CSCI-561 - Spring 2020 - Foundations of Artificial Intelligence Homework 1

Due February 12, 2020, 23:59:59

1. Overview

This is a programming assignment in which you will apply AI search techniques to some future "time travelling" problems with some limited simplifications. As shown in Figure 1, there are many possible worlds along with the timeline indexed by the years. Each world is a grid of space with (x, y) locations in which your agent can use some elementary actions to move to one of the eight neighboring grid locations. Within a possible grid world, there are some special bi-directional time-travelling channels located at certain locations that will allow your agent to travel in time forward or backward in the history. For example, in Figure 1, there is a time-travel channel located at (x=3,y=4) in the year 2020 that could allow your agent to time-travel to the future year 2023, or vice versa. Similarly, there is a channel at (x=9, y=7) in the year 2020 that could allow your agent to time-travel back to the year 2018, or vice versa.

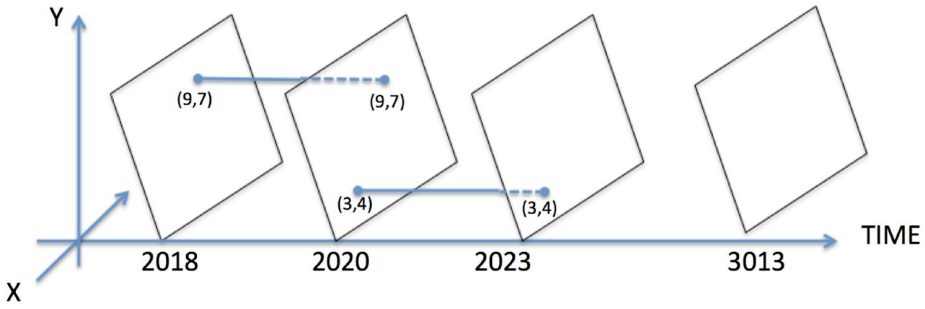

Figure 1: Time-travelling Configuration: grid worlds with bi-directional time-travel-channels.

At a special location where there is a time-travel channel, your agent may use an action called "jaunt" to time-travel to a different year in the history at the other end of the channel. To represent your agent's time-location in the history configuration, we may use the term "year-location." For example, if your agent is at (year=2020, x=4, y=7), that means your agent

is in the year 2020 at the grid location (x=4, y=7). If there is a time-travel channel at your agent's current year-location, then your agent may choose to jaunt and time-travel. For example, at (2020,3,4), your agent may jaunt to the year-location (2023,3,4). Similarly, jaunting at (2020,9,7) will take your agent back in history to (2018,9,7). Of course, at such a location, your agent may choose not to jaunt but move normally inside the same grid world of the same year. For example, moving north at (2020,3,4) will take your agent to (2020,3,5); moving southwest at (2020,3,4) will result in (2020,2,3); moving southeast at (2020,3,4) will go to (2020,2,5); and moving south at (2020,3,4) will go to (2020,3,3). Notice that if your agent is trying to move outside the grid world, then that action will have no effect and result your agent to stay where it is.

Your programming task is as follows. Given three inputs: (1) a time-travelling configuration, such as the one in Figure 1; (2) an initial year-location, and (3) a goal or target year-location, your program must search in the configuration and find the optimal shortest path from the initial year-location to the target year-location, using the elementary move actions and jaunt.

Conceptually, the specification of a time-traveling problem is given as a list of max-x, max-y, initial-year-location, target-year-location, plus a list time-travel channels. For example (Note: The exact input format will be given in section 5 and 6 below),

INPUT: 100,100, (2020,1,3), (2023,4,6), [(2020,3,4,2023), (2018,9,7,2020)]

is a specification for grid worlds of size 100 x 100, with the initial year-location of (2020,1,3), the target year-location of (2023,4,6), and the two time-travel channels as illustrated in Figure 1. Notice that the number of time-travel channels may vary from one problem to another. This example input indicates that your agent is given a time-travel configuration of grid worlds of size 100x100, plus two time-travel channels as shown in Figure 1, and your agent is required to find the shortest path from the initial year-location (2020,1,3) to the target year-location (2023,4,6). Given such inputs, your program may find and output a shortest path as follows. Notice that every step in your output must be a legal action as specified above (Note: the exact output format will be given in section 5 and 6 below.)

