CS 440/ECE 448 Spring 2018

Assignment 1: Search

Due at: Monday, February 12, 11:59 PM

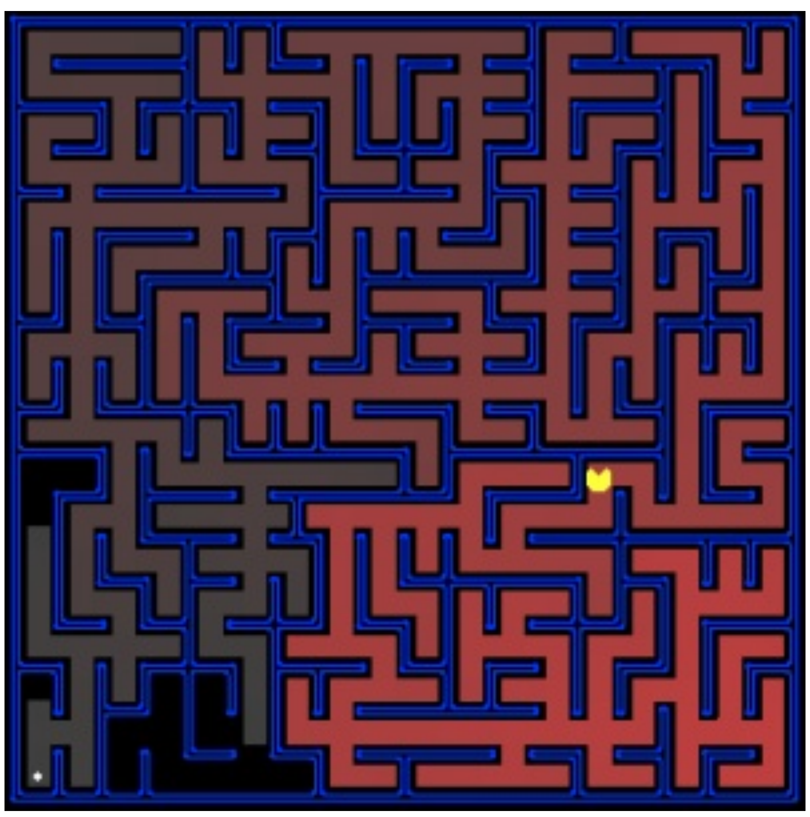
In this assignment, you will build general-purpose search algorithms and apply them to solving puzzles. In Part 1 (for everyone), you will be in charge of a "Pac-Man" agent that needs to find paths through mazes while eating one or more dots or "food pellets". In Part 2 (for four-credit students), you will tackle small instances of the Sokoban puzzle.

You are free to use any high-level programming languages you are comfortable with, including but not limited to Java, C++, Python, and MATLAB. You have the option of working in groups of up to three people, all of whom should take this course for the same number of credit hours. We focus on the entire problem solving process, and you should hand in both **the code** with your implementation and **a written report** with your analysis.

Three-credit students can earn a maximum of 24 points plus 6 extra credit points, whereas four-credit students can earn a maximum of 32 points plus 8 extra credit points. Four-credit students are required to finish Part 2, and can try Part 2 extra credit section for extra credit. Three-credit students can try Part 2 for extra credit, but are not suppoed to finish Part 2 extra credit section since your extra credit points will be capped for balance.

Contents

Part 1 (For everyone): Pac-Man



Credits: Berkeley CS188 Pacman projects, Kyo Hyun Kim and Krishna Kothapalli

1.1 Basic pathfinding

Consider the problem of finding the shortest path from a given start state while eating one or more dots or "food pellets". The above image illustrates the simple scenario of a single dot, which in this case can be viewed as the unique goal state. Here, all step costs are equal to one.

