Computer Science 5400
Artificial Intelligence

Spring 2024

Puzzle Assignment #4

Version 04.15

DUE DATE : Monday, April 22th, 11:59:59pm

Assignment:

Create a program that implements **A-star Graph Search** (**A*GS**) over the **Act-Man-II** game to find a desired sequence of game actions. You can read more about A*GS in section 3.6.1 of the textbook. If you own the Norvig textbook, you can also read Sections 3.5 & 3.6 for more details on A*GS. Your program's **objective** is to find a sequence of actions (for ActMan) that achieves **victory!**. (i.e all monsters dead and ActMan still alive)

Specifications:

Your program will follow the same input/output structure from the previous *Puzzle Assignments*. Two **command line arguments** indicate the **input** and **output** files. The input file will contain an **Act-Man-II** dungeon specification, which will include complete information about the dungeon and the initial conditions. Your program shall write to the output file a sequence of actions for Act-Man that achieves the desired postcondition : **victory!**.
As before, your output files are what is relevant to your grade, whatever is displayed to the screen during program execution will be ignored.

Heuristic Function:

As you are implementing **A*GS**, candidate plans for ActMan should be considered in an order determined by both **plan cost** and a **heuristic function**. Cost function **cost(s)** should return the cost of reaching game state **s** from the initial state. The function **h(s)** should take a game state **s** and return an estimate of how close **s** is to the goal. Plans should be explored in an order determined by:

f(s) = cost(s) + h(s).

Your code for the cost function **cost(s)** and the heuristic function **h(s)** should be **clearly identified** for the graders to find and evaluate. It is **not a requirement** for this assignment that your heuristic function lead to optimal solutions, but we would take it into consideration if you show us it does.

Initial Game Board Preconditions:

- You can safely assume that there will be **at least one** way to achieve **victory** from the initial input field.

Input File:

The **Act-Man-II** dungeon specification input file will have the same format as the one from *Puzzle Assignment #3*.

Sample Input File #0:

```
5 15
###################
#   #     D @ G#
#A#     ###   # #
#   #     D   G#
###################
```

Output File:

The output file follows the same format specification as the one from *Puzzle Assignment #3*.

Sample Output File #0:

```
9633E
46
#####################
#    #         @   #
# #  #    @###    # #
#    #A     @      #
#####################
```

Note: Your output may not necessarily be the same as the sample, as your program may find another solution that satisfies the requirements.

Sample Input File #1:

```
9 12
################
#         G   D#
#      ### ### #
#      ### ### #
# ###   A      #
#      # ### ###
#      # ### ###
#        D   G###
################
```

Sample Output File #1:

```
4123647
63
################
#              #
#      ### ### #
#      ### ### #
# ###          #
#      # ### ###
#    A#@### ###
#             ###
################
```

Sample Input File #2:

```
8 19
###################
#D                G #
#  D#           #  #
#          A   @    D#
#       @           #
#G  #           #  D#
# G                G #
###################
```

Sample Output File #2:

```
394639666
81
###################
#                 #
#  #           #  #
#  @         @  A #
#       @         @ #
#  #           #  #
#                 #
###################
```

Sample Input File #3:

```
13 13
#############
## D ##G## D ##
# ##### # ##### #
##G## ##### ##G##
##### ######### #####
### ########### ###
#A ##########  #
### ########### ###
##### ######### #####
### # ##### # ###
# ##### # ##### #
```

```
###   ### ###   ###
####################
```

Sample Output File #3:

```
38N46282
48
####################
###     # #     ###
# ### # ### # ### #
### # ### # ### ###
#### ######## ####
###@##########  ###
#   ##########   #
###A##########  ###
#### ######## ####
### # ### # ### ###
# ### # ### # ### #
###   ### ###   ###
####################
```

Notes:

- You will need to implement your own heuristic function **h(s)** in order to implement **A*GS.** Different heuristic functions may lead to **different** solutions, and some heuristics may even miss optimal solutions.

Submission:

Place all code/scripts in your git repository in the course's GitLab server, [link] (You should have the repo open this week).

Your main file shall be called "**`puzzle4`**" regardless of extension. (e.g. if you are programming in C++, your main file should be called "`puzzle4.cpp`". If you are programming in Java your main file should be called "`puzzle4.java`"). Your main file should include your name. Include any other necessary files in your submission.

