Przeszukiwanie lokalne i gry

Paweł Rychlikowski

Instytut Informatyki UWr

5 kwietnia 2023

Najpierw jeszcze trochę o więzach

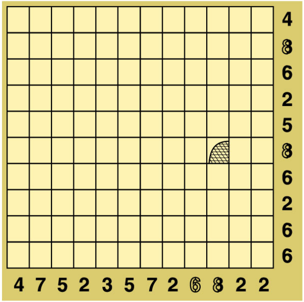
Przypomnienie

Dwie strategie rozwiązywania problemów więzowych:

1. Backtracking + dostosowana propagacja więzów
2. Użycie Constraint Solver z zaimplementowanym backtrackingiem i propagacją

(ten drugi sposób wymaga pisania specjalnego programu tłumaczącego, który generuje specyficzne więzy dla instancji problemu)

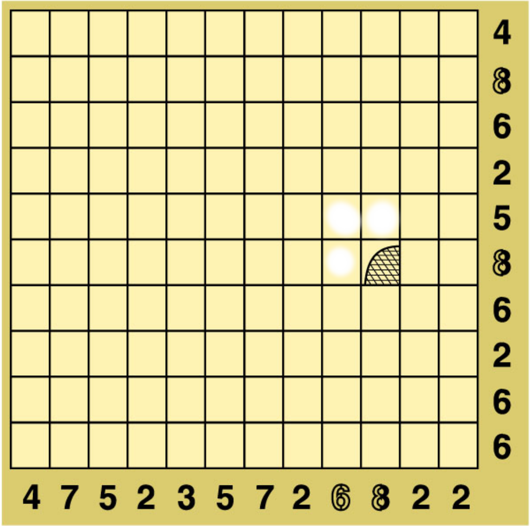
Burze. Przypomnienie



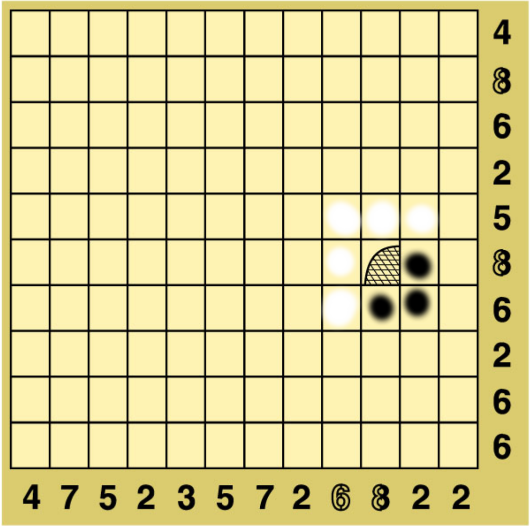
Zasady

1. Radary mówią, ile jest pól burzowych w wierszach i kolumnach.
2. Burze są prostokątne.
3. Burze nie stykają się rogami.
4. Burze mają wymiar co najmniej 2×2.

