Algorithms for searching graphs and game trees

Przemysław Klęsk

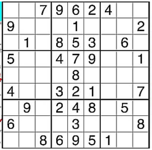
Department of Artificial Intelligence and Applied Methematics
Faculty of Computer Science and Information Technology
West Pomeranian University of Szczecin, Poland
pklesk@zut.edu.pl

Table of contents

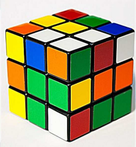
Table of contents

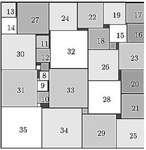
Graphs within AI

- Graphs: geographical, mazes, navigational . . . but also — puzzles, riddles that can be represented as a graph, e.g.: sudoku, sliding puzzle, Rubik's cube, solitaires, Rummikub, packing problems, etc.



		7	9	6	2	4		
9				1				2
	1		8	5	3		6	
5			4	7	9			1
				8				
4			3	2	1			7
	9		2	4	8		5	
6				3				8
		8	6	9	5	1		





- Vertices — states of a puzzle, edges — possible moves / manipulations transitting a given state into another.
- **Problem of searching graph:**
  Given an initial graph state, the task is to find a path of transitions (if exists) to a goal state. Additionally, if stated in the task, the goal is to find the minimum path.

Searching — what is needed?

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)?*
4. ***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).*

Table of contents

Open and closed sets

- Most graph searching algorithms can be formulated with use of two data sets, named by convention as: ***Open*** and ***Closed***.
- At any moment of an operating algorithm, the *Closed* set contains states that have been already visited, the *Open* set contains states that await to be visited.
- Awaiting states have been generated as descendants (graph neighbors) of states visited earlier.
- *Open* and *Closed* sets can be implemented using various data structures depending on the wanted algorithmic behaviour and efficiency.
- What kind of algorithm we deal with is essentially decided by the order according to which states are polled (taken and removed) from *Open* set for further processing.

Table of contents

Breadth-first and depth-first search

- Should be treated as uninformed graph traversal techniques rather than searching algorithms (a search process should be guided by some useful information).
- It is difficult to point original authors. Charles Pierre Trémaux (1859–1882), a French mathematician, is suspected to be the first one to study DFS as a technique for solving mazes.
- Depth is understood as the number of transitions (hops) over edges, starting from an initial state, needed to reach a given state.
- BFS algorithm must visit all states awaiting at depth d before it is allowed to visit states at depth $d + 1$.
- DFS algorithm must not visit any state at depth d as long as there exist awaiting states at detph $d + 1$.

Breadth-first and depth-first search

```
procedure BreadthFirstSearch($s_0$)                              ▷ initial state $s_0$
    $Closed := \emptyset$                                        ▷ empty set of visited states
    set reference from $s_0$ to its parent to null
    $Open := \{s_0\}$                                            ▷ queue of states to be visited
    while $Open \neq \emptyset$ do
        remove from $Open$ the state $s$ with the smallest depth ▷ 'poll' operation
        if $s$ is the goal state then return $s$                 ▷ solution found
        generate descendants $\{t\}$ of $s$                      ▷ set their parent pointers to $s$
        for all $t$ do
            if $t \notin Closed$ and $t \notin Open$ then add $t$ to $Open$
        add $s$ to $Closed$
    return null                                                  ▷ no solution found
```

```
procedure DepthFirstSearch($s_0$)                                ▷ initial state $s_0$
    $Closed := \emptyset$                                        ▷ empty set of visited states
    set reference from $s_0$ to its parent to null
    $Open := \{s_0\}$                                            ▷ queue of states to be visited
    while $Open \neq \emptyset$ do
        remove from $Open$ the state $s$ with the largest depth  ▷ 'poll' operation
        if $s$ is the goal state then return $s$                 ▷ solution found
        generate descendants $\{t\}$ of $s$                      ▷ set their parent pointers to $s$
        for all $t$ do
            if $t \notin Closed$ and $t \notin Open$ then add $t$ to $Open$
        add $s$ to $Closed$
    return null                                                  ▷ no solution found
```

Breadth-first and depth-first search

- We assume that states are aware of their depth (programistically: states are equipped with and integer depth field).
- When descendant *t* of *s* is being created, the depth of *t* becomes equal to the depth of *s* plus 1.
- Because of the expected order of states visiting, *Open* set can be implemented as:
  FIFO collection (ordinary queue) for BFS,
  LIFO collection (stack) for DFS.
- For graphs with size known in advance (known number of states / vertices) the *Closed* set can be implemented as an ordinary array of visits.
- For large graphs with size unknown in advance, more advanced data structures are needed to implement *Closed* set, e.g. hash map or red-black tree.

Table of contents

Dijkstra's algorithm

- **E. Dijkstra (1959), "A note on two problems in connexion with graphs",** ***Numerische Mathematik*****, 1(1), 269–271.**

  [http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf]
- Algorithm for finding ***shortest paths*** in a graph.
- Often formulated in a way allowing to find *all* shortest paths between a selected source vertex and *all* remaining vertices — single-source all shortest paths.
- Can be modified to stop earlier, i.e. when a particular goal vertex is reached.
- Notation:

  $g(s)$ — exact "travelled" cost from s_0 to s,
  $\Delta(s \to t)$ — cost of transition from s to t.

Dijkstra's algorithm

```
procedure Dijkstra($s_0$)                                          ▹ initial state $s_0$
    $Closed := \emptyset$                                          ▹ empty set of visited states
    $g(s_0) := 0$                                                  ▹ cost travelled from start
    set reference from $s_0$ to its parent to null
    $Open := \{s_0\}$                                              ▹ queue of states to be visited
    while $Open \neq \emptyset$ do
        remove from $Open$ the state $s$ with the smallest $g(s)$  ▹ 'poll' operation
        if $s$ is the goal state then return $s$                   ▹ solution found
        generate descendants $\{t\}$ of $s$
        for all $t$ do
            if $t \in Closed$ then continue                        ▹ $t$ already visited
            $g(t) := g(s) + \Delta(s \rightarrow t)$
            set reference from $t$ to its parent to $s$
            if $t \notin Open$ then
                add $t$ to $Open$
            else
                if new $g(t)$ is smaller than value known so far then
                    replace $t$ in $Open$ with the new one
                    update position of $t$ in $Open$
        add $s$ to $Closed$
    return null                                                    ▹ no solution found
```

Dijkstra's algorithm

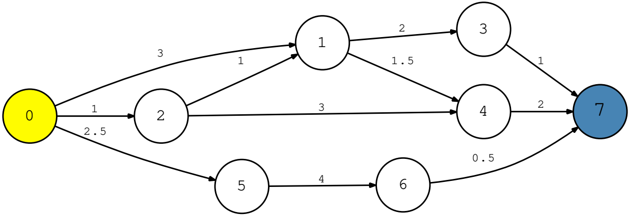
- Convenient data structure for *Open*: priority queue (binary heap, MIN-oriented).
- Complexity of poll operation (polling minimum state from *Open*): $O(\log n)$.
- Complexity of adding a state to *Open*: optimisitic $O(\log n)$, pessimistic $O(n)$, amortized $O(\log n)$.
- Complexity of replacing a state in *Open*: $O(n)$ for standard priority queue.
- Convenient data structure for *Closed* (especially when graph size unknown): hash map.
- Complexity of checking if a state present in *Closed*: $O(1)$.
- Complexity of adding a state to *Closed*: optimistic $O(1)$, pessimistic $O(n)$, amortized $O(1)$.

Dijkstra's algorithm

- ***Proof of path optimality:*** With respect to the returned state s^*, all states s residing in *Open* at stop moment have costs $g(s) \geqslant g(s^*)$. Also, it is known that all states reachable from s_0 using paths with costs smaller than $g(s^*)$ have already been processed since the cheapest state is polled in each step of main loop. ■
- Considered to be uninformed search algorithm.
- If $\Delta(s \to t) = 1$ for any s, t being neighbors then Dijkstra's algorithm is equivalent to BFS.

Example 1

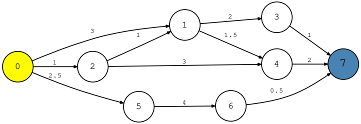
- Initial vertex: 0. Goal vertex: 7.



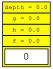
- BFS — order of visits: $(0, 1, 2, 5, 3, 4, 6, 7)$, path: $(0, 1, 3, 7)$, cost: 6.0.
- DFS — order of visits: $(0, 1, 3, 7)$, path: $(0, 1, 3, 7)$, cost: 6.0.
- Dijkstra's algo. — order of visits: $(0, 2, 1, 5, 4, 3, 7)$, path: $(0, 2, 1, 3, 7)$, cost: 5.0.

Example 1 — BFS

- Initial vertex: 0. Goal vertex: 7.



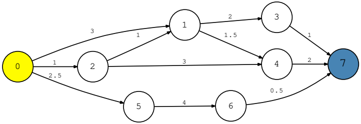
- BFS — search graph on successive steps:



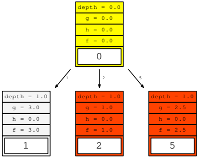
[Results generated by ***SaC*** library: `https://pklesk.github.io/sac`, illustrations owing to: ***Graphviz*** `https://www.graphviz.org`.]

Example 1 — BFS

- Initial vertex: 0. Goal vertex: 7.



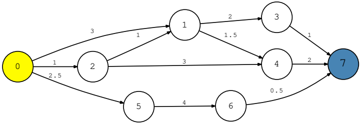
- BFS — search graph on successive steps:



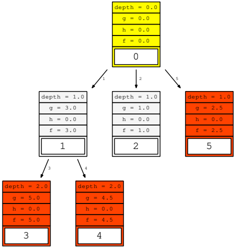
[Results generated by ***SaC*** library: `https://pklesk.github.io/sac`, illustrations owing to: ***Graphviz*** `https://www.graphviz.org`.]

