POROČILO ZA 2. SEMINARSKO NALOGO PRI PREDMETU UMETNA INTELIGENCA

Šolsko leto 2020/21

Nik Prinčič (63190240)

[1. Prostor stanj 3](#_Toc61195397)

[2. Implementirani algoritmi 3](#_Toc61195398)

[2.1. BFS 4](#_Toc61195399)

[2.2. DFS 4](#_Toc61195400)

[2.3. IDDFS 5](#_Toc61195401)

[2.4. A\* 5](#_Toc61195402)

[2.5. Hill Climbing 6](#_Toc61195403)

[2.6. Best First 7](#_Toc61195404)

[3. Primerjava učinkovitosti algoritmov 7](#_Toc61195405)

[3.1. Primerjava algoritmov, ki dobijo optimalno pot 8](#_Toc61195406)

[3.2. Primerjava učinkovitosti algoritma A\* z uporabo različnih hevristik 8](#_Toc61195407)

[3.1. Časovno in prostorsko najbolj učinkoviti algoritmi (optimalni in neoptimalni) na posameznih primerih 9](#_Toc61195408)

# Prostor stanj

Prostor stanj je predstavljen z matrikami, ki vsebujejo trenutno postavitev škatel. Iz vsakega stanja pa se naslednike lahko razvije tako, da se vse vrhnje škatle premakne na vsa dovoljena mesta.

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
| A | B | C |

Primer:

Začetno stanje:

Nasledniki se bodo generirali v naslednjem zaporedju:

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  | A |  |
|  | B | C |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  | A |
|  | B | C |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| B |  |  |
| A |  | C |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  | B |
| A |  | C |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| C |  |  |
| A | B |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  | C |  |
| A | B |  |

Matrike sem implementiral tako, da matrika hrani indekse že premaknjenih stolpcev, tako da se iz enega stanja ne bi dvakrat razvilo isto stanje.

Pri algoritmih BFS, DFS, Hill climbing in Best first sem razvijanje novih stanj omejil tudi tako da se eno stanje lahko razvije samo enkrat, pri algoritmu IDDFS sem pa razvijanje omejil tako da se v trenutnem skladu ne smeta pojaviti dve isti stanji.

# Implementirani algoritmi

Implementiral sem 6 algoritmov in pri informiranih algoritmih uporabil 5 različnih hevristik. Vse algoritme sem implementiral iterativno v jeziku Python. Uspešnost algoritmov pa sem meril z sledečimi metrikami:

* Število opravljenih korakov do cilja (Steps)
* Število razvitih stanj/vozlišč (Total nodes generated)
* Število stanj/vozlišč hranjenih v spominu v trenutku ko je bil najden cilj (Nodes in memory)
* Maksimalno število hranjenih stanj/vozlišč v spominu tekom izvajanja (Max. nodes in memory)

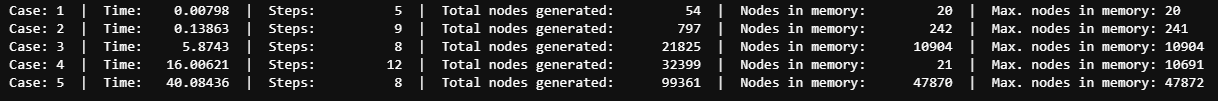
Optimalne premike sem dobil z uporabo DFS, IDDFS (izvedel sem samo do 3 primera zaradi predolgega trajanja) in A\* z uporabo hevristike »misplaced«.

Optimalne rešitve so sledeče:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Primer | 1 | 2 | 3 | 4 | 5 |
|  |  |  |  |  |  |
| Število korakov | 5 | 9 | 8 | 12 | 8 |

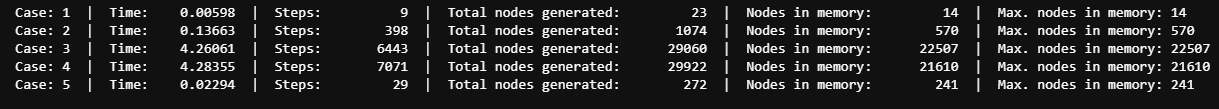
## BFS

Algoritem sem implementiral iterativno z uporabo vrste (queue). Razvijanje novih stanj sem omejil tako, da se stanje ki je že bilo razvito ne sme več razviti. S takim pristopom sem dobil sledeče rezultate:



