Motywacja

Jesteśmy ekspertem/doradcą w jednej z agencji rządowych, i w niedzielę wieczorem prezes agencji dzwoni do nas na komórkę, żeby poinformować, że w poniedziałek rano o godzinie 10-tej odbędzie się ważna narada w Ministerstwie Infrastruktury, i konieczna jest nasza niezawodna obecność.

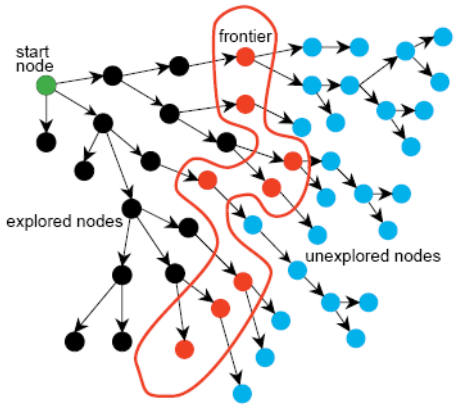
Co robić?

Mamy wiele możliwości, możemy wrócić do kolacji z rodziną, możemy pójść na spacer żeby się odstresować.

Wiemy jednak, że musimy zaplanować podróż do Warszawy. Samolot nie jest dobrą opcją. W tanich liniach lotniczych wszystkie miejsca są dawno wykupione, a w LOT są pewnie jeszcze miejsca, ale jak rano będzie mgła i samolot nie odleci, to zostawimy szefa na lodzie, i potem pewnie możemy się już w ogóle nie pokazywać w agencji (odprawę przyślą nam na konto).

Co gorsza, PKP w dzisiejszych czasach również ma zwyczaj odwoływać pociągi z dnia na dzień, i nie ma pewności, czy możemy liczyć na nocny pociąg 1:32 (jest w Warszawie o 7:32). Możemy sprawdzić tę opcję, ale może się okazać, że czeka nas jazda samochodem.

Ogólnie, jest szereg możliwości, każda z nich wymaga starannego rozważenia. Prowadzi to do **przeszukiwania**.



Przeszukiwanie jest elementem składowym wszystkich metod sztucznej inteligencji, i zdolność skutecznego przeszukiwania w ogóle zdaje się być inherentnym elementem inteligencji.

Reprezentacja problemu w przestrzeni stanów

1. przestrzeń stanów
   - może mieć postać iloczynu kartezjańskiego dziedzin parametrów opisu
   - przestrzeń może być skończona lub nieskończona, choć nie musi to być związane z trudnością problemu (np. szachy)
   - czasem część całej formalnie zdefiniowanej przestrzeni stanowią stany niedozwolone (inaczej: nieosiągalne)
2. stan początkowy, zawsze jawnie podany
3. stan docelowy, jawny lub niejawny (warunek osiągnięcia celu)
4. dostępne operatory przejścia od stanu do stanu, inaczej: funkcja następnika, *successor function*
   - np. w postaci warunków stosowalności i efektów działania
   - operator może być sparametryzowany (np. w labiryncie możemy mieć jeden operator ruchu, cztery operatory, albo liczbę miejsc razy cztery)

⇒ Zadaniem jest wyznaczenie sekwencji operatorów prowadzących ze stanu początkowego do celowego.

Ogólny schemat przeszukiwania w przestrzeni stanów

```
PROCEDURE GT(St)                ; St - opis stanu poczatkowego
BEGIN
UNTIL Term(St) DO               ; stan St spelnia warunek celu
   BEGIN
      Op := first(ApplOps(St)) ; wybierz operator stosowalny w stanie St
      St := Apply(Op, St)      ; rezultat zastosowania Op do stanu St
   END
END
```

Co prawda powyższy zapis algorytmu GT (*Generate-and-Test*) sugeruje, że wybiera on pierwszy możliwy do zastosowania w stanie St operator, jednak algorytm ma wpływ na ten wybór operatora przez odpowiednie posortowanie listy operatorów. Metodę wyboru operatora przez algorytm przeszukiwania nazywamy **strategią**.

Zastosowanie dobrej strategii jest w algorytmach przeszukiwania zagadnieniem kluczowym.

Strategie ślepe i poinformowane

Strategia może być całkowicie ogólna, bazująca tylko na syntaktycznych własnościach reprezentacji zagadnienia, i dająca się wykorzystać we wszystkich możliwych przypadkach. Takie strategie nazywa się **ślepymi**.

Przykład: całkiem użyteczną ślepą (i to dosłownie) strategią w przeszukiwaniu labiryntów jest strategia prawej ręki. Strategia ta pozwala znaleźć wyjście z labiryntu, jeśli tylko takowe istnieje.

Strategie mogą również wykorzystywać informacje o stanie, specyficzne dla danej dziedziny problemowej. Takie strategie nazywamy **poinformowanymi**.

Strategie poinformowane korzystają z informacji, które w ogólnym przypadku nie są dostępne, i mogą być niezrozumiałe dla osoby postronnej, oraz dla całkowicie ogólnego algorytmu przeszukiwania.

Przykład: wyobraźmy sobie, że poszukując wyjścia z labiryntu wiemy, że na zewnątrz jest hałas (np. szum morza), a w labiryncie całkowita cisza. Wtedy zwyczajne nadsłuchiwanie we wszystkich kierunkach mogłoby być źródłem strategii poinformowanej, pomagając w wyborze właściwych kroków (choć strategia ta może być skuteczna tylko w pewnej niewielkiej odległości od wyjścia).

Krótkie podsumowanie — pytania sprawdzające

1. Z czego składa się reprezentacja problemu w przestrzeni stanów?

2. Co to są ślepe i poinformowane strategie przeszukiwania? Czym się różnią?

Przeszukiwanie z nawracaniem (BT)

```
FUNCTION BT(st)
BEGIN
   IF Term(st)          THEN RETURN(NIL)   ; trywialne rozwiazanie
   IF DeadEnd(st)       THEN RETURN(FAIL)  ; brak rozwiazania
   ops := ApplOps(st)                      ; lista oper.stosowalnych
L: IF null(ops)         THEN RETURN(FAIL)  ; brak rozwiazania
   o1 := first(ops)
   ops := rest(ops)
   st2 := Apply(o1,st)
   path := BT(st2)
   IF path == FAIL      THEN GOTO L
   RETURN(push(o1,path))
END
```

Algorytm BT skutecznie przeszukuje przestrzeń rozwiązań bez jawnego budowania drzewa przeszukiwania przestrzeni. Struktury jakich używa do zapamiętania stanu przeszukiwań są niejawne (na stosie). Można skonstruować iteracyjną wersję tego algorytmu, która buduje te struktury jawnie.

Przeszukiwanie z nawracaniem — własności

BT ma minimalne wymagania pamięciowe. W trakcie pracy pamięta tylko pojedynczą ścieżkę do rozwiązania (oraz pewien kontekst dla każdego elementu tej ścieżki). Zatem jego **złożoność pamięciowa przypadku średniego wynosi** $O(d)$, gdzie d - odległość stanu początkowego od rozwiązania (w sensie liczby operatorów).

Efektywność czasowa jest gorsza. W najgorszym przypadku algorytm BT **może odwiedzić wszystkie stany przestrzeni** przed znalezieniem rozwiązania. Pozwala jednak na użycie strategii — poinformowanej lub ślepej — w momencie tworzenia listy operatorów, przez jej odpowiednie posortowanie.

Poważnym problemem algorytmu BT jest fakt, że **może on nie znaleźć rozwiązania**, nawet jeśli istnieje ono w niewielkiej odległości od stanu startowego. Jeśli np. przestrzeń stanów jest nieskończona, algorytm może w pewnym momencie przeszukiwania wybrać operator prowadzący do stanu, z którego prowadzą drogi do nieskończonej liczby stanów, ale żaden z nich nie jest stanem docelowym. W takim przypadku algorytm BT nigdy nie zakończy przeszukiwania tej części przestrzeni stanów, i nigdy nie będzie mógł wycofać się z niewłaściwego wyboru operatora.

Wykrywanie powtarzających się stanów

Jednym z problemów algorytmu BT — 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 BT 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 jest **ograniczenie głębokości** przeszukiwania do jakiejś „rozsądnej" wartości. Zauważmy, że poza zabezpieczeniem przed nieskończonymi przestrzeniami, zabezpiecza ono jednocześnie przed wpadnięciem w pętle, 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 jak BT optymistycznych (preferujących ruchy wgłąb) stosuje się zatem **ograniczenie głębokości z iteracyjnym pogłębianiem**. Ten wariant gwarantuje znalezienie rozwiązania, o ile istnieje.

