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

DT8042, Halmstad University

November 17, 2021

Introduction

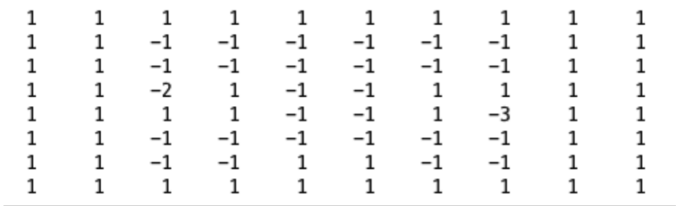
This lab is designed to introduce you to search algorithms (random, exhaustive, greedy search algorithm and A* search algorithm with customized heuristics). You will implement different search algorithms and apply them in two domains:

- Path planning: find the shortest path from the agent's 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 Amit's pages on A* algorithm). Additionally, you will compare the outputs of A* and other search algorithms, including the random, exhaustive and greedy search algorithms, to determine the fitness of each algorithm in solving the given tasks.

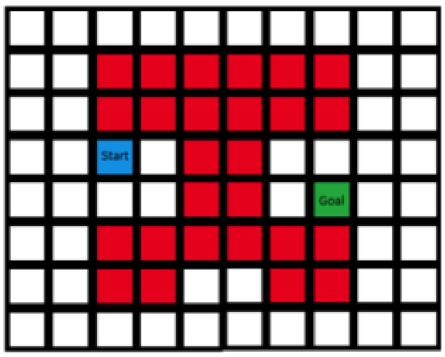
Application 1: Path Planning

In this task, you will implement the A* algorithm and use it for the path planning problem. The task is to find the 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, the 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:



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

- '**generateMap2d(...)**': generates a map of a given size, with randomly placed obstacles.
- '**generateMap2d_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 visualize and understand the results of your A* algorithm. It will color map cells according to a Value you provide, and it also draws the path found. You can use this function to analyze the behavior 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);
- An example of a searching algorithm is provided in **search_algorithm.py**

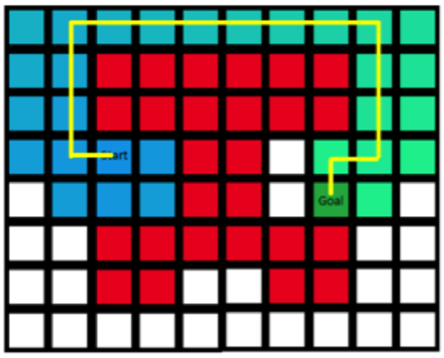
Task 1

a. Use **generateMap2d(...)** to generate to map and find a shortest path from the start point to the end point by implementing the following algorithms with the help of function **search(...)**:

- **Uninformed Search Algorithms**: Random Search, BFS and DFS: Test your algorithm with a number of randomly generated maps, with different sizes, obstacles as well as start/goal points.
- **Informed Search Algorithms**: Greedy Search and A* with Euclidean and Manhattan distances. Compare and analyze how well do they work

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

by visualizing the paths with the help of function **plotMap(...)**. What have you observed? 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)?

b. Use **generateMap2d_obstacle(...)** to generate a map with predefined obstacles. The environment contains a rotated-H shaped obstacle. Y coordinate of the two horizontal walls and X coordinate of the vertical wall are available. The starting point is always somewhere on the left side of the map and the goal is on the right side of the map. Your agent can only pass near the top edge of the map or near the bottom edge of the map.

- Perform a random search, exhaustive search (BFS and DFS) and greedy search to find an optimal path from the starting point to the ending point.
- Find an optimal path from the starting point to the ending point using the A* algorithm and make sure the path is optimal.
- Design new heuristic(s) that is specific for this type of map, which expands fewer nodes compare to the general-purpose heuristics, e.g. Euclidean and Manhattan distance. The following are two examples of heuristic functions for this map:
  - Heuristic I: Don't search/explore towards the right before you have explored either above or below the obstacle;
  - Heuristic II: Explore either upward or downward before searching to the right, e.g., if the starting point is close to the upper edge, you always only search upwards before you explore towards the right;

  Implement at least one heuristic function, explain your approach and result in the report, including a comparison of the A* algorithm using the customized heuristic function with the A* algorithm using other distance measures (e.g. Euclidean, and Manhattan distance) as heuristics.

- Compare different search algorithms (e.g. random search, BFS, DFS, A* algorithm using generic and customized heuristics) for the map with a rotated-H obstacle, from a statistical perspective (e.g. perform searches for 20 times and compare number of nodes expanded by different methods). Please include this comparison in the report, note that the length of the path (i.e. solution) found is required as well.

Application 2: Poker Bidding

In this task, you will use the A* algorithm to plan a sequence of actions that beat your opponent in a poker game (rules are more complicated than the one in the first lab but still simpler than the real game). The goal is to win as much as possible of your opponent's money. 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. The 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 the money in the pot.
- Call: in this poker game, the player can call by paying a certain amount of money (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.) 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.

The rules of the game are stated as follows:

- There will be two players play against each other within the game.
- Every agent has 100 coins to bet and if one of the agents loses all the coins, the game ends.
- There will be (maximum) n number of hands each game (you decide n).

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

Table 1: Responses to opponent's actions

- 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 or Fold}, (x could be drawn from a set of numbers, e.g. $\{5, 10, 25\}$.
     b) After your agent's action performed, your opponent act 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).
     c) Based on the opponent's action, available actions for the player to perform are {Call, bet x coin, Fold}. A general illustration of the 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 the bidding phase, both agents show their hands and the agent with stronger hands gets the pot (If one of the player's folds, there is no need to show the cards).
- The game ends (when) one player loses all the money.

A number of useful functions are provided in **poker_environment.py**.

- generate_2hands(...): randomly draw 10 cards from a deck and assign them into two piles.
- identify_hand(...): identify the type and the rank of the given hand (5 cards).
- The strategy of your opponent is given, function '**poker_strategy_example(...)**' returns your opponent's strategy. Table 2 explains the input of this function required.

Task 2

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

a. Read and understand helper functions provided in **poker_environment.py**. You are encouraged to implement the successor function, which generates successor states of the poker game. An example of the successor function (**get_next_states(...)**) is provided, within **poker_game_example.py**.

b. 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:

   - Use random search (maximum 10000 steps), exhaustive search (breadth-first and depth-first search) and greedy search algorithm (e.g. utilize least amount of times both players have bet) to compute the optimal sequence of actions. 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. What have you observed? Do all of them found a solution? How many nodes have been expended?

c. Given complete information of the game, design one or more heuristic functions (does not have to be admissible) to find a series of actions for your agent to win more than 100 coins within 4 hands. Evaluate your solution, e.g. How many nodes have expanded using the proposed heuristic function? Does it reduce the search space? Is the solution optimal?

1 Grading Criteria

- Pass: Complete task 1a, and 1b.
- Deadline is 2 weeks starting from the introduction session.
- Your submission should include **code** and a **report** of what you have done, observed, and learned. Please note that the report shall include a comparison between different algorithms and plots of the solution found.
- Extra credits: Complete all tasks.