Implement the state representation, transition model, and goal test needed for solving the problem in the general case of multiple dots. For the state representation, besides your current position in the maze, is there anything else you need to keep track of? For the goal test, keep in mind that in the case of multiple dots, the Pacman does not necessarily have a unique ending position. Next, implement a unified top-level search routine that can work with all of the following search strategies, as covered in class:

- Depth-first search;
- Breadth-first search;
- Greedy best-first search;
- A* search.

For this part of the assignment, use the Manhattan distance from the current position to the goal as the heuristic function for greedy and A* search.

The maze layout will be given to you in a simple text format, where '%' stands for walls, 'P' for the starting position, and '.' for the dot(s). Run each of the four search strategies on the following inputs:

- Medium maze;
- Big maze;
- Open maze.

For each problem instance and each search algorithm, include the following in your report:

- The solution, displayed by putting a '.' in every maze square visited on the path.
- The path cost of the solution, defined as the number of steps taken to get from the initial state to the goal state.
- Number of nodes expanded by the search algorithm.

This section is worth **16 points** for everyone.

1.2: Search with multiple dots

Now consider the harder problem of finding the shortest path through a maze while hitting multiple dots. Once again, the Pacman is initially at P, but now there is no single goal position. Instead, the goal is achieved whenever the Pacman manages to eat all the dots. Once again, we assume unit step costs.

As instructed in 1.1, your state representation, transition model, and goal test should already be adapted to deal with this scenario. The next challenge is to solve the following inputs using A* search with an admissible heuristic designed by you:

- Tiny search;
- Small search;
- Medium search.

You should be able to handle the tiny search using uninformed BFS -- and in fact, it is a good idea to try that first for debugging purposes, to make sure your representation works with multiple dots. However, to successfully handle all the inputs, it is crucial to come up with a good heuristic. For full credit, your heuristic should be admissible and should permit you to find the solution for the medium search in a reasonable amount of time. In your report, explain the heuristic you chose, and discuss why it is admissible and whether it leads to an optimal solution.

For each maze, give the solution cost and the number of nodes expanded. Show your solution by numbering the dots in the order in which you reach them (once you run out of numbers, use lowercase letters, and if you run out of those, uppercase letters).

This section is worth **8 points** for everyone.

1.2 Extra Credit: Suboptimal search

Sometimes, even with A* search and a good heuristic, finding the optimal path through all the dots is hard. In these cases, we would still like to find a reasonably good path, quickly. Write a suboptimal search algorithm that will do a good job on this big maze. Your algorithm could either be A* search with a non-admissible heuristic, or something different altogether.

In your report, discuss your approach and output the solution cost and number of expanded nodes. You don't have to show the solution path unless you can come up with a nice animation.

This section is worth **2 extra credit points** for everyone.

Tips for Part 1

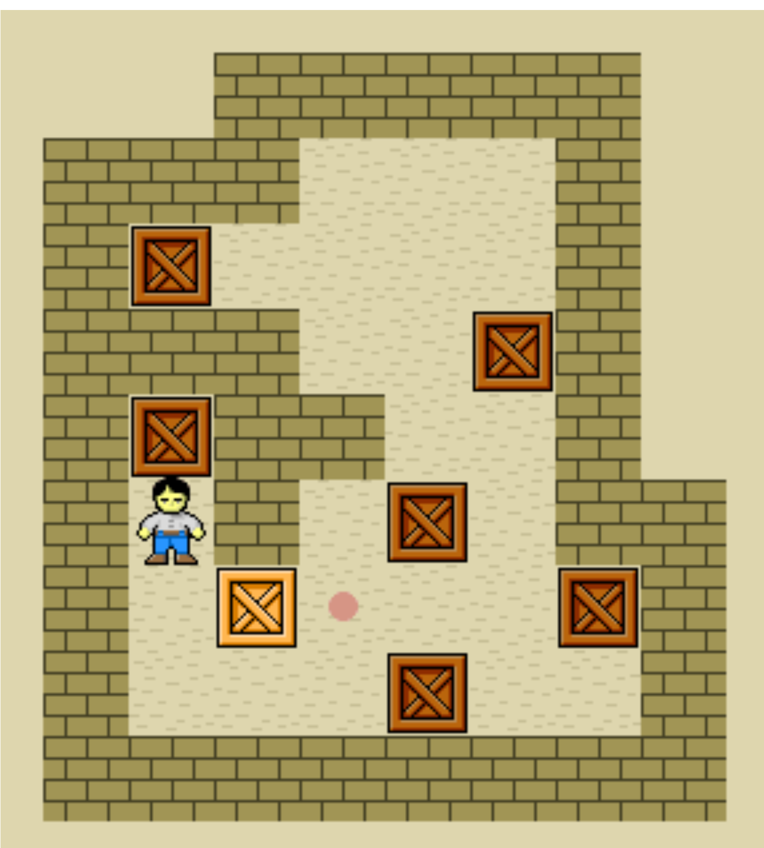
- In your implementation, make sure you get all the bookkeeping right. This includes handling of repeated states (in particular, what happens when you find a better path to a state already on the frontier) and saving the optimal solution path. These topics have been extensively covered during lectures.

