Deadline:	2016-10-07 23:59
Recitation (øvingsforelesning):	2016-09-27 18:15, S2

Using the A* Algorithm

Purpose: Gain hands-on experience with best-first search using the A* Algorithm.

Overview

In this assignment, you will become familiar with the A* algorithm by applying it to two different problems. The first problem is a classical use case for A*, namely that of finding the shortest path in a two-dimensional grid-like world. The second problem concerns using A* to solve a puzzle, in this case the puzzle game *Rush Hour*.

The first problem consists of three subproblems that give you one point each, for a possible total of three points. The second problem gives you one point. The assignment thus contains four possible points in total. **You need 3 out of 4 points in order to pass this assignment.**

Each of the four problems specifies a set of required deliverables. All problems include both programming and report writing. The main purpose of the report is for you to present your results so that the student assistants don't have to run your code to evaluate your assignment.

A* Implementation

In order to solve the problems in this assignment, you first need to obtain an implementation of the A* algorithm by either a) writing it from scratch in the language of your choice, or b) downloading it from the internet. It is strongly recommended that you write the code yourself, since this will provide you with an in-depth understanding of this important AI algorithm. The accompanying document entitled "Essentials of the A* Algorithm" can be of assistance.

Should you choose to download a version of A*, then you will not receive much (if any) assistance from the course instructor or assistants, unless you fortuitously download code with which they are familiar. You cannot expect them to spend a lot of time trying to learn it.

Whether you implement or download the code, you must include it (along with appropriate comments) in your hand-in on It's Learning.

Problem A: Pathfinding in 2D Games

A common demonstration problem for A* is that of finding shortest paths in two-dimensional square-grid boards. Imagine for example a two-dimensional top-down perspective video game—in such games, you might move your character around by clicking on locations in the game board to which you want to move. The game would then calculate a specific path—a sequence of adjacent cells—for the character to follow to get to that location. This is called *pathfinding*, and is a common use case for the A* algorithm.

See Figure 1 for an example of such a 2D game board. The goal is to find the shortest path from the square marked A (top left) to the square marked B (bottom right). The gray cells represent obstacles/walls which have to be avoided by the algorithm. Assuming the game character can only move in the four cardinal directions—north, south, west, east—the shortest path will be as shown in Figure 2.

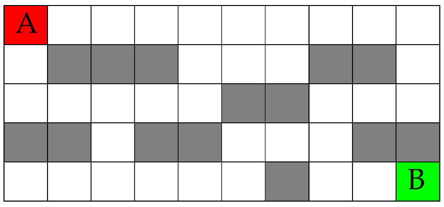


Figure 1: Example of pathfinding problem in a 2D game world. The goal is to get from A to B while avoiding the gray obstacles. The character can move north, south, west and east.

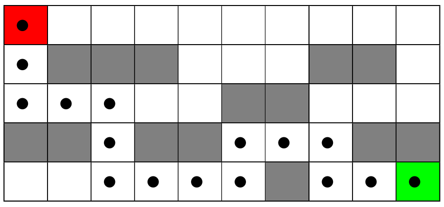


Figure 2: A solution to the pathfinding problem in Figure 1.

For this problem, you will implement a program that reads in a board configuration—a text file describing where the obstacles are located and where the start (A) and goal (B) squares are—and then finds the shortest path from the start to the goal using the A* algorithm.

This problem consists of three parts. In the first part (Subproblem A.1), the game board will only consist of open cells or blocked cells—as seen in the example above. In the second part (Subproblem A.2), the cells will have different *costs* attached to them. In other words, the shortest path will no longer be determined merely by the number of cells traversed, but by the *sum of the costs* of the cells traversed. In the final part of the problem (Subproblem A.3), you will experiment with using Breadth-First Search (BFS) and Dijkstra's Algorithm instead of the A* algorithm and examine how this affects the outcome of the search.

In all three parts of the problem, you will be asked to visualize your program's results. The specifics of the visualization are up to you, but the board configuration and the calculated path should be clearly visible. The visualization can for example be in the form of tables (as above), text or simple 2D images. A possible textual representation of the path found in Figure 2 is shown in Figure 3, while an image-based representation is shown in Figure 4. (The image was drawn using the *Python Imaging Library*, which is a good choice if you're using Python for this assignment.)

```
o.........
o###...##.
ooo..##...
##o##ooo##
..oooo#ooo
```

Figure 3: A textual representation of the path found in Figure 2.

