Metody przeszukiwania

Wstęp

Jednym z elementów powszechnie stosowanych w zagadnieniach metod sztucznej analizy są algorytmy przeszukiwania. Wśród nich można dokonać podziału na metody poszukiwania ślepego oraz metody heurystyczne. Pierwsze z metod nie mają żadnych informacji pozwalającej na ocenę aktualnego stanu rozwiązania, natomiast druga grupa metod wykorzystuje koncepcję funkcji heurystycznej pozwalającej na przybliżoną ocenę kierunku prowadzonego procesu przeszukiwania.

Typowym przykładem metod ślepych są metody wszerz i w głąb.

Metodę wszerz można opisać algorytmem:

1. Zainicjuj kolejkę Queue
2. Queue <- stan początkowy
3. While ((S <- Queue.Head) != null){
    1. If S == rozwiązanie Then koniec
    2. For ((C <- S.Children) & C != Visited)
        1. Queue <- C

}

Jak widać bazuje ona na kolejce, do której dodawane są kolejne węzły przeznaczone do odwiedzenia, pod warunkiem że nie były dotychczas odwiedzone. Proces przeszukiwania pobiera z kolejki kolejne węzły przechodząc do nich w poszukiwaniu określonej wartości, jeśli dany węzeł nie spełnia kryterium stopu jest on dodawany na koniec kolejki.

Metodę w głąb można opisać algorytmem:

1. Zainicjuj Stos
2. Stos <- stan początkowy
3. While ((S <- Stos.tail) != null){
    1. If S == rozwiązanie Then koniec
    2. For ((C <- S.Children) & C != Visited)
        1. Stos <- C

}

Działa ona podobnie jak wyżej z tą różnicą, że kolejne węzły do odwiedzenia dodawane są na stos a nie do kolejki. Powoduje to, iż w pierwszej kolejności rozwijana jest cała gałąź drzewa. Jeśli nie można dalej rozwijać danej gałęzi,

pobierana jest kolejny węzeł ze stosu, który jest pierwszym nieodwiedzonym węzłem drzewa.

Zadanie

Korzystając z algorytmu przeszukiwania *wszerz* i *w głąb* napisz program umożliwiający odnalezienie rozwiązania dla gry typu przesuwanka. Poniżej opisano sposób jego realizacji w środowislu Matlab.

Aby tak postawione zadanie można było zrealizować konieczne jest zdefiniowanie struktury reprezentującej stan gry. Jednym z możliwych rozwiązań jest zdefiniowanie węzła grafu (stanu gry) jako wektora opisującego poszczególne pola, czyli składającego się z 9 elementów. W takim przypadku aktualny stan przesuwanki pokazany na rys. 1 może być zapisany jako:

state=[1 3 8 5 7 2 6 0 4],

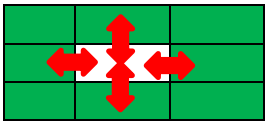
gdzie wolny element reprezentowany jest przez 0.
Taka forma nazywana będzie nieskompresowaną, gdyż składa się z tablicy 9 liczb. Dodatkowo z uwagi na szybkość działania w kodzie będziemy posługiwali się formą skompresowaną, czyli pojedynczą liczbą *138572604*.
Kolejnym niezbędnym krokiem do realizacji zadania jest zdefiniowanie funkcji, która dla danego stanu generować będzie wszystkie możliwe kolejne węzły (stany) do odwiedzenia. W tym celu należy zdefiniować funkcję

stany = generujStanys(stan)

która przyjmuje jeden pojedynczy stan i zwraca listę możliwych do wykonania ruchów. W celu jej implementacji należy zidentyfikować położenie 0 i w zależności od miejsca, w którym ono się znajduje dokonać możliwych modyfikacji.

Np.:

Dla *state(5) == 0* możliwe są przesunięcia:



Czyli mając 4 możliwe do realizacji ruchy należy opisać wartości stanów wynikowych.

Identyfikujemy ile jest możliwych stanów – czasem będzie ich 2, np. jeśli puste miejsce znajduje się w rodu, a czasem 4 – dla sytuacji jak na rysunku

a następnie dokonujemy odpowiednich przesunięć:

```
nowyStan = stan;
nowyStan(5)=stan(4); nowyStan(4) = 0; %Przesunięcie w prawo
nowyStan (5)=stan(2); nowyStan (2) = 0; %przesunięcie w dół
```

nowyStan (5)=stan(6); nowyStan(6) = 0; %Przesunięcie w lewo
nowyStan (5)=stan(8); nowyStan(8) = 0; %Przesunięcie w górę

Zastanów się jak to efektywnie zaimplementować, ponieważ należy rozważyć położenie 0 w każdym z możliwych pozycji!!
Dodatkowo, każdy nowyStan dodaj do zmiennej *stany* , która zwracana jest przez funkcję.

Ponieważ obliczenia mogą długo trwać jako stan początkowy ustaw

```
state = [1 2 3 0 5 6  7 8 4];
```

a jako stan końcowy [1 2 3 4 5 6 7 8 0].

Do realizacji zadań mogą być przydatne klasy *queue* oraz *stack,* dostępne na stornie prowadzącego w sekcji materiały.

Klasy queue i stack mają identyczną składnię, a różnią się jedynie wewnętrzną reprezentacją danych.

Przykład queue/stact

```
q = queue();
q = add(q,1); %Dodoanie do kolejki liczby 1
q = add(q, [1 2 3 4]); %Dodanie do kolejki wektora [1 2 3 4]
size(q) %zwraca liczbę elementów w kolejce
v = element(q); %zwraca wartość z początku kolejki
[q,v] = remove(q); %pobiera ze szczytku kolejki wartość i jednocześnie ją usuwa z
kolejki
```

Ponieważ gra umożliwia powrót do już odwiedzonych stanów, w programie należy dokonać weryfikacji, czy przypadkiem nie jesteśmy w węźle już odwiedzonym. Jeśli tak to nie należy już ponownie odwiedzać tego stanu, gdyż dojdzie do zapętlenia! W tym celu można wykorzystać funkcję

out=contains(wCzym,co)

która zwraca informację czy w macierzy stanów *wCzym* znajduje się już stan reprezentowany przez wektor *co*. Zwróć uwagę że implementacja metody *contains* jest dosyć prymitywna. Do zmiennej zmiennej reprezentującej już odwiedzone stany należy każdorazowo dodać nowo otwiedzany stan np. poprzez

stanyOdwiedzone(end+1,:) = state;

Zadania:

Zad 1 Dokończ implementacji metody przeszukiwania *wszerz,* a kod źródłowy umieść w sprawozdaniu

Zad 2 Zarejestruj ścieżkę dojścia do rozwiązania oraz czas działania programu

Zad 3 Zaimplementuj metodę przeszukiwania *w głąb,* a kod źródłowy umieść w sprawozdaniu

Zad 4 Zarejestruj ścieżkę dojścia do rozwiązania oraz czas działania programu

Zad 5 Porównaj wyniki uzyskane w zadaniach 2 i 4

Zad 6 Zastanów się czy/jak można efektywnie zaimplementować funkcję *contains*. Gdybyś implementował program w językach niższego poziomu jak C/C++/C#/Java jakiej struktury danych użyłbyś do implementacji funkcji contains aby była ona jak najefektywniejsza?