

Sztuczna Inteligencja

Soma Dutta

Wydział Matematyki i Informatyki, UWM w Olsztynie
soma.dutta@matman.uwm.edu.pl

Wykład - 2: Algorytmy Przeszukiwania

Semestr letni 2020

Podsumowanie ostatniego wykładu

- ▶ Komponenty inteligentnego agenta
 - ▶ Opis środowiska zadań agenta (PEAS)
 - ▶ Funkcja agenta i program agenta
- ▶ Różne klasyfikacje programu agenta
 - (i) Reprezentacja atomowa (Atomic representation)
 - (ii) Reprezentacja rozposzona (Factored representation)
 - (iii) Reprezentacja ustrukturyzowana (Structured representation)

Algorytmy Przeszukiwania

- ▶ Algorytmy przeszukiwania w prostym przypadku należą do kategorii **reprezentacji atomowej** - np. znalezienie **najkrótszej trasy** między dwoma miastami
- ▶ Znalezienie **możliwie najkrótszej trasy** z jednego miasta do drugiego może zawierać połączenia, czas, koszty itp - Tutaj potrzebujemy algorytmów przeszukiwania z **reprezentacją rozproszoną** : Często algorytmy dla tego typu problemów nazywamy **Algorytmami spełniającymi więzy** (Constrained satisfaction algorithms)
- ▶ Dodajmy do poprzedniej sytuacji, że w jednym mieście na możliwie najkrótszej trasie nagle zdarzył się wypadek i wszystkie połączenia są opóźnione - znalezienia rozwiązania tej sytuacji nie da się opisać zestawem wcześniej zdefiniowanych atrybutów, tak jak w przypadku reprezentacji rozproszonych : Potrzebujemy tutaj relacyjnego opisu takich sytuacji, który jest związany z **ustrukturyzowanymi reprezentacjami**

Algorytmy przeszukiwania w prostym przypadku

- ▶ Agent oparty na celach (Goal-based agent) (agent bazujący na celu)
- nazywamy agentem do rozwiązywania problemów (problem solving agent)
- ▶ Budowanie agenta opartego na celach
 - ▶ Sformułuj problem (cel): Zestaw stanów spełniających cel
 - ▶ Przeszukaj Ścieżkę od stanu początkowego problemu do jednego ze stanów spełniających cel
 - ▶ Wykonaj algorytm przeszukiwania

Sformułowanie problemu

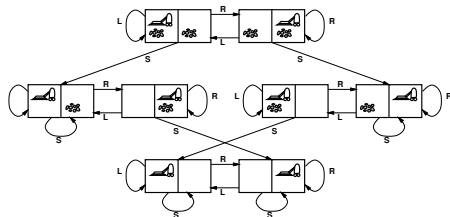
- ▶ Stan początkowy (Initial state): $ln(s)$
- ▶ Opis możliwych działań dostępnych dla agenta: $ACTION(s)$
- ▶ Relacja przejścia opisujący wynik każdej akcji z określonego stanu : $RESULT(s, a)$ - zwraca stan, do którego można dotrzeć za pomocą jednej akcji - ten stan nazywany jest **następnikiem poprzedniego stanu** (successor of the previous state)
- ▶ (**Goal test**) Test celu w celu ustalenia, czy stan jest stanem celu
- ▶ (**Path cost**) Funkcja kosztu ścieżki przypisująca wartość liczbową każdej ścieżce. Koszt ścieżki może być sumą kosztów każdego pojedynczego działania ($c(s, a, s')$) tzn. kroku na ścieżce.

(**State space of the problem**) Przestrzeń stanu problemu jest zbiorem wszystkich stanów, które można osiągnąć ze stanu początkowego za pomocą sekwencji działań

Rozwiązanie problemu przeszukiwania

- ▶ (**Solution**) Rozwiązaniem jest sekwencja działań, która poczynając od stanu początkowego prowadzi do stanu docelowego (stanu spełniającego cel)
- ▶ (**Optimal solution**) Optymalne rozwiązanie to rozwiązanie, które ma najmniejszy koszt spośród wszystkich kosztów rozwiązań
- ▶ Algorytm przeszukiwania został zaprojektowany w celu znalezienia możliwej sekwencji działań, od stanu początkowego do stanu docelowego