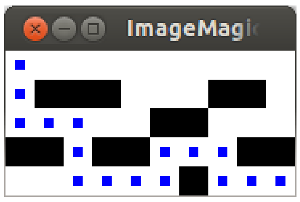


Figure 4: An image-based representation of the path found in Figure 2.

Subproblem A.1: Grids with Obstacles (1 point)

Subproblem A.1 of this assignment is thus to find shortest paths in grids with obstacles. Using your A* implementation, search for a path from the start cell to the goal cell. For a given state in the A* search, i.e. a given cell on the board, the successor states will be the adjacent cells to the north, south, west and east that are within the boundaries of the board and that do not contain obstacles. Suggestions for the heuristic function $h()$ for this problem are to calculate either the Manhattan distance or the Euclidean distance between the current cell and the goal cell.

Your program should solve the following four boards:

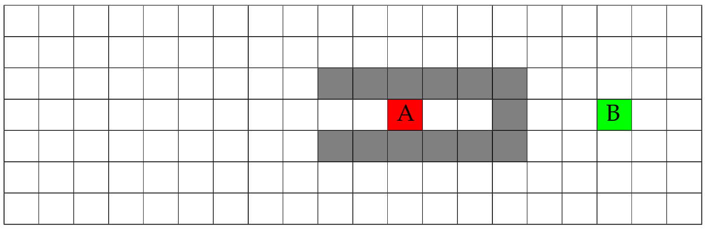


Figure 5: Board file `board-1-1.txt` from the supplementary files on It's Learning.

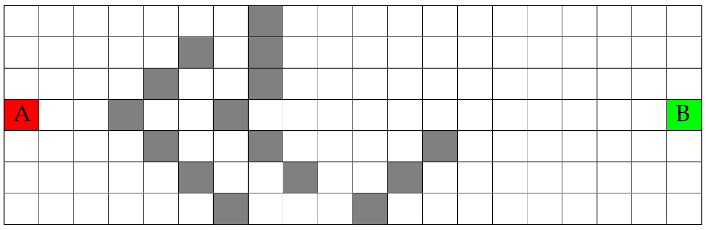


Figure 6: Board file `board-1-2.txt` from the supplementary files on It's Learning.

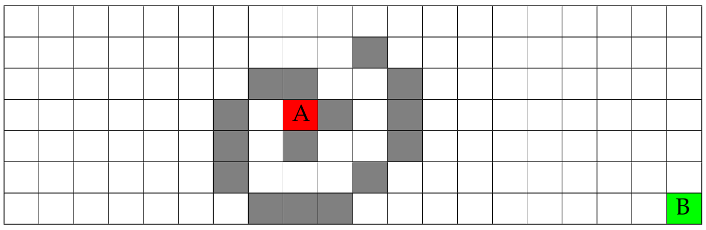


Figure 7: Board file `board-1-3.txt` from the supplementary files on It's Learning.

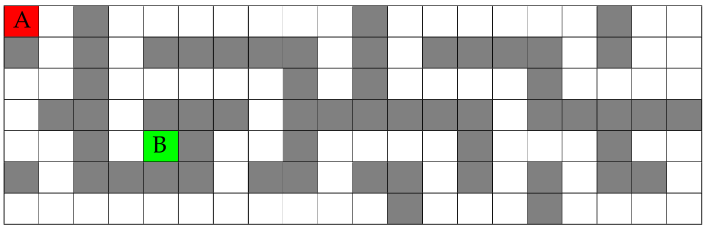


Figure 8: Board file `board-1-4.txt` from the supplementary files on It's Learning.

Deliverables

- **A.1.1:** Well-commented source code for a program that finds shortest paths for boards with obstacles and that visualizes the results. If you did not write the A* implementation youself, specify where you got the code from.
- **A.1.2:** Visualizations of the shortest path for the four boards given above.