- Pay attention to tiebreaking. If you have multiple nodes on the frontier with the same minimum value of the evaluation function, the speed of your search and the quality of the solution may depend on which one you select for expansion.

- Make sure you implement a unified top-level search algorithm that can take each of the four strategies as special cases. In particular, while DFS can be implemented very compactly using recursion, we want you to avoid this approach for the sake of the assignment (among other things, you can much more easily exceed the maximum depth of the recursion stack than if you explicitly represent the frontier as a stack).

- In Part 1, you will be graded primarily on the correctness of your solution, not on the efficiency and elegance of your data structures. For example, we don't care whether your priority queue or repeated state detection uses brute-force search, as long as you end up expanding (roughly) the correct number of nodes and find the optimal solution. So, feel free to use "dumb" data structures as long as it makes your life easier and still enables you to find the solutions to all the inputs in a reasonable amount of time. However, in Part 2, you will have to handle repeated state detection more efficiently in order to successfully solve all the inputs.

Part 2 (For four-credit students): Sokoban



Source: Wikipedia

In this part of the assignment, you will adapt your A* search to solve tiny instances of the Sokoban puzzle. Here is the description of the rules from Wikipedia:

> The game is played on a board of squares, where each square is a floor or a wall. Some floor squares contain boxes, and some floor squares are marked as storage locations. The player is confined to the board, and may move horizontally or vertically onto empty squares (never through walls or boxes). The player can also move into a box, which pushes it into the square beyond. Boxes may not be pushed into other boxes or walls, and they cannot be pulled. The number of boxes is equal to the number of storage locations. The puzzle is solved when all boxes are at storage locations.

You need to solve the four Sokoban instances below. In the following input files, '%' denotes walls, 'b' denotes boxes, '.' denotes storage locations, and 'B' denotes boxes on top of storage locations. The initial position of the agent is given by 'P'.

- Input 1
- Input 2
- Input 3
- Input 4

Implement the Sokoban transition model, then try to come up with interesting admissible heuristics that expand as few nodes as possible. For full credit, you should run two versions of search on each input: either uninformed search plus A* search with one heuristic, or A* search with two different heuristics (probably at different levels of sophistication).

In your report, describe your heuristic(s) and justify whether they are admissible. For each input and each method of search, give the number of moves needed to reach a goal configuration, the number of nodes expanded, and the running time.

This section is worth **8 points** for four-credit students, and **4 extra credit points** for three-credit students.

Part 2 Extra Credit: Suboptimal search

Come up with a suboptimal search algorithm to solve harder Sokoban inputs:

- Extra 1
- Extra 2
- Extra 3

In your report, discuss your approach and output the number of moves needed to reach a goal configuration, the number of nodes expanded, and the running time. You don't have to show the solution path unless you can come up with a nice animation.

