Jozef Waldhauser

Úloha 8

Hľadanie cesty v 2D mape

Obsah

[Úloha 8 – zadanie 2](#_Toc198507871)

[Úloha 8 – zdroje 2](#_Toc198507872)

[Predslov autora 3](#_Toc198507873)

[Design Algoritmu 4](#_Toc198507874)

[Podporné funkcie 6](#_Toc198507875)

[Teória a analýza 7](#_Toc198507876)

[Greedy best first search 7](#_Toc198507877)

[Analýza greedy best first search algoritmu 7](#_Toc198507878)

[Dijkstrov algoritmus 8](#_Toc198507879)

[Analýza dijkstrovho algoritmu 8](#_Toc198507880)

[A star (A\*) algoritmus 9](#_Toc198507881)

[Analýza a\* algoritmu 9](#_Toc198507882)

[Porovnanie a podrobnejšia analýza 10](#_Toc198507883)

# Úloha 8 – zadanie

1. Naimplementujte metodu path\_planner(self, start, goal), která vezme dvojici souřadnic start a dvojici souřadnic cíl a v mapě block.py vrátí dvě proměnné - frontu obsahující souvislou posloupnost dvojic souřadnic ze startu do cíle a pole expandovaných uzlů k vybarvení
2. V metodě implementujte algoritmy Greedy best first search, Dijkstra a A\*
3. Díky vizualizaci porovnejte množství expandovaných uzlů u jednotlivých algoritmů

# Úloha 8 – zdroje

<https://www.youtube.com/watch?v=GVvN0ikNekw> – greedy bfs

<https://www.geeksforgeeks.org/greedy-best-first-search-in-ai/> - greedy bfs

<https://www.youtube.com/watch?v=6TsL96NAZCo> – a\*

<https://colab.research.google.com/github/cloudpedagogy/data-science-programming/blob/main/python-programming/07_Introduction_to_File_I_O.ipynb/#scrollTo=s_LodqCfrYTV> – nacitanie suborov

<https://en.wikipedia.org/wiki/Taxicab_geometry> - manhattan distance

<https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm> - dijkstra

# Predslov autora

Algoritmus načítava mapu zo súboru *2Dmapa.txt*, mapy si môžete vybrať z ďalšieho súboru *2dmapy.txt*, ktoré som zahrnul v odovzdaní úlohy.

Pôvodne som to spracoval takto, keďže som vedel že máme načítať dáta z iného súboru, ale súbor som nemal dostupný, tak som vymyslel „vlastný prístup“.

Vďaka konzultácií 18.5. som zistil že realizáciu implementácie sme mali robiť cez block\_world.py (zadaniu som neporozumel, hľadal som tento zdroj v prílohách uuManagementKit, kde boli zadania úloh, radil som sa so spolužiačkou, ktorá tiež nevedela čo *block.py* je).

Dúfam že sa Vám moje spracovanie bude páčiť, prajem príjemné hodnotenie!