Jednak w przypadku algorytmu BT ta metoda może być bardzo nieefektywna.

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.

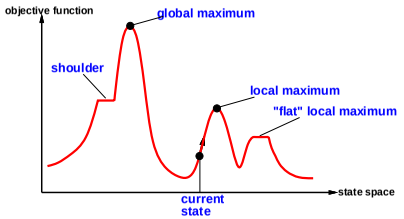
Posiadanie heurystyki pozwala budować strategie poinformowane. Ogólnym i często stosowanym schematem konstrukcji strategii wykorzystującym informację heurystyczną, jest **statyczna funkcja oceny stanu**. Dla każdego stanu określa ona jego „dobroć", czyli szanse, że przez ten stan prowadzi droga do rozwiązania.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

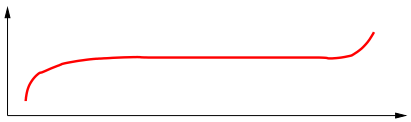
Metody gradientowe

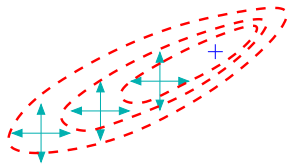
Funkcję oceny stanu można w przeszukiwaniu zastosować bezpośrednio. Prowadzi to do metody lub metod **gradientowych** (*hill-climbing*). Metody te określa się w informatyce jako metody zachłanne.

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

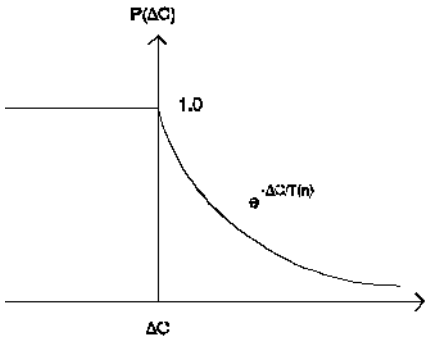






Wyżarzanie

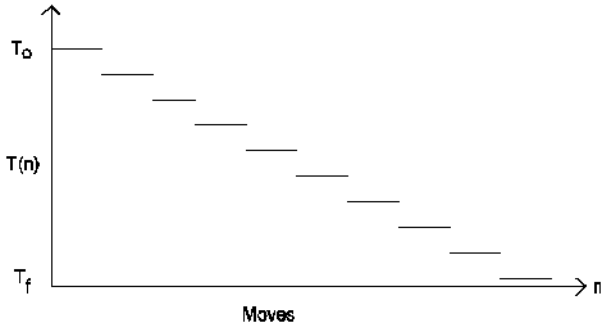
Skuteczną i często stosowaną grupę metod gradientowych stanowi technika zwana wyżarzaniem (*simulated annealing*). Jej nazwa odwołuje się do analogii z procesem wytapiania metalu, kiedy stopniowe i powolne zmniejszanie temperatury pozwala osiągnąć stan globalnego optimum energetycznego, z pełnym uporządkowaniem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

Krótkie podsumowanie — pytania sprawdzające

1. Jakie wymagania algorytmu BT są bardziej krytyczne (istotne, ograniczające): pamięciowe czy czasowe? Uzasadnij odpowiedź.

2. W jakich sytuacjach algorytm BT może nie znaleźć rozwiązania, gdy ono istnieje?

3. Na czym polega zjawisko powtarzających się stanów w algorytmach przeszukiwania? Jakie są jego możliwe konsekwencje?

4. Jaki problem rozwiązuje metoda iteracyjnego pogłębiania? W jakich przypadkach konieczne jest jej stosowanie?

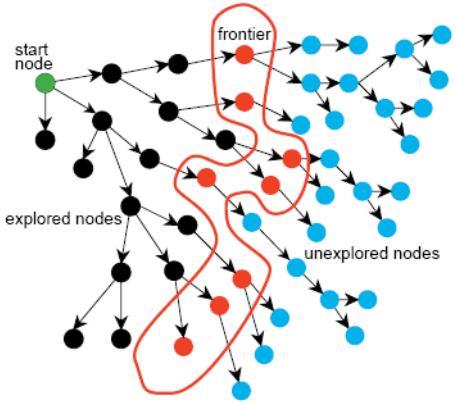
5. Jakie są główne problemy jakościowe (nie uwzględniając złożoności) w zastosowaniu gradientowych metod przeszukiwania?

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**.

Ogólne strategie przeszukiwania grafów (ślepe):

- strategia wszerz BFS (*breadth-first search*)
- strategia wgłąb DFS (*depth-first search*),
- inne strategie.

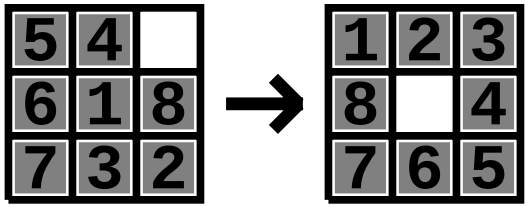


Przykład: 8-ka (8-puzzle)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

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).
- Nie jest inherentnie odporny na wpadanie w pętle stanów i może wymagać zastosowania listy *Closed*.
- 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).
- Uwaga implementacyjna: dodawaj nowo odkryte stany na koniec listy *Open*. (Pomimo iż mówi się o listach węzłów, ze względu na częste odwołania w praktyce stosuje się szybsze struktury danych, np. tablice haszowe.)

Przeszukiwanie wszerz — przykład

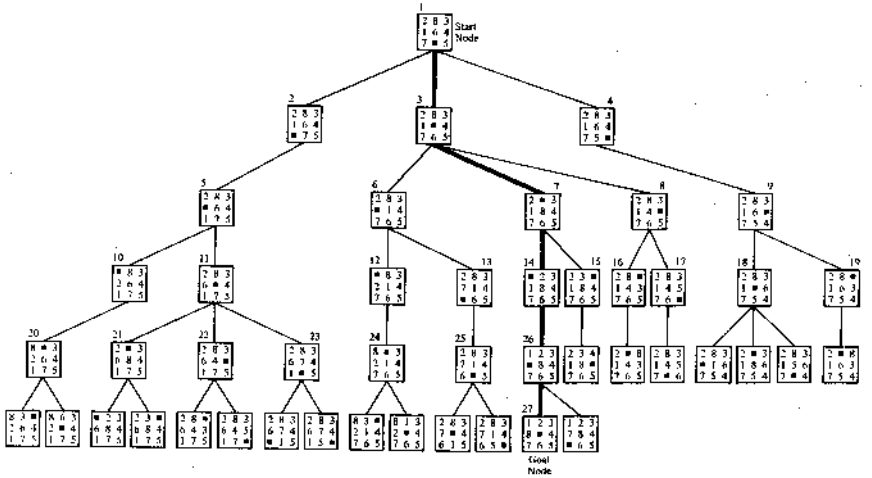
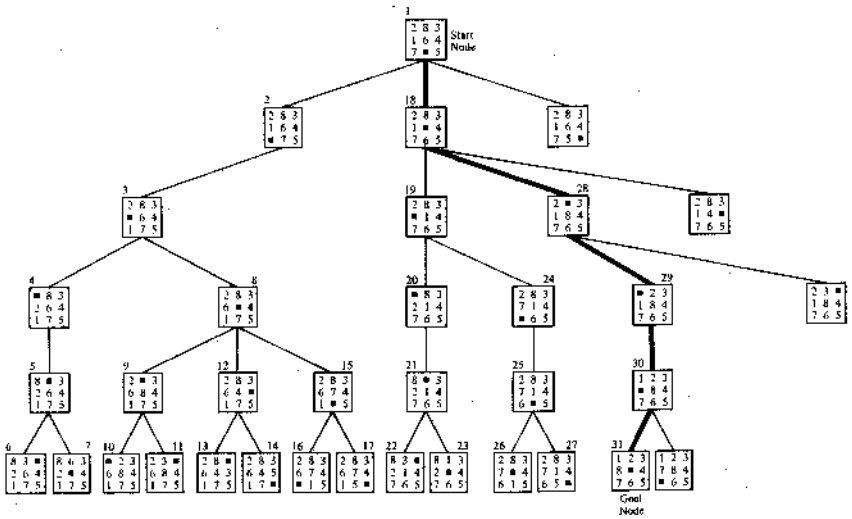


Diagram przedstawia fragment grafu przeszukiwania wszerz. Numery nad planszami (1–26) pokazują tu kolejność wyboru węzłów do ekspansji grafu.

