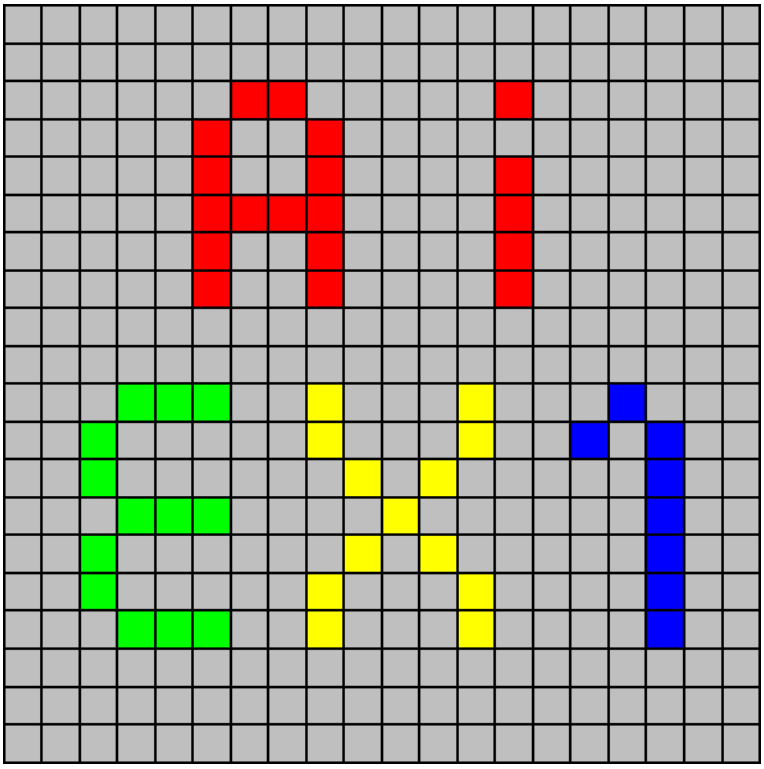
Introduction to Artificial Intelligence
Project 1: Search in Blokus



Deadline: May 30 2024 23:59

Introduction

In this project you will write search algorithms for a simple game based on a game called Blokus. Please familiarize yourself with the game online. You should read the rules and you can play against each other or computer players. We have provided several puzzles based on the game, which you will solve using search algorithms.

The code for this project consists of several Python 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 (including this description) as a zip archive.

Files you'll edit:

serach.py - Where all of your search algorithms will reside.

blokus_problems.py - Where your problem definitions and helper functions will reside.

Files you might want to look at:

game.py - Main game-running file. This file also allows you to watch a game of Blokus to get a feel for how it works.

board.py - Defines the board layout and rules. Also holds some supporting definitions and classes.

util.py - Useful data structures for implementing search algorithms.

Supporting files you can ignore:

displays.py - Graphics for Blokus.

inputs.py - Contains a random player and an interface allowing human interaction.

pieces.py - Loads a list of game pieces from a file.

What to submit:

Code - You will fill in portions of search.py and blokus_problems.py during the assignment. You should submit these two files (only) and a `README.txt`.

Answers - You will answer question that will require you to think about the theory, your results and what's between them. These questions are marked by "Understanding Questiron". You should submit a pdf file with your answers.

Each team should submit exactly one tar file that contains the four files from above.

Evaluation: Your code will be autograded for technical correctness. The autograder machine is running Python 3 (don't use Python 2.7). Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. The final grade will be normalized such that the maximum grade is 100. Please make sure you follow the `README` format **exactly**.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy code from someone else 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: If you have any question, you are probably not alone. Please post your questions via the relevant exercises forum on the course Moodle. **Please do not write to our personal e-mail addresses!**

README format: Please submit a `README.txt` file. The `README` should include the following lines (exactly):

1. id1 --- student 1 id
2. id2 --- student 2 id
3. ***** --- 5 stars denote end of i.d information
4. comments

For an example check out the README.txt provided with your project. This `README` will be read by a script, calling the autograder. Note that if you decide to submit alone, you should remove line 2, i.e.:

1. id1 --- student 1 id
2. ***** --- 5 stars denote end of i.d information
3. comments

1 Welcome to Blokus

To run the game, you will need the python `tkinter` package. To install it, use your package manager. This package is already installed on the school computers. Playing against an intelligent computer player will require an adversarial search agent, which we will learn about later in the course. For now, you can watch demo agents that choose moves randomly (implemented in inputs.py):

```
python3 game.py
```

1.1 Depth First Search (2 points)

Implement the depth-first search (DFS) algorithm in the `depth_first_search` function in search.py. To make your algorithm complete, write the graph search version of DFS, which avoids expanding any already visited states (textbook section 3.5). Your code should quickly find a solution for:

```
python3 game.py -p tiny_set.txt -s 4 7 -z fill
```

The game will output the board states on the way to the solution your search found, step by step. You will also get the number of search nodes expanded and the solution cost. Does your solution use all the moves that were checked?