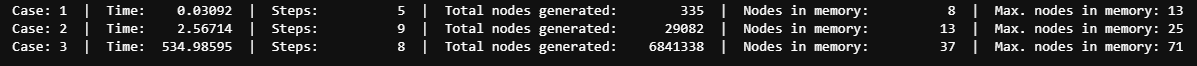
## DFS

Algoritem sem implementiral iterativno z uporabo sklada (stack). Razvijanje novih stanj sem omejil tako, da se stanje ki je že bilo razvito ne sme več razviti, zavedam da to ni ravno najboljša rešitev saj se tako lahko zgodi da se stanje najprej razvije na večji globini in se zaradi tega ne more več razviti na manjši globini, a glede na to da DFS ne išče optimalne rešitve se mi je zdela pridobljena časovna pohitritev vredna teg. S takim pristopom sem dobil sledeče rezultate:



## IDDFS

Algoritem sem implementiral iterativno kliče DFS z omejeno globino in jo po vsaki iteraciji poveča za 1. Razvijanje novih stanj sem omejil tako, da se stanje ki je v trenutni poti, ne sme več razviti, ko DFS zaide v slepo ulico in se vrne na manjšo globino se lahko že nastala stanja na večji globini spet ponovijo. Časovna zahtevnost tega algoritma seveda eksponentno narašča, zato sem uspel izvesti algoritem samo na prvih treh primerih in dobil sledeče rezultate:



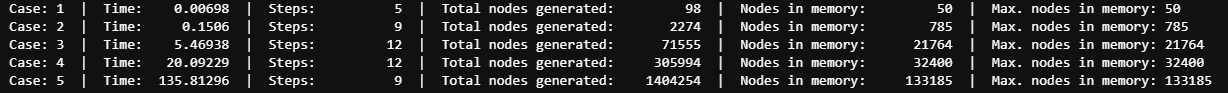
## A\*

Pri algoritmu A\* in tudi pri naslednjih dveh informiranih algoritmih sem uporabil sledeče hevristike:

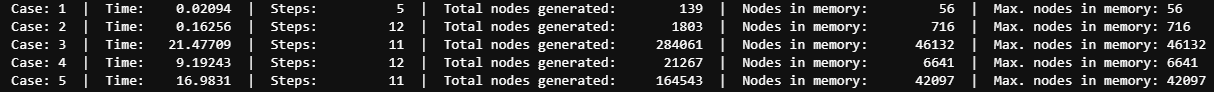
* row: za vsako škatlo upošteva absolutno razliko vertikalne pozicije škatle v trenutnem stanju in končnem/začetnem
* column: za vsako škatlo upošteva absolutno razliko horizontalne pozicije škatle v trenutnem stanju in končnem/začetnem
* manhattan: Manhatanska razdalja med škatlama v trenutnem in končnem/začetnem stanju
* euclidean: Evklidska razdalja med škatlama v trenutnem in končnem/začetnem stanju
* misplaced: za vsako škatlo ki ni na enakem mestu upošteva njeno vertikalno pozicijo

Algoritem A\* sem implementiral s pomočjo prioritetne vrste (priority queue), ki sem jo uporabil kot »open« in množico (set), ki sem jo uporabil kot »closed«. Algoritem sem pognal z vsemi hevristikami, a samo z hevristiko »misplaced« sem dobil optimalno pot na vseh primerih.

