Artificial Intelligence Laboratory 2: A*(A star) Search Algorithm

DT8012, Halmstad University

November 21, 2016

Introduction

This lab is designed to introduce you to search algorithms, using one of the most common ones, namely A*. You will implement the A* algorithm, and apply it in two domains:

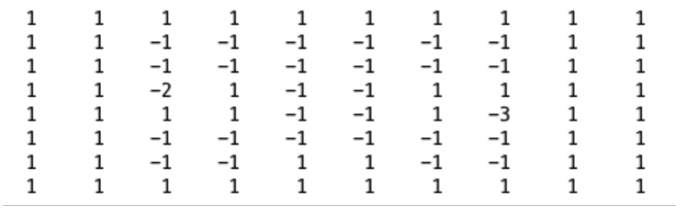
- Path planning: find the shortest path from agent's current position to the goal.
- Poker game: find a sequence of bids that maximizes your winnings (given a known, deterministic strategy for an opponent).

Implementation of A*, like many similar graph algorithms, is quite simple in a high level programming language such as Python (we recommend this tutorial and Amits pages on A* algorithm).

Task 1: A* Path Planning

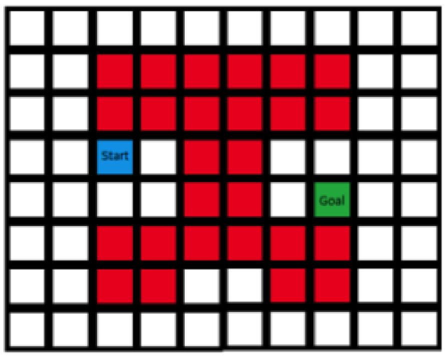
In this task you will implement the A* algorithm and use it for path planning problem. The task is to find a shortest path from a starting position to the goal position within a two-dimensional grid, without passing through any obstacles. In each position you have (up to) four possible actions: up $\uparrow$, right $\rightarrow$, down $\downarrow$ and left $\leftarrow$. The map is represented as a 2D matrix. Figure 1 shows an example of how it can look like. The '-1's represent obstacles (or impassable cells), '-2 represents the starting position and '-3 represents the goal and the numbers in the rest of cells are the cost for moving into them. In the case of Figure 1, all accessible cells cost 1 to move into. In this example, agent has to move from (4, 3) to (5, 8).
In the lab2.zip file you can find a library in python that does the following:

- **generateMap(...)**: generates a map of given size, with randomly placed obstacles.
- **generateMap_obstacle(...)**: generates a map with a special, H-shaped impassable obstacle (as shown above).



```
1   1   1   1   1   1   1   1   1   1
1   1  -1  -1  -1  -1  -1  -1   1   1
1   1  -1  -1  -1  -1  -1  -1   1   1
1   1  -2   1  -1  -1   1   1   1   1
1   1   1   1  -1  -1   1  -3   1   1
1   1  -1  -1  -1  -1  -1  -1   1   1
1   1  -1  -1   1   1  -1  -1   1   1
1   1   1   1   1   1   1   1   1   1
```

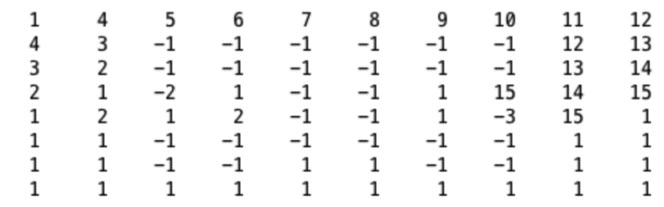
(a) Matrix representation of the grid map



(b) Grid map with obstacles

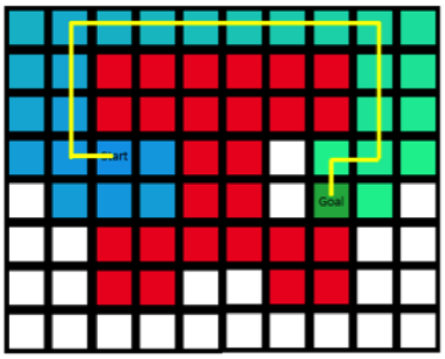
Figure 1: An example of the grid map generated by the library provided

