CS 4365 Artificial Intelligence
Spring 2021
Assignment 1: Basic Search
Part I: *Due electronically by Friday, Feburary 5, 11:59 p.m.*
Part II: *Due electronically by Friday, February 19, 11:59 p.m.*

Instructions:

1. Your solution to this assignment must be submitted via eLearning.
2. For the written problems, submit your solution as a **single PDF** file.
   - Only use **blue or black pen** (black is preferred). Scan your PDF using a **scanner** and upload it. Make sure your final PDF is **legible**. **Regrades due to non-compliance will receive a 30% score penalty**.
   - Verify that both your answers and procedure are **correct, ordered, clean, and self-explanatory** before writing. Please ask yourself the following questions before submitting:
     - Are my answers and procedure legible?
     - Are my answers and procedure in the same order as they were presented in the assignment? Do they follow the specified notation?
     - Are there any corrections or scratched out parts that reflect negatively on my work?
     - Can my work be easily understood by someone else? Did I properly define variables or functions that I am using? Can the different steps of my development of a problem be easily identified, followed, and understood by someone else? Are there any gaps in my development of the problem that need any sort of justification (be it calculations or a written explanation)? Is it clear how I arrived to each and every result in my procedure and final answers? Could someone describe my submission as messy?
3. **You may work individually or in a group of two**. Only one submission should be made per group. If you work in a group, **make sure to indicate both group members when submitting** through eLearning.
4. **IMPORTANT:** As long as you follow these guidelines, your submission should be in good shape; if not, we reserve the right to penalize answers and/or submissions as we see fit.

Part I: Written Problems (50 points)

1. **Uninformed Search (23 points)**

   We define a state space as follows. The start state is 1. The successor function for state n returns two states, which are numbered $2n$ and $2n + 1$. The goal state is 13.

   (a) **(4 pts)** Draw the state space involving only states 1 to 13.

   (b) **(5 pts)** List the order in which nodes will be *visited* for breadth first search.

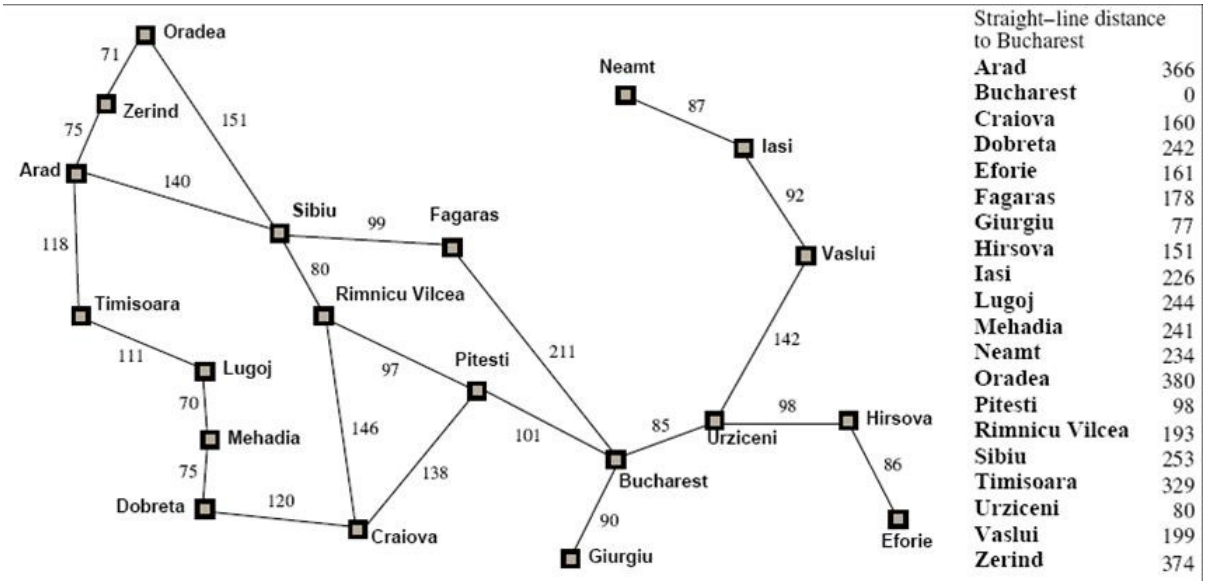
   (c) **(5 pts)** List the order in which the nodes will be *visited* for depth-limited search with limit 3.