Subproblem A.2: Grids with Different Cell Costs (1 point)

Consider a game where the squares on the game board have different *costs* attached to them—for example, in a game where the board represents an outdoor environment, different squares could contain different kinds of landscape such as forest, mountains, grasslands, etc. Walking across a square of mountains would naturally take a longer time than a square of grasslands. Thus, a mountain square should have a higher cost attached to it than a grasslands square.

In this part of the assignment, your code from subproblem A.1 will be extended to take different cell costs into account in the pathfinding process. Table 1 specifies the cell types that should be supported, while Figure 9 shows an example of a game board where these cell types are used. The shortest path, i.e. the path with the lowest cost, is indicated in the figure.

Table 1: Cell types and their associated costs.

	CHAR.	DESCRIPTION	COST
🟦	`w`	Water	100
⬜	`m`	Mountains	50
🟩	`f`	Forests	10
🟩	`g`	Grasslands	5
🟫	`r`	Roads	1

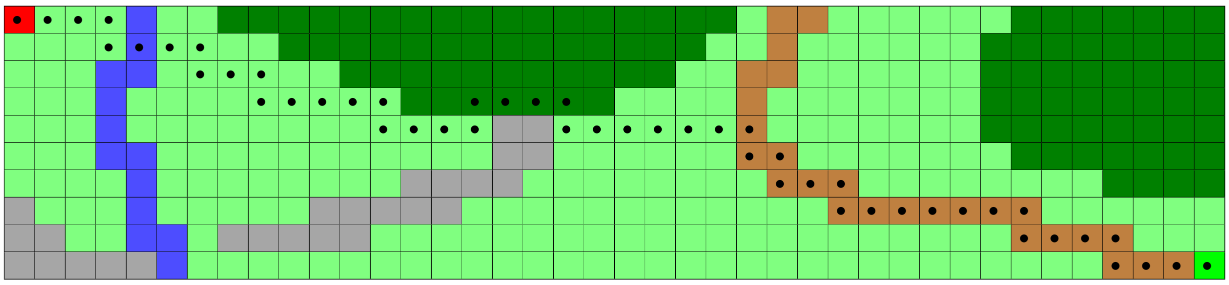


Figure 9: Example of a game board with different cell costs, with the cheapest path indicated.

Your program should solve the following four boards:

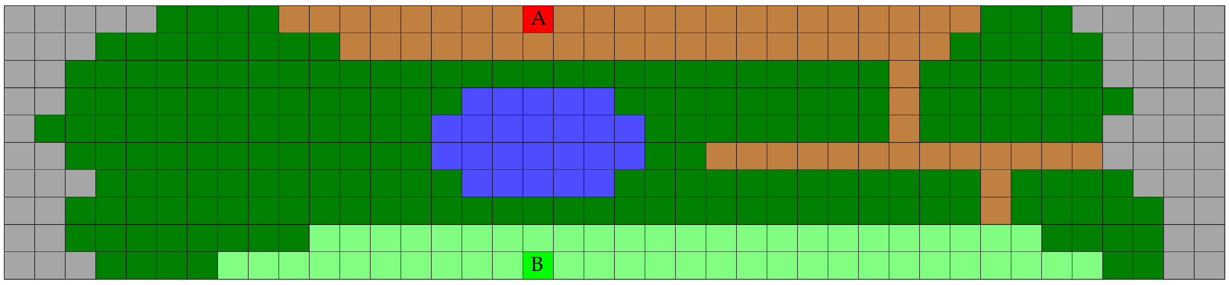


Figure 10: Board file `board-2-1.txt` from the supplementary files on It's Learning.

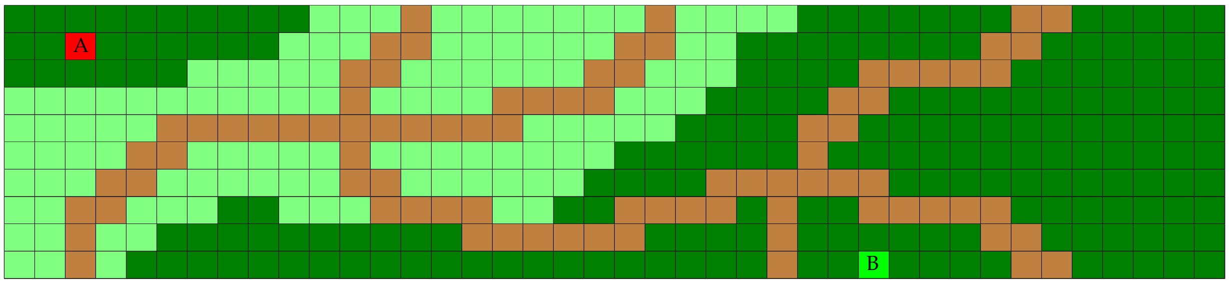


Figure 11: Board file `board-2-2.txt` from the supplementary files on It's Learning.

