Division of Engineering Science
UNIVERSITY OF TORONTO

ROB311: Artificial Intelligence

Project #1: State-Space Search

Winter 2021

Overview

In this project, you will implement and analyze a number of uninformed and informed search strategies for different problem domains. The goals are to:

- experiment with basic uninformed search strategies on an explicit graph; and
- implement A* for a simple 2D maze domain and analyze the difficulty of random problem instances.

The project has two parts, worth a total of **50 points**. All submissions will be via Autolab (more details will be provided in class); you may submit as many times as you wish until the deadline. To complete the project, you will need to review some material that goes beyond that discussed in the lectures—more details are provided below. The due date for project submission is **Monday, February 1, by 11:59 p.m. EST**.

Preliminaries - Handout Code

We have provided 'starter code,' consisting of a series of Python files that you will use to implement your solutions to Parts 1 and 2. Your task is to complete the following template files:

1. `breadth_first_search.py`: a simple breadth-first search (BFS) algorithm for problem domains with uniform action costs;
2. `bidirectional_search.py`: a bidirectional BFS algorithm that simultaneously searches from the initial and goal states until the intersection of each search's frontier representing an optimal path is found (also only for the uniform action cost case); and
3. `a_star_search.py`: a best-first search that computes optimal paths when supplied with a consistent heuristic for its problem domain.

These files contain function templates that you must fill in as your solution. You should **not** change or add to the import statements at the top of these files: the autograding software will strip out any additional imports and replace them with the default upon submission, causing your code to produce an error when the removed import is referenced. If you need additional libraries for testing or visualizing, use them in a separate file that imports your solutions from these 3 files. Each of these files has some simple problem instances after the `if __name__ == '__main__':` statement. You will need to create more extensive tests to ensure the correctness of your solutions. For **all** search implementations, you should return `None` or the empty list when a solution cannot be found.

The file `search_problems.py`, which you must **not** modify, implements a number of classes and a function you will need for your solutions:

1. a `Node` class that contains the `parent`, `state`, `action`, and `path_cost` fields required for state space search (see class notes and AIMA);
2. abstract base classes `SearchProblem` and `SimpleSearchProblem` which our main problem classes will inherit from;
3. a `GraphSearchProblem` class which you will use with your `breadth_first_search` and `bidirectional_search` solutions;
4. a `GridSearchProblem` class which you will use with your `a_star_search` solution; and
5. `get_random_grid_problem()`, a helper function that produces random `GridSearchProblem` instances.

This file, along with the standard library and permitted imports at the top of the submission templates, are sufficient for you to implement your solutions. You do not need to submit `search_problems.py` as part of your solutions: it is already on the grading server.

The class `BasicSearchProblem` is an abstract base class with required functionality for the network and grid search problems you will be solving. It is extended by the `GridSearchProblem` and `GridSearchProblem` classes, which together provide all the methods needed to implement a generic search algorithm like breadth-first search. Note that while these classes support multiple goal states, our problems will only involve one goal state, so you may assume that you can simply use `problem.goal_states[1]` to access the goal state (or just use the `goal_test()` method).

Part 1: Uninformed Search

You have been hired by PluggedUp, a tech startup that runs a social network focussed on "plugging" professionals into their dream careers. Your first task is to determine the shortest path connecting two users in the network so that recruiters can introduce employers to prospective hires through mutual acquaintances. The social network is stored as an undirected graph G where the vertices $v \in V$ representing users are labelled with unique integers and users that are in one another's contact lists are connected with an edge $e \in E$. Connections are assumed to be uniformly weighted so that the shortest path is simply the number of edges or hops between two vertices.

The `GridSearchProblem` class has a constructor that takes in an initial state, a list of goal states (we will only ever have a single goal state in this assignment), and a graph specified by a set of vertices V and edges E. The example at the bottom of `breadth_first_search.py` and `bidirectional_search.py` gives an example problem setup using the provided file `stanford_large_network_facebook_combined.txt`. These edges are from real anonymized Facebook data made available by the Stanford Large Network Dataset Collection. Your tasks are to:

1. Implement breadth-first search (BFS) to find the shortest path between two users in the graph. Use the function template called `breadth_first_search` in `breadth_first_search.py`. Your function should return a tuple with list of states representing your path from the initial state to the goal state, the number of nodes expanded by your search, and the maximum frontier size encountered during the search. Only the first element of this tuple (the path) will be graded, but the other fields are useful for comparing algorithms and will be needed in Part 2. You are free to implement your own data structures in this file, but you would be wise to make use of the `deque` structure provided in the import. The `set()` data structure is also potentially useful.

