# Algorytmy sztucznej inteligencji

## Ilustrowany przewodnik

Rishal Hurbans

Helion

Tytuł oryginału: Grokking Artificial Intelligence Algorithms

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-7507-9



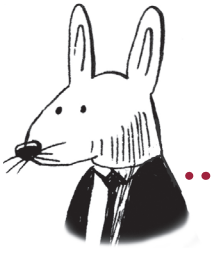
Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
*http://helion.pl/user/opinie/algszt*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

# Spis treści

# Inteligentne przeszukiwanie 3



## Zawartość rozdziału:

- Omówienie i projektowanie heurystyk na potrzeby przeszukiwania sterowanego
- Identyfikowanie problemów, które można rozwiązywać za pomocą przeszukiwania sterowanego
- Omówienie i projektowanie algorytmu przeszukiwania sterowanego
- Projektowanie algorytmu przeszukiwania do gry w grę dwuosobową

## Definiowanie heurystyk — projektowanie hipotez opartych na wiedzy

W rozdziale 2. dowiedziałeś się, jak działają algorytmy przeszukiwania siłowego. Teraz dowiesz się, jak je usprawnić, wykorzystując dodatkowe informacje na temat problemu. Posłuży do tego *przeszukiwanie sterowane* (ang. *informed search*). Polega ono na tym, że algorytm ma określony kontekst dotyczący rozwiązywanego problemu. Do reprezentowania tego kontekstu służą heurystyki, opisywane często jako *proste reguły*. *Heurystyka* jest regułą lub zestawem reguł i służy do oceny stanu. Można ją wykorzystać do definiowania kryteriów, jakie stan musi spełniać, lub do oceny danego stanu. Z heurystyki korzysta się, gdy nie można zastosować jednoznacznej metody znajdującej optymalne rozwiązanie. Heurystykę można traktować jak opartą na wiedzy hipotezę; należy posługiwać się nią bardziej jak wskazówką niż naukowo wywiedzioną prawdą dotyczącą rozwiązywanego problemu.

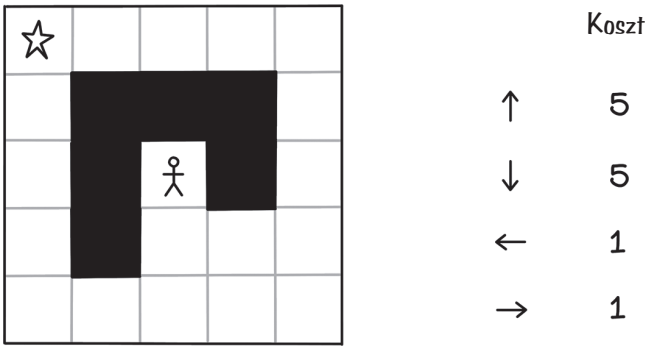
Na przykład gdy zamawiasz pizzę, heurystyczną ocenę jej jakości możesz opierać na składnikach i rodzaju ciasta. Jeśli lubisz pizzę z dodatkowym sosem, dodatkowym serem, pieczarkami i ananasem na grubym chrupiącym cieście, to większa liczba tych cech może oznaczać dla Ciebie większą atrakcyjność i wyższą ocenę heurystyczną. Pizza z mniejszą liczbą opisanych właściwości może wydać Ci się mniej smaczna i uzyskać niższą ocenę.

Innym przykładem jest pisanie algorytmów wyznaczających trasy na mapie. Heurystykę działania takich algorytmów można opisać tak: „Dobra trasa oznacza minimalny czas w korkach i minimalną odległość" lub „Dobra trasa oznacza minimalne opłaty i maksymalnie korzystne warunki jazdy". Minimalizowanie odległości w linii prostej w takim programie jest kiepską heurystyką. Może się ona sprawdzić dla ptaków lub samolotów, ale w praktyce ludzie chodzą lub jeżdżą, a te metody przemieszczania się wymagają dróg i ścieżek omijających budynki i przeszkody. W heurystykach trzeba uwzględnić kontekst.

Zastanów się nad sprawdzaniem, czy przesyłane nagranie dźwiękowe znajduje się w bibliotece materiałów chronionych prawem autorskim. Ponieważ nagrania składają się z częstotliwości, jednym ze sposobów jest porównanie wszystkich fragmentów przesyłanego nagrania z wszystkimi materiałami z biblioteki. Jednak to zadanie jest bardzo wymagające obliczeniowo. Prostym punktem wyjścia do zbudowania lepszego systemu wyszukiwania jest zdefiniowanie heurystyki, która minimalizuje różnicę w rozkładzie częstotliwości między dwoma nagraniami (rysunek 3.1). Zauważ, że częstotliwości są tu identyczne; różnica występuje tylko w czasie, a nie w rozkładzie częstotliwości. To rozwiązanie nie jest idealne, ale stanowi dobry punkt wyjścia do tworzenia mniej kosztownych obliczeniowo algorytmów.


**Rysunek 3.1.** Porównanie dwóch nagrań pod kątem rozkładu częstotliwości

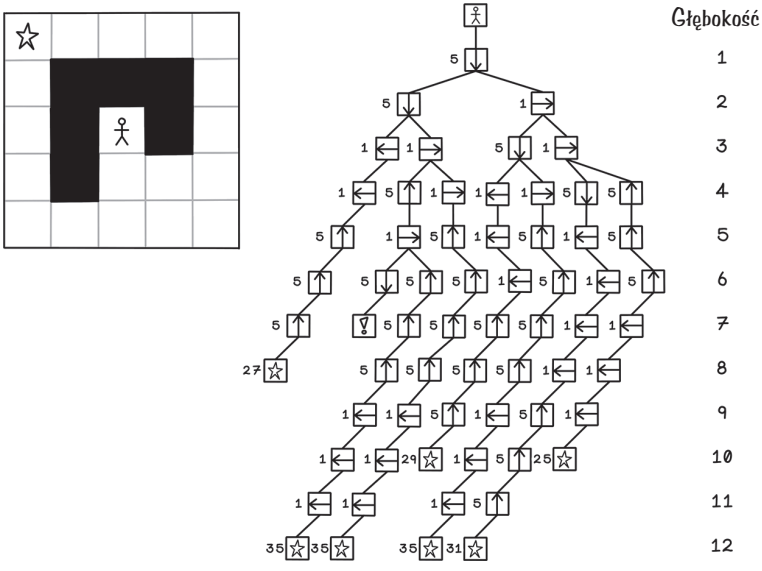
Heurystyki są zależne od kontekstu, a dobra heurystyka pozwala znacznie zoptymalizować rozwiązanie. Oto zmodyfikowana wersja problemu labiryntu z rozdziału 2., która pozwala zilustrować tworzenie heurystyk dzięki wprowadzeniu ciekawej zmiany. Zamiast traktować wszystkie ruchy w ten sam sposób i mierzyć jakość rozwiązań wyłącznie na podstawie długości ścieżki (głębokości w drzewie), można przypisać różny koszt krokom w poszczególnych kierunkach. Przyjmij, że w nowym labiryncie występują dziwne zawirowania grawitacji, dlatego kroki na północ i na południe są pięć razy bardziej kosztowne od kroków na wschód lub na zachód (rysunek 3.2).




**Rysunek 3.2.** Zmodyfikowany problem labiryntu — grawitacja

W zmodyfikowanym labiryncie czynnikami wpływającymi na jakość drogi do celu są liczba kroków i łączny koszt wszystkich kroków w ścieżce.

Na rysunku 3.3 pokazane są wszystkie ścieżki w drzewie z uwzględnieniem kosztów poszczególnych kroków. Ten przykład ilustruje przestrzeń rozwiązań w prostym labiryncie i w praktyce zwykle nie da się zastosować tego podejścia. Algorytm generuje drzewo w ramach przeszukiwania.




**Rysunek 3.3.** Wszystkie możliwe ruchy przedstawione w formie drzewa

Heurystykę w problemie labiryntu można zdefiniować tak: „Dobre ścieżki cechują się minimalnym kosztem ruchów i minimalną łączną liczbą ruchów do celu". Ta prosta heurystyka dzięki zastosowaniu wiedzy o problemie pomaga wybrać, które węzły należy odwiedzić.

**Eksperyment myślowy — jaką heurystykę zaproponujesz w następującym przykładzie?**

Kilku górników specjalizuje się w wydobyciu różnych surowców, w tym diamentów, złota i platyny. Wszyscy górnicy produktywnie pracują w każdej kopalni, ale są wydajniejsi w miejscach zgodnych z ich specjalizacją. W regionie znajduje się kilka kopalni, w których mogą występować diamenty, złoto i platyna, a w różnych odległościach między kopalniami stoją magazyny. Zaproponuj heurystykę dla problemu przydziału górników do kopalń, jeśli celem jest maksymalizacja wydajności i skrócenie czasu podróży.

**Eksperyment myślowy — możliwe rozwiązanie**

Oto sensowna heurystyka: przydziel każdego górnika do kopalni zgodnej z jego specjalizacją i nakaż mu transportowanie urobku do magazynu znajdującego się najbliżej danej kopalni. Można to zinterpretować jako minimalizowanie przydziałów górników do kopalń, które nie są zgodne z ich specjalizacją, i minimalizowanie odległości do magazynów.

# Przeszukiwanie sterowane — szukanie rozwiązań z wykorzystaniem wskazówek

*Przeszukiwanie sterowane* (nazywane też *heurystycznym*) to algorytm wykorzystujący przeszukiwanie wszerz i w głąb w połączeniu z „inteligencją". Przeszukiwanie jest sterowane heurystyką bazującą na wiedzy o problemie.

W zależności od natury problemu można stosować różne algorytmy przeszukiwania, w tym przeszukiwanie zachłanne. Jednak najpopularniejszym i najprzydatniejszym algorytmem przeszukiwania sterowanego jest A*.

## Algorytm A*

*Algorytm A** zwykle poprawia wydajność dzięki heurystyce minimalizowania kosztu następnego odwiedzanego węzła.

Łączny koszt jest obliczany na podstawie dwóch cech: łącznej odległości od węzła początkowego do danego i szacowanego kosztu przejścia do danego węzła za pomocą heurystyki. Gdy chcesz zminimalizować koszt, niższa wartość oznacza wydajniejsze rozwiązanie (rysunek 3.4).

$$f(n) = g(n) + h(n)$$

**$g(n)$: koszt ścieżki z węzła początkowego do węzła $n$**

**$h(n)$: koszt obliczony przez funkcję heurystyczną dla węzła $n$**

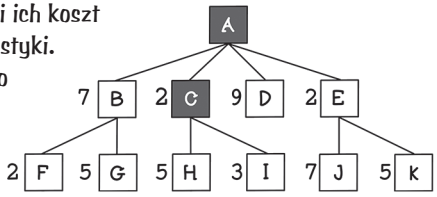
**$f(n)$: koszt ścieżki z węzła początkowego do węzła $n$ plus koszt obliczony przez funkcję heurystyczną dla węzła $n$**

**Rysunek 3.4.** Funkcja reprezentująca algorytm A*

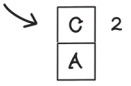
Przyjrzyj się teraz abstrakcyjnemu przykładowi odwiedzania węzłów drzewa, gdy przeszukiwanie odbywa się zgodnie z heurystyką. Istotne są tu obliczenia heurystyczne dla poszczególnych węzłów drzewa.

