Algorytmy przeszukiwania grafów i drzew gier

Przemysław Klęsk

Katedra Metod Sztucznej Inteligencji i Matematyki Stosowanej
Wydział Informatyki, ZUT w Szczecinie
pklesk@wi.zut.edu.pl

Spis treści

Spis treści

Grafy w ramach SI

- Grafy geograficzne, labirynty, nawigacje . . . ale także układanki, łamigłówki reprezentowalne jako graf, np.: sudoku, puzzle przesuwne, kostka Rubika, pasjansy, Rummikub, upakowania, itp.

- Węzły — stany układanki, krawędzie — możliwe ruchy, manipulacje przeprowadzające dany stany w inny.
- **Problem przeszukiwania grafu stanów:**
  Mając dany pewien stan początkowy, należy znaleźć ścieżkę przejść (o ile istnieje) do stanu docelowego. Jeżeli dodatkowo określono w zadaniu, należy znaleźć ścieżkę najkrótszą.

Przeszukiwanie — co potrzeba?

1. ***Generowanie potomków*** — *Jakie nowe stany (bezpośredni potomkowie) mogą być wygenerowane z danego stanu?*
2. ***Identyfikacja*** — *Jakie identyfikatory (reprezentacje napisowe lub całkowitoliczbowe) mogą zostać przypisane do stanów, tak aby ten sam stan nie był odwiedzany niepotrzebnie wielokrotnie?*
3. ***Warunek stopu*** — *Czy dany stany jest stanem końcowym? Tzn. stanem będącym rozwiązaniem (grafy) lub stanem zwycięzkim (drzewa gier)?*
4. ***Heurystyka (opcjonalnie)*** — *Oszacowanie, jak daleko dany stan jest od rozwiązania (grafy) lub ocena, w jakim stopniu dany stan reprezentuje przewagę gracza maksymalizującego lub minimalizującego (drzewa gier).*

Spis treści

Zbiór otwarty i zamknięty

- Większość algorytmów przeszukiwania grafów może być sformułowana z użyciem dwóch zbiorów danych nazywanymi zwyczajowo:
  zbiorem otwartym — ***Open***
  i *zbiorem zamkniętym* — ***Closed***.
- W danym momencie pracy algorytmu zbiór *Closed* zawiera stany, które zostały już odwiedzone, a zbiór *Open* zawiera stany oczekujące potencjalnie na bycie odwiedzonymi.
- Stany oczekujące zostają wygenerowane jako potomkowie (sąsiedzi w grafie) stanów odwiedzonych wcześniej.
- Zbiory *Open* i *Closed* można realizować (implementować) się za pomocą różnych struktur danych w zależności od pożądanego zachowania algorytmu i wydajności.
- O typie algorytmu w sposób krytyczny decyduje porządek, według którego stany są pobierane (i usuwane) ze zbioru *Open* w celu dalszego przetwarzania.

Spis treści

Breadth-first i Depth-first search

- Odpowiednio: przeszukiwanie *wszerz* i *w głąb*.
- Należy traktować bardziej jako niepoinformowane techniki przechodzenia grafu niż przeszukiwania (proces przeszukiwania powinien być kierowany pewną użyteczną informacją).
- Trudno o wskazanie autorów. Przypuszcza się, że pierwszą wersję DFS badał francuski matematyk Charles Pierre Trémaux jako technikę do rozwiązywania labiryntów.
- Przez głębokość będzie rozumiana minimalna liczba przejść po krawędziach (przeskoków, hopów) potrzebna, aby osiągnąć dany stan, licząc od stanu wyróżnionego jako początkowy.
- Algorytm BFS musi odwiedzić wszystkie oczekujące stany na głębokości d zanim może przystąpić do stanów oczekujących na głębokości $d + 1$.
- Algorytm DFS nie może odwiedzić żadnego z oczekujących stanów na głębokości d dopóki istnieją oczekujące stany o głębokościach $d + 1$.

Breadth-first i Depth-first search

```
1:  procedura BreadthFirstSearch(s_0)                          ▷ stan początkowy: s_0
2:      Closed := ∅                                            ▷ pusty zbiór odwiedzonych stanów
3:      ustaw pusty wskaźnik rodzica dla s_0
4:      Open := {s_0}                                          ▷ kolejka stanów oczekujących
5:      dopóki Open ≠ ∅ wykonaj
6:          pobierz (i usuń) z Open stan s o najmniejszej głębokości
7:          jeżeli s jest stanem końcowym to zwróć s           ▷ znaleziono rozwiązanie
8:          wygeneruj zbiór stanów {t} potomnych dla s         ▷ ustawiając wskaźniki na rodzica s
9:          dla wszystkich t wykonaj
10:             jeżeli t ∉ Closed i t ∉ Open to dodaj t do Open
11:         dodaj s to Closed
12:     zwróć wynik pusty                                     ▷ nie znaleziono rozwiązania
```

```
1:  procedura DepthFirstSearch(s_0)                            ▷ stan początkowy: s_0
2:      Closed := ∅                                            ▷ pusty zbiór odwiedzonych stanów
3:      ustaw pusty wskaźnik rodzica dla s_0
4:      Open := {s_0}                                          ▷ kolejka stanów oczekujących
5:      dopóki Open ≠ ∅ wykonaj
6:          pobierz (i usuń) z Open stan s o największej głębokości
7:          jeżeli s jest stanem końcowym to zwróć s           ▷ znaleziono rozwiązanie
8:          wygeneruj zbiór stanów {t} potomnych dla s         ▷ ustawiając wskaźniki na rodzica s
9:          dla wszystkich t wykonaj
10:             jeżeli t ∉ Closed i t ∉ Open to dodaj t do Open
11:         dodaj s to Closed
12:     zwróć wynik pusty                                     ▷ nie znaleziono rozwiązania
```

Breadth-first i Depth-first search

- Zakładamy, że każdy stan jest świadomy swojej głębokości (na poziomie programistycznym: stan wyposażony w pole całkowite przechowujące głębokość).
- Gdy dla stanu s tworzony jest potomek t, to głębokość t staje się równa głębokości s plus 1.
- Ze względu na oczekiwany porządek odwiedzania, zbiór *Open* może być realizowany jako:
  kolekcja FIFO (zwykła kolejka) dla BFS,
  kolekcja LIFO (stos) dla DFS.
- Dla grafów o znanym z góry rozmiarze (znanej liczbie stanów / węzłów) zbiór *Closed* może być realizowany jako tablica odwiedzin.
- Dla dużych grafów o nieznanym z góry rozmiarze do realizacji zbioru *Closed* potrzebna jest bardziej zaawansowana struktura danych np. mapa haszująca lub drzewo czerwono-czarne.

Spis treści

Algorytm Dijkstry

- **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]
- Algorytm do znajdowania ***najkrótszej ścieżki*** w grafie.
- Często formułowany w sposób pozwalający znaleźć *wszystkie* najkrótsze ścieżki pomiędzy wyróżnionym węzłem i *wszystkimi* pozostałymi węzłami — single-source all shortest paths.
- Można zatrzymywać wcześniej, tj. w chwili osiągnięcia stanu wyróżnionego jako końcowy.
- Notacja:
  $g(s)$ — dokładny koszt przebyty od s_0 do s,
  $\Delta(s \to t)$ — koszt przejścia z s do t.

Algorytm Dijkstry

```
1:  procedura Dijkstra(s_0)                                   ▷ stan początkowy: s_0
2:      Closed := ∅                                           ▷ pusty zbiór odwiedzonych stanów
3:      g(s_0) := 0                                           ▷ koszt przebyty od startu
4:      ustaw pusty wskaźnik rodzica dla s_0
5:      Open := {s_0}                                         ▷ kolejka stanów oczekujących
6:      dopóki Open ≠ ∅ wykonaj
7:          pobierz (i usuń) z Open stan s o najmniejszej wartości g(s)   ▷ operacja "poll"
8:          jeżeli s jest stanem końcowym to zwróć s          ▷ znaleziono rozwiązanie
9:          wygeneruj zbiór stanów {t} potomnych dla s
10:         dla wszystkich t wykonaj
11:             jeżeli t ∈ Closed to kontynuuj od kolejnej iteracji   ▷ t już odwiedzone
12:             g(t) := g(s) + Δ(s → t)
13:             ustaw wskaźnik rodzica t na s
14:             jeżeli t ∉ Open to
15:                 dodaj t do Open
16:             w przeciwnym razie
17:                 jeżeli nowy koszt g(t) jest mniejszy niż znany dotychczas to
18:                     zastąp t w Open nowym egzemplarzem (aktualnie badanym)
19:                     uaktualnij pozycję t w Open
20:         dodaj s do Closed
21:     zwróć wynik pusty                                    ▷ nie znaleziono rozwiązania
```

Algorytm Dijkstry

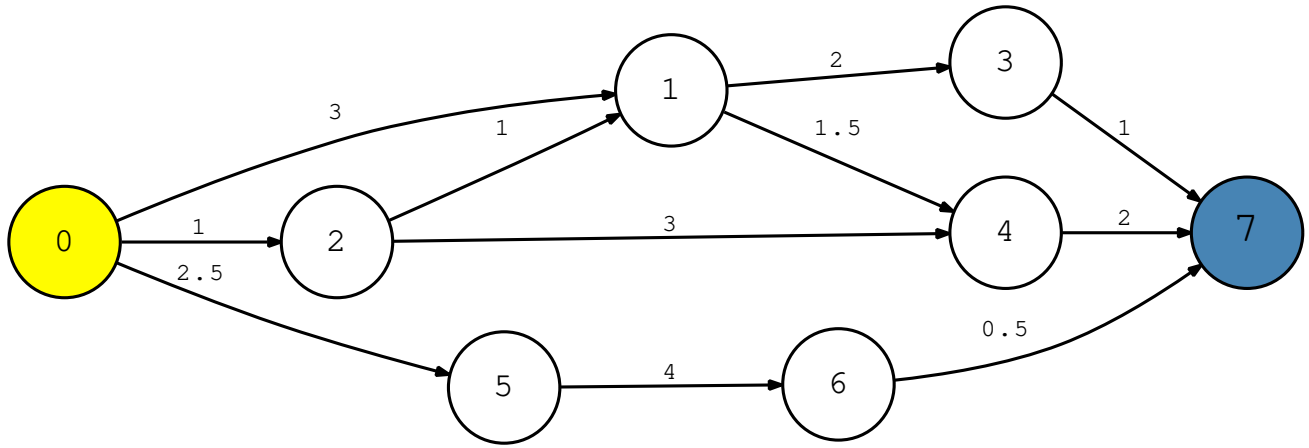
- Wygodna struktura danych dla *Open*: kolejka priorytetowa (kopiec binarny MIN).
- Złożoność pobierania (stanu minimalnego) z *Open*: $O(\log n)$.
- Złożoność dodawania stanów do *Open*: optymistyczna $O(\log n)$, pesymistyczna $O(n)$, zamortyzowana $O(\log n)$.
- Złożoność podmiany stanu w *Open*: $O(n)$ dla standardowej kolejki priorytetowej.
- Wygodna struktura danych dla *Closed* (szczególnie gdy rozmiar grafu nieznany): mapa haszująca.
- Złożoność sprawdzenia czy stan jest obecny w *Closed*: $O(1)$.
- Złożoność dodawania stanów do *Closed*: optymistyczna $O(1)$, pesymistyczna $O(n)$, zamortyzowana $O(1)$.

Algorytm Dijkstry

- ***Dowód optymalności ścieżki:*** Dla zwracanego stanu s^* stany s przebywające w *Open* w chwili stopu mają koszty $g(s) \geqslant g(s^*)$. Jednocześnie wiadomo, że wszystkie stany osiągalne z s_0 ścieżkami o koszcie mniejszym niż $g(s^*)$ zostały już zbadane ze względu na pobieranie stanu o najmniejszym koszcie w każdym kroku pętli. ■
- Uznawany za algorytm przeszukiwania niepoinformowanego.
- Jeżeli $\Delta(s \to t) = 1$ dla dowolnych s, t będących sąsiadami to równoważny przeszukiwaniu wszerz (BFS).

Przykład 1

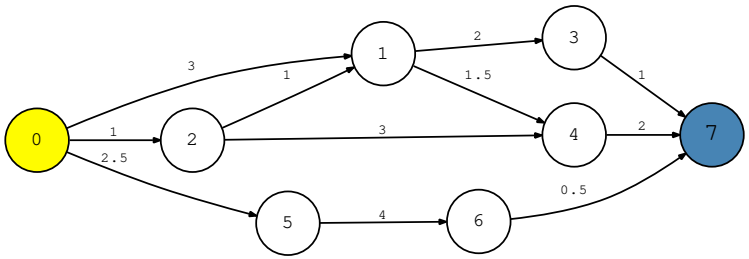
- Węzeł 0 początkowy. Węzeł 7 końcowy.



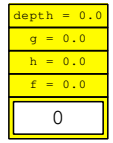
- BFS — porządek odwiedzania: $(0, 1, 2, 5, 3, 4, 6, 7)$, ścieżka: $(0, 1, 3, 7)$, koszt: 6.0.
- DFS — porządek odwiedzania: $(0, 1, 3, 7)$, znaleziona ścieżka: $(0, 1, 3, 7)$, koszt: 6.0.
- a. Dijkstry — porządek odwiedzania: $(0, 2, 1, 5, 4, 3, 7)$, ścieżka: $(0, 2, 1, 3, 7)$, koszt: 5.0.

Przykład 1 — BFS

- Węzeł 0 początkowy. Węzeł 7 końcowy.



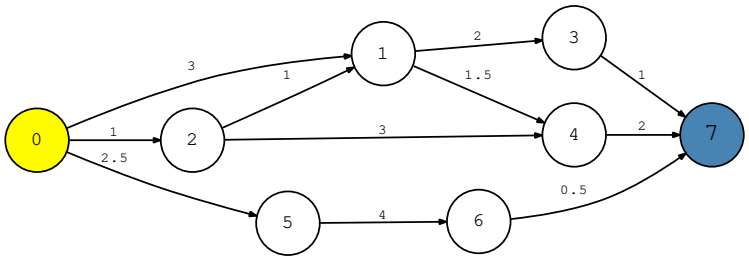
- BFS — graf przeszukiwań w kolejnych krokach:



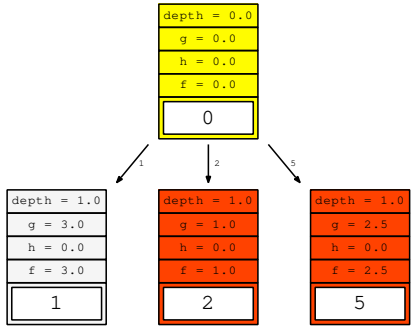
[Wyniki generowane przez bibliotekę *SaC*: `https://pklesk.github.io/sac`, ilustracje dzięki: *Graphviz* `https://www.graphviz.org`.]

Przykład 1 — BFS

- Węzeł 0 początkowy. Węzeł 7 końcowy.



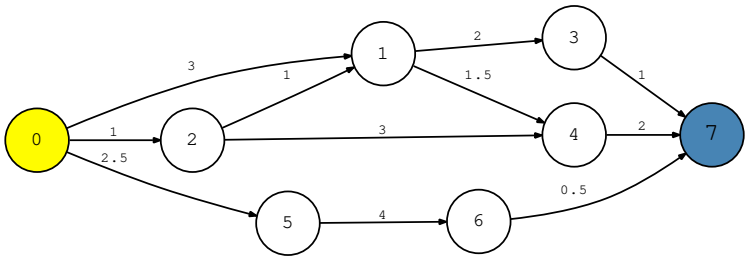
- BFS — graf przeszukiwań w kolejnych krokach:



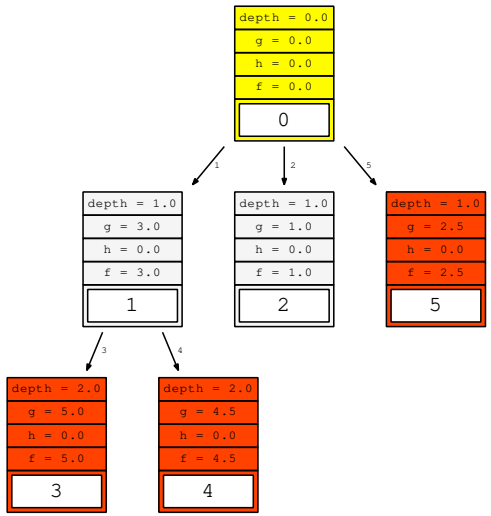
[Wyniki generowane przez bibliotekę *SaC*: `https://pklesk.github.io/sac`, ilustracje dzięki: *Graphviz* `https://www.graphviz.org`.]

Przykład 1 — BFS

- Węzeł 0 początkowy. Węzeł 7 końcowy.