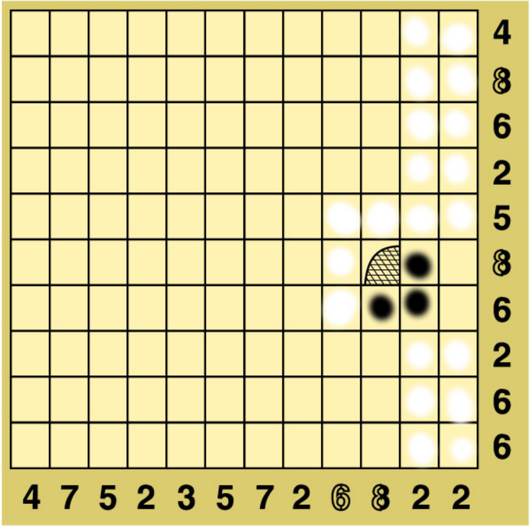
Burze: wnioskowanie



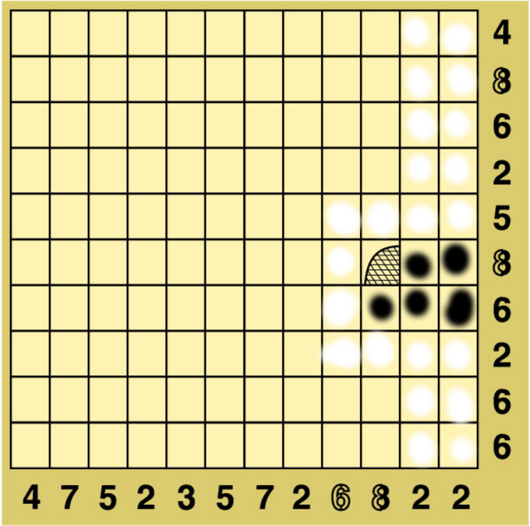
Burze: wnioskowanie



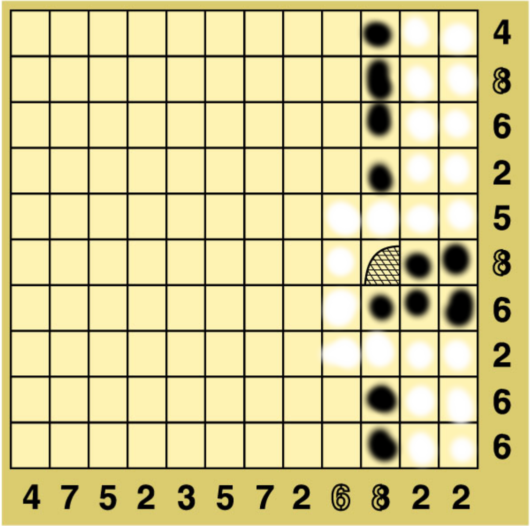
Burze: wnioskowanie



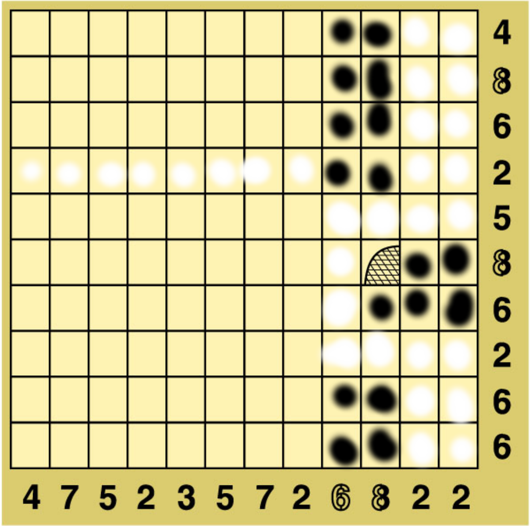
Burze: wnioskowanie



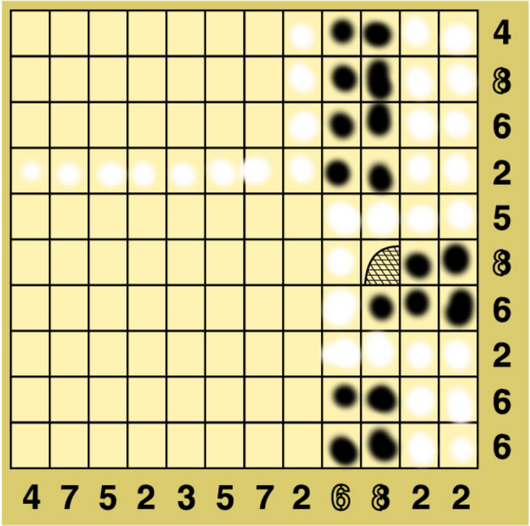
Burze: wnioskowanie



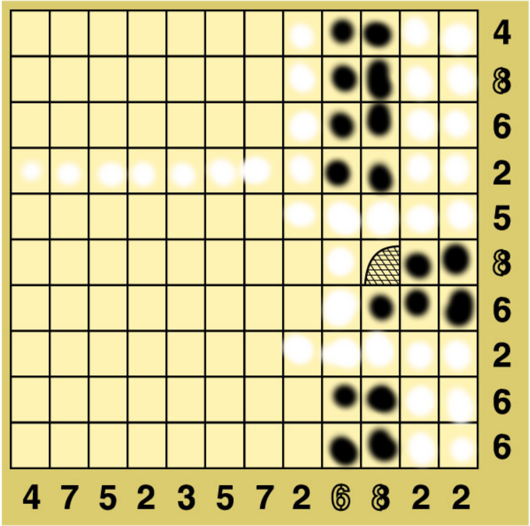
Burze: wnioskowanie



Burze: wnioskowanie



Burze: wnioskowanie



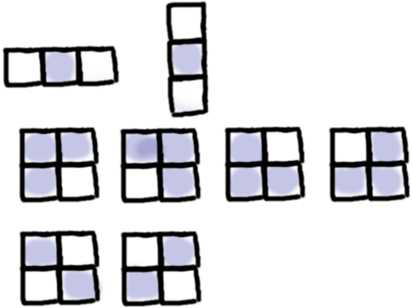
Kodowanie burz

Do wykorzystania w Constraint Solverze

- Zmienne, dziedziny: piksele, 0..1
- Radary: $b_1 + b_2 + \cdots + b_n = K$
- Prostokąty: ?
- Co najmniej 2×2?
- Nie stykają się rogami.