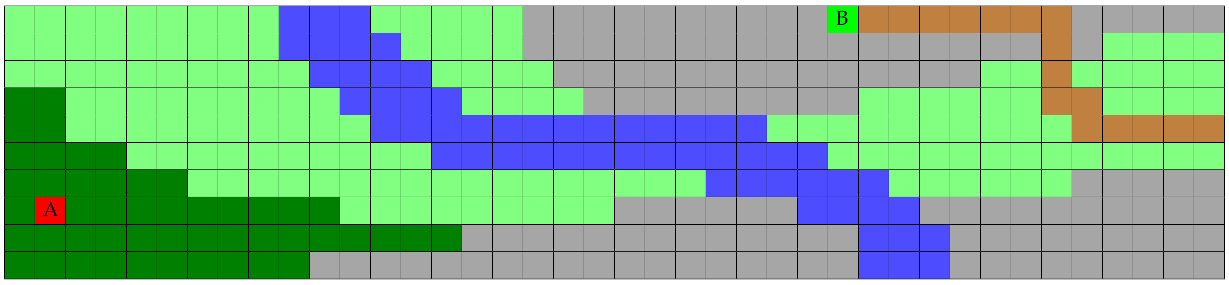


Figure 12: Board file `board-2-3.txt` from the supplementary files on It's Learning.

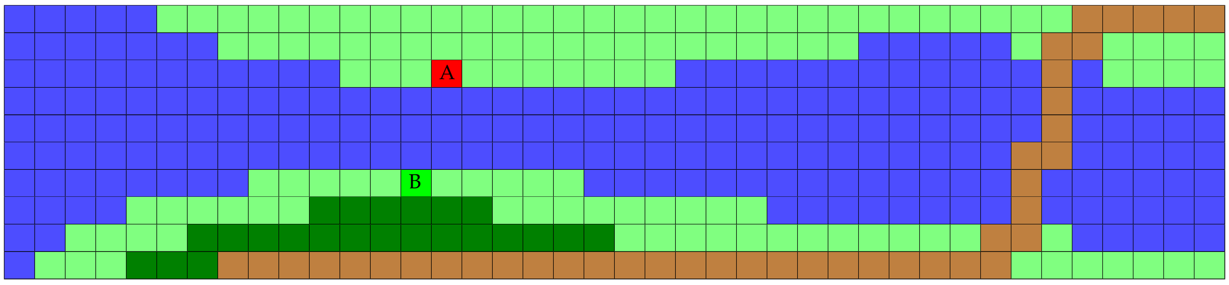


Figure 13: Board file `board-2-4.txt` from the supplementary files on It's Learning.

Deliverables

- **A.2.1:** Well-commented source code for a program that finds shortest paths for boards with different cell costs (as specified in Table 1) and that visualizes the results.
- **A.2.2:** Visualizations of the shortest path for the four boards given above.

Subproblem A.3: Comparison with BFS and Dijkstra's Algorithm (1 point)

The A* algorithm can, with a few changes, be modified to instead implement Breadth-First Search (BFS) or Dijkstra's Algorithm. With A*, the open nodes are sorted according to their expected cost $f(s) = g(s) + h(s)$ (see the accompanying document "Essentials of the A* Algorithm"). By maintaining the list of open nodes as a queue (first-in first-out) instead of as a priority queue, the algorithm becomes BFS. By sorting the open nodes according to only $g(s)$, the algorithm becomes Dijkstra's Algorithm.

In this part of the assignment, you will search for paths from A to B in the previously completed game boards using BFS and Dijkstra's Algorithm. The visualizations should be presented in your report, along with brief analyses of the differences between A*, BFS and Dijkstra in each case.

