Sztuczna inteligencja

Mianem sztucznej inteligencji (ang. *Artificial Intelligence — AI*) można określić dziedzinę wiedzy zajmującą się poszukiwaniem technik rozwiązywania — i ich formalnym sformułowaniem pozwalającym na implementację maszynową — problemów **trudnych**, czyli takich które ludzie rozwiązują — mniej lub bardziej wysilając swój intelekt — ale których dokładnego i ogólnego algorytmu rozwiązania nie potrafią podać.

Nie jest to precyzyjna definicja.

Czy to jest trudny problem:
$98731269868414316984251684351 \times 985316846315968463198643541684$?

A to: „Proszę, kup ładny kawałek wołowiny na pieczeń!"

Problem naprawdę mega trudny: przelać wodę ze szklanki do pojemnika. Dokładniej: mając podłączoną do komputera kamerę wideo (albo nawet dwie) i sterowaną mechanicznie rekę z palcami i przegubami, napisz program zdolny podnieść ze stolika szklankę z wodą, i przelać wodę do pojemnika. Dowolną szklankę. Z dowolnego stolika. Do dowolnego pojemnika.

Co jest istotą inteligencji naturalnej?

Komputery są tanie i szybkie, mają potężne i niezawodne pamięci, a przy tym są dokładne, nie mylą się (no, powiedzmy), i nie męczą się, zachowując swoją dokładność przez wiele godzin pracy. W czym więc problem, co jest takiego w inteligencji człowieka, z czym mają trudności komputery?

Częściowo, problem tkwi właśnie w tej wytrwałej i niezawodnej dokładności!

Ludzie rozwiązują trudne problemy stosując **abstrakcję** — wielopoziomową analizę problemu i zdolność nieschematycznej **dekompozycji** problemu, tzn. rozbijania większego problemu na mniejsze. Ich myślenie cechuje **elastyczność** — zmienny punkt widzenia i myślenie wielokierunkowe. Są zdolni do efektywnego **rozpoznawania wzorców**, **kojarzenia faktów**, oraz **wykorzystywania analogii**.

Komputery natomiast mają trudności z rozpoznawaniem odmiennych sytuacji, zmianą sposobu myślenia, i dostosowaniem go do sytuacji. Algorytmy rozpoznawania wzorców mogą być efektywne jeśli są bardzo wyspecjalizowane, ale wtedy przestają działać gdy tylko zmienia się sytuacja.

Uczenie maszynowe

Mówimy, że agent sztucznej inteligencji uczy się, jeśli poprawia wyniki swoich przyszłych działań na podstawie obserwacji swojego środowiska i wyników działań poprzednich.

Tradycyjnie, sztuczna inteligencja zajmuje się metodami **reprezentacji wiedzy** i **wnioskowania**. Przeważająca większość metod sztucznej inteligencji nie ma zdolności uczenia się. To może rodzić dwa rodzaje wątpliwości.

Po pierwsze, inteligencja naturalna niepodzielnie posiada zdolność uczenia się, nie da się oddzielić inteligentnego działania i uczenia się. Nie uznalibyśmy za inteligentnego człowieka, który nie uczy się ze swoich doświadczeń, przynajmniej w najprostszy sposób. Dlaczego więc rozdzielamy te zdolności dla inteligencji sztucznej?

Niestety, nie ma dobrej odpowiedzi na to pytanie. Prawie wszystkie najważniejsze paradygmaty sztucznej inteligencji działają bez uczenia się. Zdolność uczenia się musi być dodana.

Czy maszynowe uczenie się jest potrzebne?

Pojawia się więc druga wątpliwość: jeśli zdolność uczenia się nie jest oczywista albo konieczna, to czy na pewno jest niezbędna? Być może algorytmy sztucznej inteligencji mogą być dopracowane w 100% do perfekcji, i agent sztucznej inteligencji nie będzie już w stanie nic zyskać przez uczenie się.

Na to pytanie istnieje odpowiedź, i można wymienić szereg powodów.

Po pierwsze, twórcy systemów sztucznej inteligencji nie są w stanie przewidzieć wszystkich możliwych sytuacji w jakich znajdzie się system. Na przykład, robot poruszający się w labiryncie musi nauczyć się topografii konkretnego labiryntu, w którym się znajdzie.

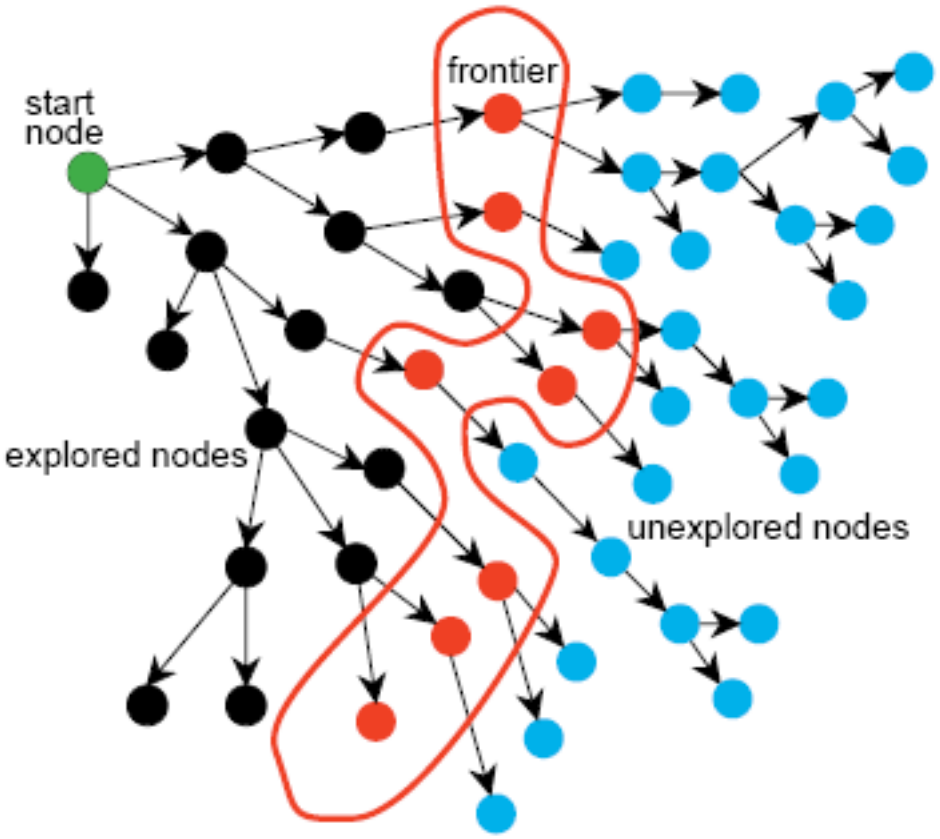
Po drugie, podobnie nie można przewidzieć wszystkich możliwych zmian w czasie. Np. program mający przewidywać zmiany kursów akcji musi nauczyć się dostosować swoje przewidywania, gdy warunki zmienią się w nieoczekiwany sposób.

Po trzecie, niekiedy programiści po prostu nie potrafią zaprogramować pewnych rozwiązań. Na przykład, ludzie potrafią sprawnie rozpoznawać twarze osób znajomych. Nie są jednak znane żadne skuteczne algorytmy pozwalające osiągnąć podobną zdolność, z wyjątkiem za pomocą metod maszynowego uczenia się.

Rozwiązywanie problemów przez przeszukiwanie

Proces poszukiwania rozwiązania można wyrazić za pomocą następujących pojęć, i przedstawić na skierowanym **grafie przeszukiwania**:

- węzeł startowy odpowiada początkowemu stanowi agenta,
- inne węzły grafu odpowiadają różnym stanom agenta tworzącym **przestrzeń stanów**,
- istnieje **stan/stany docelowy/we**, który agent chce osiągnąć,
- łuki grafu odpowiadają możliwym przejściom od stanu do stanu wynikającym z wykonywanych przez agenta akcji; mogą one mieć koszty,
- część przestrzeni jest zbadana i znana agentowi, ale pozostała część nie została jeszcze odkryta, i może ona być nieskończona.



Przeszukiwanie jest składnikiem wszystkich metod sztucznej inteligencji, a zdolność efektywnego przeszukiwania wydaje się być nieodłączną cechą samej inteligencji.

Eksponencjalne przestrzenie stanów

Spróbujmy dokonać charakterystyki numerycznej przestrzeni stanów. Dla uproszczenia załóżmy, że w każdym stanie dostępny jest stały zestaw przejść między stanami, których liczbę oznaczamy b (*branching factor*).

Ile stanów musi zbadać agent, aby przeszukać przestrzeń do głębokości d? Będzie to b dla $d = 1$, b^2 dla $d = 2$, b^3 dla $d = 3$, ... i ogólnie b^d dla głębokości d.

A jak głęboko agent będzie musiał zazwyczaj przeszukiwać? Zależy to od tego, jak głęboko znajduje się stan rozwiązania. Może być blisko stanu początkowego, co oznacza, że problem jest łatwy, lub może być daleko od stanu początkowego, lub głęboko w przestrzeni poszukiwań, co oznacza, że problem jest trudny do rozwiązania.

Zakładając, że agent będzie musiał przeszukać znaczną część przestrzeni dla danego problemu, możemy powiedzieć, że ilość przeszukiwań rośnie wykładniczo w stosunku do trudności problemu, więc proces wyszukiwania będzie miał złożoność $O(b^d)$.

To zła wiadomość. W przypadku problemów wykładniczych, zakładając $b = 10$, rozwiązanie łatwego problemu, takiego jak $d = 5$, będzie wymagało zbadania 10^5 stanów, co może zająć zaledwie 1 sekundę. Bardziej skomplikowany problem z $d = 10$ będzie wymagał 10^{10} stanów i 10^5 sekund, czyli nieco ponad jednego dnia obliczeń, ale problem z $d = 15$ będzie wymagał 317 lat obliczeń. Nawet w przypadku masowo równoległych procesorów prawdopodobnie będzie to nieakceptowalne.

Heurystyki i funkcje oceny stanu

Algorytmy dotychczas przedstawione są ogólne i nie wymagają do swojej pracy strategii poinformowanej. Jednak w każdym praktycznym zagadnieniu posiadanie takiej strategii jest bardzo pożądane.

Heurystyką będziemy nazywać wiedzę o dziedzinie problemowej:

- której nie można uzyskać z syntaktycznej analizy opisu problemu,
- która może nie mieć formalnie poprawnego uzasadnienia, a także — co więcej — która może nie w każdym przypadku sprawdzać się, i czasami dawać mylne wskazówki,
- ale która ogólnie pomaga w dokonywaniu dobrych wyborów w przeszukiwaniu.

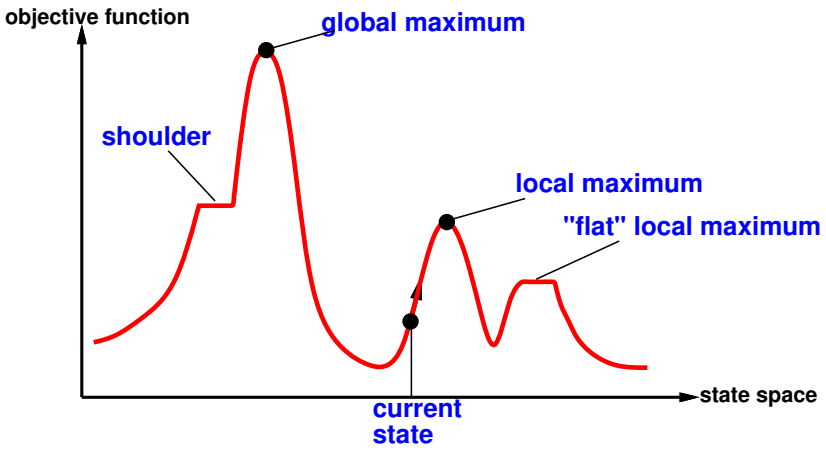
Posiadanie heurystyki pozwala budować strategie poinformowane. Ogólnym i często stosowanym schematem konstrukcji strategii wykorzystującym informację heurystyczną, jest **statyczna funkcja oceny stanu**. Dla każdego stanu określa ona jego „dobroć", czyli szanse, że przez ten stan prowadzi droga do rozwiązania. Wartość tej funkcji można również interpretować jako miarę odległości stanu od rozwiązania.

