattan + LC heuristics: 335.325
Variance of number of closed states for Manhattan + LC heuristics: 196093.519375
Max duration time for A* and standard open set [ms]: 120.0
Max duration time for A* and FCFR open set [ms]: 80.0
```
CHAPTER 5. TOOLS

As one can figure out the extraction of statistics from the stats object is based somehow on a suitable usage of the null symbol. The wanted aggregation operation (mean, variance, max, min) should be called on the stats object and two elements should be specified: (1) with respect to which quantity the aggregation should be taken, (2) what is the multiindex pattern to aggregate over — this part involves the usage of nulls. The measured quantities at disposal are defined by the sac.stats.StatsCategory enumeration:

```
public enum StatsCategory {
    GRAPH_SEARCH_DURATION_TIME,
    GRAPH_SEARCH_CLOSED_STATES,
    GRAPH_SEARCH_OPEN_STATES,
    GRAPH_SEARCH_SOLUTIONS,
    GRAPH_SEARCH_PATH_LENGTH,
    GRAPH_SEARCH_PATH_G,
    GAME_SEARCH_DURATION_TIME,
    GAME_SEARCH_CLOSED_STATES,
    GAME_SEARCH_OPEN_STATES,
    GAME_SEARCH_SOLUTIONS,
    GAME_SEARCH_TRANSPOSITION_TABLE_SIZE,
    GAME_SEARCH_TRANSPOSITION_TABLE_USES,
    GAME_SEARCH_REFUTATION_TABLE_SIZE,
    GAME_SEARCH_REFUTATION_TABLE_USES,
    GAME_SEARCH_DEPTH_REACHED;
}
```

As regards the multiindex pattern, it is, again, specified by the ellipsis mechanism. So, the programmer is supposed to give a sequence of objects according to the same scheme (order) that was used when experiments were being registered. The mentioned trick at this stage is that any position within the multiindex can be either fixed or replaced by a null symbol, which is intended to mean 'any'. In other words a particular arrangement of nulls defines the scope (range) for the aggregation. For example, the two last printouts to the screen in the former example involve a fixed algorithm and a fixed type of Open set, whereas all other parameters are free to vary.

Additionally, SaC offers a possibility to automatically generate plots out of statistics. This functionality is underneath based on the jFreeChart library. Let us start by an example of a bar chart generated using the statistics. We believe the code below is self-explanatory and the resulting jpeg file with the chart is presented in the Fig. 5.3.

```
StatsBarChart statsBarChart1 = new StatsBarChart(stats, "sliding_puzzle_duration", "algorithm_and_heuristics", "time_[ms]");
statsBarChart1.setValue("LC", "A*", StatsOperationType.MEAN, StatsCategory.
    GRAPH_SEARCH_DURATION_TIME.toString(), null, AStar.class,
    HFunctionLinearConflicts.class, null, null);
statsBarChart1.setValue("M", "A*", StatsOperationType.MEAN, StatsCategory.
    GRAPH_SEARCH_DURATION_TIME.toString(), null, AStar.class,
    HFunctionManhattan.class, null, null);
statsBarChart1.setValue("LC", "BFS", StatsOperationType.MEAN, StatsCategory.
    GRAPH_SEARCH_DURATION_TIME.toString(), null, BestFirstSearch.class,
    HFunctionLinearConflicts.class, null, null);
statsBarChart1.setValue("M", "BFS", StatsOperationType.MEAN, StatsCategory.
    GRAPH_SEARCH_DURATION_TIME.toString(), null, BestFirstSearch.class,
    HFunctionManhattan.class, null, null);
statsBarChart1.saveAsJPEG("./sliding_puzzle_duration.jpg");
```
The code below produces three more exemplary plots: one bar chart and two x-y plots, shown in figures 5.4, 5.5 and 5.6, respectively.

```java
// sliding puzzle - duration over open sets (bar chart)
StatsBarChart statsBarChart2 = new StatsBarChart(stats, "sliding_puzzle_duration", "open_set_and_algorithm", "time_[ms]");
statsBarChart2.setValue("A\*", "PQ", StatsOperationType.MEAN, StatsCategory.GRAPH_SEARCH_DURATION_TIME.toString(), null, AStar.class, null,
OpenSetAsPriorityQueue.class, null);
statsBarChart2.setValue("BFS", "PQ", StatsOperationType.MEAN, StatsCategory.GRAPH_SEARCH_DURATION_TIME.toString(), null, BestFirstSearch.class, null,
OpenSetAsPriorityQueue.class, null);
statsBarChart2.setValue("A\*", "PQFC", StatsOperationType.MEAN, StatsCategory.GRAPH_SEARCH_DURATION_TIME.toString(), null, AStar.class, null,
OpenSetAsPriorityQueueFastContains.class, null);
statsBarChart2.setValue("BFS", "PQFC", StatsOperationType.MEAN, StatsCategory.GRAPH_SEARCH_DURATION_TIME.toString(), null, BestFirstSearch.class, null,
OpenSetAsPriorityQueueFastContains.class, null);
statsBarChart2.setValue("A\*", "PQFCFR", StatsOperationType.MEAN, StatsCategory.GRAPH_SEARCH_DURATION_TIME.toString(), null, AStar.class, null,
OpenSetAsPriorityQueueFastContainsFastReplace.class, null);
statsBarChart2.setValue("BFS", "PQFCFR", StatsOperationType.MEAN, StatsCategory.GRAPH_SEARCH_DURATION_TIME.toString(), null, BestFirstSearch.class, null,
OpenSetAsPriorityQueueFastContainsFastReplace.class, null);
statsBarChart2.saveAsJPEG("./sliding_puzzle_duration_2.jpg");

