Due date Saturday November 26, 2022 11:59pm
Late Submission 30% per day
Teams You can do the project individually or in teams of at most 3.
Teams must submit only 1 copy of the project via the team leader's account.
Purpose In this project, you will implement and analyse a heuristic search.

1 Rush Hour

In this mini-project you will implement and analyse a variety of search algorithms to play Rush Hour. If you already know Rush Hour, do be careful and read the handout carefully as we will modify the rules of game for this project.

1.1 Rules of the Game

The game of Rush Hour is a type of sliding-puzzle developed by Thinkfun. You can play online here. In a nutshell, the game involves an ambulance (or a red car) parked in a lot, and while the driver was away, other vehicles parked and blocked the ambulance. Your goal is to find the shortest sequence of moves so that the ambulance can reach the exit. In the original game:

- The parking lot is a 6 × 6 grid (see Figure 1).
- Each vehicle has a size of at least 2 and can only move on the `x` or the `y` axis, depending on its orientation.
- A vehicle can only slide to a free position inside the grid.
- Moving a vehicle up, down, left or right has the same cost (a cost of 1) regardless of the distance moved.
- The goal is to find the shortest number of moves to get the ambulance to the exit.

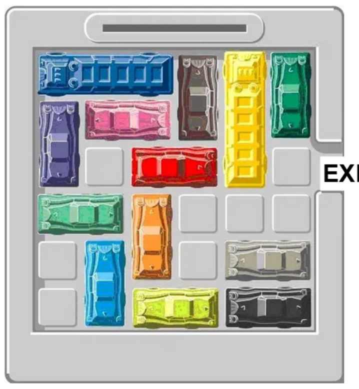

Figure 1: Rush Hour Board

In our version of the game:

- The board of Figure 1 will be represented in 2-D as shown in Figure 2; where A A represents the ambulance, "." represents an empty cell and a sequence of identical letters (eg. E E) represents a vehicle.
- Each vehicle will have an initial quantity of fuel which will limit its displacement. Moving a vehicle by 1 position will reduce its fuel level by 1 unit. For example, in Figure 2, if vehicle D had a fuel level of 1, it would only be able to move down by 1 position, then its fuel level would be zero.
- Once a vehicle reaches position *3f* (the exit), then a mandatory but complimentary valet service will take the vehicle, and remove it from the parking lot. Since this service is free of charge, it will not add anything to the cost of the solution. So, really, you can think of the goal of the game as trying to position the ambulance at position *3f*.

	a	*b*	*c*	*d*	*e*	*f*
1	M	M	M	B	C	D
2	F	E	E	B	C	D
3	F	.	A	A	C	.
4	G	G	H	.	.	.
5	.	I	H	.	K	K
6	.	I	J	J	L	L

Figure 2: Representation of the Board in Figure 1

1.2 Examples

Below are some examples of grids to illustrate the rules of the game you need to implement.

Example 1:

	a	*b*	*c*	*d*	*e*	*f*
1	I	I	B	.	.	.
2	C	.	B	H	H	H
3	C	.	A	A	D	.
4	.	.	.	.	D	.
5	E	E	G	G	G	F
6	.	.	.	.	.	F

- assume here that all vehicles have an infinite fuel level.
- the lowest-cost solution is: $F\uparrow\ 2; G\rightarrow\ 1; E\rightarrow\ 1; C\downarrow\ 2; A\leftarrow\ 2;$ $B\downarrow\ 1; I\rightarrow\ 3; B\downarrow\ 1; A\rightarrow\ 1; C\uparrow\ 3; E\leftarrow\ 1; A\leftarrow\ 1; B\downarrow\ 3;$ $H\leftarrow\ 2; F\uparrow\ 2; I\leftarrow\ 1; D\uparrow\ 2; A\rightarrow\ 5$ with a cost of ~~17~~**18**.

Example 2:

	a	*b*	*c*	*d*	*e*	*f*
1	C	.	B	.	.	.
2	C	.	B	H	H	H
3	A	A	D	D	.	.
4	.	.	.	.	.	.
5	E	E	G	G	G	F
6	.	.	.	.	.	F

- assume here that all vehicles have an infinite fuel level.
- vehicle D can move right by a distance of 2 and reach the exit (i.e. position *3f*). Then the valet will remove D from the parking free of charge (aka at no additional cost).
- then A can move right by 4, reach *3f*, and again let the valet remove A from the parking lot at no additional cost.
- so the lowest-cost solution is: $D\rightarrow\ 2; A\rightarrow\ 4$ with a cost of 2.