   (d) **(5 pts)** List the order in which the nodes will be *visited* for iterative deepening search.

   (e) **(4 pts)** Is bidirectional search appropriate for this problem? Explain your answer.

2. **Informed Search (27 points)**

   Consider the problem of getting to Bucharest from Oradea in the map below:



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

   When answering the following questions, assume that (1) each node should be expanded at most once; and (2) if needed, break ties alphabetically.

   (a) **(10 pts)** Trace the operation of the UCS search algorithm applied to this problem by showing the state that is expanded and its successor states (together with their g values) in each iteration of the algorithm.

   (b) **(7 pts)** Trace the operation of the best-first greedy search algorithm applied to this problem by showing the state that is expanded and its successor states 9together with their h values) in each iteration of the algorithm. Use the straight-line distance heuristic values shown to the right of the map.

(c) **(10 pts)** Trace the operation of the A* search algorithm applied to this problem by showing the state that is expanded and its successor states (together with their g, h, and f values) in each iteration of the algorithm. Use the straight-line distance heuristic values shown to the right of the map.

Part II: Programming (100 points)

Task Description

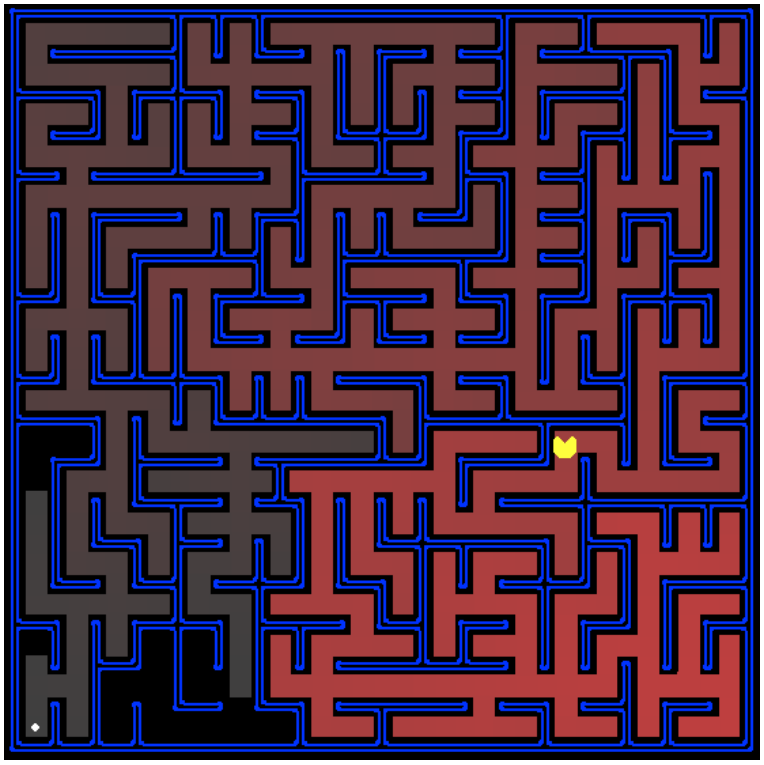


Figure 1: All those colored walls, Mazes give Pacman the blues, So teach him to search.

Introduction

In this project, your Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. You will build general search algorithms and apply them to Pacman scenarios.

This project includes an autograder for you to grade your answers on your machine. This can be run with the command:

```
python autograder.py
```

The code for this project consists of several Python 3 files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore. You can download all the code and supporting files as a zip from the course website.

Files you'll edit:

- **search.py**
  - Where all of your search algorithms will reside.
- **searchAgents.py**
  - Where all of your search-based agents will reside.

Files you might want to look at:

- **pacman.py**
  - The main file that runs Pacman games. This file describes a Pacman GameState type, which you use in this project.
- **game.py**
  - The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.
- **util.py**
  - Useful data structures for implementing search algorithms.

File Download: Please see the course website.

Files to Submit: You will fill in portions of **search.py** and **searchAgents.py** during the assignment. You should submit these files with your code and comments. Please *do not* change the other files in this distribution or submit any of our original files other than these files.