// sliding puzzle - closed states distribution over optimal path length (xy chart)
StatsXYChart statsXYChart1 = new StatsXYChart(stats, "sliding_puzzle_duration", "closed_states_as_path_length_grows", "optimal_path_length", "closed_states");
statsXYChart1.addSeries("A\*_LC", StatsOperationType.MEAN, StatsCategory.GRAPH_SEARCH_CLOSED_STATES.toString(), 4, null, AStar.class,
HFunctionLinearConflicts.class, null, null);
```
statsXYChart1.addSeries("A∗_M", StatsOperationType.MEAN, StatsCategory.GRAPH_SEARCH_CLOSED_STATES.toString(), 4, null, AStar.class, HFunctionManhattan.class, null, null);
statsXYChart1.addSeries("BFS_LC", StatsOperationType.MEAN, StatsCategory.GRAPH_SEARCH_CLOSED_STATES.toString(), 4, null, BestFirstSearch.class, HFunctionLinearConflicts.class, null, null);
statsXYChart1.addSeries("BFS_M", StatsOperationType.MEAN, StatsCategory.GRAPH_SEARCH_CLOSED_STATES.toString(), 4, null, BestFirstSearch.class, HFunctionManhattan.class, null, null);
statsXYChart1.saveAsJPEG("./sliding_puzzle_closed_states.jpg");

// sliding puzzle - found path length over optimal path length (xy chart)
StatsXYChart statsXYChart2 = new StatsXYChart(stats, "sliding_puzzle - found path length over optimal path length", "found_path_length", "optimal_path_length");
statsXYChart2.addSeries("A∗_any", StatsOperationType.MEAN, StatsCategory.GRAPH_SEARCH_PATH_LENGTH.toString(), 4, null, AStar.class, null, null);
statsXYChart2.addSeries("BFS_LC", StatsOperationType.MEAN, StatsCategory.GRAPH_SEARCH_PATH_LENGTH.toString(), 4, null, BestFirstSearch.class, HFunctionLinearConflicts.class, null, null);
statsXYChart2.addSeries("BFS_M", StatsOperationType.MEAN, StatsCategory.GRAPH_SEARCH_PATH_LENGTH.toString(), 4, null, BestFirstSearch.class, HFunctionManhattan.class, null, null);
statsXYChart2.saveAsJPEG("./sliding_puzzle_path_lengths.jpg");

Figure 5.4: Bar chart generated by SaC from a batch experiment over random sliding puzzles: duration times for three data structures (groups) and two algorithms (subgroups).
Figure 5.5: $x$-$y$ plot generated by SaC from a batch experiment over random sliding puzzles: number of closed (visited) states as a function of solution path length, for different algorithms and heuristics.

Figure 5.6: $x$-$y$ plot generated by SaC from a batch experiment over random sliding puzzles: discovered path lengths vs actual optimal path length, for different algorithms and heuristics.
5.3 Graph search monitors

For some sufficiently hard graph search problems, the search procedure may take minutes or even hours. Often, it is impossible for the user to give even a rough estimate on the time duration needed by an algorithm, prior to its execution. In such situations the user is typically interested in monitoring the procedure, i.e. observing its progress and quantities involved. SaC’s API allows to do so.

Within the sac.graph.GraphSearchConfiguration object, there are three options related to monitors: monitorOn, monitorClassName, monitorRefreshTime. They can be modified either directly by suitable setters or via a properties file. The first option — monitorOn — serves as a switch, and is by default set to false. It means no monitoring is wanted. Setting this option to true, releases the remaining options. The second option — monitorClassName — is expected to be set with a name of a particular search monitor class. There are two ready-made monitor classes within SaC: sac.graph.ConsoleGraphSearchMonitor and sac.graph.GraphicalGraphSearchMonitor. Note however, that new custom classes can be developed (e.g. logging the progress to a file or a database, etc.) as long as they extend the base abstract class called: sac.graph.GraphSearchMonitor. Finally, the third option — monitorRefreshTime — specifies the time period in milliseconds after which the monitor is supposed to refresh its output for the user.

The fragment below shows an example of a console monitor configured for a particular 4 × 4 sliding puzzle problem. The refresh period is set to 1000 ms, so the monitor prints out a summary of the current progress to the console (every second) by invoking the printCurrentSummary() method.