- **generateMap_swamp(...)**: generates a map with a swamp in the middle. Swamp can be passed through, however, doing so is slow, so it is often better to walk around it.
- **plotMap(...)**: plots the map together with additional information. Use it to visualise and understand the results of your A* algorithm. It will colour map cells according to a Value you provide, and it also draws the path found. You can use this function to analyse the behaviour of your algorithm. In particular, make sure that you plot at least the following as the values for the cells:
  - Value of the heuristic h(x)
  - Cost of moving from the starting point to this cell g(x)
  - Total cost f(x) (movement cost + heuristic value)
  - Expansion order (which nodes in the graph were evaluated first);



```
1   4   5   6   7   8   9  10  11  12
4   3  -1  -1  -1  -1  -1  -1  12  13
3   2  -1  -1  -1  -1  -1  -1  13  14
2   1  -2   1  -1  -1   1  15  14  15
1   2   1   2  -1  -1   1  -3  15   1
1   1  -1  -1  -1  -1  -1  -1   1   1
1   1  -1  -1   1   1  -1  -1   1   1
1   1   1   1   1   1   1   1   1   1
```

(a) Matrix representation of the grid map



(b) Grid map with obstacles

Figure 2: An example of the grid map with optimal path and heuristic values shown

Task 1

a. Use **generateMap(...)** to generate a grid map with obstacles randomly distributed. Implement A* that finds the shortest path, if it exists, from start point to end point. Test your algorithm on a number of random size maps, with randomly placed obstacles as well as a start and goal points.

b. Try different heuristics (at least Euclidean and Manhattan distances). Compare and analyse how well do they work (hint: visualise heuristic value using function **plotMap(...)**). Is the path found by A* optimal in both cases? Which of them is more efficient? How is this efficiency measured? How many nodes are expanded when using different heuristics?

c. Use **generateMap2d_obstacle(...)** or **generateMap2d_swamp(...)** to generate a map with some predefined obstacles. Find an optimal path from the starting point to the ending point using A* algorithm in this new setting, and make sure it is still finding the optimal path.

d. Design new heuristic(s), specific for this particular the scenario, which will work better than general-purpose ones you have used before. Make sure they still lead to optimal paths, but also that the algorithm will expand less nodes. You can use the following information for designing your heuristics:

- For maps generated by **generateMap_obstacle(...)** (Fig. 3), the starting point is always somewhere on the left side of the map and goal is on the right side of the map. The environment contains a rotated-H shaped obstacle. Your agent can only pass near the top edge of the map or near the bottom edge of the map. However, you can deduce which way is better based on the start and goal positions, without doing any search.

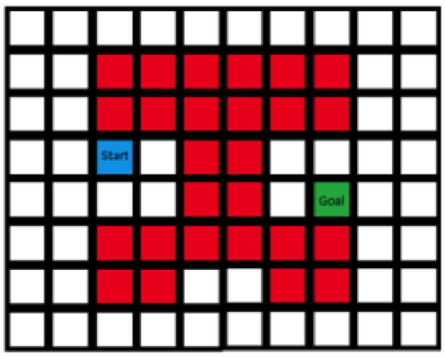


Figure 3: Map generated using function **generateMap_obstacle(...)**

- Maps generated by **generateMap_swamp(...)** contain a swamp that cost extra effort to move into. The swamp is consisted of three repulsive fields around their corresponding centers. You can utilise the center position of each repulsive fields to design your heuristic function.

You only need to come up with better heuristic for one of the two maps, either

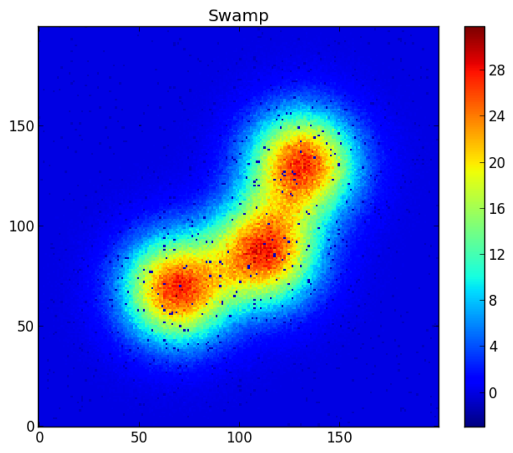


Figure 4: Map generated using function **generateMap_swamp(...)**

the one generated by generateMap_obstacle(...) or generateMap_swamp(...). However, you can earn some extra credit if you do both.