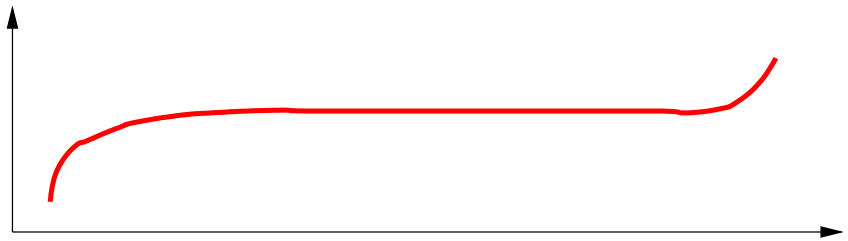
Metody gradientowe

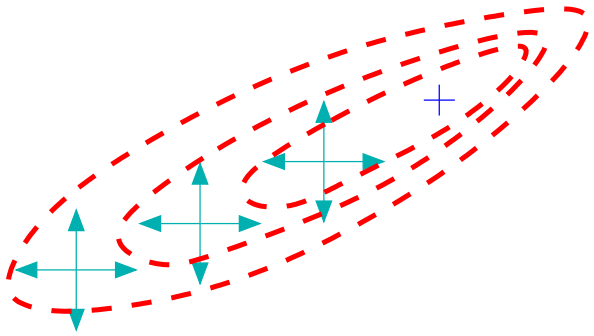
Funkcję oceny stanu można w przeszukiwaniu zastosować bezpośrednio. Prowadzi to do metody lub metod **gradientowych** (*hill-climbing*). Metody te określa się w informatyce jako metody zachłanne.

Ich bezpośrednie zastosowanie ograniczone jest do dziedzin z bardzo regularną funkcją oceny (np. ściśle monotoniczną). W praktyce mamy typowo do czynienia z następującymi problemami:

1. lokalne maksima funkcji oceny
2. obszary „plateau” funkcji oceny
3. ukośne granie funkcji oceny

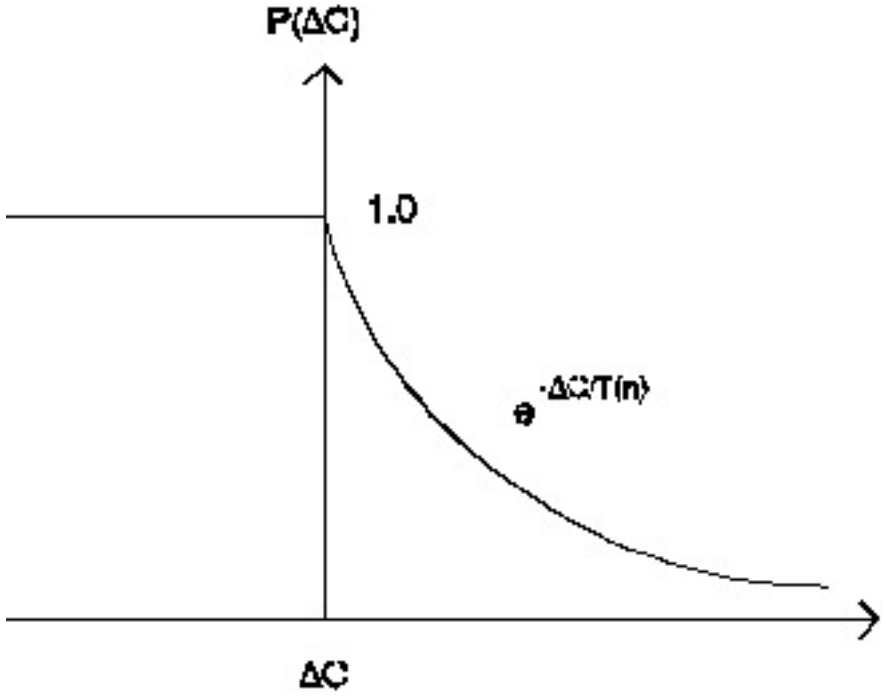






Wyżarzanie

Skuteczną i często stosowaną grupę metod gradientowych stanowi technika zwana wyżarzaniem (*simulated annealing*). Jej nazwa odwołuje się do analogii z procesem wytapiania metalu, kiedy stopniowe i powolne zmniejszanie temperatury pozwala osiągnąć stan globalnego optimum energetycznego, z pełnym uporządkowaniem cząsteczek w całej objętości metalu.



Metoda polega na generowaniu ruchów losowych, i następnie wykonywaniu ich, lub nie, zgodnie z przedstawionym na wykresie rozkładem prawdopodobieństwa.

Jak widać, jeśli wygenerowany ruch poprawia wartość funkcji oceny to jest zawsze wykonywany, natomiast jeśli ją pogarsza to jest wykonywany z prawdopodobieństwem $p < 1$ zależnym od stopnia pogorszenia oceny, w porównaniu ze stanem aktualnym.

Jednocześnie, w trakcie pracy algorytmu stopniowo obniżana jest temperatura, co powoduje zmniejszanie prawdopodobieństwa wyboru ruchów „złych”.

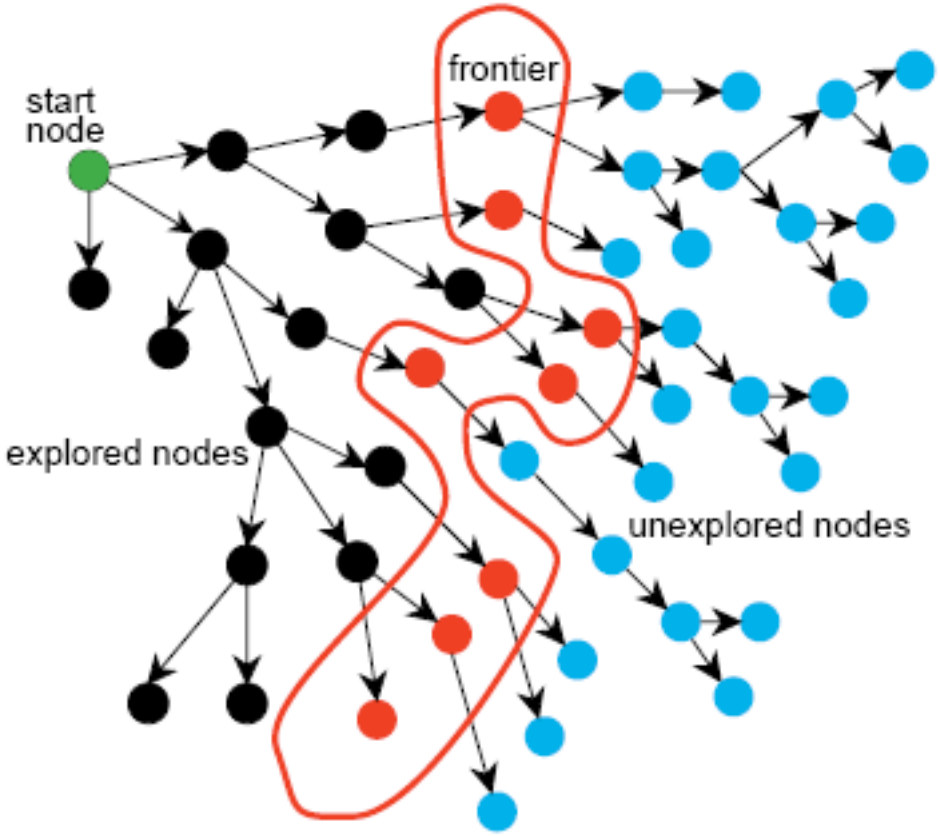
Metodę wyżarzania stosuje się z powodzeniem do projektowania układów VLSI, sieci różnego rodzaju, przydziału zadań w procesach produkcyjnych, i innych zadań optymalizacji procesów złożonych. Problemem w jej zastosowaniu jest dobór parametrów, np. algorytmu obniżania temperatury.

Przeszukiwanie grafów

Przypomnijmy sobie wersję algorytmu BT z iteracyjnym pogłębianiem, i konieczność wielokrotnego przeszukiwania początkowej części przestrzeni. Aby uniknąć wielokrotnego odwiedzania tych samych stanów można użyć struktury grafowej do pamiętania zbadanych już części przestrzeni stanów. Algorytmy, które używają takiej struktury są algorytmami **przeszukiwania grafów**.

Strategie przeszukiwania grafów:

- strategia wszerz BFS (*breadth-first search*),
- strategia wgłąb DFS (*depth-first search*),
- algorytm Dijkstry?,
- inne strategie, w tym wykorzystujące heurytyki.



Przykład: 8-ka (8-puzzle)

Układanka 15-tka (*15-puzzle*) — popularna w szkole podstawowej.

8-ka (*8-puzzle*) — zmniejszona wersja, odpowiednia do ilustracji działania różnych strategii i algorytmów sztucznej inteligencji.

Przeszukiwanie wszerz (BFS)

- Badaj wszystkie stany w odległości d od stanu początkowego s_0 przed zbadaniem jakiegokolwiek stanu w odległości $(d+1)$ od s_0.
- Zawsze gwarantuje znalezienie rozwiązania jeśli tylko istnieje.
- Co więcej, zawsze znajduje rozwiązanie optymalne (tzn. znajduje najkrótszą drogę ze stanu początkowego do każdego stanu).
- Złożoność pamięciowa i czasowa fatalna, obie $O(b^d)$, gdzie:
  b - średnia liczba gałęzi wyrastających z węzła (tzw. *branching factor*),
  d - odległość stanu początkowego od rozwiązania (liczba operatorów).
- Praktycznie jednakowa złożoność przypadku najgorszego i średniego (jak również najlepszego).