```java
GraphSearchConfigurator configurator = new GraphSearchConfigurator();
configurator.setMonitorOn(true);
configurator.setMonitorClassName(ConsoleGraphSearchMonitor.class.getName());
configurator.setMonitorRefreshTime(1000);
SlidePuzzle puzzle = new SlidePuzzle(new byte[]{0, 3, 10, 2, 1, 8, 9, 6, 12, 11, 4, 7, 14, 5, 15, 13});
System.out.println(puzzle);
GraphSearchAlgorithm algorithm = new AStar(puzzle, configurator);
algorithim.execute();
System.out.println("Done.");
```

<table>
<thead>
<tr>
<th>0</th>
<th>3</th>
<th>10</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>14</td>
<td>5</td>
<td>15</td>
<td>13</td>
</tr>
</tbody>
</table>

[SAC] *** Starting sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary().
[SAC] Time: 0.0 s.
[SAC] Solutions so far: 0.
[SAC] Best state’s h: 32.0.
[SAC] Best state’s f: 32.0.
[SAC] Best state’s depth: 0.0.
[SAC] Current state’s h: 32.0.
CHAPTER 5. TOOLS

[SAC] Current state’s f: 32.0.
[SAC] Current state’s depth: 0.0.
[SAC] Free memory: 42615384
[SAC] Used memory: 128510896
[SAC] Max memory: 190793264

*** Done with sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary().

*** Starting sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary()...

[SAC] Time: 1.612 s.
[SAC] Solutions so far: 0.
[SAC] Closed states: 75198.
[SAC] Open states: 78277.
[SAC] Best state’s h: 4.0.
[SAC] Best state’s f: 48.0.
[SAC] Best state’s depth: 36.0.
[SAC] Current state’s h: 21.0.
[SAC] Current state’s f: 42.0.
[SAC] Free memory: 100827528
[SAC] Used memory: 184942592
[SAC] Max memory: 190793264

*** Done with sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary().

*** Starting sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary()...

[SAC] Time: 2.174 s.
[SAC] Solutions so far: 0.
[SAC] Closed states: 121525.
[SAC] Best state’s h: 3.0.
[SAC] Best state’s f: 42.0.
[SAC] Best state’s depth: 39.0.
[SAC] Current state’s h: 27.0.
[SAC] Current state’s f: 44.0.
[SAC] Current state’s depth: 17.0.
[SAC] Free memory: 207060480
[SAC] Used memory: 313262080
[SAC] Max memory: 190793264

*** Done with sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary().

*** Starting sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary()...

[SAC] Time: 3.241 s.
[SAC] Solutions so far: 0.
[SAC] Closed states: 233229.
[SAC] Best state’s h: 3.0.
[SAC] Best state’s f: 42.0.
[SAC] Best state’s depth: 39.0.
[SAC] Current state’s h: 22.0.
[SAC] Current state’s f: 44.0.
[SAC] Current state’s depth: 22.0.
[SAC] Free memory: 193846936
[SAC] Used memory: 362020864
[SAC] Max memory: 190793264

*** Done with sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary().

*** Starting sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary()...

[SAC] Solutions so far: 1.
[SAC] Closed states: 282585.
[SAC] Best state’s h: 0.0.
[SAC] Best state’s f: 44.0.
[SAC] Best state’s depth: 44.0.
[SAC] Current state’s h: 0.0.
[SAC] Current state’s f: 44.0.
[SAC] Current state’s depth: 44.0.
[SAC] Free memory: 137477528
[SAC] Used memory: 302928664
[SAC] Max memory: 190793264

*** Done with sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary().

Done.
Fig. 5.7 depicts a visualization being generated by the `sac.graph.GraphicalGraphSearchMonitor` for the same sliding puzzle problem. This time the refresh period is set to 100 ms, so the reporting points are laidy more densely than before. An interesting gap can be seen in between the time moments of 1.5 s and 2.3 s. The gap can be explained by exhausted memory needed for the hash map underlying the `Closed` set and the JVM taking time to allocate new enlarged fragment of memory. By doing so, the JVM sets the monitor thread aside.

![Graphical Graph Search Monitor](image)

Figure 5.7: A screenshot from a running graph search monitor.
Bibliography


Pearl, J. (1982), The Solution for the Branching Factor of the Alpha-beta Pruning Algorithm and Its Optimality’, *Communications of the ACM* 25(8), 559–564.


Chapter 6

Appendices

6.1 Implementation of state abstraction

```java
package sac;

import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

/**
 * Abstract partial implementation of State interface.
 */
public abstract class StateImpl implements State {

    /**
     * Identifier for this state.
     */
    protected Identifier identifier = null;

    /**
     * Reference to this state's parent.
     */
    protected State parent = null;

    /**
     * List of references to this state's children.
     */
    protected List<? extends State> children = null;

    /**
     * Depth of this state (number of parent states above it).
     */
    protected double depth = 0;

    /**
     * The heuristics - estimated distance to the solution state. Remains null until the first call of getH().
     */
    protected Double h = null;

    /**
     * Name of the move that led to generating this state.
     */
    protected String moveName;