Evaluation: Your code will be autograded for technical correctness. Please *do not* change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder's judgements – will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; *please* don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact the course staff for help. Office hours and Piazza are there for your support; please use them. If you can't make our office hours, talk to your TA. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Piazza: Please be careful not to post spoilers.

Welcome to Pacman

After downloading the code (**search.zip**), unzipping it, and changing to the directory, you should be able to play a game of Pacman by typing the following at the command line:

```
python pacman.py
```

Note: The exact command you should be using depends on your distribution and python installation.

Pacman lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain.

The simplest agent in **searchAgents.py** is called the **GoWestAgent**, which always goes West (a trivial reflex agent). This agent can occasionally win:

```
python pacman.py --layout testMaze --pacman GoWestAgent
```

But, things get ugly for this agent when turning is required:

```
python pacman.py --layout tinyMaze --pacman GoWestAgent
```

If Pacman gets stuck, you can exit the game by typing CTRL-c into your terminal.

Soon, your agent will solve not only **tinyMaze**, but any maze you want.

Note that **pacman.py** supports a number of options that can each be expressed in a long way (e.g., **--layout**) or a short way (e.g., **-l**). You can see the list of all options and their default values via:

```
python pacman.py -h
```

Question 1 (12 points): Finding a Fixed Food Dot using Depth First Search

In **searchAgents.py**, you'll find a fully implemented **SearchAgent**, which plans out a path through Pacman's world and then executes that path step-by-step. The search algorithms for formulating a plan are not implemented – that's your job.

First, test that the **SearchAgent** is working correctly by running:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

The command above tells the **SearchAgent** to use **tinyMazeSearch** as its search algorithm, which is implemented in **search.py**. Pacman should navigate the maze successfully.

Now it's time to write full-fledged generic search functions to help Pacman plan routes! Pseudocode for the search algorithms you'll write can be found in the lecture slides. Remember that

a search node must contain not only a state but also the information necessary to reconstruct the path (plan) which gets to that state.

Important note: All of your search functions need to return a list of *actions* that will lead the agent from the start to the goal. These actions all have to be legal moves (valid directions, no moving through walls).

Important note: Make sure to **use** the **Stack**, **Queue** and **PriorityQueue** data structures provided to you in **util.py**! These data structure implementations have particular properties which are required for compatibility with the autograder.

Hint: Each algorithm is very similar. Algorithms for DFS, BFS, UCS, and A* differ only in the details of how the fringe is managed. So, concentrate on getting DFS right and the rest should be relatively straightforward. Indeed, one possible implementation requires only a single generic search method which is configured with an algorithm-specific queuing strategy. (Your implementation need *not* be of this form to receive full credit).

Implement the depth-first search (DFS) algorithm in the **depthFirstSearch** function in **search.py**. To make your algorithm *complete*, write the graph search version of DFS, which avoids expanding any already visited states.

Your code should quickly find a solution for:

```
python pacman.py -l tinyMaze -p SearchAgent

python pacman.py -l mediumMaze -p SearchAgent

python pacman.py -l bigMaze -z .5 -p SearchAgent
```

The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration). Is the exploration order what you would have expected? Does Pacman actually go to all the explored squares on his way to the goal?

Hint: If you use a **Stack** as your data structure, the solution found by your DFS algorithm for **mediumMaze** should have a length of 130 (provided you push successors onto the fringe in the order provided by getSuccessors; you might get 246 if you push them in the reverse order). Is this a least cost solution? If not, think about what depth-first search is doing wrong.

Question 2 (12 points): Breadth First Search

Implement the breadth-first search (BFS) algorithm in the **breadthFirstSearch** function in **search.py**. Again, write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for depth-first search.

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

```
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

Does BFS find a least cost solution? If not, check your implementation.

Hint: If Pacman moves too slowly for you, try the option **--frameTime 0**.

Note: If you've written your search code generically, your code should work equally well for the eight-puzzle search problem without any changes.