Przeszukiwanie wszerz — przykład

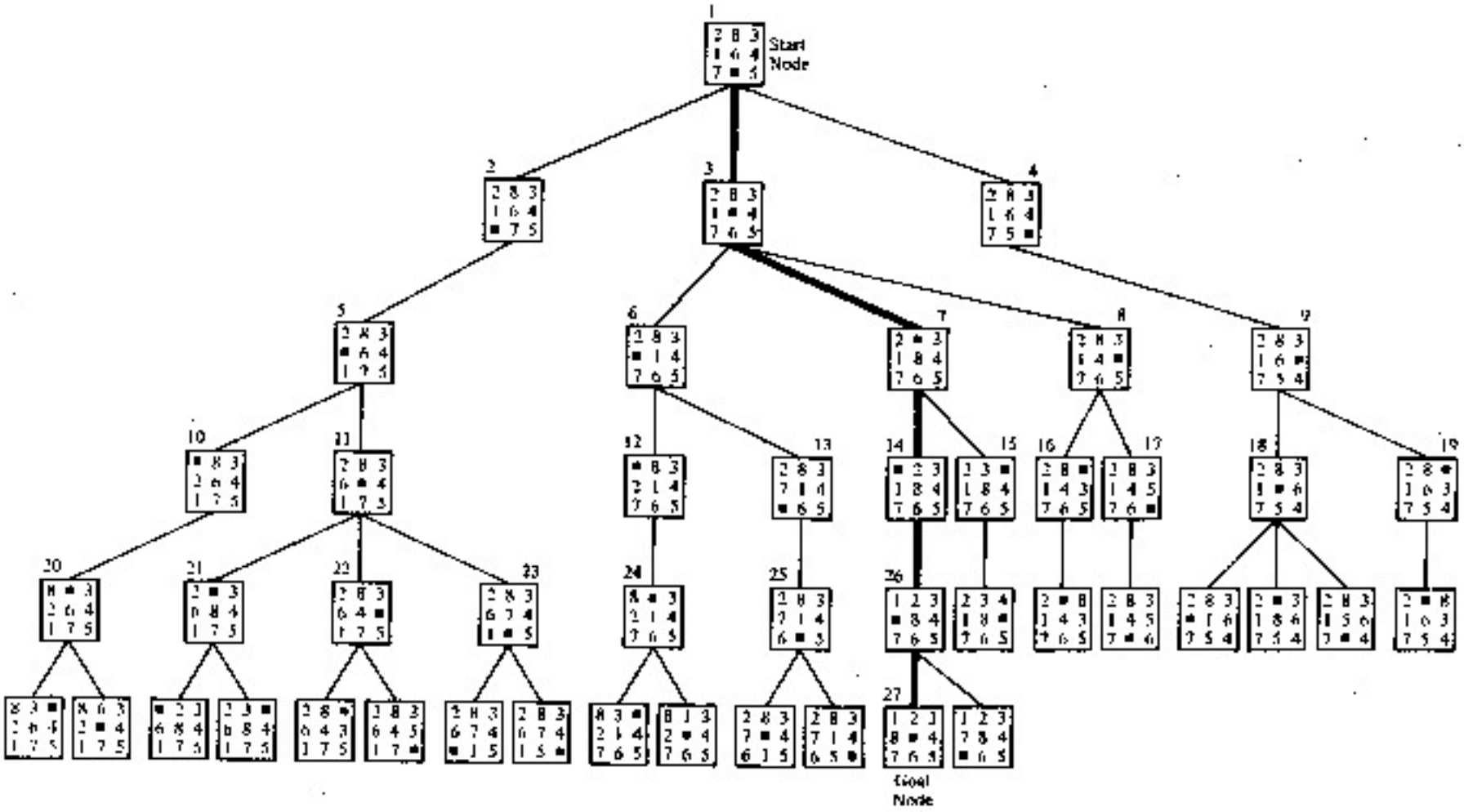
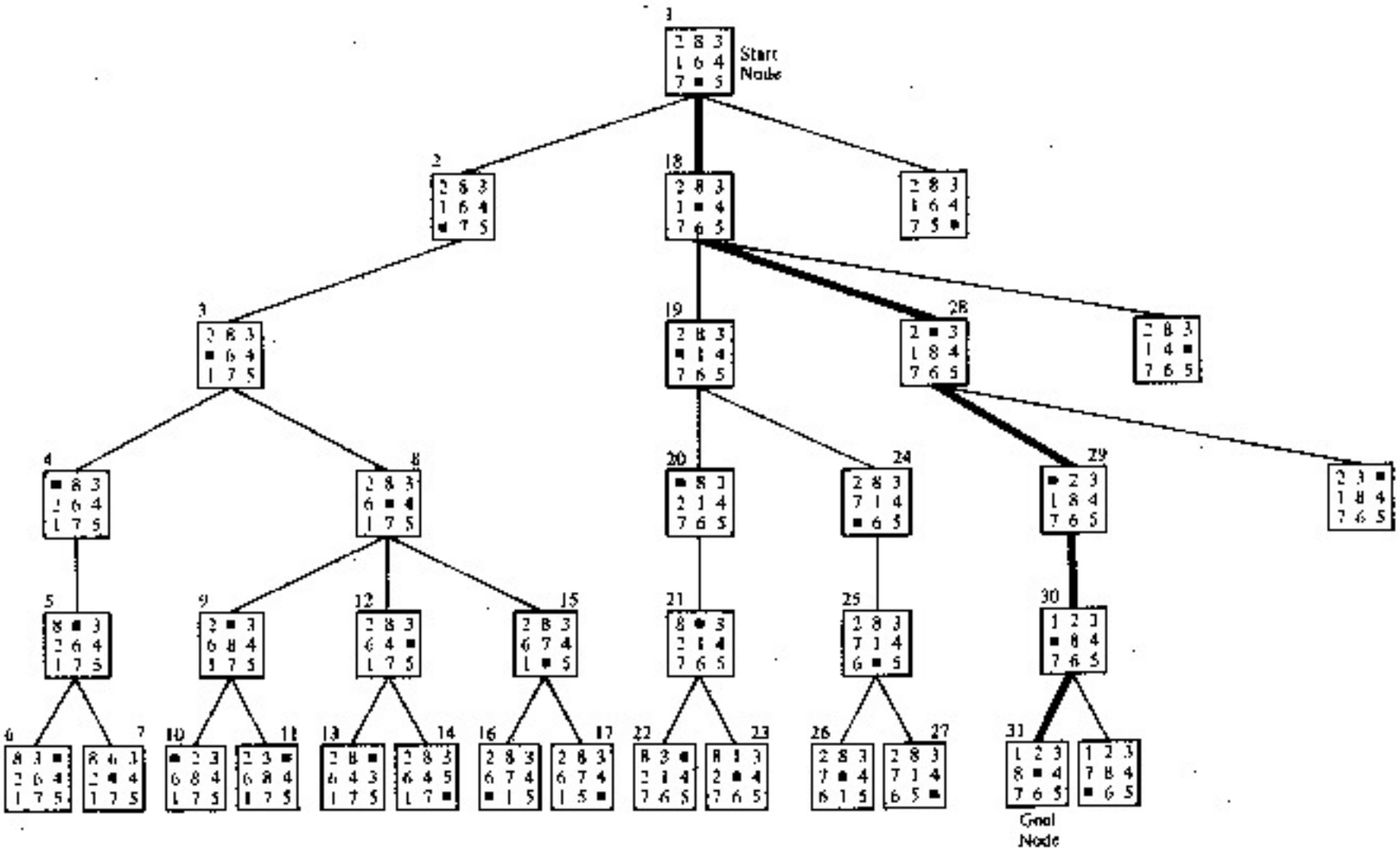


Diagram przedstawia fragment grafu przeszukiwania wszerz. Numery nad planszami (1–26) pokazują tu kolejność wyboru węzłów do ekspansji grafu.

Przeszukiwanie wgłąb (DFS)

- Badaj wszystkie nowo odkryte stany pochodne (potomki) danego stanu n przed powrotem do badania sąsiadów stanu n.
- Nie daje żadnych z gwarancji BFS (pewności znalezienia rozwiązania optymalnego, albo w ogóle znalezienia jakiegoś rozwiązania).
- Złożoność obliczeniowa przypadku najgorszego: przetwarzanie i pamiętanie wszystkich stanów.
- Złożoność przypadku średniego: $O(b^d)$ pamięciowa i czasowa.
- Dla przestrzeni nieskończonych algorytm może zagłębić się w nieskończoną podprzestrzeń (gałąź) niezawierającą rozwiązania, i ... nigdy z niej nie wrócić!!!
- Efektywność algorytmu gwałtownie polepsza się dla przypadków istotnie lepszych niż średni (czyli wyjątkowo szczęśliwych), zatem sens jego stosowania jest tylko w połączeniu z dobrymi heurystykami.

Przeszukiwanie wgłąb — przykład



Fragment „przeciętnego” grafu przeszukiwania wgłąb z ograniczeniem głębokości do 5. Numery węzłów pokazują kolejność wyboru węzłów do ekspansji grafu.

Wykrywanie powtarzających się stanów

Jednym z problemów algorytmu przeszukiwania wgłąb — jak również wszystkich innych algorytmów przeszukiwania — jest możliwość powstawania pętli. Jeśli algorytm kiedykolwiek wygeneruje opis stanu, do którego doszedł, ale który już istnieje na jego drodze od stanu początkowego, to nieuchronnie zacznie powtarzać badanie stanów wcześniej zbadanych.

Zjawisku temu można oczywiście zapobiec. Najprostszym sposobem byłoby sprawdzenie, po wygenerowaniu każdego nowego stanu, czy ten stan nie znajduje się już na bieżącej ścieżce od stanu początkowego.

Można również sprawdzić dokładniej — czy nowo wygenerowany stan nie został już w ogóle kiedykolwiek wcześniej znaleziony, i zbadany. Wymaga to pamiętania zbioru stanów zbadanych, tzw. listy *Closed*. Lista ta w algorytmie rekurencyjnym musi być globalna i każdy nowo wygenerowany opis stanu musi być porównywany ze wszystkimi stanami już obecnymi na liście.

Jedno i drugie sprawdzanie jest dość kosztowne obliczeniowo. Dla zaoszczędzenia czasu można je pominąć, ryzykując jednak zapętlenie procedury.

Ograniczenie głębokości z iteracyjnym pogłębianiem

Poważnym problemem dla algorytmu wgłąb są nieskończone przestrzenie, z którymi algorytm ogólnie sobie nie radzi. Podobnie zresztą jak inne algorytmy o charakterze (hura-)optymistycznym, które preferują marsz do przodu, o ile tylko jest możliwy.

Prostym rozwiązaniem tego problemu jest **ograniczenie głębokości** przeszukiwania do jakiejś „rozsądnej" wartości. Zauważmy, że poza zabezpieczeniem przed nieskończonymi przestrzeniami, zabezpiecza ono jednocześnie przed wpadnięciem w pętle stanów, co pozwala pominąć wykrywanie powtarzających się stanów.
W ogólnym przypadku może nie być jednak łatwe określenie takiej wartości, a jej niedoszacowanie grozi oczywiście porażką algorytmu i nieznalezieniem rozwiązania, które istnieje.

Dla szeregu algorytmów podobnie optymistycznych (preferujących ruchy wgłąb) stosuje się zatem **ograniczenie głębokości z iteracyjnym pogłębianiem**. Ten wariant gwarantuje znalezienie rozwiązania, o ile istnieje.

Grafy z wagami — algorytm Dijkstry

W przypadku zagadnienia, gdzie koszty wykonania różnych ruchów nie są sobie równe, i chcemy znaleźć rozwiązanie optymalne, możemy zamodelować to za pomocą grafu skierowanego z wagami połączeń. Dla takiego grafu znamy już dobre rozwiązanie poszukiwania najkrótszej ścieżki (a dokładniej, wszystkich najkrótszych ścieżek) z danego źródła. Jest nim np. algorytm Dijkstry.

Jednak nie zawsze jest to dobre rozwiązanie. Algorytm Dijkstry zakłada, że cały graf jest znany i w pełni wygenerowany. Nie nadaje się zatem dla przestrzeni nieskończonych, a także bardzo wielkich.

Również dla przestrzeni skończonych o ograniczonej wielkości zastosowanie tego algorytmu może być trudne. Jego złożoność wynosi $O(V^2)$, a liczba wierzchołków grafu na głębokości d może być eksponencjalna względem d, a więc efektywność tego rozwiązania nie jest dobra.

Zastosowanie heurystyk — przeszukiwanie najpierw-najlepszy

Zastosowanie heurystycznej funkcji oceny do przeszukiwania na grafach w najprostszym przypadku daje tzw. przeszukiwanie **najpierw-najlepszy** (*best-first search*). W każdej chwili wykonuje ono ruch, który minimalizuje funkcję oceny. Jeśli funkcja oceny jest dobra, właściwie wybiera stany do analizy, i odpowiednio maleje wzdłuż drogi do rozwiązania, to wtedy ta metoda „idzie" bezpośrednio do celu, nie tracąc czasu na rozwijanie jakichkolwiek niepotrzebnych węzłów grafu.

Również w przypadku drobnych defektów funkcji oceny, kiedy niektóre jej wartości są nietrafne i źle oceniają stany, ale po rozwinięciu kilku niepotrzebnych węzłów funkcja ocenia dalsze węzły w przybliżeniu poprawnie, ten schemat przeszukiwania dobrze się sprawdza.

Kłopoty zaczynają się jednak kiedy funkcja ma jakiś błąd systematyczny, np. jako najlepszą konsekwentnie wskazuje drogę, która w ogóle nie prowadzi do celu. Wtedy metoda najpierw-najlepszy ma takie same wady jak metoda wgłąb, pomimo, iż funkcja być może poprawnie oszacowuje wiele węzłów.

Przeszukiwanie najpierw-najlepszy jest przykładem metody zachłannej, która nie gwarantuje sukcesu.