    /**
     * Default h function (returns 0).
     */
    protected static StateFunction hFunction;
```
public StateImpl() {
    this.parent = null;
    this.children = new LinkedList<State>();
}

@Override
public final Identifier getIdentifier() {
    if (identifier == null) {
        identifier = new Identifier(this);
    }
    return identifier;
}

@Override
public final void refreshIdentifier() {
    identifier = new Identifier(this);
}

@Override
public State getParent() {
    return parent;
}

@Override
public final void setParent(State parent) {
    this.parent = parent;
}

@Override
public final void setDepth(double depth) {
    this.depth = depth;
}

@Override
public List<? extends State> getChildren() {

public final double getDepth() {
    return depth;
}

@Override
public List<? extends State> getPath() {
    List<State> path = new LinkedList<State>();
    State temp = this;
    path.add(temp);
    while (temp.getParent() != null) {
        temp = temp.getParent();
        path.add(temp);
    }
    Collections.reverse(path);
    return path;
}

@Override
public List<String> getMovesAlongPath() {
    List<String> moves = new LinkedList<String>();
    List<? extends State> path = getPath();
    for (State state : path) {
        if (state.getParent() == null) continue;
        moves.add(state.getMoveName());
    }
    return moves;
}

@Override
public boolean equals(Object otherState) {
    State otherState2 = (State) otherState;
    return getIdentifier().equals(otherState2.getIdentifier());
}

@Override
public int compareTo(State otherState) {
    return getIdentifier().compareTo(otherState.getIdentifier());
}

/**
 * Sets new h function.
 * @param hFunction to best
 */
public static final void setHFunction(StateFunction hFunction) {

StateImpl.hFunction = hFunction;

/* (non-Javadoc)
 * @see sac.State#getH()
 */
@Override
public final double getH() {
    if (h == null)
        h = Double.valueOf(hFunction.calculate(this));
    return h;
}

/* (non-Javadoc)
 * @see sac.State#setH(Double)
 */
@Override
public final void setH(Double h) {
    this.h = h;
}

/* (non-Javadoc)
 * @see sac.State#refreshH()
 */
@Override
public final void refreshH() {
    h = null;
    hFunction.calculate(this);
}

/* (non-Javadoc)
 * @see sac.graphviz.Graphvizable#toGraphvizLabel()
 */
@Override
public String toGraphvizLabel() {
    return (moveName != null) ? moveName : getIdentifier().toString();
}
6.2 Implementation of graph state abstraction

```java
package sac.graph;
import java.util.List;
import sac.State;
import sac.StateFunction;
import sac.StateImpl;

/**
 * Abstract partial implementation of GraphState interface. User's graph state classes should extend this class.
 */
public abstract class GraphStateImpl extends StateImpl implements GraphState {

    protected Double g = 0.0;

    protected Double f = null;

    protected static StateFunction gFunction;

    /**
     * Default g function (returns parent's g + 1).
     */
    protected static class GFunction extends StateFunction {
        @Override
        public double calculate(State state) {
            return (state.getParent() == null) ? 0.0 : ((GraphState) state.getParent()).getG() + 1.0;
        }
    }

    public static final void setGFunction(StateFunction gFunction) {
        GraphStateImpl.gFunction = gFunction;
    }