2. Implement bidirectional search to solve the shortest path problem, using the function template `bidirectional_search` in `bidirectional_search.py`. The textbook (AIMA) describes bidirectional search on page 90, but it leaves out some details that you will have to fill in to ensure that your bidirectional search is optimal. To test and debug your code, compare your results with those from the implementation of `breadth_first_search`. You should think about the relative runtime, max. frontier size, and number of nodes expanded by each algorithm.

You will submit your implementations through Autolab (instructions at the end of this document).

Part 2: Occupancy Grid Planning With A*

In this section, you will work with the `GridSearchProblem` class to find optimal paths through a 2D occupancy grid maze (see Figure 1 for an example). In the lecture slides and on page 84 of AIMA, the pseudocode describing uniform cost search provides a good guide, but there are other variants that you are free to explore. For example, you **do not** have to remove a node from the frontier if a node with a shorter path to the initial state is found. This results in a less memory efficient but sometimes faster algorithm. We recommend using the `PriorityQueue` class provided by the `queue` library (note that you do not have access to `deque` for this problem).

Our ultimate goal is to study the "hardness" of grid search problem instances in the same manner as the graphs on page 264 of AIMA. Your tasks are to:

1. Fill in the `a_star_search` function in `a_star_search.py` so that it outputs a tuple consisting of a path (once again a list of integer states), the number of nodes generated, and the maximum frontier size encountered. You may once again find the `set()` data structure useful, as well as the standard dictionary (`dict()`).
2. For different values of a square map with dimension N, recreate the graphs on page 264 of AIMA for our grid search problem. Rather than the clause to symbol ratio, our "hardness" parameter on the x-axis will be the probability p_{occ} of a grid cell being occupied. This value is the first argument of `get_random_grid_problem.py`. On the y-axis of one chart, plot the portion of n_{runs} that were solvable by A*, and on the other plot the average number of nodes *generated* over n_{runs}. Generated nodes include those **not** added to the frontier (i.e. legal state transitions that are not added because they have already been explored). Plot one curve for square grid size $N = 20$, $N = 100$, and $N = 500$ ($M = N$ in all cases). Use $n_{runs} = 100$ and use a resolution of 0.05 for values of p_{occ} from 0.1 to 0.9. For each of the n_{runs} problem instances at each value of p_{occ} and N, use `get_random_grid_problem` to generate your problem instance. This experiment may take a while to run! Does A* exhibit the same "phase transition" phenomenon as SAT? What effect does increasing N appear to have on each plot? Would you conjecture that this threshold becomes sharp as $N \to \infty$?
3. Complete the simple template function `search_phase_transition` in `a_star_search.py`. This function simply returns the hardness "transition interval" $[l, b] \subset [0, 1]$ and the single peak computational effort (in terms of nodes generated) $p_{peak} \in [0, 1]$ observed for your $N = 500$ experiment. Do not worry about being too precise, just choose the interval of length ≤ 0.2 where the probability of solvability transitions from approximately 1 to approximately 0. To be clear, you do **not** need to submit code that runs the phase transition experiment: simply modify the the 3 outputs of `search_phase_transition` to report your findings.

You will submit and check your implementations of `a_star_search` and `search_phase_transition` through Autolab (both of these should be implemented as functions in `a_star_search.py`).