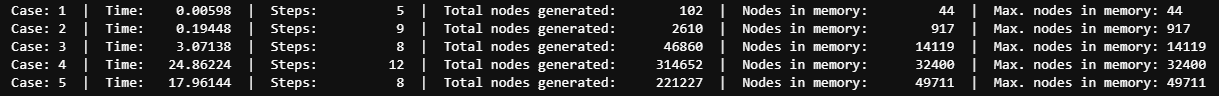
Uporabljena hevristika: row



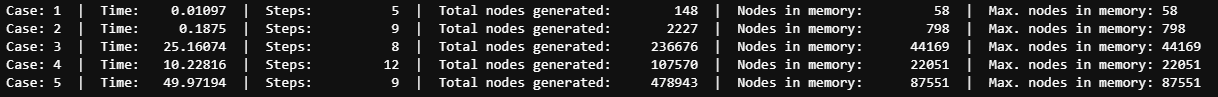
Uporabljena hevristika: column



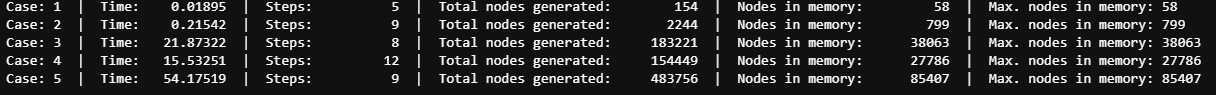
Uporabljena hevristika: misplaced



Uporabljena hevristika: manhattan



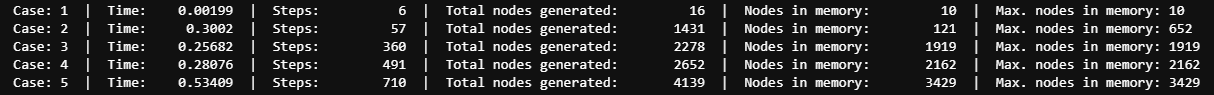
Uporabljena hevristika: euclidean



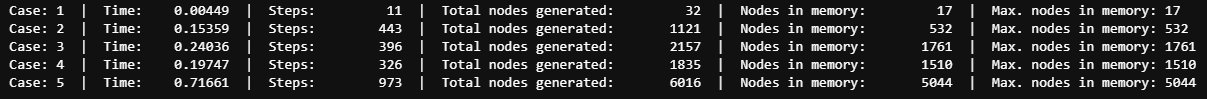
## Hill Climbing

Algoritem sem implementiral z uporaba sklada (stack) na katerega sem dodajal glede na hevristično oceno urejene naslednike, tako kot A\* sem tudi ta algoritem izvedel z vsemi hevristikami in tudi tukaj dobil različne rezultate.