    /**
     * Creates new instance of graph state.
     */
    public GraphStateImpl() {
        super();
    }
}
```
59 } 
60 */  
61 /* (non-Javadoc)  
62 * @see sac.GraphState#getParent()  
63 */  
64 @Override  
65 public final GraphState getParent() {  
66 return (parent == null) ? null : (GraphState) parent;  
67 } 
68 */  
69 /* (non-Javadoc)  
70 * @see sac.GraphState#getChildren()  
71 */  
72 @Override  
73 public final List<GraphState> getChildren() {  
74 return (List<GraphState>) children;  
75 } 
76 */  
77 /* (non-Javadoc)  
78 * @see sac.GraphState#getPath()  
79 */  
80 @Override  
81 public final List<GraphState> getPath() {  
82 return (List<GraphState>) super.getPath();  
83 } 
84 */  
85 /* (non-Javadoc)  
86 * @see sac.GraphState#getG()  
87 */  
88 @Override  
89 public final double getG() {  
90 if (g == null) {  
91 g = Double.valueOf(gFunction.calculate(this));  
92 return g;  
93 }  
94 */  
95 /* (non-Javadoc)  
96 * @see sac.GraphState#getF()  
97 */  
98 @Override  
99 public final double getF() {  
100 if (f == null) {  
101 f = Double.valueOf(getG() + getH());  
102 return f;  
103 }  
104 */  
105 /* (non-Javadoc)  
106 * @see sac.GraphState#refreshCosts()  
107 */  
108 @Override  
109 public final void refreshCosts() {  
110 g = Double.valueOf(gFunction.calculate(this));  
111 h = Double.valueOf(hFunction.calculate(this));  
112 f = Double.valueOf(getG() + getH());  
113 }  
114 static {  
115 gFunction = new GFunction();  
116 }
6.3 Implementation of game state abstraction

package sac.game;
import java.util.LinkedList;
import java.util.List;
import sac.StateImpl;

/**
 * Abstract partial implementation of GameState interface. User game state classes should extend this class.
 */
public abstract class GameStateImpl extends StateImpl implements GameState {

    /* Boolean value indicating whether it is the maximizing player turn to play now. */
    protected boolean maximizingTurnNow = true;
    
    /* List of moves along principal variation. */
    protected List<String> movesAlongPrincipalVariation = null;

    /* Boolean value indicating whether this state instance is flagged as visited during search. It is possible that
     * copies of the same state in different places of search tree will have different 'visited' flags. Some of copies
     * might be generated but cut off, or read from the transposition table. This flag is memorized only for informative
     * purposes (in particular for Graphviz functionality). */
    protected boolean visited = false;

    /* Boolean value indicating whether this state instance is flagged as read (evaluated) from transposition table. It
     * is possible that copies of the same state in different places of search tree will have different
     * 'readFromTranspositionTable' flags. This flag is memorized only for informative purposes (in particular for
     * Graphviz functionality). */
    protected boolean readFromTranspositionTable = false;

    /* Constructor for this abstract class. Initializes list for moves along principal variation as an empty list
     * (linked list). */
    public GameStateImpl() {
        movesAlongPrincipalVariation = new LinkedList<String>();
    }

    /* (non-Javadoc)
     * @see sac.game.GameState#getParent()
     */
    @Override
    public GameState getParent() {
        return (parent == null) ? null : (GameState) parent;
    }

    /* (non-Javadoc)
     * @see sac.game.GameState#GetChildren()
     */
    @Override
    @SuppressWarnings("unchecked")
    public final List<GameState> getChildren() {
        return (List<GameState>) children;
    }

    /* (non-Javadoc)
     * @see java.lang.Object#equals(java.lang.Object)
     */
    @Override
    public boolean equals(Object obj) {
        return obj instanceof GameState;
    }

    /* (non-Javadoc)
     * @see java.lang.Object#hashCode()
     */
    @Override
    public int hashCode() {
        return state.hashCode();
    }

    /* (non-Javadoc)
     * @see java.lang.Object#toString()
     */
    @Override
    public String toString() {
        return state.toString();
    }

    /* (non-Javadoc)
     * @see java.lang.Object#getClass()
     */
    @Override
    public Class<?> getClass() {
        return state.getClass();
    }
}

public class GameSimulator {
    public static void main(String[] args) {
        GAME game = new GAME();
        game.simulate();
    }
}

public interface Graphviz {
    String toGraphviz();
}
```java
package sac.game;

import java.util.List;

public final class GameState {

    public final List<GameState> getPath() {
        return (List<GameState>) super.getPath();
    }

    public final boolean isMaximizingTurnNow() {
        return maximizingTurnNow;
    }

    public final void setMaximizingTurnNow(boolean maximizingTurnNow) {
        this.maximizingTurnNow = maximizingTurnNow;
    }

    public final boolean isQuiet() {
        return true; // default implementation
    }

    public final boolean isVisited() {
        return visited;
    }

    public final void setVisited(boolean visited) {
        this.visited = visited;
    }

    public final boolean isReadFromTranspositionTable() {
        return readFromTranspositionTable;
    }

    /* (non-Javadoc)
     * @see sac.game.GameState#isReadFromTranspositionTable()
     */
}
```
package sac.game;

import java.lang.reflect.Constructor;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import sac.Identifier;
import sac.SearchAlgorithm;

/**
 * Abstract game search algorithm. Meant to be extended by actual algorithms e.g.: MIN-MAX, alpha-beta cut-offs, Scout.
 */
public abstract class GameSearchAlgorithm extends SearchAlgorithm {

    /**
     * Reference to initial state.
     */
    protected GameState initial = null;

    /**
     * Reference to currently examined state.
     */
    protected GameState current = null;

    /**
     * Map of discovered scores for moves.
     */
    protected Map<String, Double> movesScores = null;

    /**
     * Transposition table.
     */
    protected TranspositionTable transpositionTable = null;

    /**
     * Refutation table.
     */
    protected RefutationTable refutationTable = null;

    /**
     * Graph search configurator object.
     */
    

6.4 Full code of general (abstract) game search algorithm
protected GameSearchConfigurator configurator = null;

/**
 * Number of closed states (= number of calls of methods evaluateMaxState(), evaluateMinState), since last reset().
 */
protected int closedCount = 0;

/**
 * Maximum depth that was reached in the search (owing to quiescence) since last reset().
 */
protected double depthReached = 0.0;

/**
 * Boolean flag stating if stop was forced (e.g. from an outer thread).
 */
protected boolean stopForced = false;

/**
 * Creates new instance of game search algorithm.
 */
public GameSearchAlgorithm(GameState initial, GameSearchConfigurator configurator) {
    this.configurator = (configurator != null) ? configurator : new GameSearchConfigurator();
    this.initial = initial;
    this.movesScores = new HashMap<String, Double>();
    reset();
}

@override
public final void execute() {
    reset();
    startTime = System.currentTimeMillis();
    doExecute();
    endTime = System.currentTimeMillis();
}

/**
 * Actual execution of search algorithm invoked from inside of execute() method.
 */
protected void doExecute() {
    Double gameValue = null;
    if (initial.isMaximizingTurnNow())
        gameValue = evaluateMaxState(initial, Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY, 0.0, configurator.getDepthLimit());
    else
        gameValue = evaluateMinState(initial, Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY, 0.0, configurator.getDepthLimit());
    if (configurator.isTranspositionTableOn())
        transpositionTable.putOrUpdate(initial, gameValue, Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY);
    current = null;
}

protected final Double evaluateMaxState(GameState gameState, double alpha, double beta, double depth, double depthLimit) {
    if (current != null) {
        // time limit check
        if ((stopForced) || (configurator.getTimeLimit() < Long.MAX_VALUE)) {
            return null;
        }
    }

    if (gameState.isMaximizingTurnNow())
        return evaluateMaxState(gameState, alpha, beta, depth + 1, depthLimit);
    else
        return evaluateMinState(gameState, alpha, beta, depth + 1, depthLimit);
}

protected final Double evaluateMinState(GameState gameState, double alpha, double beta, double depth, double depthLimit) {
    if (current != null) {
        // time limit check
        if ((stopForced) || (configurator.getTimeLimit() < Long.MAX_VALUE)) {
            return null;
        }
    }

    if (gameState.isMaximizingTurnNow())
        return evaluateMaxState(gameState, alpha, beta, depth + 1, depthLimit);
    else
        return evaluateMinState(gameState, alpha, beta, depth + 1, depthLimit);
}
```java
long currentTime = System.currentTimeMillis();
if (currentTime - startTime > configurator.getTimeLimit()) {
    endTime = System.currentTimeMillis();
    return null;
}
closedCount++;
current = gameState;
    gameState.setVisited(true);
return doEvaluateMaxState(gameState, alpha, beta, depth, depthLimit);
}