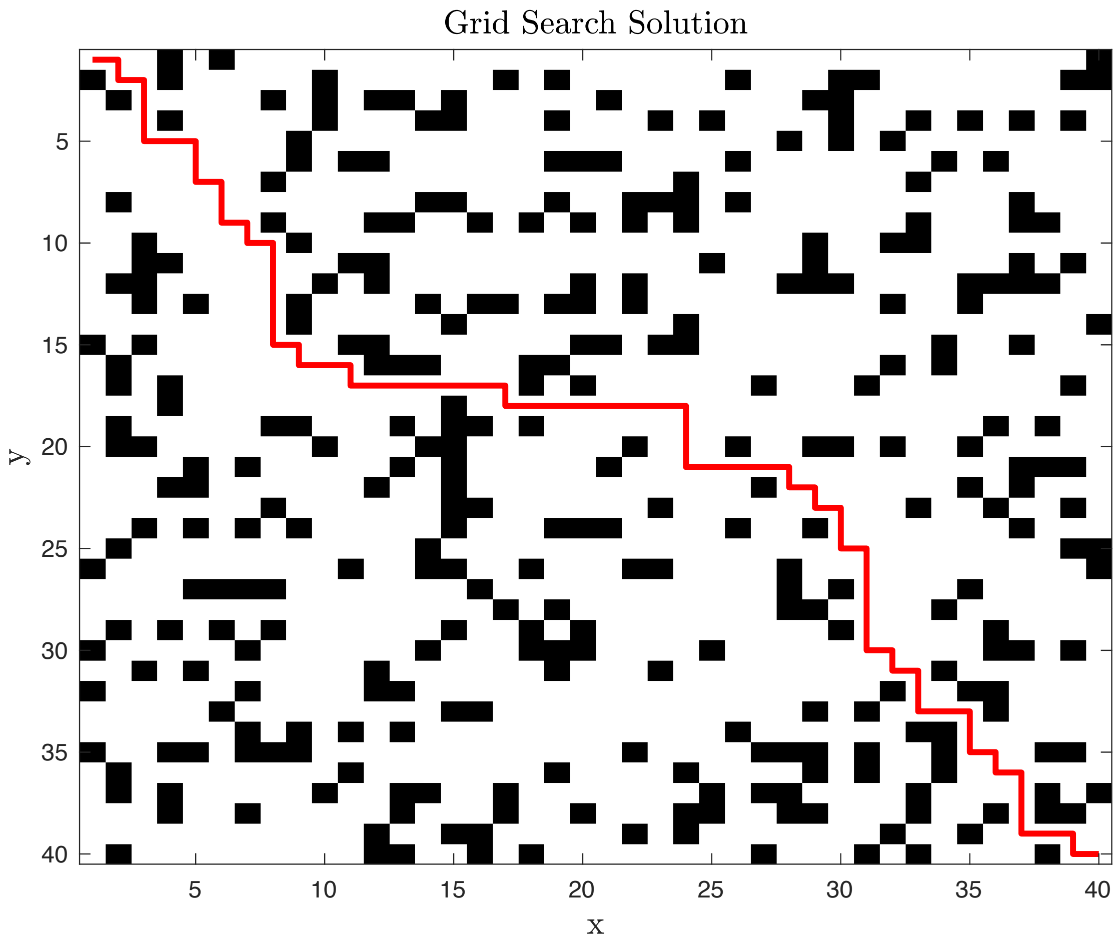

Figure 1: Example solution to a 2D maze problem. Black cells are "occupied" by an obstacle and cannot be entered. Each cell represents a state, and our agent can only transition between cells that share a side (i.e., no diagonal movement).

Grading

We would like to reiterate that **only** functions submitted via Autolab will affect your marks. There are other questions in the sections above, but only those that ask you to submit a function via Autolab will affect your total. The remainder are useful for your understanding or are there to aid you in creating your solutions. Points for each portion of the project will be assigned as follows:

- Uninformed Search – **20 points** (5 problems × 4 marks each)

  Each test will check either `breadth_first_search` or `bidirectional_search` for the shortest path from an initial state in a graph with undirected edges E to a final state.

- Occupancy Grid Planning With A* – **30 points** (4 problems × 5 marks each + 10 marks)

  Each of the 4 instances test of `a_star_search` will provide a `GridSearchProblem` instance for a total of 20 points. The test of `search_phase_transition` is worth 10 points.

Total: **50 points**

When you are ready to submit, archive the files with the following command:
`tar cvf handin.tar breadth_first_search.py bidirectional_search.py a_star_search.py`
Then upload the tar file to
http://courses.jonathankelly.info/courses/rob-311-artificial-intelligence-w21/assessments/assignment1

Grading criteria include: correctness and succinctness of the implementation of support functions, proper overall program operation, and code commenting. Please note that we will test your code *and it must run successfully*. It is worth knowing that there is a **300 seconds** time limit for each submission, which means that the presence of an infinite loop will essentially give you no marks. Please test your code for such loops rigorously before submitting. You may even add a limit on the number of loop iteration during the debugging process.

Code that is not properly commented or that looks like 'spaghetti' may result in an overall deduction of up to **10%**. The expectation is that it should be possible to understand your algorithm by simply reading your comments. There is no specific commenting format that you have to follow. Please check the file `search_problems.py` for some basic commenting examples.