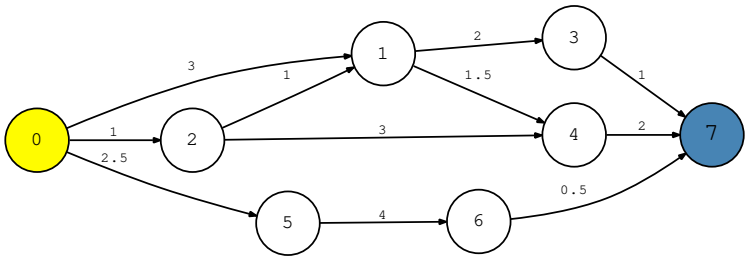
- BFS — graf przeszukiwań w kolejnych krokach:



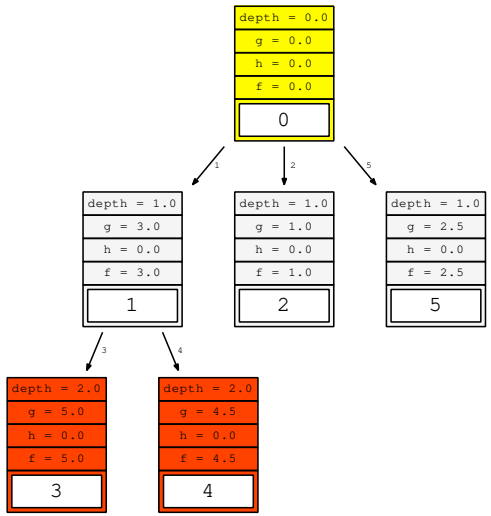
[Wyniki generowane przez bibliotekę *SaC*: `https://pklesk.github.io/sac`, ilustracje dzięki: *Graphviz* `https://www.graphviz.org`.]

Przykład 1 — BFS

- Węzeł 0 początkowy. Węzeł 7 końcowy.



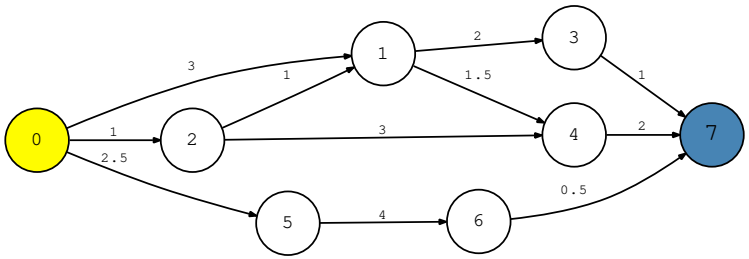
- BFS — graf przeszukiwań w kolejnych krokach:



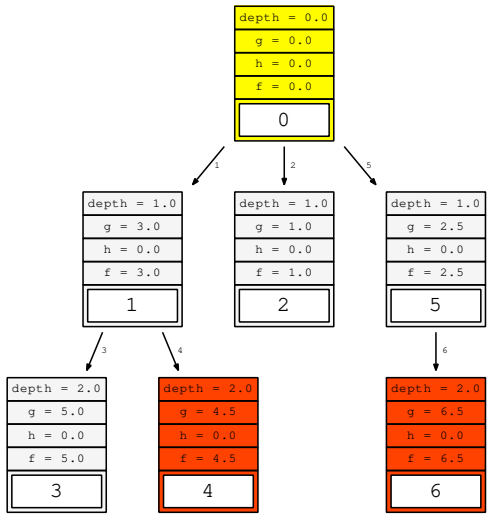
[Wyniki generowane przez bibliotekę *SaC*: `https://pklesk.github.io/sac`, ilustracje dzięki: *Graphviz* `https://www.graphviz.org`.]

Przykład 1 — BFS

- Węzeł 0 początkowy. Węzeł 7 końcowy.



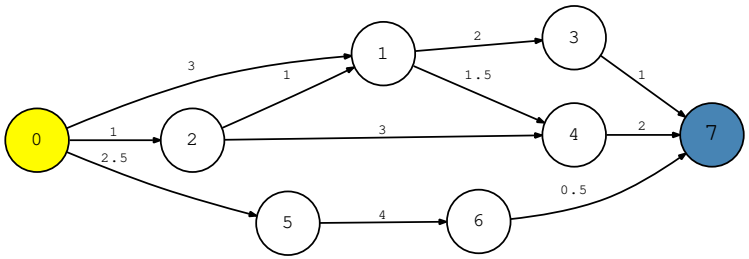
- BFS — graf przeszukiwań w kolejnych krokach:



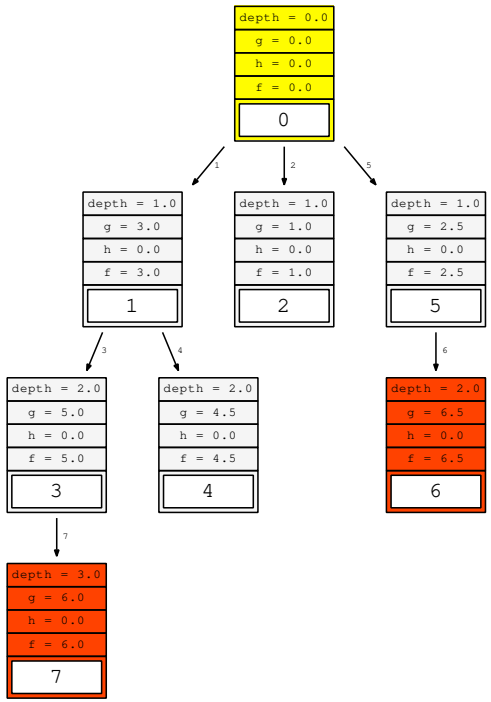
[Wyniki generowane przez bibliotekę *SaC*: `https://pklesk.github.io/sac`, ilustracje dzięki: *Graphviz* `https://www.graphviz.org`.]

Przykład 1 — BFS

- Węzeł 0 początkowy. Węzeł 7 końcowy.



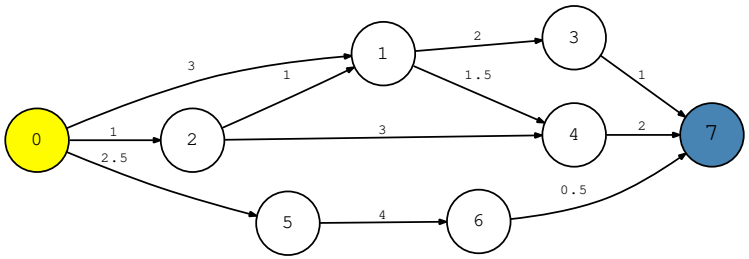
- BFS — graf przeszukiwań w kolejnych krokach:



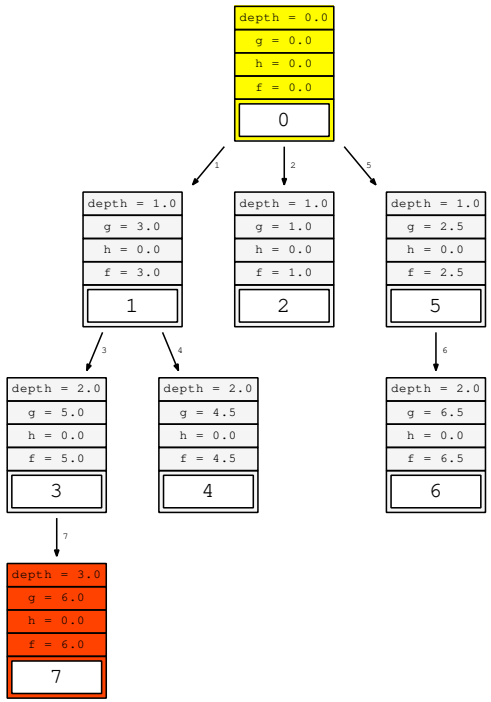
[Wyniki generowane przez bibliotekę *SaC*: `https://pklesk.github.io/sac`, ilustracje dzięki: *Graphviz* `https://www.graphviz.org`.]

Przykład 1 — BFS

- Węzeł 0 początkowy. Węzeł 7 końcowy.



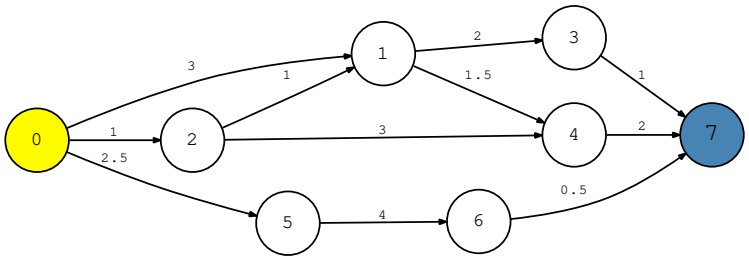
- BFS — graf przeszukiwań w kolejnych krokach:



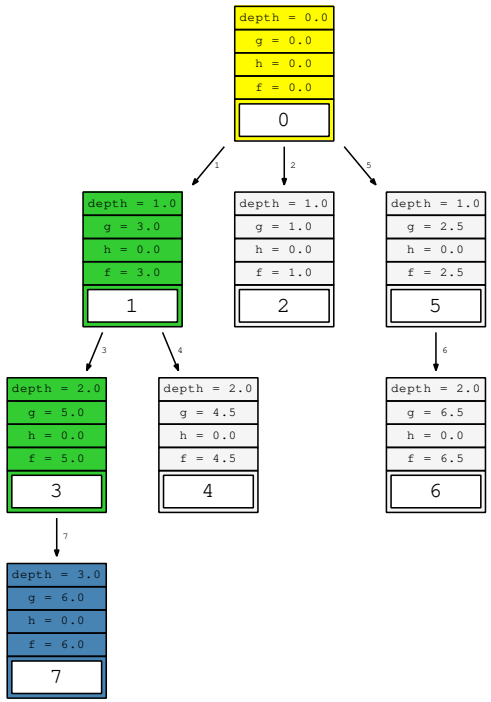
[Wyniki generowane przez bibliotekę *SaC*: `https://pklesk.github.io/sac`, ilustracje dzięki: *Graphviz* `https://www.graphviz.org`.]

Przykład 1 — BFS

- Węzeł 0 początkowy. Węzeł 7 końcowy.



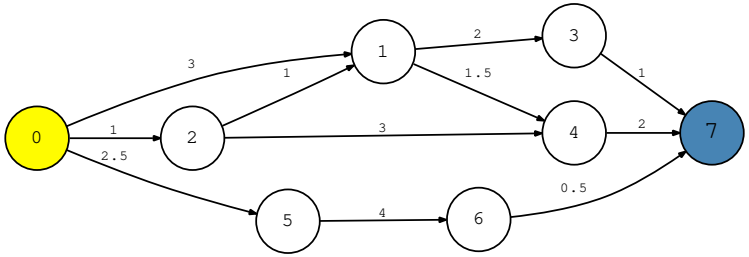
- BFS — graf przeszukiwań w kolejnych krokach:



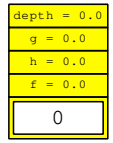
[Wyniki generowane przez bibliotekę *SaC*: `https://pklesk.github.io/sac`, ilustracje dzięki: *Graphviz* `https://www.graphviz.org`.]

Przykład 1 — DFS

- Węzeł 0 początkowy. Węzeł 7 końcowy.

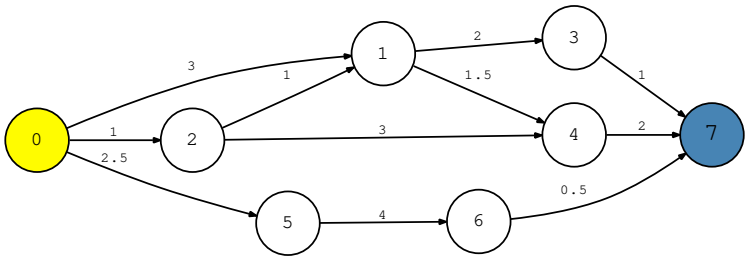


- DFS — graf przeszukiwań w kolejnych krokach:

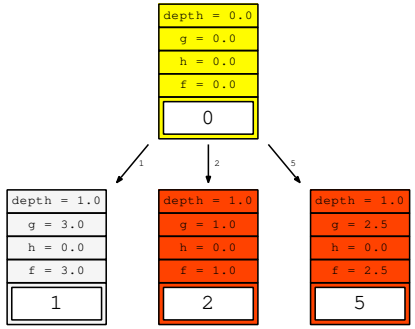


Przykład 1 — DFS

- Węzeł 0 początkowy. Węzeł 7 końcowy.

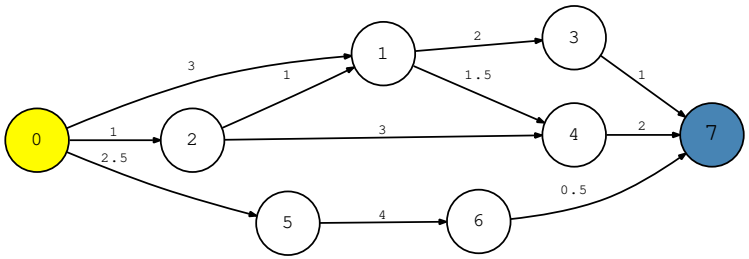


- DFS — graf przeszukiwań w kolejnych krokach:

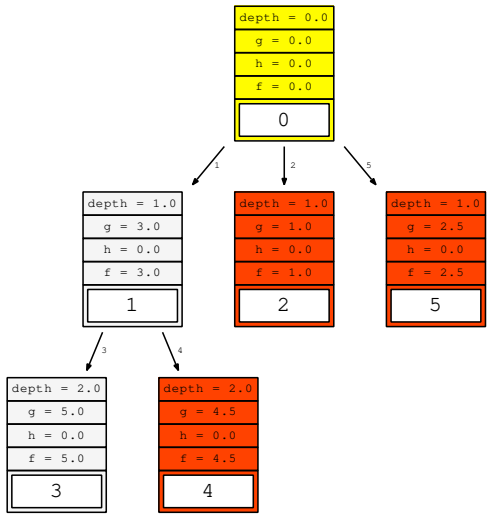


Przykład 1 — DFS

- Węzeł 0 początkowy. Węzeł 7 końcowy.

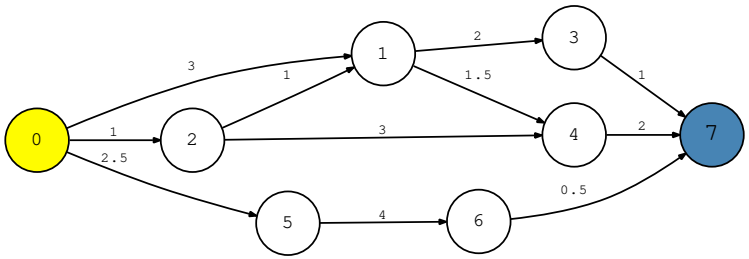


- DFS — graf przeszukiwań w kolejnych krokach:

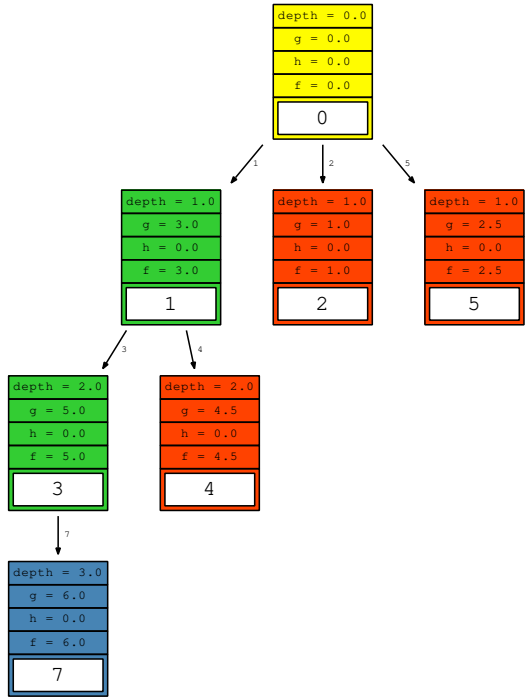


Przykład 1 — DFS

- Węzeł 0 początkowy. Węzeł 7 końcowy.

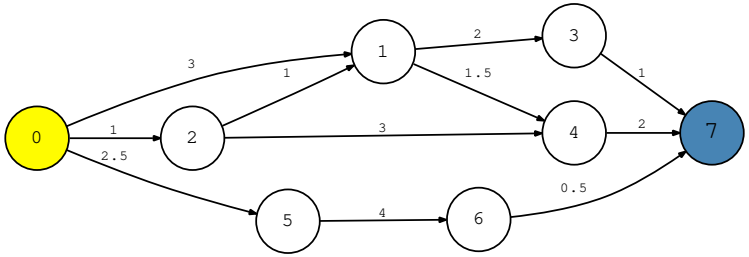


- DFS — graf przeszukiwań w kolejnych krokach:

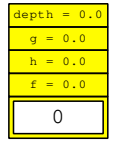


Przykład 1 — algorytm Dijkstry

- Węzeł 0 początkowy. Węzeł 7 końcowy.

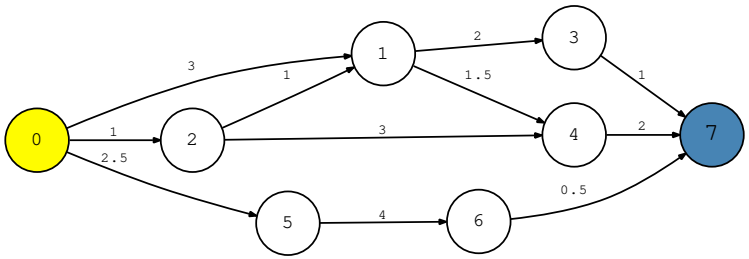


- algorytm Dijkstry — graf przeszukiwań w kolejnych krokach:

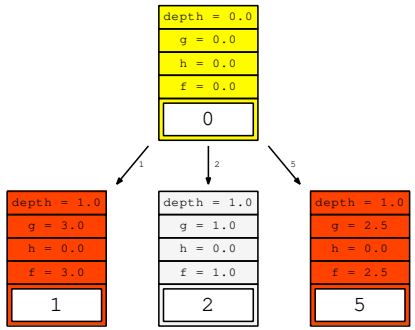


Przykład 1 — algorytm Dijkstry

- Węzeł 0 początkowy. Węzeł 7 końcowy.