```
python eightpuzzle.py
```

Question 3 (12 points): Varying the Cost Function

While BFS will find a fewest-actions path to the goal, we might want to find paths that are "best" in other senses. Consider **mediumDottedMaze** and **mediumScaryMaze**.

By changing the cost function, we can encourage Pacman to find different paths. For example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas, and a rational Pacman agent should adjust its behavior in response.

Implement the uniform-cost graph search algorithm in the **uniformCostSearch** function in **search.py**. We encourage you to look through **util.py** for some data structures that may be useful in your implementation. You should now observe successful behavior in all three of the following layouts, where the agents below are all UCS agents that differ only in the cost function they use (the agents and cost functions are written for you):

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

Note: You should get very low and very high path costs for the **StayEastSearchAgent** and **StayWestSearchAgent** respectively, due to their exponential cost functions (see **searchAgents.py** for details).

Question 4 (12 points): A* search

Implement A* graph search in the empty function **aStarSearch** in **search.py**. A* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The **nullHeuristic** heuristic function in **search.py** is a trivial example.

You can test your A* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic (implemented already as **manhattanHeuristic** in **searchAgents.py**).

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,
heuristic=manhattanHeuristic
```

You should see that A* finds the optimal solution slightly faster than uniform cost search (about 549 vs. 620 search nodes expanded in our implementation, but ties in priority may make your numbers differ slightly). What happens on **openMaze** for the various search strategies?

Question 5 (12 points): Finding All the Corners

The real power of A* will only be apparent with a more challenging search problem. Now, it's time to formulate a new problem and design a heuristic for it.

In *corner mazes*, there are four dots, one in each corner. Our new search problem is to find the shortest path through the maze that touches all four corners (whether the maze actually has food there or not). Note that for some mazes like **tinyCorners**, the shortest path does not always go to the closest food first! *Hint*: the shortest path through **tinyCorners** takes 28 steps.

Note: Make sure to complete Question 2 before working on Question 5, because Question 5 builds upon your answer for Question 2.

Implement the **CornersProblem** search problem in **searchAgents.py**. You will need to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached. Now, your search agent should solve:

```
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,
prob=CornersProblem
```

```
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,
prob=CornersProblem
```

To receive full credit, you need to define an abstract state representation that *does not* encode irrelevant information (like the position of ghosts, where extra food is, etc.). In particular, do not use a Pacman **GameState** as a search state. Your code will be very, very slow if you do (and also wrong).

Hint: The only parts of the game state you need to reference in your implementation are the starting Pacman position and the location of the four corners.

Our implementation of **breadthFirstSearch** expands just under 2000 search nodes on **mediumCorners**. However, heuristics (used with A* search) can reduce the amount of searching required.

Question 6 (12 points): Corners Problem: Heuristic

Note: Make sure to complete Question 4 before working on Question 6, because Question 6 builds upon your answer for Question 4.

Implement a non-trivial, consistent heuristic for the **CornersProblem** in **cornersHeuristic**.

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

Note: **AStarCornersAgent** is a shortcut for

```
-p SearchAgent -a fn=aStarSearch,prob=CornersProblem,
heuristic=cornersHeuristic
```

Admissibility vs. Consistency: Remember, heuristics are just functions that take search states and return numbers that estimate the cost to a nearest goal. More effective heuristics will return values closer to the actual goal costs. To be *admissible*, the heuristic values must be lower bounds on the actual shortest path cost to the nearest goal (and non-negative). To be *consistent*, it must additionally hold that if an action has cost c, then taking that action can only cause a drop in heuristic of at most c.

Remember that admissibility isn't enough to guarantee correctness in graph search – you need the stronger condition of consistency. However, admissible heuristics are usually also consistent, especially if they are derived from problem relaxations. Therefore it is usually easiest to start out by brainstorming admissible heuristics. Once you have an admissible heuristic that works well, you can check whether it is indeed consistent, too. The only way to guarantee consistency is with a proof. However, inconsistency can often be detected by verifying that for each node you expand, its successor nodes are equal or higher in in f-value. Moreover, if UCS and A* ever return paths of different lengths, your heuristic is inconsistent. This stuff is tricky!

Non-Trivial Heuristics: The trivial heuristics are the ones that return zero everywhere (UCS) and the heuristic which computes the true completion cost. The former won't save you any time, while the latter will timeout the autograder. You want a heuristic which reduces total compute time, though for this assignment the autograder will only check node counts (aside from enforcing a reasonable time limit).