Modyfikacja funkcji wyboru — koszt przebytej drogi

Rozważmy następujące deterministyczne funkcje oceny (węzła):

$h^*(n)$ – koszt kosztowo-optymalnej drogi z n do celu
$g^*(n)$ – koszt kosztowo-optymalnej drogi z s_0 do n

Wtedy:

$f^*(n) := g^*(n) + h^*(n)$
$f^*(n)$ – koszt kosztowo-optymalnej drogi z s_0 do celu biegnącej przez n

Znajomość funkcji $f^*(n)$ pozwoliłaby zawsze wybierać tylko węzły leżące na optymalnej drodze od początku do celu. Podobnie zresztą wystarczyłaby do tego znajomość samej funkcji $h^*(n)$.

Niestety, zwykle funkcje $h^*(n)$ ani $g^*(n)$ nie są dostępne. Jesteśmy zmuszeni posługiwać się ich przybliżeniami, które pozwalają jedynie aproksymować wybieranie właściwych węzłów. Jednak gdy posługujemy się przybliżeniami, wtedy przeszukiwanie bazujące na funkcji $f^*(n)$ nie musi już dawać takich samych wyników jak to opierające się na funkcji $h^*(n)$.

Modyfikacja funkcji wyboru — algorytm A*

Rozważmy zatem następujące heurystyczne funkcje oceny węzła:

$h(n)$ – funkcja heurystyczna aproksymujaca $h^*(n)$
$g(n)$ – koszt najlepszej znanej drogi z s_0 do n; zauważmy $g(n) \geq g^*(n)$
$f(n) := g(n) + h(n)$

Jak działa tak określona strategia? Jeśli funkcja $h(n)$ oszacowuje $h^*(n)$ bardzo precyzyjnie, to algorytm działa niemal idealnie, i zmierza prosto do celu.

Jednak gdy funkcja $h(n)$ popełnia błędy, i np. optymistycznie określa jakieś stany jako lepsze niż są one w rzeczywistości, to algorytm najpierw podąża w ich kierunku, zwabiony niską wartością funkcji $h(n)$, gdy $g(n)$ jest pomijalne.

Po jakimś czasie, tak błędnie oszacowane ścieżki przestają być atrakcyjne, ze względu na narastający składnik $g(n)$, i algorytm z konieczności przerzuca swoje zainteresowanie na inne atrakcyjne węzły. Przy tym na atrakcyjność nie ma wpływu, czy są one bardziej czy mniej oddalone od startu. Decyduje łączna ocena, czy przez dany stan prowadzi najlepsza droga do rozwiązania.

Algorytm przeszukiwania grafów stosujący powyższą funkcję $f(n)$ jako swoją strategię nazywa się **algorytmem A***.

Funkcja oceny w algorytmie A*

Składniki $h(n)$ i $g(n)$ reprezentują w funkcji $f(n)$ dwie przeciwstawne tendencje: optymizm ($h(n)$) i konserwatyzm ($g(n)$). Możemy całkiem swobodnie sterować strategią w jedną lub drugą stronę stosując wzór:

$$f(n) := (1 - k) * g(n) + k * h(n)$$

Zwiększając współczynnik wagi k możemy nadawać przeszukiwaniu charakter bardziej agresywny (i ryzykowny), gdy np. mamy zaufanie do funkcji $h(n)$ i chcemy posuwać się szybko do przodu. z kolei zmniejszając ten współczynnik, zapewniamy dokładniejsze badanie przestrzeni, posuwając się wolniej do przodu, ale kompensując niektóre błędy funkcji $h(n)$.

Zauważmy, że w skrajnych przypadkach, $k = 1$ daje przeszukiwanie najpierw-najlepszy, natomiast $k = 0$ daje przeszukiwanie równokosztowe.

Jednak największy wpływ na przebieg przeszukiwania ma jakość funkcji $h(n)$.

Własności funkcji h(n) w algorytmie A*

Heurytyczną funkcję oceny $h(n)$ w algorytmie A* nazywamy **dopuszczalną** (*admissible*) gdy ogranicza ona od dołu rzeczywistą funkcję $h^*(n)$, czyli $\forall n\ h(n) \leq h^*(n)$. Dopuszczalność oznacza chroniczne niedoszacowywanie przyszłych kosztów, zatem bywa nazywane optymizmem. Można dowieść, że jeśli tylko istnieje ścieżka z węzła początkowego do celowego, to A* z dopuszczalną heurystyką zawsze znajduje optymalną taką ścieżkę.

Czy trudno jest znaleźć taką dopuszczalną heurystykę? Niekoniecznie, np. $h(n) \equiv 0$ rzeczywiście ogranicza z dołu $h^*(n)$, dla dowolnego zagadnienia. Czy taka trywialna heurystyka może być przydatna? Odpowiedź brzmi: raczej rzadko. Taki algorytm wybiera zawsze węzły o najkrótszej drodze z s_0, a zatem jest to algorytm wszerz (ogólniej: równego kosztu), który rzeczywiście zawsze znajduje optymalną drogę, ale samo w sobie to jeszcze nie jest wielka zaleta.

Oczywiście im lepiej $h(n)$ przybliża $h^*(n)$ tym efektywniejsze przeszukiwanie. Jeśli mamy dwie funkcje $h_1(n)$, $h_2(n)$, takie że dla wszystkich węzłów $h_1(n) < h_2(n) \leq h^*(n)$ to można dowieść, że użycie h_1 prowadzi do rozwinięcia co najmniej tyle samo węzłów co h_2.

Własności funkcji h(n) w algorytmie A* (cd.)

Dopuszczalność funkcji $h(n)$ jest ciekawą własnością, którą często można udowodnić dla funkcji bardzo zgrubnie oszacowującej $h^*(n)$, ale już niekoniecznie dla mozolnie opracowanej funkcji, np. z wykorzystaniem numerycznego uczenia się na serii przykładów (co jak się okaże jest jedną z metod konstrukcji takich funkcji).

Jeszcze mocniejszą własnością heurystycznej funkcji oceny $h(n)$ jest jej **spójność** (*consistency*), zwana również **ograniczeniem monotonicznym** (*monotone restriction*), lub po prostu nierównością trójkąta:

$$\underset{n_i \to n_j}{\forall} \; h(n_i) - h(n_j) \leq c(n_i, n_j)$$

Można dowieść, że dla funkcji $h(n)$ spełniających ograniczenie monotoniczne algorytm A* zawsze ma już znalezioną optymalną drogę do każdego węzła, który decyduje się rozwijać. W praktyce pozwala to nieco uprościć implementację algorytmu przeszukiwania, jeśli wiemy, że funkcja oceny jest spójna.

Złożoność obliczeniowa algorytmu A*

Dla większości praktycznych problemów liczba węzłów przestrzeni stanów rośnie eksponencjalnie z długością poszukiwanego rozwiązania. Oczywiście, efektywna heurystyka mogłaby zmniejszyć złożoność obliczeniową algorytmu.
Pytanie, kiedy moglibyśmy na to liczyć?

Można dowieść, że aby to osiągnąć, czyli aby algorytm A* działał w czasie wielomianowym, błąd w heurystycznej funkcji oceny nie powinien przekroczyć logarytmu rzeczywistej długości rozwiązania:

$$|h(n) - h^*(n)| \leq O(\log h^*(n))$$

Pytanie: czy takie heurystyki są praktyczne?

W praktycznych przypadkach nie można liczyć na znalezienie tak dobrych heurystyk. Zatem algorytm A* należy ogólnie uważać za eksponencjalny. To jednak zwykle nie jest jego największą wadą. Podobnie jak wszystkie algorytmy przeszukiwania na grafach, przechowuje on wszystkie węzły grafu w pamięci i z reguły wyczerpuje pamięć komputera dużo wcześniej niż dostępny czas!!

Modyfikacje algorytmu A* z ograniczonymi wymaganiami pamięciowymi

Istnieją modyfikacje algorytmu A* pozwalające pokonać problemy z zapotrzebowaniem na pamięć.

Algorytm IDA* (*Iterative-Deepening A**) dodaje standardowe ograniczenie głębokości. Po osiągnięciu limitu głębokości przeszukiwania jest on zwiększany, przy jednoczesnym usunięciu przebadanych węzłów grafu z pamięci.

Algorytm RBFS (*Recursive Best-First Search*) jest podobny do algorytmu BT w wersji rekurencyjnej. Przeszukuje on rekurencyjnie pojedynczą ścieżkę grafu, pamiętając jednocześnie na każdym poziomie rekurencji najlepszą alternatywę pojedynczego kroku. Kiedy aktualnie analizowana ścieżka okazuje się gorsza od tej alternatywy, algorytm wraca, kasując wyniki swojej pracy (lecz wchodząc w nowe wywołania rekurencyjne, ponownie zapamiętuje najlepszą alternatywę).

Algorytm SMA* (*Simplified Memory-Bounded A**) działa dokładnie jak A*, ale tylko do momentu zapełnienia całej dostępnej pamięci. W tym momencie, algorytm kontynuuje pracę, kasując jednak najgorsze znane węzły grafu aby zrobić miejsce na nowo odkrywane stany. Jednak oszacowanie skasowanych węzłów jest przechowywane w ich rodzicach, aby możliwe było ponowne podjęcie przeszukiwania w danej części grafu.

Algorytm A* w praktyce

Dobrym pytaniem jest, czy algorytmy heurystycznego przeszukiwania grafów, takie jak A*, mają zastosowania praktyczne w świecie rzeczywistym.

Odpowiedź na to pytanie brzmi: tak, w pewnych ograniczonych dziedzinach, jak np. planowanie optymalnej trasy przejazdu robota, albo znajdowanie najkrótszej drogi w grach komputerowych.

Algorytm A* jest heurystyczną wersją algorytmu Dijkstry (1959) obliczającego najkrótsze drogi od ustalonego węzła do wszystkich pozostałych węzłów grafu.

Algorytm Dijkstry jest również stosowany w wielu zagadnieniach technicznych, jak np. sieciowe protokoły trasowania (*routing*), takie jak OSPF, oraz znajdowanie drogi na mapie w nawigacjach GPS. W tych ostatnich zastosowaniach, ze względu na wielkość grafu, algorytm Dijkstry musi być wspomagany przez dodatkowe techniki. Mogą to być właśnie heurystyki, albo wprowadzenie abstrakcji i hierarchii ścieżek. Jednak ze względu na komercyjny aspekt tej bardzo rozwijającej się technologii, techniki nie są zbyt często szczegółowo opisywane.

Znaczenie heurystyk — efektywny współczynnik rozgałęzienia

Niezależnie od bezwzględnej eksponencjalnej złożoności obliczeniowej algorytmu A*, ma on szereg zastosowań w ograniczonych dziedzinach. W tych zastosowaniach efektywność pracy algorytmu zależy od zastosowanej heurystyki. Chcemy przyjrzeć się praktycznym aspektom użycia heurystyk. Jedną z miar jakości heurystyki jest **efektywny współczynnik rozgałęzienia** (*effective branching factor*) b^*.

Współczynnik ten definiujemy następująco. Jeśli drzewo przeszukiwania ma w każdym węźle dokładnie b potomków, to liczba węzłów tego drzewa wygenerowanych przez algorytm BFS od węzła startowego do głębokości d N spełnia warunek:

$$N + 1 = 1 + b + b^2 + b^3 + \ldots + b^d$$

Jeżeli algorytm przeszukiwania heurystycznego A* w czasie pracy dla rozwiązania na głębokości d wygeneruje N^* węzłów, to efektywny współczynnik rozgałęzienia odpowiadający temu wyszukiwaniu spełnia warunek:

$$N^* + 1 = 1 + b^* + (b^*)^2 + (b^*)^3 + \ldots + (b^*)^d$$

Na przykład, jeśli algorytm A* znajdzie rozwiązanie na głębokości d =5, generując $N^* = 52$ węzły, to wartość $b^* \approx 1.92$

Przykład: proste heurystyki dla 8-puzzle

Jako ilustrację kolejnych zagadnień wykorzystamy przykład układanki 8-puzzle. Liczba osiągalnych stanów w tej grze wynosi $9!/2 = 181400$, a więc nie przekracza możliwości pamięciowych komputera. Jednak już dla poważniejszego warianty 15-puzzle istnieje $16!/2$, czyli ponad 10 bilionów stanów, i już w tym przypadku skuteczna heurystyka jest bardzo przydatna.