* S úctou, Waldhauser

## Design Algoritmu

* Class: PathFinder
  + Method: heuristic
    - Manhattan distance current node a cieľa
  + Method: Get\_neighbors
    - Kontroluje stĺpce, riadky a podľa toho rozhoduje či je „platný“ sused (keď je v prvom stĺpci nepriradí ľavého suseda atď.)
    - Je potrebne kontrolovať riadky a stĺpce kvôli tomu že ak dáme do matice matrix[x][y] kde x = -1 tak vráti posledný riadok čo je nechcené chovanie
  + Method: greedy\_best\_first\_search
    - Vstupy štart, cieľ (self, start, goal)
    - Načíta heuristickú hodnotu štartovacieho bodu
    - Dokým nie je heuristika 0 (tzn. nie sme v cieli) opakuje while loop:
      * Zisti susedov bodu, v ktorom sme (metóda get\_neighbors)
      * Vypočítaj iteratívne heuristiku susedov (metóda heuristic)
      * Vyber náhodne z susedov, ktorí majú nižšiu heuristiku
        + Ak nie sú susedia ukončím hľadanie
      * Nastav vybraného suseda ako bod, v ktorom sme
      * Pridaj ho do path array (array of tuples)
      * Pridané počítadlo expandovaných nodes
    - Return path
  + Method: Dijkstra
    - Vstupy štart, cieľ (self, start, goal)
    - Vytvorí množinu - set a dva slovníky - dict
      * Unvisited – set
      * Vzdialenosti – dict
      * Predchodcovia – dict
    - Nastaví všetkým bodom okrem počiatočného vzdialenosť infinity
    - Opakuje while loop dokým nenájde cieľ
      * Vyberie z nenavštívených bodov ten čo sa skúma, a ten čo má najmenšiu celočíselnú vzdialenosť
      * Porovná novo vypočítané vzdialenosti susedov oproti predošlým vzdialenostiam (ak to je možné)
      * Ak táto vzdialenosť doposiaľ neexistuje alebo je menšia ako doteraz bola, tak ju zapíše
      * Nastavíme skúmany bod na ten, ktorí ma najmenšiu hodnotu – vzdialenosť od štartu
      * Keď sa dostaneme do konečného bodu algoritmus končí
    - Zrekonštruujeme cestu
  + Method: A star
    - Vytvorí heap (frontu), dve množiny – sety, tri slovníky – dict
      * openSet – heap
      * openSet\_hash – set
      * closedSet – set
      * cameFrom – dict
      * gScore – dict
      * fScore - dict
    - Každý bod má hodnotu f(n) = g(n) + h(n)
      * g(n) = najkratšia známa vzdialenosť od štartu po bod n
      * h(n) = odhad vzdialenosti z bodu n do cieľa (manhattan distance)
    - Vyberieme bod s najnižšou hodnotou ( f(n) ...) z monžiny openSet
    - Pre každý susedný bod vypočítame novú hodnotu g(n) cez aktuálny bod
      * Ak je cesta kratšia, aktualizujeme g(n), f(n) a predchodcu
      * Ak ešte nie je v openSet, pridáme ho tam
    - Keď sa dostane do cieľa, rekonštruujeme cestu podľa predchodcov
    - Ak openSet zostane prázdny, a cieľ nenájdeme, cesta neexistuje
  + Method: path\_planner
    - Vstupy štart, cieľ (self, start, goal)
    - Vytvorí mapu zo vstupu do class (self.map)
    - Vytvorí array ciest
    - Zavolá greedy bfs a priradí výstup ku premenným
      * Nastaví expanded nodes instance variable (self.expanded\_nodes\_greedy)
      * Pridá cestu z výstupu greedy bfs do array ciest
    - Zavolá dijkstru a priradí výstup ku premenným
      * Nastaví expanded nodes instance variable (self.expanded\_nodes\_dijkstra)
      * Pridá cestu z výstupu dijkstru do array ciest
    - Zavolá a star a priradí výstup ku premenným
      * Nastaví expanded nodes instance variable (self.expanded\_nodes\_astar)
      * Pridá cestu z výstupu astar do array ciest
    - Premení array ciest na queue – frontu
  + Method: draw\_expanded\_nodes\_graph
    - Vykreslí porovnanie expandovaných nodes pre každý graf v porovnaní s všetkými nodes grafu
    - Vstupy: self
    - Zoberie z class premenných expanded nodes greedy, dijkstra, astar grafu
    - Vypočíta all\_nodes a nastaví graph\_data a labels
    - Zobrazí graf

### Podporné funkcie

* Import\_map\_as\_matrix:
  + Načíta 2D mapu zo súboru a prevedie ju na maticu
  + Súbor musí obsahovať validnú python maticu
    - Reprezentácia v matici:
    - 0 = voľné miesto
    - 1 = štart
    - 2 = prekážka
    - 3 = cieľ
  + Načíta súbor, skontroluje maticu cez eval
  + Iteruje cez maticu
    - nastaví štartovaciu pozíciu tam kde je 1 a cieľ tam kde je 3
* Visualize\_path\_graphic:
  + Vizualizuje cesty na mape pomocou matplotlib
  + Prekonvertuje mapu na numpy pole
  + Výšku a šírku nastaví podľa tohto poľa
  + Nastaví farby ciest
  + Vytvorí pre každý algoritmus samostatný plot – vykreslenie
  + Vyfarbí ich a vykreslí
  + Pridá legendu pre každý algoritmus

# Teória a analýza

## Greedy best first search

Greedy best first search (GBFS) je neoptimálny vyhľadávací algoritmus, ktorý sa snaží čo najrýchlejšie priblížiť k cieľu na základe heuristiky. V každom kroku vyberá ten uzol, ktorý má najnižšiu hodnotu heuristickej funkcie (napr. Manhattanova vzdialenosť do cieľa).

### Analýza greedy best first search algoritmu

GBFS je veľmi rýchly a pamäťovo nenáročný, pretože sa zameriava len na najperspektívnejšie uzly podľa heuristiky. Algoritmus neberie do úvahy skutočnú vzdialenosť od štartu, iba odhad do cieľa. Preto môže ľahko zablúdiť do slepých uličiek alebo obísť optimálnu cestu, ak heuristika nie je dokonalá.

A grid with different colored squares

AI-generated content may be incorrect.A graph with different colored squares

AI-generated content may be incorrect.A graph with different colored squares

AI-generated content may be incorrect.