In addition to showing the calculated paths, your visualizations should also show which nodes belong to the open list and which nodes belong to the closed list. This applies to your visualizations for all three algorithms (A*, BFS, Dijkstra). Your analyses should also briefly discuss any differences seen in the number of open/closed nodes between the different algorithms.

For an example of such a visualization, see Figure 14, which shows the board from Figure 9 with the open nodes marked with stars ($\star$) and the closed nodes marked with crosses ($\times$).

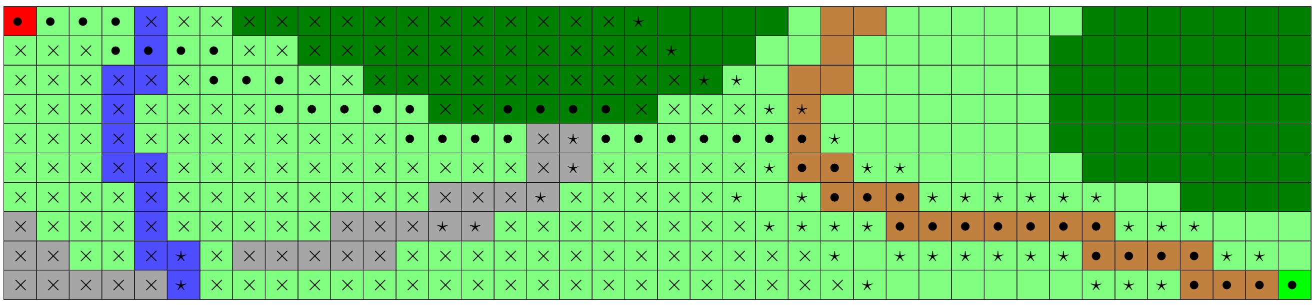


Figure 14: The game board from Figure 9 with open and closed nodes marked ($\star$ and $\times$, respectively).

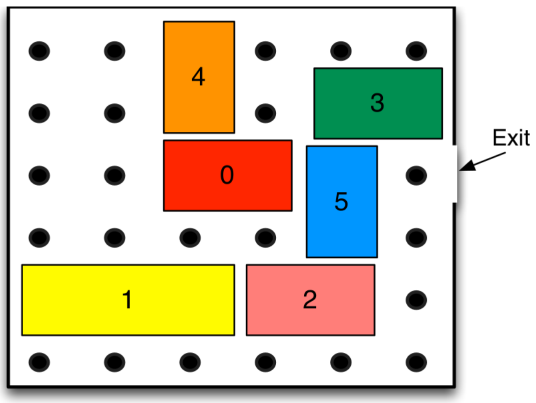
Deliverables

- **A.3.1:** Well-commented source code for a program that finds paths using A*, BFS and Dijkstra's Algorithm and that produces visualizations with open and closed nodes, such as the one above.
- **A.3.2:** Visualizations using A*, BFS and Dijkstra for each of the game boards processed previously. This should give 12 or 24 visualizations in total, depending on whether or not you did both A.1 and A.2 previously and thus have respectively four or eight game boards to process.
- **A.3.3:** For each game board, a brief analysis of a) any differences in the path found by A*, BFS and Dijkstra, and b) any interesting differences in the number of open and closed between for the different algorithms.

Problem B: Rush Hour Puzzle (1 point)

Rush Hour is a popular puzzle (and also a free App from the Apple Store). The standard board is 6×6, with the pieces having dimensions 2×1 (cars) and 3×1 (trucks). One special car (car-0) needs to get to a particular spot on the edge of the board, i.e., the exit. Most (if not all) other cars and trucks must be moved to enable car-0 to reach this goal. Each vehicle is positioned either horizontally or vertically, and it can only move along that row or column, respectively, throughout the solution sequence. So a car or truck positioned horizontally along row 3 can only move back and forth within row 3.

