SPRAWOZDANIE

PROJEKT 4

PROJEKTOWANIE ALGORYTMÓW I METOD SZTUCZNEJ INTELIGENCJI

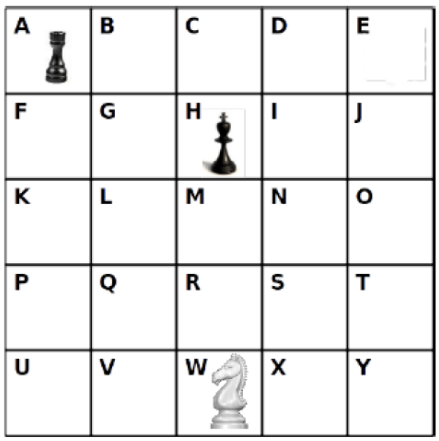
AUTOR: EMILIA SZYMAŃSKA

PROWADZĄCY: DR INŻ. KRZYSZTOF HALAWA

GRUPA: WTOREK 13:15 – 15:00

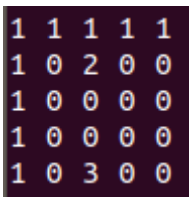
1. Wstęp

Celem projektu jest implementacja algorytmu przeszukiwania wgłąb (DFS) oraz efektywnego algorytmu wyszukiwania najkrótszej ścieżki do danego wierzchołka w grafie – tzw. A*. Problemem, które te algorytmy mają rozwiązywać, jest problem zbicia czarnego króla białym skoczkiem. Skoczek nie może znaleźć się z zasięgu bicia czarnej wieży ani nie może odwiedzić żadnego pola więcej niż raz. Rozpatrywana plansza znajduje się na grafice poniżej.



2. Rozwiązanie analityczne wraz z omówieniem algorytmów

Aby skoczek zaszachował króla, nie stając w zasięgu bicia wieży ani na niej (A, B, C, D, E, F, K, P, U), musiałby się znaleźć na polu Q, S lub O. Skoczek wykonuje ruchy w konkretnie zdefiniowany sposób (na kształt litery L), niektóre pola nie są z punktu widzenia skoczka ,,połączone" w możliwą ścieżkę.
By rozwiązać problem znalezienia drogi skoczka do celu, należałoby przekształcić planszę w ,,skoczkowy" graf (skoczkowy, bo przy dodawaniu krawędzi bralibyśmy pod uwagę jedynie te połączenia pól, których odwzorowanie w ruchach jest możliwe dla skoczka). Na samym początku przekonwertowałam widoczną planszę na macierz uzupełnioną 0 w pustych polach, indeksem wieży (1) w polach zasięgu bicia wieży, indeksem skoczka (3) na pozycji skoczka oraz indeksem króla (2) na pozycji króla. Poniżej znajduje się zobrazowanie tego zabiegu.



Następnym krokiem jest stworzenie grafu o 25 wierzchołkach (bo tyle jest pól na planszy) numerowanych od 0 (dla pola A) do 24 (dla pola Y). Dla każdego wierzchołka rozpatrujemy możliwy ruch skoczka z danego pola - ruch nie może wybiegać poza planszę ani zakładać postawienia figury na polu, które ma wartość różną od 0 (pole puste i dozwolone) lub 2 (pole, które końcowo chcielibyśmy zająć). Ruch taki dodajemy

jako krawędź w grafie o wartości 1 pomiędzy obecnie rozpatrywanym polem a danym możliwym ruchem. Przykładowo dla pola B (wierzchołek 1) istnieje krawędź skierowana do pola M (wierzchołek 12), jednak dla M nie istnieje krawędź skierowana do pola B. Po dokonaniu konwersji planszy w graf możemy wywoływać założone algorytmy.