This section is worth **6 extra credit points** for four-credit students, but no points for three-credit students.

Report Checklist

Your report should briefly describe your implementation and fully answer the questions for every part of the assignment. Your description should focus on the most "interesting" aspects of your solution, i.e., any non-obvious implementation choices and parameter settings, and what you have found to be especially important for getting good performance. Feel free to include pseudocode or figures if they are needed to clarify your approach. Your report should be self-contained and it should (ideally) make it possible for us to understand your solution without having to run your source code. For full credit, in addition to the algorithm descriptions, your report should include the following.

Part 1 (for everyone):

1. For every algorithm (DFS, BFS, Greedy, A*) and every one of the three mazes (medium, big, open): give the solved maze, the solution cost, and the number of expanded nodes (12 cases total).
2. For each of the three mazes (tiny, small, medium): give the solved maze, the solution cost, and the number of expanded nodes for your A* algorithm. Discuss your heuristic, including its admissibility.

Part 2 (for four-credit students):

- Discuss your implementation and heuristic(s), including admissibility. For each of the four inputs and two methods of search (either BFS plus A* with one heuristic, or A* with two different heuristics): give the solution cost, the number of expanded nodes, and the running time.

Extra credit:

- We reserve the right to give bonus points for any advanced exploration or especially challenging or creative solutions that you implement. Besides dedicated extra credit section in Part 1, three-credit students can get additional extra credit for solving Part 2, but not extra credit section of Part 2.

Statement of individual contribution:

- All group reports need to include a brief summary of which group member was responsible for which parts of the solution and submitted material. We reserve the right to contact group members individually to verify this information.

WARNING: You will not get credits for any solutions that you have obtained, but not included in your report! For example, if your code prints out path cost and number of nodes expanded on each input, but you do not put down the actual numbers in your report, or if you include pictures/files of your output solutions in the submitted `.zip` file but not in your PDF file, you will not get credits for your solution. The only exception is animated paths (videos or gifs).

Submission Instructions

By the submission deadline, **one designated person from the group** will need to upload the following to [Compass2g](https://compass2g.illinois.edu):

1. A **report** in **PDF format**. Be sure to put the **names** of all the group members at the top of the report, as well as the number of credits (3 or 4). The name of the report file should be **lastname_firstname_a1.pdf** (based on the name of the designated person).

2. Your **source code** compressed to a **single ZIP file**. The code should be well commented, and it should be easy to see the correspondence between what's in the code and what's in the report. You don't need to include executables or various supporting files (e.g., utility libraries) whose content is irrelevant to the assignment. If we find it necessary to run your code in order to evaluate your solution, we will get in touch with you. The name of the code archive should be **lastname_firstname_a1.zip**.

Compass2g upload instructions:

1. Log into **https://compass2g.illinois.edu** and find your section.
2. Select **Assignment 1 (three credits)** or **Assignment 1 (four credits)** from the list, as appropriate.
3. Upload **your PDF report** and **the ZIP file containing your code** as two attachments.
4. Hit **Submit**. -- ***If you don't hit Submit, we will not receive your submission and it will not count!***

Multiple attempts will be allowed but only your last submission before the deadline will be graded. We have variable grace periods, so if you submit your files on 12:00 am the next day due to network latency, do not panic. However, please never rely on the grace period and go any further. **We reserve the right to take off points for not following instructions.**

Late policy: For every day that your assignment is late, your score gets multiplied by 0.75. The penalty gets saturated after four days, that is, you can still get up to about 32% of the original points by turning in the assignment at all. If you have a compelling reason for not being able to submit the assignment on time and would like to make a special arrangement, you must send me email **at least four days before the due date** (any genuine emergency situations will be handled on an individual basis). **If you receive an approved extension, be sure to make a note of it at the top of your report.**

Be sure to also refer to course policies on academic integrity, etc.