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 jedynym praktycznie użytecznym wariantem jest ograniczenie głębokości z iteracyjnym pogłębianiem (ale przeszukiwanie grafu DFS nie jest aż tak bezsensownie stratne jak algorytm BT).
- 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.
- Uwaga implementacyjna: dodawaj nowo odkryte stany na początek listy *Open*.

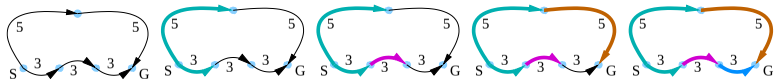
Przeszukiwanie wgłąb — przykład



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

Przeszukiwanie równokosztowe UCS

W przypadku, gdy koszty pojedynczych ruchów nie są równe, przeszukiwanie wszerz oparte na liczbie ruchów w oczywisty sposób nie gwarantuje znalezienia optymalnej ścieżki. Można określić prostą modyfikację algorytmu wszerz, która znajdzie optymalną ścieżkę dla dowolnych (dodatnich) kosztów pojedynczych ruchów. Ta modyfikacja, zwana algorytmem **równego kosztu** (*uniform-cost search* UCS), wymaga każdorazowo wybrania węzła o najniższym koszcie ścieżki.



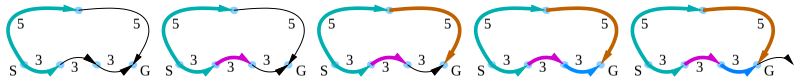
W przypadku równych kosztów ruchów sprowadza się to do metody wszerz.

Optymalność algorytmu można (trywialnie) wykazać pod warunkiem, że koszt pojedynczego ruchu jest jakąś wartością dodatnią ($\geq \epsilon$). Ponieważ algorytm kieruje się długością ścieżki, jego złożoności nie można scharakteryzować jako funkcji b i d. Zamiast tego, oznaczając przez C^* koszt optymalnego rozwiązania, można otrzymać złożoność najgorszego przypadku, zarówno czasową jak i pamięciową, jako $O(b^{1+\lfloor C^*/\epsilon \rfloor})$.

W przypadku równych kosztów formuła ta redukuje się do $O(b^d)$.

Zakończenie przeszukiwania

Celem przeszukiwania może być samo znalezienie ścieżki do rozwiązania, bądź znalezienie ścieżki optymalnej. W pierwszym przypadku, algorytm może zakończyć pracę już w momencie, kiedy nowy stan, wygenerowany w wyniku kolejnego ruchu, okaże się stanem docelowym, a więc zostanie umieszczony na liście *Open*. Ale czy tak samo możemy postąpić w przypadku poszukiwania rozwiązania optymalnego?



Przeszukiwanie należy zakończyć w momencie, gdy algorytm przeszukiwania optymalnego wybierze do ekspansji węzeł, który jest węzłem docelowym (czyli już wcześniej znalazł jeden lub kilka węzłów docelowych). Jego ekspansji można wtedy zaniechać, a najlepsza znaleziona do niego ścieżka jest rozwiązaniem optymalnym. Ponieważ algorytm systematycznie znajduje wszystkie najtańsze ścieżki, więc moment wybrania węzła do ekspansji oznacza, że nie może się już w grafie znaleźć żadna tańsza ścieżka do niego.

Jednak zanim to nastąpi, algorytm bada ścieżki o niższych kosztach, i nie ma pewności, że któraś z nich nie okaże się nową, lepszą ścieżką do węzła celowego.

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.

Algorytmy przeszukiwania grafu

Rozważane tu algorytmy przeszukiwania grafów działają według schematu:

```
PROCEDURE GS(s0)                ; s0 - opis stanu poczatkowego
BEGIN
n  := s0
G  := {s0}                      ; graf przeszukiwania: wezly i luki
Open := [s0]; Closed := []      ; listy: nowych i sprawdzonych
UNTIL Term(n) DO                ; stan n spelnia warunek celu
   BEGIN
      Open := Remove(n,Open)
      new := Successors(n)      ; wygeneruj nastepniki wezla n
      G := AddSuccessors(G,New) ; dodaj strukture do grafu G
      Open += (new-Closed)      ; dodaj nowo odkryte nastepniki
      Closed += {n}
      n := SelectNext(Open)     ; wybierz wezel do eksploracji
   END
solution := BuildPath(s0,n,G)   ; zrekonstruuj sciezke do celu
END
```

W powyższym algorytmie dla uproszczenia pominięto operacje zapamiętania i korekty kosztów ścieżek od węzła startowego do wszystkich węzłów.

Ślepe algorytmy BFS i DFS ignorują koszty i dodają nowe węzły na koniec (BFS) lub początek (DFS) listy *Open*, która jest w tym przypadku zwykłą kolejką.

Algorytmy równokosztowy i najpierw-najlepszy wybierają kolejny węzeł do ekspansji według kryterium kosztu (odpowiednio: długości ścieżki początkowej lub oszacowania odległości od rozwiązania). Wtedy ma sens implementacja listy *Open* jako listy sortowanej, albo jeszcze lepiej, jako kolejki priorytetowej, dającej natychmiastowy wybór najlepszego węzła i tanie operacje dodawania i usuwania z listy $O(\log(N))$.

Warto tu dodać, że istnieje algorytm Dijkstry (1959) znajdowania najkrótszych dróg na grafie z jednego węzła do wszystkich węzłów grafu. Jednak Dijkstra zakładał operacje na grafie skończonym, w całości znanym, zbudowanym, i załadowanym do pamięci.

Krótkie podsumowanie — pytania sprawdzające

1. Czym różni się przeszukiwanie równokosztowe od przeszukiwania wszerz?
2. Czym różni się przeszukiwanie wgłąb od przeszukiwania najpierw-najlepszy?
3. Opisz główny cykl pracy algorytmu przeszukiwania grafów.
4. Opisz operacje na listach Open i Closed w różnych algorytmach przeszukiwania grafów.

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.

Przeszukiwanie wstecz

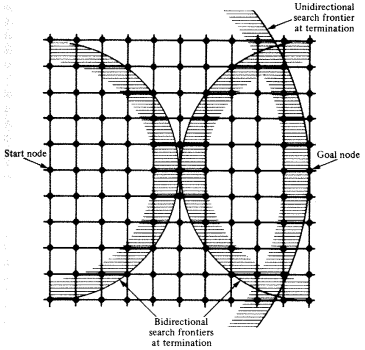
Przeszukiwanie przestrzeni stanów można prowadzić równie dobrze wprzód jak i wstecz. **Przeszukiwanie wstecz** zaczyna się od stanu końcowego (lub całego zbioru stanów końcowych), i w pierwszym kroku znajduje zbiór stanów poprzedzających, z których można osiągnąć jakiś stan końcowy w jednym kroku przez któryś z dostępnych operatorów. W kolejnych krokach proces jest kontynuowany.

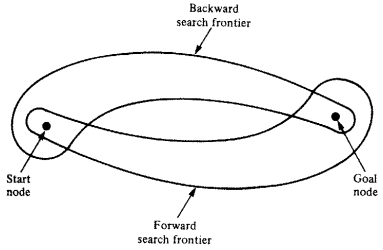
Przeszukiwanie wstecz może być równie łatwe w realizacji obliczeniowej jak przeszukiwanie wprzód, albo może być utrudnione ze względu na własności przyjętej reprezentacji. W tym drugim przypadku konieczna może być zmiana reprezentacji.

Kluczowa jest jednak łatwość pozyskania heurystyk. W przypadku przeszukiwania wprzód heurystyka powinna podpowiadać nam, jakie kroki należy wybierać, aby skutecznie przybliżać się do celu. W niektórych zagadnieniach brak jest właściwych intuicji. W przypadku przeszukiwania wstecz herustyka powinna podpowiadać, które kroki przybliżają nas od nieznanego stanu docelowego, do dobrze znanego środowiska startowego. Czasem łatwiej jest o intuicje wspomagające podejmowanie takich decyzji.