W przeszukiwaniu wszerz odwiedzane są wszystkie węzły z danego poziomu, a potem algorytm przechodzi do następnego poziomu. W przeszukiwaniu w głąb algorytm odwiedza wszystkie węzły do końcowej głębokości, a następnie cofa się do korzenia i sprawdza następną ścieżkę. Algorytm A* działa inaczej, ponieważ nie korzysta ze wzorca ustalonego z góry. Węzły są odwiedzane w kolejności wynikającej z kosztów zależnych od heurystyki. Zauważ, że ten algorytm początkowo nie zna kosztów wszystkich węzłów. Koszty są obliczane w trakcie eksplorowania i generowania drzewa, a każdy odwiedzony węzeł jest umieszczany na stosie. Powoduje to, że węzły o koszcie wyższym niż koszt już odwiedzonych węzłów są ignorowane, co pozwala uniknąć części obliczeń (rysunki 3.5, 3.6 i 3.7).

Dane jest drzewo z węzłami i ich koszt ustalony na podstawie heurystyki. A* odwiedza pierwsze dziecko o najniższym koszcie. Tu jest to C (koszt 2)

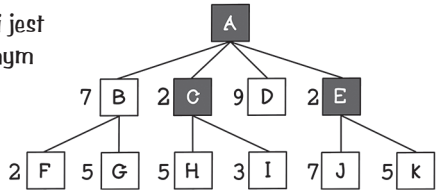


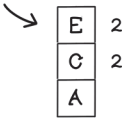

Kolejność przetwarzania elementów stosu



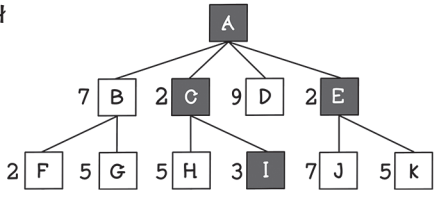

Gdy koszt dla dwóch węzłów jest taki sam, wybierany jest pierwszy z nich

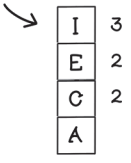
Ponieważ E też ma koszt 2 i jest dzieckiem A, będzie następnym odwiedzanym węzłem



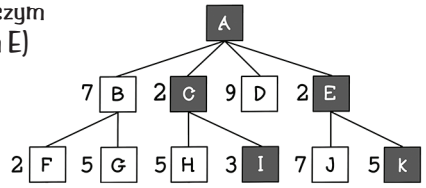

Następnie A* odwiedza węzeł o najniższym koszcie spośród dzieci A i dzieci już odwiedzonych węzłów



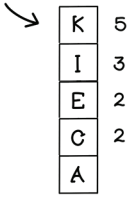




**Rysunek 3.5.** Sekwencja przetwarzania drzewa za pomocą algorytmu A* (część 1.)

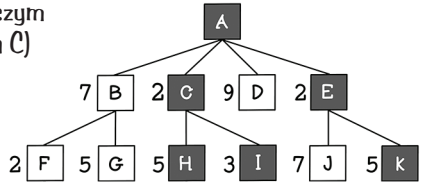
Następnym węzłem o najniższym koszcie jest K (dziecko węzła E)

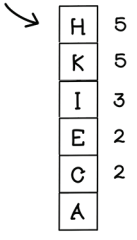



Kolejność przetwarzania elementów stosu

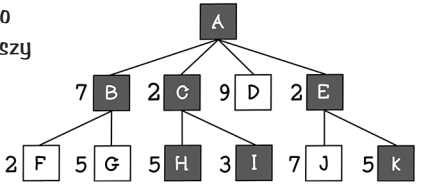


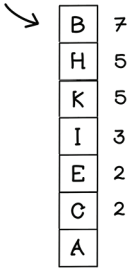

Następnym węzłem o najniższym koszcie jest H (dziecko węzła C)



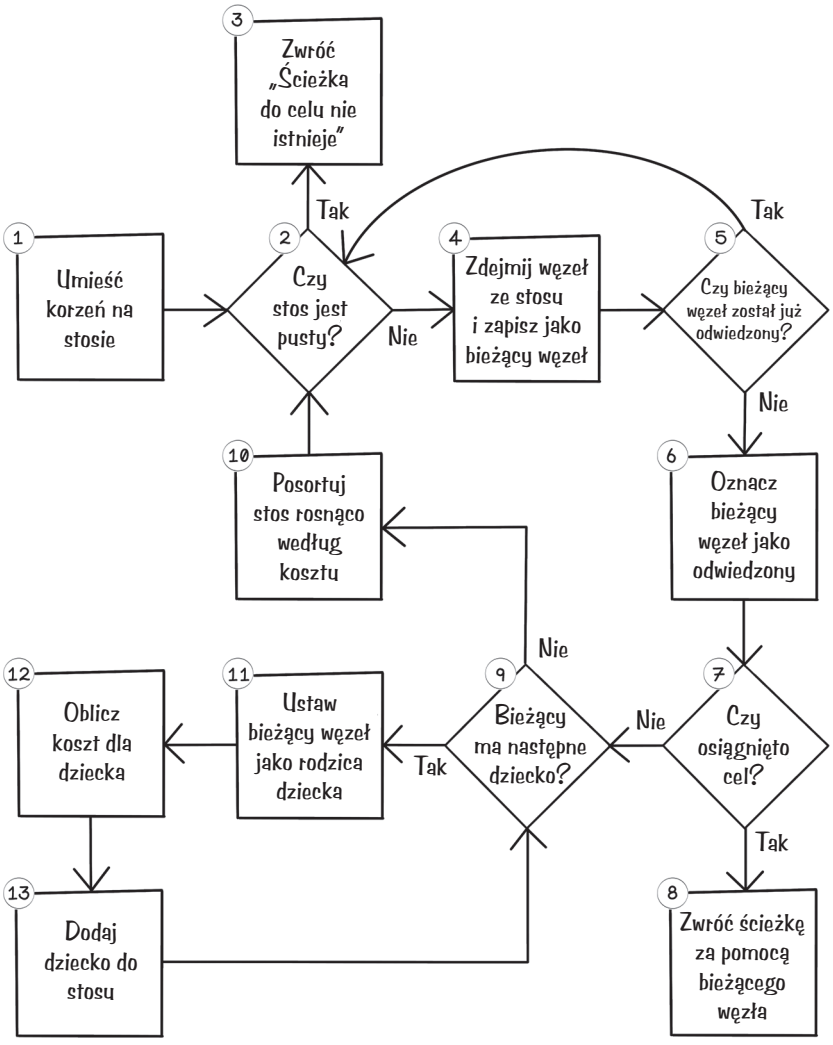

Teraz odwiedzane jest dziecko węzła A, ponieważ ma najniższy koszt spośród dzieci A i dzieci pozostałych odwiedzonych węzłów




Węzły o koszcie wyższym niż aktualna ścieżka o najniższym koszcie można zignorować, ponieważ prowadzące do rozwiązania ścieżki obejmujące te węzły będą bardziej kosztowne

**Rysunek 3.6.** Sekwencja przetwarzania drzewa za pomocą algorytmu A* (część 2.)



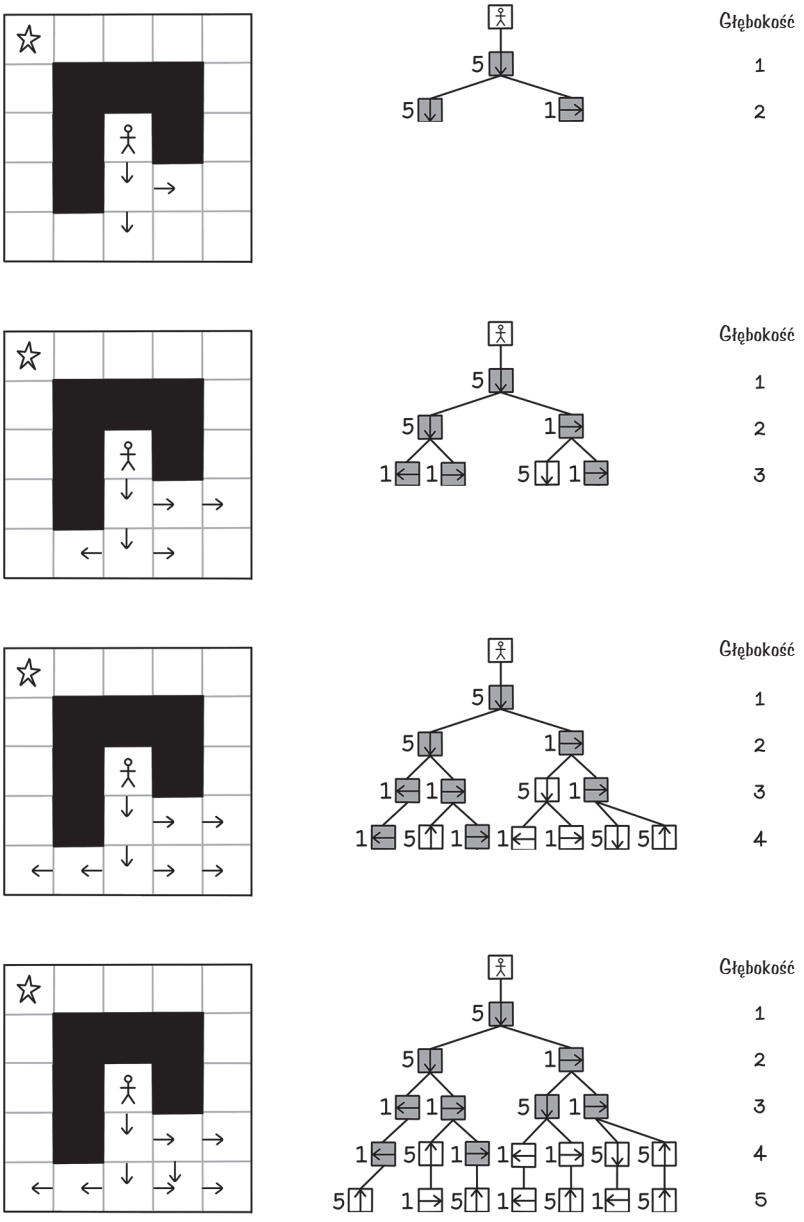

**Rysunek 3.7.** Schemat blokowy dla algorytmu A*

Przyjrzyj się przepływowi działania algorytmu A*:

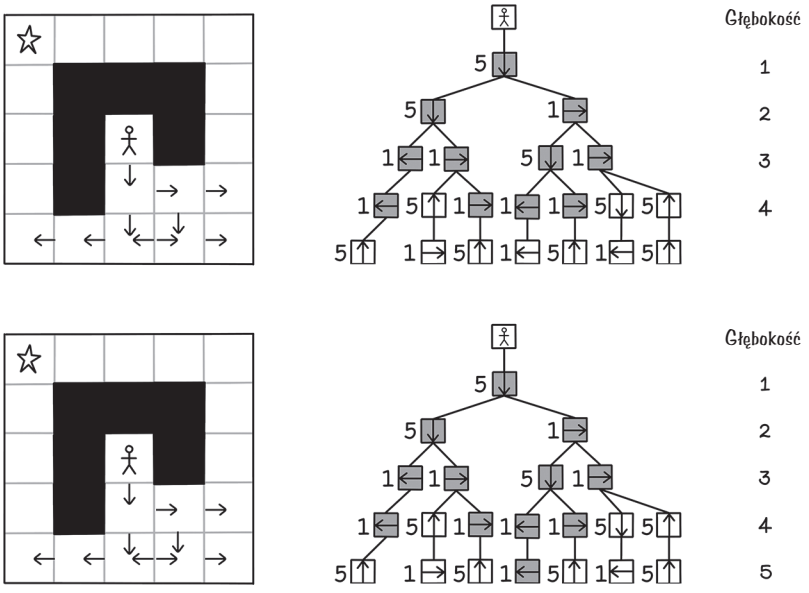
1. *Umieść korzeń na stosie*. Algorytm A* można zaimplementować za pomocą stosu, gdzie jako pierwszy przetwarzany jest ostatni dodany obiekt (model LIFO). Pierwszy krok polega na umieszczeniu korzenia na stosie.
2. *Czy stos jest pusty?* Jeśli stos jest pusty i w kroku 8. algorytm nie zwrócił ścieżki, ścieżka do celu nie istnieje. Jeżeli w kolejce występują dalsze węzły, algorytm może kontynuować wyszukiwanie.
3. *Zwróć „Ścieżka do celu nie istnieje"*. Ten krok jest jednym z możliwych zakończeń pracy algorytmu, wybieranym, gdy nie istnieje ścieżka do celu.

4. *Zdejmij węzeł ze stosu i zapisz jako bieżący węzeł.* Pobranie następnego obiektu ze stosu i zapisanie go jako bieżącego węzła pozwala zbadać związane z nim możliwości.
5. *Czy bieżący węzeł został odwiedzony?* Jeśli bieżący węzeł nie został jeszcze odwiedzony, algorytm nie przeanalizował go i może to zrobić teraz.
6. *Oznacz bieżący węzeł jako odwiedzony*. Ten krok powoduje oznaczenie, że węzeł został odwiedzony. Chroni to przed niepotrzebnym ponownym przetwarzaniem węzła.
7. *Czy osiągnięto cel?* Ten krok określa, czy bieżące dziecko zawiera szukany cel.
8. *Zwróć ścieżkę za pomocą bieżącego węzła*. Pobierając rodzica bieżącego węzła, a następnie rodzica tego rodzica itd., można opisać ścieżkę od celu do korzenia. Korzeniem jest węzeł niemający rodzica.
9. *Bieżący węzeł ma następne dziecko?* Jeśli bieżący węzeł umożliwia wykonanie innych ruchów w labiryncie, można dodać następny ruch do sprawdzenia. W przeciwnym razie algorytm przechodzi do kroku 2. i przetwarzania kolejnego obiektu ze stosu (jeżeli stos nie jest pusty). Stos działa w modelu LIFO, dlatego algorytm może przetworzyć wszystkie węzły do poziomu liści, a następnie cofnąć się i odwiedzić inne dzieci korzenia.
10. *Posortuj stos rosnąco według kosztu.* Gdy stos jest posortowany rosnąco według kosztu poszczególnych węzłów, jako następny przetwarzany jest węzeł o najniższym koszcie. Dzięki temu węzeł o najniższym koszcie zawsze zostaje odwiedzony.
11. *Ustaw bieżący węzeł jako rodzica bieżącego dziecka.* Ustaw węzeł źródłowy jako rodzica bieżącego dziecka. Ten krok jest ważny, ponieważ umożliwia prześledzenie ścieżki z bieżącego dziecka do korzenia. Na mapie źródłowym węzłem jest pozycja, z której gracz wykonuje ruch, a bieżącym dzieckiem jest pole wybrane przez gracza.
12. *Oblicz koszt dla dziecka.* W algorytmie A* funkcja kosztu jest niezwykle ważna. Tu koszt to suma odległości od korzenia i heurystycznej oceny następnego ruchu. Bardziej inteligentna heurystyka bezpośrednio wpływa na wzrost wydajności algorytmu A*.
13. *Umieść dziecko na stosie*. Dziecko jest umieszczane na stosie, aby później można było przetworzyć jego dzieci. Użycie stosu sprawia, że można przetworzyć węzły do poziomu liści przed przetworzeniem węzłów z niższych głębokości.

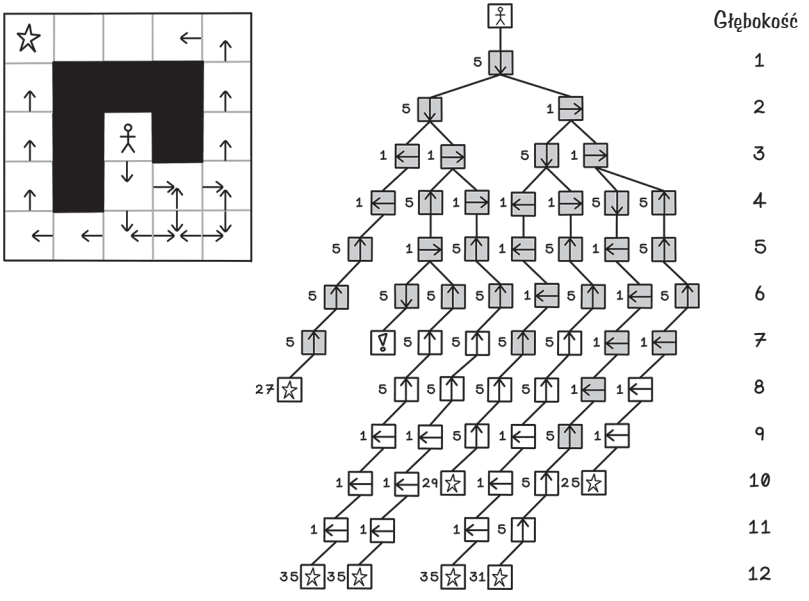
Podobnie jak w przeszukiwaniu w głąb kolejność dzieci wpływa tu na wybierane ścieżki, choć w mniejszym stopniu. Jeśli koszt dwóch węzłów jest taki sam, najpierw odwiedzany jest pierwszy węzeł, a następnie drugi (rysunki 3.8, 3.9 i 3.10).




**Rysunek 3.8.** Sekwencja przetwarzania drzewa za pomocą algorytmu A* (część 1.)




**Rysunek 3.9.** Sekwencja przetwarzania drzewa za pomocą algorytmu A* (część 2.)




**Rysunek 3.10.** Całe drzewo z węzłami odwiedzonymi za pomocą algorytmu A*

Zauważ, że istnieje kilka ścieżek do celu, ale algorytm A* znajduje ścieżkę o minimalnym koszcie — z mniejszą liczbą ruchów i o niższym ich koszcie (przy założeniu, że ruchy na północ i na południe są bardziej kosztowne).

## Pseudokod

W algorytmie A* używane jest podejście podobne jak w algorytmie przeszukiwania w głąb, ale celowo wybierane są węzły o niższym koszcie. Do przetwarzania węzłów używany jest stos, który jednak jest sortowany rosnąco przy każdych nowych obliczeniach. To sortowanie gwarantuje, że ze stosu zawsze zdejmowany jest obiekt o najniższym koszcie (ponieważ zajmuje pierwszą pozycję po posortowaniu).

```
wykonaj_a_gwiazdka(labirynt,korzeń,odwiedzone_punkty):
  niech s będzie nowym stosem
  umieść korzeń na s
  dopóki s nie jest pusty
    zdejmij element z s i przypisz go do aktualny_punkt
    jeśli aktualny_punkt nie został jeszcze odwiedzony:
      oznacz aktualny_punkt jako odwiedzony
      jeśli wartością w aktualny_punkt jest cel:
        zwróć ścieżkę, używając aktualny_punkt
      w przeciwnym razie:
        dodaj dostępne pola północne, wschodnie, południowe i
zachodnie do listy dzieci
        dla każdego elementu dziecko na liście dzieci:
          ustaw rodzica dziecko jako aktualny_punkt
          ustaw koszt dla dziecko na oblicz_koszt(aktualny_
punkt,dziecko)
          umieść dziecko na stosie s
        posortuj s rosnąco według kosztu
  zwróć "Ścieżka do celu nie istnieje"
```

Wydajność algorytmu A* zależy od funkcji do obliczania kosztu. Funkcja kosztu zapewnia algorytmowi informacje pozwalające znaleźć ścieżkę o najniższym koszcie. W zmodyfikowanym problemie labiryntu wyższy koszt jest powiązany z ruchami w górę i w dół. Jeśli użyjesz nieodpowiedniej funkcji kosztu, algorytm nie będzie działał poprawnie.

Dwie następne funkcje ilustrują obliczanie kosztu. Odległość od korzenia jest dodawana do kosztu następnego ruchu. W omawianym przykładzie koszt ruchów w kierunkach północnym i południowym wpływa na łączny koszt odwiedzin danego węzła.

```
oblicz_koszt(źródłowy,docelowy):
  niech odległość_do_korzenia będzie równa długości ścieżki z
źródłowy do docelowy
```

```
  niech koszt_ruchu będzie równy pobierz_koszt_
ruchu(źródłowy,docelowy)
  zwróć odległość_do_korzenia + koszt_ruchu

pobierz_koszt_ruchu(źródłowy,docelowy):
  jeśli docelowy znajduje się na północ lub na południe względem
źródłowy:
    zwróć 5
  w przeciwnym razie:
    zwróć 1
```

Algorytmy przeszukiwania siłowego, na przykład przeszukiwanie wszerz i przeszukiwanie w głąb, sprawdzają wszystkie możliwości i zwracają optymalne rozwiązanie. Algorytm A* jest dobrą techniką, jeśli można opracować sensowną heurystykę wspomagającą przeszukiwanie. Działa on wydajniej niż algorytmy przeszukiwania siłowego, ponieważ ignoruje węzły o koszcie wyższym niż koszt już odwiedzonych węzłów. Jeżeli jednak heurystyka jest niewłaściwie dobrana (nie ma sensu dla danego problemu i w danym kontekście), algorytm będzie znajdował nieoptymalne rozwiązania.