Your code should be general and work with any search problem. For example, Pacman maze navigation:

```
python3 pacman.py -a fn=dfs
```

1.2 Breadth First Search (1 point)

Implement the breadth-first search (BFS) algorithm in the `breadth_first_search` 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:

```
python3 game.py -p tiny_set.txt -f bfs -s 4 7 -z fill
python3 pacman.py -a fn=bfs
```

Does BFS find the shortest path through pacman's maze? If not, check your implementation.

Hint: BFS takes longer and more memory than DFS for these problems. Have patience - the search could take several minutes.

Note: If you've written your search code generically, your code should work equally well for the eight-puzzle search problem (textbook, section 3.2) without any changes:

```
python3 eightpuzzle.py
```

2 BFS vs. DFS (Understanding Question)

In this question we will compare the BFS and DFS search algorithms, theoretically and empirically. We will do this with respect to the problems you ran (filling a blokus board and finding a route in the Pacman maze).

Please note: when you present empirical results, choose a form of presentation that is easy to draw conclusions from, for example a suitable graph or table.

2.1 Optimality (1 point)

1. Theoretically, for each of the algorithms - is its optimality guaranteed? Answer the question theoretically with respect to the problems you ran. Explain your answers.
2. Empirically, what are the costs of the algorithms' solutions in the problems you ran? For each of them - is it possible to conclude from the results that it is optimal or reject its optimality?
3. Are the theoretical and empirical results consistent with each other? If so, explain. If not, explain how this is possible.

2.2 Running Time (2 points)

1. Theoretically-asymptotically, which of the algorithms has better time complexity? Assume the time complexity is the number of nodes expanded and express the answer using the parameters of the problem - b, d, m, and briefly explain the calculation.
2. Empirically, how many vertices did the search algorithms expand in the problems you ran? So which one is faster? Intuitively explain the difference in the algorithms' behaviors that caused the difference in running time in the Pacman maze problem.
3. Are the theoretical and empirical results consistent with each other? If so, explain. If not, explain how this is possible.

3 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 a puzzle where we'd like to cover the corners of the board (introduced next): we would like a solution where the number of tiles placed is minimal, not just the number of moves. To do this, we will need to vary the cost function.

3.1 BlokusCornersProblem (2 points)

In BlokusCornersProblem (in blokus_problems.py), there are three locations to cover, one in each corner (other than the starting location). Our new search problem is to find the most efficient way through the board to cover all four corners. Note that you may place a tile on the board only if it touches at least one piece with only corner-to-corner contact allowed; edges cannot touch! The cost of an action in this search problem is the size of the tile. That is, we want to cover all the corners while leaving the board as vacant as possible.

If you implemented the problem correctly, you should be able to run the following command:

```
python3 game.py -p tiny_set_2.txt -f bfs -s 6 6 -z corners
```

3.2 Uniform Cost Search (2 points)

Implement the uniform-cost graph search algorithm in the `uniform_cost_search` 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 be able to find the minimal solution for the following problems:

```
python3 game.py -p tiny_set_2.txt -f ucs -s 6 6 -z corners

python3 game.py -p small_set.txt -f ucs -s 5 5 -z corners
```

Note: The run time for the corners covering problems is very long. We can do better by choosing which moves to try first - see the next question. However, you can check that the solutions found have lower cost than the ones found by DFS and BFS.

3.3 A* Search (3 points)

Implement A* graph search in the empty function `a_star_search` 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. Use it to test your A* search:

```
python3 game.py -p tiny_set_2.txt -f astar -s 6 6 -z corners -H null_heuristic
```

Note: make sure that the A* works properly as we will be using it a lot from this stage on. Test your algorithm on different board sizes, compare the cost with the cost given be your Uniform Cost Search implementation.

```
python3 game.py -p tiny_set_2.txt -f ucs -s 6 6 -z corners
```

4 Writing Heuristics

The real power of A* will only be apparent with a more challenging search problem. Now, it's time to formulate new problems and design heuristics for it. You've already written `BlokusCornersProblem` - now you will write a heuristic for it.

4.1 Blokus Corners Heuristic (4 points)

Implement a heuristic for the `BlokusCornersProblem` in `blokus_corners_heuristic` (blokus_problems.py).