Przeszukiwanie dwukierunkowe

Ideę przeszukiwania wstecz można łatwo uogólnić do **przeszukiwania dwukierunkowego**. Jeśli reprezentacja na to pozwala, to dlaczego nie robić na przemian kroków przeszukiwania wprzód i wstecz. Jak widać na rysunku po lewej, mogłoby to przynieść oszczędności rzędu 50% ($2 \times \pi(\frac{r}{2})^2$ zamiast πr^2):





Jednak jak pokazuje rysunek po prawej, zamiast zaoszczędzić, można nadłożyć pracy. Przeszukiwanie dwukierunkowe łatwo przynosi oszczędności w przypadku algorytmu Dijkstry (równokosztowego), jednak posiadając wyrafinowaną, ukierunkowaną heurystykę, lepiej jej zaufać i podążać za nią w jednym kierunku.

Krótkie podsumowanie — pytania sprawdzające

1. Czym różni się algorytm A* od przeszukiwania najpierw-najlepszy? Jaki skutek wywiera ta różnica na proces przeszukiwania?

2. Co to są dopuszczalne heurystyki dla algorytmu A*? Jakie mają znaczenie praktyczne?

3. Algorytm heurystycznego przeszukiwania grafów A* z dopuszczalną funkcją oceny h gwarantuje znalezienie optymalnego rozwiązania problemu, o ile tylko takie istnieje na grafie. Rozważ poniższe modyfikacje funkcji f i odpowiedz, czy zachowują one powyższą własność algorytmu A*. Odpowiedź uzasadnij.

   (a) wprowadzenie górnego ograniczenia (kresu) na wartość funkcji h(n)
   (b) wprowadzenie dolnego ograniczenia (kresu) na wartość funkcji g(n)

Konstrukcja funkcji heurystycznych

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

Eksperymentować, eksperymentować, eksperymentować!

Przykład: heurystyki dla 8-puzzle

Heurystyka 1: policz elementy nie na swoich miejscach, funkcja $h_1(n) = W(n)$

Heurystyka 2: dla wszystkich elementów nie na swoich miejscach, zsumuj odległości od ich właściwych miejsc (tzw. odległość Manhattanu). Otrzymana liczba będzie na pewno mniejsza niż liczba ruchów w każdym rozwiązaniu (dolne oszacowanie kosztu rozwiązania). Nazwijmy ją funkcją $h_2(n) = P(n)$

Heurystyka 3: $h_3(n) = P(n) + 3 * S(n)$
gdzie funkcja $S(n)$ jest 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.

Ani $S(n)$ ani $h_3(n)$ nie są dolnymi oszacowaniami rzeczywistej odległości do rozwiązania układanki, a jednak $h_3(n)$ 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.

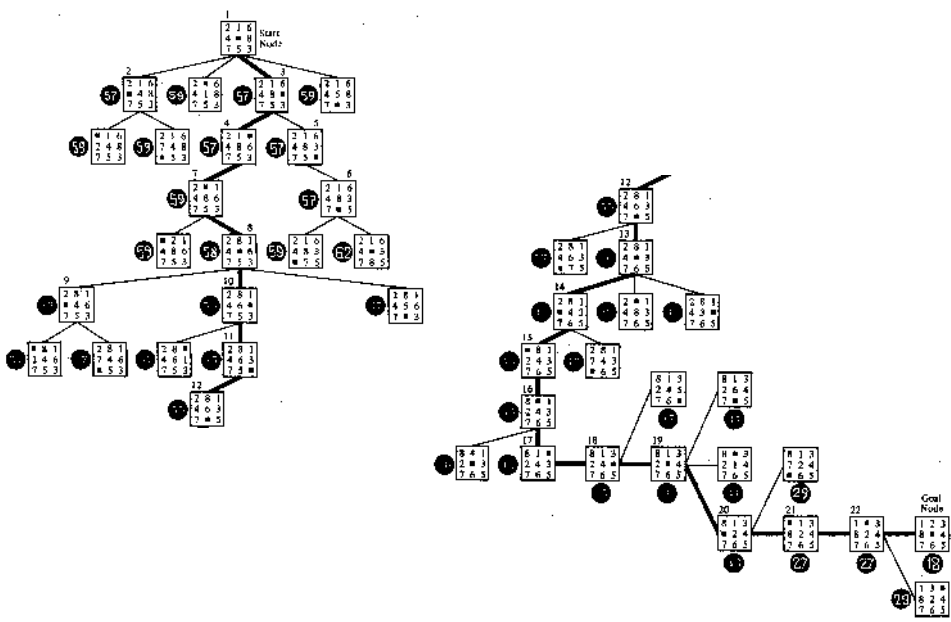
Jakość funkcji heurystycznej a koszt przeszukiwania A*

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Figure 3.29 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with h_1, h_2. Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths d.

Przybliżona liczba węzłów IDS dla d=24: 54,000,000,000

Przeszukiwanie heurystyczne drzewa 8-puzzle



Konstrukcja funkcji heurystycznych (cd.)

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 funkcji heurystycznych (cd.)

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 $W(n)$ i $P(n)$ można uznać za przydatne do budowy dobrej heurystyki. Można też uznać, że funkcja $S(n)$ dobrze oddaje trudność osiągnięcia stanu docelowego. Stosując funkcję $h(n) = a * W(n) + b * P(n) + c * S(n)$ 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 $h_3(n)$.

Krótkie podsumowanie — pytania sprawdzające

1. Wymień i opisz znane Ci ogólne metody tworzenia heurystycznych funkcji oceny.

Przeszukiwanie dla gier dwuosobowych

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

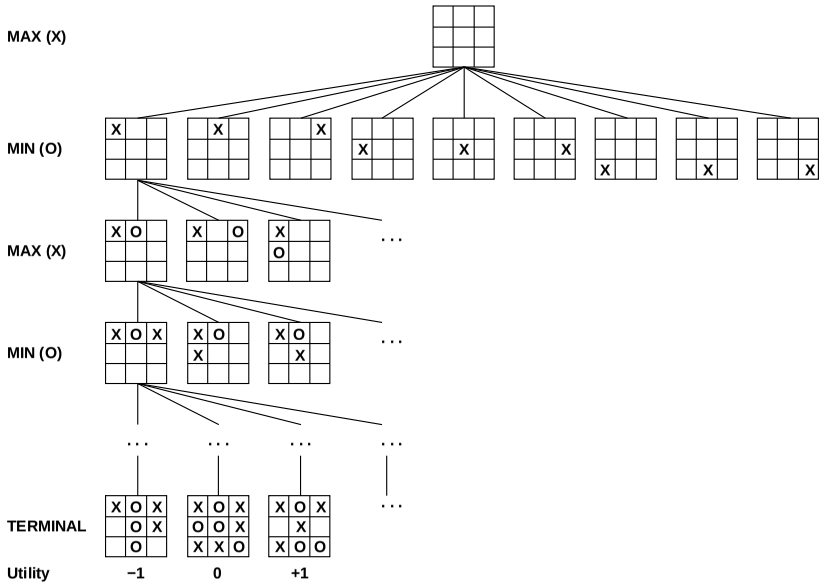
Metody przeszukiwania w przestrzeni stanów nie dają się bezpośrednio zastosować w typowej grze dwuosobowej ze względu na konieczność uwzględnienia ruchów przeciwnika, które nie są znane. „Rozwiązaniem" musi być tu schemat uwzględniający wszystkie możliwe reakcje przeciwnika.

Dodatkowo, w niektórych grach pełna wiedza o stanie w ogóle nie jest dostępna dla obu graczy.

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

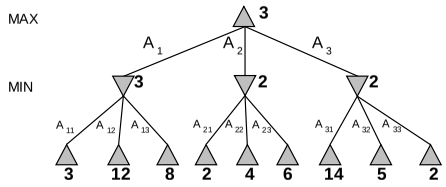
Drzewo gry dwuosobowej



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.

