

HiPerA*

Paweł Świder, Modelowanie i symulacja systemów

1. Analiza problemu i dziedziny.

Problemem, którym będziemy się zajmować jest implementacja efektywnego algorytmu znajdowania najkrótszych ścieżek w grafie w symulacji ruchu miejskiego na wielką skalę. Problem jest rozwiązywalny algorytmicznie gdzie do najprostszych algorytmów można algorytm Dijkstry, lub algorytmy heurystyczne takie jak GreedyBFS, GreedyBFS jest oparty o prostą heurystykę, szybszy niż

Dijkstra ale może dawać błędne wyniki. Algorytm Dijkstry poświęca czas na eksplorację nieobiecujących kierunków, natomiast GreedyBFS daje złe wyniki, połączeniem zalet obu algorytmów jest A* który jest podobny do Dijkstry, posiada jednak heurystykę która pozwala mu na poruszanie się w optymalnym kierunku i znajdowanie poprawnej trasy. Warunkiem gwarantującym to że A* zwróci poprawną trasę (najkrótszą) jest heurystyka która nie estymuje większego kosztu na dotarcie do celu niż w rzeczywistości. Poza tym heurystyka może być dowolna.

Algorytmem ulepszającym A* jest HPA (Hierarchical pathfinding A*) – polegające na stworzeniu hierarchii obszarów, najpierw wyznaczamy trasę na największym poziomie abstrakcji, potem schodzimy na niższe obszary i tam znajdujemy bardziej szczegółową trasę. Taka metoda działania jest kilka razy szybsza niż zwykły A*, może dawać nieco mniej optymalne wyniki. Takie podejście wymaga jednak dodatkowych obliczeń w celu stworzenia hierarchii – jest to jednak koszt jednorazowy.

Do wyszukiwania drogi dla pojazdu powstały specjalizowane algorytmy takie jak:

SHPA* - lepsza wydajność pamięciowa i czasowa, przeznaczona dla statycznych grafów.

DHPA* - więcej pamięci, szybsze obsługiwanie zapytań w porównaniu do HPA

SHP - Significant path based Hub Pushing – używające Hub Labelling

LPA* - Livelong planning A*, wagi się zmieniają wraz z czasem (wolniejszy od HPA)

transit node routing – technika, pozwalająca przyspieszyć znajdowanie ścieżek poprzez wcześniejsze obliczenie niektórych tras

HHL – Hierarchical Hub Labelling

Odnosniki do prac naukowych znajdują się tutaj: [hiperastar/Analiza Problemu I Dziedziny.docx at main · TheTryton/hiperastar \(github.com\)](#)

Przystępne wprowadzenie do tematu: [Introduction to the A* Algorithm \(redblobgames.com\)](#)