/**
 * Wrapping method around doEvaluateMinState(...).
 *
 * @param gameState given game state
 * @param alpha lower bound on game value known for given game state
 * @param beta upper bound on game value known for given game state
 * @param depth current depth
 * @param depthLimit depth limit
 * @return calculated game value (or null if time limit is reached)
 */
protected final Double evaluateMinState(GameState gameState, double alpha, double beta, double depth, double depthLimit) {
    // time limit check
    if ((stopForced) || (configurator.getTimeLimit() < Long.MAX_VALUE)) {
        long currentTime = System.currentTimeMillis();
        if (currentTime - startTime > configurator.getTimeLimit()) {
            endTime = System.currentTimeMillis();
            return null;
        }
    }
    closedCount++;
current = gameState;
    gameState.setVisited(true);
return doEvaluateMinState(gameState, alpha, beta, depth, depthLimit);
}

/**
 * Evaluates given game state associated with the maximizing player.
 *
 * @param gameState given game state
 * @param alpha lower bound on game value known for given game state
 * @param beta upper bound on game value known for given game state
 * @param depth current depth
 * @param depthLimit depth limit
 * @return calculated game value (or null if time limit is reached)
 */
public abstract Double doEvaluateMaxState(GameState gameState, double alpha, double beta, double depth, double depthLimit);

/**
 * Evaluates given game state associated with the minimizing player.
 *
 * @param gameState given game state
 * @param alpha lower bound on game value known for given game state
 * @param beta upper bound on game value known for given game state
 * @param depth current depth
 * @param depthLimit depth limit
 * @return calculated game value (or null if time limit is reached)
 */
public abstract Double doEvaluateMinState(GameState gameState, double alpha, double beta, double depth, double depthLimit);