Example 3:

	a	*b*	*c*	*d*	*e*	*f*
1	.	.	.	G	F	.
2	.	.	B	G	F	.
3	A	A	B	C	F	.
4	.	.	.	C	D	D
5	.	.	.	C	.	.
6	.	.	E	E	.	.

- there is no solution

Example 4:

	a	*b*	*c*	*d*	*e*	*f*
1	.	.	.	.	F	.
2	.	.	B	.	F	.
3	A	A	B	C	F	.
4	.	.	.	C	.	.
5	.	.	.	C	.	.
6	.	.	E	E	.	.

- if all vehicles have a fuel level $\geq$ 3, then a solution would be: $E\leftarrow\ 1; C\downarrow\ 1; B\uparrow\ 1; F\downarrow\ 3; A\rightarrow\ 4$ with a cost of 5.
- if vehicle F has an initial fuel level $<$ 3, then there is no solution.

2 Implementation

2.1 State Space Search

Implement a solver for our version of the Rush Hour game in Python using state-space search. Your program should implement the following search algorithms:

1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. Algorithm A or $A^\star$ (A/$A^\star$)

For GBFS and $A/A^\star$, you must experiment with 4 different heuristics:

$\mathbf{h_1}$: The number of blocking vehicles.
For example, $h_1(\textit{Example } 1) = 1$, $h_1(\textit{Example } 2) = 1$, $h_1(\textit{Example } 3) = 3$.

$\mathbf{h_2}$: The number of blocked positions.
For example, $h_2(\textit{Example } 1) = 1$, $h_2(\textit{Example } 2) = 2$, $h_2(\textit{Example } 3) = 3$.

$\mathbf{h_3}$: The value of h_1 multiplied by a constant λ of your choice, where $\lambda > 1$.
For example, if you choose $\lambda = 5$, then $h_3(\textit{Example } 1) = 5$, $h_3(\textit{Example } 2) = 5$, $h_3(\textit{Example } 3) = 5$.

$\mathbf{h_4}$: Your own admissible heuristic.

.

2.2 Input

Your code should be able to receive an input file that contains initial puzzles (see the sample input file available on Moodle). The input file will contain a sequence of lines containing:

1. the initial puzzle represented by 36 characters, optionally followed by the initial fuel level of the vehicles (if no fuel level is indicated,assume the vehicle has a fuel level of 100), or
2. an empty line (useful for formatting) – you should skip that line, or
3. a line that starts with a # (useful for inserting comments) – you should skip this line also.

For example, the input file could contain:

`# this is an input file`	skip that line
`AAABCDFEEBCDF.RRC.GGH....IH.KK.IJJLL`	puzzle #1, all vehicles have a fuel of 100
`AAB...C.BHHHC.RRDF....DFEEGGG.......`	puzzle #2, all vehicles have a fuel of 100
`AAB...C.BHHHC.RRDF....DFEEGGG....... D0`	puzzle #3, same as puzzle #2, but D has no fuel
	an empty line
`#AAB...C.BHHHC.RRDD......EEGGGF.....F C3 B4 H1`	puzzle #4 that you should skip. Here, C's fuel is 3, B's fuel is 4, H's fuel is 1 and all other vehicles have a fuel level of 100

You can assume that the input file will represent valid puzzles (i.e. there will be one and only one ambulance, each vehicle will be of size at least 2, ...). However, there is no guarantee that the puzzles will be solvable.

2.3 Output

For each input puzzle and each search algorithm, your program should generate 2 output files: one for the solution path, and one for the search path. Sample output files are available on Moodle.