Example 1 — BFS

- Initial vertex: 0. Goal vertex: 7.



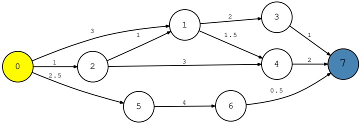
- BFS — search graph on successive steps:



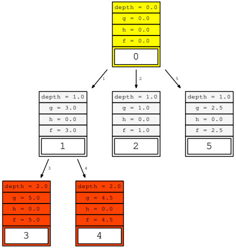
[Results generated by ***SaC*** library: `https://pklesk.github.io/sac`, illustrations owing to: ***Graphviz*** `https://www.graphviz.org`.]

Example 1 — BFS

- Initial vertex: 0. Goal vertex: 7.



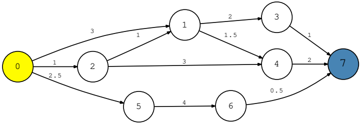
- BFS — search graph on successive steps:



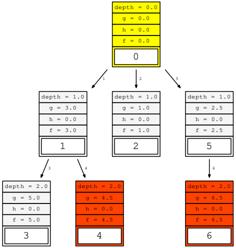
[Results generated by ***SaC*** library: `https://pklesk.github.io/sac`, illustrations owing to: ***Graphviz*** `https://www.graphviz.org`.]

Example 1 — BFS

- Initial vertex: 0. Goal vertex: 7.



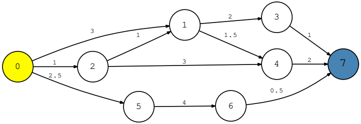
- BFS — search graph on successive steps:



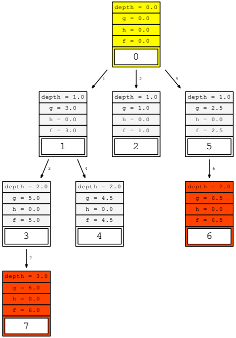
[Results generated by ***SaC*** library: `https://pklesk.github.io/sac`, illustrations owing to: ***Graphviz*** `https://www.graphviz.org`.]

Example 1 — BFS

- Initial vertex: 0. Goal vertex: 7.



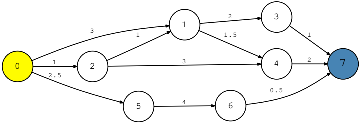
- BFS — search graph on successive steps:



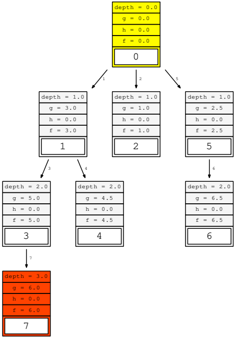
[Results generated by ***SaC*** library: `https://pklesk.github.io/sac`, illustrations owing to: ***Graphviz*** `https://www.graphviz.org`.]

Example 1 — BFS

- Initial vertex: 0. Goal vertex: 7.



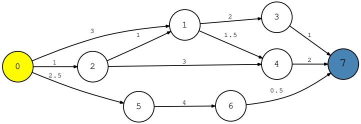
- BFS — search graph on successive steps:



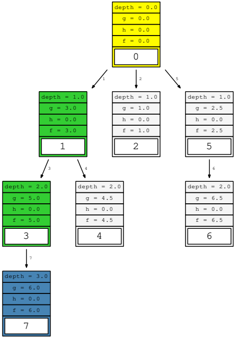
[Results generated by ***SaC*** library: `https://pklesk.github.io/sac`, illustrations owing to: ***Graphviz*** `https://www.graphviz.org`.]

Example 1 — BFS

- Initial vertex: 0. Goal vertex: 7.



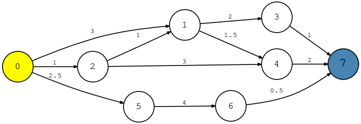
- BFS — search graph on successive steps:



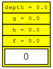
[Results generated by ***SaC*** library: `https://pklesk.github.io/sac`, illustrations owing to: ***Graphviz*** `https://www.graphviz.org`.]

Example 1 — DFS

- Initial vertex: 0. Goal vertex: 7.

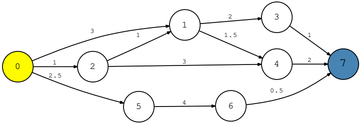


- DFS — search graph on successive steps:

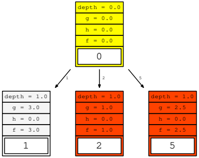


Example 1 — DFS

- Initial vertex: 0. Goal vertex: 7.

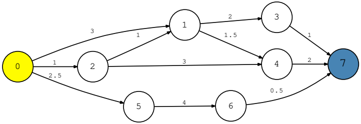


- DFS — search graph on successive steps:

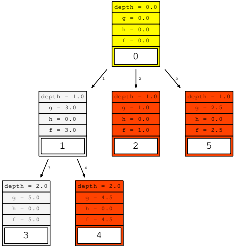


Example 1 — DFS

- Initial vertex: 0. Goal vertex: 7.

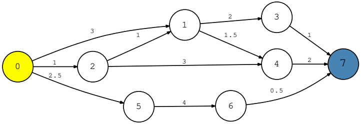


- DFS — search graph on successive steps:

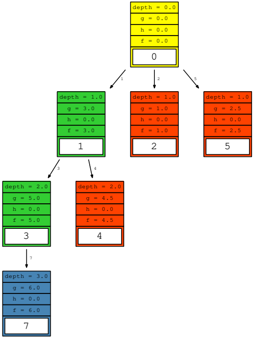


Example 1 — DFS

- Initial vertex: 0. Goal vertex: 7.

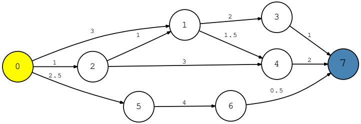


- DFS — search graph on successive steps:

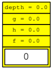


Example 1 — Dijkstra's algorithm

- Initial vertex: 0. Goal vertex: 7.

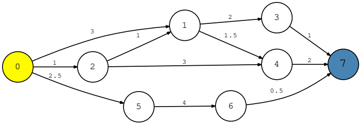


- Dijkstra's algorithm — search graph on successive steps:

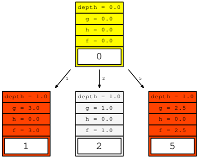


Example 1 — Dijkstra’s algorithm

- Initial vertex: 0. Goal vertex: 7.

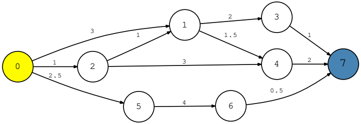


- Dijkstra’s algorithm — search graph on successive steps:

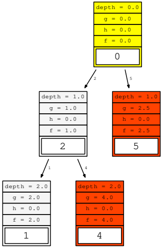


Example 1 — Dijkstra's algorithm

- Initial vertex: 0. Goal vertex: 7.

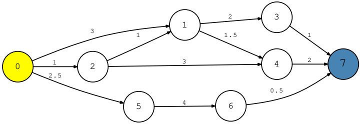


- Dijkstra's algorithm — search graph on successive steps:

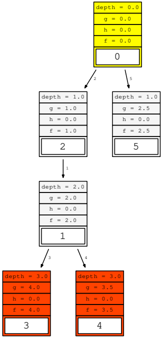


Example 1 — Dijkstra's algorithm

- Initial vertex: 0. Goal vertex: 7.

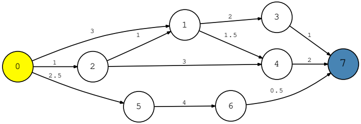


- Dijkstra's algorithm — search graph on successive steps:

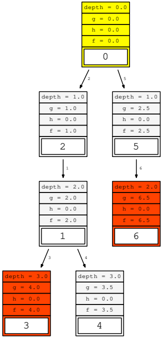


Example 1 — Dijkstra’s algorithm

- Initial vertex: 0. Goal vertex: 7.

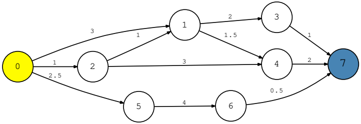


- Dijkstra’s algorithm — search graph on successive steps:

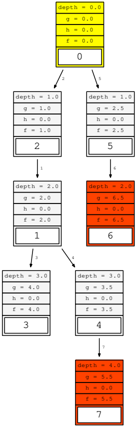


Example 1 — Dijkstra's algorithm

- Initial vertex: 0. Goal vertex: 7.

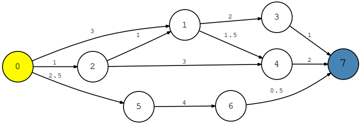


- Dijkstra's algorithm — search graph on successive steps:

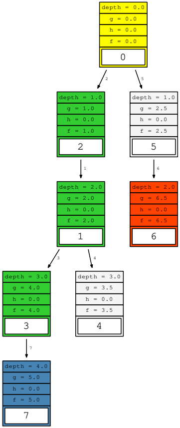


Example 1 — Dijkstra's algorithm

- Initial vertex: 0. Goal vertex: 7.

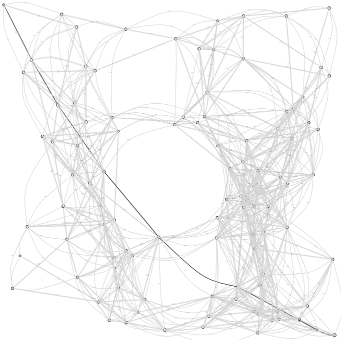


- Dijkstra's algorithm — search graph on successive steps:



"Geographical" graph

- Graph generated synthetically: 100 vertices, 10% of possible edges.
- Vertices placed randomly within $[0,1] \times [0,1]$ square, except for initial and goal state — $(0,0)$ and $(1,1)$, respectively.
- Edge weights (transition costs) proportional to Euclidean distances with small random perturbations.