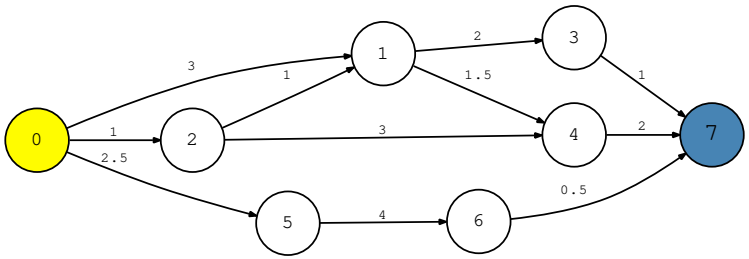


- algorytm Dijkstry — graf przeszukiwań w kolejnych krokach:

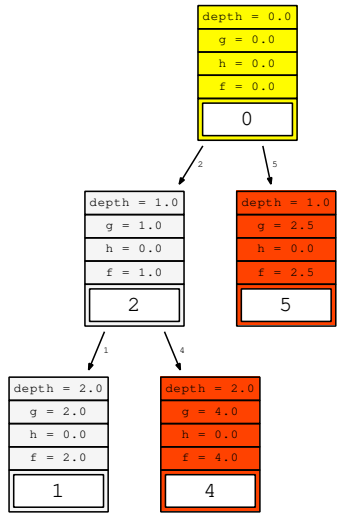


Przykład 1 — algorytm Dijkstry

- Węzeł 0 początkowy. Węzeł 7 końcowy.

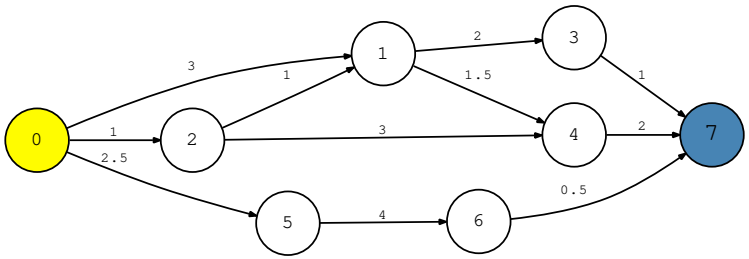


- algorytm Dijkstry — graf przeszukiwań w kolejnych krokach:

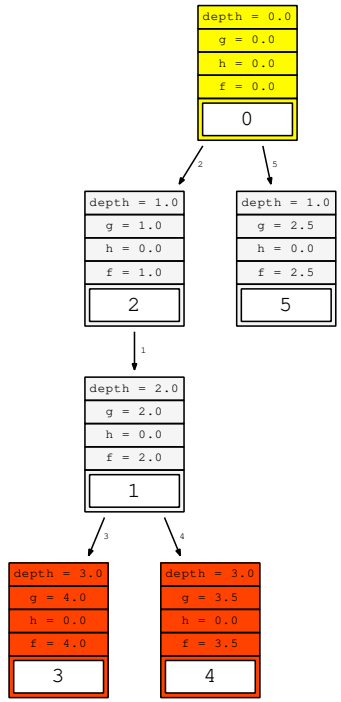


Przykład 1 — algorytm Dijkstry

- Węzeł 0 początkowy. Węzeł 7 końcowy.

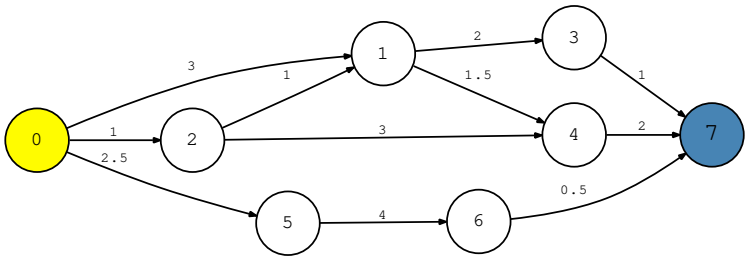


- algorytm Dijkstry — graf przeszukiwań w kolejnych krokach:

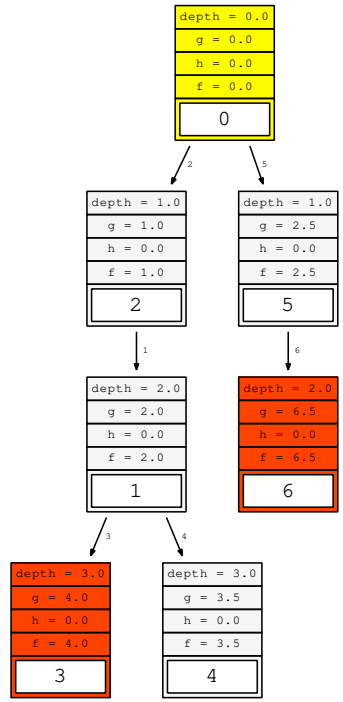


Przykład 1 — algorytm Dijkstry

- Węzeł 0 początkowy. Węzeł 7 końcowy.

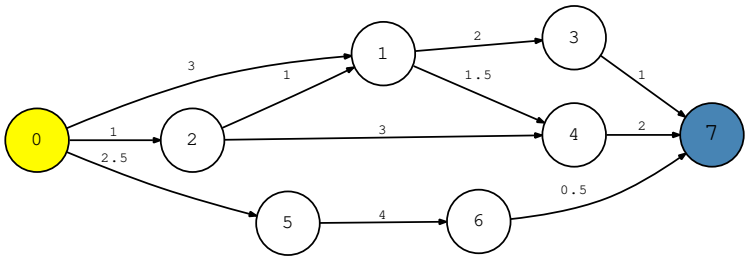


- algorytm Dijkstry — graf przeszukiwań w kolejnych krokach:

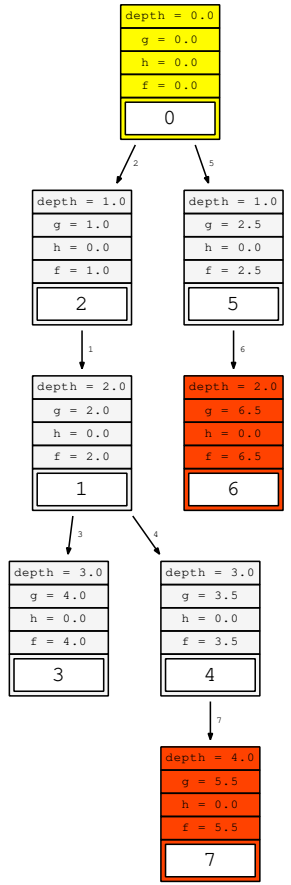


Przykład 1 — algorytm Dijkstry

- Węzeł 0 początkowy. Węzeł 7 końcowy.

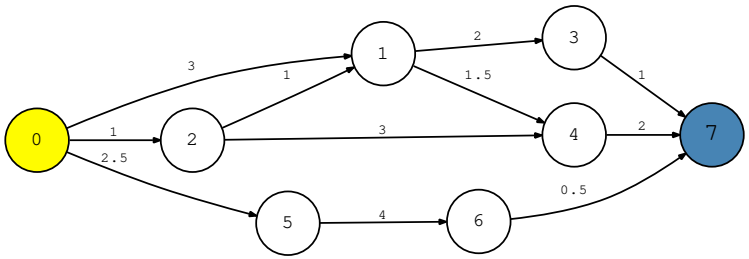


- algorytm Dijkstry — graf przeszukiwań w kolejnych krokach:

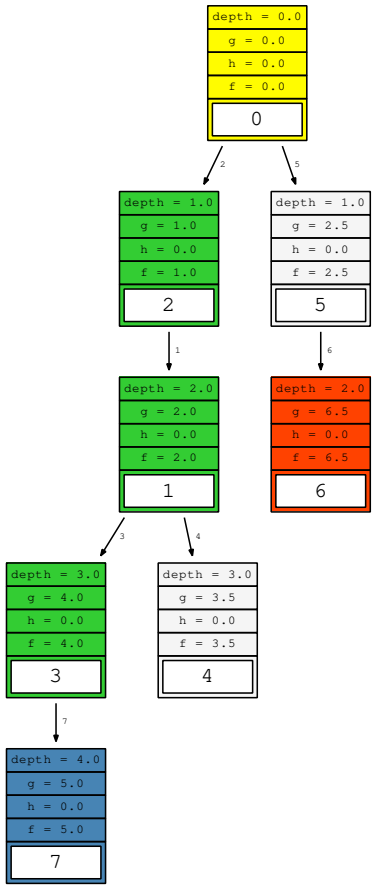


Przykład 1 — algorytm Dijkstry

- Węzeł 0 początkowy. Węzeł 7 końcowy.

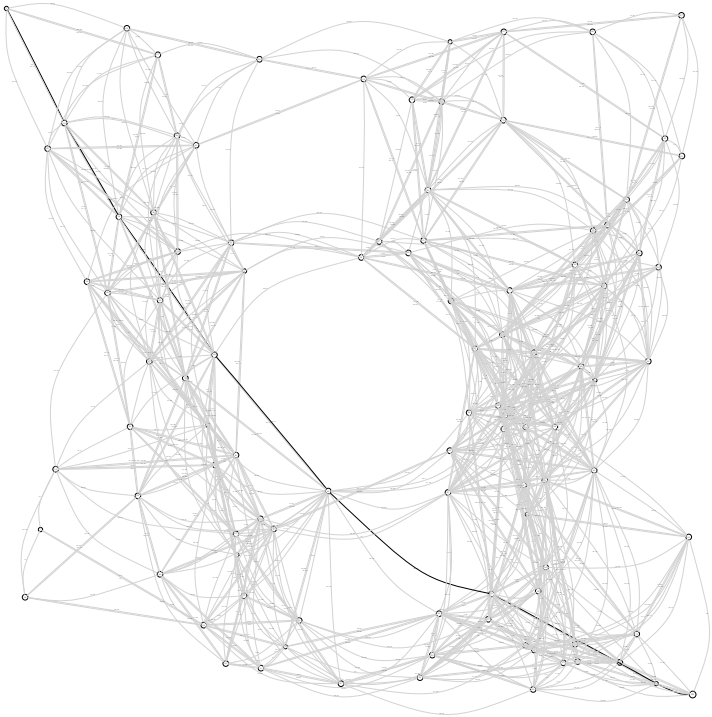


- algorytm Dijkstry — graf przeszukiwań w kolejnych krokach:



Graf „geograficzny”

- Graf utworzony sztucznie: 100 węzłów, 10% możliwych krawędzi.
- Węzły losowane w kwadracie $[0,1] \times [0,1]$ poza początkowym $(0,0)$ i końcowym $(1,1)$.
- Wagi krawędzi (koszty przejść) proporcjonalne do odległości euklidesowych z losowymi zaburzeniami.



- Najkrótsza ścieżka $(0, 18, 14, 64, 60, 10, 5, 99)$ o koszcie ≈ 149.52.
- ***Algorytm Dijkstry*** odwiedza *wszystkie* stany przed natrafieniem na najkrótszą ścieżkę dla powyższego grafu.

Spis treści

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]
- W pierwszej kolejności rozwijany zawsze najbardziej obiecujący („najlepszy”) stan.
- Ocena, na ile stan s jest obiecujący, za pomocą funkcji heurystycznej $h(s)$ — przeszukiwanie poinformowane.
- Różne możliwości konstruowania $h(s)$:
  - na podstawie informacji zawartej w samym s,
  - na podstawie informacji zebranych wzdłuż ścieżki z s_0 do s,
  - na podstawie ogólnej znajomości problemu lub właśności stanu końcowego (rozwiązania).
- Zwyczajowo $h(s) \geqslant 0$. Małe wartości sugerują bliskość do rozwiązania.
- Podejście *best-first* ma służyć szybkiemu osiągnięciu rozwiązania *dowolną* ścieżką. Nie dba się o jej minimalizację (brak pojęcia kosztu ścieżki).
- Struktury danych (ponownie): *Open* (kolejka priorytetowa), *Closed* (mapa haszująca).

Best-first search

```
1:  procedura BestFirstSearch(s_0)                                   ▷ stan początkowy: s_0
2:      Closed := ∅                                                  ▷ pusty zbiór odwiedzonych stanów
3:      oblicz h(s_0)                                                ▷ heurystyka wg podanego przepisu
4:      ustaw pusty wskaźnik z s_0 na jego rodzica
5:      Open := {s_0}                                                ▷ kolejka stanów oczekujących
6:      dopóki Open ≠ ∅ wykonaj
7:          pobierz (i usuń) z Open stan s o najmniejszej wartości h(s)   ▷ operacja "poll"
8:          jeżeli s jest stanem końcowym to zwróć s                 ▷ znaleziono rozwiązanie
9:          wygeneruj zbiór stanów {t} potomnych dla s
10:         dla wszystkich t wykonaj
11:             jeżeli t ∈ Closed to kontynuuj od kolejnej iteracji   ▷ t już odwiedzone
12:             oblicz h(t)
13:             ustaw wskaźnik rodzica t na s
14:             jeżeli t ∉ Open to
15:                 dodaj t do Open
16:             w przeciwnym razie
17:                 jeżeli nowa wartość h(t) jest mniejsza niż znana dotychczas to   ▷ np. gdy h(s) zależy od informacji na ścieżce
18:                     zastąp t w Open nowym egzemplarzem (aktualnie badanym)
19:                     uaktualnij pozycję t w Open
20:         dodaj s do Closed
21:     zwróć wynik pusty                                           ▷ nie znaleziono rozwiązania
```

Sudoku — przykład 1

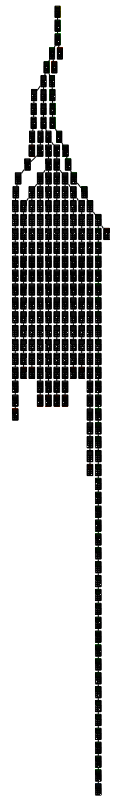
- Poziom trudny:

*	*	*	*	*	*	*	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 + heurystyka „liczba pustych miejsc”*, potomkowie w „polu minimalnym”, stanów odwiedzonych 222, stanów oczekujących 14:



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

Sudoku — przykład 2

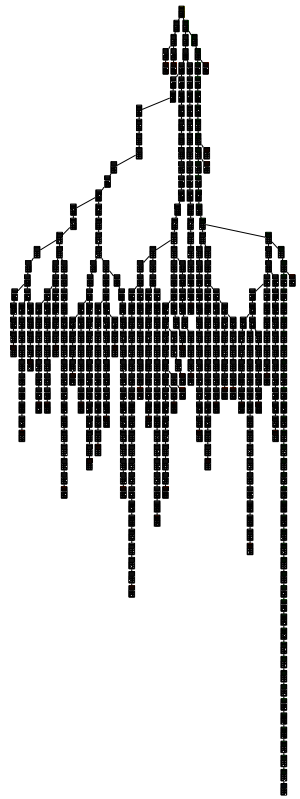
- Poziom trudny:

*	*	*	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 + heurystyka „liczba pustych miejsc”*, potomkowie w „polu minimalnym”, stanów odwiedzonych 418, stanów oczekujących 41:



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

Sudoku — „Qassim Hamza”

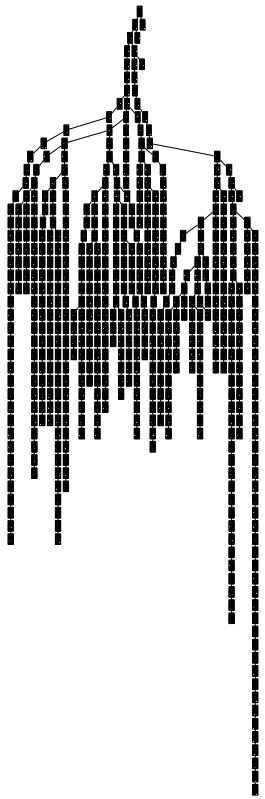
- Poziom bardzo trudny:

*	*	*	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 + heurystyka „liczba pustych miejsc”*, potomkowie w „polu minimalnym”, stanów odwiedzonych 525, stanów oczekujących 40:



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

Sudoku — inna heurystyka

- Utożsamijmy stan s z dwuwymiarową tablicą sudoku.
- Niech $s(i,j)$ oznacza zawartość komórki o indeksie (i,j).
- Niech $r(s,i,j)$ oznacza zbiór możliwych cyfr w komórce (i,j) po odjęciu od zbioru $\{1,\ldots,9\}$ cyfr obecnych w i-ym wierszu, j-ej kolumnie i w podkwadracie zawierającym komórkę (i,j).
- Heurystyka *„suma pozostałych możliwości”*:

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

Sudoku — przykład 1

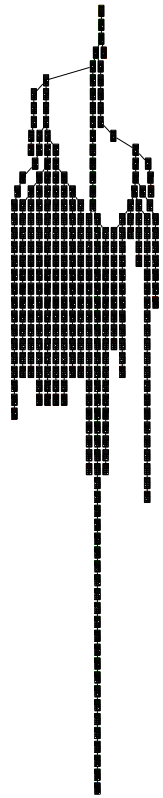
- Poziom trudny:

*	*	*	*	*	*	*	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 + heurystyka „suma pozostałych możliwości”***,
potomkowie w „polu minimalnym”,
stanów odwiedzonych 304, stanów oczekujących 20:



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

Sudoku — przykład 2

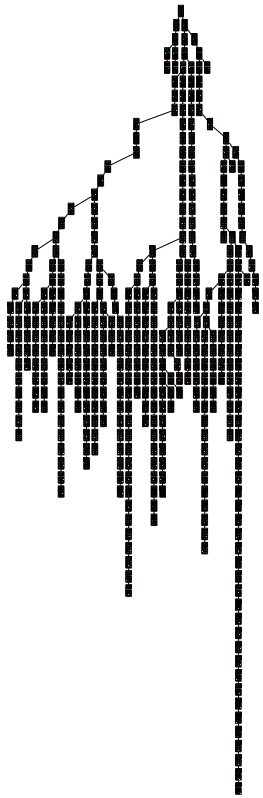
- Poziom trudny:

*	*	*	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 + heurystyka „suma pozostałych możliwości”***,
potomkowie w „polu minimalnym”,
stanów odwiedzonych 381, stanów oczekujących 37:



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

Sudoku — „Qassim Hamza”

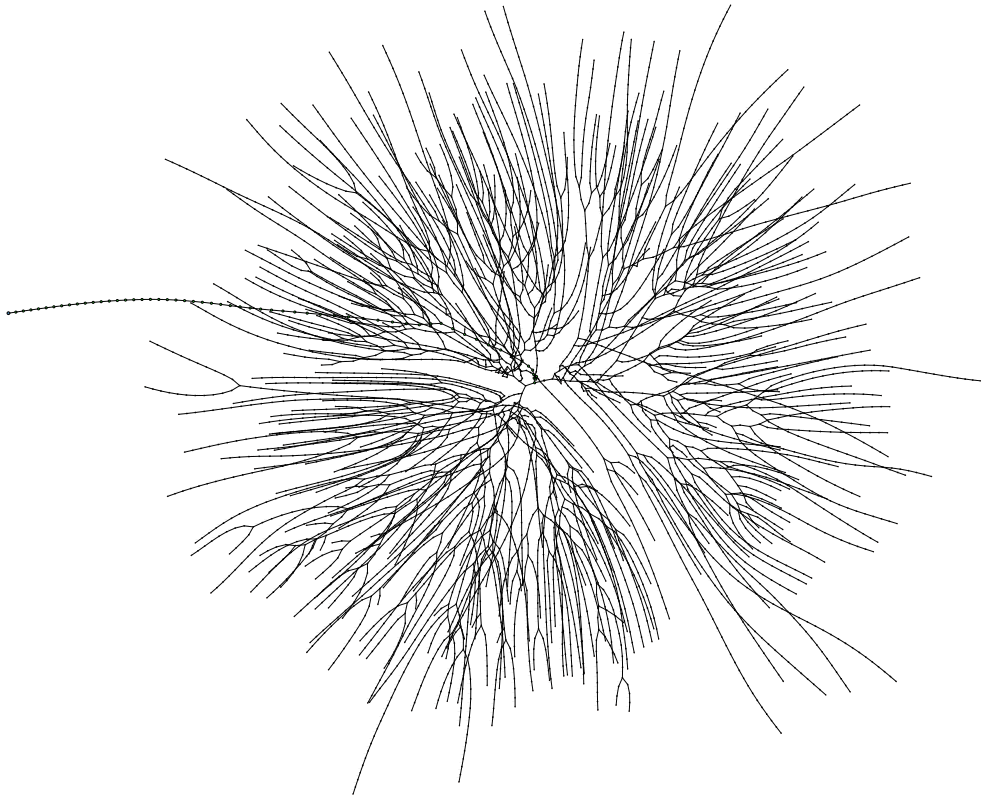
- Poziom bardzo trudny:

*	*	*	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 + heurystyka „suma pozostałych możliwości”***,
potomkowie w „polu minimalnym”,
stanów odwiedzonych 5 267, stanów oczekujących 452:



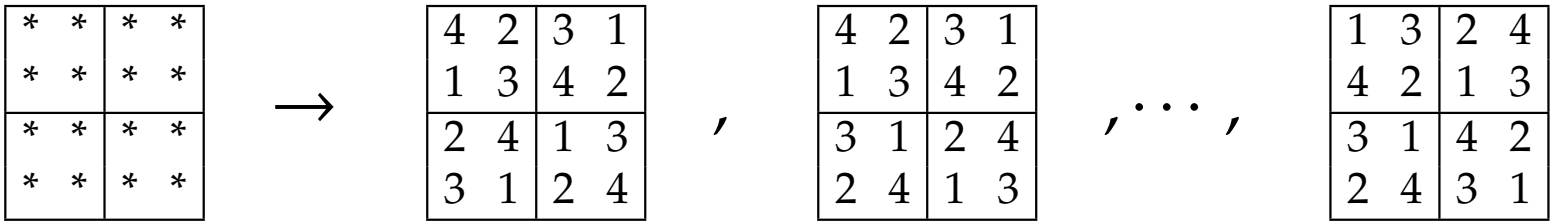
[czas: 208 ms, Intel Xeon CPU E3-1505M v5 2.8 GHz (boost 3.7 GHz)]

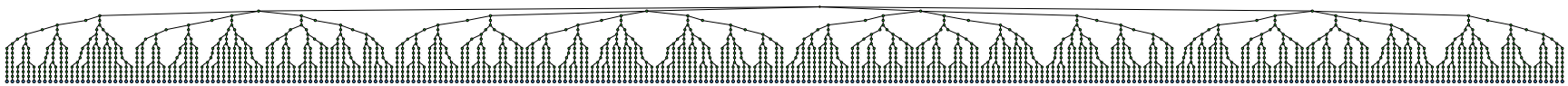
Sudoku — porównanie heurystyk

- Porównanie dla 50 planszy sudoku — źródło:
  [https://projecteuler.net/project/resources/p096_sudoku.txt]
- ***Best-first search + heurystyka „liczba pustych miejsc":***
  - Średnia liczba stanów odwiedzonych: 166.92.
  - Średnia liczba stanów oczekujących (w chwili stopu): 14.32.
  - Średni czas: 11.88 ms.
- ***Best-first search + heurystyka „liczba pozostałych możliwości":***
  - Średnia liczba stanów odwiedzonych: 176.64.
  - Średnia liczba stanów oczekujących (w chwili stopu): 15.08.
  - Średni czas: 13.16 ms.

Wszystkie sudoku 4×4

- Rozwiązań: 288.





- Stanów odwiedzonych 2 273, stanów oczekujących 0.

Spis treści

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/]
- Nieformalnie algorytm A^* może być traktowany jako kombinacja algorytmów Dijkstry i Best-first search (lub ich ogólniejsza forma).
- Funkcja decydująca o porządku pobierania stanów z *Open* ma postać:

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

  gdzie: $g(s)$ — dokładny koszt przebyty od s_0 do s, natomiast $h(s)$ — heurystyczne oszacowanie kosztu pozostałego od s do stanu końcowego.
- Skoro h jest heurystyką, to także jest nią f.
- Dla znajdowania najkrótszych ścieżek h musi być tzw. ***dopuszczalną heurystyką (ang. admissible heuristics)*** — tj. dolnym oszacowaniem pozostałego kosztu — nie może przeszacowywać prawdziwego kosztu.
- Dla grafów geograficznych dopuszczalną heurystyką jest na pewno odległość w linii prostej (euklidesowa).
- Struktury danych (ponownie): *Open* (kolejka priorytetowa), *Closed* (mapa haszująca).

A^*

```
1:  procedura AStar(s_0)                                   ▷ stan początkowy: s_0
2:      Closed := ∅                                        ▷ pusty zbiór odwiedzonych stanów
3:      g(s_0) := 0                                        ▷ koszt przebyty od startu
4:      oblicz h(s_0)                                      ▷ heurystyka wg podanego przepisu
5:      f(s_0) := g(s_0) + h(s_0)                          ▷ suma decydująca o porządku pobierania z Open
6:      ustaw pusty wskaźnik rodzica dla s_0
7:      Open := {s_0}                                      ▷ kolejka stanów oczekujących
8:      dopóki Open ≠ ∅ wykonaj
9:          pobierz (i usuń) z Open stan s o najmniejszej wartości f(s)   ▷ operacja "poll"
10:         jeżeli s jest stanem końcowym to zwróć s       ▷ znaleziono rozwiązanie
11:         wygeneruj zbiór stanów {t} potomnych dla s
12:         dla wszystkich t wykonaj
13:             jeżeli t ∈ Closed to kontynuuj od kolejnej iteracji   ▷ t już odwiedzone
14:             g(t) := g(s) + Δ(s → t)
15:             oblicz h(t)
16:             f(t) := g(t) + h(t)
17:             ustaw wskaźnik rodzica t na s
18:             jeżeli t ∉ Open to
19:                 dodaj t do Open
20:             w przeciwnym razie
21:                 jeżeli nowa wartość f(t) jest mniejsza niż poprzednio znana to
22:                     zastąp t w Open nowym egzemplarzem (aktualnie badanym)
23:                     uaktualnij pozycję t w Open
24:         dodaj s do Closed
25:     zwróć wynik pusty                                 ▷ nie znaleziono rozwiązania
```

A^*

Twierdzenie „o optymalności ścieżki dla heurystyki dopuszczalnej"

Jeżeli algorytm A^* używając heurystyki dopuszczalnej znajduje stan końcowy, to skojarzona z nim ścieżka jest najkrótsza.

Dowód: W chwili stopu (linia 10) algorytm zwraca pewien stan s^* o przebytym koszcie $g(s^*)$. Skoro s^* spełnia warunek stopu to $h(s^*) = 0$. Wiadomo, że dla wszystkich stanów s pozostających w chwili stopu w *Open* zachodzi $f(s) \geqslant f(s^*)$. Wsród tych stanów można wyróżnić trzy przypadki. Przypadek 1: dany stan s spełnia warunek stopu tj. $h(s) = 0$, ale $g(s) \geqslant g(s^*)$, ponieważ $f(s) \geqslant f(s^*)$. Przypadek 2: dany stan s nie spełnia warunku stopu tj. $h(s) > 0$, ale mógłby w wyniku dalszej drogi prowadzić do stanu końcowego, i ma aktualnie $g(s) < g(s^*)$; skoro $h(s)$ jest dolnym oszacowaniem brakującego kosztu oraz $f(s) \geqslant f(s^*)$, to prawdziwy koszt dotarcia do celu idąc przez s musi spełniać nierówności $g(s) + \Delta(s \to s^*) \geqslant g(s) + h(s) \geqslant g(s^*)$. Przypadek 3: dany stan s ma $h(s) > 0$ i $g(s) \geqslant g(s^*)$ — nieistotny. ■

Heurystyka monotoniczna

- Dodatkowe użyteczne pojęcie: ***heurystyka monotoniczna***.
- Mówimy, że h jest *monotoniczna* jeżeli dla wszystkich par s, t (gdzie t jest potomkiem s) zachodzi:

$$f(s) \leqslant f(t), \tag{3}$$

co można przepisać jako

$$g(s) + h(s) \leqslant g(t) + h(t) \tag{4}$$

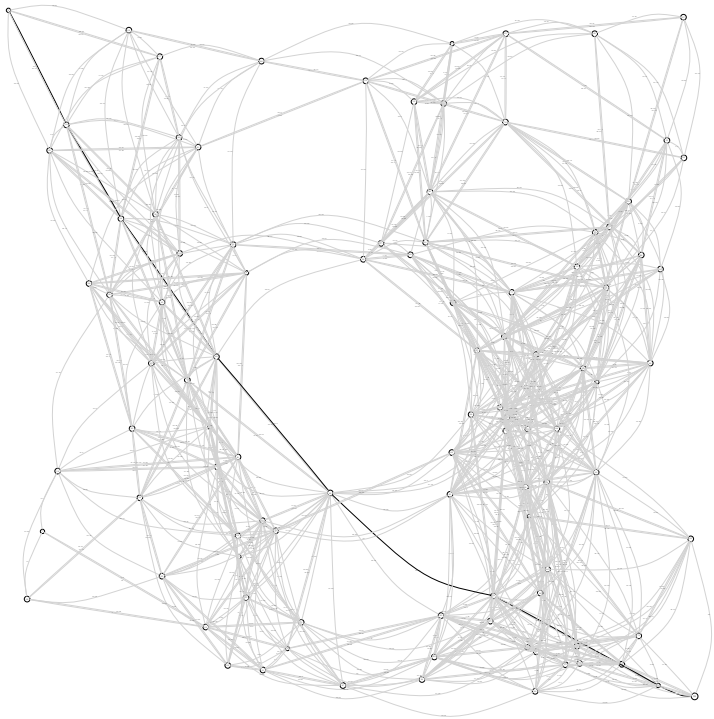
$$h(s) \leqslant g(t) - g(s) + h(t) \tag{5}$$

$$h(s) \leqslant \Delta(s \to t) + h(t). \tag{6}$$

- Jest to forma *nierówności trójkąta*: heurystyka w punkcie s nie może być większa niż koszt przejścia $s \to t$ plus heurystyka w punkcie t.
- Przypadek równości w zapisie (6) zachodzi tylko w sytuacji, gdy podróżujemy do celu po linii prostej (w sensie metryki skojarzonej z danym grafem).
- Jeżeli heurystyka jest *monotoniczna*, to jest *dopuszczalna*.

Graf „geograficzny” ponownie

- Graf utworzony sztucznie: 100 węzłów, 10% możliwych krawędzi.



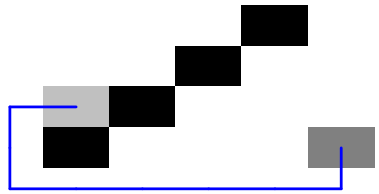
- Najkrótsza ścieżka $(0, 18, 14, 64, 60, 10, 5, 99)$ o koszcie ≈ 149.52.
- ***Algorytm Dijkstry*** odwiedza *wszystkie* stany przed natrafieniem na najkrótszą ścieżkę.
- ***A^* + odległość euklidesowa:*** stanów odwiedzonych 18, stanów otwartych: 38 — przeszukiwanie poinformowane.

Dobra i zła (przeszacowująca) heurystyka

- Dobra:

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

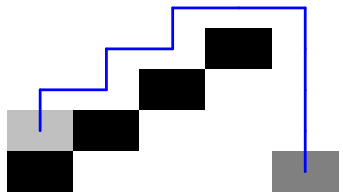
$$\text{lub} \quad h_2(s) = |s_x - s_x^*| + |s_y - s_y^*| \tag{8}$$



- Zła:

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

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



Puzzle przesuwne

- **Puzzle przesuwne** (puzzle $n^2 - 1$):
  Rozpoczynając od danego stanu początkowego i przesuwając numery sąsiadujące z polem pustym (o numerze 0) w jego miejsce, należy w jak najmniejszej liczbie ruchów dojść do stanu docelowego, w którym numery $\{0, 1, \ldots, n^2 - 1\}$ są uporządkowane kolejno wierszami.

0	1	2
3	4	5
6	7	8

Puzzle przesuwne

- *"misplaced tiles"* — liczba pól nie na swoim miejscu (z pominięciem pola '0' w zliczaniu).
- *"Manhattan"* — suma odległości w metryce Manhattan poszczególnych pól od ich miejsc docelowych (z pominięciem pola '0' w zliczaniu).

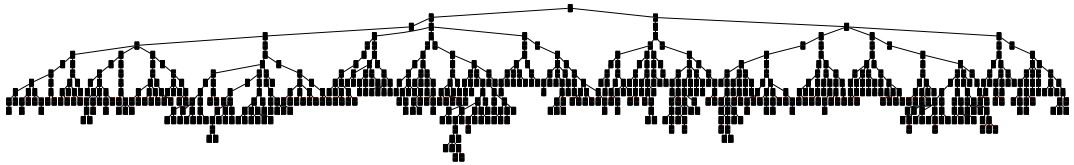
$$h(s) = \sum_{\substack{0 \leqslant i,j < n \\ s(i,j) \neq 0}} \sum \left| i - \lfloor s(i,j)/n \rfloor \right| + \left| j - s(i,j) \bmod n \right|. \tag{9}$$

- *"Manhattan + konflikty liniowe"* — jak wyżej + zliczenie dodatkowych dwuruchów wynikających z konfliktów liniowych — patrz:
  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...]
- Czy te heurystyki są monotoniczne?

Puzzle przesuwne

- Grafy przeszukiwań dla stanu $(0,3,2;4,7,8;1,5,6)$ i różnych heurystyk.
- ***A^* + "misplaced tiles"***



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

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

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

- ***A^* + "Manhattan + konflikty liniowe"***

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

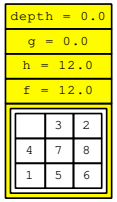
- Długość najkrótszej ścieżki 16. Ścieżka: $(D,R,D,R,U,L,L,D,R,U,U,L,D,R,U,L)$.

Puzzle przesuwne — porównanie heurystyk

- Porównanie dla 100 losowych planszy dla $n = 3$, każda mieszana za pomocą 1000 ruchów.
- ***A^* + "misplaced tiles"***
  - Średnia liczba stanów odwiedzonych: 12 263.89.
  - Średnia liczba stanów oczekujących (w chwili stopu): 5 865.45.
  - Średni czas: 28.57 ms.
- ***A^* + "Manhattan"***
  - Średnia liczba stanów odwiedzonych: 1 024.44.
  - Średnia liczba stanów oczekujących (w chwili stopu): 588.19.
  - Średni czas: 8.09 ms.
- ***A^* + "Manhattan + konflikty liniowe"***
  - Średnia liczba stanów odwiedzonych: 530.14.
  - Średnia liczba stanów oczekujących (w chwili stopu): 316.81.
  - Średni czas: 7.37 ms.