Výhodou je to, že algoritmus je rýchly a nenáročný na pamäť, keďže skúma len najperspektívnejšie smery.

**Dal by sa zlepšiť** napríklad tým, **že by sa pridalo ošetrenie cyklov alebo spätné sledovanie**, aby algoritmus neuviazol v slepej uličke, to ale nie je jeho účel.  
 Bohužiaľ, nevie garantovať optimálnu cestu a môže zlyhať na zložitejších mapách s veľa prekážkami.

#### Výhody a nevýhody greedy best first search algoritmu

Výhody:

* Rýchly a efektívny na veľkých grafoch s dobrou heuristikou
* Jednoduchá implementácia
* Nízka pamäťová náročnosť

Nevýhody:

* Nemusí nájsť najkratšiu (optimálnu) cestu
* Môže uviaznuť v slepej uličke
* Výsledok silne závisí od kvality heuristiky

## Dijkstrov algoritmus

Dijkstrov algoritmus je klasický algoritmus na hľadanie najkratšej cesty v grafe s nezápornými hranami. V každom kroku vyberá uzol s najnižšou známou vzdialenosťou od štartu a aktualizuje vzdialenosti jeho susedov.

### Analýza dijkstrovho algoritmu

Dijkstra je garantovane optimálny – vždy nájde najkratšiu cestu, ak existuje. Prehľadáva všetky možné cesty od štartu a postupne rozširuje „najlacnejšie“ cesty. Je však pomalší a pamäťovo náročnejší ako heuristické algoritmy, pretože musí preskúmať veľa uzlov, najmä vo veľkých grafoch.

A graph with a blue and purple line

AI-generated content may be incorrect.

Moja implementácia nenájde vždy optimálnu cestu ale pracuje správne s prekážkami. Dalo by sa to zlepšiť použitím prioritnej fronty (heapu) namiesto prehľadávania celej množiny nenavštívených uzlov, čo by zrýchlilo algoritmus na väčších mapách. Taktiež nie je najlepšie, že sa v pamäti držia informácie o všetkých uzloch vrátane tých, ktoré nikdy nebudú navštívené.

Algoritmus ide šikmo pretože pri viacerých uzloch s rovnakou vzdialenosťou nemá implementovaný žiadny tie-breaker, ktorý by preferoval priamy smer.   
Pri výbere *min(unvisited, key=lambda node: distances[node])* sa berie prvý uzol s najmenšou vzdialenosťou, ktorý sa nájde, čo vedie k zdanlivo náhodnému výberu medzi rovnocennými cestami.

#### Výhody a nevýhody dijkstrovho algoritmu

Výhody:

* Vždy nájde optimálnu (najkratšiu) cestu
* Funguje na všetkých grafoch s nezápornými hranami

Nevýhody:

* Vyžaduje viac pamäte a času, najmä na veľkých mapách
* Nevyužíva žiadnu heuristiku, takže môže skúmať aj zbytočné smery

## A star (A\*) algoritmus

A\* algoritmus kombinuje výhody Dijkstrovho algoritmu a Greedy best first search. Pri výbere ďalšieho uzla berie do úvahy skutočnú vzdialenosť od štartu (g) aj heuristický odhad do cieľa (h). Uzly vyberá podľa najnižšej hodnoty f(n) = g(n) + h(n).

### Analýza a\* algoritmu

A\* je veľmi efektívny a zároveň optimálny, ak je použitá heuristika "admisibilná" (nikdy nepreceňuje skutočnú vzdialenosť do cieľa). Vďaka heuristike sa vyhýba zbytočnému skúmaniu vzdialených oblastí a sústreďuje sa na najperspektívnejšie smery. Je však náročnejší na pamäť, pretože musí uchovávať viac informácií o uzloch.

A graph with red and blue squares

AI-generated content may be incorrect.

Moja implementácia je dobrá v tom, že kombinuje výhody Dijkstrovho algoritmu a Greedy search - je efektívna a zároveň nájde optimálnu cestu, samozrejme so správnou heuristickou funkciou. Prioritná fronta (heapq) zvyšuje výpočtovú rýchlosť algoritmu.  
Dalo by sa to zlepšiť optimalizáciou pamäťovej náročnosti, napríklad uchovávaním informácií len o relevantných uzloch. Taktiež nie je najlepšie, že výkon algoritmu závisí od kvality heuristiky - pri zlej heuristike môže byť pomalý alebo neefektívny. Pri veľmi veľkých mapách môže byť problémom aj rastúca pamäťová náročnosť.