- Shortest path $(0, 18, 14, 64, 60, 10, 5, 99)$ with cost ≈ 149.52.
- ***Dijkstra's algorithm*** visits *all* states before finding the shortest path for this graph.

Table of contents

Best-first search

- **J. Pearl (1984), *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley.**

  [http://mat.uab.cat/ alseda/MasterOpt/Judea_Pearl-Heuristics_Intelligent_Search_Strategies_for_Computer_Problem_Solving.pdf]
- The most promising ("best") state is always expanded in first order.
- Quantitative assessment of how promising s is, made by means of a heuristic function $h(s)$ — informed search.
- Various possibilities to construct $h(s)$:
  - based on static information contained in s,
  - based on information collected along the way from s_0 to s,
  - based on general knowledge about the problem and about properties of the goal state (solution).
- By convention $h(s) \geqslant 0$. Small values suggest closeness to solution.
- *Best-first* approach is focused on achieving the solution fast, via *any* path. One does not care about path minimization (the notion of path cost does not exist).
- Data structures (again): *Open* (priority queue), *Closed* (hash map).

Best-first search

```
procedure BestFirstSearch($s_0$)                                   ▷ initial $s_0$
    $Closed := \emptyset$                                          ▷ empty set of visited states
    calculate $h(s_0)$                                             ▷ heuristic according to provided recipe
    set reference from $s_0$ to its parent to null
    $Open := \{s_0\}$                                              ▷ queue of states to be visited
    while $Open \neq \emptyset$ do
        remove from $Open$ the state $s$ with smallest $h(s)$      ▷ 'poll' operation
        if $s$ is the goal state then return $s$                   ▷ solution found
        generate descendants $\{t\}$ of $s$
        for all $t$ do
            if $t \in Closed$ then continue                        ▷ $t$ already visited
            calculate $h(t)$
            set reference from $t$ to its parent to $s$
            if $t \notin Open$ then
                add $t$ to $Open$
            else
                if new $h(t)$ is smaller than value known so far then   ▷ e.g. when $h(s)$ depends on information along path
                    replace $t$ in $Open$ with the new one
                    update position of $t$ in $Open$
        add $s$ to $Closed$
    return null                                                    ▷ no solution found
```

Sudoku — example 1

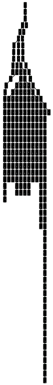
- Level hard:

*	*	*	*	*	*	8	*	*
8	*	*	7	*	1	*	4	*
*	4	*	*	2	*	*	3	*
3	7	4	*	*	*	9	*	*
*	*	*	*	3	*	*	*	*
*	*	5	*	*	*	3	2	1
*	1	*	*	6	*	*	5	*
*	5	*	8	*	2	*	*	6
*	8	*	*	*	*	*	*	*

↓

7	6	1	5	4	3	2	8	9
8	3	2	7	9	1	6	4	5
5	4	9	6	2	8	1	3	7
3	7	4	2	1	5	9	6	8
1	2	8	9	3	6	5	7	4
6	9	5	4	8	7	3	2	1
4	1	7	3	6	9	8	5	2
9	5	3	8	7	2	4	1	6
2	8	6	1	5	4	7	9	3

- ***Best-first search + "empty cells" heuristic***, descendants at "minimum cell", closed states: 222, open states: 14



[time: 7 ms, Intel Xeon CPU E3-1505M v5 2.8 GHz (boost 3.7 GHz)]

Sudoku — example 2

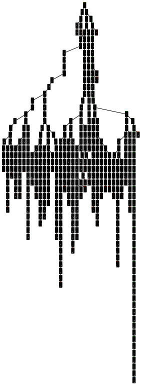
- Level hard:

*	*	*	9	*	*	*	*	2
*	5	*	1	2	3	4	*	*
*	3	*	*	*	*	1	6	*
9	*	8	*	*	*	*	*	*
*	7	*	*	*	*	*	9	*
*	*	*	*	*	*	2	*	5
*	9	1	*	*	*	*	5	*
*	*	7	4	3	9	*	2	*
4	*	*	*	*	7	*	*	*

↓

8	1	4	9	7	6	5	3	2
6	5	9	1	2	3	4	7	8
7	3	2	8	5	4	1	6	9
9	4	8	2	6	5	3	1	7
2	7	5	3	4	1	8	9	6
1	6	3	7	9	8	2	4	5
3	9	1	6	8	2	7	5	4
5	8	7	4	3	9	6	2	1
4	2	6	5	1	7	9	8	3

- ***Best-first search* + *“empty cells” heuristic***, descendants at “minimum cell”, closed states: 418, open states: 41



[time: 19 ms, Intel Xeon CPU E3-1505M v5 2.8 GHz (boost 3.7 GHz)]

Sudoku — „Qassim Hamza”

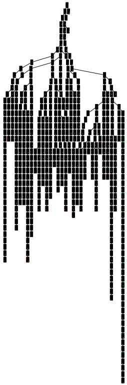
- Level very hard:

*	*	*	7	*	*	8	*	*
*	*	*	*	4	*	*	3	*
*	*	*	*	*	9	*	*	1
6	*	*	5	*	*	*	*	*
*	1	*	*	3	*	*	4	*
*	*	5	*	*	1	*	*	7
5	*	*	2	*	*	6	*	*
*	3	*	*	8	*	*	9	*
*	*	7	*	*	*	*	*	2

↓

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

- ***Best-first search + “empty cells” heuristic***, descendants at “minimum cell”, closed states: 525, open states: 40



[time: 70 ms, Intel Xeon CPU E3-1505M v5 2.8 GHz (boost 3.7 GHz)]

Sudoku — other heuristic

- Identify s with a two-dimensional array (board).
- Let $s(i,j)$ denote the contents of cell (i,j).
- Let $r(s,i,j)$ denote set of possible values (digits) for cell (i,j) once we substract from set $\{1,\ldots,9\}$ values present in i-th row, j-th column and subsquare that contains cell (i,j).
- *"Sum of remaining possibilities"* heuristic:

$$h(s) = \sum_{i,j}\sum \#r(s,i,j). \tag{1}$$

Sudoku — example 1

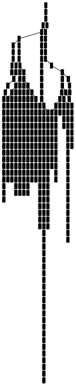
- Level hard:

*	*	*	*	*	*	8	*	*
8	*	*	7	*	1	*	4	*
*	4	*	*	2	*	*	3	*
3	7	4	*	*	*	9	*	*
*	*	*	*	3	*	*	*	*
*	*	5	*	*	*	3	2	1
*	1	*	*	6	*	*	5	*
*	5	*	8	*	2	*	*	6
*	8	*	*	*	*	*	*	*

↓

7	6	1	5	4	3	2	8	9
8	3	2	7	9	1	6	4	5
5	4	9	6	2	8	1	3	7
3	7	4	2	1	5	9	6	8
1	2	8	9	3	6	5	7	4
6	9	5	4	8	7	3	2	1
4	1	7	3	6	9	8	5	2
9	5	3	8	7	2	4	1	6
2	8	6	1	5	4	7	9	3

- ***Best-first search + "sum of remaining possibilities" heuristic***,
  descendants at "minimum cell",
  closed states: 304, open states: 20



[time: 16 ms, Intel Xeon CPU E3-1505M v5 2.8 GHz (boost 3.7 GHz)]

Sudoku — example 2

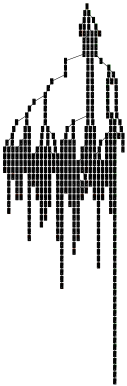
- Level hard:

*	*	*	9	*	*	*	*	2
*	5	*	1	2	3	4	*	*
*	3	*	*	*	*	1	6	*
9	*	8	*	*	*	*	*	*
*	7	*	*	*	*	*	9	*
*	*	*	*	*	*	2	*	5
*	9	1	*	*	*	*	5	*
*	*	7	4	3	9	*	2	*
4	*	*	*	*	7	*	*	*

↓

8	1	4	9	7	6	5	3	2
6	5	9	1	2	3	4	7	8
7	3	2	8	5	4	1	6	9
9	4	8	2	6	5	3	1	7
2	7	5	3	4	1	8	9	6
1	6	3	7	9	8	2	4	5
3	9	1	6	8	2	7	5	4
5	8	7	4	3	9	6	2	1
4	2	6	5	1	7	9	8	3

- ***Best-first search + "sum of remaining possibilities" heuristic***,
  descendants at "minimum cell",
  closed states: 381, open states: 37



[time: 16 ms, Intel Xeon CPU E3-1505M v5 2.8 GHz (boost 3.7 GHz)]

Sudoku — „Qassim Hamza”

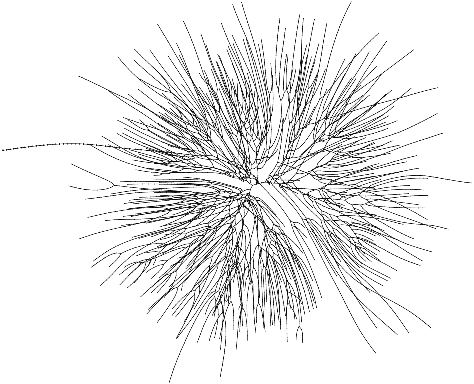
- Level very hard:

*	*	*	7	*	*	8	*	*
*	*	*	*	4	*	*	3	*
*	*	*	*	*	9	*	*	1
6	*	*	5	*	*	*	*	*
*	1	*	*	3	*	*	4	*
*	*	5	*	*	1	*	*	7
5	*	*	2	*	*	6	*	*
*	3	*	*	8	*	*	9	*
*	*	7	*	*	*	*	*	2

↓

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

- ***Best-first search + “sum of remaining possibilities” heuristic***,
  descendants at “minimum cell”,
  closed states: 5 267, open states: 452



[time: 208 ms, Intel Xeon CPU E3-1505M v5 2.8 GHz (boost 3.7 GHz)]

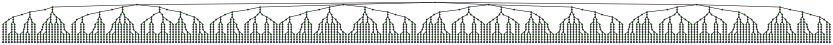
Sudoku — comparison of heuristics

- Comparison for 50 sudoku boards — source:
  [https://projecteuler.net/project/resources/p096_sudoku.txt]
- *Best-first search + "empty cells" heuristic:*
  - Average number of closed states: 166.92.
  - Average number of open states (at stop moment): 14.32.
  - Average time: 11.88 ms.
- *Best-first search + "sum of remaining possibilities" heuristic:*
  - Average number of closed states: 176.64.
  - Average number of open states (at stop moment): 15.08.
  - Average time: 13.16 ms.