/**
 * Resets this algorithm. I.e. resets initial state (cuts off its children, if they exist), clears moves scores,
 * clears transposition table. Refutation table remains not cleared due to its purpose - the progressive search.
 */
@SuppressWarnings("unchecked")
protected void reset() {
    stopForced = false;
    // identifiers
    Identifier.setType(configurator.getIdentifierType()); // in case, it changed since last
    // execute() call
    // root of the tree resetting
    if (initial != null) {
```
initial.refresh();
initial.setParent(null);
initial.setMoveName(" ");
initial.setDepth(0.0);
initial.getChildren().clear();
recalculateHIfLarge(initial);
}

// clearing moves scores
movesScores.clear();

// transposition table
try {
    Constructor<TranspositionTable> constructor = (Constructor<TranspositionTable>) Class.forName(configurator.
        getTranspositionTableClassName());
    transpositionTable = (TranspositionTable) constructor.newInstance();
} catch (Exception e) {
    transpositionTable = new TranspositionTableAsHashMap();
e.printStackTrace();
}

// refutation table
if (refutationTable == null) {
    try {
        Constructor<RefutationTable> constructor = (Constructor<RefutationTable>) Class.forName(configurator.
            getRefutationTableClassName());
        refutationTable = (RefutationTable) constructor.newInstance(configurator.getRefutationTableDepthLimit());
    } catch (Exception e) {
        refutationTable = new RefutationTableAsHashMap();
e.printStackTrace();
    }
} else
    refutationTable.reset();

closedCount = 0;
depthReached = 0.0;
}

/**
 * Returns the map with scores of moves.
 */
public Map<String, Double> getMovesScores() {
    return movesScores;
}

/**
 * Returns the best move, taking into account the player to move first (maximizing or minimizing) at initial state.
 * If more than one best move (with equal value) is available returns the first one that occurred in the scores map.
 * It is possible that this method returns a null if time limit has been reached and no move score has been
 * evaluated so far.
 */
public final String getFirstBestMove() {
    double factor = ((initial != null) && (initial.isMaximizingTurnNow())) ? 1 : -1;
    String bestMove = null;
double bestMoveValue = Double.NEGATIVE_INFINITY * factor;
    for (String move : movesScores.keySet()) {
        double value = movesScores.get(move);
        if (value * factor > bestMoveValue * factor) {
            bestMove = move;
            bestMoveValue = value;
        }
    }
    if ((bestMove == null) && (!movesScores.isEmpty())) // all moves are plus or minus
        // infinities, taking the first
    bestMove = movesScores.keySet().iterator().next();
    return bestMove;
}
public final List<String> getBestMoves() {
    double factor = ((initial != null) && (initial.isMaximizingTurnNow())) ? 1 : -1;
    double bestMoveValue = Double.NEGATIVE_INFINITY * factor;
    for (String move : movesScores.keySet()) {
        double value = movesScores.get(move);
        if (value * factor > bestMoveValue * factor) {
            bestMoveValue = value;
        }
    }
    List<String> bestMoves = new ArrayList<String> ;
    for (String move : movesScores.keySet()) {
        if (value * factor == bestMoveValue * factor) {
            bestMoves.add(move);
        }
    }
    return bestMoves;
}

/**
 * Returns reference to initial game state.
 * @return reference to initial game state
 */
public final GameState getInitial() {
    return initial;
}

/**
 * Sets reference to initial game state.
 * @param initial reference to initial game state
 */
public final void setInitial(GameState initial) {
    this.initial = initial;
}

/**
 * Returns reference to currently examined state.
 * @return reference to currently examined state
 */
public GameState getCurrent() {
    return current;
}

/**
 * Returns reference to transposition table.
 * @return reference to transposition table
 */
public final TranspositionTable getTranspositionTable() {
    return transpositionTable;
}

/**
 * Returns reference to refutation table.
 * @return reference to refutation table
 */
public final RefutationTable getRefutationTable() {
    return refutationTable;
}
** Gets reference to configurator object.
* @return reference to configurator object
*/
public final GameSearchConfigurator getConfigurator() {
  return configurator;
}

/**
* Sets reference to configurator object.
* @param configurator reference to configurator object
*/
public final void setConfigurator(GameSearchConfigurator configurator) {
  this.configurator = configurator;
}

@Override
public final int getClosedStatesCount() {
  return closedCount;
}

/**
* Returns the maximum depth that was reached in the search (owing to quiescence) since last reset().
* @return maximum depth that was reached in the search
*/
public final double getDepthReached() {
  return depthReached;
}

/**
* Calls parent.generateChildren() method and increments depths of children by 0.5 with respect to their parent.
* @param parent reference to game state object for which children should be generated
*/
protected final List<GameState> generateChildrenWrapper(GameState parent) {
  List<GameState> children = parent.generateChildren();
  for (GameState child : children) {
    child.setParent(parent);
    child.setDepth(parent.getDepth() + 0.5);
    if (configurator.isParentsMemorizingChildren())
      parent.getChildren().add(child);
    recalculateHIfLarge(child);
  }
  return children;
}

/**
* Recalculates heuristic value for given state if its absolute value is greater than H_SMALLEST_INFINITY constant. The
* recalculation is done according to the formula: \( h = \text{signum}(h) \times H_SMALLEST_INFINITY \times (1.0 + 1.0 / \text{state.getDepth()}) \).
* @param state reference to game state for which the recalculation is executed