- Jak wygląda każdy kwadrat 2×2?
- Jak wygląda każdy prostokąt 1×3 albo 3×1?

Zabronione układy



Pytanie z zeszłego tygodnia

Jak wyrazić to językiem relacji arytmetycznych?

Warunek dobrych 3 pól

Mamy zmienne A, B, C

- $A + 2B + 3C \neq 2$
- $B \times (A + C) \neq 2$

Reifikacja. Warunki w więzach

Naturalne sformułowanie

Jeżeli środkowy piksel jest ustawiony, to wówczas przynajmniej 1 z otaczających go jest jedynką.

$$B \Rightarrow (A + C > 0)$$

Reifikacja (cd)

- Inny przykład: `A #<=> (B #> C)` (nawias dla czytelności)
- Naturalna propagacja:
  - Ustalenie A dorzuca więz
  - Jak wiemy, czy prawdziwy jest `B #> C`, to znamy wartość `A`

Więz uniweralny w SWI-Prologu

tuples_in

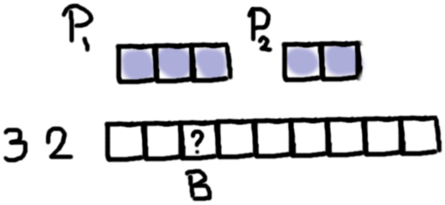
Wymieniamy explicite krotki wartości, jakie może przyjmować krotka zmiennych

Uwaga

Zauważmy, że ten więz pasuje do lokalnych warunków dla burz, na przykład dla prostokątów 3×1::

```
tuples_in( [[A,B,C]], [ [0,0,0], [1,1,0], [1,0,0],
[0,1,1], [0,0,1], [1,1,1], [1,0,1]])
```

Reifikacja i obrazki logiczne



- Użycie zmiennych P_1 i P_2 określających położenie bloku pozwala zmniejszyć dziedziny ($|P_1| + |P_2|$ zamiast $|P_1| \times |P_2|$ (mniejsze zużycie pamięci, niezmniejszona liczba kombinacji)
- Zmienna B ma wartość logiczną:

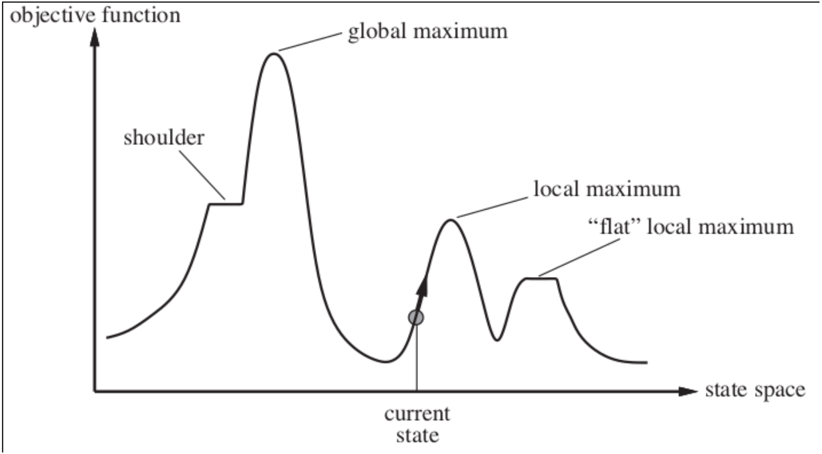
  3 *jest przykryte przez blok rozpoczynający się w P_1 lub przez blok rozpoczynający się w P_2*

Na tym skończymy o więzach i przejdziemy do przeszukiwania lokalnego

Przeszukiwania lokalne (ogólnie)

- Powiemy sobie o paru ideach związanych z przeszukiwaniem lokalnym.
- Można je wykorzystywać w zadaniach więzowych (MinConflicts z poprzedniego wykładu), ale nie tylko.