All 4 × 4 sudokus

- Solutions: 288.

$$\begin{array}{|cc|cc|}\hline * & * & * & * \\ * & * & * & * \\ \hline * & * & * & * \\ * & * & * & * \\ \hline \end{array} \longrightarrow \begin{array}{|cc|cc|}\hline 4 & 2 & 3 & 1 \\ 1 & 3 & 4 & 2 \\ \hline 2 & 4 & 1 & 3 \\ 3 & 1 & 2 & 4 \\ \hline \end{array}, \begin{array}{|cc|cc|}\hline 4 & 2 & 3 & 1 \\ 1 & 3 & 4 & 2 \\ \hline 3 & 1 & 2 & 4 \\ 2 & 4 & 1 & 3 \\ \hline \end{array}, \ldots, \begin{array}{|cc|cc|}\hline 1 & 3 & 2 & 4 \\ 4 & 2 & 1 & 3 \\ \hline 3 & 1 & 4 & 2 \\ 2 & 4 & 3 & 1 \\ \hline \end{array}$$



- Closed states: 2 273, open states: 0.

Table of contents

A^*

- **P. Hart, N. Nilsson, B. Raphael (1968), "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100–107.**

  [http://ieeexplore.ieee.org/document/4082128/]
- Informally, A^* algorithm can be seen as a combination of Dijkstra's algorithm and Best-first search (or a more general form of those).
- Function deciding about the order of states polled from *Open* queue is of form:

$$f(s) = g(s) + h(s), \tag{2}$$

  where: $g(s)$ — exact travelled cost from s_0 to s, whereas $h(s)$ — heuristic estimation of cost remaining from s to the goal state.
- Since h is a heuristic then also is f.
- For shortest paths finding, h must be a so called ***admissible heuristic*** — i.e. a lower bound on remaining cost — it *must not* overestimate the true cost.
- For geographical graphs, the distance along straight line (Euclidean) is admissible heuristic for certain.
- Data structures (again): *Open* (priority queue), *Closed* (hash map).

A^*

```
procedure AStar($s_0$)                                                ▷ initial state $s_0$
    $Closed := \emptyset$                                              ▷ empty set of visited states
    $g(s_0) := 0$                                                      ▷ distance covered from start
    calculate $h(s_0)$                                                 ▷ heuristic according to provided recipe
    $f(s_0) := g(s_0) + h(s_0)$                                        ▷ sum deciding about order of Open queue
    set reference from $s_0$ to its parent to null
    $Open := \{s_0\}$                                                  ▷ queue of states to be visited
    while $Open \neq \emptyset$ do
        remove from $Open$ the state $s$ with smallest $f(s)$          ▷ 'poll' operation
        if $s$ is the goal state then return $s$                       ▷ solution found
        generate descendants $\{t\}$ of $s$
        for all $t$ do
            if $t \in Closed$ then continue                            ▷ $t$ already visited
            $g(t) := g(s) + \Delta(s \to t)$
            calculate $h(t)$
            $f(t) := g(t) + h(t)$
            set reference from $t$ to its parent to $s$
            if $t \notin Open$ then
                add $t$ to $Open$
            else
                if new $f(t)$ is smaller than value known so far then
                    replace $t$ in $Open$ with the new one
                    update position of $t$ in $Open$
        add $s$ to $Closed$
    return null                                                        ▷ no solution found
```

A^*

Theorem "path optimality for admissible heuristic"

When A^* algorithm, using an admissible heuristic, finds the goal state then the path associated with it is the shortest.

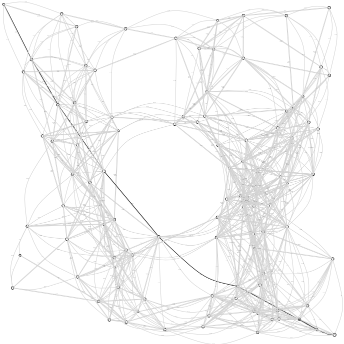
Proof: At stop moment (line 10) the aglorithm returns state s^* with travelled cost $g(s^*)$. Since s^* satisfies the stop condition then $h(s^*) = 0$. For all states s residing in *Open* at stop moment it is known that $f(s) \geqslant f(s^*)$. Among these states three cases can be distinguished. Case 1: a state s satisfies the stop condition, i.e. $h(s) = 0$, but $g(s) \geqslant g(s^*)$, because $f(s) \geqslant f(s^*)$. Case 2: a state s does not satisfy the stop condition, i.e. $h(s) > 0$, but can potentially be driven to the goal state, and currently has the cost $g(s) < g(s^*)$; knowing that $h(s)$ is a lower bound on the remaining cost and that $f(s) \geqslant f(s^*)$, then the true cost of reaching the goal state, traveling through s, must satisfy inequalities: $g(s) + \Delta(s \to s^*) \geqslant g(s) + h(s) \geqslant g(s^*)$. Case 3: a state s has $h(s) > 0$ and $g(s) \geqslant g(s^*)$ — irrelevant. ■

Monotonous heuristic

- Additional useful notion: ***monotonous heuristic***.
- We say that h is *monotonous* if for all pairs s, t (where t is a descendant of s) the following inequality holds:
$$f(s) \leqslant f(t), \tag{3}$$
which can be rewritten as
$$\begin{aligned} g(s) + h(s) &\leqslant g(t) + h(t) \quad (4) \\ h(s) &\leqslant g(t) - g(s) + h(t) \quad (5) \\ h(s) &\leqslant \Delta(s \rightarrow t) + h(t). \quad (6) \end{aligned}$$
- The above is a form of *triangle inequality*: heuristic at s must not be greater than the cost of $s \rightarrow t$ transition plus heuristic at t.
- The equality case in (6) occurs only when one travels towards the goal state along a straight line (with respect to the metric associated with the given graph).
- If a heuristic is *monotonous* than it is *admissible*.

"Geographical" graph (again)

- Graph generated synthetically: 100 vertices, 10% of possible edges.



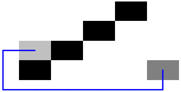
- Shortest path $(0, 18, 14, 64, 60, 10, 5, 99)$ with cost ≈ 149.52.
- ***Dijkstra's algorithm*** visits *all* states before finding the optimal path.
- ***A^* + Euclidean distance*** — closed states: 18, open states: 38 — informed search.

Good and bad (overestimating) heuristic

- Good:

$$h_1(s) = \sqrt{(s_x - s_x^*)^2 + (s_y - s_y^*)^2} \tag{7}$$

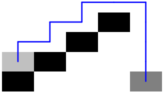
$$\text{or}\quad h_2(s) = |s_x - s_x^*| + |s_y - s_y^*| \tag{8}$$



- Bad:

$$h_3(s) = {\color{red}4}\sqrt{(s_x - s_x^*)^2 + (s_y - s_y^*)^2}$$

$$\text{or}\quad h_4(s) = {\color{red}4}\left(|s_x - s_x^*| + |s_y - s_y^*|\right)$$



Sliding puzzle

- **Sliding puzzle** ($n^2 - 1$-puzzle):
  Starting from an initial state and sliding tiles into the empty space (tile numbered as 0), the task is to reach the goal state (with numbers $\{0, 1, \ldots, n^2 - 1\}$ ordered in successive rows) in the fewest number of moves.

$$\begin{array}{ccc} \boxed{0} & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{array}$$

Sliding puzzle — heuristics

- *"misplaced tiles"* — number of tiles at incorrect positions (not counting the '0' tile).
- *"Manhattan"* — sum of distances (using Manhattan metric) of all tiles from their target positions (not counting the '0' tile).

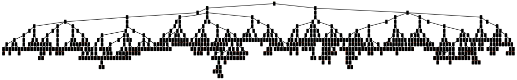
$$h(s) = \sum_{\substack{0\leqslant i,j<n\\ s(i,j)\neq 0}}\sum \Big|i - \lfloor s(i,j)/n \rfloor\Big| + \Big|j - s(i,j) \bmod n\Big|. \tag{9}$$

- *"Manhattan + linear conflicts"* — as above + counting additional 2 moves implied by each linear conflict — see:
  O. Hansson, A.E. Mayer, M.M. Yung (1985), "Generating Admissible Heuristics by Criticizing Solutions to Relaxed Models", ***Columbia University Computer Science Technical Reports*****, https://doi.org/10.7916/D89Z9CW3.**

  [https://www.researchgate.net/profile/Moti_Yung/publication...]
- Are the above heurisitcs monotonous?