## Sytuacje, w których warto stosować algorytmy przeszukiwania sterowanego

Algorytmy przeszukiwania sterowanego są uniwersalną i przydatną techniką w różnych praktycznych zastosowaniach, w których można zdefiniować heurystyki. Oto przykłady:

- *Znajdowanie ścieżki dla autonomicznych postaci w grach komputerowych*. Programiści gier często posługują się tym algorytmem do sterowania ruchem jednostek wrogich graczowi w grach, w których celem jest znalezienie gracza.
- *Przetwarzanie akapitów w przetwarzaniu języka naturalnego*. W trakcie analizy tekstu akapit można rozbić na zdania, a te na słowa różnego rodzaju (na przykład rzeczowniki i czasowniki), aby utworzyć przetwarzane drzewo. Przeszukiwanie sterowane może być przydatne także do analizy znaczenia tekstu.
- *Trasowanie w sieciach telekomunikacyjnych*. Algorytmy przeszukiwania sterowanego można wykorzystać do znajdowania najkrótszych ścieżek w sieciach telekomunikacyjnych, aby poprawić ich wydajność. Serwery, węzły sieciowe i połączenia można przedstawić w formie grafów obejmujących węzły i krawędzie.
- *Gry jednoosobowe i łamigłówki*. Algorytmy przeszukiwania sterowanego można zastosować do znajdowania rozwiązań w grach jednoosobowych i łamigłówkach takich jak kostka Rubika, ponieważ każdy ruch oznacza decyzję w drzewie możliwości, a zadanie polega na dotarciu do stanu docelowego.

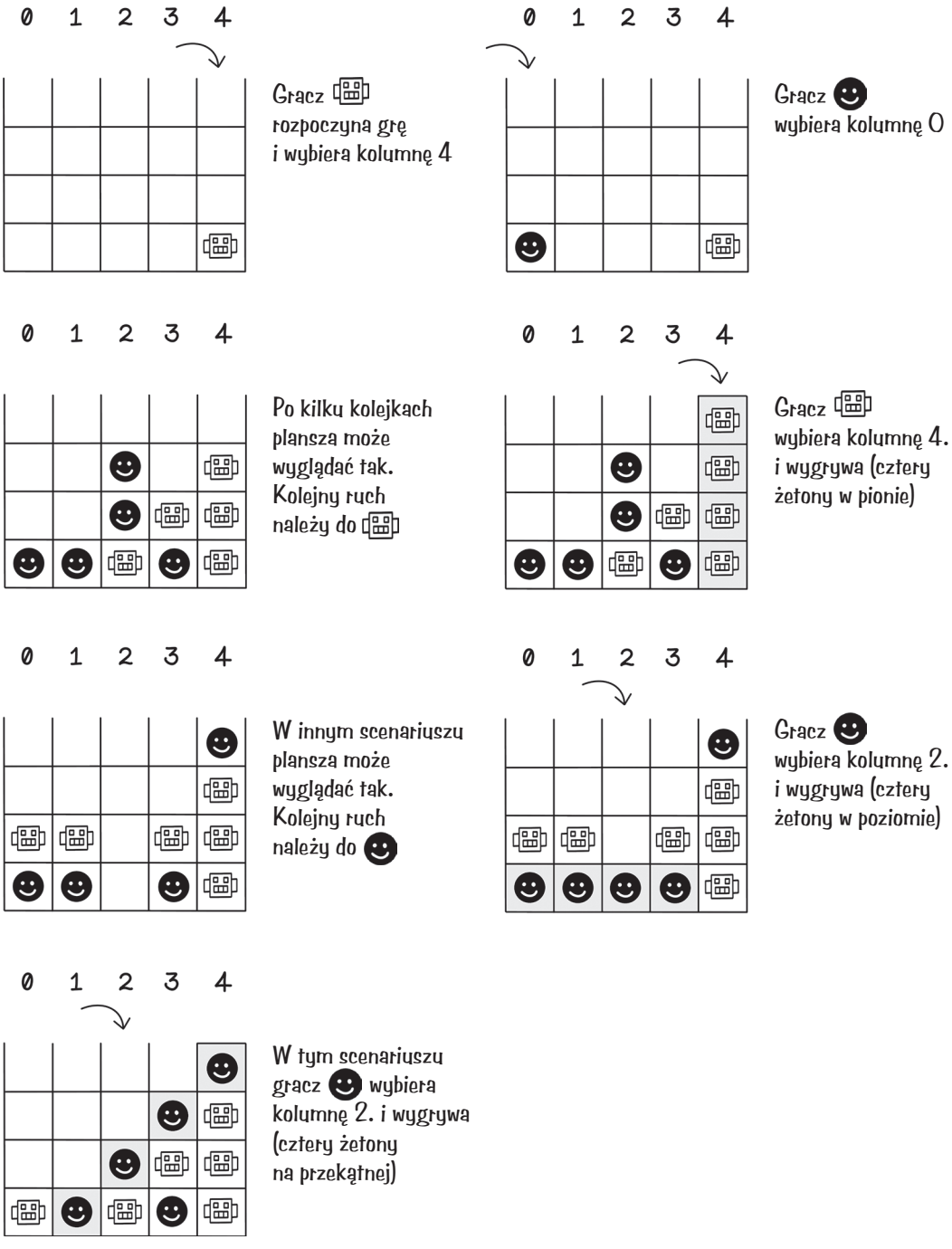
# Przeszukiwanie antagonistyczne — szukanie rozwiązań w zmiennym środowisku

W problemie labiryntu występuje jeden agent — gracz. To środowisko zależy tylko od jednego gracza, dlatego to on generuje wszystkie możliwości. Do tej pory celem było zmaksymalizowanie korzyści z perspektywy gracza — znalezienie ścieżki do celu o najmniejszej odległości i najniższym koszcie.

*Przeszukiwanie antagonistyczne* cechuje się obecnością opozycji lub konfliktów. Takie problemy wymagają przewidywania działań przeciwnika, zrozumienia ich i przeciwdziałania im, aby zrealizować cele. Przykładowe problemy antagonistyczne to gry dwuosobowe z naprzemiennie wykonywanymi ruchami takie jak kółko i krzyżyk lub czwórki. Gracze na zmianę wykonują ruch i mogą zmienić stan środowiska gry na swoją korzyść. Zestaw reguł określa, w jaki sposób można wprowadzać zmiany w środowisku oraz jak wyglądają stany oznaczające wygraną i koniec gry.

## Prosty problem antagonistyczny

W tym punkcie problemy antagonistyczne są omawiane na przykładzie gry czwórki (rysunek 3.11). Jest to gra, w której gracze na zmianę wrzucają żeton w wybraną kolumnę planszy. Żetony w kolumnie są umieszczane jeden na drugim, a gracz, który zdoła umieścić cztery swoje żetony obok siebie (w pionie, w poziomie lub na przekątnej), wygrywa. Jeśli plansza się zapełni bez wyłonienia zwycięzcy, gra kończy się remisem.




**Rysunek 3.11.** Gra czwórki

## Algorytm min-max — symulowanie działań i wybieranie optymalnego przyszłego stanu

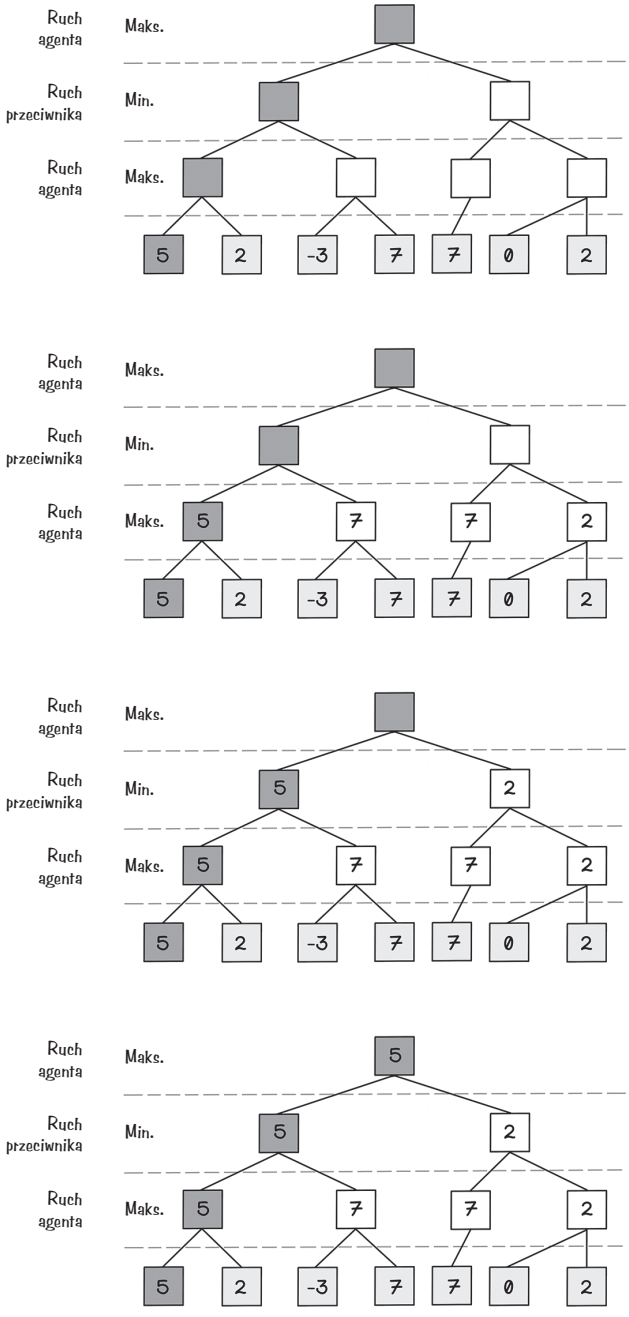
*Algorytm min-max* ma generować drzewo możliwych skutków oparte na ruchach, jakie może wykonać każdy gracz, i preferować ścieżki korzystne dla danego agenta oraz unikać ścieżek korzystnych dla przeciwnika. W tym celu algorytm symuluje możliwe ruchy i na podstawie heurystyki ocenia stan po wykonaniu poszczególnych posunięć. Algorytm min-max próbuje sprawdzić możliwie dużo przyszłych stanów. Jednak z powodu ograniczeń pamięciowych i obliczeniowych sprawdzenie całego drzewa gry może okazać się niewykonalne, dlatego przeszukiwanie jest kontynuowane do określonej głębokości. Algorytm min-max symuluje wykonywanie ruchów na zmianę przez każdego gracza, dlatego głębokość przeszukiwania jest bezpośrednio zależna od liczby kolejek. Na przykład głębokość 4 oznacza, że każdy gracz wykonuje dwa posunięcia — gracz A wykonuje ruch, gracz B odpowiada, gracz A robi następne posunięcie, gracz B wykonuje kolejny ruch.

### Heurystyki

Algorytm min-max podejmuje decyzje na podstawie heurystycznych ocen. Ocena jest wyznaczana zgodnie z opracowaną heurystyką (algorytm nie uczy się sposobu oceniania stanu). Dla danego stanu efekt każdego możliwego dozwolonego ruchu w tym stanie oznacza węzeł dziecka w drzewie gry.