#### Výhody a nevýhody a\* algoritmu

Výhody:

* Nájde optimálnu cestu (pri správnej heuristike)
* Efektívne prehľadáva graf a vyhýba sa zbytočným uzlom
* Flexibilný – heuristika sa dá prispôsobiť konkrétnemu problému

Nevýhody:

* Vyššia pamäťová náročnosť (uchováva viac informácií o uzloch)
* Výkon závisí od kvality heuristiky
* Pri zlej heuristike sa môže správať podobne ako Dijkstra

# Porovnanie a podrobnejšia analýza

A screenshot of a graph

AI-generated content may be incorrect.

Mapa s blokovaním línie do cieľa - Dijkstra má vizuálne najkratšiu cestu, ale je to klamlivé - v skutočnosti majú všetci optimálnu cestu (44, 44, 44 ak je výpočet správny).

## A screenshot of a crossword puzzle AI-generated content may be incorrect.

Náhodne vygenerovaná mapa - v tomto prípade BFS narazilo do slepej uličky a nenašlo cestu. Dĺžka cesty pre dijkstrov algoritmus je 46 a pre a\* to je 45, čiže je efektivnejší.   
Greedy BFS je rýchly, ale nespoľahlivý na komplikovaných mapách,  
Dijkstra nie je najoptimálnejší, ale cestu nájde, je výpočtovo náročnejší.  
A\* nájde vo väčšine prípadov lepšie alebo rovnaké riešenie ako dijkstrov algoritmus.

A screenshot of a computer screen

AI-generated content may be incorrect.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Druh mapy** | | | | |
| **Algoritmy** | Empty | Random | Veľa prekážok | Line-of-sight block |
| Greedy BFS | Optimálna | N/A | N/A | Len ak pri stene |
| Dijkstra | Dlhšia | Dlhšia | Zväčša optimal. | Dlhšia |
| A\* | Optimálna | Optimálna | Optimálna | Optimálna |

**Vysvetlivky k tabuľke:**

* **Empty:** Mapa bez prekážok, len štart a cieľ
* **Random:** Mapa s náhodne rozmiestnenými prekážkami
* **Veľa prekážok:** Mapa s vysokou hustotou prekážok
* **Line-of-sight block:** Mapa s prekážkou blokujúcou priamy výhľad medzi štartom a cieľom
* **Optimálna:** Algoritmus našiel najkratšiu možnú cestu
* **Dlhšia:** Algoritmus našiel cestu, ale nie optimálnu (najkratšiu)
* **N/A:** Algoritmus nenašiel cestu alebo skončil v slepej uličke
* **Len ak pri stene:** Algoritmus úspešný iba v špecifických prípadoch

Kvôli charakteristike môjho hracieho poľa - pohyby sú možné len v 4 smeroch (hore, dole, vľavo, vpravo), Dijkstrov algoritmus nedosahuje najlepšie výsledky. Často generuje dlhšie cesty v porovnaní s A\*, ktorý preferuje priame línie a nachádza kratšie cesty.

Greedy Best-First Search dokáže v mapách s líniovými prekážkami (line-of-sight block) prekonať Dijkstru, ale len za špecifických podmienok - keď prekážka nie je umiestnená pri stene. Akonáhle sa pridá prekážka ku stene, algoritmus často končí v slepej uličke. Toto správanie je spôsobené tým, že algoritmus sa orientuje výhradne podľa heuristiky a vždy vyberá bod s najmenšou heuristickou hodnotou.

V mojej prvej implementácií som mal náhodný výber so susedov ak boli rovnakí resp. mali rovnakú hodnotu heuristiky, a to ho dokázalo dostať z tohto lokálneho minima raz za cca. 3 behy.

A\* algoritmus konzistentne dosahuje najlepšie výsledky vo všetkých testovaných scenároch. Vždy našiel optimálnu cestu, čo potvrdzuje, že ide o najefektívnejší algoritmus pre hľadanie cesty v 2D mapách. Jeho kombinácia presného merania prejdenej vzdialenosti (ako Dijkstra) a heuristického odhadu zostávajúcej cesty (ako Greedy BFS) mu umožňuje efektívne navigovať aj v komplexných prostrediach.

A graph with blue and purple lines

AI-generated content may be incorrect.

Obrázok 1 prázdna mapa

A screenshot of a puzzle

AI-generated content may be incorrect.

Obrázok 2 mapa s veľkým množstvom prekážok

A screenshot of a graph

AI-generated content may be incorrect.

Obrázok 3 mapa s blokovanou líniou k cieľu a pri ľavej stene

A screenshot of a crossword puzzle

AI-generated content may be incorrect.

Obrázok 4 náhodné prekážky

A screenshot of a graph

AI-generated content may be incorrect.