Sliding puzzle

- Search graphs for initial state $(0,3,2;4,7,8;1,5,6)$ and different heuristics.
- ***A^* + "misplaced tiles"***



[states: 672, time: 34 ms, Intel Xeon CPU E3-1505M v5 2.8 GHz (boost 3.7 GHz)]

- ***A^* + "Manhattan"***

[states: 106, time: 21 ms, Intel Xeon CPU E3-1505M v5 2.8 GHz (boost 3.7 GHz)]

- ***A^* + "Manhattan + linear conflicts"***

[states: 78, time: 16 ms, Intel Xeon CPU E3-1505M v5 2.8 GHz (boost 3.7 GHz)]

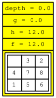
- Shortest path of length 16: $(D,R,D,R,U,L,L,D,R,U,U,L,D,R,U,L)$.

Sliding puzzle — comparison of heuristics

- Comparison for 100 random boards for $n = 3$, each board shuffled with 1000 moves.
- **A^* + *"misplaced tiles"***
  - Average number of closed states: 12 263.89.
  - Average number of open states (at stop time): 5 865.45.
  - Average time: 28.57 ms.
- **A^* + *"Manhattan"***
  - Average number of closed states: 1 024.44.
  - Average number of open states (at stop time): 588.19.
  - Average time: 8.09 ms.
- **A^* + *"Manhattan + linear conflicts"***
  - Average number of closed states: 530.14.
  - Average number of open states (at stop time): 316.81.
  - Average time: 7.37 ms.

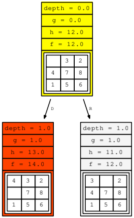
Sliding puzzle — example

- *A^** **+** ***"Manhattan*** **+** ***linear conflicts"*** — search graph in first 5 steps and the last:



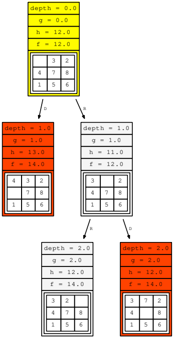
Sliding puzzle — example

- ***A^** + *“Manhattan + linear conflicts”*** — search graph in first 5 steps and the last:



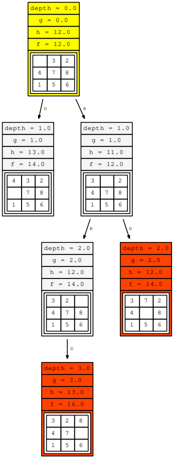
Sliding puzzle — example

- *A^** **+** *“Manhattan* **+** *linear conflicts”* — search graph in first 5 steps and the last:



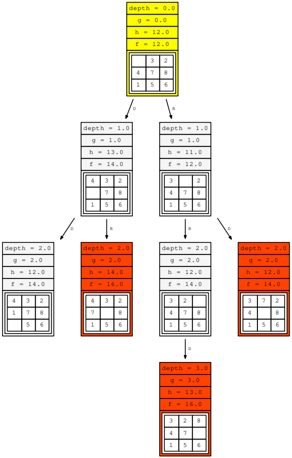
Sliding puzzle — example

- *A^** **+** ***"Manhattan*** **+** ***linear conflicts"*** — search graph in first 5 steps and the last:



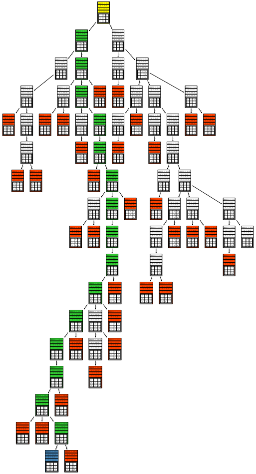
Sliding puzzle — example

- *A^** **+** *“Manhattan* **+** *linear conflicts”* — search graph in first 5 steps and the last:



Sliding puzzle — example

- *A^** **+ *"Manhattan* + *linear conflicts"*** — search graph in first 5 steps and the last:



A^* vs Best-first search

- ***A^* + "Manhattan + linear conflicts"***

[states: 78, time: 16 ms, Intel Xeon CPU E3-1505M v5 2.8 GHz (boost 3.7 GHz)]

Shortest path of length 16: $(D, R, D, R, U, L, L, D, R, U, U, L, D, R, U, L)$.

- ***Best-first search + "Manhattan + linear conflicts"***

[states: 41, time: 13 ms, Intel Xeon CPU E3-1505M v5 2.8 GHz (boost 3.7 GHz)]

Shortest path of length 18: $(R, D, D, R, U, L, U, L, D, D, R, U, U, L, D, R, U, L)$.

A^* vs Best-first search

- ***A^* + "Manhattan"***

[states: 78, time: 16 ms, Intel Xeon CPU E3-1505M v5 2.8 GHz (boost 3.7 GHz)]

Shortest path of length 16: $(D, R, D, R, U, L, L, D, R, U, U, L, D, R, U, L)$.

- ***Best-first search + "Manhattan"***

[states: 681, time: 32 ms, Intel Xeon CPU E3-1505M v5 2.8 GHz (boost 3.7 GHz)]

Shortest path of length 134: $(R, D, L, D, R, R, U, L, L, D, R, U, L, U, R, D, R, U, L, L, D, R, U, R, D, L, L, U, R, D, D, R, U, L, U, L, ..., L, L, U)$

Sliding puzzle — examples for $n = 4$

- Selected examples from **O. Hansson, A.E. Mayer, M.M. Yung (1985), "Generating Admissible Heuristics by Criticizing Solutions to Relaxed Models", *Columbia University Computer Science Technical Reports*, https://doi.org/10.7916/D89Z9CW3.**

  [https://www.researchgate.net/profile/Moti_Yung/publication...]

- *IDA^* (Iterative Deepening A^*)* — memory-economic version of A^*, but computationally expensive.

no.	initial state	path length	IDA^* closed	IDA^* time [s]	A^* states closed and open	A^* time [s]
85	$4, 7, 13, 10, 1, 2, 9, 6, 12, 8, 14, 5, 3, 0, 11, 15$	44	$1.5 \cdot 10^7$	12.3	$1.7 \cdot 10^5, \ 1.6 \cdot 10^5$	0.9
5	$4, 7, 14, 13, 10, 3, 9, 12, 11, 5, 6, 15, 1, 2, 8, 0$	56	$2.6 \cdot 10^7$	20.4	$1.6 \cdot 10^6, \ 1.4 \cdot 10^6$	11.7
2	$13, 5, 4, 10, 9, 12, 8, 14, 2, 3, 7, 1, 0, 15, 11, 6$	55	$3.8 \cdot 10^7$	31.2	$2.6 \cdot 10^6, \ 2.1 \cdot 10^6$	26.9
54	$12, 11, 0, 8, 10, 2, 13, 15, 5, 4, 7, 3, 6, 9, 14, 1$	56	$1.9 \cdot 10^8$	150.5	brak RAM (2 GB) przy: $3.1 \cdot 10^6, \ 2.5 \cdot 10^6$	—
1	$14, 13, 15, 7, 11, 12, 9, 5, 6, 0, 2, 1, 4, 8, 10, 3$	57	$2.5 \cdot 10^8$	212.3	brak RAM (2 GB) przy: $3.4 \cdot 10^6, \ 2.8 \cdot 10^6$	—

[time: 7 ms, Intel Xeon CPU E3-1505M v5 2.8 GHz (boost 3.7 GHz)]

A^* — concluding remarks

- When $h(s) = 0$ for all s then: A^* = Dijkstra's algorithm.
- When $g(s) = 0$ for all s then: A^* = Best-first search.
- The better information carried by h the less work A^* has to do.
- Monotonicity of a heuristic implies three consequences:
  1. solution found is optimal (shortest path),
  2. algorithm itself is optimal with respect to h, i.e. no other algorithm, using h, cannot visit fewer states than A^* (differences only due to tie-breaking),
  3. *let h^* denote a perfect heuristic representing the true distance / cost to the goal, then an algorithm using h^* is perfect too — visits the smallest number of states possible (hence originally two names distinguished: A and A^*)*.

Table of contents

IDA*

- **R. Korf (1985), "Depth-first Iterative Deepening: An Optimal Admissible Tree Search", *Artificial Intelligence*, 27, 97–109.**
  [https://pdfs.semanticscholar.org/7eaf/535ca7f8d1e920e092483d11efb989982f19.pdf]
- For some suitably large problems, A^* may exhaust RAM memory (very large *Open* and *Closed* sets).
- *IDA** can be seen as memory-economic version of A^*.
- *IDA** does *not* keep evidence of visited states — no *Closed* set.
- *IDA** keeps in memory only the states that are on currently studied path.
- The algorithm can be formulated recursively (with no *Open* set) or traditionally with a main loop (then only a small *Open* set occurs).

IDA* — sketch

- The algorithm uses $h(s_0)$ value to establish an initial search horizon:

$$H = f(s_0) = 0 + h(s_0). \tag{10}$$

- Then, it studies various paths outgoing from s_0 (e.g. with a *depth-first* approach).
- If the goal state is reached within H, then it is returned.
- Any state "touched" outside H is not expanded further, but the information about cost observed for that state is useful to establish the next search horizon:

$$H' = \min_{\{s\colon g(s)>H\}} f(s). \tag{11}$$

- Once all the paths within H are exhausted, the horizon is deepened i.e. $H := H'$ and whole process is repeated.