DFS:

Algorytm ten pozwoli nam na znalezienie ścieżek z pola startowego do danych pól, jeśli taka ścieżka istnieje. Do jego implementacji potrzeba dodatkowej tablicy przetrzymującej informację, czy odwiedziliśmy dany wierzchołek oraz tablicy do przechowywania wyniku przejścia algorytmu. Do wywołania funkcji niezbędny jest także rozpatrywany w danym obejściu wierzchołek oraz graf. Na początku ustawiamy ten wierzchołek na odwiedzony, wrzucamy go do tablicy wynikowej. Pobieramy informację o wierzchołkach, z którymi sąsiaduje i jeśli nie zostały one już odwiedzone, to rekurencyjnie wywołujemy funkcję od każdego z nich. Po takich przejściach otrzymujemy przykładową tablicę:

```
22 19 12 9 18 11 8 17 14 23 16 13 6 6 24 24 13 16 23 14 17 8 11 21 21 18 9 12 19 22
```

Na podstawie jej zawartości możemy odczytać ścieżkę do danego wierzchołka (bierzemy po kolei wartości do momentu napotkania wierzchołka szukanego, ,,powtórki" pomijamy) lub kolejność przejścia algorytmu (bierzemy po kolei wartości, jeśli napotkamy ,,powtórkę", to cofamy się do jej poprzedniego wystąpienia, usuwając wraz z ,,powtórkami" występujące między powtarzającymi się wartościami elementy).

DFS wywołałam dwukrotnie w programie – raz tworząc ścieżkę do trzech szachujących pól (nie dodawałam pola 'H' jako możliwości ruchu dla skoczka), a raz tworząc ścieżkę do 'H' (dodałam pole 'H' jako możliwość ruchu skoczka).

Kwestia zamiany numerów wierzchołków na litery to zrzutowanie sumy litery 'A' i numeru wierzchołka na typ char.

A*:

Algorytm A* jest rozszerzeniem algorytmu Dijkstry o heurystykę. A* dla danego grafu, wierzchołka startowego i wierzchołka końcowego zwraca najkrótszą ścieżkę między tymi wierzchołkami wraz z jej kosztem. Jest to szybsze rozwiązanie niż Dijkstra, gdyż po dotarciu algorytmu do wierzchołka końcowego kończymy działanie programu z gotowym rozwiązaniem. Wymaga to wprowadzenia pewnej estymacji kosztu drogi do celu. Na kolejce priorytetowej bowiem ustawiamy wierzchołki na podstawie ich estymowanego, całkowitego kosztu danego wzorem $f(n) = g(n)+h(n)$, gdzie $g(n)$ to koszt drogi do danego wierzchołka, a $h(n)$ to przewidywany koszt drogi z wierzchołka n do wierzchołka końcowego. Heurystyka w naszym przypadku dana jest wzorem wynikającej z metryki Manhattan: $h(n) = |u-p|+|v-q|$ (u,v to współrzędne wierzchołka n wynikające z planszy, a p,q to współrzędne pozycji króla).

Algorytm A* zaczynam więc od utworzenia struktury, do której będę zapisywała wyniki (struktura ta zawiera pary odpowiednio kosztów i poprzedzających wierzchołków w

ścieżce z wierzchołka startowego dla konkretnego wierzchołka). Ustawiam wszystkie elementy struktury na wartość nieskończoność (u mnie w kodzie jest to wartość -1). Ustawiam koszt ścieżki z punktu startowego do punktu startowego na 0. Następnie tworzę kolejkę priorytetową par (para (estymowany całkowity koszt f(n), koszt rzeczywisty g(n)), wierzchołek n) i wrzucam do niej parę z wierzchołkiem startowym. Dopóki kolejka nie jest pusta lub nie natrafiliśmy na wierzchołek końcowy sprawdzam parę z początku kolejki i ją zdejmuję z niej. Jeśli dystans ze zdjętego elementu jest większy niż ten w strukturze wynikowej, to przechodzę do następnego obiegu pętli (oznacza to, że lepszy koszt drogi jest w wyniku i nasza wartość jest gorszą alternatywą). W przeciwnym razie pobieramy listę przylegających do wierzchołka krawędzi i sąsiednich wierzchołków i dla każdego połączenia za pomocą iteratora – jeśli dystans w strukturze wynikowej ma wartość nieskończoność lub jest większy niż znaleziona alternatywa – do struktury wynikowej dodajemy nowy koszt ścieżki, nowy wierzchołek poprzedni i wrzucamy nową parę z wartością heurystyczną do kolejki. Do kolejki wrzucam wartości z odwróconym znakiem, gdyż w na początku kolejki znajduje się wartość największa, więc jeśli chcemy mieć na wierzchu najmniejszy koszt, to wystarczy dodać ,,-” w koszcie.