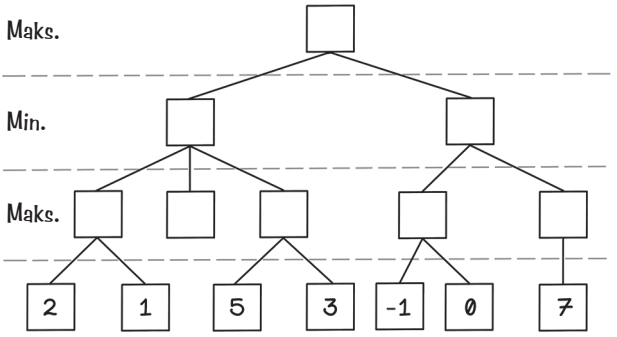
Przyjmij, że zgodnie z heurystyką dodatnie oceny oznaczają korzystniejsze stany niż oceny ujemne. Symulując wszystkie możliwe dozwolone ruchy, algorytm min-max próbuje zminimalizować prawdopodobieństwo wyboru ruchów, po których przeciwnik będzie miał przewagę, i maksymalizuje prawdopodobieństwo wykonania posunięć dających przewagę (lub wygraną) agentowi.

Na rysunku 3.12 widoczne jest drzewo min-max. Na tym rysunku liście są jedynymi węzłami, dla których obliczana jest heurystyczna ocena, ponieważ te stany oznaczają zwycięstwo lub remis. Pozostałe węzły w drzewie oznaczają stan w toczącej się grze. Jeśli zaczynasz od poziomu, na którym obliczone są heurystyczne oceny, i kierujesz się w górę, możesz wybrać dziecko o minimalnej lub maksymalnej ocenie (w zależności od tego, na kogo przypada następny ruch w przyszłych symulowanych stanach). Zaczynając od góry, agent próbuje zmaksymalizować ocenę. Po co drugiej kolejce cel się zmienia, ponieważ wartość dla agenta ma być maksymalizowana, a dla przeciwnika — minimalizowana.

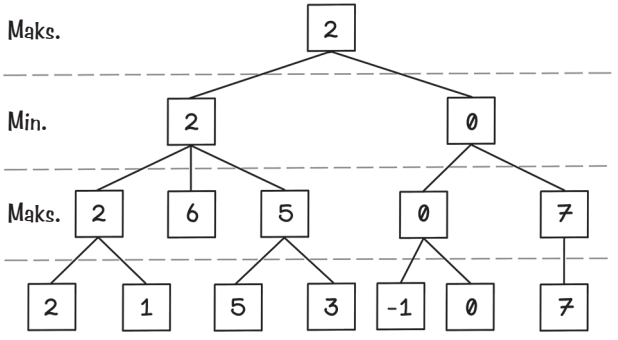



**Rysunek 3.12.** Sekwencja przetwarzania elementów drzewa za pomocą algorytmu min-max

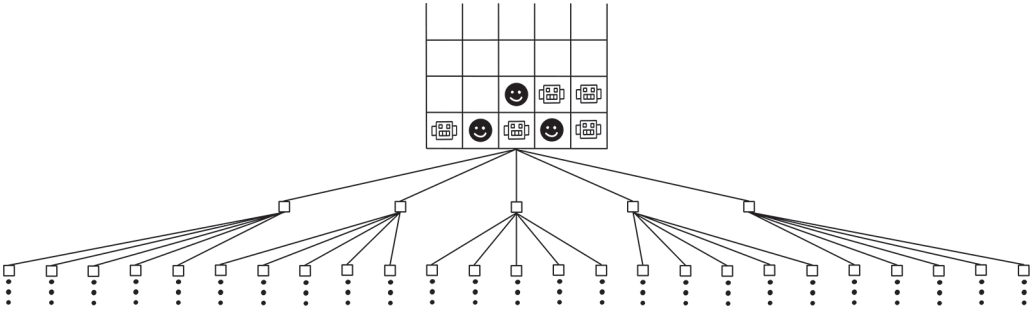
**Ćwiczenie — które wartości będą przekazywane dalej w poniższym drzewie min-max?**




**Rozwiązanie — które wartości będą przekazywane dalej w poniższym drzewie min-max?**

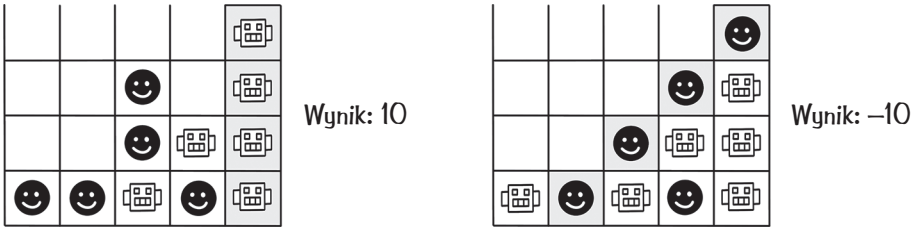



Ponieważ algorytm min-max symuluje możliwe skutki, w grach z dużą liczbą możliwości drzewo gry błyskawicznie się rozrasta, a zbadanie całego drzewa szybko staje się zbyt kosztowne obliczeniowo. Nawet w prostej grze czwórki na planszy o wymiarach 5×4 liczba możliwości sprawia, że analiza całego drzewa gry w każdej kolejce jest niewydajnym rozwiązaniem (rysunek 3.13).

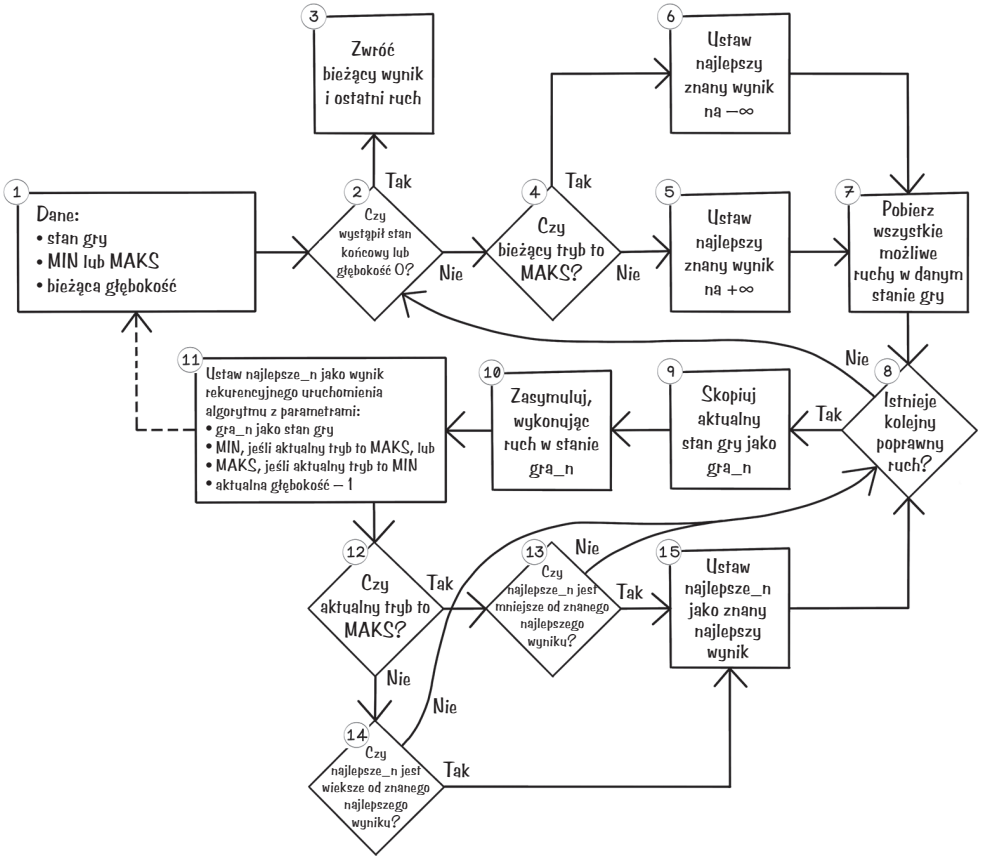


**Rysunek 3.13.** Gwałtowny wzrost liczby możliwości w trakcie przeszukiwania drzewa gry

Aby wykorzystać algorytm min-max w grze czwórki, algorytm musi wykonać wszystkie możliwe ruchy w aktualnym stanie gry, następnie ocenić wszystkie możliwe posunięcia w każdym uzyskanym stanie i powtarzać te kroki do czasu znalezienia optymalnej ścieżki. Stany gry prowadzące do wygranej agenta mają tu wartość 10, a stany skutkujące zwycięstwem przeciwnika mają wartość –10. Algorytm min-max stara się zmaksymalizować wynik na korzyść agenta (rysunki 3.14 i 3.15).




**Rysunek 3.14.** Ocena stanów korzystnych dla agenta i korzystnych dla przeciwnika




**Rysunek 3.15.** Schemat blokowy obrazujący działanie algorytmu min-max

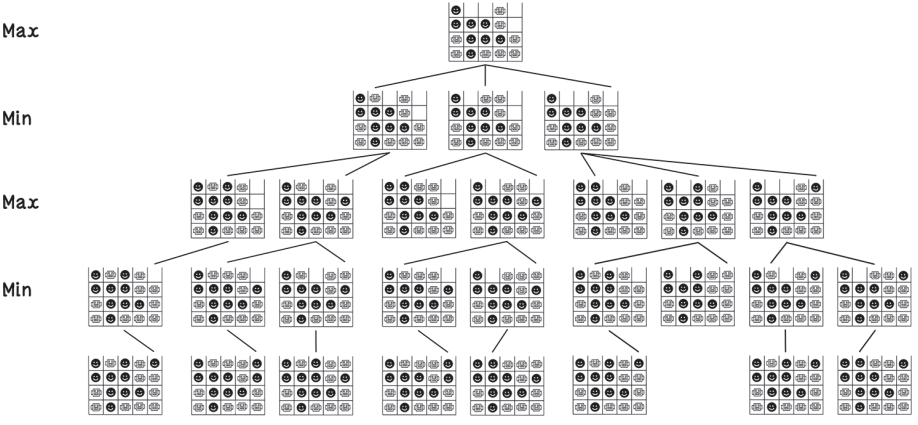
Choć z powodu swojej wielkości schemat blokowy algorytmu min-max wygląda na skomplikowany, sam algorytm jest dość prosty. Wielkość schematu jest spowodowana dużą liczbą warunków sprawdzających, czy w bieżącym stanie należy maksymalizować, czy minimalizować wynik.