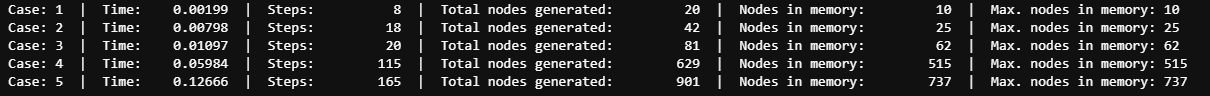
Uporabljena hevristika: row



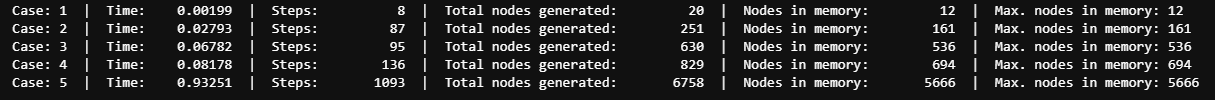
Uporabljena hevristika: column



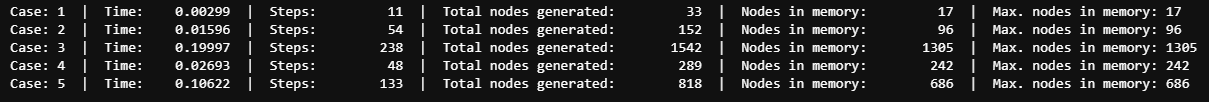
Uporabljena hevristika: misplaced



Uporabljena hevristika: manhattan



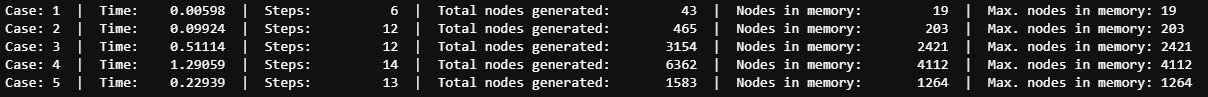
Uporabljena hevristika: euclidean



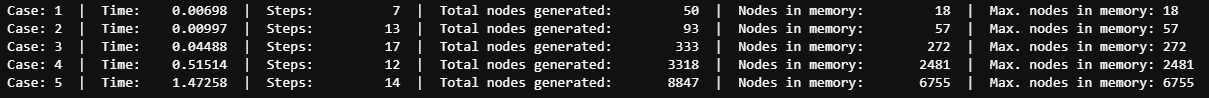
## Best First

Algoritem sem implementiral z uporaba prioritetne vrste (priority queue)v katero dodajam naslednike, njihovo prioriteto pa določa hevristična ocena, tako kot A\* sem tudi ta algoritem izvedel z vsemi hevristikami in tudi tukaj dobil različne rezultate.