Grading: inadmissible heuristics will get no credit. 2 point for any admissible heuristic. The other 2 point will be awarded based on how many nodes your heuristic expands. The top 40% submissions will receive full credit +1 bonus point; the next 35% will get 2 points and the other submissions will be awarded with 1 point.

```
python3 game.py -p tiny_set_2.txt -f astar -s 6 6 -z corners -H blokus_corners_heuristic
```

5 Admissible and Non-Admissible Heuristics (Understanding Question)

In this question you will come up with the heuristics for the blokus corners problem. You will notice that there are heuristics that better approximate the distance to a target but are not admissible (and therefore do not guarantee the optimality of the solution).

5.1 a Valid Heuristic (2 points)

1. Propose an admissible and consistent heuristic for the problem. This can be the heuristic you implemented or some other reasonable heuristic, it should be non-trivial.
2. Explain why the heuristic you proposed is admissible.

5.2 a New Heuristic (2 points)

1. Suggest another non-admissible heuristic that is a better approximation of the minimum distance in a significant number of cases. You can take the heuristic from the previous section and modify it.
2. Show an example of a state where the new heuristic better approximates the minimal distance, and prove that it is not admissible using a counterexample. You can choose examples with board size, game pieces and current state as you wish.

6 Cover All Locations

The next problem that you will define is `BlokusCoverProblem` in blokus_problems.py. This problem is similar to `BlokusCornersProblem`, while this time the goal is to cover all given locations (targets argument). As before, your agent should find a solution that leaves the board as vacant as possible.

6.1 `BlokusCoverProblem` (2 points)

Fill in the missing parts of `BlokusCoverProblem` in blokus_problems.py.

```
python3 game.py -p small_set.txt -f astar -s 6 6 -H null_heuristic -z cover -x 3 3 "[(2,2),
(5, 5), (1, 4)]"
```

6.2 Blokus Cover Heuristic (6 points)

Implement a heuristic for the `BlokusCoverProblem` in `blokus_cover_heuristic`.

Grading: inadmissible heuristics will get no credit. 2 points for any admissible heuristic, a consistent heuristic will get another point. The other 3 points will awarded based on the performance of your heuristic compared to your classmates.

```
python3 game.py -p small_set.txt -f astar -s 10 10 -H blokus_cover_heuristic -z cover -x
3 3 "[(2,2), (5, 5), (6, 7)]"
```

Blokus Memory Hints

In the field of artificial intelligence, most of the problems we are trying to solve belong to a computation class called NP-Hard (at least). While no efficient (in the complexity theory sense) algorithm exists for these problems, we must still do our best to make the algorithms we use run in a reasonable amount of time. Search is the standard workhorse of classical artificial intelligence, so it is worth taking the time to do it right (also your exercise grade will likely depend on it).

Memory Management

The first exercise in this course involves writing search algorithms - specifically BFS and DFS - in python. Theoretically, python does memory management for you, so you don't have to worry about any of this. In practice, nothing is ever that simple.

All the information you use when running an algorithm must reside somewhere in the computer's memory. While python allocates and frees memory resources for you (such as objects, lists and other variables), doing so takes time. A typical memory allocation (such as creating a new list or object) takes several milliseconds, or about as long as several million calculation operations. This (almost) does not depend on the size of the object created. While milliseconds does not sound like very long, for a search algorithm that opens (even without expanding) hundreds of thousands of nodes, memory allocation alone will take minutes or even hours. All this memory must later be freed, which takes similar amounts of time.

Reducing Memory Usage

This exercise does not use a lot of memory (in modern terms), but for your exercise to run fast you must be smart about memory allocation. Here are some general practical tips:

- Avoid copying objects whenever possible. If an object is needed in several parts of your code, hold a single copy and pass it by reference.
- Avoid creating new lists when you can. This means that list comprehension is not your friend anymore, as it creates a new list by filtering and applying functions to each element of the old list. A simple loop might be faster, and should be considered before turning to advanced (high-level) functions such as map. If you are not sure, use the profiling tool (described in the next subsection) to test both and see which is faster.

Specifically, this means you should consider the following:

- The only place where you absolutely must copy a board is in creating a new search state for a newly opened node.
- Think carefully what objects should be stored in the visited list, and how:
  - Checking if an element is in a set is faster than checking if it is in a list. What functions does this require? When should two search nodes/states be considered equal?
  - What do you want to store in visited list? Do you need the entire search node?

Checking Your Code's Efficiency

PyCharm has a built-in profiling tool, which you can run from the button to the right of the "run" and "debug" buttons (the one with the clock). This will run your code and generate a report that tells you how much time each part of you code is taking. Some parts of your code will inevitably take time, but you might find that others are taking unexpectedly long - these are the parts you should look at and try to improve.

Expected Times

If you have written your code efficiently, all of the questions should each take less than 30 seconds.

Getting Help

This explanation page includes concepts and terms introduced in courses you may not yet have taken or do not feel confident about (computation complexity and memory management). You should feel free to come to your TA for help with these topics if you feel they are preventing you from doing your best work on this exercise.

Good Luck!