OUTPUT: (2020,1,3), (2020,2,4), (2020,3,4), (2023,3,4), (2023,4,5), (2023,4,6).

To assist your programming, you will be provided some sample inputs and outputs (see below). Please understand that the goal of these samples is to check that you can correctly parse the problem definitions and generate a correctly formatted output. The samples are very simple and it should not be assumed that if your program works on the samples it would definitely work on all test cases for grading. There will be more complex test cases and it is your task to make sure that your program will work correctly on any valid input. You are encouraged to design and try your own test cases to check how your program would behave in some complex special cases that you might think of. Since **each homework is checked via an automated A.I. script**, your output should match the specified format ***exactly***. Failure to do so will most certainly cost some points. The output format is simple and examples are provided. You should upload and test your code on vocareum.com at their terminal window which is a linux-like environment. Please make sure you test your program at the terminal window at

vocareum.com before you click the submit button there. You can submit as many times as you like, and the last submission before the due time will be used to grade your results. You may use any of the following programming languages: **C++, Java, Python**, but Python may be the preferred language to use in today's large-scale AI program applications.

2. Grading

Your code will be tested and graded as follows: Your program should not require any command-line argument. It should read a text file called "input.txt" in the current directory that contains a problem definition. It should create and write a file "output.txt" with your solution in the same current directory. Format for input.txt and output.txt are specified as in section 5 below, and will be supplemented with some details in section 6. End-of-line character is LF (since vocareum is a Unix system and follows the Unix convention).

The grading A.I. script will test your program for 50 test cases for grading as follows:

- Create an input.txt file, and delete any old output.txt file.
- Run your code to create your output.txt file.
- Check correctness of your program's output.txt file.
- If your outputs for all 50 test cases are correct, you get 100 points.
- If one or more test case fails, you get 50–N points where N is the number of failed test cases.

Note that if your code does not compile, or somehow fails to load and parse input.txt, or writes an incorrectly formatted output.txt, or no output.txt at all, or OuTpUt.TxT, **you will get zero points**. Anything you write to stdout or stderr will be ignored and is ok to leave in the code you submit (but it will likely slow you down). Please test your program on Vocareum's terminal window with the provided sample files to avoid any problems.

3. Academic Honesty and Integrity

All homework material is checked vigorously for dishonesty using several methods. All detected violations of academic honesty are forwarded to the Office of Student Judicial Affairs. To be safe you are urged to err on the side of caution. Do not copy work from another student or off the web. Keep in mind that sanctions for dishonesty are reflected in *your permanent record* and can negatively impact your future success. As a general guide:

Do not copy code or written material from another student. Even single lines of code should not be copied.

Do not collaborate on this assignment. The assignment is to be solved individually.

Do not copy code off the web. This is easier to detect than you may think.

Do not share any custom test cases you may create to check your program's behavior in more complex scenarios than the simplistic ones considered below.

Do not copy code from past students. We keep copies of past work to check for this. Even though this problem differs from those of previous years, do not try to copy from homeworks of previous years.

Do not ask on piazza how to implement some function for this homework, or how to

calculate something needed for this homework.

Do not post code on piazza asking whether or not it is correct. This is a violation of academic integrity because it biases other students who may read your post.

Do not post test cases on piazza asking for what the correct solution should be.

Do ask the professor or TAs if you are unsure about whether certain actions constitute dishonesty. It is better to be safe than sorry.

4. Project Description

In this project, we twist the problem of path planning a little bit just to give you the opportunity to deepen your understanding of search algorithms by modifying search techniques to fit the criteria of a more realistic application. To give you a realistic context for expanding your ideas about search algorithms, we invite you to take part in a time-traveling mission in the future. The goal of this mission is to send your sophisticated and intelligent agent from a specific year-location to a target year-location. You are invited to develop an algorithm to find the optimal and shortest path and navigate through a complex time-traveling configuration based on a particular objective.

We assume a time-travel configuration that is specified by a set of possible grid worlds along the timeline indicated by the year. Every grid world has the same format and is a matrix (X, Y) with H rows (where H is a strictly positive integer) and W columns (W is also a strictly positive integer). For example, a grid world may be a map with W=3 columns and H=2 rows. By convention, we will use East (E) as the positive direction for X; West (W) as the negative direction of X; North (N) as positive direction for Y; and South (S) as the negative direction for Y. In addition, each possible grid world has a time value specified as a year in the timeline history, as illustrated in Figure 1.