*IDA*** recursively

```
procedure RecursiveIterativeDeepeningAStar($s_0$)        ▷ initial state $s_0$
    $g(s_0) := 0$                                         ▷ cost travelled from start
    calculate $h(s_0)$                                    ▷ heuristic according to provided recipe
    $f(s_0) := g(s_0) + h(s_0)$
    set reference from $s_0$ to its parent to null
    $H := f(s_0)$                                         ▷ initial search horizon
    while true do
        $(s, H') :=$Search$(s_0, H)$
        if $s \neq$ null then return $s$                  ▷ solution found
        if $H' = \infty$ then return null                 ▷ no solution found
        $H := H'$
```

```
procedure Search($s$, $H$)
    if $f(s) > H$ then return $\big(\text{null}, f(s)\big)$
    if $s$ is the target state then return $\big(s, g(s)\big)$     ▷ solution found
    $H' := \infty$
    generate descendants $\{t\}$ of $s$
    for all $t$ do
        $g(t) := g(s) + \Delta(s \to t)$
        $f(t) := g(t) + h(t)$
        $(u, H'') :=$ Search$(t, H)$
        if $u \neq$ null then return $(u, g(u))$                  ▷ solution found
        $H' := \min\{H', H''\}$                                   ▷ deepening the horizon
    return (null, $H'$)
```

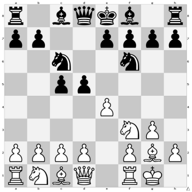
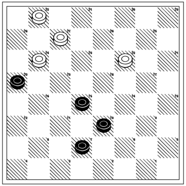
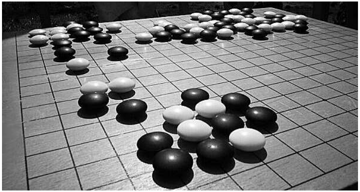
*IDA** non-recursively

```
procedure IteratativeDeepeningAStar($s_0$)                                      ▹ initial state $s_0$
    $g(s_0) := 0$                                                                ▹ cost travelled from start
    calculate $h(s_0)$                                                           ▹ heuristic according to provided recipe
    $f(s_0) := g(s_0) + h(s_0)$
    set reference from $s_0$ to its parent to null
    $Open := \{s_0\}$                                                            ▹ queue of states to be visited
    $H := f(s_0), H' := \infty$                                                  ▹ initial and next search horizons
    while $Open \neq \emptyset$ do
        remove from $Open$ the state $s$ with smallest $f(s)$                     ▹ 'poll' operation
        if $g(s) > H$ then
            $H' := \min\{H', f(s)\}$
            if $Open = \emptyset$ then
                $H := H', H' := \infty, Open := \{s_0\}$                         ▹ deepening the horizon
            continue
        if $s$ is the target state then return $s$                               ▹ solution found
        generate descendants $\{t\}$ of $s$
        for all $t$ do
            $g(t) := g(s) + \Delta(s \rightarrow t)$
            calculate $h(t)$
            $f(t) := g(t) + h(t)$
            set reference from $t$ to its parent to $s$
            if $t \notin Open$ then
                add $t$ to $Open$
            else
                if new $f(t)$ is smaller than value known so far then
                    replace $t$ in $Open$ with the new one
                    update position of $t$ in $Open$
    return null                                                                  ▹ no solution found
```

Table of contents

Games

- Commonly, two-person games considered: chess, checkers, GO, ...

- Game — a situation of conflict, where players have opposite goals, and where we have clearly defined rules.
- **Problem of searching game tree:**
  Given a game position (in particular, an initial position), the task is to provide ***quantitative evaluations (scores)*** for particular ***moves*** at current player's disposal. An evaluation should represent exact or approximate ***payoff*** for the player if he chooses a given move, assuming the optimal counter-play by opponent.

Games — initial chess tree fragment

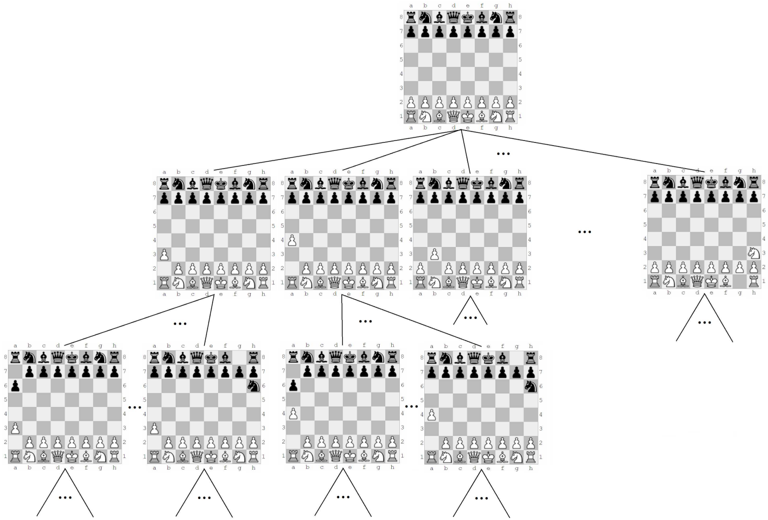
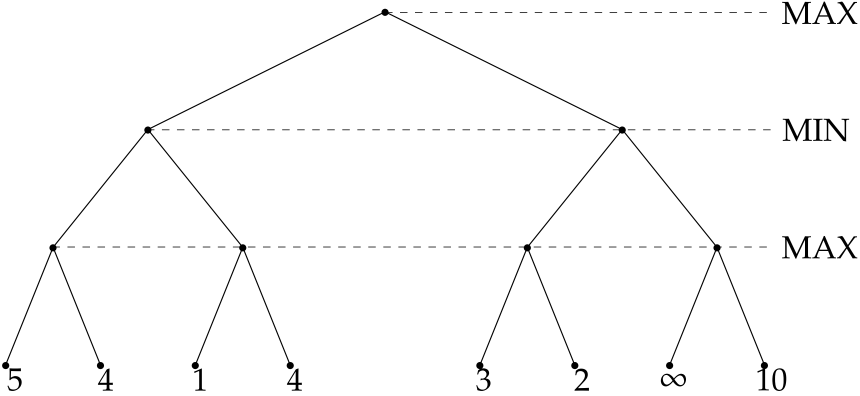


Table of contents

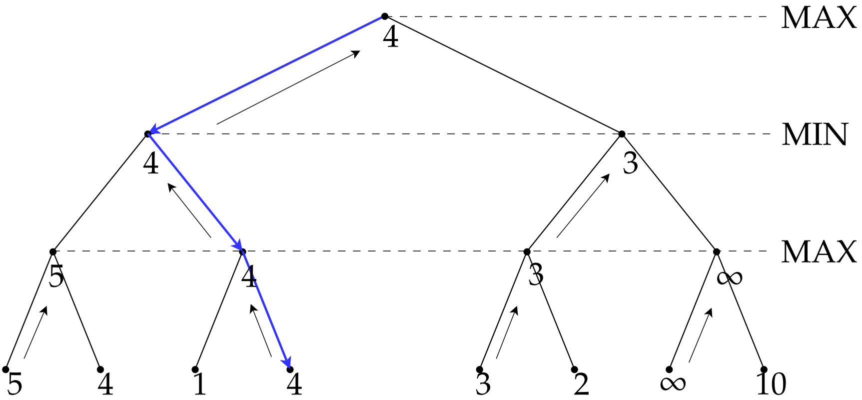
Min-Max algorithm

- Sketch: given an initial position, a tree of game states is expanded up to the imposed depth. Terminal positions (leaves) are associated with ***quantitative evaluations***. Tree traversal follows, and evaluations are propagated up the tree. In effect, direct descendants of the initial state are evaluated too (and so are possible initial moves).
- Position evaluation function is a heuristic function working according to people's knowledge and intuition about the game.
- E.g. for chess: difference between materialistic value of white and black pieces.
- Commonly, each player is named as: maximizing or minimazing player.
- The win of minimizing player is represented by $-\infty$.
- The win of maximizing player is represented by $+\infty$.
- When game tree is suitably small (or when studied is a strict endgame) and true terminal states are reached, then possible values of leaves are: $-\infty$, $+\infty$, 0 (tie). In that case, heuristic evaluation is not needed.

Min-Max — illustration



Min-Max — illustration



Min-Max — notions and notation

- half-move (or ply) — a move by one of players; moving by one tree level is by convention counted as $\pm\frac{1}{2}$; 2 half-moves (made by both players) are treated as a whole move.
- branching factor — average or constant number of moves for a player in the given game; commonly, denoted by b (e.g. for chess $b \approx 40$ in the middle game).
- search horizon — imposed number of tree levels to be studied; commonly, denoted by D.
- horizon effect — general flaw of all minimax procedures implied by the limited search depth; this phenomenon means that a state residing just outside the horizon can significantly differ in its evaluation (with respect to parent) and e.g. turn out catastrophic for a player, even though its ancesteors seemed attractive (or vice-versa).
- Quiescence — helper technique that partially mitigates horizon effect; it consists in expanding states on the horizon frontier (or behind it) until so-called *quiet positions* are reached (e.g. with no possible captures).

Min-Max algorithm

```
procedure MMEvaluateMaxState($s, d, D$)
    if IsTerminal($s, d, D$) then return $h(s)$        ▹ $h(s)$ — heuristic evaluation of position
    $v := -\infty$
    generate descendants $\{t\}$ of $s$
    for all $t$ do
        $w$ :=MMEvaluateMinState($t, d + \frac{1}{2}, D$)
        if $s$ is the root state then memorize $w$ as the score of $s \to t$ move
        $v := \max\{v, w\}$
    return $v$
```

```
procedure MMEvaluateMinState($s, d, D$)
    if IsTerminal($s, d, D$) then return $h(s)$        ▹ $h(s)$ — heuristic evaluation of position
    $v := \infty$
    generate descendants $\{t\}$ of $s$
    for all $t$ do
        $w$ :=MMEvaluateMaxState($t, d + \frac{1}{2}, D$)
        if $s$ is the root state then memorize $w$ as the score of $s \to t$ move
        $v := \min\{v, w\}$
    return $v$
```

Min-Max — stop points

- IsTerminal(s, d, D) — a routine method that checks if we are at stop point, implemented accoring to the game rules.
- Commonly, any of the following condition should be satisfied:
  - $d \geqslant D$ and s is *quiet*,
  - $h(s) = \pm\infty$ — s is the *win* state,
  - $h(s) \neq \pm\infty$, but s is a *draw* state by the rules (e.g. for chess: stalemate, perpetual check, three-time repetition of position).

