Homework #1

Due by 11:59pm 10/02/2017

Objective

This assignment explores the connection between problem formulation and search strategies. You will need to implement and compare several search strategies on a few different problems.

Programming Language

Your program should be in either Python, Java or .Net C# language, the program should be named

- "puzzlesolver.py" if Python is used.
- ["puzzlesolver.cs" and "puzzlesolver.exe"] if C# is used.
- ["puzzlesolver.java" and "puzzlesolver.jar"] if Java is used.

Program Specification

The program should take *two mandatory input arguments*. The first argument is a configuration file for the problem your program has to solve (see the next section for more information). The second argument is a keyword that specifies which search algorithm to use to solve the problem: **bfs, unicost, greedy, iddfs** and **Astar** (with the exception of the **iddfs**, assume "graph-search").

Your program should solve the input problem using the specified search algorithm and strategy. Upon completion, it should output the following information:

- The solution path (or "No solution" if nothing was found): print one state per line.
- **Time**: expressed in terms of the total number of nodes created.
- **Space**: expressed as the *maximum* number of nodes kept in memory (the biggest size that the frontier list grew to) as well as the maximum number of states stored on the explored list. Please report these as two separate numbers.
- **Cost**: the final cost of the specified path or configuration.

Problem Types

An important element of this assignment is to separate the problem specific parts from the generic search algorithm part. Make sure that your implementation of the search does not hard code in any problem specific details. For this assignment, your program will be working with three problems: the wireless sensor monitoring problem and the data aggregation problem. And one more to be revealed a week later. If you've kept the problem/search abstraction clean, when you get the new problem, you ought to be able to just code up the functions relevant to

setting up that problem as a state space search (e.g., get_successor_states, and goal_test, etc.)

Wireless sensor monitoring

You are given a set of sensors and a set of targets, we want the sensors to monitor the targets for the longest time possible. Each sensor runs on a battery, a sensor loses power with a rate dependent on the distance between sensor and the target it is monitoring. We need to maximize the time of monitoring all the targets (the time before the first target gets out monitoring range) by minimizing the power consumption of all sensors. Two sensors can monitor one target so that if one is dead the other can still continue, that means the target is still monitored.

We will use the configuration file to specify the particular instance of the problem. It has the following format:

- Line 1 is a keyword (**monitor**) that tells you this is the monitoring problem
- Line 2 is a list of the sensors IDs, location and power to start with; e.g., [("S_1",1,1,100),(" S_2",10,5,88),("S_3",3,4,120), …..]
- Line 3 is a list of the target IDs and locations; e.g., [("T_1",0,0),("T_2",2,3),("T_3",1, 5)]
- Power loss function is, $P_t = P_{(t-1)}$ - Euclidian distance between target and sensor.

You may assume that we **will not** stress test with bad inputs like negative power or impossible states.

Data Aggregation Problem

For this problem, we will read in a graph representation of nodes and connections between them are undirected edges with time delay. We need to find the node to start from and the path so that we can visit all nodes and aggregate the data within these nodes with minimum time delay. Here is the format of the input configuration file:

- Line 1 is a keyword (**aggregation**) that tells you that this is the data aggregation problem.
- Line 2 is a list of the node IDs and locations of the nodes; e.g., [("N_1",1,1),("N_2",2,3),("N_3",1, 5)]
- Each of the following lines specifies a connection between two nodes and the time delay between the two nodes; e.g.,: ("N_1", "N_2", 4) specifies that it the time delay between N_1 to N_2 is 4 or vice versa.

You may assume that we **will not** stress test with bad inputs like connections with negative or zero cost. We may try a large graph with this problem, however.

Timeline

- Begin by implementing the uninformed search strategies (**bfs**, **iddfs** and **unicost)** for the two specified problems.

- Next, implement the informed search algorithms (**greedy** and **Astar**) for the two specified problems. What would be a reasonable heuristic function for each problem type?
- Submit the first part by **Monday (09/25/2017)**. At this point, you should have completed the uninformed search strategies and gotten started on the informed strategies. You may still modify any part of the code you want during the last week. We will not grade this first week version for correctness, but we will check to see your early implementation decisions. **Points will be deducted according to the course late penalty policy if this is not submitted**.
- The third problem type will be announced on **Monday (09/25/2017)**. At that time, we will also give out a few test cases for your program to solve (you will need to submit a transcript of your program's outputs for these test cases). You will have a week to complete the assignment, including the write-ups, and make a final submit by **Monday (10/02/2017)**.