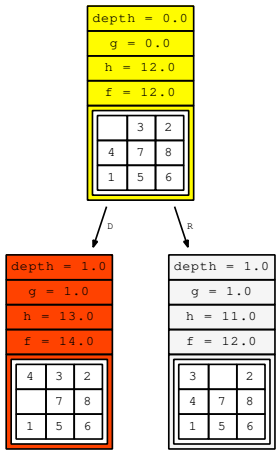
Puzzle przesuwne — przykład

- A^* **+ *„Manhattan + konflikty liniowe”*** — graf przeszukiwań w pierwszych 5 krokach i w ostatnim:



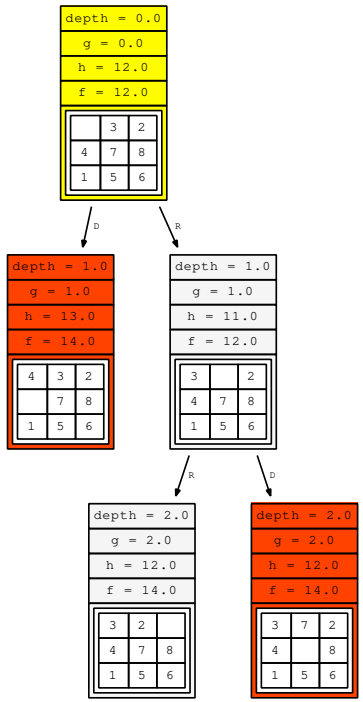
Puzzle przesuwne — przykład

- A^* **+ *„Manhattan + konflikty liniowe”*** — graf przeszukiwań w pierwszych 5 krokach i w ostatnim:



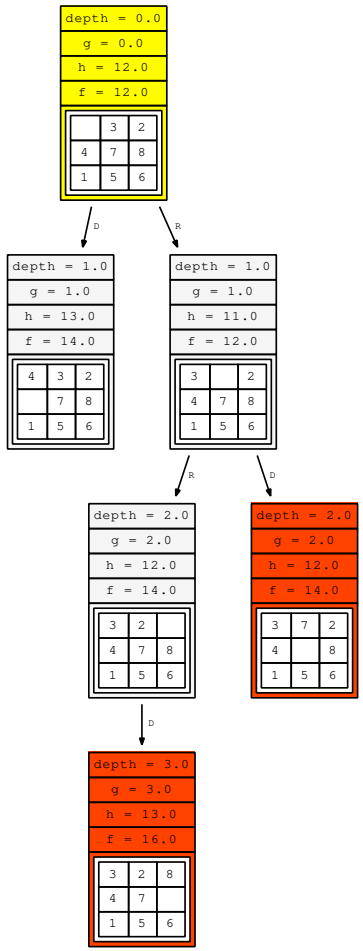
Puzzle przesuwne — przykład

- A^* **+ *„Manhattan + konflikty liniowe”*** — graf przeszukiwań w pierwszych 5 krokach i w ostatnim:



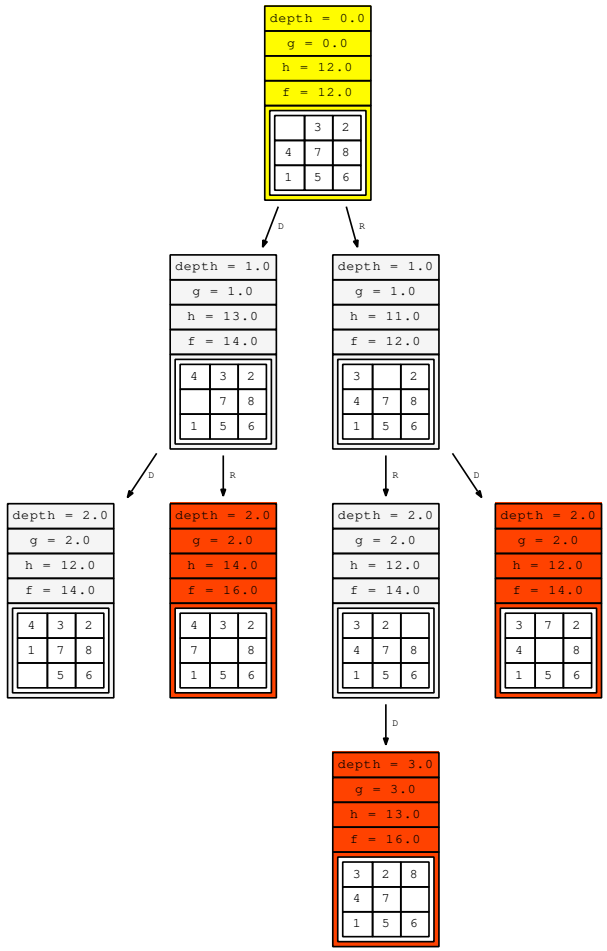
Puzzle przesuwne — przykład

- A^* ***+ „Manhattan + konflikty liniowe”*** — graf przeszukiwań w pierwszych 5 krokach i w ostatnim:



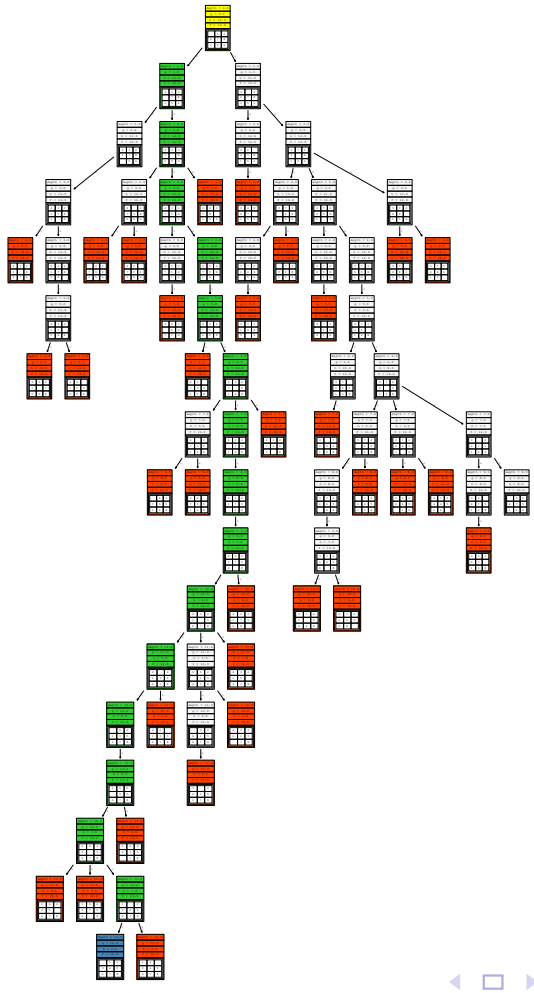
Puzzle przesuwne — przykład

- A^* **+ *„Manhattan + konflikty liniowe”*** — graf przeszukiwań w pierwszych 5 krokach i w ostatnim:



Puzzle przesuwne — przykład

- A^* **+ *„Manhattan + konflikty liniowe”*** — graf przeszukiwań w pierwszych 5 krokach i w ostatnim:



A^* vs Best-first search

- ***A^* + "Manhattan + konflikty liniowe"***

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

Długość najkrótszej ścieżki 16. Ścieżka: $(D, R, D, R, U, L, L, D, R, U, U, L, D, R, U, L)$.

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

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

Długość ścieżki 18. Ścieżka: $(R, D, D, R, U, L, U, L, D, D, R, U, U, L, D, R, U, L)$.

A^* vs Best-first search

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

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

Długość najkrótszej ścieżki 16. Ścieżka: $(D, R, D, R, U, L, L, D, R, U, U, L, D, R, U, L)$.

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

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

Długość ścieżki 134. Ścieżka: $(R, D, L, D, R, R, U, L, L, D, R, U, L, U, R, D, R, U, L, L, D, R, U, R, D, L, L, U, R, D, D, R, U, L, U, L, ..., L, L, U)$

Puzzle przesuwne — przykłady dla $n = 4$

- Wybrane przykłady wg **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*)*** — oszczczędna pamięciowo wersja A^*, ale kosztowna obliczeniowo.

nr	stan początkowy	długość ścieżki	*IDA** odwiedzonych	*IDA** czas [s]	A^* stanów odwiedzonych i oczekujących	A^* czas [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: $3.4 \cdot 10^6$, $2.8 \cdot 10^6$	—

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

A^* — uwagi końcowe

- Jeżeli $h(s) = 0$ dla każdego s, to: A^* = algortym Dijkstry.
- Jeżeli $g(s) = 0$ dla każdego s, to: A^* = Best-first search.
- Im heurystyka h niesie lepszą informację, tym algorytm A^* mniej się „napracuje”.
- Monotoniczność heurystyki implikuje trzy ważne konsekwencje:
  1. znalezione rozwiązanie jest optymalne (ścieżka najkrótsza),
  2. sam algorytm jest optymalny w ramach h, tj. żaden inny algorytm używający h nie odwiedzi mniejszej liczby stanów niż A^* (z dokładnością do sposobu rozstrzygania remisów),
  3. ***niech h^* oznacza doskonałą heurystykę reprezentującą dokładną odległość do celu, wtedy algorytm pracujący na podstawie h^* jest także doskonały — odwiedza możliwie najmniej węzłów (stąd też oryginalnie rozróżniano dwie nazwy tego algorytmu A i A^*)***.

Spis treści

*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]
- Dla niektórych odpowiednio dużych problemów algorytm A^* może wyczerpać pamięć (zbyt dużo RAMu dla zbiorów *Open* i *Closed*).
- *IDA** może być traktowana jako oszczędna pamięciowo wersja A^*.
- *IDA* nie* przechowuje ewidencji stanów odwiedzonych — brak zbioru *Closed*.
- *IDA** trzyma w pamięci tylko stany na aktualnie badanej ścieżce.
- Algorytm może być sformułowany rekurencyjnie (brak zbioru *Open*) lub tradycyjnie z główną pętlą (wówczas w użyciu nieduży zbiór *Open*).

IDA^* — szkic działania

- Algorytm używa wartości $h(s_0)$, aby ustalić początkowy horyzont przeszukiwań:

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

- Następnie algorytm bada różne ścieżki idące od s_0 (np. wg podejścia *depth-first*).
- Gdy osiągnięty zostanie stan końcowy w obrębie horyzontu H, to jest on zwracany.
- Każdy stan osiągnięty („dotknięty") poza horyzontem nie jest rozwijany dalej, ale informacja o zaobserwowanym w nim koszcie jest użyteczna dla ustalenia następnego horyzontu:

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

- Po wyczerpaniu wszystkich ścieżek w obrębie H, horyzont jest pogłębiany tj. $H := H'$ i cały proces powtarza się.

*IDA** rekurencyjnie

```
1:  procedura RecursiveIterativeDeepeningAStar(s_0)        ▷ stan początkowy: s_0
2:      g(s_0) := 0                                        ▷ koszt przebyty od startu
3:      oblicz h(s_0)                                      ▷ heurystyka wg podanego przepisu
4:      f(s_0) := g(s_0) + h(s_0)
5:      usaw pusty wskaźnik na rodzica dla s_0
6:      H := f(s_0)                                        ▷ początkowy horyzont przeszukiwań
7:      dopóki prawda wykonaj
8:          (s, H') :=Search(s_0, H)
9:          jeżeli s ≠ null to zwróć s                     ▷ znaleziono rozwiązanie
10:         jeżeli H' = ∞ to zwróć null                    ▷ nie znaleziono rozwiązania
11:         H := H'
```

```
1:  procedura Search(s, H)
2:      jeżeli f(s) > H to zwróć (null, f(s))
3:      jeżeli s jest stanem końcowym to zwróć (s, g(s))   ▷ znaleziono rozwiązanie
4:      H' := ∞
5:      wygeneruj zbiór stanów {t} potomnych dla s
6:      dla wszystkich t wykonaj
7:          g(t) := g(s) + Δ(s → t)
8:          f(t) := g(t) + h(t)
9:          (u, H'') := Search(t, H)
10:         jeżeli u ≠ null to zwróć (u, g(u))             ▷ znaleziono rozwiązanie
11:         H' := min{H', H''}                             ▷ pogłębianie horyzontu
        zwróć (null, H')
```

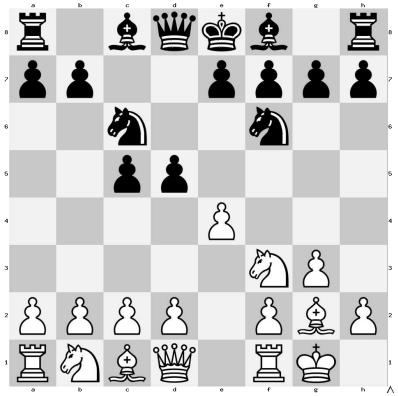
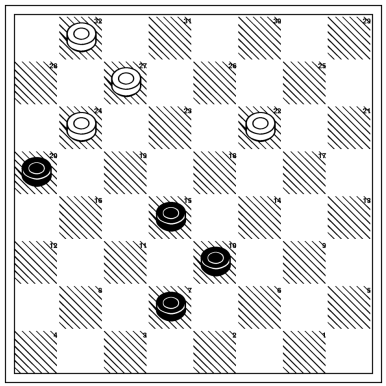
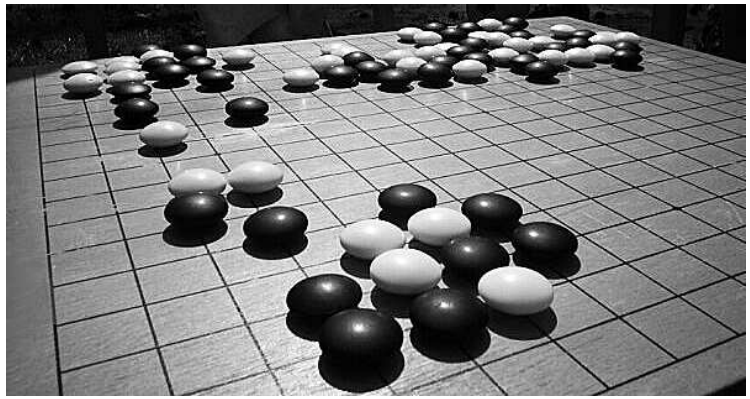
*IDA** nierekurencyjnie

```
1:  procedura IterativeDeepeningAStar(s_0)                          ▷ stan początkowy: s_0
2:      g(s_0) := 0                                                 ▷ droga przebyta od startu
3:      oblicz h(s_0)                                               ▷ heurystyka wg podanego przepisu
4:      f(s_0) := g(s_0) + h(s_0)
5:      ustaw pusty wskaźnik na rodzica dla s_0
6:      Open := {s_0}                                               ▷ kolejka stanów oczekujących
7:      H := f(s_0), H' := ∞                                        ▷ początkowy i następny horyzont
8:      dopóki Open ≠ ∅ wykonaj
9:          pobierz (i usuń) z Open stan s o najmniejszej wartości f(s)   ▷ operacja 'poll'
10:         jeżeli g(s) > H to
11:             H' := min{H', f(s)}
12:             jeżeli Open = ∅ to
13:                 H := H', H' := ∞, Open := {s_0}                 ▷ pogłębianie horyzontu
14:             kontynuuj od następnej iteracji
15:         jeżeli s jest stanem końcowym to zwróć s                ▷ znaleziono rozwiązanie
16:         wygeneruj zbiór stanów {t} potomnych dla s
17:         dla wszystkich t wykonaj
18:             g(t) := g(s) + Δ(s → t)
19:             oblicz h(t)
20:             f(t) := g(t) + h(t)
21:             ustaw wskaźnik rodzica t na s
22:             jeżeli t ∉ Open to
23:                 dodaj t do Open
24:             w przeciwnym razie
25:                 jeżeli nowy koszt g(t) jest mniejszy niż znany dotychczas to
26:                     zastąp t w Open nowym egzemplarzem (aktualnie badanym)
27:                     uaktualnij pozycję t w Open
28:     zwróć null                                                 ▷ nie znaleziono rozwiązania
```