Przykład: Odkurzacz



- stany: stan pomieszczeń (czysto/brudno) i lokalizacja robota
- akcje: *Lewo*, *Prawo*, *Odkurzaj*, *NicNieRob*
- cel: czysto
- koszt rozwiązania: 1 dla każdej akcji (0 dla *nicNieRob*)

Liczba stanów: $2 \times 2^2 = 8$

W większym środowisku z n lokalizacjami: $n \times 2^n$ stany

Przeszukiwanie drzewa

- ▶ Przeszukiwanie drzewa (Tree search):
 - ▶ **Możliwe sekwencje akcji tworzą drzewo**, którego korzeniem jest stan początkowy, a gałęzie odpowiadają różnym akcjom
 - ▶ **Algorytm przeszukiwania rozpoczyna się od sprawdzenia, czy węzeł główny jest węzłem celu**
 - ▶ Następnie proces realizowany przez algorytm **rozwija kolejny zestaw węzłów, do których można dotrzeć z poprzedniego węzła** - Zestaw węzłów dostępnych do rozszerzenia nazywa się granicą (**frontier**)
 - ▶ Proces rozszerzania granicy trwa aż do znalezienia rozwiązania lub do momentu gdy nie ma już stanów do rozszerzenia
 - ▶ Mogą istnieć różne algorytmy przeszukiwania drzewa w zależności od strategii wyboru węzła do rozwinięcia

Pseudokod do przeszukiwania drzewa

function TREE-SEARCH(**problem**) **returns** a solution, or failure
initialize the **frontier** with **initial state** of the problem

Loop do

if the **frontier** is empty then return failure

choose a **leaf-node** and remove it from **frontier**

if the node contains a goal state return the corresponding solution

expand the chosen node, adding the resulting node to the **frontier**

Przeszukiwanie grafów

- ▶ Przeszukiwanie grafu pozwala **uniknąć powtarzania odwiedzania stanów**

function GRAPH-SEARCH(**problem**) **returns** a solution, or failure
initialize the **frontier** with **initial state** of the problem
initialize the **explored set** to be empty

Loop do

if the **frontier** is empty then return failure
choose a **leaf-node** and remove it from **frontier**
if the node contains a goal state return the corresponding solution
add the node the explored set
expand the chosen node, adding the resulting node to the **frontier**
only if not in the **frontier or explored set**

Struktura danych potrzebna do algorytmów przeszukiwania

- ▶ Algorytmy przeszukiwania korzystają z odpowiednich struktur danych, aby móc śledzić budowane drzewo lub graf
- ▶ Dla każdego węzła n potrzebuje wykorzystywane są struktury danych, która przechowujące następujące informacje
 - ▶ $n.STATE$: Stan w przestrzeni stanów, któremu odpowiada węzeł n
 - ▶ $n.PARENT$: Węzeł w drzewie wyszukiwania, z którego generowane jest n
 - ▶ $n.ACTION$: Działanie (akcja) zastosowane w węźle nadrzędnym w celu osiągnięcia węzła n
 - ▶ $n.PATH-COST$: $g(n)$ - Koszt ścieżki ze stanu początkowego do węzła n
- ▶ Dlatego węzeł zawiera informacje o stanie węzła, takie jak węzeł nadrzędny, dla którego generowane jest działanie, oraz koszt osiągnięcia tego stanu

Wykorzystanie kolejki (queue) w algorytmach przeszukiwania

- ▶ Załóżmy, że musimy przechowywać 'frontier' w taki sposób, aby algorytmy przeszukiwania mogły łatwo wybrać następny węzeł do rozszerzenia zgodnie z preferowaną strategią
- ▶ Różne warianty kolejki są używane bazując na preferowanej strategii
 - ▶ **FIFO** (First In First Out): zwraca najstarszy element w kolejce
 - ▶ **LIFO** (Last In First Out): zwraca najnowszy element w kolejce
 - ▶ **PRIORITY QUEUE**: element o najwyższym priorytecie pojawia się w określonej kolejności przy tej strategii

Aby śledzić eksplorowany zbiór

- ▶ Eksplorowany zbiór może być zaimplementowany przez 'hash table', która może skutecznie sprawdzać powtarzalność stanów