Krajobraz przeszukiwania lokalnego



Czym może być funkcja, którą minimalizujemy?

1. Liczbą **niespełnionych** więzów.

Czym może być funkcja, którą minimalizujemy?

1. Liczbą **niespełnionych** więzów.
2. **Wagą** niespełnionych więzów. Porównaj więzy:
    1. Nauczyciel ma tylko z jedną klasą lekcje na raz
    2. nikt nie ma dwóch biologii jednego dnia.

Czym może być funkcja, którą minimalizujemy?

1. Liczbą **niespełnionych** więzów.
2. **Wagą** niespełnionych więzów. Porównaj więzy:
   1. Nauczyciel ma tylko z jedną klasą lekcje na raz
   2. nikt nie ma dwóch biologii jednego dnia.
3. Czymś niezwiązanym bezpośrednio z więzami
   - produktywnością zespołu robotników (maksymalizemy, nie minimalizujemy!)
   - zadowoleniem gości weselnych z towarzystwa przy stolikach,
   - potencjalnym zyskiem sklepu,
   - dopasowaniem do danych uczących

Czym może być funkcja, którą minimalizujemy?

1. Liczbą **niespełnionych** więzów.
2. **Wagą** niespełnionych więzów. Porównaj więzy:
    1. Nauczyciel ma tylko z jedną klasą lekcje na raz
    2. nikt nie ma dwóch biologii jednego dnia.
3. Czymś niezwiązanym bezpośrednio z więzami
    - produktywnością zespołu robotników (maksymalizemy, nie minimalizujemy!)
    - zadowoleniem gości weselnych z towarzystwa przy stolikach,
    - potencjalnym zyskiem sklepu,
    - **dopasowaniem do danych uczących**

Czym może być funkcja, którą maksymalizujemy?

Uwaga

Ważna część uczenia maszynowego dotyczy **maksymalizacji dopasowania do danych uczących**

Hill climbing

Hill climbing jest chyba najbardziej naturalnym algorytmem inspirowanym poprzednim rysunkiem.

- Dla stanu znajdujemy wszystkie następniki i wybieramy ten, który ma największą wartość.
- Powtarzamy aż do momentu, w którym nie możemy nic poprawić

Problem

Oczywiście możemy utknąć w lokalnym maksimum.

Hill climbing z losowymi restartami

Uwaga

Możemy podjąć dwa działania, oba testowaliśmy w obrazkach logicznych:

1. Dorzucać ruchy niekoniecznie poprawiające (losowe, ruchy w bok)
2. Gdy nie osiągamy rozwiązania przez dłuższy czas rozpoczynamy od początku.

Hill climbing + random restarts (w trywialny sposób) jest algorytmem zupełnym z p-stwem 1 (bo „kiedyś” wylosujemy układ startowy)

Inne warianty Hill climbing

a) Stochastic hill climbing – wybieramy losowo ruchy w górę (p-stwo stałe, albo zależne od wielkości skoku).
b) First choice hill climbing – losujemy następnika tak długo, aż będzie on ruchem w górę
    - dobre, jeżeli następników jest bardzo dużo

Uwaga

Idee z tego i kolejnych algorytmów można dowolnie mieszać – na pewno coś wyjdzie!

Symulowane wyżarzanie

- Motywacja fizyczna: ustalanie struktury krystalicznej metalu.
- Jeżeli będziemy ochładzać powoli, to metal będzie silniejszy (bliżej globalnego minimum energetycznego).
- **Symulowane** wyżarzanie – próba oddania tej idei w algorytmie.

Algorytm

Symulujemy opadającą temperaturę, prawdopodobieństwo ruchu chaotycznego zależy malejąco od temperatury.

Symulowane wyżarzanie (2)

- Przykładowa implementacja bazuje na first choice hill climbing.
- Jak wylosowany ruch (**r**) jest lepszy (czyli $\Delta F > 0$), to go wykonujemy (maksymalizacja F).
- W przeciwnym przypadku wykonujemy ruch **r** z p-stwem $p = e^{\frac{\Delta F}{T}}$
- Pilnujemy, żeby T zmniejszało się w trakcie działania (i było cały czas dodatnie)

Komentarze do wzoru

- $\Delta F \leq 0$, $T > 0$, czyli $0 \leq p \leq 1$.
- Im większe pogorszenie, tym mniejsze p-stwo
- Im większa temperatura, tym większe p-stwo.