Prześledź teraz działanie algorytmu min-max:

1. *Algorytm może zacząć pracę, gdy dane są: stan gry, tryb (minimalizacji lub maksymalizacji) i bieżąca głębokość*. Trzeba zrozumieć dane wejściowe tego algorytmu, ponieważ działa on w sposób rekurencyjny. Taki algorytm wywołuje samego siebie w jednym lub kilku krokach. W algorytmie rekurencyjnym ważny jest warunek zakończenia pracy, zapobiegający wywoływaniu algorytmu w nieskończoność.
2. *Czy wystąpił stan końcowy lub głębokość 0?* Ten warunek pozwala sprawdzić, czy bieżący stan gry jest stanem końcowym lub czy osiągnięto oczekiwaną głębokość. Stan końcowy oznacza, że jeden z graczy wygrał lub gra zakończyła się remisem. Wartość 10 reprezentuje wygraną agenta, wartość –10 to wygrana przeciwnika, a 0 oznacza remis. Głębokość jest podawana, ponieważ sprawdzenie całego drzewa możliwości i dojście do wszystkich stanów końcowych jest obliczeniowo kosztowne i na standardowych komputerach zapewne będzie trwać zbyt długo. Dzięki wyznaczeniu głębokości algorytm może sprawdzić kilka następnych kolejek, aby stwierdzić, czy prowadzą one do stanów końcowych.
3. *Zwróć bieżący wynik i ostatni ruch.* Jeśli bieżący stan jest stanem końcowym lub algorytm dotarł do podanej głębokości, należy zwrócić wynik dla bieżącego stanu.
4. *Czy bieżący tryb to MAKS?* Jeśli w danej iteracji algorytm ma maksymalizować wartość, będzie próbował zmaksymalizować wynik na korzyść agenta.
5. *Ustaw najlepszy znany wynik na +∞.* Jeżeli w bieżącym trybie algorytm ma minimalizować wartość, najlepszy wynik jest ustawiany na dodatnią nieskończoność, ponieważ wiadomo, że wartości zwracane dla stanów gry zawsze będą mniejsze. W rzeczywistej implementacji zamiast nieskończoności używana jest bardzo duża liczba.
6. *Ustaw najlepszy znany wynik na –∞.* Jeżeli w bieżącym trybie algorytm ma maksymalizować wartość, najlepszy wynik jest ustawiany na ujemną nieskończoność, ponieważ wiadomo, że wartości zwracane dla stanów gry zawsze będą większe. W rzeczywistej implementacji zamiast nieskończoności używana jest bardzo duża liczba ujemna.
7. *Pobierz wszystkie możliwe ruchy dla danego stanu gry*. W tym kroku tworzona jest lista ruchów dozwolonych w danym stanie gry. Wraz z postępem gry niektóre dostępne początkowo ruchy stają się niewykonalne. W grze czwórki kolumna może zostać zapełniona, dlatego nie można jej wybrać.
8. *Czy istnieje następny dozwolony ruch?* Jeśli nie ma już więcej dozwolonych ruchów do wykonania, algorytm skraca przetwarzanie i zwraca najlepsze posunięcie w danym wywołaniu funkcji.

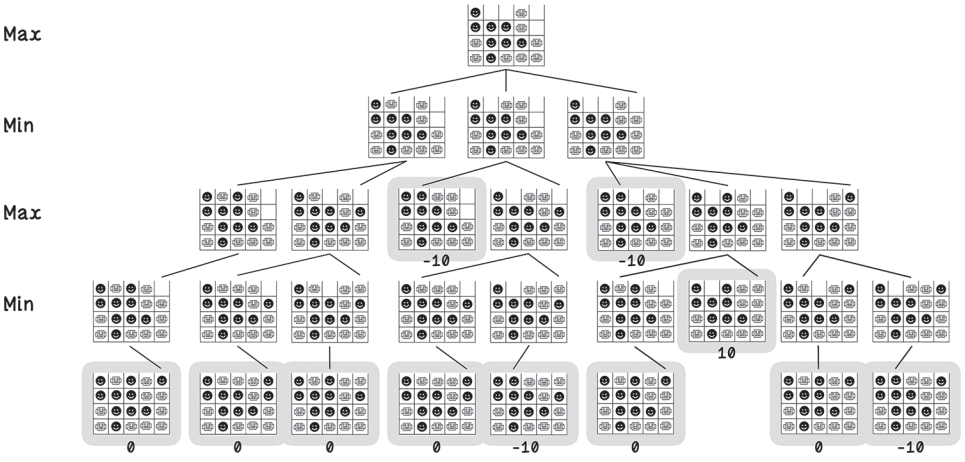
9. *Skopiuj bieżący stan gry jako gra_n.* Kopia bieżącego stanu gry jest potrzebna do przeprowadzania symulacji możliwych przyszłych ruchów.
10. *Zasymuluj ruch, wykonując go dla stanu gra_n.* Ten krok powoduje wykonanie analizowanego ruchu w skopiowanym stanie gry.
11. *Przypisz do najlepszy_n wynik rekurencyjnego uruchomienia algorytmu.* Tu pojawia się rekurencja. Zmienna *najlepszy_n* służy do przechowywania najlepszego następnego ruchu. Algorytm ma przeanalizować dalsze możliwości, zaczynając od tego posunięcia.
12. *Czy bieżący tryb to MAKS?* Gdy rekurencyjne wywołanie zwróci najlepszego kandydata, ten warunek pozwoli określić, czy w bieżącym trybie należy maksymalizować wartość.
13. *Czy najlepszy_n jest mniejsze od znanej najlepszej wartości?* Ten krok pozwala ustalić, czy algorytm znalazł wynik wyższy od poprzedniego, gdy tryb wymaga maksymalizacji wartości.
14. *Czy najlepszy_n jest większe od znanej najlepszej wartości?* Ten krok pozwala stwierdzić, czy algorytm znalazł wynik wyższy od poprzedniego, gdy tryb wymaga minimalizacji wartości.
15. *Zapisz najlepszy_n jako znaną najlepszą wartość.* Jeśli algorytm znalazł nowy najlepszy wynik, należy zapisać go jako znaną najlepszą wartość.

Dany jest określony stan gry czwórki. Algorytm min-max generuje w nim drzewo pokazane na rysunku 3.16. W stanie początkowym sprawdzany jest każdy możliwy ruch. Następnie analizowane są wszystkie ruchy w otrzymanych stanach do czasu uzyskania stanu końcowego — zapełnienia planszy lub wygranej jednego z graczy.



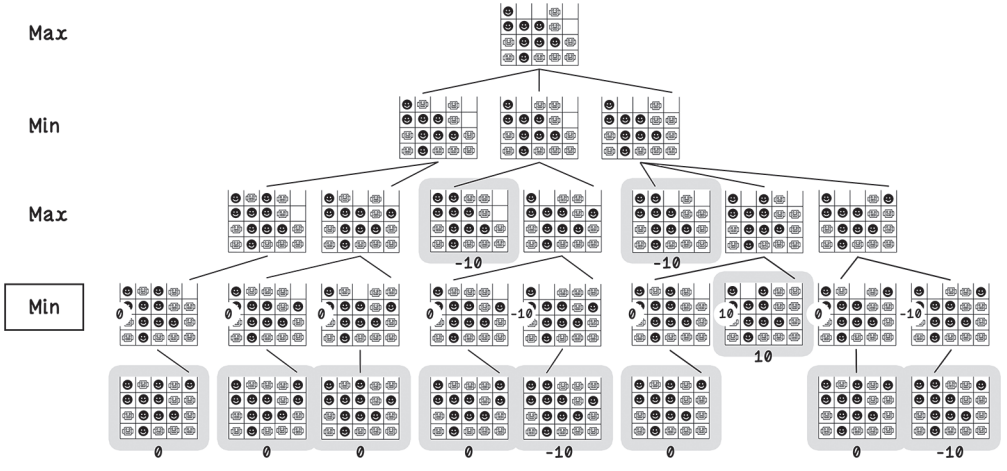

**Rysunek 3.16.** Reprezentacja możliwych stanów w grze czwórki

Węzły wyróżnione na rysunku 3.17 są węzłami ze stanem końcowym, w których remis ma wartość 0, przegrana ma wartość –10, a wygrana to wartość 10. Ponieważ algorytm ma maksymalizować wynik, oczekiwana jest wartość dodatnia, natomiast wygranym przeciwnika przypisana jest wartość ujemna.



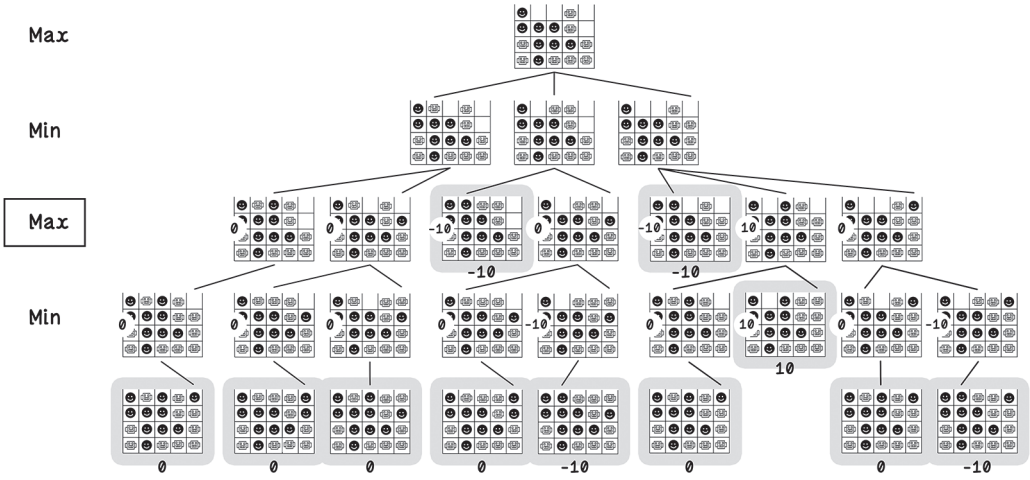

**Rysunek 3.17.** Możliwe stany końcowe w grze czwórki

Gdy wartości są znane, algorytm min-max zaczyna od najniższego poziomu i wybiera węzeł, dla którego wynikiem jest wartość minimalna (rysunek 3.18).



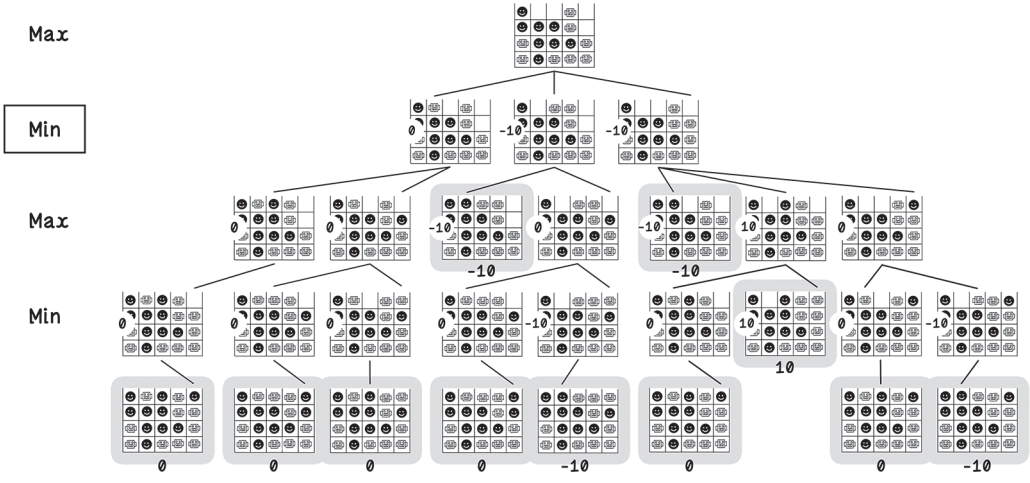

**Rysunek 3.18.** Możliwe wartości stanów końcowych w grze czwórki (część 1.)

Potem, na kolejnym poziomie, algorytm wybiera węzeł z wartością maksymalną (rysunek 3.19).