Task 2: Poker Bidding

In this task, you will use A* algorithm to plan sequence of actions that beat your opponent in a poker game (rules are more complicated than the one in first lab, but still simpler than the real game). The goal is to win as much of your opponent's money as possible. Definitions of some concepts involved in this poker game are listed and explained as follows:

- Hand: Starts with the dealing phase and ends after the showdown.
- Bet: to bet, a player can bet any amount of money (regardless of how much other players bet). Player can not bet more money than they have.
- Pot: pot refers to the sum of money that players wager during a single hand. The winner of the hand gets all money in the pot.
- Call: in this poker game, the player can call by putting 5 coins into the pot. If any player performs 'Call' action, the game directly proceed into showdown phase, i.e. all player show hand and determine the winner based on the strength of their hand. (Note that this 'Call' action is different from the one in traditional poker games.) It's allowed to 'Call' as long as player possesses coin(s) (can be less than 5, in this case 'Call' uses all coins left), but not below zero.
- Fold: the player does not accept the bet. The pot goes to the other player. There is no need to show the cards.

Opponent Actions	**Player Responses**
Bet	Bet, Call, Fold
Call	Proceed to showdown
Fold	Player wins this hand

Table 1: Responses to opponent's actions

The rules of the game are stated as follows:

- There will be two players playing against each other within the game.
- Every agent has 100 coins to bet and if one agent loses all the coins, the game ends.
- There will be (maximum) n number of hands each game (you decide n).
- Each hand includes the following flow:
  a. **Card dealing phase**: assigning a hand (**five** cards) to each agent.
  b. **Bidding phase**:
     a) Bidding phase always starts with your agent. There are several actions available: {Bet x coins, Call, Fold}, (x could be drawn from a set of numbers, e.g. $\{5, 10, 25\}$.
     b) After your agent action, your opponent acts based on your action and the current state of the game. In this lab, opponent's action is given, by calling the function **pokerStrategyExample(...)**, with correct inputs (see Table 2 for details).
     c) Based on opponent's action, available actions for player to perform are {Call, bet x coin, Fold}. A general illustration of player's actions and possible responses are listed in Table 1.
     d) The bidding phase ends when any player performs 'Call' or 'Fold'.
  c. Showdown phase: after bidding phase, both agent show their hands and the agent with stronger hands gets the pot (If one of the players folds, there is no need to show the cards).
- The game ends (when) one player loses all the money.

You are required to implement the flow of the game and hand evaluation function. The strategy of your opponent is given, check **pokerStrategy.py** within **lab2.zip**, in which, function **pokerStrategyExample(...)** returns your opponent's strategy. Table 2 explains the input of this function required.

Task 2

Information/input	**Details**
playerAction	Bet, Call or Fold
playerActionValue	the amount of coins used to Bet or Call
playerStack	total amount of coins your agent have
agentHand	current opponent's hand type
agentHandRank	current opponent's hand rank
agentStack	total amount of coins your opponent have

Table 2: Responses to opponent's actions

a. Build the environment of the game and determine important parameters: 1) number of hands each game; 2) amount of money for betting ({5, 10, 25} is recommended). Following functions are required for this poker game.

   - Hand identification function that evaluates the hand strength. Note that you also need to provide hand strength for your opponent, check pokerStrategyExample(...), read the code and make sure you provide the right input format.
   - Function for setting the state of the game, i.e. continuously updates a set of values: number of hands played, current hand for both agent, coins left for both agent, coins in pot, actions performed etc.

b. Implement two fixed agents play against each other, make sure the flow of the game is implemented correctly.

c. Given complete information of the game, use your A* implementation to generate a series of actions that maximize your player's winning, against an agent with known strategy provided.

   - Generate a tree that contains all possible states. (hint: expand all possible actions that can be performed by your player.)
   - Set heuristic to zero (the search algorithm become a breadth-first search) and compute the optimal sequence of actions.
   - The overall goal is to win all your opponent's money while playing as few hands as possible.
   - The game starts with each agent owning 100 coins and ends with one agent losing all the money.

d. Propose one or more heuristics (does not have to be admissible) to reduce the search space of your implementation. Explain how and why it works or why it does not work.

Grading Criteria

- Pass: Complete task 1a - 1d (1d: solve one of the map given) and 2a - 2c.
- Deadline is 2 weeks starting from the introduction session.
- Your submission should include **code** and a **short report** of what you have done, observed and learnt.
- Extra credits: Complete task 1d (both maps) and 2d.