This means that for each line of the input file (see Section 2.2), your program should generate 18 text files:

Solution Files			Search Files		
`ucs-sol-#.txt`	`gbfs-h1-sol-#.txt`	`a-h1-sol-#.txt`	`ucs-search.txt`	`gbfs-h1-search-#.txt`	`a-h1-search-#.txt`
	`gbfs-h2-sol-#.txt`	`a-h2-sol-#.txt`		`gbfs-h2-search-#.txt`	`a-h2-search-#.txt`
	`gbfs-h3-sol-#.txt`	`a-h3-sol-#.txt`		`gbfs-h3-search-#.txt`	`a-h3-search-#.txt`
	`gbfs-h4-sol-#.txt`	`a-h4-sol-#.txt`		`gbfs-h4-search-#.txt`	`a-h4-search-#.txt`

where `#` is the puzzle number starting at one. For example, for puzzle #1 of the input file in Section 2.2, you should generate: `ucs-sol-1.txt`, `gbfs-h1-sol-1.txt`, `gbfs-h2-sol-1.txt`, ..., `a-h4-sol-1.txt` and `ucs-search-1.txt`, `gbfs-h1-search-1.txt`, `gbfs-h2-search-1.txt`, ..., `a-h4-search-1.txt`.

The content of the solution and the search files are described below.

2.3.1 Solution Files

Each solution file (eg. `ucs-sol-1.txt`) should contain details of the final solution found by the algorithm, or the string `no solution`. If a solution is found, the output file should contain:

1. the initial configuration of the puzzle and the fuel level of all vehicles
2. the execution time in seconds
3. the length of the search path (i.e. the number of states explored)
4. the length of the solution path (i.e. the number of moves in the solution)
5. the list of moves in the solution path
6. for each subsequent line: the vehicle to move, the direction of the move, the number of positions to move, the vehicle's fuel level after the move, and the new configuration of the puzzle after the move
7. the final configuration

For example:

```
Initial board configuration: BBBJCCH..J.KHAAJ.K..IDDLEEI..L....GG H3 K4 J2

B B B J C C
H . . J . K
H A A J . K
. . I D D L
E E I . . L
. . . . G G

Car fuel available: B:100, J:100, C:100, H:100, K:100, A:100, I:100, D:100, L:100, E:100, G:100

Runtime: 7.83 seconds
Search path length: 4269 states
Solution path length: 7 moves
Solution path: I down 1; D left 2; J down 1; C left 1; K up 1; J down 2; A right 3

I  down 1       99 BBBJCCH..J.KHAAJ.K...DDLEEI..L..I.GG
D  left 2       98 BBBJCCH..J.KHAAJ.K.DD..LEEI..L..I.GG
J  down 1       99 BBB.CCH..J.KHAAJ.K.DDJ.LEEI..L..I.GG
C  left 1       99 BBBCC.H..J.KHAAJ.K.DDJ.LEEI..L..I.GG
K    up 1        3 BBBCCKH..J.KHAAJ...DDJ.LEEI..L..I.GG
J  down 2        0 BBBCCKH....KHAA....DDJ.LEEIJ.L..IJGG
A right 3       97 BBBCCKH....KH...AA.DDJ.LEEIJ.L..IJGG

B B B C C K
H . . . . K
H . . . A A
. D D J . L
E E I J . L
. . I J G G
```

If your program cannot find a solution, then it should output:

```
no solution
```

2.3.2 Search Files

Each search file (eg. `ucs-search-1.txt`) should contain details of the the search path (the list of nodes that have been searched) by the algorithm. The files should contain 1 line for each node searched. Each line should contain the $f(n)$, $g(n)$ and $h(n)$ values of the node, followed by the configuration of the puzzle. For example:

```
15 10 5 BBBJCCH..J.KHAAJ.K...DDLEEI..L..I.GG
9 3 6 BBBJCCH..J.KHAAJ.K.DD..LEEI..L..I.GG
```

$f(n) = 15, g(n) = 10, h(n) = 5$, state = BBBJCCH..J.KHAAJ.K...DDLEEI..L..I.GG
$f(n) = 9, g(n) = 3, h(n) = 6$, state = BBBJCCH..J.KHAAJ.K.DD..LEEI..L..I.GG

If the value of $f(n)$, $g(n)$ or $h(n)$ are irrelevant for a search algorithm, just display their value as zero (`0`).