Różne aspekty pomiaru wydajności

- ▶ Wydajność (jakość algorytmu można mierzyć na cztery sposoby:
 - ▶ **Kompletność (Completeness)**: Czy algorytm gwarantuje znalezienie rozwiązania, o ile istnieje rozwiązanie?
 - ▶ **Optymalność (Optimality)**: Czy algorytm znajduje optymalne rozwiązanie?
 - ▶ **Złożoność czasowa (Time complexity)**: Ile czasu zajmuje znalezienie rozwiązania?
 - ▶ **Złożoność pamięciowa (Space complexity)**: Ile pamięci potrzeba przy przeszukiwaniu?
- ▶ Złożoności czasowa i pamięciowa zależą od następujących czynników:
 - ▶ **Czynnik rozgałęziający (Branching factor)**: b - maksymalna liczba następników dowolnego wężła
 - ▶ **Głębokość (Depth)**: d - głębokość najbardziej płytkiego wężła docelowego
 - ▶ **Maksymalna długość (Maximum length)**: m - maksymalna długość dowolnej ścieżki w przestrzeni stanów

Koszt przeszukiwania

- ▶ Aby ocenić jakość algorytmu przeszukiwania, możemy wziąć pod uwagę koszt przeszukiwania.
- ▶ Koszt przeszukiwania może obejmować złożoność czasową i pamięciową.
- ▶ Można również rozważać całkowity koszt, który obejmuje koszt przeszukiwania i koszt ścieżki rozwiązania.

Podział strategii przeszukiwania

- ▶ Ślepe (Brutalne) strategie przeszukiwania (Uninformed/blind search strategies):
 - ▶ W tych strategiach nie są dostępne żadne dodatkowe informacje oprócz definicji problemu
 - ▶ Mogą generować następniki i odróżniać stan docelowy od innych stanów
- ▶ Heurystyczne strategie przeszukiwania (Informed/heuristic search strategies):
 - ▶ W tych strategiach poza definicją problemu wiadomo, czy jeden stan celu jest bardziej obiecujący niż drugi stan celu
 - ▶ Ponieważ heurystyki te mogą mieć zakodowaną wiedzę na temat konkretnego problemu, mogą skuteczniej znajdować rozwiązania

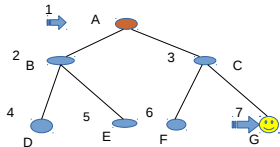
Brutalne (Ślepe) strategie przeszukiwania

- ▶ **Breadth-first search** (Przeszukiwanie wrzecz) przeszukiwania grafowego
 - ▶ najpierw rozwijany jest węzeł główny
 - ▶ następnie rozwijane są wszystkie następniki węzła głównego
 - ▶ ogólnie wszystkie węzły na danej głębokości są rozszerzane przed dowolnym węzłem na następnym poziomie
 - ▶ wykorzystywana jest kolejka FIFO
 - ▶ zawsze wybiera najkrótszą ścieżkę do każdego węzła należącego do zbioru wierzchołków bezpośrednio osiągalnych z danego wierzchołka 'frontier'
- ▶ **Pomiar wydajności** (Miary jakości)
 - ▶ **Kompletny**: Jeśli najbliższy węzeł celu znajduje się na skończonej głębokości d , algorytm w końcu go znajdzie
 - ▶ **Nieoptymalny**: Najbliższy (do danego wierzchołka) węzeł docelowy niekoniecznie musi być optymalny. Jest optymalny, gdy koszt ścieżki nie jest funkcją malejącą (np. wszystkie działania mają taki sam koszt)

- ▶ **Złożoność czasowa:** $b + b^2 + \dots + b^d = O(b^d)$
- ▶ **Złożoność pamięciowa:** $O(b^{d-1})$ ponieważ każdy węzeł wygenerowany dla ostatniego poziomu jest przechowywany w pamięci jako zbiór eksplorowany

```

function BREADTH-FIRST-SEARCH(problem) returns solution or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /* chooses the shallowest node */
    add node.STATE to explored
    for each action in problem.ACTIONs(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in frontier or explored then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
  
```



Brutalne strategie przeszukiwania

▶ Uniform cost search:

- ▶ Zamiast rozszerzać ścieżkę do najpłytszego wężła, rozszerza węzeł n o ścieżkę o najniższym koszcie $g(n)$
- ▶ Odbywa się to poprzez przechowywanie 'frontiers' jako kolejki priorytetowej (PRIORITY QUEUE) uporządkowanej według g
- ▶ Test celu jest dodawany do wężła, gdy jest on wybierany do rozwinięcia, a nie kiedy jest generowany po raz pierwszy
- ▶ Dodawany jest test, aby sprawdzić, czy istnieje lepsza ścieżka do wężła znajdującego się obecnie na 'frontier'