Układanka ta ma długą historię badań, zarówno dla 15-puzzle, jak i dla większych wariantów. Podstawowe, niemal oczywiste, propozycje heurystyk są następujące:

- $h_1(s)$ — policz elementy nie na swoich miejscach,
- $h_2(s)$ — dla wszystkich elementów nie na swoich miejscach, zsumuj odległości od ich właściwych miejsc w pionie i w poziomie (tzw. odległość Manhattanu).

Oba te warianty wydają się bardzo uproszczonym podejściem do szacowania długości rozwiązania, ale mają wspólną interesującą własność. Obie są heurystykami dopuszczalnymi, to znaczy obie niedoszacowują prawdziwej odległości od rozwiązania we wszystkich przypadkach. Z tego wynika, że wykorzystanie ich w algorytmie A* gwarantuje, że o ile tylko rozwiązanie istnieje (stan docelowy jest osiągalny ze startowego), to zostanie znalezione rozwiązanie optymalne (najmniejsza liczba kroków).

Jakość funkcji heurystycznej a koszt przeszukiwania A*

Poniższa tabelka ilustruje porównanie kosztu przeszukiwania i efektywnego współczynnika rozgałęzienia dla algorytmów BFS i A* z funkcjami h_1 i h_2. Wyniki zostały uśrednione dla 100 losowo wygenerowanych przypadków 8-puzzle dla różnych długości rozwiązania d.

	Koszty przeszukiwania (liczba węzłów)			Efektywny współczynnik rozgałęzienia b^*		
d	BFS	A*(h_1)	A*(h_2)	BFS	A*(h_1)	A*(h_2)
6	128	24	19	2.01	1.42	1.34
8	368	48	31	1.91	1.40	1.30
10	1033	116	48	1.85	1.43	1.27
12	2672	279	84	1.80	1.45	1.28
14	6783	678	174	1.77	1.47	1.31
16	17270	1683	364	1.74	1.48	1.32
18	41558	4102	751	1.72	1.49	1.34
20	91493	9905	1318	1.69	1.50	1.34
22	175921	22955	2548	1.66	1.50	1.34
24	290082	53039	5733	1.62	1.50	1.36
26	395355	110372	10080	1.58	1.50	1.35
28	463234	202565	22055	1.53	1.49	1.36

(Tabelka zaczerpnięta z podręcznika Russella i Norviga.)

Lepsza heurystyka dla 8-puzzle

Można skonstruować znacznie lepsze heurystyki dla układanek takich jak 8-puzzle i 15-puzzle, które znacznie bardziej realistycznie oszacowują trudność rozwiązania danej instancji problemu. Chcemy oszacować liczbę kroków niezbędną do przejścia od danego stanu początkowego do określonego stanu docelowego.

Dobrą miarą takiej trudności rozwiązania jest funkcja $h_3(s)$ obliczana dla elementów na obrzeżu układanki, biorąc 0 dla elementów, po których następuje ich właściwy prawy sąsiad, i 2 dla każdego elementu, po którym następuje niewłaściwy element. Środek wnosi 1, jeśli jest.

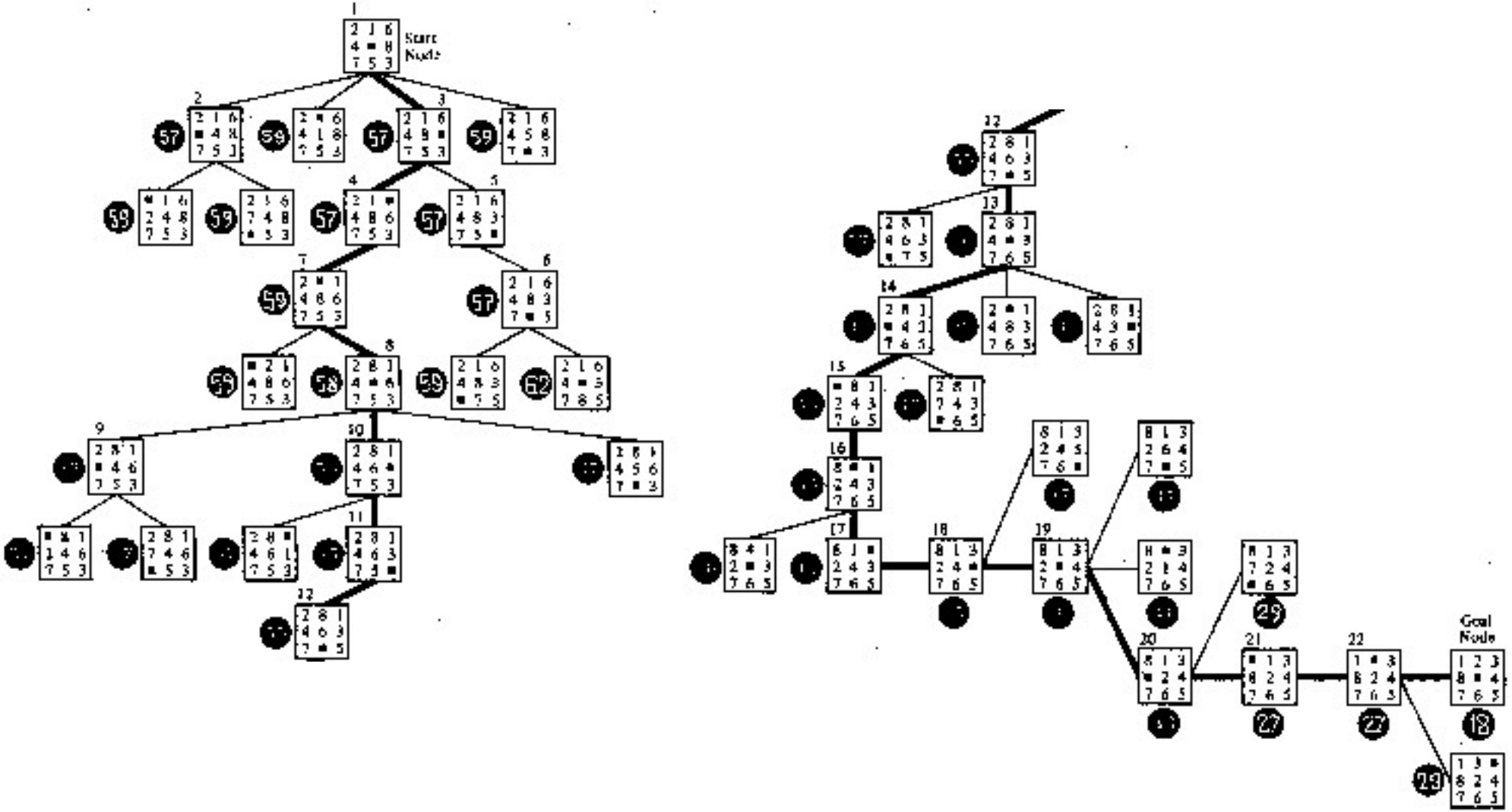
Jeszcze lepszą heurystyką okazuje się funkcja $h_4(s) = h_2(s) + 3 * h_3(s)$

Ani $h_3(s)$ ani $h_4(s)$ nie są dolnymi oszacowaniami rzeczywistej odległości do rozwiązania układanki, a pomimo to $h_4(n)$ jest jedną z najlepszych funkcji oceny dla układanki 8-puzzle, dającą niezwykle ukierunkowane i efektywne przeszukiwanie.

Zauważmy, że dolnym oszacowaniem odległości od rozwiązania, zatem gwarantującym znalezienie rozwiązania optymalnego, jest funkcja $h_0(n) \equiv 0$. Jest to ilustracja ogólnego faktu, że poprawność formalna nie zawsze idzie w parze z dobrą efektywnością obliczeniową.

Przeszukiwanie heurystyczne drzewa 8-puzzle

W przedstawionej tabelce porównania funkcji heurystycznych brak jest wyników dla najlepszej funkcji h_4. Pewną ilustracją działania tej funkcji jest poniższe przykładowe drzewo przeszukiwania, w którym rozwiązanie znajduje się na poziomie 18, a drzewo zawiera łącznie 44 węzły. Jego efektywny współczynnik rozgałęzienia wynosi 1.09



Konstrukcja funkcji heurystycznych

Jak w ogólnym przypadku skonstruować funkcję heurystyczną, gdy nie znamy dostatecznie dobrze zagadnienia, żeby ją po prostu wymyśleć?

Z definicji, wiedza heurystyczna pochodzi z doświadczenia. Zatem, aby ją pozyskać, konieczna jest praktyka: długi okres stażu pracy z dziedziną przedmiotową, zdobywanie doświadczeń (własnych i cudzych) z rozwiązywaniem różnorodnych instancji problemów z tej dziedziny.

A co może zrobić inżynier systemów sztucznej inteligencji, który nie ma takiego stażu pracy i doświadczeń, ale chciałby w krótkim czasie wypracować użyteczną heurystykę dla danego zagadnienia praktycznego?

Eksperymentować, eksperymentować, eksperymentować!

Konstrukcja heurystyk — metoda problemu uproszczonego

Jedna z ogólnych metod tworzenia funkcji heurystycznych jest następująca. Należy rozważyć zadanie uproszczone, w którym rezygnuje się z jakiegoś trudnego wymagania, aby zadanie dawało się rozwiązać. Dla każdego wygenerowanego stanu rozwiązuje się zadanie uproszczone (np. metodą pełnego przeglądu). Koszt optymalnego rozwiązania zadania uproszczonego przyjmuje się następnie jako oszacowanie (dolne) kosztu rozwiązania zadania oryginalnego.

> Na przykład, jeśli stany w zagadnieniu mają n parametrów, czyli są elementami n-wymiarowej przestrzeni, to możemy porzucić jeden z tych parametrów, czyli zrzutować stany do przestrzeni $(n-1)$-wymiarowej.

Jeśli istnieje kilka wersji uproszczenia, pomiędzy którymi nie wiemy jak wybrać (np. którą zmienna stanu odrzucić), to możemy użyć ich kombinacji jako funkcji oceny: $h(n) = \max_k(h_1(n), ..., h_k(n))$

Zauważmy, że gdybyśmy w układance 8-puzzle zezwolili na teleportację elementów jednym ruchem na swoje miejsce, to byłoby to przykładem takiego właśnie podejścia, i dałoby w efekcie funkcję $h_1(n)$. Natomiast zgoda na przesuwanie elementów o jedną pozycję, ale niezależnie od położenia innych elementów, dałaby funkcję oceny $h_2(n)$.

Konstrukcja heurystyk — wykorzystanie bazy danych wzorców

Rozważmy przykładową instancję problemu 8-puzzle na rysunku po lewej:

Zagadnienie przedstawione na rysunku po prawej możemy traktować jako **podproblem** tego zagadnienia, przy założeniu, że bloczki oznaczone gwiazdką są nierozróżnialne. Na pewno koszt rozwiązania tego podproblemu jest mniejszy niż koszt rozwiązania oryginalnego zadania. I ten koszt można wykorzystać jako heurystyczne oszacowanie oryginalnego problemu.

Metoda bazy wzorców polega na zbudowaniu bazy taki podproblemów i kosztów ich rozwiązania. Wielkość takiej bazy będzie zależała od stopnia uproszczenia podproblemu w odniesieniu do pełnego problemu. Dla 8-puzzle i bloczków: 1,2,3,4 jest $9 \times 8 \times 7 \times 6 \times 5 = 15120$ takich wzorców.

Ta metoda przypomina heurystykę Manhattanu, gdzie za każdym razem bloczek był przesuwany bez względu na położenie innych. Jednak w tym przypadku uproszczenie nie idzie tak daleko — część bloczków pozostaje i „przeszkadza” w przesuwaniu innych.