What to submit

- The Python, Java or C# source code.
    - If code in Python, just submit **puzzlesolver.py** that can be run as specified in program specification section.
    - If code in C#, submit two files **(puzzlesolver.cs)** and the generated **puzzlesolver.exe** file. Make sure your exe file can be run from console as specified in program specification section.
    - If code in Java, submit two files **(puzzlesolver.java**) and the generated **puzzlesolver.jar** file. Make sure your jar file can be run from console as specified in program specification section.
    - **If the program couldn't be run from console by passing the specified arguments, you will be graded with score 1.**

- A README file that contains the following information:
    - The version of Python you used (if python used), framework used in case of C#
    - Any additional libraries needed to run your code.
    - List any additional resources, references, or web pages you've consulted.
    - List any person with whom you've discussed the assignment and describe the nature of your discussions.
    - **(final submission)** Describe any component of your program that is not working.

- The output of running the test cases on your code, make your program print the output in a text file in the same format it is printed on the console.

- A report that discusses the relative performances of the search algorithms and the heuristic functions.
    - What heuristic function did you use for each problem class?
    - Did all the outcomes make sense (e.g., do the time/space complexities of different search strategies match your expectation based on our class discussions? What about optimality and completeness?)

- Was there anything that surprised you? If so, elaborate.

Grading Guideline

Assignments are graded qualitatively on a nonlinear five point scale. Below is a rough guideline:

- 5 (100%): The program works as expected; has a clear enough README for the grader; includes a transcript; complete report with insightful observations.

- 4 (93%): The program works as expected, but possibly with a minor flaw; has a clear enough README for the grader; includes a transcript; complete report.

- 3 (80%): The program works for **two problem** types, and there is a clear abstraction between the search and problem parts; OR, the program works for at least **3 search** strategies for all **three problem** types; Has a clear enough README for the grader; Includes a transcript; Cursory report.

- 2 (60%): The program works for at least **2 search** strategies for at least **two problem** types; has a clear enough README for the grader; includes a transcript; cursory report. OR: has a better program but the report is missing.

- 1 (40%): The program has major problems, but a serious attempt has been made on the assignment; has a clear enough README for the grader.

Homework #1 Puzzle Update

Due by 11:59pm 10/02/2017

The Burnt Pancake Problem

This is a variation on the more straight-forward pancake flipping problem. In the original pancake flipping problem, you are given a stack of pancakes of varying diameters. You want to sort the pancakes so that they are in order from the largest (at the bottom) to the smallest (at the top). You are only allowed to flip the pancakes by sticking a spatula into the stack and flipping all pancakes above it. For example, if I have a stack of pancakes in the order of (top to bottom):

3 1 2 5 4

and if I stick my spatula between pancake size 2 and pancake size 5 and flip, the result will be:

2 1 3 5 4

In the burnt pancake variant, we distinguish between the top and bottom of the pancakes (which are apparently all burnt). We still want to sort the pancakes in order of their sizes, but we now also want all the burnt side to face down. If a pancake has its burnt side up, we'll represent it with a negative sign. For example, the pancake stack:

3 -1 2 5 4

means that the pancake of size 1 (the second from the top position) has its burnt side up. And if we stick our spatula between 2 and 5 and flip, the result will be:

-2 1 -3 5 4

Now, the pancakes of diameter 2 and 3 have their burnt sides facing up.

We will use a configuration file to specify the particular instance of the puzzle. It has the following format:

Line 1 is a keyword (**pancakes**) that tells you this is the pancakes problem
Line 2 is the initial state; e.g., (3, -1, 2, 5, 4)
Line 3 is the goal state; e.g., (1, 2, 3, 4, 5) // we don't really need this line

You may assume that we will always give n pancakes, all with unique sizes ranging from 1 to n.

Test Cases

For each of the test cases for the 3 problems, capture your program's outputs for the specified search strategies and store them in the "transcript" file. If the search strategy takes longer than 30 minutes on any test case, you may interrupt it and report that the search strategy was not able to finish within the allotted time.

Report Discussions

Your report should discuss the following issues:

1. For each of the three puzzle types, what do you think is the best search strategy, and why? You should take all four factors into considerations (completeness, optimality, time complexity, space complexity).
2. For the 3 problems, what heuristic function(s) did you make? Give your rationale. Also, discuss whether you think they are admissible and consistent.

You may also include additional discussions on any observations that you find surprising and/or interesting.