▶ Pomiar wydajności

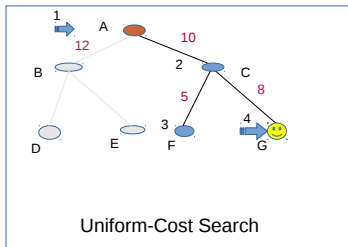
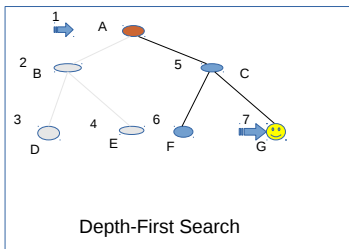
- ▶ **Optymalny**
- ▶ **Niekompletny**: Ponieważ nie ma znaczenia liczba kroków na ścieżce, jeśli istnieje nieskończona pętla z działaniami o zerowym koszcie, program może utknąć
- ▶ **Złożoność czasowa i pamięciowa**: Jeśli C jest kosztem optymalnego rozwiązania, a ϵ najmniejszym kosztem dla każdego działania, to złożoność czasowa i pamięciowa jest $O(b^{1+\lceil \frac{C}{\epsilon} \rceil})$.

Brutalne strategie przeszukiwania

- ▶ Depth-first search (Przeszukiwanie w głąb)
 - ▶ Rozszerza najgłębszy węzeł na bieżącej 'frontier'.
 - ▶ Natychmiast przechodzi do najgłębszego poziomu, w którym węzły nie mają następników, a potem przechodzi do następnego najgłębszego węzła, który wciąż nie został odwiedzony.
 - ▶ Używa kolejki LIFO.

Pomiar wydajności

- ▶ **Kompletny**: Kompletny gdy jest używany do przeszukiwania grafów w skończonej przestrzeni, i niekompletny gdy używana jest wersja przeszukiwania drzewa. W nieskończonej przestrzeni stanów, jeśli napotkamy nieskończoną ścieżkę nieposiadającą wężła celu, wówczas algorytm przeszukiwania nie byłby kompletny.
- ▶ **Nieoptymalny**
- ▶ **Złożoność czasowa**: Może wygenerować wszystkie węzły $O(b^m)$, co może być znacznie większe niż rozmiar przestrzeni stanów. Również m może być znacznie większe niż d .
- ▶ **Złożoność pamięciowa**: Przechowuje tylko jedną ścieżkę od korzenia do liścia i innego nieodwiedzonego 'syna'. Wymaga tylko przechowywania $O(bm)$ węzłów.



Brutalne strategie przeszukiwania

- ▶ **Depth-limited search**: Dodano wstępnie zdefiniowane ograniczenia na głębokość l
 - ▶ **Niekompletny**: Gdy $l < d$
 - ▶ **Nieoptymalny**: Gdy $l > d$
 - ▶ **Złożoność czasowa**: $O(b^l)$.
 - ▶ **Złożoność pamięciowa**: $O(bl)$.
- ▶ **Iterative deepening search**: Określa ograniczenie l poprzez iteracyjne poszukiwanie najlepszego ograniczenia, zaczynając od $l = 0$, aż do osiągnięcia celu gdy $l = d$.
- ▶ **Kompletny**: Gdy b jest skończone.
- ▶ **Nieoptymalny**: Gdy koszt ścieżki jest niemalejącą funkcją węzła głębokości.
- ▶ **Złożoność czasowa**: $O(b^d)$.
- ▶ **Złożoność pamięciowa**: $O(bd)$.

Brutalne strategie przeszukiwania

▶ Bidirectional search (Przeszukiwanie dwukierunkowe)

- ▶ Wprowadza dwa jednoczesne przeszukiwania - jedno w przód od stanu początkowego, a drugie w tył z jednego z węzłów docelowych.
- ▶ Kiedy 'frontiers' dwóch przeszukiwań przecinają się, osiągnąony jest węzeł celu.
- ▶ Złożoność czasowa jest znacznie mniejsza - $2b^{\frac{d}{2}}$.

Dziękuję za uwagę