Podstawy Sztucznej Inteligencji – Projekt MM.P2

1. Skład zespołu:

Jakub Strawa - 300266
Jarosław Zabuski - 300288

2. Treść zadania:

MM.P2 Najkrótsza ścieżka w grafie ważonym:

Zaimplementować i przetestować algorytm A* dla zadania znalezienia ścieżki o najmniejszej wadze od punktu A do B. Porównać działanie algorytmu A* z algorytmem zachłannym i przeszukiwaniem „brute force” (dla brute force przerwać obliczenia w pewnym momencie).

WEJŚCIE: plik z listą krawędzi grafu (punkt początkowy, końcowy i waga), punkt startowy A i docelowy B.
WYJŚCIE: najkrótsza ścieżka od punktu A do punktu B i jej koszt.

3. Podział pracy:

Jakub Strawa – Implementacja programu obsługującego dane wejściowe, funkcjonalność algorytmu Dijkstry oraz funkcjonalność algorytmu A*.
Jarosław Zabuski – Implementacja funkcjonalności algorytmu „brute force”, raport oraz wnioski.

4. Implementacja:

Program rozwiązujący podany problem znalezienia ścieżki o najmniejszym koszcie od punktu A do B zaimplementowano w języku C++. Jedyną biblioteką poza biblioteką obsługi standardowego wejścia i wyjścia, była ***chrono***, którą wykorzystaliśmy w celu zmierzenia czasów wykonania algorytmów. Wykorzystujemy plik tekstowy *graph.txt w którym z*najduje się spis wszystkich krawędzi danego grafu skierowanego, w formacie trzech intów oddzielonych spacją. Nie stworzyliśmy wbudowanego generatora losowych danych testowych. Zadbaliśmy jednak o wiarygodność wprowadzanych danych poprzez zaimplementowane funkcje. Za pomocą tego oraz otwartego generatora danych losowych (https://www.generatedata.com) udało nam się poprawnie zaprezentować grafy: rzadki i gęsty, odpowiednio w plikach: graphSparse.txt oraz graph.txt. Algorytmy operują na macierzy sąsiedztwa.

4.1. Implementacja algorytmu „brute force”:

W trakcie działania algorytmu tworzona jest lista kombinacji ścieżek, po których algorytm „brute force” przechodzi i sprawdza, czy dana ścieżka dążąca do punktu końcowego jest poprawna i najmniejsza możliwa. Realizacja ta sprowadza się do „naiwnego” sumowania wag ścieżek po których w obecnym momencie przechodzi algorytm i porównywania ich z dotychczas najmniej kosztowną ścieżką, zawartą w grafie. Gdy takiej ścieżki nie udało się odnaleźć, algorytm zwraca koszt przejścia po algorytmie równy INF.

4.2. Implementacja algorytmu Dijkstry:

Algorytm zachłanny, który wybraliśmy to algorytm Dijkstry. Tak jak typowy algorytm Dijkstry oznacza wszystkie wierzchołki jako „nieodwiedzone” i oblicza minimalny koszt przejścia do każdego wierzchołka. Struktury wykorzystywane przez nas do implementacji tego algorytmu to

wektory zawierające odwiedzone i oddzielnie nieodwiedzone wierzchołki, jak również wektor par (int, int), przechowujący dotychczas najkrótsze znalezione drogi do danego wierzchołka oraz z którego wierzchołka wychodzi taka krawędź. Pozwala to na odszukanie aktualnie najdłuższej ścieżki do danego wierzchołka oraz prześledzenie poprzedniego wierzchołka na ścieżce do wierzchołka początkowego.

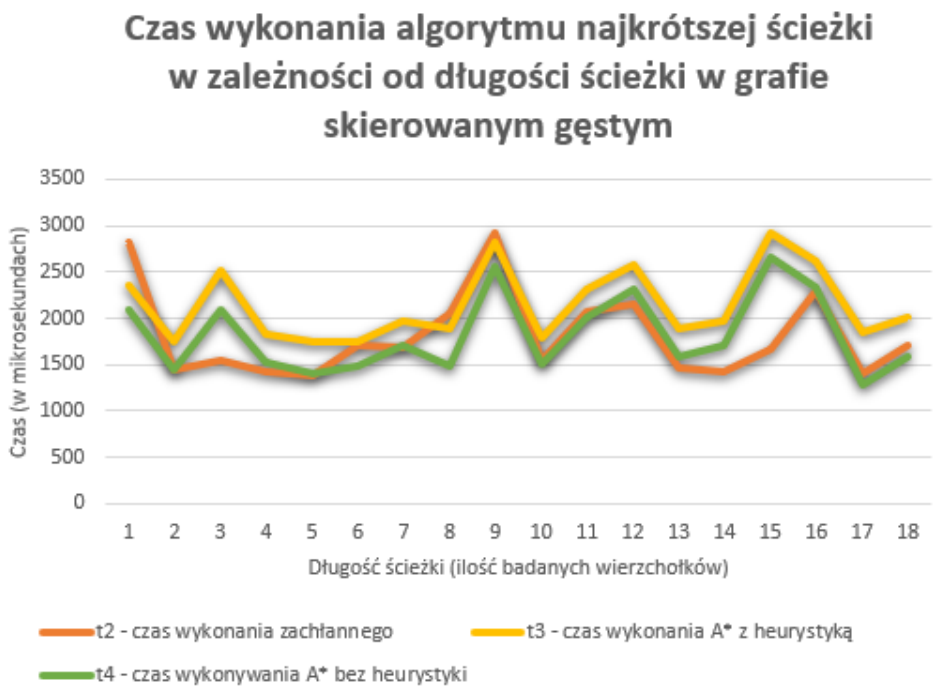
4.3. Implementacja algorytmu A*:

Wykorzystywaną przez nas heurystyką jest minimalna liczba krawędzi dzielących dany wierzchołek od wierzchołka końcowego. Unikamy dzięki temu problemu algorytmu zachłannego, jakim jest rozwiązywanie mniejszych problemów dla każdego wierzchołka, a nie ogólnego problemu znalezienia najkrótszej ścieżki w grafie pomiędzy 2 wierzchołkami. Obrana heurystyka nie jest jednak zbyt dokładna przy kilku możliwych ścieżkach o podobnych (i dość dużych) wagach, ponieważ nie odzwierciedla faktycznego rozstawienia wierzchołków w przestrzeni. Pozwala natomiast wyeliminować sprawdzanie ścieżek, które na pewno nie prowadzą do celu.

5. Przeprowadzone eksperymenty, obserwacje oraz wnioski:

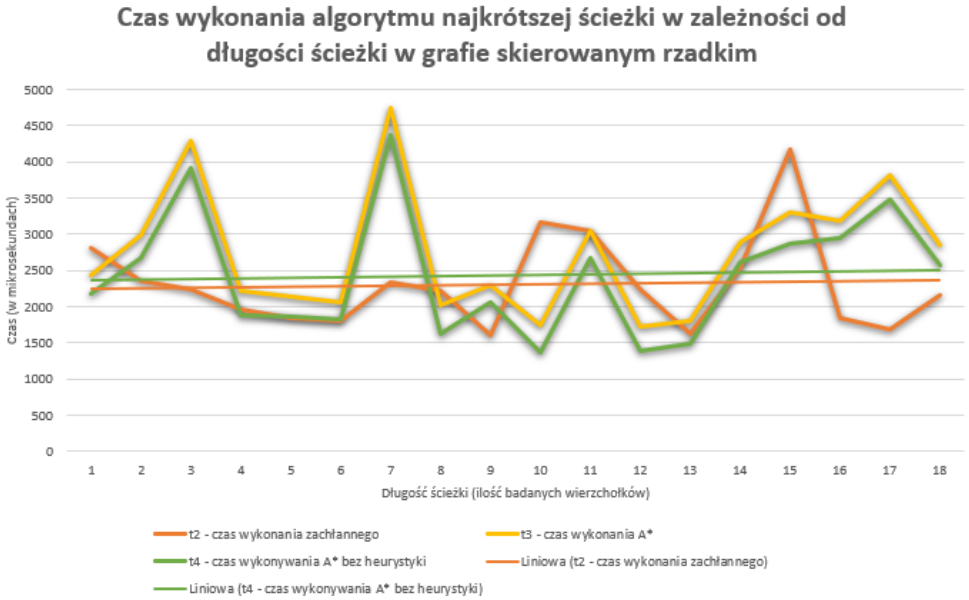
Skupiliśmy się przede wszystkim na zbadaniu czasu potrzebnego powyższym algorytmom na wyznaczenie najkrótszej możliwej ścieżki. Dla grafu gęstego, wyniki eksperymentów wskazały, że złożoność czasowa algorytmu brute force, zgodnie z przewidywaniami, jest równa O(n!) i nie pozwala ona na optymalne rozwiązywanie problemu najkrótszej ścieżki w grafie. Co ciekawe, okazało się, że w wielu przypadkach algorytm zachłanny rozwiązuje problem szybciej od A*. Jest to spowodowane koniecznością wyliczenia przez algorytm A* heurystyki, co dla małej ilości wierzchołków jest dość kosztowne. W przypadku, gdy tworzenie heurystyki nie było brane pod uwagę algorytm A* radził sobie bardzo podobnie lub lepiej od Dijkstry.

Rysunek 1: Wykres zależności czasu wykonywania algorytmu od wielkości grafu, z uwzględnionym algorytmem "brute force".



Rysunek 2: Wykres zależności czasu wykonywania algorytmu od wielkości grafu, bez uwzględnionego algorytmu "brute force".

W przypadku pracy algorytmów nad grafami rzadkimi wykresy nie przedstawiają dużej różnicy pomiędzy algorytmem Dijkstry oraz A*. Przypominają one również o bardzo dużej złożoności czasowej algorytmu „brute force” (wykres bardzo podobny do rys.1). Przy małej ilość wierzchołków, widoczny staje się wpływ liczby krawędzi w grafie na czas wykonywania algorytmów. Z wykresu wynika, że na dane obiekty w bardzo małym stopniu wpływa zmieniająca się ilość wierzchołków, a także sugeruje, że to ilość krawędzi jest stałą, która warunkuje czas wykonywania operacji.