*/
protected final static void recalculateHIfLarge(GameState state) {
  double h = state.getH();
  if (Math.abs(h) > GameState.H_SMALLEST_INFINITY) {
    h = Math.signum(h) * GameState.H_SMALLEST_INFINITY * (1.0 + 1.0 / state.getDepth());
    state.setH(h);
  }
}

/**
* Returns a boolean flag showing if given state is a terminal (leaf).
* @param gameState reference to game state
* @param depth current depth
* @param depthLimit depth limit
* @return boolean flag showing if given state is a terminal (leaf)
*/
public final boolean isGameStateTerminal(GameState gameState, double depth, double depthLimit) {
  depthReached = Math.max(depthReached, depth);
  return depthReached >= depthLimit;
}
6.5 MST implementation using Kruskal’s algorithm for the TSP solver
public MinimumSpanningTree(MinimumSpanningTree toCopy) {
    connections = new ArrayList<Connection>();
    cost = toCopy.cost;
}

/**
 * Creates a minimum spanning tree from a collection of places. The Kruskal's algorithm is performed inside this
 * constructor.
 */
public MinimumSpanningTree(SortedSet<Place> places) {
    connections = new ArrayList<Connection>();
    cost = 0.0;

    SortedSet<Connection> orderedConnections = new TreeSet<Connection>(new ConnectionCostComparator());

    // building sorted set of all valid connections
    for (Place place : places) {
        for (Connection connection : place.getConnections()) {
            if (!places.contains(connection.getPlace1()))
                continue;
            if (!places.contains(connection.getPlace2()))
                continue;
            orderedConnections.add(connection);
        }
    }

    SortedSet<Place> placesInsideMSP = new TreeSet<Place>();

    // adding first connection
    Connection firstConnection = orderedConnections.first();
    connections.add(firstConnection);
    cost += firstConnection.getCost();
    placesInsideMSP.add(firstConnection.getPlace1());
    placesInsideMSP.add(firstConnection.getPlace2());
    orderedConnections.remove(firstConnection);

    while (placesInsideMSP.size() < places.size()) {
        Iterator<Connection> iterator = orderedConnections.iterator();
        while (iterator.hasNext()) {
            Connection connection = iterator.next();
            boolean place1InsideMSP = placesInsideMSP.contains(connection.getPlace1());
            boolean place2InsideMSP = placesInsideMSP.contains(connection.getPlace2());
            if ((((place1InsideMSP) && !place2InsideMSP) || (!place1InsideMSP) && place2InsideMSP)) {
                connections.add(connection);
                cost += connection.getCost();
                placesInsideMSP.add(connection.getPlace1());
                placesInsideMSP.add(connection.getPlace2());
                iterator.remove();
                break;
            }
        }
    }

    /**
     * Returns the connections.
     */
    public List<Connection> getConnections() {

    }
    return connections;

/**
 * Returns the cost of this minimum spanning tree.
 * @return cost of this minimum spanning tree
 */
public double getCost() {
    return cost;
}

/**
 * Explicitly sets the cost of this minimum spanning tree. This method may be useful in case some simple
 * modifications from outside (e.g. removal of some connection) are made on this tree, and its cost is the
 * recalculated in a control manner.
 * @param cost new cost to be set
 */
public void setCost(double cost) {
    this.cost = cost;
}

/**
 * Checks if a given place is associated with only one connection in this minimum spanning tree. If so the
 * connection is returned, otherwise a null is returned.
 * @param place reference to given place
 * @return single connection or null if more than one connection exists
 */
public Connection isPlaceWithSingleConnection(Place place) {
    int occurrences = 0;
    Connection singleConnection = null;
    for (Connection connection : connections) {
        if (connection.getPlace1().equals(place) || connection.getPlace2().equals(place)) {
            occurrences++;
            singleConnection = connection;
        }
    }
    return null;
}

return singleConnection;
}
Index

algorithm
A*, 17, 37, 38, 43, 48, 49, 58, 64, 68, 76, 82, 159, 162, 165
IDA*, 39, 41, 68
alpha-beta pruning, 104, 107, 115, 118, 124
fail-hard, 105
fail-soft, 105
Best-first search, 36, 37, 49, 89, 91, 97, 99, 159–162
Breadth-first search, 13, 33, 34
Depth-first search, 33, 34, 89
Dijkstra’s, 11, 17, 35, 37, 43
Kruskal’s, 31, 75
Min-Max, 102, 103, 105, 115, 116, 143
Prims’s, 31
Quiescence, 103, 142, 143, 147, 151
Scout, 105, 108, 115, 120, 124, 143, 151

binary heap, 36, 51
checkers, 134
Closed set, 28, 54, 97, 132, 156, 165
depth limit, 22, 103

Graphviz, 6, 12, 22, 28, 67, 84, 143, 155
hash map, 36, 51, 103, 123, 124, 128, 160, 167
heuristics, 5, 13, 17, 21, 29, 35, 36, 38, 40, 61, 64
admissible, 38, 89
for checkers, 142
for sliding puzzle
Manhattan, 61
Manhattan + linear conflicts, 56, 62
misplaced tiles, 61, 68
for sudoku
empty cells, 88, 94
sum of remaining possibilities, 88, 94
for TSP
based on MST, 69
monotonous, 39

minimax, 102
minimum spanning tree, 31, 69, 75, 184
monitor, 6, 47, 82, 165

Nim, 151
Open set, 33, 50
principal variation, 22, 110, 115, 128, 143, 147
principal variation search, 128
priority queue, 36, 51
red-black tree, 53, 124, 128
refutation table, 114, 120, 127
search horizon, 22, 103
sliding puzzle, 55
state
along principal variation, 22, 110, 115, 143
147
cutoff, 22, 128
goal, 13
initial, 13, 22
on path to solution, 13
quiet, 110, 115, 142
solution, 4, 5, 11, 13, 29, 36, 46, 76, 83, 97
terminal, 5, 22, 72, 103
visited, 13, 22
INDEX

win, 5
statistics, 6, 82, 158
sudoku, 83
  minimum, 96, 97

theorem
  Knuth-Moore, 105, 115, 126
  minimax, 102
  time control, 114
transposition table, 22, 110, 114, 118, 123, 128
Traveling Salesman Problem, 69