A file named 'ReadyForGrading.txt' will be placed in your repository.
IMPORTANT: When your assignment is complete and ready, modify the content of the 'ReadyForGrading.txt' file to 'Yes'. This will indicate to the graders that your assignment is ready to be graded.

Additionally, add to "ReadyForGrading.txt" the language of which you are coding in: **one** of the strings `c` , `c++` , `python` , `javascript` , `java` , `typescript` , `rust` , `c#`

You can implement your assignment using one of the supported programming languages, but it must be in a way compatible with the Computer Science Department's Linux machines. In order to accommodate such different languages, your submission should include a bash script named 'run.sh' that **compiles** and **runs** your program with all the necessary options and commands.

For example, the following is a possible 'run.sh' script for C++ 11.

```
#!/bin/bash
g++ -std=c++11 *.cpp -o puzzle4.ex
./puzzle4.ex $1 $2
```

An example 'run.sh' script for Python3 if the program is called 'puzzle4.py3' :

```
#!/bin/bash
python3 puzzle4.py3 $1 $2
```

An example 'run.sh' script for Java if the program is called 'puzzle4.java' :and the main class is called 'puzzle4'.

```
#!/bin/bash
javac puzzle4.java
java puzzle4 $1 $2
```

Your program will be evaluated and graded using the command:

```
./run.sh gradinginput.txt gradingoutput.txt
```

IMPORTANT: Remember to make your '`run.sh`' file executable as a script with the LINUX command:

```
chmod +x run.sh
```

Grading:

Your program will be evaluated and graded on the Computer Science department's Linux machines so your program needs to be compatible with the current system, compilers and environment.

Rubric:

Description	Weight
Correct Actions and Final State Generation.	40%
Specified Search Algorithm.	40%
Good Programming Practices (adherence to coding standards, modular design, documentation etc.)	20%

A Primer for Puzzle Assignment #4:

Intuition:

We continue exploring the **Act-Man-II** game tree, but now we explore plans by imposing into them a priority based on a *function of cost + heuristic function*. We use a priority queue to store candidate plans. The final implementation is a variant of Breadth-First search but with a **priority queue** rather than a **regular queue**.

Implementation:

We reuse from previous *Puzzle Assignments* the **GameBoard** structure and the **TransitionFunction()**.

As a reminder:

GameBoard : A "snapshot" of a **Act-Man-II** game, including location of the

of Act-Man and the monsters.

TransitionFunction (s, p) → s'
IINPUT: s : a GameBoard
p : A sequence of Act-Man actions
OUTPUT: s' : the GameBoard that results from executing p in s.

Refining A*GS Act-Man-II game:

The main change from before is the frontier's priority-queue, where sequences of actions / plans are now ordered by how far they are from the initial state as determined by a cost function plus how close they get to the final goal, as determined by a heuristic function.

```
PROCEDURE Act-Man-2-Astar-GS
INPUT s0 : the initial GameBoard
      goal(s) : boolean function that tests if the goal is true in
                GameBoard s.
      cost(s) : A cost function that returns the "cost"
             from the initial board to s.
      h(s) : A heuristic function that returns an estimate of the "cost"
             from s to the goal.

OUTPUT : a sequence of actions that takes the game from s0 to a state that
         satisfies the goal.

VAR frontier : Priority-Queue of sequences of actions.
             // Smallest priority value is highest priority

BEGIN
    enqueue [] (empty sequence) into frontier
    WHILE frontier is not empty
        dequeue sequence of actions p = [a0, a1, a2, ..., ak] from frontier
        // p has highest priority / smallest priority in frontier.
        sk ← TransitionFunction( s0, p )
        // sk is the GameBoard that results from executing p in s0

        IF goal(sk)
            RETURN p
```

```
        FOR every valid action a at sk
            px ← [a0, a1, a2, ..., ak, a]
            sx ← TransitionFunction( s0, px )
            // sx is the GameBoard that results from executing px in s0
            enqueue px into frontier with priority cost(sx) + h(sx)
            // cost() is the cost function
            // h() is the heuristic function
    ENDW
END.
```

This is, of course, just a first draft.

Further refinement:

You can experiment with different cost and heuristic functions to find the one that works the best with your program. You are **not required** to implement pruning on this assignment, but it may help it speed up execution. Many of the components carry from previous *Puzzle Assignments*, but you need to be careful as this is a different algorithm.

Good Luck!