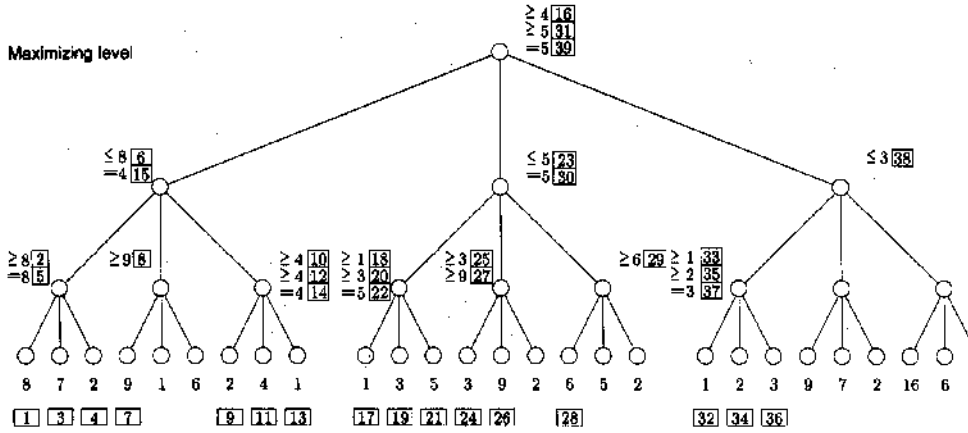
Przeszukiwanie minimax — odcinanie przeszukiwania