Bazę wzorców można zbudować w trakcie rozwiązywania problemów praktycznych, przy założeniu, że będzie ona wielokrotnie wykorzystywana w przyszłości. Każdą sekwencję rozwiązania można analizować od końca, i dla każdej kombinacji wybranych bloczków zapamiętywać długość końcowej sekwencji. Jest to przykład spamiętywania, czyli programowania dynamicznego.

Pytanie, czy taka baza wzorców ma sens właśnie dla bloczków 1-2-3-4 a nie innych? Ten wybór jest oczywiście arbitralny. Można zbudować bazę danych dla różnych kombinacji bloczków, a potem wykorzystać je łącznie, biorąc maksymalny koszt podproblemu dla danego pełnego problemu.

Może pojawić się dodatkowe pytanie: czy kosztów rozwiązania dla podproblemu 1-2-3-4 i 5-6-7-8 nie można dodać, aby uzyskać lepszą heurystykę dla pełnego problemu? Tu jednak odpowiedź brzmi: nie, ponieważ rozwiązania tych podproblemów nakładają się, i niektóre ruchy rozwiązania podproblemów 1-2-3-4 i 5-6-7-8 są współdzielone.

Jednak ma sens inny pomysł. Zamiast zamieniać część bloczków na gwiazdki, można je usunąć, i jako podproblem przesuwać tylko 1-2-3-4. Wtedy koszty rozwiązania takich podproblemów można sumować, i uzyskujemy jeszcze lepszą heurystykę.

Ten pomysł uogólnia się do metody bazy danych wzorców rozłącznych, dla których wykonywane w przestrzeni stanów operacje dotyczą tylko konkretnego podproblemu.

Konstrukcja heurystyk — modelowanie statystyczne

Inną metodą opracowania funkcji heurystycznej jest jej zamodelowanie statystyczne.

Należy wyznaczyć atrybuty stanu, które można uważać za znaczące dla oszacowania odległości do rozwiązania. Wtedy definiując funkcję heurystyczną jako kombinację liniową tych atrybutów, z nieznanymi współczynnikami, można nauczyć się tych współczynników wykonując pewną liczbę eksperymentów wykorzystujących pełne przeszukiwanie lub inną funkcję heurystyczną.

Otrzymane długości optymalnych rozwiązań można użyć do skonstruowania układu równań i w efekcie wyznaczenia przybliżonych wartości współczynników.

Zauważmy, że tą metodą możnaby otrzymać funkcję oceny $h_3(n)$ dla 8-puzzle. Funkcje $h_1(n)$ i $h_2(n)$ można uznać za przydatne do budowy dobrej heurystyki. Można też uznać, że funkcja $h_3(n)$ dobrze oddaje trudność osiągnięcia stanu docelowego. Stosując funkcję $h(n) = a * h_1(n) + b * h_2(n) + c * h_3(n)$ i przeprowadzając wiele eksperymentów obliczenia $h(n)$, jest możliwe, że optymalne wartości okazałyby się zbliżone do: $a \approx 0$, $b \approx 1$ i $c \approx 3$, co w efekcie dałoby funkcję $h_4(n) = h_2(n) + 3 * h_3(n)$.

Przeszukiwanie dla gier

Gry są fascynującą rozrywką i często stanowią wyzwanie dla intelektu człowieka. Nic dziwnego, że od dawna były obiektem zainteresowania sztucznej inteligencji.

Metody przeszukiwania w przestrzeni stanów nie dają się bezpośrednio zastosować w typowej grze dwu- lub wieloosobowej ze względu na konieczność uwzględnienia ruchów przeciwnika/ów, które nie są znane. Rozwiązaniem w takim przypadku musi być schemat uwzględniający wszystkie możliwe reakcje przeciwnika/przeciwników.

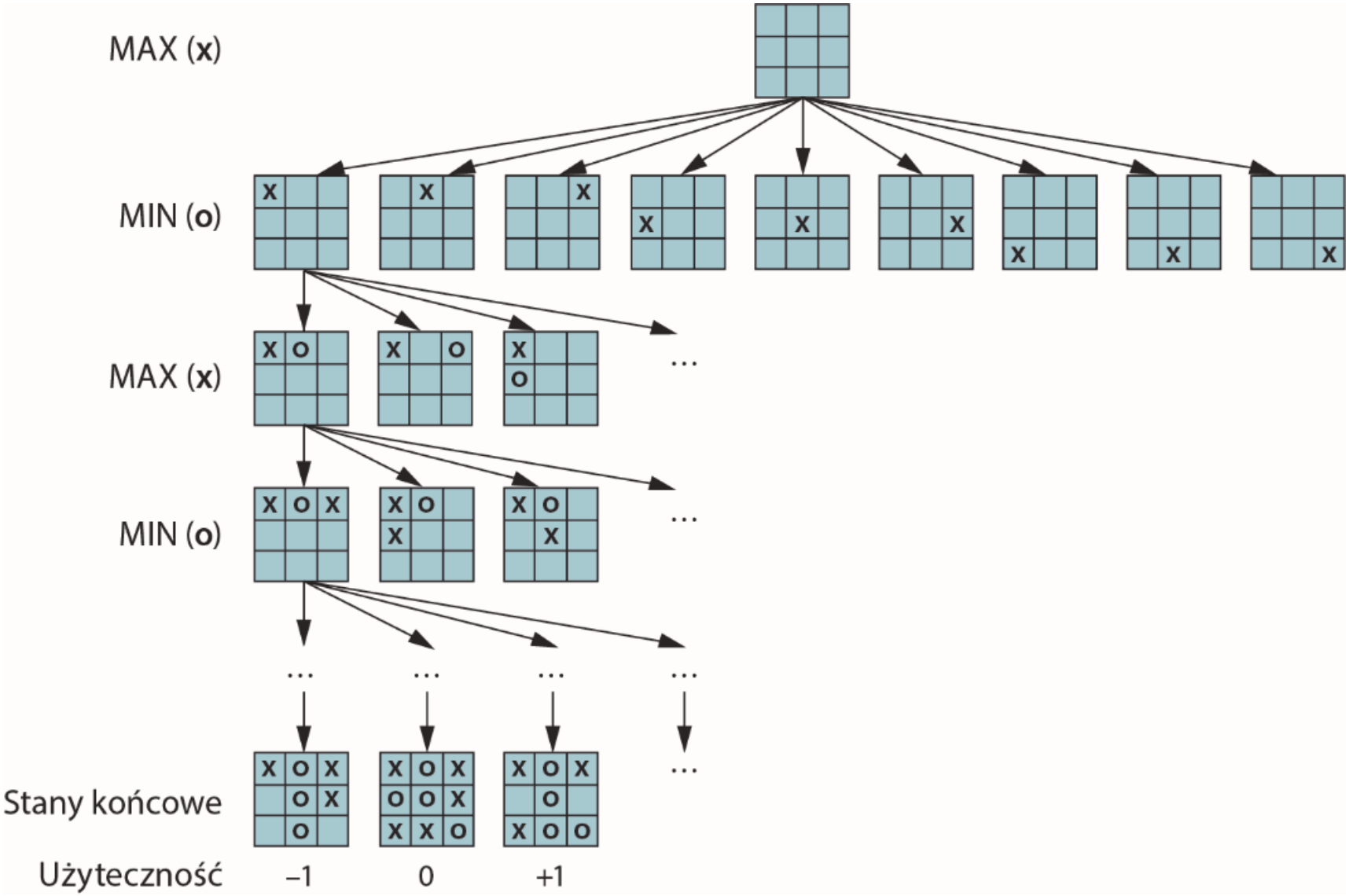
W części gier pełna wiedza o stanie w ogóle nie jest dostępna dla obu graczy. Ponadto, w niektórych grach występuje czynnik losowy (rozdanie kart, rzut kostką, itp.).

Rodzaje gier:

	deterministyczne	losowe
z pełną informacją	szachy, warcaby, go, othello	backgammon, monopol
z niepełną informacją	statki, kółko i krzyżyk na ślepo	brydż, poker, scrabble

Drzewo gry dwuosobowej

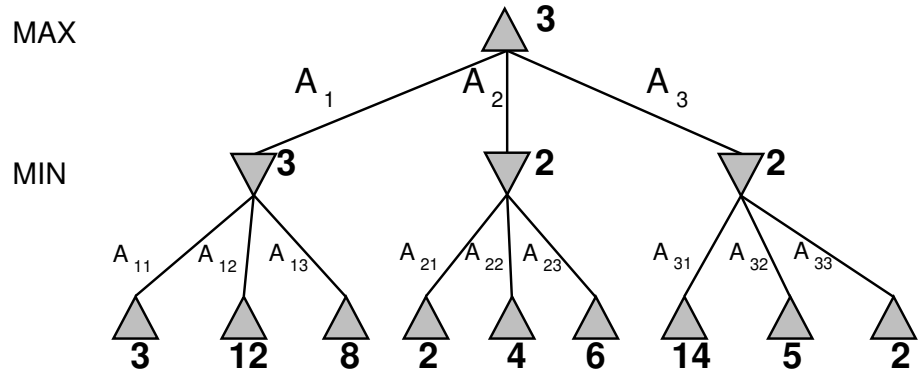
Na początek zajmiemy się grami dwuosobowymi, gdzie zadaniem jest wyznaczenie ruchu jednego z graczy, ale kolejny ruch należy do jego przeciwnika.



Procedura minimaksu

Można wyznaczyć optymalną strategię gry dla gry deterministycznej z pełną informacją za pomocą następującej procedury, zwanej procedurą **minimax**. Oblicza ona wartość węzła startowego przez propagację wartości końcowych (wartości wygranej dla naszego gracza) w górę drzewa gry:

1. poziomy drzewa odpowiadają ruchom graczy: MAX-a i MIN-a; przyjmujemy, że MAX ma pierwszy ruch,
2. stanom terminalnym w liściach drzewa przypisujemy wartość wygranej MAX'a (ujemną, jeśli faktycznie jest to jego przegrana)
3. węzłom drzewa powyżej liści przypisujemy stopniowo wartości: maksymalną ze wszystkich gałęzi jeśli węzeł odpowiada ruchowi MAX-a, i minimalna ze wszystkich gałęzi jeśli węzeł odpowiada ruchowi MIN-a,
4. najwyższa gałąź o największą wartością wskazuje optymalny ruch MAX-a.



Ograniczenie zasobów — zastosowanie heurystyki

Procedura minimaksu definiuje optymalną strategię gracza przy założeniu, że przeciwnik również gra optymalnie. Jednak tylko pod warunkiem, że da się przeanalizować całe drzewo gry.

> Dla prawdziwego drzewa gry może być z tym problem. Np., dla szachów $b \approx 35, m \approx 100$ dla sensownej rozgrywki, i kompletne drzewo gry może mieć około $35^{100} \approx 10^{155}$ węzłów.
> (Liczba atomów w znanej części Wszechświata szacowana jest na 10^{80}.)

Aby rozwiązać ten problem, można posłużyć się **heurystyczną funkcją oceny wartości pozycji**, aby podobnie jak w zwykłych metodach przeszukiwania przestrzeni stanów, wyznaczać dobry ruch bez posiadania jawnej reprezentacji całej przestrzeni. W przypadku gry dwuosobowej pozwoliłoby to zastosować tę samą zasadę minimaksu, ale ograniczyć analizę do kilku ruchów.

> Dla szachów, można taką ocenę obliczyć jako **wartość materialną** posiadanych figur, np. 1 za piona, 3 za wieżę lub gońca, 5 za skoczka, i 9 za hetmana. Dodatkowo można uwzględnić wartość pozycji takich jak „dobre rozstawienie pionów", albo wyższą wartość wieży w końcówce gry, a jeszcze wyższą dwóch wież.

Zastosowanie heurystyk — sytuacje specjalne

Ograniczenie głębokości analizy czasami prowadzi do sytuacji szczególnych, które wymagają nieco zmodyfikowanego podejścia.

Jedną z nich jest zagadnienie **opanowanego zagrożenia**. W niektórych sytuacjach funkcja oceny może sugerować wartości korzystne dla jednego z graczy, ale najbliższe ruchy — poza głębokością uwzględnioną przez funkcję przeszukiwania — nieuchronnie doprowadza do drastycznej zmiany. Rozwiązaniem jest wykrywanie takich niestabilnych sytuacji zagrożenia i pogłębienie przeszukiwania aż do osiągnięcia stanów bardziej stabilnych (*quiescent states*).

