Projektowanie algorytmów i metody sztucznej inteligencji

Projekt 4

Termin zajęć: środa 18.55-20.35

1. Wstęp

Celem projektu było zaimplementowanie dwóch algorytmów przeszukiwania DFS oraz A*, dla zadanego problemu. Należało znaleźć kombinację ruchów białego skoczka po której czarny król zostanie zbity jednocześnie uniemożliwiając skoczkowi znalezienie się na polu które mogła by zaatakować czarna wieża.

2. Analityczne rozwiązanie problemu

1. DFS

   Przeszukiwanie w głąb nie jest rozbudowanym algorytmem i co najważniejsze algorytm znajduje dowolną ścieżkę nie tą najlepszą. Na początku stworzono graf który uwzględniał wszystkie dozwolone i możliwe ruchu skoczka z każdego pola na planszy. Po przygotowaniu grafu wystarczyło sprawdzać sąsiadów każdego z wierzchołków a następnie również jego sąsiada do momentu w którym trafiliśmy na wierzchołek docelowy jeśli taki istniał. Algorytm zaczyna od punktu początkowego umieszcza go na stosie inicjuje go jako zmienna i usuwa ze stosu, następnie sprawdza każdego z sąsiadów dodaje ich na stos jeśli nie byli jeszcze odwiedzeni ustala kto jest poprzednikiem danego sąsiada. Po sprawdzeniu sąsiadów uznaje ten wierzchołek jako odwiedzony. Następnie zdejmuje najwyższy element ze stosu usuwa go z niego i wykonuje ponownie wszystkie czynności aż trafi na wierzchołek docelowy, po czym odtwarza ścieżkę przejścia i wyświetla ją na ekranie.

2. Algorytm A*

   Przeszukiwanie A* jest bardziej złożonym algorytmem. Na początku określamy wzór na funkcje heurystyczną h(n), w tym przypadku określona jest ona metryką Manhattan'a, następnie ustalamy jak będzie wyglądała funkcja g(n) względem której będziemy porównywać czy dana ścieżka jest lepsza niż inna. Istnieje jeszcze funkcja f(n) określona wzorem f(n)=g(n)+h(n) dla najlepszej ścieżki funkcja ta będzie miała wartość minimalną. W projekcie jako wartość funkcji g(n) określono ilość przeanalizowanych przez program wierzchołków, została wybrana taka wartość ponieważ była jedną z najprostszych do zaimplementowania w stworzonym kodzie która pozwalała na poprawne działanie programu. Przeszukiwanie algorytmem A* zaczynamy od stworzenia dwóch zbiorów w jednym będą znajdować się już odwiedzone elementy a w drugim te które możemy jeszcze odwiedzić, zostały one zaimplementowane za pomocą struktury std::set która w znacznym stopniu ułatwiła implementację algorytmu. Do zbioru nieodwiedzonych elementów dodajemy wierzchołek startowy następnie sprawdzamy czy któryś z jeszcze nieodwiedzonych elementów ma niższą wartość dla funkcji f(n) jeśli taki istnieje przeszukujemy dla wierzchołka który jest najbardziej opłacalny. Po określeniu najlepszej dotychczasowej ścieżki sprawdzamy czy dany element nie jest przypadkiem szukanym wierzchołkiem. Jeśli nie jest to dla każdego z sąsiadów określamy czy był on już odwiedzony jeśli nie lub jeśli zakładana wartość funkcji g(n) jest mniejsza niż określona dla tego punktu to określamy go jako aktualnie rozpatrywany wierzchołek aktualizujemy wartości funkcji f(n)

oraz g(n) i powtarzamy proces do odnalezienia wierzchołka docelowego lub do momentu gdy zbiór wierzchołków jeszcze nie odwiedzonych będzie pusty. Po odnalezieniu ścieżki wyświetlany jest numer kroku następnie wierzchołek oraz określone dla niego funkcje f(n) , h(n) oraz g(n).

3. Wnioski

Przeszukiwanie DFS nie znajduje najkrótszej ścieżki chyba że akurat trafiło by się ze pierwsza odnaleziona była by również najkrótszą (w tym przypadku tak jest, przynajmniej przy testach przeprowadzanych w domu aby sprawdzić że nie szuka on najkrótszej drogi wystarczy zmienić pozycje króla z pozycji 7 na np. 9 oraz jego położenie x na 1 a y na 4 w funkcji zbudujGraf()) , jest on natomiast łatwy w implementacji i przy analizie podstawowych problemów na pewno się sprawdza, jego złożoność czasowa wynosi O(V+E) co nie jest złym wynikiem. Algorytm A* jest zdecydowanie trudniejszy do implementacji program wymagał wielu modyfikacji jak również zmuszony byłem do zapoznania się z nowymi bibliotekami aby być w stanie zaimplementować go poprawnie. Algorytm działa bardzo dobrze znajduje on jedną z najbardziej optymalnych dróg (najlepszych rozwiązań jest 3 wszystkie wykonane w trzech ruchach). Jego złożoność czasowa wynosi O(E), oznacza to że jest on szybszy od algorytmu DFS.
W programie dano możliwość zmiany pozycji czarnego króla czarnej wieży i białego skoczka, w przypadku zmiany położenia króla lub chęci zmiany położenia wieży należy uwzględnić do w funkcji zbudujGraf().W momencie gdy nie ma ścieżki wyświetlany jest stosowny komunikat. Możliwe jest również wyświetlenie listy sąsiadów dla każdego wierzchołka. Dałem skoczkowi możliwość zbicia wieży niestety nie jest to uwzględniane w dalszym przeszukiwaniu, lecz w aktualnym ułożeniu nie ma to znaczenia ponieważ jedynym możliwym ruchem po zbiciu wieży jest zbicie króla czyli osiągnięcie celu i jest to jedna z 3 najlepszych dróg.

4. Literatura

[1] https://eduinf.waw.pl/inf/alg/001_search/index.php

[2] https://pl.wikipedia.org/wiki/Algorytm_A*

[3] https://pl.wikipedia.org/wiki/Przeszukiwanie_w_g%C5%82%C4%85b

[4] https://en.wikipedia.org/wiki/A*_search_algorithm

[5] http://lukasz.jelen.staff.iiar.pwr.wroc.pl/styled-2/page-2/index.php (wykłady)

[6] http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

[7] https://en.cppreference.com/w/cpp/container/set

[8] http://cpp0x.pl/kursy/Kurs-C++/Poziom-5/Kontenery-asocjacyjne-std-set-i-std-map/589

[9] http://cpp0x.pl/kursy/Kurs-C++/Poziom-5/Kontener-std-vector/588

[10] https://en.cppreference.com/w/cpp/container/vector

[11] http://cpp0x.pl/dokumentacja/standard-C++/vector/819

[12] https://www.geeksforgeeks.org/vector-in-cpp-stl/

[13] https://en.wikipedia.org/wiki/Depth-first_search

[14] https://en.wikipedia.org/wiki/Taxicab_geometry