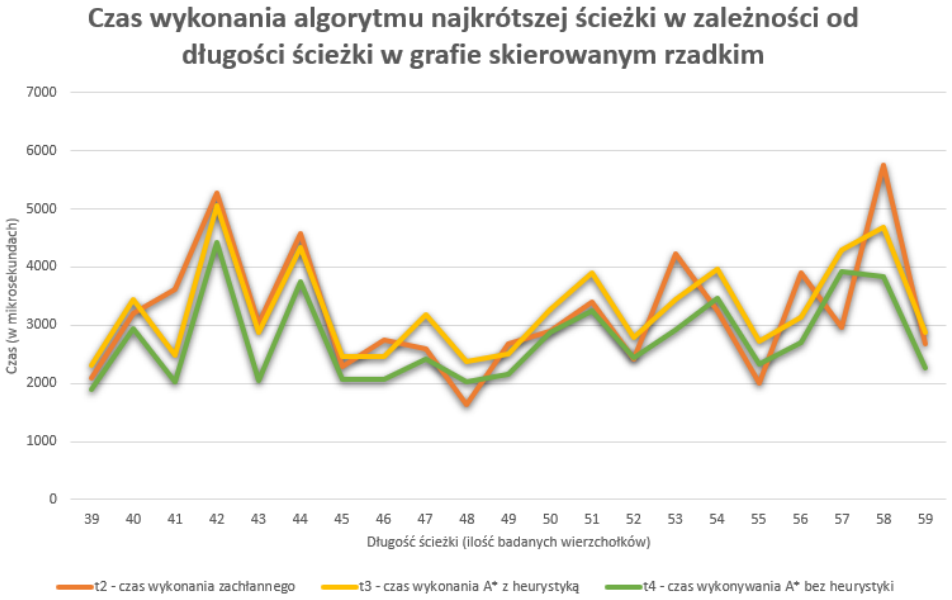


Rysunek 3: Wykres zależności czasu wykonywania algorytmu od wielkości grafu, bez uwzględnionego algorytmu "brute force", lecz uwzględnionymi liniami trendu dla algorytmu zachłannego i algorytmu A bez obliczania heurystyki w locie.*

Postanowiliśmy również zbadać wpływ większej ilości wierzchołków na czas wykonania algorytmów A* i Dijkstry. Dzięki temu uzyskaliśmy dane wskazujące na to, że dla większej ilości wierzchołków to A* staje się optymalny, poza paroma przypadkami. Przykładowo, na wykresie 4 możemy zauważyć, że algorytm zachłanny był szybszy dla długości ścieżki równej 48. Wynika to z faktu, że ścieżka do tego wierzchołka była krótka (zawierała w sobie tylko dwie krawędzie) i były to pierwsze sprawdzone wierzchołki. Oznacza to, że w specyficznych przypadkach obrana

przez nas heurystyka nie jest optymalna, i jej zmianę możemy zaznaczyć jako potencjalne usprawnienie działania algorytmu A*.

Nasze pomiary zgadzają się to z teoretycznymi założeniami o złożoności czasowej obu algorytmów, albowiem algorytm Dijkstry działa w czasie O(|V|^2) w grafie gęstym i O(|E| log |V|) w grafie rzadkim.



Rysunek 4: Wykres zależności czasu wykonywania algorytmu od wielkości grafu, bez uwzględnionego algorytmu "brute force".

6. Instrukcja

Aby stworzyć graf, który program ma przebadać, wystarczy do pliku graph.txt wstawić oddzielone spacją trzy liczby, oznaczające odpowiednio: wierzchołek źródłowy, z którego wychodzi krawędź, wierzchołek docelowy, na który dana krawędź wskazuje, oraz waga, jaką posiada dana krawędź. Na samym końcu pliku znajdują się dane o początkowym wierzchołku A oraz końcowym wierzchołku B, pomiędzy którymi ścieżkę chcemy wyznaczyć. Wystarczy zmodyfikować odpowiednią liczbę przy etykiecie „A” lub „B”, aby zmienić początek i koniec ścieżki, którą chcemy znaleźć.

Uwaga: Numery wierzchołków grafu muszą być kolejnymi liczbami naturalnymi. Czasem lepiej jest zakomentować algorytm brute-force, ponieważ ma złożoność O(n!).

Krótka instrukcja:
1. Sklonuj repozytorium na swój komputer

2. Wykonaj komendę: "cmake [ścieżka do katalogu z plikami]" na przykład:

   "cmake ." (Będąc w katalogu z plikami)

   "cmake path-finding-with-a-star"

3. Wykonaj komendę: "make" (w katalogu z plikami)

4. Uruchom program:

   - pathfinding_with_A_star (dla gałęzi master)

   - pathfinding_with_A_star_heuristic (dla gałęzi heuristic)

5. Ścieżkę do pliku .txt możesz zmienić w pliku main.cpp - zmienna "graphFilePath"