Innym problemem jest **problem horyzontu**. Ma on miejsce wtedy, gdy nadchodzi nieuchronne zagrożenie dla jednego z graczy, ale jest on w stanie odsuwać je w czasie wykonując ruchy, które jednak nie rozwiązują problemu.

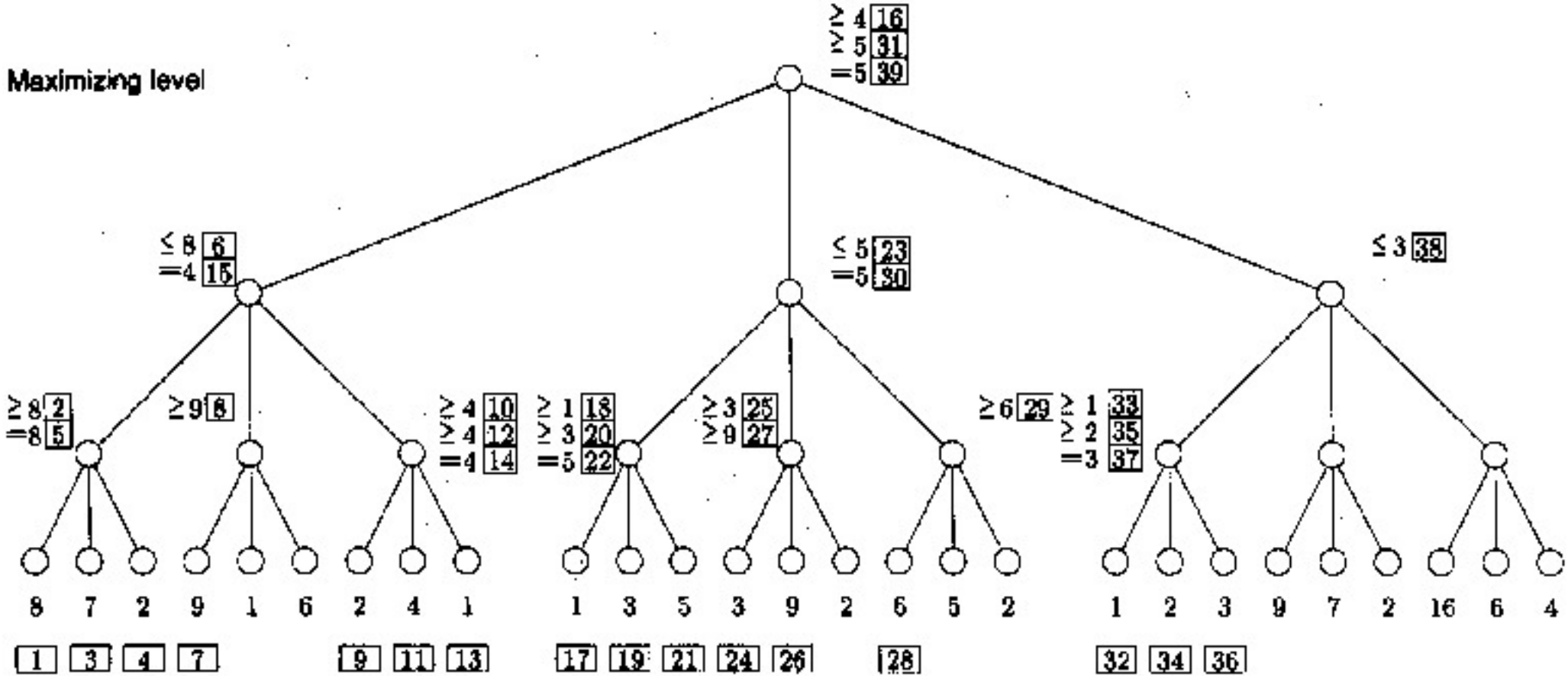
Przeszukiwanie minimax — odcinanie przeszukiwania

Na jakie efekty praktyczne można liczyć stosując ocenę heurystyczną na głębokości kilku ruchów?

Np., dla szachów, przyjmując 10^6 węzłów na sekundę, i 180 sekund na ruch, możemy zbadać $10^8 \approx 35^5$ pozycji, czyli do 5 ruchów wprzód. Program grający w ten sposób zachowuje się racjonalnie, lecz przeciętnemu człowiekowi nietrudno z nim wygrać. Potrzebne są dodatkowe metody zwiększające efektywność przeszukiwania.

Łatwo zauważyć, że można w analizie minimaksowej drzewa gry poczynić pewne oszczędności. Najprostsze z nich nazywane są **odcięciami alfa-beta**.

Odcięcia α–β — przykład



Ćwiczenie: znajdź błąd w powyższym drzewie (źródło: Patrick Henry Winston, *Artificial Intelligence*, 3rd ed.).

Odpowiedź: w kroku 10

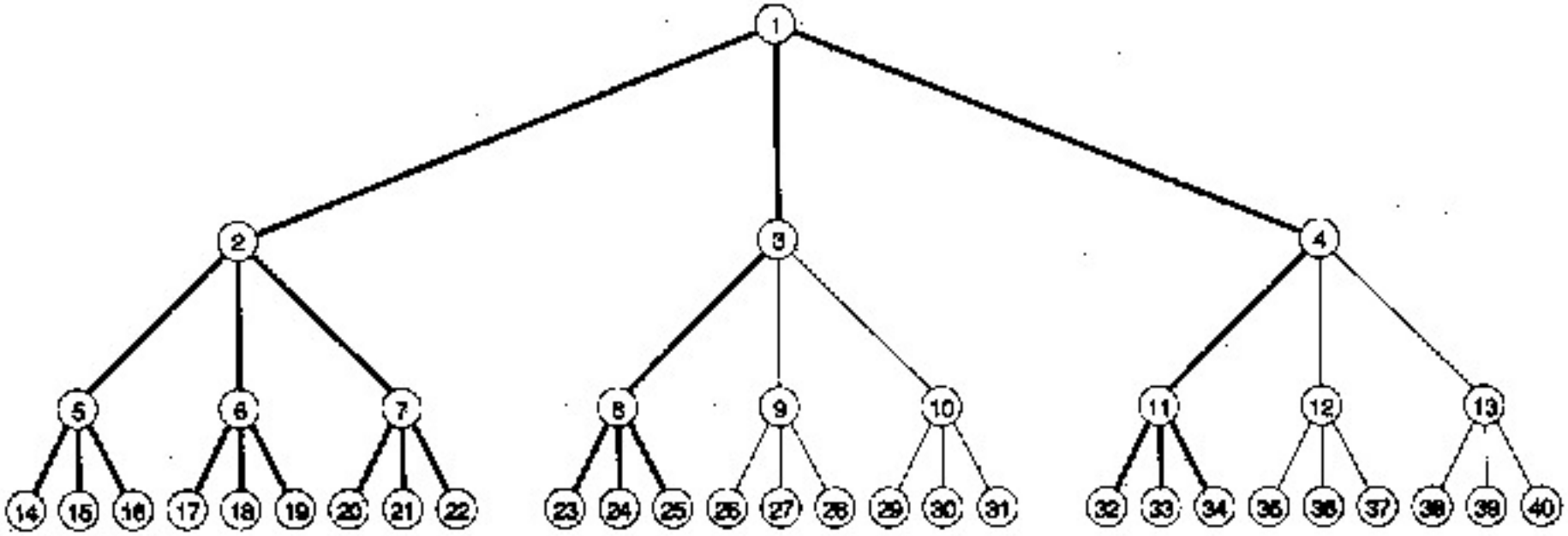
Odcięcia α–β — algorytm

```
PROCEDURE MINIMAX-ALPHA-BETA(n,alpha,beta,depth)
BEGIN
   IF depth==MAXDEPTH THEN RETURN(h(n))

   choices := Lista_potomkow(n)
   WHILE (NOT Empty(choices)) AND (alpha < beta) DO
   ;; zaniechanie badania kolejnych potomkow wezla n oznacza odciecie
   BEGIN
      n1 := First(choices)
      choices := Rest(choices)
      w1 := MINIMAX-ALPHA-BETA(n1,alpha,beta,depth+1)

      IF EVEN(depth) THEN                ; dla wezlow MAX'a
         IF w1 > alpha THEN alpha := w1
      IF ODD(depth) THEN                 ; dla wezlow MIN'a
         IF w1 < beta THEN beta := w1
   END

   IF EVEN(depth) THEN RETURN(alpha)     ; wezel MAX'a
                  ELSE RETURN(beta)      ; wezel MIN'a
END
```

$\Rightarrow$ w pierwszym wywołaniu przyjmujemy $\alpha = -\infty$, $\beta = +\infty$

Ocięcia α–β — przypadek optymalny

Optymalny przypadek przeszukiwania minimaksowego z odcięciami alfa-beta zachodzi gdy na każdym poziomie węzły są rozpatrywane w kolejności od najbardziej korzystnego, dla danego gracza. Wtedy w każdym poddrzewie obliczana jest tylko jedna „seria" węzłów, natomiast przy każdym powrocie w górę drzewa następuje odcięcie.



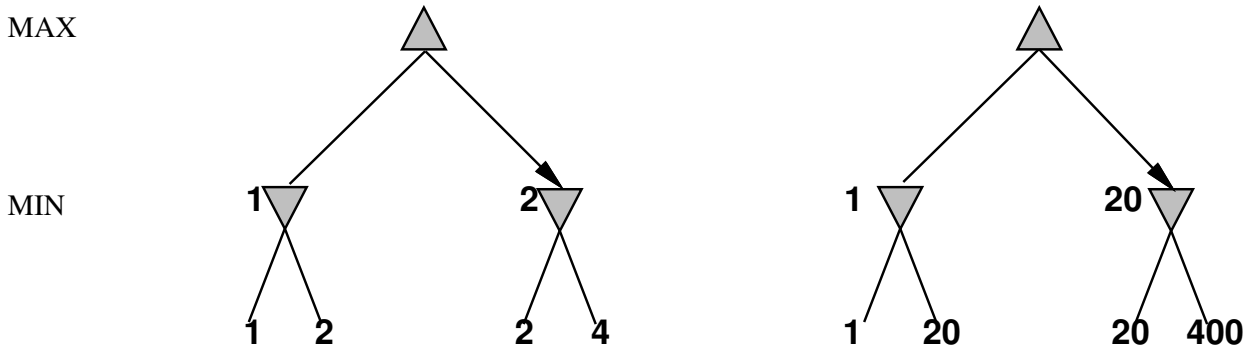
Na powyższym diagramie oszczędność wynosi 16 węzłów; na 27 węzłów na najniższym poziomie obliczonych musi być tylko 11.
Źródło: Patrick Henry Winston, *Artificial Intelligence*, 3rd ed. (uwaga, błąd: węzły 18, 19, 21, i 22 mogłyby również być odcięte).

Własności algorytmu α–β

Zasadniczą ideą algorytmu α–β jest że odcięcia w analizie drzewa nie zmieniają ostatecznego wyniku, tzn. ruchu gracza.