Spis treści

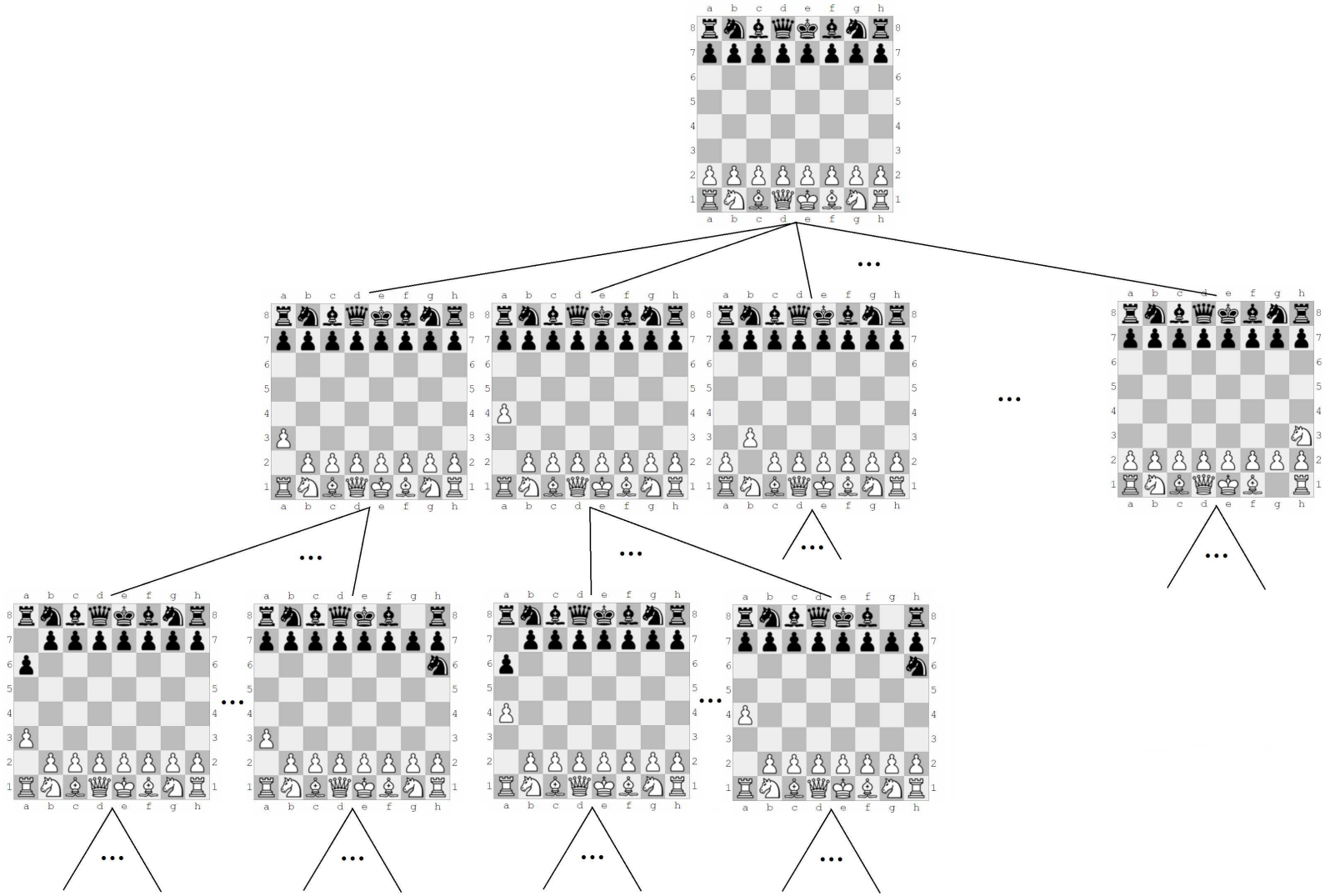
Gry

- Zwykle rozpatrywane gry dwuosobowe: szachy, warcaby, GO (batuk), . . .

- Gra — pewna sytuacja konfliktowa, w której gracze mają sprzeczne interesy, i gdzie mamy jasno zdefiniowane reguły.
- **Problem przeszukiwania drzewa gry:**
  Mając daną pewną pozycję w grze (w szczególności początkową), należy wystawić ***oceny liczbowe*** dla poszczególnych ***ruchów*** możliwych dla gracza, na którego przypada teraz kolej ruchu. Ocena ma reprezentować dokładną lub prawdopodobną ***wypłatę*** (ang. *payoff*) gracza, jeżeli wybierze on dany ruch przy założeniu optymalnego postępowania drugiego gracza.

Gry — początek drzewa szachów

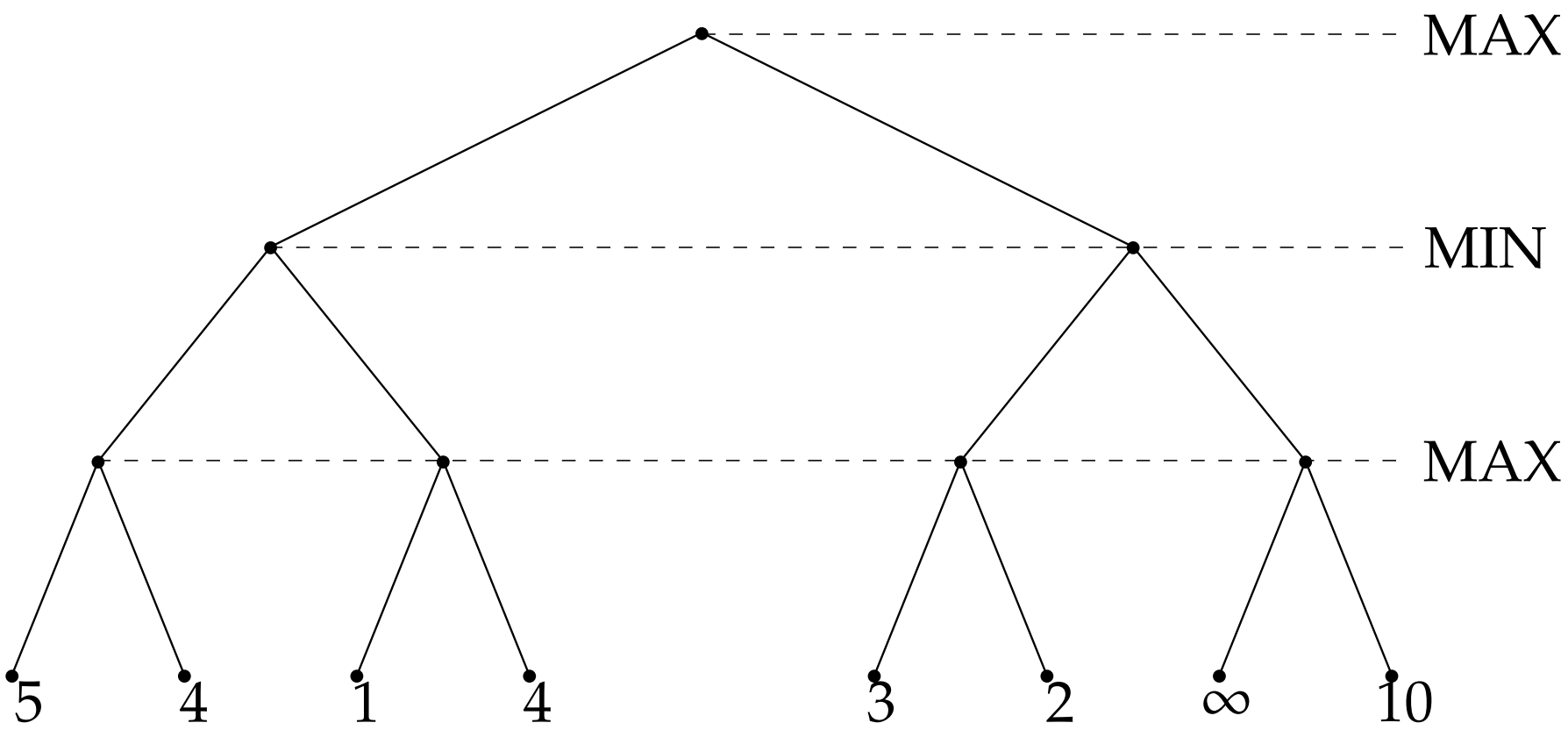


Spis treści

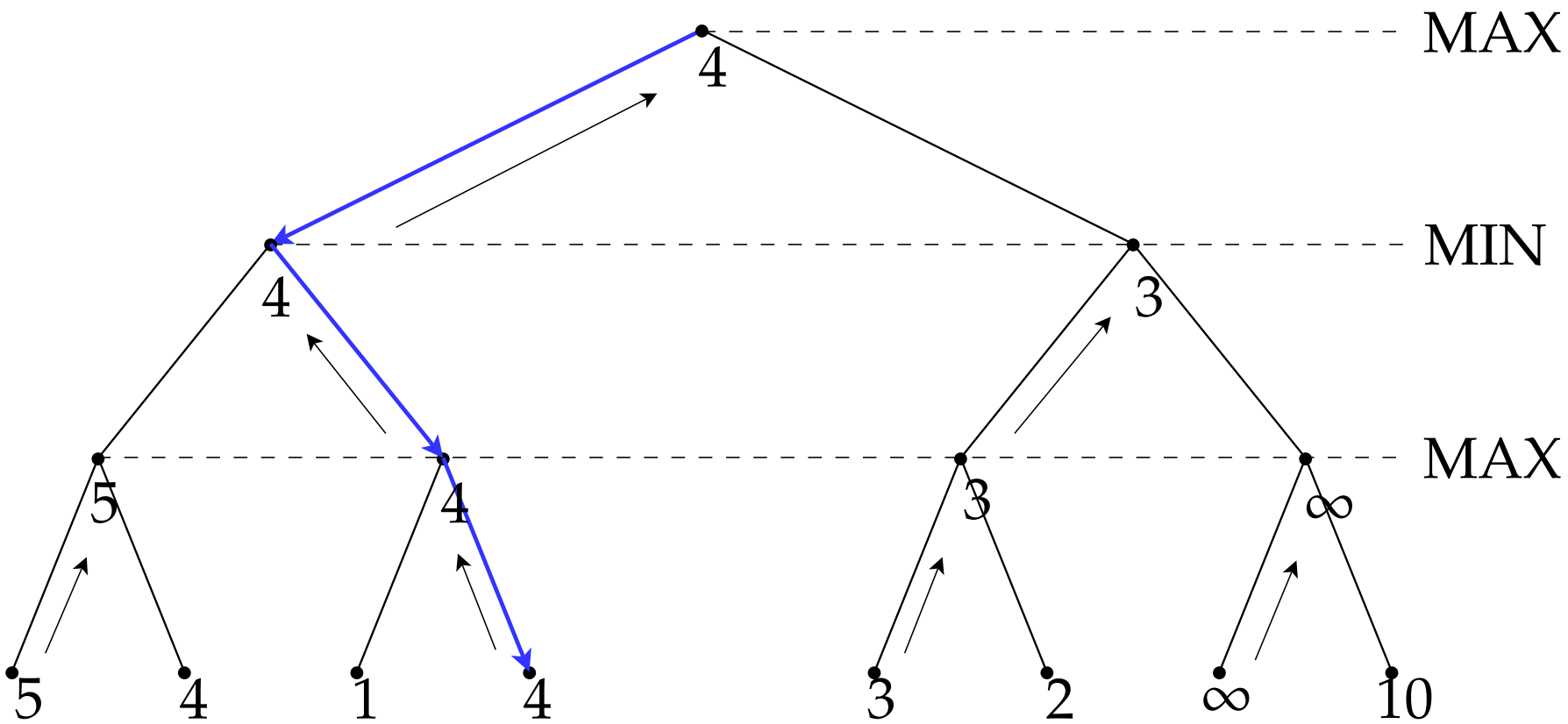
Algorytm minimax (lub min-max)

- Szkic: dla danej pozycji początkowej rozwijane jest drzewo gry do pewnej zadanej głębokości. Pozycjom końcowym (liściom, terminalom) nadawane są ***oceny liczbowe***. Następuje „przechodzenie" drzewa od dołu propogujące oceny w górę drzewa. W efekcie ocenione zostają możliwe ruchy pochodzące od stanu początkowego.
- Funkcja oceny pozycji (ang. *evaluation function*) jest zwykle pewną funkcją heurystyczną zgodną z wiedzą i intuicją na temat danej gry.
- Np. dla szachów: różnica pomiędzy wartością bierek białych i czarnych.
- Zwyczajowo graczy nazywa się: minimalizującym i maksymalizującym.
- Zwycięstwo gracza minimalizuącego reprezentuje $-\infty$.
- Zwycięstwo gracza maksymalizującego reprezentuje $+\infty$.
- Gdy drzewo gry jest odpowiednio małe (lub gdy badana jest ścisła końcówka) i osiągnięte zostaną w drzewie faktyczne stany końcowe gry, to możliwe wartości liści wynoszą: $-\infty$, $+\infty$, 0 (remis). Wtedy heurystyczna ocena jest niepotrzebna.

Algorytm minimax — ilustracja



Algorytm minimax — ilustracja



Minimax — pojęcia, oznaczenia

- półruch (ang. *ply* lub *half-move*) — nazwa oznaczająca ruch jednego z graczy; przesuwanie się o jeden poziom w drzewie liczone jest zwyczajowo jako $\pm\frac{1}{2}$, dopiero 2 półruchy każdego z graczy traktowane są jako całe posunięcie.
- współczynnik rozgałęziania (ang. *branching factor*) — przeciętna lub stała liczba ruchów przypadająca na każdego z graczy w danej grze; oznaczany zwykle literą b (np. dla szachów w grze środkowej $b \approx 40$).
- horyzont przeszukiwań — zadana do zbadania liczba poziomów drzewa; oznaczany zwykle literą D.
- efekt horyzontu — ogólna wada wszystkich procedur minimaksowych wynikająca z ograniczonej głębokości przeszukiwania; zjawisko polegające na tym, że pewien stan tuż poza horyzontem przeszukiwań może całkowicie zmieniać ocenę pozycji i np. okazać się katastrofalny dla gracza, pomimo że poziom wyżej pozycja była atrakcyjna (lub odwrotnie).
- Quiescence — technika pomocnicza łagodząca częściowo efekt horyzontu, polegająca na rozwijaniu stanów na granicy horyzontu przeszukiwań (i poza nim) aż do osiągnięcia tzw. *pozycji cichych* (np. nie zawierających możliwych zbić).

Algorytm minimax (lub min-max)

```
1: procedura MMEvaluateMaxState(s, d, D)
2:     jeżeli IsTerminal(s, d, D) to zwróć h(s)        ▷ h(s) — heurystyczna ocena pozycji
3:     v := −∞
4:     wygeneruj zbiór stanów {t} potomnych dla s
5:     dla wszystkich t wykonaj
6:         w :=MMEvaluateMinState(t, d + 1/2, D)
7:         jeżeli s jest korzeniem to zapamiętaj w jako ocenę ruchu s → t
8:         v := max{v, w}
9:     zwróć v
```

```
1: procedura MMEvaluateMinState(s, d, D)
2:     jeżeli IsTerminal(s, d, D) to zwróć h(s)        ▷ h(s) — heurystyczna ocena pozycji
3:     v := ∞
4:     wygeneruj zbiór stanów {t} potomnych dla s
5:     dla wszystkich t wykonaj
6:         w :=MMEvaluateMaxState(t, d + 1/2, D)
7:         jeżeli s jest korzeniem to zapamiętaj w jako ocenę ruchu s → t
8:         v := min{v, w}
9:     zwróć v
```

Minimax — punkty stopu

- IsTerminal(s, d, D) — metoda sprawdzająca, czy jesteśmy w punkcie stopu, do zaimplementowania zgodnie z zasadami danej gry.
- Zwykle sprawdza się, czy zachodzi któryś z warunków:
  - $d \geqslant D$ i stan s jest *cichy*,
  - $h(s) = \pm\infty$ — stan s jest *zwycięski*,
  - $h(s) \neq \pm\infty$, ale stan s jest *remisowym* wg zasad gry
    (np. w szachach: pat, wieczny szach, trzykrotne powtórzenie pozycji).

Szachy — ocena pozycji

- Przykład funkcji zaproponowanej przez C. Shannona (1949):

$$f(s) = 200(K_s - K'_s) + 9(Q_s - Q'_s) + 5(R_s - R'_s) + 3(B_s - B'_s + N_s - N'_s) + 1(P - P') \quad (\textit{materialne})$$
$$- 0.5(D_s - D'_s + S_s - S'_s + I_s - I'_s) + 0.1(M_s - M'_s), \quad (\textit{pozycyjne}) \qquad (12)$$

gdzie K, Q, R, B, N, P oznaczają odpowiednio liczbę: króli, hetmanów, wieży, gońców, skoczków i pionów;
D, S, I oznaczają piony: podwojone, zablokowane, odizolowane;
M oznacza mobilność (liczbę dozwolonych ruchów);
symbol $'$ oznacza powyższe wielkości dla strony przeciwnej.

- Współcześnie, funkcje oceny wyraża się w tzw. centypionach.
- Jeden pion = 100 centypionów. Najmniejsza przewaga pozycyjna to 1 centypion.
- Elementy uwzględniane w ocenie:
  - kontrola nad centrum,
  - aktywność figur (i ich „łączność”),
  - struktura pionów,
  - bezpieczeństwo króla,
  - piony idące do przemiany,
  - przestrzeń,
  - ...
- Popularne także podejścia nastrajające parametryczną funkcję oceny (np. genetycznie).

Warcaby — ocena pozycji

- Przykład funkcji materialno-strukturalnej (M. Bożykowski, 2009):

$$f(s) = 13(P_s - P'_s) + 85(K_s - K'_s) \quad (\textit{materialne}) \\ + 6(T_s - T'_s) + 1(I_s - I'_s) - 1(F_s - F'_s), \quad (\textit{pozycyjne}) \tag{13}$$

gdzie: P, K oznaczają odpowiednio liczbę: pionów i damek;
T, I, F oznaczają odpowiednio liczbę pionów: oddalonych o 1 pole od przemiany, niezbijalnych, unieruchomionych;
symbol $'$ oznacza powyższe wielkości dla strony przeciwnej.