Na koniec za pomocą stosu odzyskuję ścieżkę do interesującego mnie wierzchołka końcowego oraz jej koszt, zwracam to jako ostateczny wynik działania algorytmu.

3. Wynik działania programu

```
                DFS


Paths to all winning positions (,,check'' positions as final positions)
Visiting order: W T M J S L I R O X Q N G Y V
Paths:
W T M J S L I R O
W T M J S L I R O X Q
W T M J S

One path (king's position as final position)
Visiting order: W T M J S L I R O H Q N G Y X V
Path:
W T M J S L I R O H

                A*


Shortest path to check the king:
W L S H
```

4. Wnioski

Jak można zauważyć, algorytm A* zwrócił nam lepsze rozwiązanie od DFS. Wszystkie wypisane możliwości dają ten sam efekt – zaszachowanie króla – jednak DFS nie służy bezpośrednio do znalezienia najkrótszej ścieżki w grafie, bardziej do potwierdzenia, że taka istnieje.

5. Opis kodu

5.1. Algorytm A*

Na implementację algorytmu składają się dwa pliki – AStar.cpp i AStar.hh. Działanie algorytmu zostało dokładnie omówione w punkcie 2., tutaj jest jego odzwierciedlenie.

5.2. Algorytm DFS

Na implementację algorytmu składają się dwa pliki – DFS.cpp i DFS.hh. Działanie algorytmu zostało dokładnie omówione w punkcie 2., tutaj jest jego odzwierciedlenie.

5.3. Macierz

Implementacja macierzy jest w pliku Matrix.hh. Pola tej struktury to ilość kolumn, wierszy oraz dynamiczna macierz. Związane są z nią konstruktor oraz przeciążenie operatora ().

5.4. Graf

W pliku GraphList.hh znajduje się implementacja grafu w reprezentacji listy sąsiedztwa. Na potrzeby programu są tu tylko trzy metody (dodanie krawędzi, zwrócenie rozmiaru grafu, a także listy sąsiadujących wierzchołków dla rozpatrywanego) i konstruktor, więcej nie funkcjonalności nie było potrzebnych.

5.5. Plansza

W plikach Board.hh oraz Board.cpp można znaleźć implementację klasy oraz metod związanych z planszą. Sama plansza jest macierzą wypełnioną ID figur, jest tu także plansza pomocnicza wypełniona odpowiednimi literami. Król, wieża oraz skoczek są reprezentowane jako pary ID i współrzędnych pól przez nie zajmowanych. Metody, które są kluczowe w tym programie, to zwrócenie szachujących pól oraz zamiana planszy w listę par sąsiadujących wierzchołków.

5.6. Tworzenie ścieżki/kolejności przejścia algorytmu

W plikach MakePath.cpp i MakePath.hh znajduje się odzwierciedlenie zamiany wyniku przejścia DFS na kolejność przechodzenia algorytmu i ścieżkę do konkretnego wierzchołka (dokładniejszy opis w punkcie 2.).

5.7. Funkcja main

W funkcji main znajduje się zadeklarowanie wszystkich niezbędnych zmiennych, tworzenie planszy, ustawianie figur na odpowiednich polach, zamiana planszy w graf, wywoływanie algorytmów DFS i A* oraz wypisywanie wyników.

6. Literatura:

https://eduinf.waw.pl/inf/alg/001_search/0125.php

https://elektron.elka.pw.edu.pl/~jarabas/ALHE/notatki3.pdf