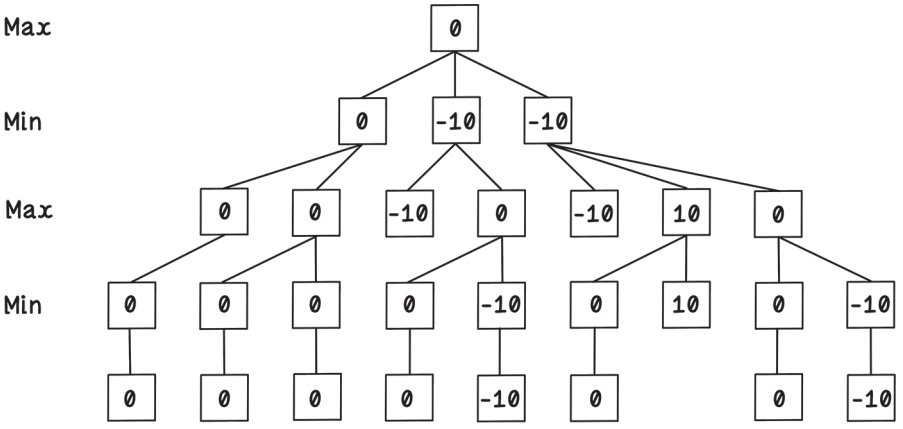

**Rysunek 3.19.** Możliwe wartości stanów końcowych w grze czwórki (część 2.)

Wreszcie na następnym poziomie algorytm wybiera węzły z minimalnymi wartościami, a w korzeniu wybierany jest węzeł z wartością maksymalną. Jeśli prześledzisz wybierane węzły i ich wartości oraz spróbujesz intuicyjnie zrozumieć problem, zobaczysz, że algorytm wybiera remisową ścieżkę, aby uniknąć przegranej. Jeżeli algorytm wybierze ścieżkę prowadzącą do wygranej, z wysokim prawdopodobieństwem przegra w następnej kolejce. Algorytm zawsze zakłada, że przeciwnik wykona najlepszy ruch, maksymalizujący prawdopodobieństwo jego wygranej (rysunek 3.20).




**Rysunek 3.20.** Możliwe wartości stanów końcowych w grze czwórki (część 3.)

Uproszczone drzewo z rysunku 3.21 ilustruje działanie algorytmu min-max dla przykładowego stanu gry.




**Rysunek 3.21.** Uproszczone drzewo gry z wartościami z algorytmu min-max

## Pseudokod

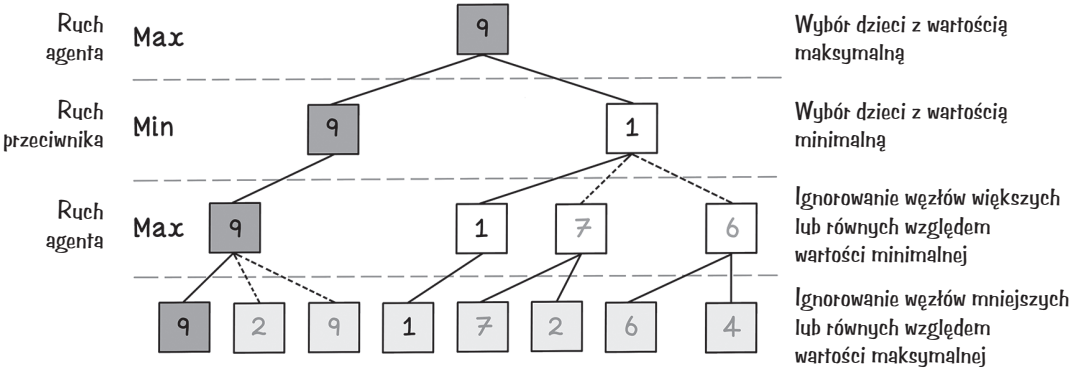
Algorytm min-max jest tu zaimplementowany jako funkcja rekurencyjna. Ta funkcja przyjmuje bieżący stan, oczekiwaną głębokość przeszukiwania, tryb (minimalizacji lub maksymalizacji) i ostatni ruch. Algorytm kończy pracę, zwracając najlepszy ruch i wartość dla każdego dziecka z każdego poziomu drzewa. Gdy porównasz kod ze schematem blokowym z rysunku 3.15, zobaczysz, że żmudne warunki sprawdzające tryb nie są tu tak widoczne. W pseudokodzie wartości 1 i –1 reprezentują tryby maksymalizacji i minimalizacji. Dzięki zastosowaniu wyrafinowanej logiki najlepsze wartości, warunki i zmiany stanów można obsługiwać za pomocą mnożenia przez wartość ujemną (wartość ujemna pomnożona przez inną wartość ujemną daje wartość dodatnią). Jeśli więc –1 oznacza ruch przeciwnika, pomnożenie jej przez –1 daje 1, co oznacza posunięcie agenta. Potem, w następnej kolejce, 1 jest mnożone przez –1, co daje –1 oznaczające ruch przeciwnika.

```
minmax(stan,głębokość,min_lub_maks,ostatni_ruch):
  niech aktualny_wynik równa się stan.pobierz_wynik
  jeśli aktualny_wynik jest różne od 0 lub stan.pełny lub głębokość
jest równe 0:
    zwróć nowy Ruch(ostatni_ruch,aktualny_wynik)
  niech najlepszy_wynik równa się min_lub_maks razy -∞
  niech najlepszy_ruch = -1
  dla każdego możliwego wyboru (od 0 do 4 ) jako ruch:
    niech dziecko równa się kopii stan
    wykonaj bieżący ruch dla dziecko
    niech najlepsze_dziecko równa się minmax(dziecko,głębokość-1,min_
lub_maks*-1,ruch)
    jeśli (najlepsze_dziecko.wynik jest większe niż najlepszy_wynik i
           min_lub_maks to MAKS)
    lub (najlepsze_dziecko.wynik jest mniejsze niż najlepszy_wynik i
```

```
        min_lub_maks to MIN):
    niech najlepszy_wynik = najlepsze_dziecko.wynik
    niech najlepszy_ruch = najlepsze_dziecko.ruch
zwróć nowy Ruch(najlepszy_ruch,najlepszy_wynik)
```

## Algorytm alfa-beta — optymalizacja polegająca na analizie tylko sensownych ścieżek

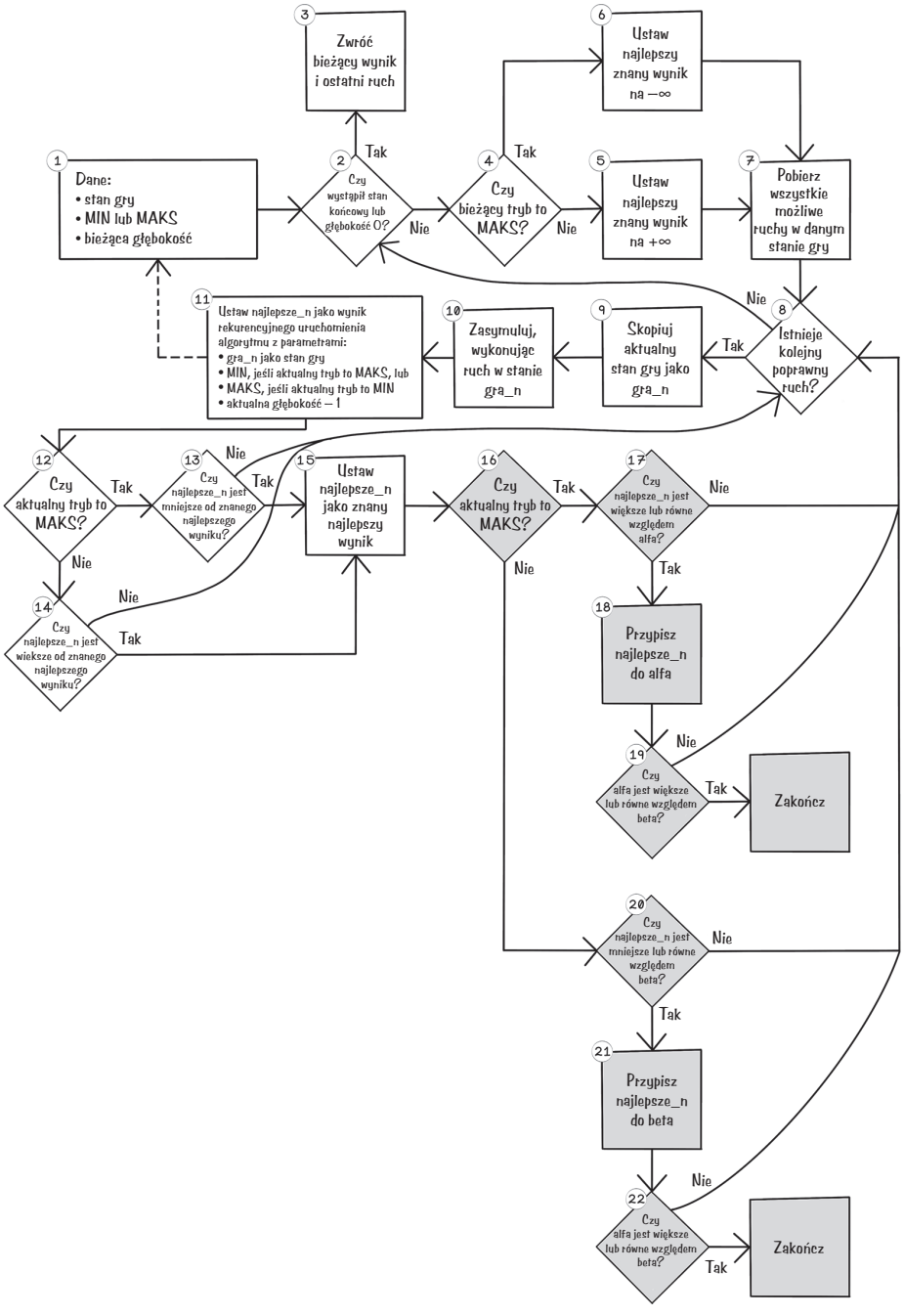
*Algorytm alfa-beta* to technika używana razem z algorytmem min-max do pomijania obszarów drzewa gry, o których wiadomo, że zawierają nieoptymalne rozwiązania. Ta technika optymalizuje algorytm min-max i zmniejsza ilość obliczeń, ponieważ nieistotne ścieżki są ignorowane. Ponieważ wiadomo, że drzewo gry czwórki gwałtownie się rozrasta, łatwo jest dostrzec, że ignorowanie ścieżek pozwala znacznie poprawić wydajność (rysunek 3.22).