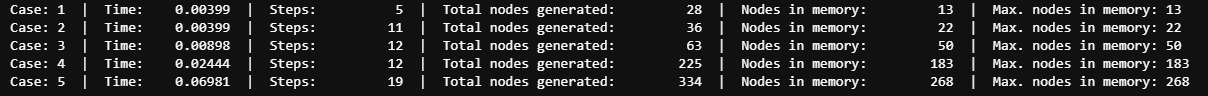
Uporabljena hevristika: row



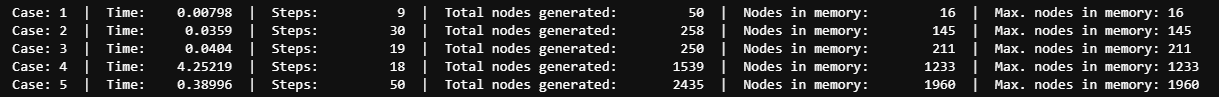
Uporabljena hevristika: column



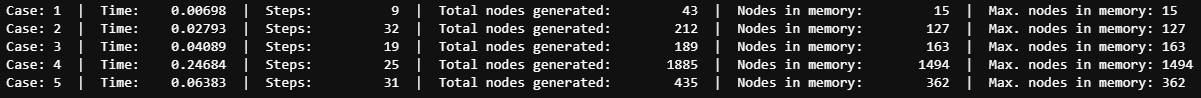
Uporabljena hevristika: misplaced



Uporabljena hevristika: manhattan

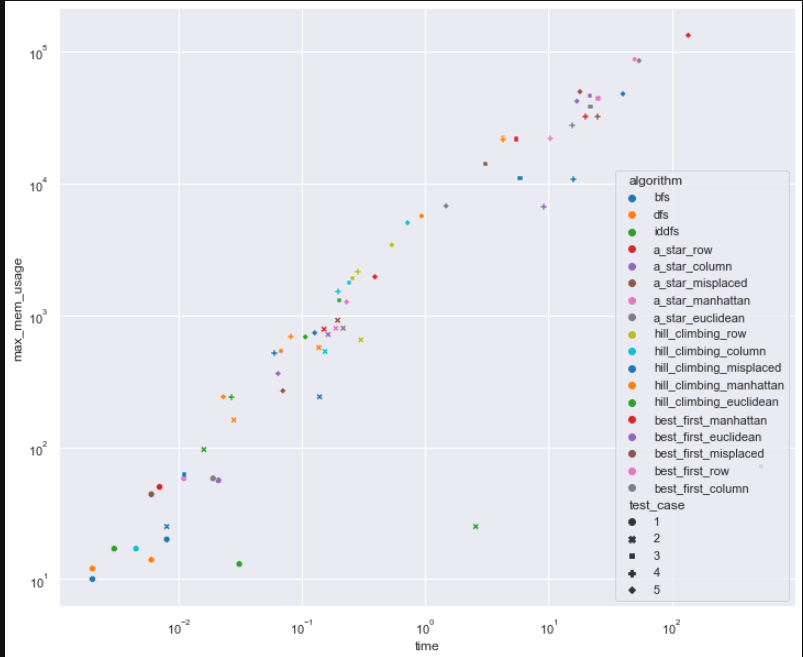


Uporabljena hevristika: euclidean



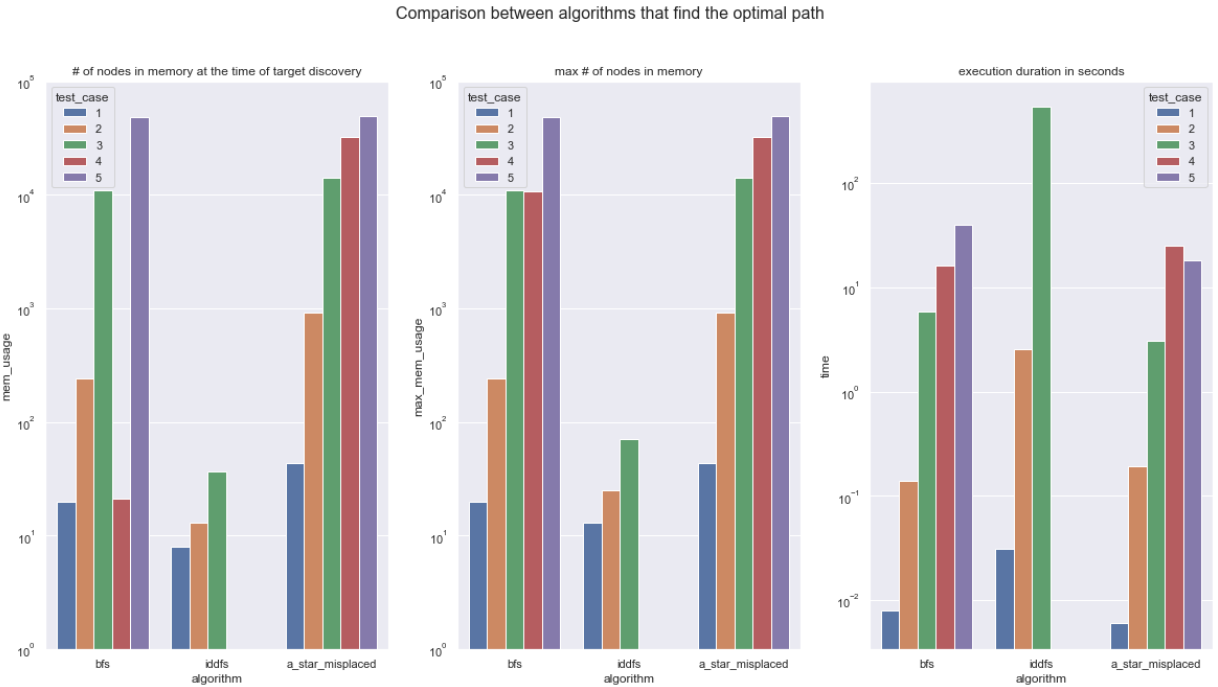
# Primerjava učinkovitosti algoritmov

Na spodnjem grafu lahko približno razberemo, da sta maksimalna poraba pomnilnika in časovna zahtevnost algoritmov linearno povezana.



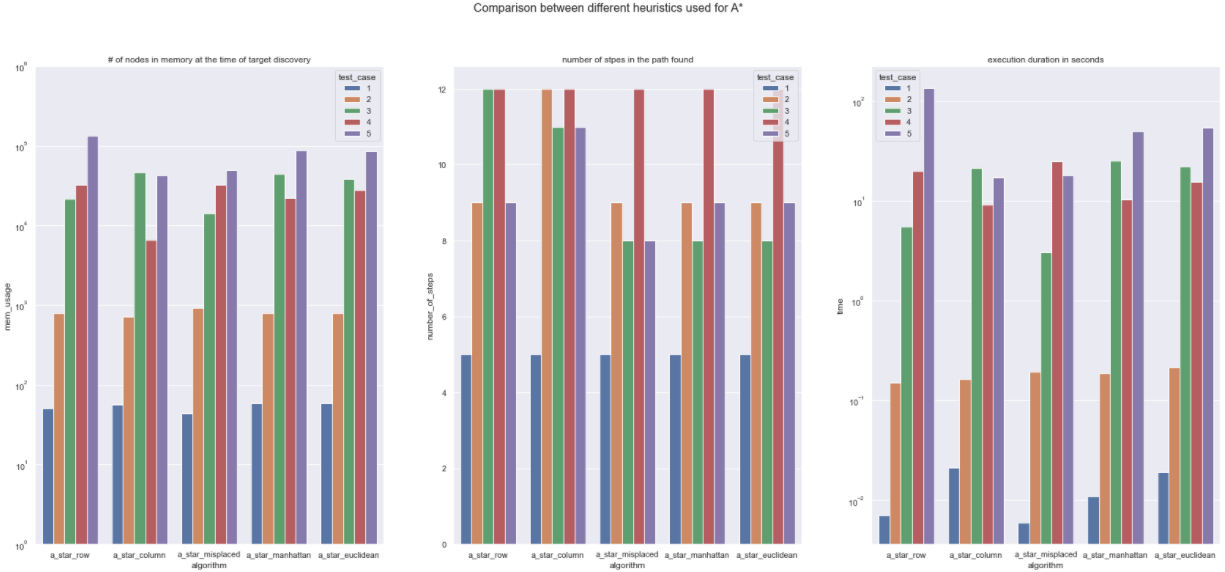
## Primerjava algoritmov, ki dobijo optimalno pot