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 — ilustracja



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

Odpowiedź: w kroku 10

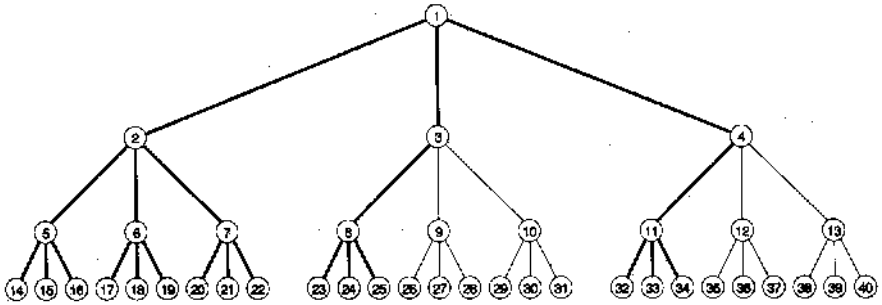
Odcięcia α–β — algorytm

```
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.



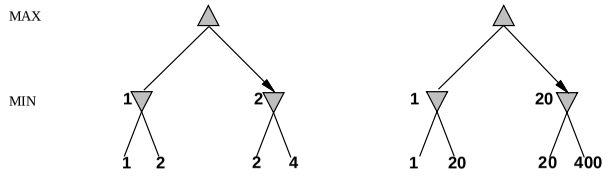
Na powyższym diagramie oszczędność wynosi 16 węzłów; na 27 węzłów na najniższym poziomie obliczonych musi być tylko 11.
Źródło: Patrick Henry Winston, *Artificial Intelligence*, 3rd ed. (uwaga, błąd: węzły 18, 19, 21, i 22 mogłyby również być odcięte).

Własności algorytmu α–β

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

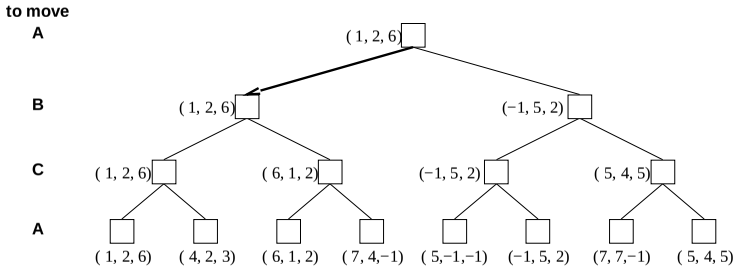
Dobre uporządkowanie pozwala osiągnąć większą efektywność odcinania.
W granicy, optymalne odcięcia pozwalają osiągnąć $O(b^{m/2})$.
W praktyce pozwala to podwoić głębokość przeszukiwania.

Wynik analizy minimax/α–β nie zależy od konkretnych wartości funkcji oceny. Istotne jest tylko uporządkowanie wartości. Oznacza to, że dowolna transformacja monotoniczna funkcja 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.

Praktyka komputerowych gier dwuosobowych

Warcaby: program Chinook w 1994 zakończył 40-letnią dominację mistrza świata Mariona Tinsleya[1] Rok później program pokonał kolejnego mistrza.

Szachy: program Deep Blue w 1997 po raz pierwszy pokonał mistrza świata Gary Kasparowa w otwartym meczu (rok wcześniej mistrz przegrał z tym samym programem pierwszą grę). Przez kolejne 10 lat programy szachowe nie odniosły większych zwycięstw. W roku 2006 program Deep Fritz pokonał w turnieju mistrza świata Vladimira Kramnika. Po tym zwycięstwie zainteresowanie rozgrywkami najlepszych programów z ludźmi zaczęło spadać.

Othello: mistrzowie nie chcą grać z komputerami, które są zbyt dobre.

Go: mistrzowie nie chcą grać z komputerami, które są zbyt słabe. Często rozgrywane są partie z gorszą pozycją startową dla człowieka. W go $b > 300$ więc zamiast systematycznego przeszukiwania drzewa gry, większość programów używa regułowej bazy wiedzy do generacji ruchów.

Wiadomość z frontu: w marcu 2016 program AlphaGo pokonał mistrza 9-dan w równej grze na pełnowymiarowej planszy.

[1] Aczkolwiek mistrz wycofał się z turnieju z przyczyn zdrowotnych i wkrótce potem zmarł na raka.

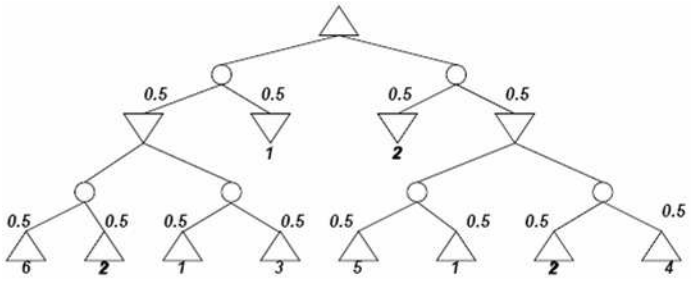
Gry z elementami losowymi — expectimax

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**.

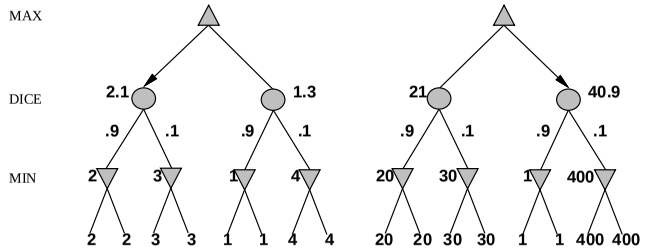
Dalsze uogólnienie — expectiminimax

Pełne uogólnienie algorytmu minimaksu o czynniki losowe nazywamy 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.

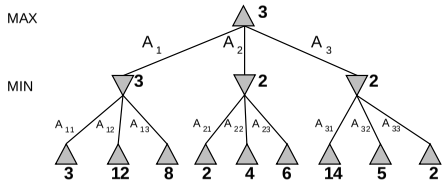
Można obliczyć prawdopodobieństwa wszystkich kombinacji rozdania.

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).



Przeszukiwanie z więzami

Zagadnienia przeszukiwania z więzami (*Constrained Satisfaction Problem, CSP*), albo z ograniczeniami, stanowią specyficzną grupę problemów przeszukiwania w przestrzeni stanów, zdefiniowane jako:

- skończony zbiór zmiennych $X = \{x_1, x_2, ..., x_n\}$
- dla każdej zmiennej x_i skończony zbiór możliwych jej wartości, zwany **dziedziną** danej zmiennej
- skończony zbiór więzów (*constraints*), zwanych również ograniczeniami, na kombinacje wartości zmiennych, np., jeśli $x_1 = 5$, to x_2 musi być parzyste, a kombinacja $(x_1 = 5, x_2 = 8, x_3 = 11)$ jest niedozwolona

Rozwiązaniem problemu CSP jest każda kombinacja wartości zmiennych, która spełnia wszystkie więzy.

Zauważmy, że problemy CSP są w istocie szczególnym przypadkiem ogólnego przeszukiwania w przestrzeni stanów, jeśli potraktujemy zbiór więzów jako specyfikację celu, a przypisywanie wartości zmiennym jako operatory. Można zatem zastosować do tych zagadnień wszystkie wcześniej poznane metody.

Przeszukiwanie z więzami — bliższa analiza

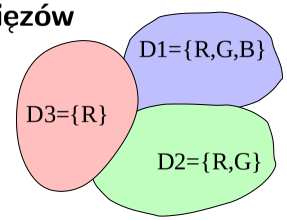
Przykłady problemów CSP: kolorowanie grafu lub mapy, problem 8-hetmanów (klasyka), problem SAT (przypisanie wartości 0/1 elementom formuły logicznej spełniające tę formułę), kryptoarytmetyka, projektowanie układów VLSI, tzw. problem etykietowania węzłów pozwalający rozpoznawać obiekty na obrazach po ich konturowaniu, kolejkowanie zadań, planowanie, i wiele innych.

Wiele spośród tych zagadnień stanowią problemy NP-trudne.

Rozwiązanie problemu CSP może istnieć lub nie, bądź może istnieć wiele rozwiązań. Celem przeszukiwania może być znalezienie jednego dowolnego rozwiązania, wszystkich rozwiązań, bądź rozwiązania optymalnego w sensie jakiejś danej funkcji kosztu.

Można traktować więzy występujące w CSP jako binarne, czyli obejmujące pary zmiennych. Więzy obejmujące więcej zmiennych można przekształcić na binarne, a więzy na pojedyncze zmienne można wbudować w definicje ich dziedziny.

Lokalne spełnianie więzów



Rozważmy problem kolorowania mapy. Należy tak przypisać obszarom danej mapy kolory ze zbiorów dopuszczalnych kolorów, być może różnych dla różnych obszarów, aby obszary sąsiadujące miały przypisane różne kolory.

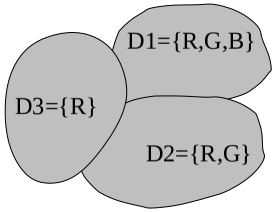
Zanim zaczniemy przeszukiwać przestrzeń możliwych podstawień wartości zmiennych, możemy przeprowadzić pewne analizy **lokalnego** spełniania więzów.

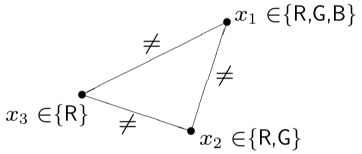
Rozważmy **graf więzów**, którego węzły odpowiadają zmiennym, a łuki — więzom oryginalnego problemu (binarnym). Traktujemy łuk grafu jako parę przeciwsobnych łuków skierowanych, i określamy **spójność łukową** (*arc consistency*) łuku skierowanego $x_i \longrightarrow x_j$ grafu, która zachodzi jeśli $\forall x \in D_i \; \exists y \in D_j$ i para (x, y) spełnia wszystkie więzy określone na łuk.

Można przywrócić spójność niespójnym łukom przez usuwanie wartości z dziedzin pewnych zmiennych (konkretnie, tych wartości $x \in D_i$ dla których nie istnieje wartość $y \in D_j$ spełniająca jakiś wybrany więz).
Ta procedura pozwala na zredukowanie i uproszczenie oryginalnego problemu.

Spójność łukowa

Rozważmy następujące przykładowe zagadnienie kolorowania mapy:
$D_1 = \{R, G, B\}$,
$D_2 = \{R, G\}$,
$D_3 = \{R\}$,
$\mathcal{C} = \{x_1 \neq x_2, x_2 \neq x_3, x_1 \neq x_3\}$.





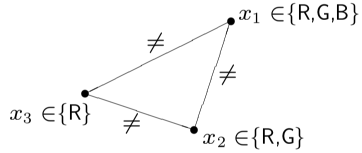
Dla łuku (x_1—x_2) spójność łukowa zachodzi, ponieważ zarówno $\forall x \in D_1 \; \exists y \in D_2 \; x \neq y$ jak i $\forall y \in D_2 \; \exists x \in D_1 \; x \neq y$.

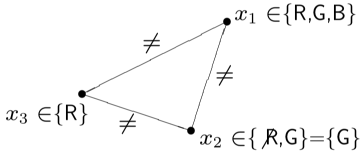
Fakt zachodzenia spójności łukowej to w efekcie niezbyt pozytywna wiadomość. Oznacza on, że analiza spójności tego konkretnego łuku w grafie nic nie wnosi. Jednak pełna analiza grafu CSP może czasami przynieść bardzo wymierne wyniki.

Przykład: kolorowanie mapy

Ponownie rozważmy zagadnienie kolorowania mapy: $D_1 = \{R, G, B\}$, $D_2 = \{R, G\}$, $D_3 = \{R\}$, $\mathcal{C} = \{x_1 \neq x_2, x_2 \neq x_3, x_1 \neq x_3\}$.

Analiza pierwszego więzu ($x_1 \neq x_2$) nie daje żadnych wyników, bo, jak już stwierdziliśmy, dla tego więzu istnieje spójność łukowa. (Dla każdej wartości z D_1 istnieje w D_2 wartość spełniająca ograniczenie, i na odwrót.)

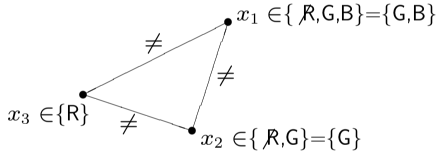