**Rysunek 3.22.** Przykład działania algorytmu alfa-beta

Algorytm alfa-beta zapisuje najlepsze wartości dla trybów maksymalizacji i minimalizacji jako alfę i betę. Początkowo alfa jest równa $-\infty$, a beta to $\infty$ (są to najgorsze wartości dla obu graczy). Jeśli najlepsza wartość w trybie minimalizacji jest niższa od najlepszej wartości w trybie maksymalizacji, oczywiste jest, że inne ścieżki z już odwiedzonymi węzłami nie zmienią najlepszego wyniku.

Na rysunku 3.23 pokazane są zmiany potrzebne w schemacie blokowym algorytmu min-max, aby uwzględnić optymalizację z użyciem algorytmu alfa-beta. Wyróżnione bloki to dodatkowe kroki w algorytmie min-max.




**Rysunek 3.23.** Działanie algorytmu min-max z algorytmem alfa-beta

Oto kroki, jakie należy dodać do algorytmu min-max. Używane tu warunki pozwalają zakończyć badanie ścieżek, jeśli wiadomo, że najlepszy znaleziony wynik nie zmieni ogólnego efektu:

1. *Czy bieżący tryb to MAKS?* Należy ustalić, czy algorytm w danym kroku dąży do maksymalizacji, czy minimalizacji wartości.
2. *Czy najlepsze_n jest większe lub równe względem alfa?* Jeśli bieżący tryb to MAKS, a aktualny najlepszy wynik jest większy lub równy względem alfa, w dzieciach danego węzła nie ma lepszych wyników. Algorytm może wtedy zignorować ten węzeł.
3. *Przypisz najlepsze_n do alfa.* Należy przypisać do zmiennej *alfa* wartość *najlepsze_n*.
4. *Czy alfa jest większa lub równa względem beta?* Wynik jest równie dobry jak inne znalezione wyniki. Można pominąć dalsze analizowanie węzła, wychodząc z pętli.
5. *Czy najlepsze_n jest mniejsze lub równe względem beta?* Jeśli bieżący tryb to MIN, a aktualny najlepszy wynik jest mniejszy lub równy względem beta, w dzieciach danego węzła nie ma lepszych wyników. Algorytm może wtedy zignorować ten węzeł.
6. *Przypisz najlepsze_n do beta.* Należy przypisać do zmiennej *beta* wartość *najlepsze_n*.
7. *Czy alfa jest większa lub równa względem beta?* Wynik jest równie dobry jak inne znalezione wyniki. Można pominąć dalsze analizowanie węzła, wychodząc z pętli.

## Pseudokod

Kod algorytmu alfa-beta jest bardzo podobny do kodu algorytmu min-max. Dodatkowo trzeba tylko zapisywać wartości alfa i beta oraz zarządzać nimi w trakcie poruszania się po drzewie. Zauważ, że w trybie minimalizacji zmienna `min_lub_maks` jest równa –1, a w trybie maksymalizacji zmienna `min_lub_maks` jest równa 1.

```
minmax_alfa_beta(stan,głębokość,min_lub_maks,ostatni_ruch,alfa,beta):
  niech aktualny_wynik równa się stan.pobierz_wynik
  jeśli aktualny_wynik nie jest równe 0 lub stan.pełny lub głębokość
jest równe 0:
    zwróć nowy Ruch(ostatni_ruch,aktualny_wynik)
  niech najlepszy_wynik równa się min_lub_maks razy -∞
  niech najlepszy_ruch = -1
  dla każdego możliwego ruchu (od 0 do 4 dla planszy 5×4) jako ruch:
    niech dziecko równa się kopii stan
    wykonaj bieżący ruch dla dziecko
    niech najlepsze_dziecko równa się
      minmax(dziecko,głębokość-1,min_lub_maks*-1,ruch,alfa,beta)
    jeśli (najlepsze_dziecko.wynik jest większe niż najlepszy_wynik i
           min_lub_maks to MAKS)
    lub (najlepsze_dziecko.wynik jest mniejsze niż najlepszy_wynik i
         min_lub_maks to MIN):
```

```
      niech najlepszy_wynik = najlepsze_dziecko.wynik
      niech najlepszy_ruch = najlepsze_dziecko.ruch
      jeśli najlepszy_wynik >= alfa:
        alfa = najlepszy_wynik
      jeśli najlepszy_wynik <= beta:
        beta = najlepszy_wynik
    jeśli alfa >= beta:
      zakończ
zwróć nowy Ruch(najlepszy_ruch,najlepszy_wynik)
```

## Zastosowania algorytmów przeszukiwania antagonistycznego

Algorytmy przeszukiwania sterowanego są wszechstronne i przydają się w praktycznych zastosowaniach takich jak:

- *Tworzenie agentów w grach turowych z dostępnymi kompletnymi informacjami*. W niektórych grach gracze działają w tym samym środowisku. Istnieją wysokiej jakości agenty do gier w szachy, warcaby itd. W grach z dostępnymi kompletnymi informacjami nie występują ukryte informacje i losowość.
- *Tworzenie agentów w grach turowych z niepełnymi informacjami*. W takich grach przyszłe możliwości są nieznane. Do tej grupy zalicza się na przykład poker i Scrabble.
- *Przeszukiwanie antagonistyczne i algorytm mrówkowy do optymalizacji tras*. Przeszukiwanie antagonistyczne można zastosować razem z algorytmem mrówkowym (opisanym w rozdziale 6.) do optymalizacji tras dostarczania przesyłek w miastach.

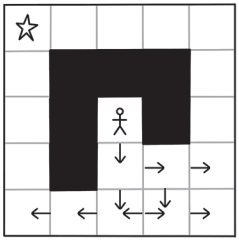
## Podsumowanie inteligentnego przeszukiwania

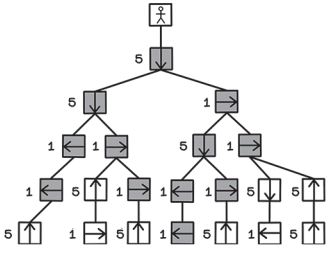
**Przeszukiwanie sterowane wbudowuje w algorytmy pewien stopień inteligencji.**

Czasem trudno jest wymyślić heurystykę, jednak jeśli jest ona dobra, stanowi wartościowe narzędzie do wydajnego znajdowania rozwiązań.

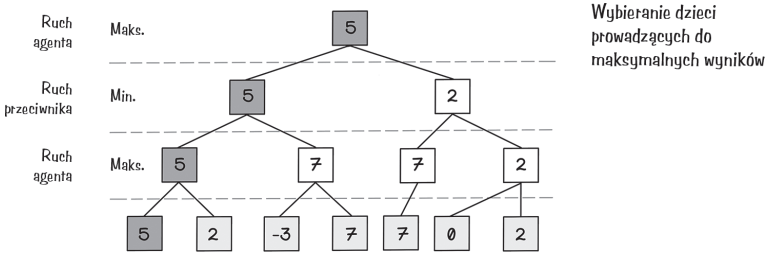
Algorytm A* wykorzystuje heurystyki i odległość od korzenia do optymalnego wyszukiwania rozwiązań.

$$f(n) = g(n) + h(n)$$

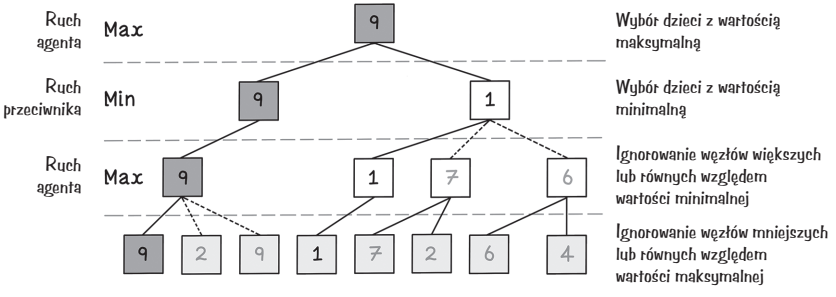




Przeszukiwanie antagonistyczne, na przykład algorytm min-max, jest przydatne, gdy coś wpływa na środowisko.




Algorytm alfa-beta pomaga zoptymalizować działanie algorytmu min-max dzięki eliminowaniu zbędnych ścieżek.




PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

## Zrozum algorytmy, a pojmiesz istotę sztucznej inteligencji!

Sztuczna inteligencja ma umożliwiać wykorzystywanie danych i algorytmów do podejmowania lepszych decyzji, rozwiązywania trudnych problemów i automatyzowania złożonych zadań. Ma też zwiększać produktywność człowieka. Obecnie sztuczna inteligencja z rozmachem wkracza do kolejnych dziedzin. Budzi zachwyt, ale też kontrowersje i obawy. Nowe narzędzia, choćby były tworzone z najlepszymi intencjami, zawsze mogą zostać wykorzystane w niewłaściwy czy szkodliwy sposób. Oznacza to, że każdy, kto rozwija nowe technologie, powinien to robić odpowiedzialnie. Aby to było możliwe, trzeba dobrze zrozumieć podstawy działania sztucznej inteligencji — algorytmy.

To praktyczny przewodnik po algorytmach sztucznej inteligencji. Skorzystają z niego programiści i inżynierowie, którzy chcą zrozumieć zagadnienia i algorytmy związane ze sztuczną inteligencją na podstawie praktycznych przykładów i wizualnych wyjaśnień. Książka pokazuje, jak radzić sobie z takimi zadaniami programistycznymi jak wykrywanie oszustw bankowych czy sterowanie pojazdem autonomicznym. Pierwsze rozdziały dotyczą podstawowych koncepcji i algorytmów i stają się punktem wyjścia do bardziej złożonych tematów: wydajnych algorytmów przeszukiwania oraz poszukiwania rozwiązań w środowisku konkurencyjnym. Przedstawiono tu zagadnienia uczenia maszynowego, w tym proces przygotowania danych, modelowania i testowania. Omówiono też zasady uczenia przez wzmacnianie za pomocą algorytmu Q-learning.

W książce:

- kategorie i znaczenie algorytmów sztucznej inteligencji
- inteligentne przeszukiwanie w procesie podejmowania decyzji
- algorytmy genetyczne i inteligencja rozproszona
- uczenie maszynowe i sieci neuronowe
- uczenie przez wzmacnianie

***Rishal Hurbans*** jest pasjonatem z cechami szalonego naukowca. Był kierownikiem zespołów i projektów, założył start-up, zajmował się też planowaniem strategicznym dla międzynarodowych firm. Wystąpił na dziesiątkach konferencji na całym świecie. Jest znawcą mechanizmów i strategii biznesowych oraz podejścia *design thinking*. Pasjonuje się sztuczną inteligencją, kulturą pragmatyzmu oraz filozofią.


ISBN 978-83-283-7507-9
9 788328 375079