Taboo search

Problem

Być może płaskie maksimum lokalne.

Rozwiązanie

Dodajemy pamięć algorytmowi, zabraniamy powtarzania ostatnio odwiedzanych stanów.

Local beam search

- Zamiast pamiętać pojedynczy stan, pamiętamy ich k (wiązkę).
- Generujemy następniki dla każdego z k stanów.
- Pozostawiamy k liderów.

Uwaga 1

To nie to samo co k równoległych wątków hill-climbing (bo uwaga algorytmu może przerzucać się do bardziej obiecujących kawałków przestrzeni)

Uwaga 2

Beam search jest bardzo popularnym algorytmem w różnych zadaniach wykorzystujących sieci neuronowe do modelowania sekwencji (np. tłumaczenie maszynowe).

Algorytmy ewolucyjne

- Zarządzamy populacją osobników (czyli np. pseudorozwiązań jakiegoś problemu więzowego).
- Mamy dwa rodzaje operatorów:
  a) Mutacja, która z jednego osobnika robi innego, podobnego.
  b) Krzyżowanie, która z dwóch osobników robi jednego, w jakiś sposób podobnego do „rodziców”.
- Nowe osobniki oceniane są ze względu na wartość funkcji przystosowania
- Przeżywa k najlepszych.

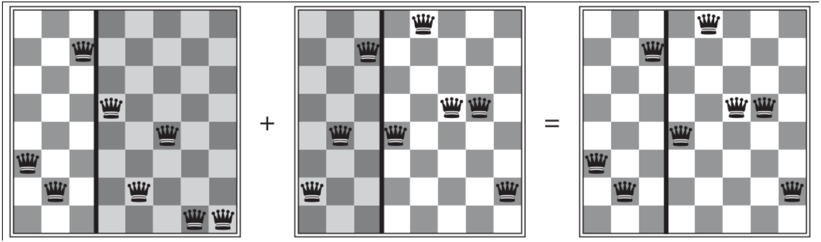
Uwaga

Zauważmy, że choć zmienił się język, jeżeli pominiemy krzyżowanie, to otrzymamy wariant Local beam search (mutacja jako krok w przestrzeni stanów).

Krzyżowanie. Przykład

Pytanie

Czym mogłoby być krzyżowanie dla zadania z N hetmanami?



Algorytmy ewolucyjne. Kilka uwag

1. Krzyżowanie i mutacje można zorganizować tak, że najpierw powstają dzieci, a następnie się mutują z pewnym prawdopodobieństwem.
2. Wybór osobników do rozmnażania może zależeć od funkcji dopasowania (większe szanse na reprodukcję mają lepsze osobniki)
3. Można mieć wiele operatorów krzyżowania i mutacji.

Przeszukiwanie w grach

Przykładowa gra

- Gracz **A** wybiera jeden z trzech zbiorów:
  1. $\{-50, 50\}$
  2. $\{1, 3\}$
  3. $\{-5, 15\}$
- Następnie gracz **B** wybiera liczbę z tego zbioru.

Pytanie

Co powinien zrobić **A**, żeby uzyskać jak największą liczbę?

Przykładowa gra

Nasza gra

1. $\{-50, 50\}$
2. $\{1, 3\}$
3. $\{-5, 15\}$

Racjonalny wybór dla **A** zależy od (modelu) gracza **B**

- Współpracujący: Oczywiście 1.
- Losowy (z $p = \frac{1}{2}$)) Wybór 3 (średnio 5)
- „Złośliwy":wybór 2 (gwarantujemy wartość 1)

- Nieco inna rodzina zadań wyszukiwania, w których mamy dwóch (lub więcej) agentów.
- Interesy agentów są (przynajmniej częściowo) rozbieżne.
- Rozgrywka przebiega w turach, w których gracze na zmienę wybierają swoje ruchy.

Definicja

Gra jest problemem przeszukiwania, zadanym przez następujące składowe:

1. Zbiór stanów, a w nim S_0, czyli stan początkowy
2. player(s), funkcja określająca gracza, który gra w danym stanie.
3. actions(s) – zbiór ruchów możliwych w stanie s
4. result(s,a) – funkcja zwracająca stan powstały w wyniku zastosowania akcji a w stanie s.
5. terminal(s) – funkcja sprawdzająca, czy dany stan kończy grę.
6. utility(s, player) – funkcja o wartościach rzeczywistych, opisująca wynik gry z punktu widzenia danego gracza.