- Przykład funkcji materialno-wierszowej (M. Bożykowski, 2009):

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

gdzie: $P_s(i)$ oznacza liczbę pionów w i-ym wierszu warcabnicy (wersja stupolowa);
wagi nastrojone genetycznie: $w = (2, 1, 2, 2, 2, 2, 1, 3, 6)$;

Minimax — złożoność obliczeniowa

- R_d — liczba stanów, które trzeba odwiedzić w drzewie o d poziomach, aby poznać dokładną wartość danego stanu — suma ciągu geometrycznego.
- Podejście rekurencyjne (przydatne do analizy dalszych algorytmów drzewkowych):

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

- Rozwinięcie:

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

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

- W uproszczeniu schemat: $O(b \cdot b \cdots b)$ — d-krotnie b.

Spis treści

Przycinanie α–β

- Wielu niezależnych odkrywców: **(Samuel, 1952)**, **(Edwards i Hart, 1963)**, **(Brudno, 1963)**, **(Newell i Simon, 1958; 1976)**.
- Dokładna analiza: **D. Knuth, R. Moore, R. (1975), "An analysis of alpha-beta pruning", *Artificial Intelligence*, 6(4), 293–326.**

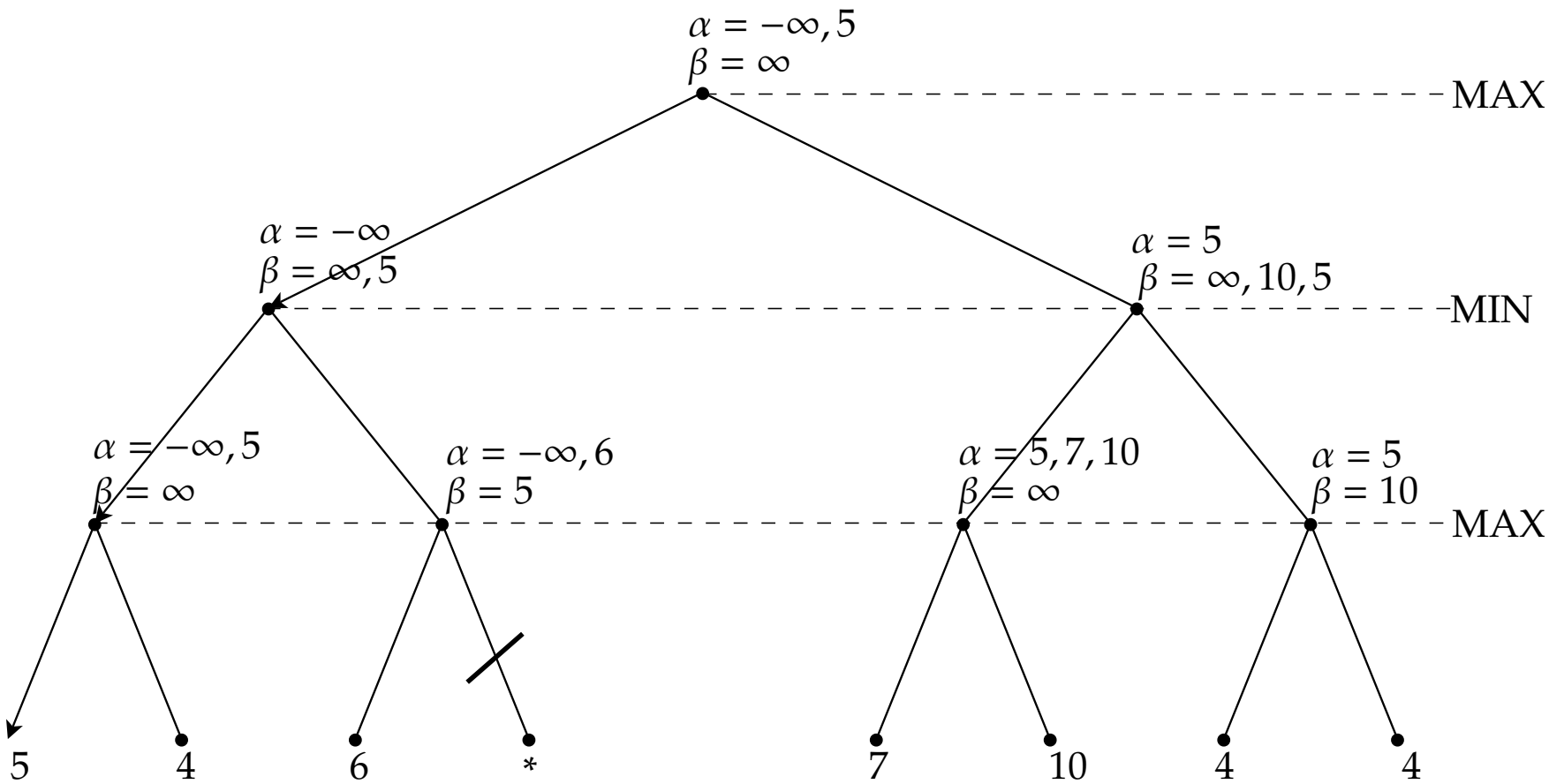
  [https://pdfs.semanticscholar.org/dce2/6118156e5bc287bca2465a62e75af39c7e85.pdf]
- Należy do klasy algorytmów: branch and bound.
- W trakcie analizy propagowane są w dół i górę drzewa wartości:
  α — gwarantowana dotychczas wypłata gracza maksymalizującego,
  β — gwarantowana dotychczas wypłata gracza minimalizującego.
- W wywołaniu dla korzenia zadaje się $\alpha = -\infty$, $\beta = \infty$.
- Dzieci (i ich poddrzewa) są analizowane dopóki $\alpha < \beta$.
- W momencie gdy $\alpha \geqslant \beta$, przestajemy rozpatrywać kolejne dzieci (i ich poddrzewa) — nie będą one miały wpływu na wynik całego drzewa, są wynikiem nieoptymalnego postępowania któregoś z graczy.
- $\alpha > \beta$ to logiczna sprzeczność; przypadek równości można dołączyć do wykluczeń, ponieważ nie wnosi poprawy wyniku.
- Pomimo redukcji drzewa przycinanie α–β oddaje te same wyniki (oceny ruchów) co minimax.

Przycinanie α–β

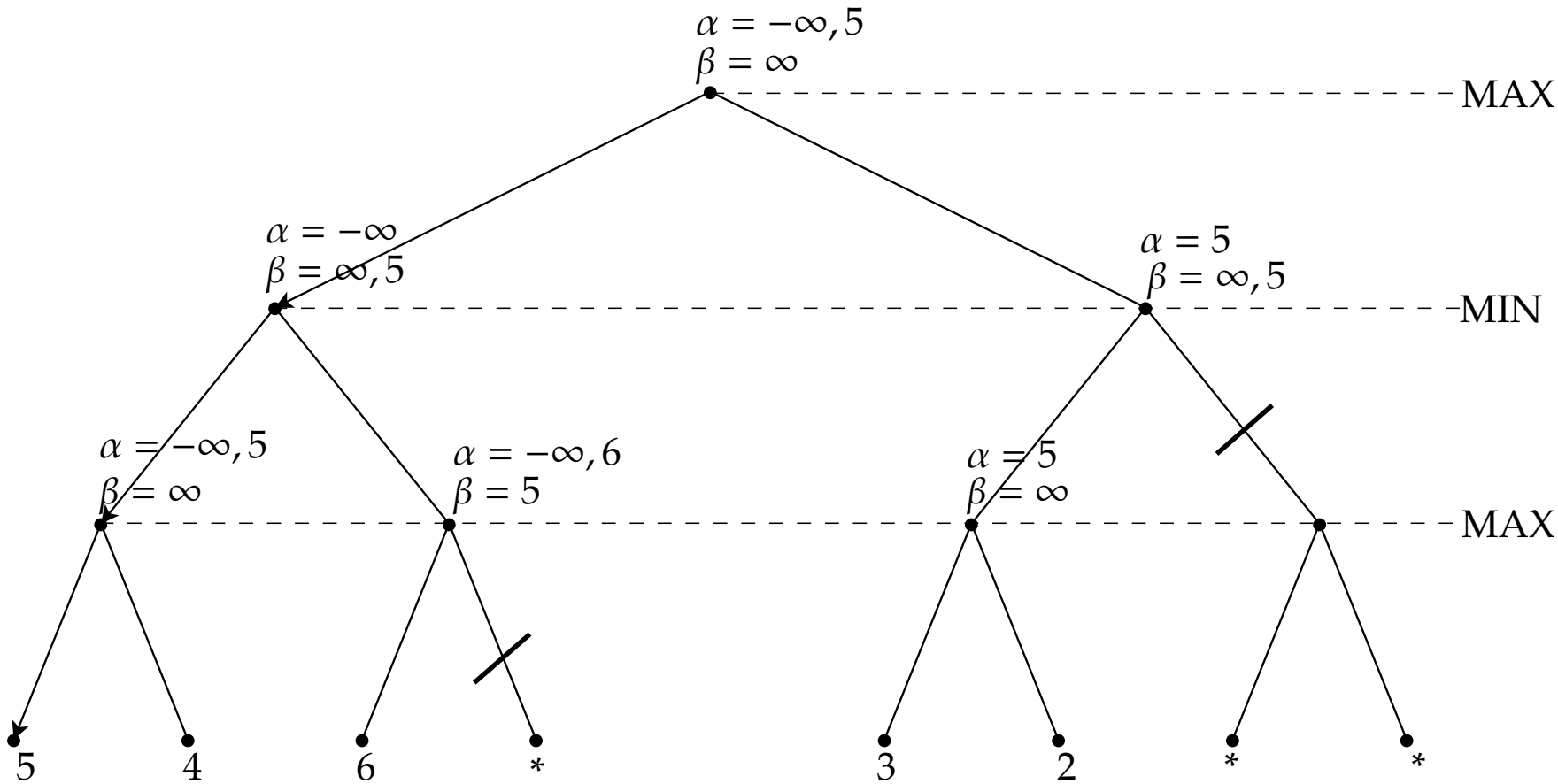
```
1: procedura AlphaBetaEvaluateMaxState(s, d, D, α, β)
2:     jeżeli IsTerminal(s, d, D) to zwróć h(s)          ▷ h(s) — heurystyczna ocena pozycji
3:     wygeneruj zbiór stanów {t} potomnych dla s
4:     dla wszystkich t wykonaj
5:         v :=AlphaBetaEvaluateMinState(t, d + 1/2, D, α, β)
6:         jeżeli s jest korzeniem to zapamiętaj v jako ocenę ruchu s → t
7:         α := max{α, v}
8:         jeżeli α ≥ β to zwróć α                       ▷ przycięcie (!) — kolejne t nie będą sprawdzane
9:     zwróć α
```

```
1: procedura AlphaBetaEvaluateMinState(s, d, D, α, β)
2:     jeżeli IsTerminal(s, d, D) to zwróć h(s)          ▷ h(s) — heurystyczna ocena pozycji
3:     wygeneruj zbiór stanów {t} potomnych dla s
4:     dla wszystkich t wykonaj
5:         v :=AlphaBetaEvaluateMaxState(t, d + 1/2, D, α, β)
6:         jeżeli s jest korzeniem to zapamiętaj v jako ocenę ruchu s → t
7:         β := min{β, v}
8:         jeżeli α ≥ β to zwróć β                       ▷ przycięcie (!) — kolejne t nie będą sprawdzane
9:     zwróć β
```

Przycinanie α–β — przykład 1



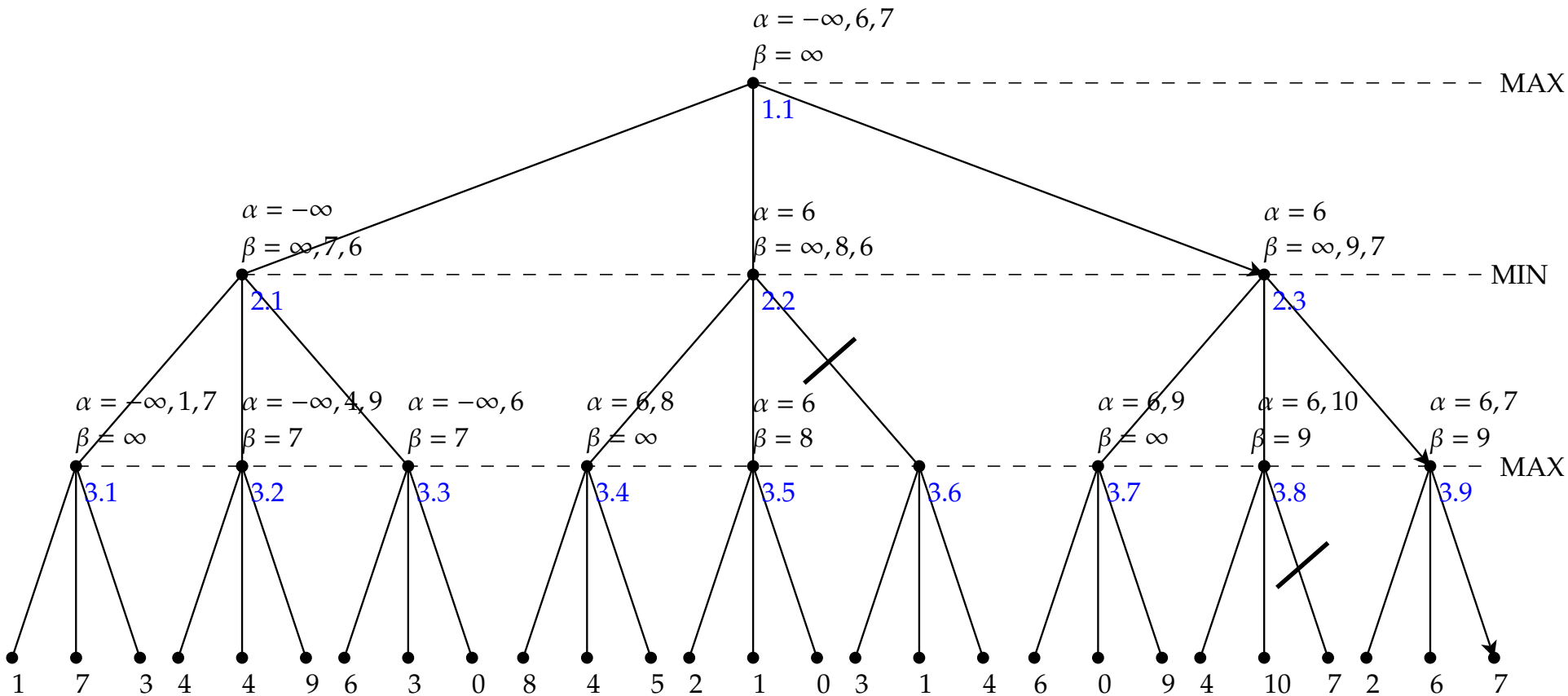
Przycinanie α–β — przykład 2



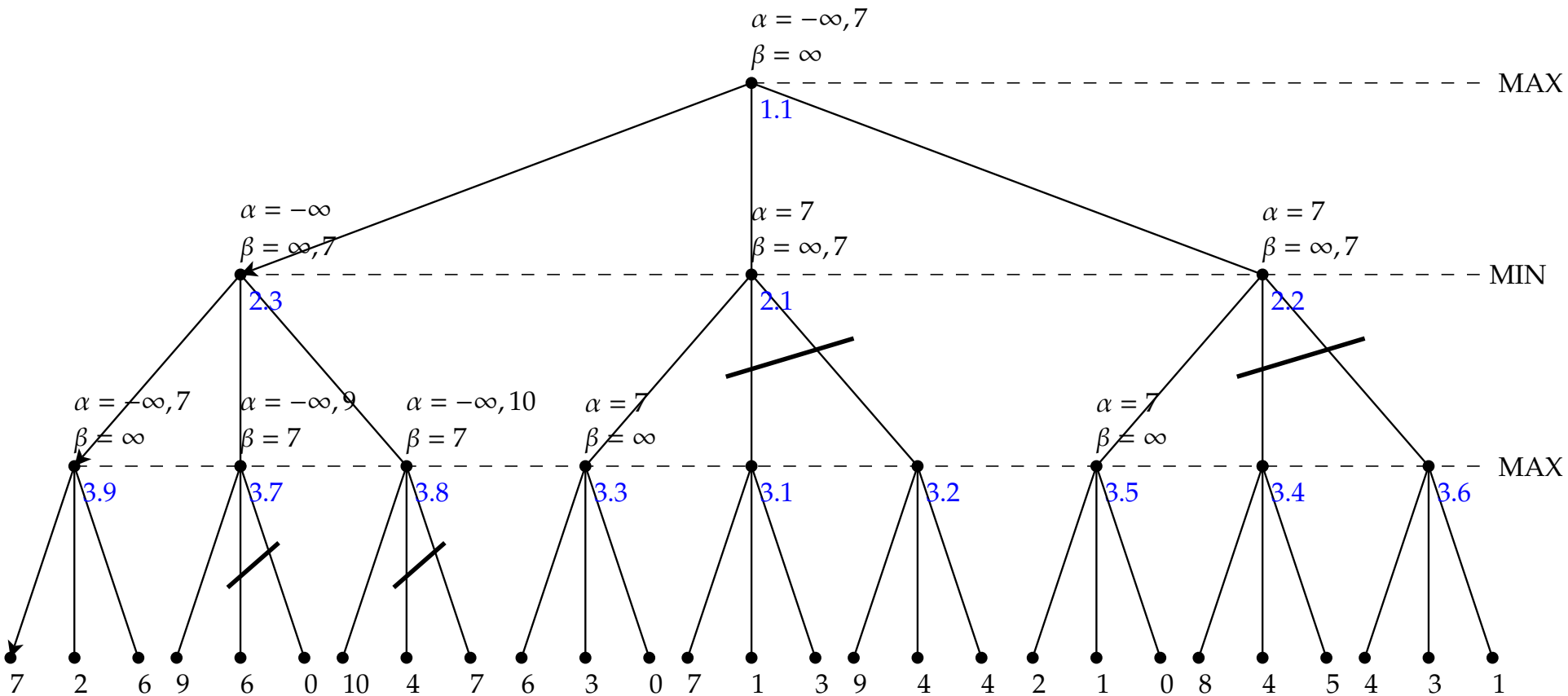
Przycinanie α–β — złożoność

- Złożoność obliczeniowa zależna od *porządku* odwiedzania stanów potomnych.
- Sprzyjają sytuacje, gdy potomek powodujący odcięcie jest bliżej początku listy.
- Istnieją techniki pomocnicze sortowania potomków, aby zwiększyć częstość przycięć (ale w ogólności dobry porządek nie jest znany z góry),
- W przypadku pesymistycznym złożoność (dla d poziomów): $O\left(b^d\right)$.
- W przypadku optymistycznym złożoność: $O\left(b^{d/2}\right)$.