Chess — position evaluation

- Example of a function proposed by C. Shannon (1949):

$$f(s) = 200(K_s - K_s') + 9(Q_s - Q_s') + 5(R_s - R_s') + 3(B_s - B_s' + N_s - N_s') + 1(P - P') \quad \textit{(materialistic)}$$
$$- 0.5(D_s - D_s' + S_s - S_s' + I_s - I_s') + 0.1(M_s - M_s'), \quad \textit{(positional)} \qquad (12)$$

  where K, Q, R, B, N, P denote counts of: kings, queens, rooks, bishiops, knights and pawns;
  D, S, I denote counts of pawns that are: doubled, blocked, isolated;
  M denotes mobility (number of moves at disposal);
  the $'$ (prime symbol) denotes same features for the opposing side.
- Commonly in contemporary chess engines, evaluations expressed in so-called centipawns.
- One pawn = 100 centipawns. Smallest positional advantage is worth 1 centipawn.
- Elements taken into account:
  - control over board center,
  - activeness of pieces (and their “connectivity”),
  - pawn structure,
  - king safety,
  - pawns close to promotion,
  - space,
  - . . .
- Popular are also approaches self-tuning a parametric evaluation function (e.g. based on genetic algorithms).

Checkers — position evaluation

- Example of materialistic and positional heuristic (M. Bożykowski, 2009):

$$\begin{aligned} f(s) = {} & 13(P_s - P'_s) + 85(K_s - K'_s) \quad (\textit{materialistic}) \\ & + 6(T_s - T'_s) + 1(I_s - I'_s) - 1(F_s - F'_s), \quad (\textit{positional}) \end{aligned} \tag{13}$$

  where: P, K denote counts of pawns and kings, respectively; T, I, F denote counts of pawns that are: 1 square away from promotion, incapturable, frozen.

- Example of materialistic and row-oriented heuristic (M. Bożykowski, 2009):

$$f(s) = \sum_{i=1}^{9} w_i \left(P_s(i) - P'_s(11-i)\right) + 12(K_s - K'_s), \tag{14}$$

  where: $P_s(i)$ denotes the number of pawns in i-th board row (for a board with 100 squares); integer weights tuned genetically: $w = (2, 1, 2, 2, 2, 2, 1, 3, 6)$;

Min-Max — computational complexity

- R_d — number of states that must be visited in a tree with d levels, in order to get to know the value of given state — sum of geometric sequence.
- Recursive approach (useful for analysis of more advanced tree-search algorithms):

$$\begin{aligned} R_0 &= 1; \\ R_d &= 1 + bR_{d-1}. \end{aligned} \tag{15}$$

- Expansion:

$$\begin{aligned} R_d &= 1 + bR_{d-1} \\ &= 1 + b(1 + bR_{d-2}) = 1 + b + b^2 R_{d-2} \\ &\;\vdots \\ &= 1 + b + b^2 + \cdots + b^d R_{d-d} = \frac{b^{d+1} - 1}{b - 1} \end{aligned} \tag{16}$$

$$< \frac{b^{d+1}}{b-1} = \underbrace{\frac{b}{b-1}}_{\leqslant 2} \frac{1}{b} b^{d+1} \leqslant 2b^d \sim O(b^d) \tag{17}$$

- Simplfied scheme: $O(b \cdot b \cdots b)$ — d-times b.

Table of contents

α–β pruning

- Many independent discoverers: **(Samuel, 1952)**, **(Edwards and Hart, 1963)**, **(Brudno, 1963)**, **(Newell and Simon, 1958; 1976)**.
- Exact analysis: **D. Knuth, R. Moore, R. (1975), "An analysis of alpha-beta pruning", *Artificial Intelligence*, 6(4), 293–326.**

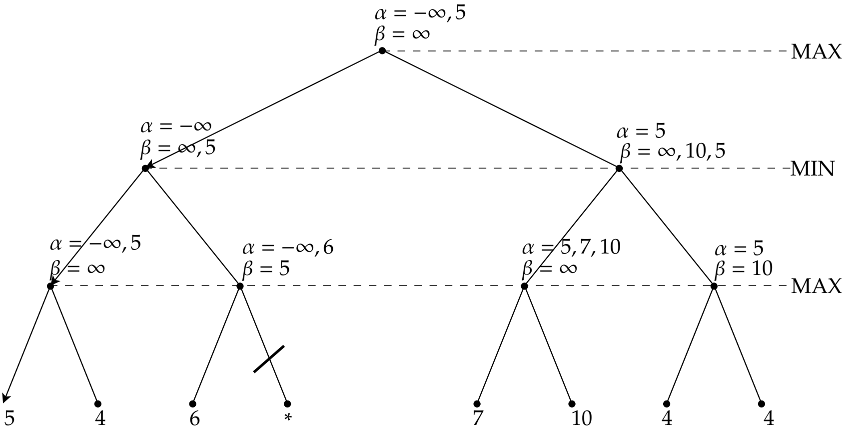
  [https://pdfs.semanticscholar.org/dce2/6118156e5bc287bca2465a62e75af39c7e85.pdf]
- Belongs to branch and bound class of algorithms.
- At operation time two values are tracked along the tree:
  α — guaranteed so far pay-off for the maximizing player,
  β — guaranteed so far pay-off for the minimizing player.
- On invocation for the root state, one imposes $\alpha = -\infty$, $\beta = \infty$.
- Children states (and their subtrees) analyzed as long as $\alpha < \beta$.
- Whenever $\alpha \geqslant \beta$ we stop to analyze subsequent children (and their subtrees) — they shall not affect result for the whole tree, they are effect of non-optimal play by one of players.
- $\alpha > \beta$ is a logical contradiction; equality case can be additionally included to pruning because it does not introduce an improvement of result.
- Despite tree reductions, α–β pruning algorithm yields exactly same results (move evaluations) as Min-Max.

α–β pruning

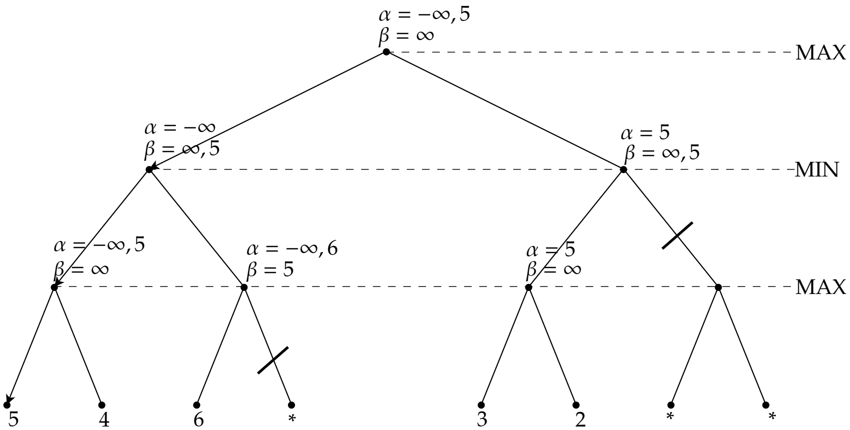
```
procedure AlphaBetaEvaluateMaxState($s, d, D, \alpha, \beta$)
    if IsTerminal($s, d, D$) then return $h(s)$                          ▹ $h(s)$ — heuristic position evaluation
    generate descendants $\{t\}$ of $s$
    for all $t$ do
        $v$ :=AlphaBetaEvaluateMinState($t, d + \frac{1}{2}, D, \alpha, \beta$)
        if $s$ is the root state then memorize $v$ as the score of $s \to t$ move
        $\alpha := \max\{\alpha, v\}$
        if $\alpha \geq \beta$ then return $\alpha$                      ▹ cut-off (!) — subsequent $t$ states not studied
    return $\alpha$
```

```
procedure AlphaBetaEvaluateMinState($s, d, D, \alpha, \beta$)
    if IsTerminal($s, d, D$) then return $h(s)$                          ▹ $h(s)$ — heuristic position evaluation
    generate descendants $\{t\}$ of $s$
    for all $t$ do
        $v$ :=AlphaBetaEvaluateMaxState($t, d + \frac{1}{2}, D, \alpha, \beta$)
        if $s$ is the root state then memorize $v$ as the score of $s \to t$ move
        $\beta := \min\{\beta, v\}$
        if $\alpha \geq \beta$ then return $\beta$                       ▹ cut-off (!) — subsequent $t$ states not studied
    return $\beta$
```