Jednak analiza drugiego więzu ($x_2 \neq x_3$) przynosi pewne pozytywne rezultaty. O ile dla $x_3 =$ R istnieje odpowiednia wartość dla x_2, to dla $x_2 =$ R nie da się dobrać wartości x_3 spełniającej ten więz. Zatem wartość R możemy usunąć z dziedziny zmiennej x_2.

Przykład: kolorowanie mapy (cd.)

Podobna analiza w odniesieniu do więzu ($x_1 \neq x_3$) pozwala wykluczyć z dziedziny zmiennej x_1 wartość R:



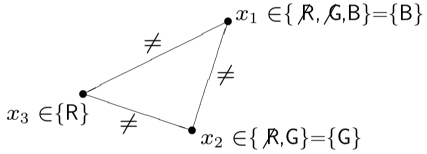
Analiza wszystkich więzów zakończyła się częściowym usunięciem wartości z dziedzin zmiennych. Zagadnienie zostało zredukowane (mniej jest możliwych przypisań wartości zmiennym), ale nadal istnieje więcej niż jedno potencjalne rozwiązanie.

Łatwo jednak zauważyć, że analizę spójności łukowej można, i należy, kontynuować.

Propagacja więzów

Ponieważ wynikiem sprawdzania spójności łukowej jest redukcja dziedzin niektórych zmiennych, ma sens jego powtórzenie nawet dla łuków grafu, które pierwotnie spójność posiadały, bądź została im ona przywrócona. Prowadzi to do **propagacji więzów**, czyli powtarzania analizy spójności więzów i redukcji dziedzin tak długo jak daje to jakieś efekty.

Propagacja więzów w omawianym przykładzie kolorowania mapy powoduje dla łuku $(x_1 \neq x_2)$ — początkowo spójnego — usunięcie wartości G z dziedziny x_1:

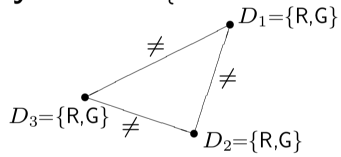


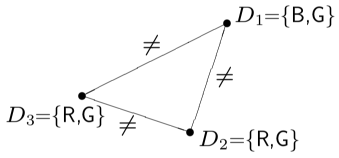
Ostatecznie wszystkie zmienne mają jednoelementowe dziedziny, i w dodatku zawarte w nich wartości spełniają wszystkie więzy. Zatem propagacja więzów doprowadziła w tym przypadku do znalezienia jedynego rozwiązania.

W ogólności jednak analiza spójności i propagacja więzów prowadzi jedynie do uproszczenia, a niekoniecznie do całkowitego rozwiązania problemu.

Spójność łukowa — przykłady nierozwiązane

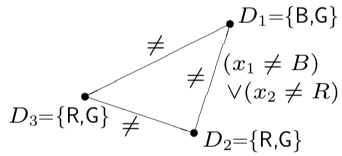
Jak łatwo zauważyć, w przedstawionym tu innym przykładzie zagadnienia kolorowania mapy, wszystkie łuki są spójne. Pomimo to, zagadnienie nie posiada rozwiązania.





W kolejnym przykładzie ponownie wszystkie łuki są spójne. Zagadnienie posiada dwa rozwiązania, lecz propagacja więzów nie pozwala ich wyznaczyć, a wręcz nie przynosi żadnych redukcji dziedzin zamiennych.

Jeśli do poprzedniego przykładu dodamy więz: $(x_1 \neq B) \vee (x_2 \neq R)$ to otrzymamy wariant, w którym tylko jedno z dwóch rozwiązań jest dopuszczalne, ale nie da się go wyznaczyć metodą propagacji więzów.



Zatem obliczanie spójności łukowej i propagacja więzów same w sobie nie gwarantują znalezienia rozwiązania. Konieczne jest przeszukiwanie.

Algorytmy propagacji więzów

Najprostszym sposobem sprawdzania spójności łukowej jest branie po kolei wszystkich więzów, i obliczanie ich warunków logicznych, oraz powtarzanie procesu tak długo, aż pełny cykl sprawdzania wszystkich więzów nie przyniesie żadnych zmian. Przy wielu więzach może to trwać długo. Jednak ponieważ te same warunki sprawdzane są wiele razy, możliwe są pewne oszczędności.

Można zauważyć, że po wykonaniu redukcji pewnej dziedziny D_i propagacja może przynieść nowy wynik tylko przy powtórnym sprawdzaniu łuków o postaci (D_k, D_i), i tylko takie warto sprawdzać. Co więcej, przy jakiejkolwiek redukcji w D_k nie ma już potrzeby sprawdzania łuku (D_i, D_k), ponieważ elementy usunięte w wyniku tej redukcji z D_k nie były już potrzebne dla zapewnienia spełnienia jakiegokolwiek więzu dla żadnego elementu z D_i. Taki sposób propagacji więzów nazywa się algorytmem AC-3 (1977).

Kiedy spójność łuku sprawdzana jest po raz kolejny, warunki są sprawdzane dla tych samych par wartości. Zapamiętanie tych sprawdzonych par wartości w odpowiedniej strukturze danych pozwoliłoby nie powtarzać ich sprawdzania w kolejnych cyklach. Wykorzystuje to algorytm propagacji więzów AC-4 (1986).

Niebinarne więzy

Na początku przyjęliśmy założenie, że można ograniczyć się do rozważania więzów binarnych, tzn. obejmujących dokładnie dwie zmienne. Więzy obejmujące więcej zmiennych można przekształcić na binarne.

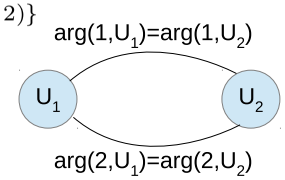
Jeden z najprostszych schematów konwersji więzów na binarne, tzw. **kodowanie dualne** (*dual encoding*). Polega ono na wprowadzeniu nowej zmiennej dla każdego więzu oryginalnego problemu. Dziedziną każdej nowej zmiennej jest zbiór n-ek wartości tych oryginalnych zmiennych, które występowały w więzie, spełniających oryginalny więz.

W ten sposób, oryginalne więzy są automatycznie spełnione przez wartości zmiennych zawartych w nowych zmiennych. Pozostaje zapewnić by wartości spełniały wszystkie więzy naraz. W tym celu kodowanie dualne wprowadza nowe więzy pomiędzy wszystkimi parami tych zmiennych (nowych), które zawierają te same zmienne oryginalne. Treścią więzów jest by odpowiednie zmienne (oryginalne) miały te same wartości.

Niebinarne więzy — przykład

Rozważmy przykład zagadnienia CSP z trzema zmiennymi: $X = \{x, y, z\}$, ich dziedzinami $D_{x,y,z} = \{1, 2, 3\}$, i dwoma więzami: $\mathcal{C} = \{x + y = z, x < y\}$. Kodowanie dualne dla tego problemu zawiera dwie zmienne i dwa więzy:

$$
\begin{array}{lcl}
U_1 & : & \langle oc_1, [x, y, z]\rangle, D_{U_1} = \{(1,2,3),(2,1,3),(1,1,2)\} \\
U_2 & : & \langle oc_2, [x, y]\rangle, \quad D_{U_2} = \{(1,2),(1,3),(2,3)\} \\
C_1 & : & arg(1, U_1) = arg(1, U_2) \\
C_2 & : & arg(2, U_1) = arg(2, U_2)
\end{array}
$$



Niestety, analiza spójności dla tego problemu nic nie wnosi, ponieważ dla każdej wartości jednej zmiennej istnieją wartości drugiej ze zgodnymi poszczególnymi argumentami. Jednak większość wartości zmiennych dualnych stanowią n-ki niedopuszczalne w rozwiązaniu oryginalnego zadania.

Ogólnie, konwersja więzów wieloargumentowych na binarne prowadzi czasami do sformułowań problemów, które nie poddają się analizie spójności. Dlatego opracowano również szereg algorytmów spójności łukowej dla więzów wieloargumentowych. Te algorytmy nie będą tu omawiane.

Spójność ścieżkowa

Definiujemy dla grafu więzów problemu CSP pojęcie **K-spójności**. Graf jest K-spójny (dla pewnego K), jeśli dla dowolnych (K-1) zmiennych, które mają spełnione wszystkie więzy między sobą, dla dowolnej (K-1)-tki wartości tych (K-1) zmiennych spełniającej wszystkie więzy dla (K-1) zmiennych, istnieje wartość w dziedzinie dowolnie dobranej K-tej zmiennej, taka, że powstała w ten sposób K-tka wartości zmiennych spełnia wszystkie więzy dla K zmiennych. Graf więzów jest **silnie K-spójny** jeśli jest K-spójny dla każdego J,J<K.

Zauważmy, że zdefiniowana wcześniej spójność łukowa jest równoważna silnej 2-spójności grafu więzów.

Silna 3-spójność grafu jest również nazywana **spójnością ścieżkową** (*path consistency*).

Znaczenie K-spójności jest takie, że jeśli graf problemu CSP z n węzłami jest silnie n-spójny, to problem można rozwiązać bez przeszukiwania. Jednak algorytmy osiągania K-spójności są eksponencjalne, więc zwykle się to nie opłaca. Wyjątkiem jest osłabiona wersja spójności ścieżkowej — **ograniczona spójność ścieżkowa** (*restricted path consistency*), dla której algorytm jest czasem stosowany.

Przeszukiwanie w zagadnieniach CSP

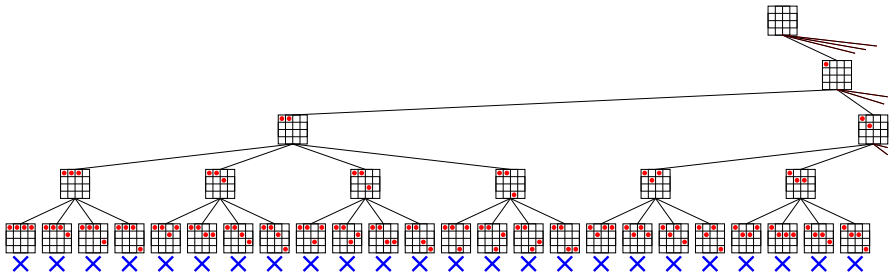