2. Analiza i wybór narzędzi:

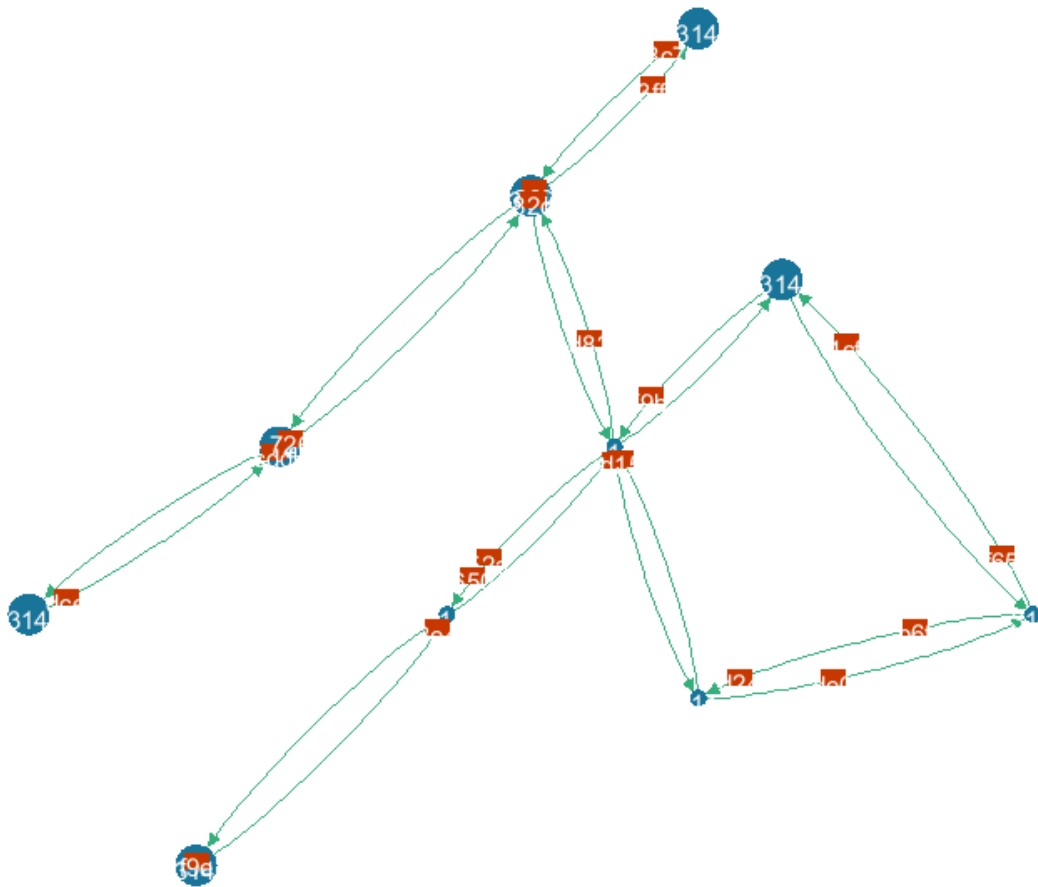
Odpalenie i analiza HiPUTS

W celu odpalenia HiPUTS należało zmodyfikować plik: settings.json ustawić wartości:

```
"testMode":false
"mapPath": "src\\test\\resources\\minimalMap.osm"
"pauseAfterStep":10,
```

Wyłączają one testMode, ustawiają ścieżkę do pliku osm z mapą a także przyspieszają animację.

Działanie można symulatora widzieć poniżej:



Obecnie ścieżka jest losowana w funkcji:

```
private RouteWithLocation generateRoute(LaneId startLaneId, int hops)
```

która jest wywoływana przez funkcję:

```
public Car generateCar(double position, LaneId startLaneId, int hops,  
double length, double maxSpeed)
```

Odpalenie i analiza poprzedniego projektu:

W projekcie znajduje się infrastruktura potrzebna do tworzenia grafów, generowaniu losowych punktów (start, koniec) oraz kilka algorytmów znajdowania najkrótszej ścieżki jak np.:

```
- ContractionHierarchyBidirectionalDijkstra  
- TransitNodeRoutingShortestPath  
- AStarShortestPath
```

Całość korzysta z biblioteki JGraphT, specjalizującej się w algorytmach grafowych. Algorytm który w niej nie występuje i którego była próba implementacji polega na zamienieniu Dijkstry na AStar:

```
ContractionHierarchyAStarShortestPath
```

Jednak ten kod zawiera wiele błędów uniemożliwiających jego skompilowanie.

Dokumentacja JGraphT:

[Overview \(JGraphT : a free Java graph library\)](#)

Oprócz tego jest możliwość zaimplementowania algorytmu HPA* przykładowy link:

[GitHub - Maceris/HPAStar: A java implementation of the HPA* algorithm](#)

3. Określone cele i zakres prac, uruchomione narzędzia

Celem pracy jest zaimplementowanie jak najszybszego algorytmu do znajdowania najkrótszych ścieżek dla dużych grafów (ok. 200000 wierzchołków) w symulatorze HiPUTS. Projekt posiada osobno zaimplementowaną część algorytmów, nie są one jednak podpięte do symulatora.