2.4 Programming Environment

To program the project:

1. you must use Python 3.8 or above
2. you must use GitHub (make sure your project is private while developing)

3 Analysis

Once your code is running, you will analyse and compare the performance of the algorithms and the heuristics. Generate a file with 50 random puzzles, then use this file as input to all your algorithms & heuristics. In a spreadsheet, compile the following data:

Puzzle Number	**Algorithm**	**Heuristic**	**Length of the Solution**	**Length of the Search Path**	**Execution Time (in seconds)**
1	UCS	NA	48	123	1.56
1	GBFS	h1	48	123	1.56
1	GBFS	h2	48	123	1.56
1	GBFS	h3	48	123	1.56
1	GBFS	h4	48	123	1.56
1	A/A*	h1	48	123	1.56
1	A/A*	h2	48	123	1.56
1	A/A*	h3	48	123	1.56
1	A/A*	h4	48	123	1.56
2	UCS	NA	48	123	1.56
2	GBFS	h1	48	123	1.56
...	...	...	...	...	...
50	A/A*	h4	50	123	1.56

Compare and analyse:

1. The length of the solutions across algorithms and heuristics. When do you have the lowest-cost solution?
2. The admissibility of each heuristic and its influence on the optimally of the solution.
3. The execution time across algorithms and heuristics. Is an informed search always faster?
4. Other interesting facts that you deem worthy of describing.

You will present these slides at the demo (see Section 5).

4 Competition (Just for fun)

Just for the fun of it, we will organize a competition to compare the solutions found by different teams with the same set of input puzzles. This competition will not count for points, we are just doing this for fun (and for the thrill of having your team publicly acknowledged on the Moodle page as the winner of the competition!).

For the competition, we will give all teams a file with puzzles, and teams will be ranked based on the total cost of the solutions and execution time. More details on the competition will be posted on Moodle.

5 The Demo

You will have to demo your project to the TAs. Regardless of the demo time, you will demo the program that was uploaded as the official submission on or before the due date. The schedule of the demos will be posted on Moodle. The demos will last 15 minutes and will consist in 4 parts:

1. **Presentation of your code (≈ 2min)** Explain the main functions and data structures of your code to the TA. Show your actual code for this.
2. **Presentation of your analysis (≈ 4min)** Explain your heuristic and present the analysis of your all your results (see Sections 3). Use a few slides for this.
3. **Q/A (≈ 7min)** Your TA will ask each student questions on the code or on the analysis. Only the student asked is expected to answer – this is not a "team effort". Hence every member of team is expected to attend the demo. No special preparation is necessary for the Q/A.
4. **Sample Run (≈ 2min)**

   At the end of the demo, your TA will give you a a small input file of puzzles and ask you to run your code with this data. The output generated by your program will have to be uploaded on EAS.

6 Evaluation Scheme

Students in teams can be assigned different grades based on their individual contribution to the project.

Individual grades (10%) will be based on:

1. a peer-evaluation done after the deadline.
2. the contribution of each student as indicated on GitHub.
3. the Q/A of each student during the demo (see Section 5).

Team grades (90%) will be based on:

Implementation (see Section 2)	**55%**
functionality of each search algorithm	35%
format of the output files	2.5%
heuristic h_4: originality, usability for $A^\star$ ...	5%
design & programming style	2.5%
other functionality	10%
Analysis (see Section 3)	**35%**
comparison of the heuristics (informedness, admissibility)	10%
comparison of the algorithms	10%
other interesting analysis	10%
clarity of slides, presentation skills, time-management	5%
Total	**100%**

7 Submission

If you work in a team, identify one member as the team leader. The only additional responsibility of the team leader is to upload all required files from their account and book the demo on the Moodle scheduler.

Submission Checklist

on GitHub:

☐ In your GitHub project, include a `README.md` file that contains on its first line the URL of your GitHub repository, as well as specific and complete instructions on how to run your program.

☐ November 27, make your GitHub repository public.

on EAS:

☐ Create a PDF of your slides, and name your slides `472-Slides-ID1-ID2-ID3.pdf` where `ID1` is the ID of the team leader.

☐ Create one zip file containing all your code, your file of 50 random puzzles, the corresponding output files and the `README.md` file. Name this zip file `472-Code-ID1-ID2-ID3.zip` where `ID1` is the ID of the team leader.

☐ Zip `472-Slides-ID1-ID2-ID3.pdf` and `472-Code-ID1-ID2-ID3.pdf` and name the resulting zip file: `472-Assignment-ID1-ID2-ID3.zip` where `ID1` is the ID of the team leader.

☐ Have the team leader upload the zip file at: https://fis.encs.concordia.ca/eas/ as as Programming Assignment Group 2 / Submission Number 1

Have fun!