W zagadnieniach CSP można stosować wszystkie omówione wcześniej algorytmy przeszukiwania. Jednak w większości istotnie trudnych problemów CSP, gdzie więzy mają charakter trudnych do rozwiązania, ciasnych kompromisów, największe znaczenie ma właśnie syntaktyczna i semantyczna analiza więzów.

Natomiast zwykle trudno jest wypracować użyteczną heurystykę, która byłaby zdolna pokierować procesem przeszukiwania przestrzeni przypisań wartości zmiennym.

Dlatego często stosowanym algorytmem jest najprostszy z algorytmów przeszukiwania, czyli przeszukiwanie z nawracaniem BT. Zamiast dobrej heurystyki wybierającej najlepsze możliwości w pierwszej kolejności, ten algorytm jest uzupełniony lokalną analizą więzów. Powoduje to redukcję liczby możliwości w kolejnych krokach wgłąb algorytmu. W skrajnym przypadku, gdyby dziedzina którejś ze zmiennych zredukowała się do zbioru pustego, algorytm dokonuje natychmiastowego nawrotu do alternatywnych wartości wcześniejszych przypisań.

Przykład: problem 4 hetmanów

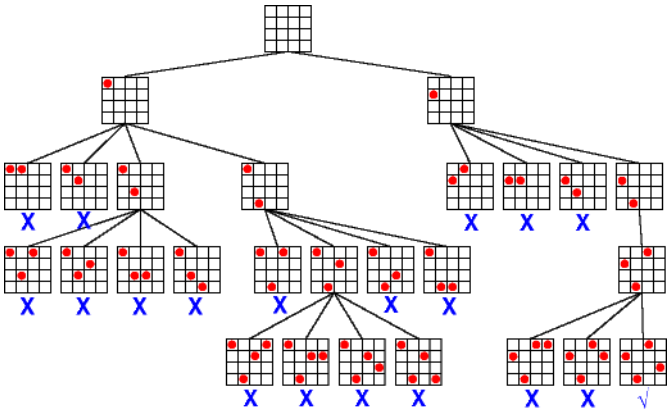
Rozważmy zastosowanie algorytmu przeszukiwania z nawracaniem (BT) do problemu 4 hetmanów. Poniższy rysunek przedstawia fragment drzewa przeszukiwania (należy pamiętać, że algorytm BT przeszukuje to drzewo systematycznie, lecz nie pamięta całej jego struktury, a jedynie bieżącą ścieżkę):



Algorytm oczywiście poradzi sobie z problemem, jednak sprawdza wiele możliwości, które możnaby łatwo wykluczyć z rozważań sprawdzając spełnienie więzów. Zauważmy, że wszystkie zaznaczone konfiguracje końcowe są niepoprawne ze względu na umieszczenie drugiego hetmana, i widać to już na drugim poziomie zagłębienia.

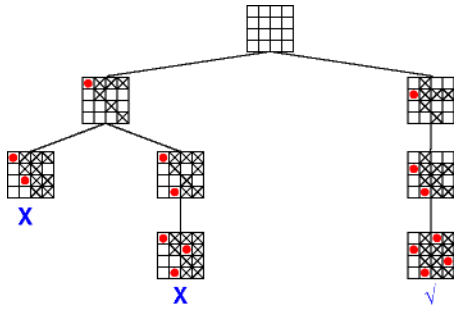
Przykład: problem 4 hetmanów (cd.)

Oczywistym ulepszeniem algorytmu jest więc natychmiastowe sprawdzanie wszystkich więzów binarnych, gdy tylko przypisane zostaną zmienne wchodzące w skład danego więzu. Stwierdzenie naruszenia któregokolwiek więzu powoduje nawrót. Nazwiemy ten algorytm BT-EC (*early checking*). Zauważmy, że wcześniejsze sprawdzanie więzów zawsze się opłaca, ponieważ te same więzy musiałyby i tak być sprawdzone później.



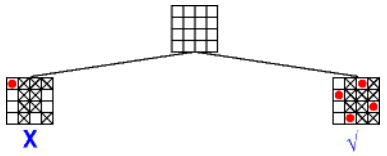
Przykład: problem 4 hetmanów (cd.)

Połączenie przeszukiwania z nawracaniem z minimalną formą sprawdzania spójności więzów zwane jest algorytmem **sprawdzania wprzód** BT-FC (*forward checking*). Sprawdzane są wszystkie więzy dotyczące każdorazowo podstawionej zmiennej, i tylko one. Ten algorytm w prawie każdym przypadku daje lepsze wyniki niż BT-EC, i oczywiście BT.



Przykład: problem 4 hetmanów (cd.)

Możemy również zastosować pełne sprawdzanie spójności łukowej, z propagacją więzów na dziedziny wszystkich zmiennych, również tych jeszcze niepodstawionych. Taki wariant przeszukiwania nazywany jest algorytmem BT-LA (*look-ahead*). W wielu przypadkach pozwala to wyeliminować większość przeszukiwania, jednak ilość obliczeń w tym przypadku może być większa niż w przypadku innych algorytmów, takich jak BT-FC, i stosowanie BT-LA nie zawsze się opłaca.



Nawracanie sterowane zależnościami

W przeszukiwaniu drzewa CSP możemy natrafić na porażkę wywołującą nawrót algorytmu BT, której przyczyną nie było jednak ostatnio dokonane przypisanie wartości, ale któryś z wcześniejszych kroków. W takim przypadku algorytm będzie próbował różnych możliwości, stale generując porażki, aż do momentu, kiedy nawróci dostatecznie głęboko, i zmieni wartość kluczowej zmiennej.

Jest możliwe wykrycie takiej sytuacji — gdy zbiór zmiennych wchodzących w więzy z bieżącą zmienną, tzw. zbiór konfliktowy (*conflict set*), nie obejmuje zmiennej ostatnio przypisanej. Wtedy nawrót mógłby zostać wykonany do ostatnio przypisanej zmiennej z tego zbioru. Algorytm taki zwany jest BJ (*backjumping*).

Prosty algorytm BJ ma znaczenie raczej tylko historyczne, ponieważ rozwiązuje problem, do którego nawet nie dopuszczają algorytmy spójności: BT-FC i dalsze. Jednak można skonstruować bardziej wyrafinowaną (i szerszą) definicję zbioru konfliktowego, jako zbioru tych zmiennych, których przypisane wartości spowodowały porażkę bieżącej zmiennej, wraz z później przypisanymi zmiennymi. Wersja BJ oparta na takiej definicji, zwana *conflict-directed backjumping* wprowadza dalsze usprawnienie procesu przeszukiwania.

Dynamiczne porządkowanie

Wspomnieliśmy wcześniej, że w większości problemów CSP trudno o rzeczywiste heurystyki wspomagające wybór „dobrych" ruchów na drzewie przeszukiwania. Istnieją jednak inne techniki wspomagające przeszukiwanie, związane z **dynamicznym porządkowaniem** (*dynamic ordering*) zarówno zmiennych, dla których opłaca się dokonywać przypisania w pierwszej kolejności, jak i wartości, które warto próbować wpierw.

Heurystyka **najbardziej ograniczonej zmiennej** (*most constrained variable*) (zwana również MRV, *minimum remaining values*), sugeruje wybór zmiennych z najbardziej ograniczonymi (licznymi) dziedzinami. Taki wybór daje największą szansę natrafienia na niespójności, i wynikających z nich redukcji problemu. Ta heurystyka dobrze współpracuje z algorytmem BT-FC.

Inną jest heurystyka **rzędu** (*degree heuristic*), sugerująca wybór zmiennej występującej w największej liczbie więzów z niepodstawionymi zmiennymi.

Po wyborze zmiennej przydaje się heurystyka **najmniej ograniczonej wartości** (*least constraining value*), preferująca wartości wykluczające najmniej wartości innych zmiennych.

Przeszukiwanie lokalne w zagadnieniach CSP

Możliwe jest jeszcze inne podejście do problemów CSP, polegające na mniej lub bardziej przypadkowym wyborze wstępnego przypisania wartości zmiennych, i następnie naprawienia go, jeśli naruszało jakieś więzy. Co więcej, proces tego naprawiania polega na przeszukiwaniu zachłannym, a więc niesystematycznym.

Jedną z heurystyk sprawdzających się w przeszukiwaniu lokalnym jest heurystyka zwana *min-conflict* i polegająca na losowym wyborze jednej ze zmiennych, których wartość narusza jakieś więzy, i przypisaniu jej innej wartości, wybranej tak, aby powodowała najmniej konfliktów (naruszeń więzów) z innymi zmiennymi.

Algorytmy przeszukiwania lokalnego oparte na powyższym schemacie sprawdzają się zaskakująco dobrze w wielu zagadnieniach CSP. Kluczowym ich elementem jest losowość, która pozwala uciec od maksimów lokalnych i innych pułapek, oraz natrafić na kluczową zmienną do poprawienia, lub pominąć niefortunnie wybraną zmienną, której wartość należy przypisać później.

Krótkie podsumowanie — pytania sprawdzające

1. Rozważ problem CSP z czterema zmiennymi: A, B, C, D z dziedzinami: $\{1, 2, 3\}$ dla każdej z nich, i zbiorem więzów podanym niżej. Narysuj graf więzów zadania, po czym spróbuj je rozwiązać metodą propagacji więzów (spójności łukowej). Pokaż każdy krok rozwiązania (bez rysunku). Przedstaw graf po zakończeniu propagacji więzów. Ile reprezentuje on możliwych rozwiązań problemu CSP? Napisz jedno z nich.

   Zbiór więzów:
   $\mathcal{C} = \{C \neq D, B > D, B > C\}$

Linki do literatury

Dobre wprowadzenie do zagadnień CSP Romana Bartáka
`http://ktiml.mff.cuni.cz/~bartak/constraints/constrsat.html`