Gra o sumie zerowej

Definicja

W grze o sumie zerowej suma wartości stanów terminalnych dla wszystkich graczy jest stała (niekonieczne zera, ale...)

Konsekwencje:

- Zysk jednego gracza, jest stratą drugiego.
- Kooperacja nic nie daje.

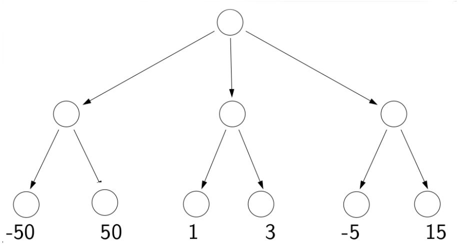
Uwaga

Zaczniemy od gier o sumie zerowej i gracza, wcześniej nazwanego złośliwym (lepiej go nazwać racjonalnym)

Różnice między grami a zwykłym przeszukiwaniem

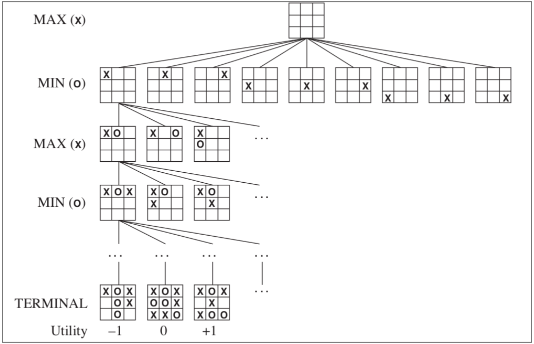
1. Mamy graczy: stan gry wskazuje na gracza, który ma się ruszać.
2. Stany końcowe mają wartości, różne dla różnych graczy.
3. Koszt jest zwykle jednostkowy (inny można uwzględnić w końcowej wypłacie, dodając do stanu „finanse” gracza)

Drzewo gry



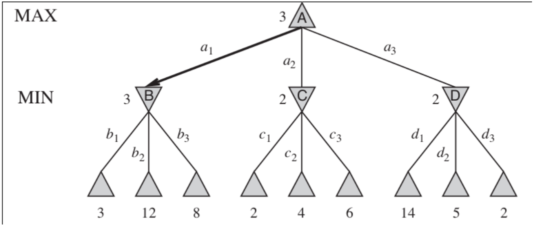
Kółko i krzyżyk. Drzewo gry

Fragment drzewa dla prawdziwej gry



Inna prosta gra (2)

- Mamy dwóch graczy Max i Min (jeden chce maksymalizacji, drugi minimalizacji).
- Wartość dla Max-a to liczba przeciwna wartości dla Min-a.
- Mamy dwa ruchy, zaczyna gracz maksymalizujący.



Algorytm MiniMax

```python
MAX = 1
MIN = 0

def decision(state):
    """decision for MAX"""
    return max(a for actions(state),
       key = lambda a : minmax(result(a,state), MIN))

def minmax(state, player):
    if terminal(state): return utility(state)

    values = [minmax(result(a,state), 1-player) for a in actions(state)]
    if player == MIN:
        return min(values)
    else:
        return max(values)
```

Drobna uwaga nazewnicza

Spotyka się różne warianty nazewnicze (niestety również na naszych slajdach):

- Algorytm MiniMax
- Algorytm min-max
- Algorytm MinMax

Algorytm MiniMax

- O(d) – pamięć
- $O(b^{2d})$ czas, gdzie d jest liczbą ply's (półruchów)
- Dla szachów $b \approx 35$, $d \approx 50$
- Dla go: 250, 150

Algorytm MiniMax (wersja realistyczna)

- Algorytm MiniMax działa jedynie dla bardzo małych, sztucznych gier (ewentualnie dla końcówek prawdziwych gier).
- Żeby go uczynić realistycznym, musimy:
  a) Przerwać poszukiwania na jakiejś głębokości.
  b) Umieć szacować wartość nieterminalnych sytuacji na planszy.

Algorytm MinMax z głębokością

```
def decision(state):
    return max(a for actions(state),
       key = lambda a : minmax(result(a,state), MIN ,0))

def minmax(state, player, depth):
    if terminal(state): return utility(state)
    if cut_off_test(state, depth):
        return heuristic_value(state)

    values = [minmax(result(a,state), 1-player, depth+1) for a in actions(state)]
    if player == 0:
        return min(values)
    else:
        return max(values)
```