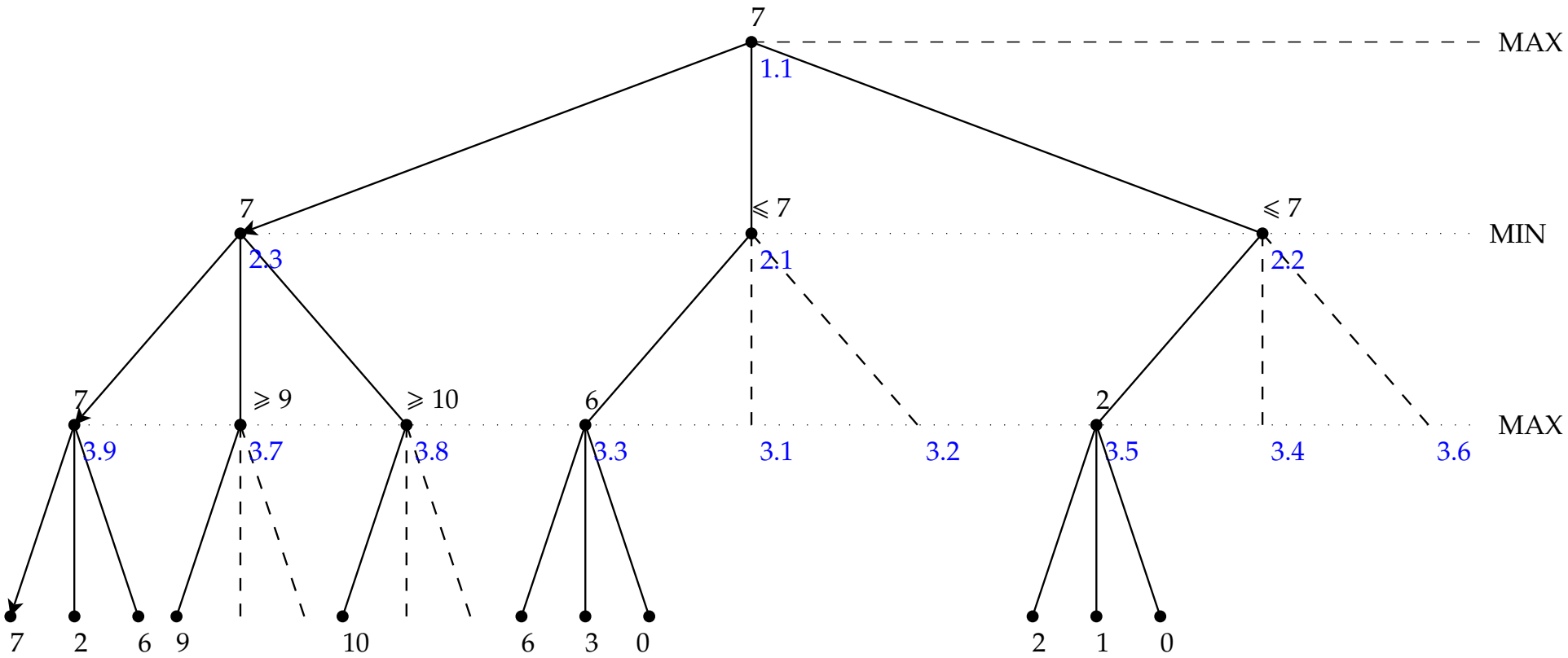
Przycinanie α–β — przykład 3



Przycinanie α–β — przykład 3a



Przycinanie α–β — przykład 3b



Przycinanie α–β — złożoność optymistyczna

- Znamy albo dokładną wartość stanu, albo ograniczenie (dolne lub górne) na tę wartość.
- Aby ustalić dokładną wartość, wystarczają (w przypadku optymistycznym): dokładna wartość 1 dziecka i ograniczenia dla $b-1$ pozostałych dzieci.
- Aby ustalić ogranicznie, wystarcza (w przypadku optymistycznym): dokładna wartość 1 dziecka.
- R_d — minimalna liczba stanów (odległych o d poziomów od danego stanu), które trzeba odwiedzić, aby poznać dokładną wartość.
- S_d — minimalna liczba stanów (odległych o d poziomów od danego stanu), które trzeba odwiedzić, aby poznać ograniczenie.
- Wartości brzegowe: $R_0 = S_0 = 1$.
- Rekurencje:

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

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

- Łącząc, otrzymujemy:

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

- Dla przykładu z poprzedniego slajdu: $R_3 = b^2 + b - 1 = 11$.

Przycinanie α–β — złożoność optymistyczna

- Oszacowanie optymistycznej liczby stanów:

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

- Efektywny współczynnik rozgałęziania co każde 2 poziomy jest mniejszy niż $2b$.
  A więc dla jednego poziomu jest mniejszy niż $\sqrt{2b}$.
- Rozwinięcie:

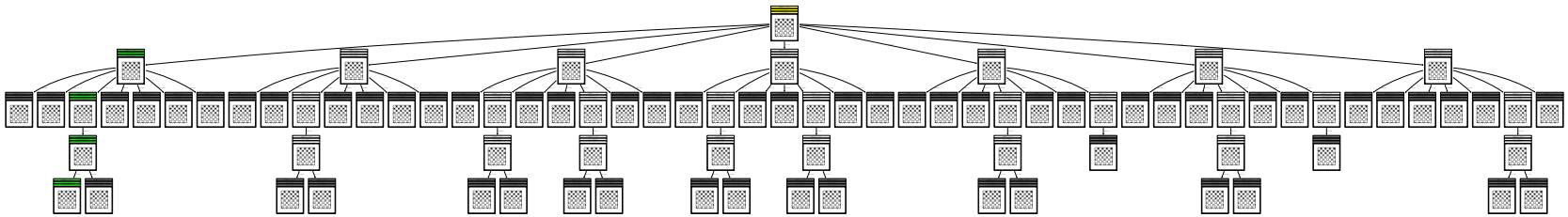
$$R_d < 2bR_{d-2} < (2b)^2 R_{d-4} < (2b)^3 R_{d-6} < \cdots < (2b)^k R_{d-2k} \tag{22}$$

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

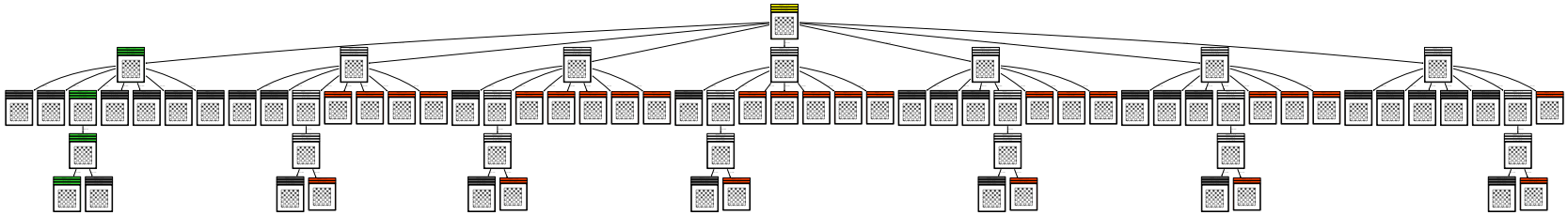
- W uproszczeniu schemat: $O(b \cdot 1 \cdot b \cdot 1 \cdots b \cdot 1)$ — $d/2$-krotnie b.
- Szacuje się, że w przypadku średnim złożoność jest $\sim O\left(b^{3d/4}\right)$.

Początki drzewa warcab

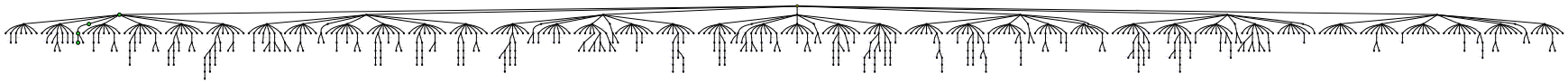
- ***min-max* + *Quiescence***, głębokość (dla pozycji cichych) 1.0, stanów 86:



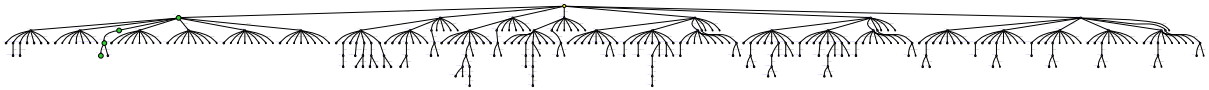
- ***przycinanie α-β* + *Quiescence***, głębokość 1.0, stanów 78:



- ***min-max* + *Quiescence***, głębokość 1.5 (gdy cicho), stanów 693:



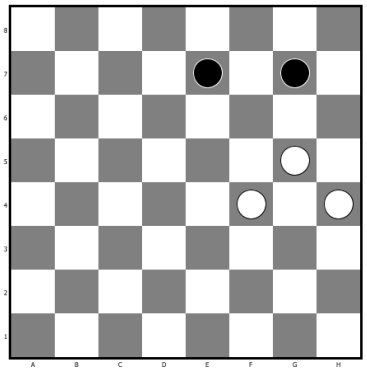
- ***przycinanie α-β* + *Quiescence***, głębokość 1.5, stanów 323:



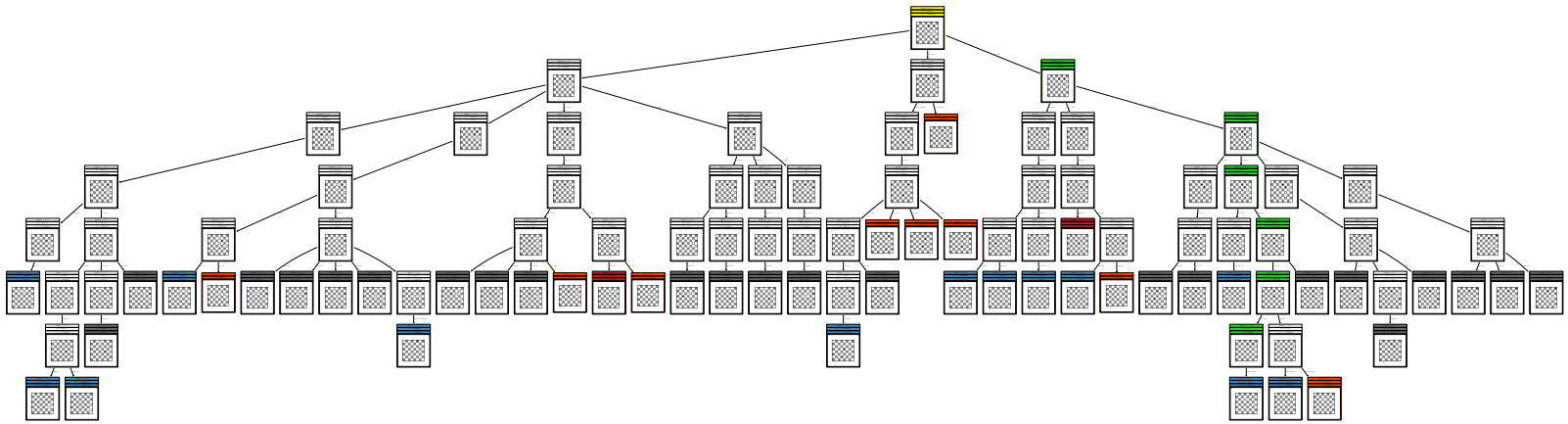
[Wyniki generowane przez bibliotekę *SaC*: `https://pklesk.github.io/sac`, ilustracje dzięki: *Graphviz* `https://www.graphviz.org`.]

Końcówka warcabowa — przykład 1

- Białe rozpoczynają i wygrywają w 4 posunięciach:



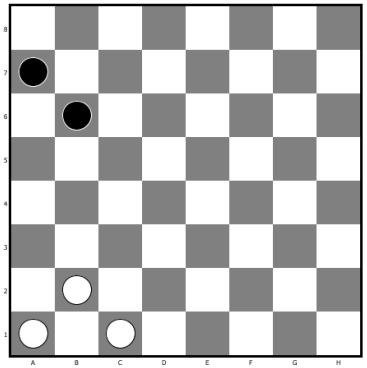
- ***przycinanie α-β + Quiescence***, głębokość 2.5, stanów 100:



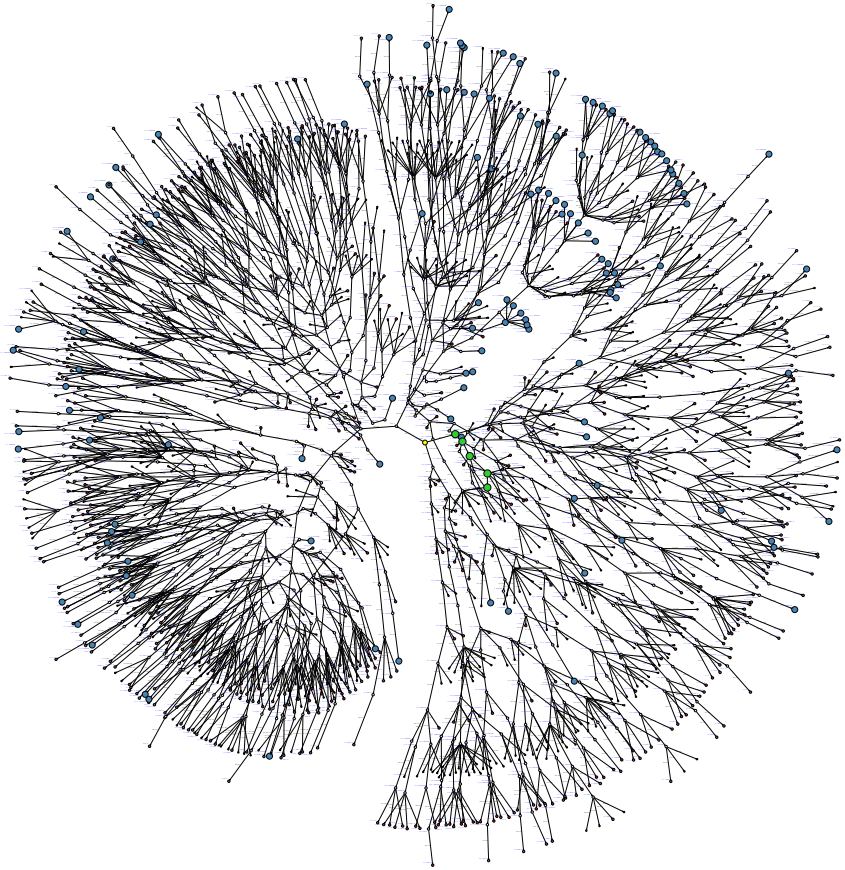
- Wariant główny: $(G5 : H6, G7 : F6, F4 : G5, F6 : E5, G5 : F6, E5 : G7, H6 : F8 : D6)$.

Końcówka warcabowa — przykład 2

- Białe rozpoczynają. Kto wygra?



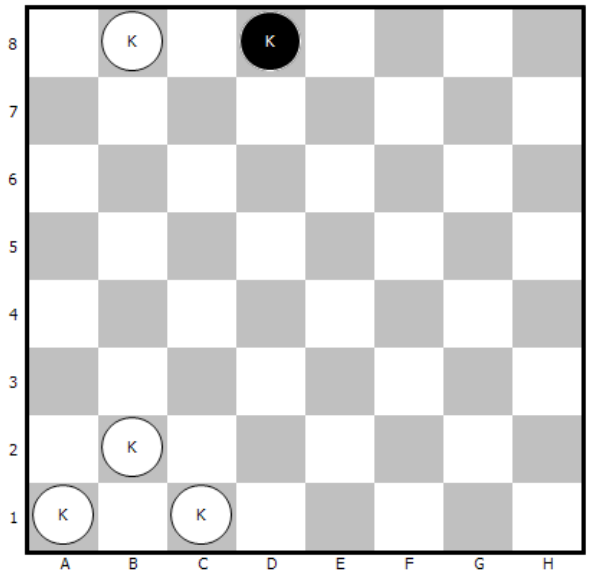
- ***przycinanie α-β + Quiescence***, głębokość 5.5, stanów 2 845:



Końcówka warcabowa — przykład 3

- "4 damki vs 1 damka"

pozycja:



wynik z biblioteki ***SaC***:

```
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]
```

- Ilustracja wariantu głównego: [https://github.com/pklesk/sac/releases/download/1.0.3/sac-1.0.3-userguide.pdf#page=150]