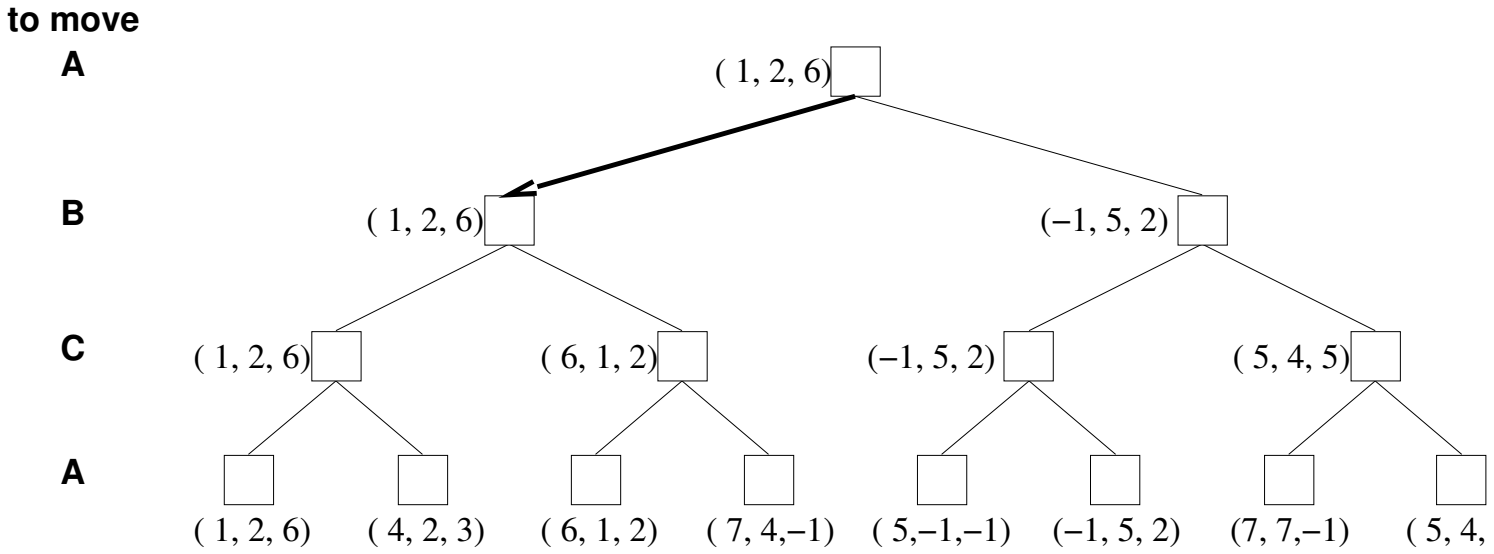
Dobre uporządkowanie pozwala osiągnąć większą efektywność odcinania. W granicy, optymalne odcięcia pozwalają osiągnąć $O(b^{m/2})$.
W praktyce pozwala to podwoić głębokość przeszukiwania.

Zauważmy, że wynik analizy minimax/α–β nie zależy od konkretnych wartości funkcji oceny. Istotne jest tylko uporządkowanie wartości. Oznacza to, że dowolna transformacja monotoniczna funkcji oceny działa tak samo jak oryginalna funkcja.



Minimax — uogólnienie dla gry wieloosobowej

Algorytm minimaksu można łatwo uogólnić na przypadek gry wieloosobowej. Zamiast skalarnej należy zastosować wektorową funkcję oceny, która zwraca wektor ocen wartości pozycji z punktu widzenia poszczególnych graczy. Każdy gracz maksymalizuje swój element wektora, i procedura propagacji oceny w górę drzewa gry przebiega podobnie jak w przypadku gry dwuosobowej.



W grach wieloosobowych pojawiają się jednak dodatkowe elementy strategii takie jak sojusze. Czasami graczom opłaca się zawierać sojusze przeciw innym graczom, a nawet dynamicznie zmieniać te sojusze w trakcie rozgrywki.

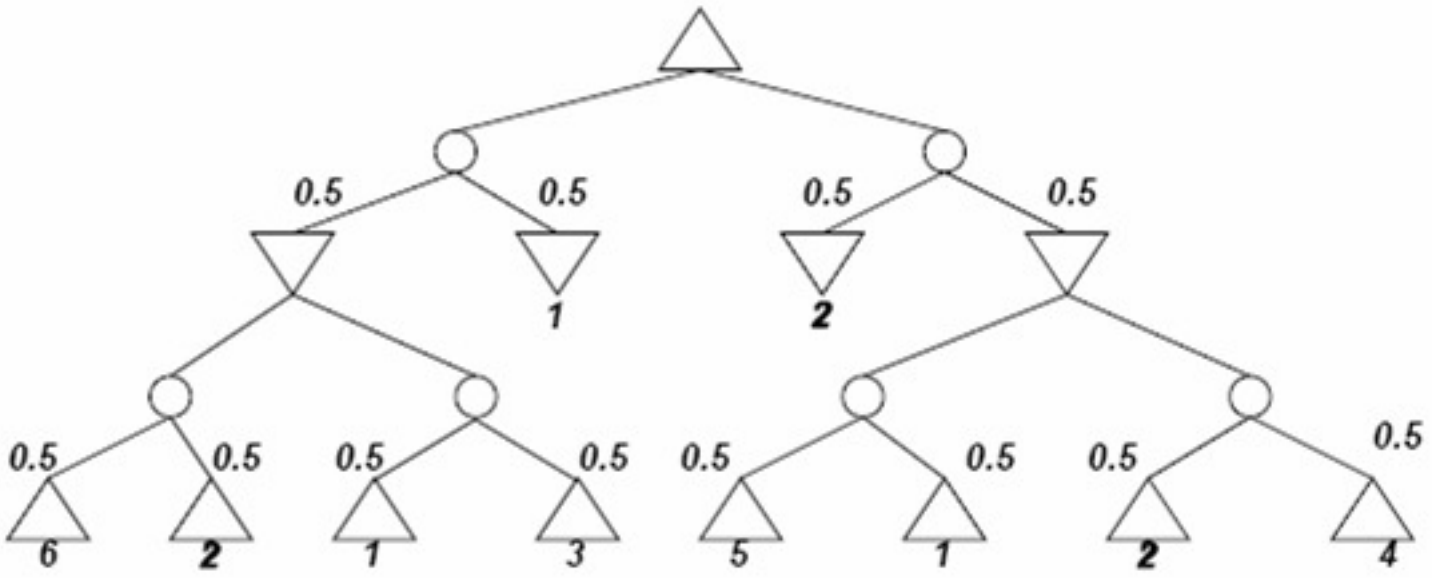
Gry z elementami losowymi — expectimax

Jeżeli w grze występuje czynnik losowy, np. wynik pewnych ruchów zależy od elementu losowego, jak rzut kostką, to analiza takiej gry jest bardziej złożona. Wymaga rozważenia wszystkich możliwości, i obliczania wartości oczekiwanej po dystrybucji zmiennych losowych reprezentujących elementy losowe.

Na przykład, można rozważać gry jednoosobowe z czynnikiem losowym. Co drugi krok jest wyborem gracza, który maksymalizuje wartość swojej funkcji oceny, a co drugi krok jest losowy (lub traktujemy go jako losowy), ze znanym zestawem wyników i znanym rozkładem prawdopodobieństwa tych wyników. Algorytm dostosowany do analizy takich gier nazywa się **expectimax**.

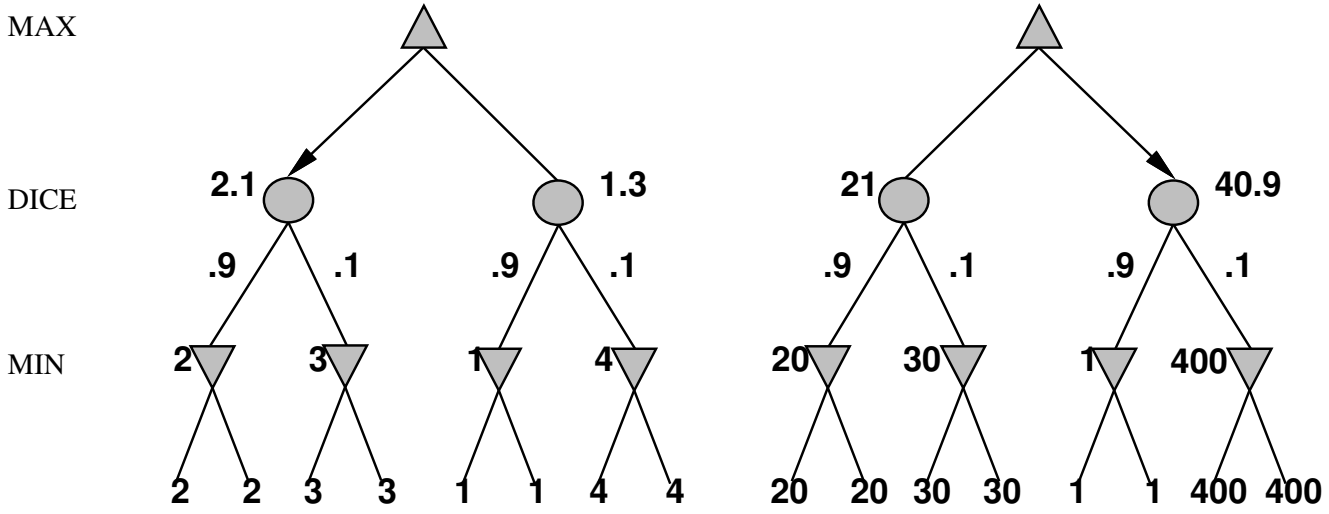
Dalsze uogólnienie — expectiminimax

Pełne uogólnienie algorytmu minimaksu o czynniki losowe uwzględnia ruchy graczy MAXa i MINa oraz kroki losowe. Algorytm analizy drzewa takiej gry nazywa się **expectiminimax**.



Heurystyczna funkcja oceny w expectiminimax

Zauważmy różnicę własności w odniesieniu do stosowanej funkcji oceny. Wybór ruchu na podstawie analizy minimax jest identyczny dla wszystkich funkcji oceny dającej ten sam porządek węzłów. Tej własności nie zachowuje expectiminimax, jak widać na poniższym rysunku. Ruch wybrany dla przedstawionych drzew jest różny, a bez kroku losowego za każdym razem wybrany byłby prawy.



W przypadku expectiminimaksu funkcja oceny nie może być dowolną funkcją poprawnie porządkującą oceniane pozycje. W praktyce musi ona odzwierciedlać oczekiwaną wygraną (lub jej dodatnią liniową transformację).

Gry z niepełną informacją

Przykład: różne gry w karty.

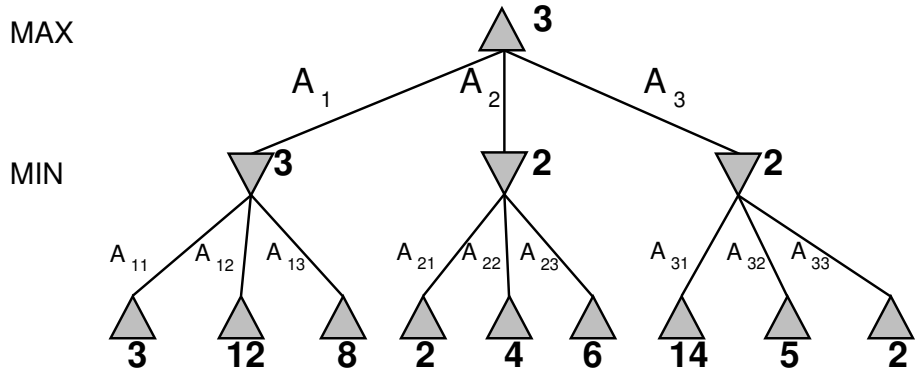
Można obliczyć prawdopodobieństwa wszystkich kombinacji rozdania.

Idea: oblicz wartość minimax każdej możliwej akcji dla każdego rozdania, następnie wybierz akcję z największą wartością oczekiwaną obliczoną dla wszystkich możliwych rozdań.

Najlepsze programy do gry w brydża implementują to podejście przez wygenerowanie wielu rozdań zgodnych z informacją wywnioskowaną z dotychczasowego przebiegu licytacji i rozgrywki, i wybierają akcję, która maksymalizuję liczbę wygranych.

Krótkie podsumowanie — pytania sprawdzające

1. Dla poniższego drzewa gry dwuosobowej, podaj dokładną sekwencję wartości funkcji oceny obliczonych przez algorytm minimaksu z odcięciami alfa/beta (w porządku od lewej do prawej).



Literatura i materiały pomocnicze

1. Stuart Russell, Peter Norvig: Sztuczna inteligencja. Nowe spojrzenie, Wydanie IV, Tom 1, Wyd.Helion, 2023, rozdział 3 i 5.