The input of your program includes three elements: a time-travel configuration, an initial year-location, and a target year-location, plus perhaps some other quantities that control the quality of the solution. Each possible grid world can be imagined as a surface in a 2-dimensional space. A popular way to represent “time” is to use a year value assigned to each grid world. At each location of a grid world, your agent can move to each of the **8 possible neighbor grid locations**: North, Northeast, East, Southeast, South, Southwest, West, and Northwest. Actions are assumed to be deterministic and error-free. If your agent’s action is legal, then your agent will always end up at the intended neighbor grid location. If your agent tries to move outside a grid world, the result will be nil and your agent will remain in its current location.

In addition to the eight normal move actions, the jaunting action will allow your agent to time-travel to a different year in the history if your agent is located at the same location of a **bi-directional** time-travel channel. If there is no time-travel channel at the current location, the jaunting action will have no effect and your agent will remain stay at its current location.

5. Search for the Optimal Paths

You will write a program that will take an input file that describes the time-travel configuration, the initial year-location, the target year-locations, and characteristics of the agent. For each target year-location, you should find the optimal (shortest) path **from the initial year-location to that target year-location**. A path is composed of a sequence of legal moves. Each legal move consists of moving the agent to one of its 8 neighbors in the same grid world in the current year, or a jaunting action through a time-travel channel at the current location.

Your agent must search through all possible paths of normal and time-travel movements and find the shortest path to travel from the initial year-location to the target year-location, and then output the results. Ideally, your agent can go anywhere in the space-time configuration, and usually the shortest path is defined as the optimal path.

To find the solution you will use the following algorithms:

- Breadth-first search (BFS)
- Uniform-cost search (UCS)
- A* search (A*).

Your algorithm should return an **optimal path**, that is, with shortest possible operational path length. Operational path length is further described below and is not equal to geometric path length as per the specifications given ahead. If an optimal path cannot be found, your algorithm should return "FAIL" as further described below.

To help us distinguish between your three algorithm implementations, you must follow the following conventions for computing operational path length:

- **Breadth-first search (BFS)**

In BFS, each move from one location to any of its 8 neighbors counts for a unit path cost of 1. You do not need to worry about the fact that moving diagonally (e.g., Northeast) actually is a bit longer than moving along the North to South or East to West directions. So, any allowed move from one location to an adjacent location costs 1. A jaunting action will cost 1 as well regardless of the size of the jump in time.

- **Uniform-cost search (UCS)**

When running UCS, you should compute unit path costs in 2D. Let us assume that a location's center coordinates projected to a 2D ground plane are spaced by a 2D distance of 10 North-South and East-West. That is, a North or South or East or West move from a grid location to one of its 4-connected straight neighbors incurs a unit path cost of 10, while a diagonal move to a neighbor incurs a unit path cost of **14 as an approximation to $10\sqrt{2}$ when running UCS.** As usual, a jaunting action will cost the number of years it time-travels.

- **A* search (A*).**

When running A*, you should compute an approximate integer unit path cost of each move as in the UCS case (unit cost of 10 when moving North to South or East to West, and unit cost of 14 when moving diagonally). A jaunting action will cost the number of years it time-travels. Notice that for A*, **you need to design an admissible heuristic for A* for this problem.**

Input: The file input.txt in the current directory of your program will be formatted as follows:

- First line: Instruction of which algorithm to use, as a string: BFS, UCS or A*
- Second line: Two strictly positive 32-bit integers separated by one space character, for "W H" the number of columns (width) and rows (height) for all grid worlds.
- Third line: Three positive 32-bit integers separated by a space character, "Year X Y" for the **initial** year-location, where $0 \leq X \leq W-1$ and $0 \leq Y \leq H-1$. That is, we use 0-based indexing into a grid world: X increases when moving east, Y increases when moving north, and (0,0) is the southwest corner of the grid world.
- Fourth line: Three positive 32-bit integers separated by a space character, "Year X Y" for the **target** year-location, where $0 \leq X \leq W-1$ and $0 \leq Y \leq H-1$. That is, we use 0-based indexing into a grid world: X increases when moving east, Y increases when moving north, and (0,0) is the southwest corner of the grid world.
- Fifth line: A strictly positive 32-bit integer N, indicating the number of time-travel channels in the configuration.
- Next N lines: Four positive 32-bit integers separated by one space character, for
  - "Year X Y Year", specifying the coordinates of the time-travel channel and the two years at the ends of the channel. Again, we have $0 \leq X \leq W-1$, $0 \leq Y \leq H-1$, and 0-based indexing into the grid worlds.

