Artificial Intelligence Laboratory 2: Search Algorithms

DT8012, Halmstad University

November 14, 2017

Introduction

This lab is designed to introduce you to several search algorithms (random, BFS, DFS, greedy search and A/A* algorithm with customized heuristics). You will implement different search algorithms and apply them in two domains:

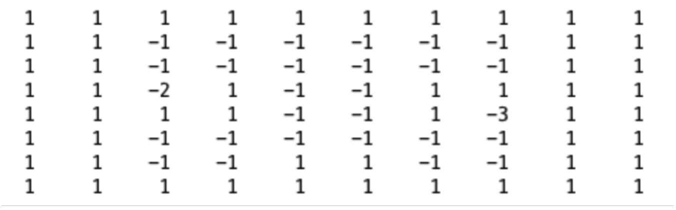
- Path planning: find the shortest path from agents current position to the goal.
- Poker game: find a sequence of bids for your agent to win more than 100 coins from the opponent within 4 hands (given a known, deterministic strategy for an opponent).

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). Additionally, you will compare different search algorithms, but they all follow the same code "skeleton" and only require using different cost functions.

Task 1: Path Planning

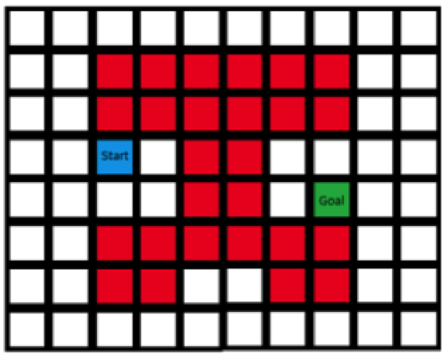
In this task you will implement search algorithms and use them for path planning problem. The task is to find a shortest path from a starting position to the goal position within a two-dimensional grid, without passing through any obstacles. In each position you have (up to) four possible actions: up ↑, right →, down ↓ and left ←. 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 'path_planning.py' that does the following:

- **generateMap(...)**: generates a map of given size, with randomly placed obstacles.



```
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
```

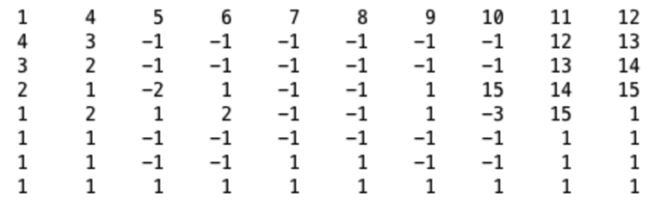
(a) Matrix representation of the grid map



(b) Grid map with obstacles

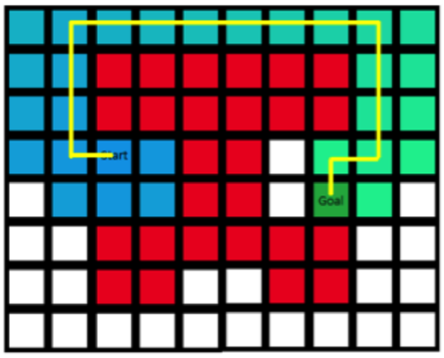
Figure 1: An example of the grid map generated by the library provided

- **generateMap_obstacle(...)**: generates a map with a special, H-shaped impassable obstacle (as shown above).
- **plotMap(...)**: plots the map together with additional information. Use it to visualise and understand the results of your search 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 three uninformed search algorithms (random search, BFS and DFS) that find the shortest path, if it exists, from start point to end point. An skeleton of search algorithm is provided (check function **search(...)** in **search_algorithm.py**). Test your algorithm on a number of random size maps, with randomly placed obstacles as well as a start and goal points.

b. Come up with different heuristics (at least Euclidean and Manhattan distances) and test them using informed search algorithms (greedy search and A*). 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 the two heuristics is more efficient (in terms of how many nodes are expanded when using different heuristics)?

c. Use **generateMap2d_obstacle(...)** to generate a map with 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 following information for designing your heuristics:

   - 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. y coordinate of the two horizontal wall and x coordinate of the vertical wall are available (check function **generateMap_obstacle(...)** in **path_planning.py**). 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.

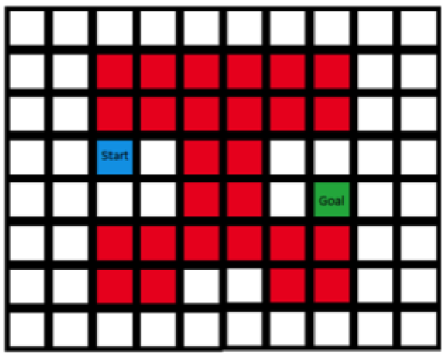


Figure 3: Map generated using function **generateMap_obstacle(...)**

Task 2: Poker Bidding

In this task, you will use search 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 more than 100 coins from your opponent within 4 hands. 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 put a certain amount of money into the pot. 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 paying 5 coins. 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.).
- Fold: the player does not accept the bet. The pot goes to the other player. There is no need to show the cards.

The rules of the game are stated as follows:

- There will be two players play against each other within the game.
- Every agent has 150 coins to bet and if one of the agent wins more than 100 coins, the game ends.
- There will be (maximum) 4 hands each game.
- 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: {Call, Bet 5 coins, Bet 10 coins, Bet 25 coins, Fold}.
     b) After your agent's action performed, your opponent acts based on your action and the current state of the game. In this lab, opponent's action is given, by calling function **poker_strategy_example(...)**, with correct input (see Table 2 for details).

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

Table 1: Responses to opponent's actions

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 after 4 hands or when one player wins at least 100 coins.

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 **poker_strategy_example(...)** returns your opponent's strategy. Table 2 explains the input of this function required.

Information/input	**Details**
opponent_hand	type of opponent's hand
opponent_hand_rank	rank of opponent's hand
opponent_stack	total amount of coins opponent has
agent_action	agent's action: Bet, Call or Fold
agent_action_value	the amount of coins agent used to Bet or Call
agent_stack	total amount of coins the agent has
current_pot	total amount of coins currently in bidding
bidding_nr	numbers of times both player has bidded

Table 2: Responses to opponent's actions

Task 2

a. Build the environment of the game. 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 **poker_strategy_example(...)**, 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 search algorithms to find a series of actions that your agent wins more than 100 coins **within 4 hands**.

- Generate a tree that contains all possible states for a hand (hint: expand all possible actions that can be performed by your player). Make sure the tree is generated correctly, you can visualize and an example is given in **graph_visualization.py**.
- Use random search (maximum 10000 steps) and exhaustive search (breadth-first and depth-first search) to compute the optimal sequence of actions (least amount of times both player have bet). The goal is to win more than 100 coins of your opponent within 4 hands, i.e., the end state is either your agent won more than 100 coins or 4 hands has been played.
- Compare the solutions of all search algorithms implemented.

d. Given complete information of the game, design three heuristic functions and use two search algorithms (greedy and A search) to find a series of actions for your agent to win more than 100 coins (no hands limit). Evaluate the solution (how many nodes are expanded? Is the solution optimal?)

1 Grading Criteria

- Pass: Complete task 1a - 1d and 2a - 2c.
- Deadline is 2 weeks starting from the introduction session.
- Send your solution by email to:

  yuantao.fan@hh.se

  Your submission should include **code** and a **short report** of what you have done, observed and learnt.
- Extra credits: Complete task 2d.