Figure 15 shows a typical scenario, with 5 cars and one (yellow) truck. The red vehicle (car 0) needs to get to the exit in as a few moves (of car 0 and all other vehicles) as possible. Optimal solutions are those involving a minimum amount of total vehicle movement. A move is thus any one-cell translation of a vehicle. So if car-0 in Figure 15 drives from (2,2) to (0,2), that counts as 2 moves. Vehicles 0 - 3 can only move horizontally, while vehicles 4 and 5 move only vertically. The grid is 6 x 6, with dots marking the center of each grid cell. No part of any vehicle can move beyond the square boundary.



Scenario
Specification

0, 2, 2, 2
0, 0, 4, 3
0, 3, 4, 2
0, 4, 1, 2
1, 2, 0, 2
1, 4, 2, 2

Figure 15: A simple Rush-Hour puzzle (left) and its corresponding text specification (right). Small circles denote the centers of each grid cell; the upper left circle, the origin, has coordinates (0,0).

For our purposes, the origin of this 6×6 discrete cartesian plane is in the upper left. So the upper left cell has coordinates (0,0), while the lower right cell has coordinates (5,5). As shown in Figure 15, a scenario is described by a set of quads, one for each vehicle. The 4 elements of a quad, (O, X, Y, S), denote the orientation, horizontal (0) or vertical (1)), the coordinates (X and Y) of the upper left portion of the vehicle, and its size (S). A specification with K vehicles consists of K quads, listed in order of the vehicle number; the quad for car-0 is always first. In the figure, the quad for car 3 is (0,4,1,2), indicating that it is oriented horizontally and that the leftmost cell that it currently occupies is (4,1) - 4 moves over and 1 move down from the origin. Finally, the 2 represents the size of car 3: it spans 2 grid cells.

For all examples given in this homework exercise, the exit is at the same location (5,2) as shown in Figure 15, and all vehicles have size 2 or 3. Solving the puzzle entails finding a combination of moves that enables the right half of car-0 to cover location (5,2); from there it can obviously move out of the traffic jam.

Puzzle Variants

You will need to run A* on the 4 scenarios of Table 2. These constitute easy, medium, hard, and expert-level puzzles found in the free Rush-Hour App.

Table 2: Specification quads for four different Rush-Hour scenarios. The name of each scenario indicates its source in the free Rush-Hour App. For example, Hard-3 denotes the 3rd scenario in the set of puzzles that the App classifies as Hard. For all scenarios, car-0 is the vehicle that must drive to the exit, location (5,2), in order to complete the puzzle.

VEHICLE	EASY-3	MEDIUM-1	HARD-3	EXPERT-2
0	(0,2,2,2)	(0,1,2,2)	(0,2,2,2)	(0,0,2,2)
1	(0,0,4,3)	(0,0,5,3)	(0,0,4,2)	(0,0,1,3)
2	(0,3,4,2)	(0,1,3,2)	(0,0,5,2)	(0,0,5,2)
3	(0,4,1,2)	(0,3,0,2)	(0,2,5,2)	(0,1,0,2)
4	(1,2,0,2)	(1,0,2,3)	(0,4,0,2)	(0,2,3,2)
5	(1,4,2,2)	(1,2,0,2)	(1,0,0,3)	(0,3,4,2)
6		(1,3,1,2)	(1,1,1,3)	(1,0,3,2)
7		(1,3,3,3)	(1,2,0,2)	(1,2,4,2)
8		(1,4,2,2)	(1,3,0,2)	(1,3,0,3)
9		(1,5,0,2)	(1,4,2,2)	(1,4,0,2)
10		(1,5,2,2)	(1,4,4,2)	(1,4,2,2)
11			(1,5,3,3)	(1,5,2,2)
12				(1,5,4,2)

Deliverables

- **B.1:** Well-commented source code for a program that solves the Rush Hour puzzle. If you did not write the A* implementation youself, specify where you got the code from.
- **B.2:** A clear, concise description (using mathematical expressions and text) of the heuristic function (h) used to solve the puzzle.
- **B.3:** A thorough description of the procedure used to generate successor states when expanding a node.
- **B.4:** Overview description of the solutions found by A*. The number of moves required, i.e. the number of nodes on the path from start to goal, must be presented, as well as a listing of the entire sequence of state transitions/actions from the start to the goal node.

  This should be done for each puzzle variant. If some of your solutions are not optimal, it is fine to just provide the best solutions that A* could find. Some of the puzzle variants are harder than others.