For example:

```
A*
100 200
2020 8 12
3472 13 17
3
2020 12 16 3011
3011 56 77 2487
3011 14 19 3472
```

In this example, all grid worlds are of size 100 x 200, the initial year-location is (year=2020,x=8,y=12), and the target year-location is (year=3472,x=13,y=17). There are

three time-travel channels, the first one is located at (12,16) between the years 2020 and 3011, the second one located at (56,77) between the years 3011 and 2487, and the third one located at (14,19) between the years 3011 and 3472.

Output: The file output.txt that your program creates in the current directory should be formatted as follows:

- First line: A single integer C, indicating the total cost of your found solution. If no solution was found (target year-location is unreachable from the given initial year-location), then write the word "FAIL".
- Second line: A single integer N, indicating the total number of steps in your solution including the starting position.
- N lines: Report the steps in your solution travelling from the initial year-location to the target year-location as were given in the input.txt file.
    - Write out one line per step with cost. Each line should contain a tuple of four integers: Year, X, Y, Cost, separated by a space character, specifying the year-location with the step cost to visit that year-location by your agent during its (time) traveling from the initial year-location to the target year-location.

For example, the following is a sample output of the corresponding input above:

```
1570
12
2020    8       12      0
2020    9       13      14
2020    10      14      14
2020    11      15      14
2020    12      16      14
3011    12      16      991
3011    13      17      14
3011    14      18      14
3011    14      19      10
3472    14      19      461
3472    13      18      14
3472    13      17      10
```

Notes and hints:

- Please name your program "**homework.xxx**" where 'xxx' is the extension for the programming language you choose ("py" for python, "cpp" for C++, and "java" for

Java). If you are using C++11, then the name of your file should be "homework11.cpp" and if you are using python3 then the name of your file should be "homework3.py". Please use the programming languages mentioned above for this homework.

- Likely (but no guarantee) we will create 15 BFS, 15 UCS, and 20 A* test cases.
- Your program will be killed after some time if it appears stuck on a given test case, to allow us to grade the whole class in a reasonable amount of time. We will make sure that the time limit for a given test case is at least five times longer than the time it took for the reference Python algorithm written by the TA to solve that test case correctly.
- There is no limit on input size, number of time-travel channels, etc., other than specified above (32-bit integers, etc). If several optimal solutions exist, then any of them will count as correct.
- Please submit your homework code through Vocareum (https://labs.vocareum.com/) under the assignment HW1. Useranme is your email address. Click "forgot password" for the first time login. You should have been enrolled in this course on Vocareum. If not, please post a private question with your email address and USC ID on Piazza so that we will invite you again.
- You can submit your homework code (by clicking the "submit" button on Vocareum) as many times as you want. Only the latest submission will be considered for grading. Late penalty will be applied as specified in the syllabus if your latest submission is after the deadline.

6. Sample Inputs and Outputs

6.1. Example 1 (BFS with no solution):

=======input.txt=============

```
BFS
99 87
2020 8 12
2021 8 12
2
2020 12 16 3011
12 56 77 1
```

==========

=======output.txt============

```
FAIL
```

==========

6.2. Example 1 (BFS with solution):

=======input.txt=============

```
BFS
99 87
2020 8 12
2020 8 12
2
2020 12 16 3011
3011 56 77 2487
```

==========

=======output.txt============

```
0
1
2020 8 12 0
```

==========

6.3. Example 3 (UCS):

=======input.txt=============

```
UCS
99 87
2020 8 12
2024 8 13
5
2020 12 16 3011
3011 56 77 1
12 0 0 2
2 0 0 2024
2020 8 13 2024
```

==========

=======output.txt=============

```
14
3
2020 8 12 0
2020 8 13 10
2024 8 13 4
```

==========

6.4. Example 4 (A*):

=======input.txt=============

```
A*
99 87
2020 8 12
0 8 12
1
2020 8 11 0
```

==========

=======output.txt=============

```
2040
4
2020 8 12 0
2020 8 11 10
0 8 11 2020
0 8 12 10
```

==========