α–β pruning — example 1



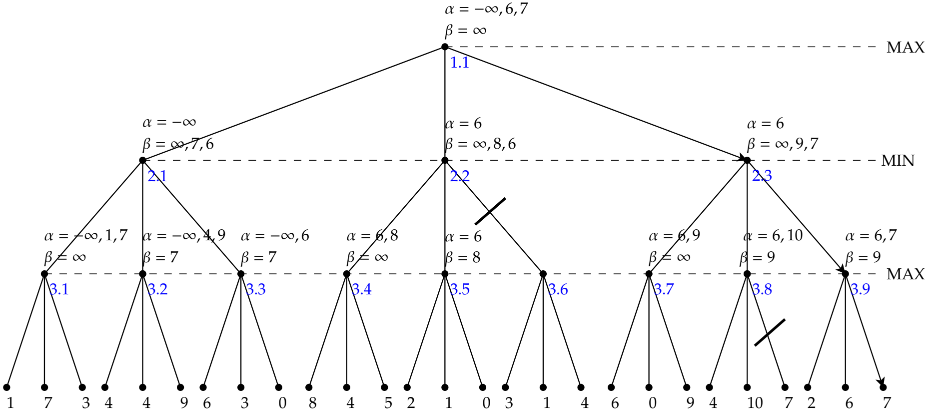
α–β pruning — example 2



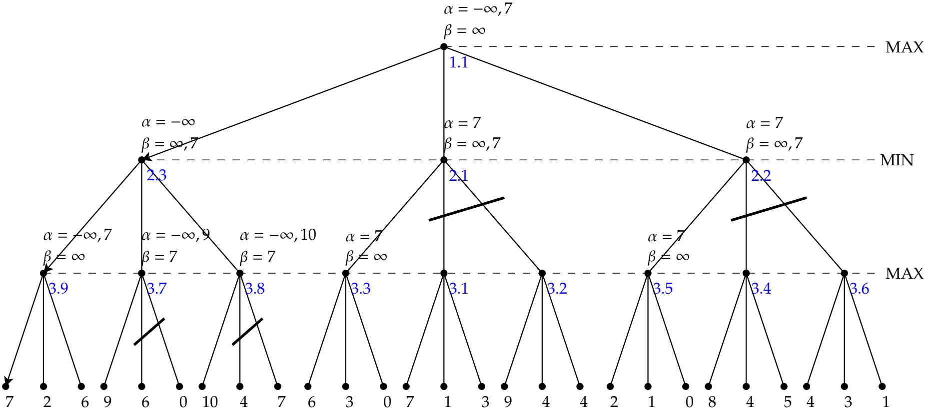
α–β pruning — complexity

- Computational complexity depends on the *order* of visiting descendants (children states).
- It is favourable when cut-off causing descendents are closer to the beginning of the list.
- There exist some helper techniques attempting to suitably order descendants and thereby increase cut-off frequency (but in general, optimal order is not known in advance),
- In pessimistic case (for d levels): $O\left(b^d\right)$.
- In optimistic case (for d levels): $O\left(b^{d/2}\right)$.

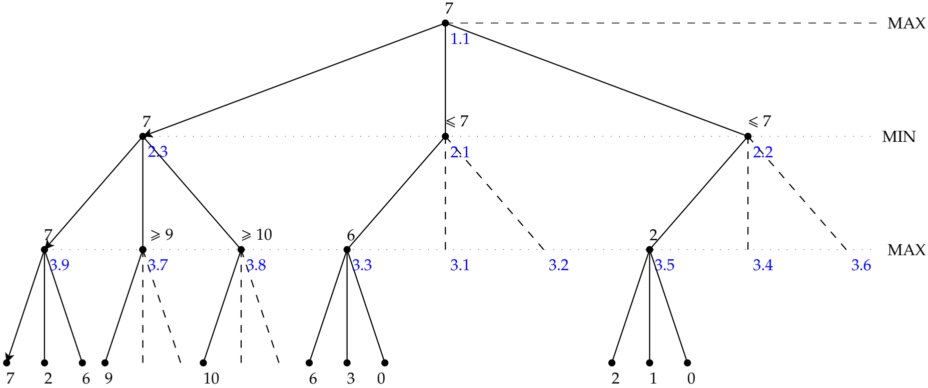
α–β pruning — example 3



α–β pruning — example 3a



α–β pruning — example 3b



α–β pruning — optimistic complexity

- We know either exact value of a state, or bound (lower or upper) on that value.
- To establish the exact value, it suffices (in optimistic case) to know:
  exact value for 1 child and bounds for $b-1$ remaining children.
- To establish a bound, it suffices (in optimistic case) to know:
  exact value for 1 child.
- R_d — minimum number of states (distant by d levels from given state) one must visit to establish the exact value.
- S_d — minimum number of states (distant by d levels from given state) one must visit to establish a bound.
- Border values: $R_0 = S_0 = 1$.
- Recursions:

$$R_d = R_{d-1} + (b-1)S_{d-1}; \tag{18}$$

$$S_d = R_{d-1}. \tag{19}$$

- By joining them, we obtain:

$$R_d = R_{d-1} + (b-1)R_{d-2}. \tag{20}$$

- For the example from previous slide: $R_3 = b^2 + b - 1 = 11$.

α–β pruning — optimistic complexity

- Estimation of optimistic number of states:

$$\begin{aligned} R_d &= R_{d-1} + (b-1)R_{d-2} \\ &= R_{d-2} + (b-1)R_{d-3} + (b-1)R_{d-2} \\ &= bR_{d-2} + (b-1)R_{d-3} \\ &< bR_{d-2} + (b-1)R_{d-2} \\ &= (2b-1)R_{d-2} \\ &< 2bR_{d-2}. \end{aligned} \tag{21}$$

- Effective branching factor is smaller than $2b$ for every 2 levels. Hence, for one level it is smaller than $\sqrt{2b}$.
- Expansion:

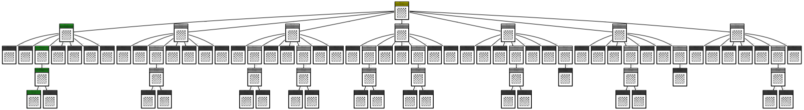
$$R_d < 2bR_{d-2} < (2b)^2R_{d-4} < (2b)^3R_{d-6} < \cdots < (2b)^kR_{d-2k} \tag{22}$$

$$< (2b)^{d/2}R_{d-2d/2} = (2b)^{d/2}R_0 \sim O(b^{d/2}) = O\left(\left(\sqrt{b}\right)^d\right) \quad \text{(treating } d \text{ as fixed)} \tag{23}$$

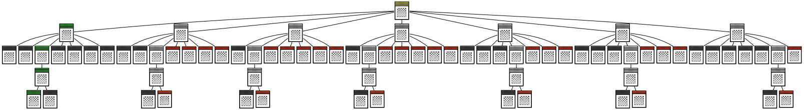
- Simplfied scheme: $O(b \cdot 1 \cdot b \cdot 1 \cdots b \cdot 1)$ — $d/2$-times b.
- In average case the complexity can be shown to be $\sim O\left(b^{3d/4}\right)$.

Initial tree fragments for checkers

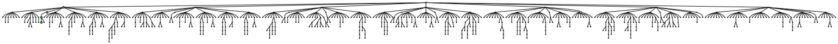
- ***Min-Max* + *Quiescence***, depth (for quiet positions): 1.0, states: 86



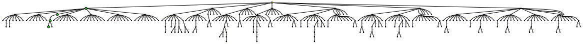
- ***α-β pruning* + *Quiescence***, depth (for quiet positions): 1.0, states 78:



- ***Min-Max* + *Quiescence***, depth (for quiet positions): 1.5, states: 693



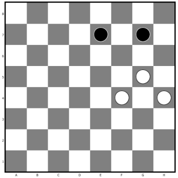
- ***α-β pruning* + *Quiescence***, depth (for quiet positions): 1.5, states: 323



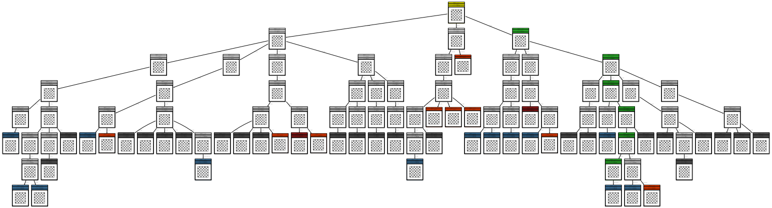
[Results generated by ***SaC*** library: `https://pklesk.github.io/sac`, illustrations owing to: ***Graphviz*** `https://www.graphviz.org`.]

Checkers endgame — example 1

- White to move and win in 4 moves:



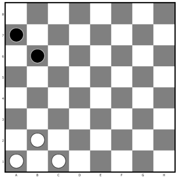
- ***α-β pruning + Quiescence***, depth: 2.5, states: 100



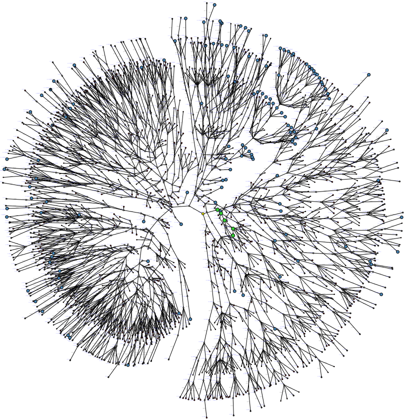
- Principal variation: ($G5 : H6, G7 : F6, F4 : G5, F6 : E5, G5 : F6, E5 : G7, H6 : F8 : D6$).

Checkers endgame — example 2

- White to move. Who wins?



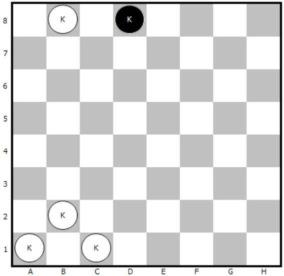
- *α-β* ***pruning*** **+** ***Quiescence***, depth: 5.5, states: 2 845



Checkers endgame — example 3

- "4 kings vs 1 king"

position:



results from ***SaC*** library:

```
Searching with sac.game.AlphaBetaPruning...
Searching done.  Time:  1789 ms.
Closed states:  54898
General depth limit:  3.5
Maximum depth reached (Quiescence):  4.5
Transposition table size:  52967
Transposition table uses:  69365
Refutation table size:  4611
Refutation table uses:  0
Moves scores:  {B2:D4=1.0985902490825263E308, B2:A3=3000.0}
Best move:  B2:D4
Principal variation:  [B2:D4, D8:A5, B8:D6, A5:E1, D6:G3,
    E1:H4, C1:G5, H4:F6:C3, A1:D4]
```

- Illustration of principal variation: [https://github.com/pklesk/sac/releases/download/1.0.3/sac-1.0.3-userguide.pdf#page=150]