W trakcie prac założono główne oraz podoczne elementy:

Główne elementy:

1. Dodanie algorytmu znajdowania najkrótszej trasy do HiPUTS – zakończone integracją z symulatorem
2. Zrównoleglenie generowania tras na wiele procesorów
3. Zapisywanie przetworzonego grafu dla Contraction Hierarchies

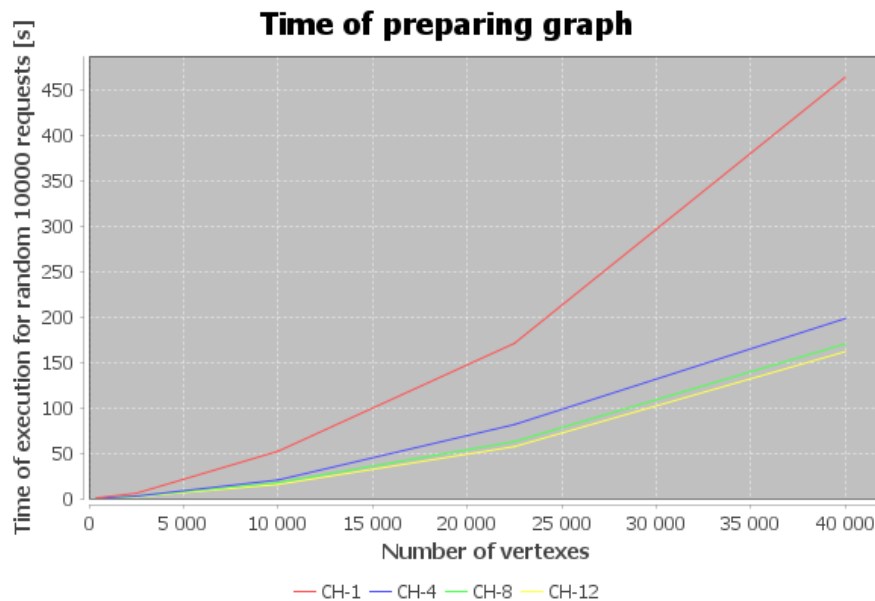
Dodatkowe elementy:

4. Analiza innej struktury grafu
5. Zaimplementowanie Contraction Hierarchies A* - w internecie takie coś nie istnieje, jednak implementacja tego algorytmu wydaje się możliwa
6. Znalezienie odpowiedniej heurystyki dla A* - zgodnie z: [Are there newer routing algorithms \(than Dijkstra, A*\) in GIS databases? - Geographic Information Systems Stack Exchange](#)

Wstępne pomiary czasu obliczeń dla obecnych algorytmów (z pominięciem TransitNodeRouting, którego preprocessing trwa za długo a wyniki dla większych grafów nie są lepsze).



Widzimy że dla algorytmu Contraction Hierarchy uzyskujemy dużo lepsze czasy niż dla zwykłego AStara. Ilość procesorów użyta obecnie również nie ma znaczenia dla szybkości znajdowania tras.



Czas wstępnych obliczeń natomiast intensywnie rośnie i daje się zrównoleglić tylko częściowo.

Czasy dla grafu zawierającego 202500 wierzchołków o stopniu 4 i losowych wagach od 1 do 9.

```
#####450:12#####
Graph generated time taken: 113.7123819s
ContractionHierarchies precomputation time taken: 2528.4408307s
AStar time taken for (requests_count=10000): 1020.5254363s
CHDijkstra time taken for (requests_count=10000): 207.8838099s
```

Oraz powód odrzucenia TransitNodeRouting:

```
#####50:1#####
Graph generated time taken: 0.2337386s
ContractionHierarchies precomputation time taken: 4.8001141s
TransitNodeRouting precomputation time taken: 12.8490009s
AStar time taken for (requests_count=10000): 5.0356376s
CHDijkstra time taken for (requests_count=10000): 2.0122671s
TransitNodeRouting time taken for (requests_count=10000): 1.9194533s
#####100:1#####
Graph generated time taken: 0.3379944s
ContractionHierarchies precomputation time taken: 48.4499683s
TransitNodeRouting precomputation time taken: 292.0560153s
AStar time taken for (requests_count=10000): 26.0978681s
CHDijkstra time taken for (requests_count=10000): 8.956126s
TransitNodeRouting time taken for (requests_count=10000): 8.9215485s
#####150:1#####
Graph generated time taken: 1.4525169s
ContractionHierarchies precomputation time taken: 173.7156793s
TransitNodeRouting precomputation time taken: 1963.2427581s
AStar time taken for (requests_count=10000): 71.7635485s
CHDijkstra time taken for (requests_count=10000): 22.2629859s
TransitNodeRouting time taken for (requests_count=10000): 23.6802435s
```

Wszystkie narzędzia zostały uruchomione na poprzednim etapie i działają.