Grading: Your heuristic must be a non-trivial non-negative consistent heuristic to receive any points. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll be graded:

Number of nodes expanded	Grade
more than 2000	0/12
at most 2000	4/12
at most 1600	8/12
at most 1200	12/12

Table 1: Grading for question 6

Remember: If your heuristic is inconsistent, you will receive *no* credit, so be careful!

Question 7 (16 points): Eating All The Dots

Now we'll solve a hard search problem: eating all the Pacman food in as few steps as possible. For this, we'll need a new search problem definition which formalizes the food-clearing problem:

FoodSearchProblem in **searchAgents.py** (implemented for you). A solution is defined to be a path that collects all of the food in the Pacman world. For the present project, solutions do not take into account any ghosts or power pellets; solutions only depend on the placement of walls, regular food and Pacman. (Of course ghosts can ruin the execution of a solution! We'll get to that in the next project.) If you have written your general search methods correctly, **A*** with a null heuristic (equivalent to uniform-cost search) should quickly find an optimal solution to **testSearch** with no code change on your part (total cost of 7).

```
python pacman.py -l testSearch -p AStarFoodSearchAgent
```

Note: **AStarFoodSearchAgent** is a shortcut for

```
-p SearchAgent -a fn=astar,prob=FoodSearchProblem,
heuristic=foodHeuristic
```

You should find that UCS starts to slow down even for the seemingly simple **tinySearch**. As a reference, our implementation takes 2.5 seconds to find a path of length 27 after expanding 5057 search nodes.

Note: Make sure to complete Question 4 before working on Question 7, because Question 7 builds upon your answer for Question 4.

Fill in **foodHeuristic** in **searchAgents.py** with a *consistent* heuristic for the **FoodSearchProblem**. Try your agent on the **trickySearch** board:

```
python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

Our UCS agent finds the optimal solution in about 13 seconds, exploring over 16,000 nodes.

Any non-trivial non-negative consistent heuristic will receive 4 points. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll get additional points:

Number of nodes expanded	Grade
more than 15000	4/16
at most 15000	8/16
at most 12000	12/16
at most 9000	16/16 (full credit; medium)
at most 7000	20/16 (optional extra credit; hard)

Table 2: Grading for question 7

Remember: If your heuristic is inconsistent, you will receive *no* credit, so be careful! Can you solve **mediumSearch** in a short time? If so, we're either very, very impressed, or your heuristic is inconsistent.

Question 8 (12 points): Suboptimal Search

Sometimes, even with A* and a good heuristic, finding the optimal path through all the dots is hard. In these cases, we'd still like to find a reasonably good path, quickly. In this section, you'll write an agent that always greedily eats the closest dot. **ClosestDotSearchAgent** is implemented for you in **searchAgents.py**, but it's missing a key function that finds a path to the closest dot.

Implement the function **findPathToClosestDot** in **searchAgents.py**. Our agent solves this maze (suboptimally!) in under a second with a path cost of 350:

```
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
```

Hint: The quickest way to complete **findPathToClosestDot** is to fill in the **AnyFoodSearchProblem**, which is missing its goal test. Then, solve that problem with an appropriate search function. The solution should be very short!

Your **ClosestDotSearchAgent** won't always find the shortest possible path through the maze. Make sure you understand why and try to come up with a small example where repeatedly going to the closest dot does not result in finding the shortest path for eating all the dots.

Submission

Once you are done, sign in to gradescope. You will be able to see *Basic Search* under the course assignments section. Directly submit your *search.py* and *searchAgents.py* files. Please **do not** rename the files or upload the files in a zip file or folder (your homework will not be graded otherwise). If you worked in a group, make sure to add your partner when you submit!

Grading

We will be using an autograder as the main grading mechanism for this submission; the assignment files includes a copy (autograder.py) so we encourage you to use it before you submit! You may notice that the task description (and autograder) points sum up to a different number than the one indicated here; this is not an issue, we will be using an updated version that normalizes scoring for your final submission. The autograder is pretty sophisticated but you may still run into some very specific (rare) cases where it does not behave correctly. If you believe the autograder made a mistake when grading your homework, feel free to stop by your TA's office hours so you can discuss it with him. Any necessary corrections will be made to ensure you receive due credit for your work.

Important note: We will be using MOSS for plagiarism detection on all programming assignments (you can look it up, it's quite powerful). Any students caught cheating will receive a 0 on the corresponding assignment and will be reported according to university regulations.