Iz spodnjih grafov (skale so logaritemske zaradi velikega razpona vrednosti) lahko razberemo, da je prostorska zahtevnost najboljša (najmanjša) pri algoritmu IDDFS, časovna zahtevnost je pa odvisno od primera boljša pri BFS ali pa A\*, sklepam da bi A\* pri bolj kompleksnih (večjih) primerih imel veliko boljšo časovno zahtevnost v primerjavi z BFS, pri primerih, kjer pa je rešitev na dokaj majhni globini, pa je BFS boljši, zardi manjšega »overhead-a« (računanje hevristike, premikanje med open in closed, …), to razliko bi se mogoče dalo rešiti z izbiro boljše hevristike, a se sam nisem spomnil boljše.



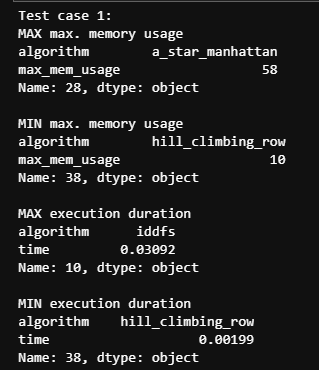
## Primerjava učinkovitosti algoritma A\* z uporabo različnih hevristik

Iz spodnjih grafov (skale so logaritemske zaradi velikega razpona vrednosti) lahko razberemo, da izbira dobre hevristike pripomore k boljši prostorski in časovni zahtevnosti, poleg tega pa samo ena od hevristik (»misplaced«) najde optimalno rešitev. Različne hevristike sicer na večini primerih dobijo optimalno pot, ne pa na vseh.

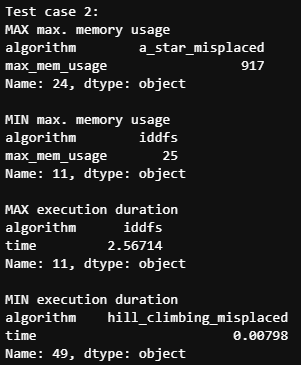


## Časovno in prostorsko najbolj učinkoviti algoritmi (optimalni in neoptimalni) na posameznih primerih

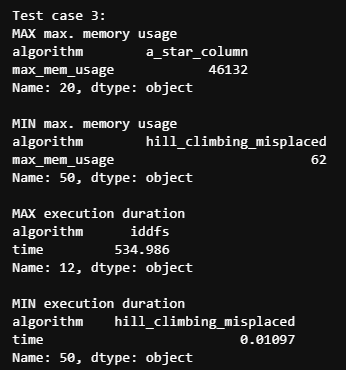
Primer 1:



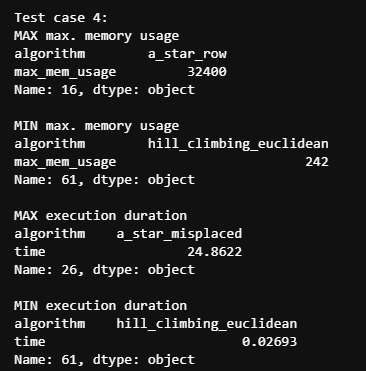
Primer 2:



Primer 3:



Primer 4:



Primer 5:

