

Computer Science 5400

Artificial Intelligence

Spring 2024

Puzzle Assignment #2

Version 24.03.06

DUE DATE : Wednesday, March 20th, 11:59:59pm

Assignment:

Create a program that implements **Breadth First Search (BFS)** over the **Act-Man-II** game to find a desired sequence of game actions.

You can read more about BFS in section 3.5.2 [[Link](#)] of the textbook. If you own the Norvig textbook, you can also read Section 3.3 for more details on BFS.

Your program's **objective** is to find a sequence of actions (for Act-Man) that achieves either: A **victory** condition, **or** that keeps Act-Man **alive** after **7** turns.

Specifications:

Your program shall read two **command line arguments**. The first argument will be the name of the **input file** and the second argument the name of the **output file** to which to write your output. 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 required postconditions.

Whatever is displayed to the screen during program execution will not have direct consequence on your grade.

Initial Game Board Preconditions:

- You can safely assume that there will be **at least one** way to achieve the desired postconditions 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 #1*.


Sample Input File #0:

```
5 11
#####
#   # @ G#
#A#   # #
#   #D  #
#####
```

Output File:

Your output file should follow the following format:

- The first line should contain the sequence of valid actions executed by Act-Man.
- Move actions will be encoded using *numeric keypad* order.

7	8	9
4		6
1	2	3

- The firing of the magic bullet action will be encoded with the letters 'N','S','E','W'.
- The next line should contain a single integer: The score attained by Act-Man. after executing the sequence of moves.
- The next lines should display, in a format similar to the input file, the final

configuration of the dungeon after executing the sequence of moves.

- You must follow the format presented or else your submission will not be graded

Sample Output File #0:

```
94E
38
#####
#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:

```
444E
47
#####
```

```
#      ## ## #
# ## ## ## #
# ## ## ## #
#  A @@ @@ #
# ## # ## ##
# ## # ## ##
#      # ## ##
#####
```

Sample Input File #2:

```
8 11
#####
#      G  #
#      #
#  @A  D#
#      #
#  @  D#
#      G #
#####
```

Sample Output File #2:

```
6N9
48
#####
#      #
#      @A #
#  @  @  #
#      #
#  @  #
#      #
#####
```

Sample Input File #3:

```
9 9
#####
```

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

Sample Output File #3:

```
1333999  
43  
#####  
## # #G##  
### # #  
## #####  
# #####A#  
## ##G##  
# #D#D# #  
# ## ##  
#####
```

Link to samples:

<https://web.mst.edu/ricardom/CS5400/24.1/hw2samples/>

Notes:

- As you are using **Breadth-First Search**, If there is a way to achieve victory in less than 7 moves, your program should find it.

Submission:

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

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.

Your main file shall be called “**puzzle2**” regardless of extension. (e.g. if you are programming in C++, your main file should be called “puzzle2.cpp”. If you are programming in Java your main file should be called “puzzle2.java”). Your main file should include your **name**. Include any other necessary files in your submission.

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 puzzle2.cpp -o puzzle2.ex
./puzzle2.ex $1 $2
```

A sample 'run.sh' script for **Python3** if the program is called 'puzzle2.py':

```
#!/bin/bash
python3 puzzle2.py $1 $2
```

A sample 'run.sh' script for **Java** if the program is called 'puzzle2.java': and the main class is called 'puzzle2'.

```
#!/bin/bash
javac puzzle2.java
java puzzle2 $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.

Grading Weights:

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.

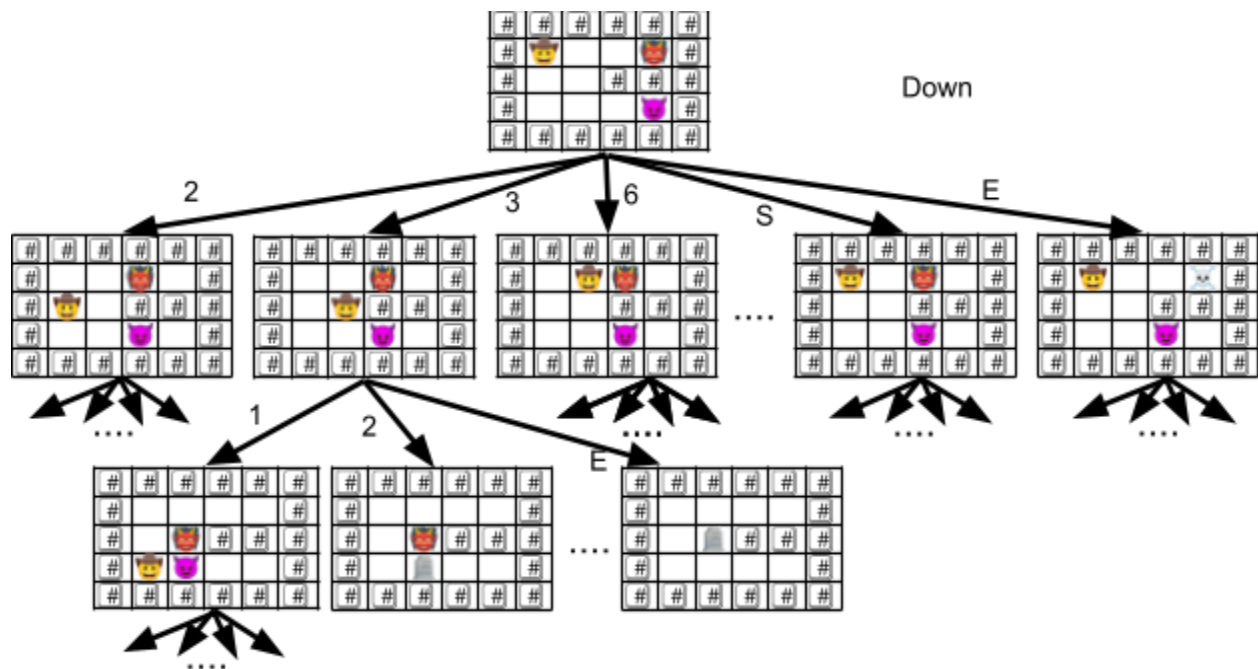
Description	Weight
Correct Actions and Final State Generation.	40%
Specified Search Algorithm.	40%
Good Programming Practices (adherence to coding standards, modular design, documentation etc.)	20%

A Primer for Puzzle Assignment #2:

Intuition:

We can represent an **Act-Man-II** game as a tree, where nodes of the tree are dungeon configurations and edges of the tree are actions (Act-Man actions) that take the game from one configuration to another. This "**Game Tree**" is the

mathematical graph over which we apply the search algorithms seen in this class.



Implementation:

From *Puzzle Assignment #1*, you should be able to develop a data structure like the following one:

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

From there, you could develop a function similar to the following:

TransitionFunction (s, p) $\rightarrow s'$

INPUT: s : a GameBoard

p : A sequence of Act-Man actions

OUTPUT: s' : the GameBoard that results from executing p in s .

Refining BFS for the Act-Man-II game:

The GameBoard data structure and the TransitionFunction() can be used to develop a first draft of an adaptation of the generic search algorithm for BFS and for the **Act-Man-II** game.

Instead of storing paths as a sequence of states, we can also store paths as a

sequence of actions. We can call these sequences of actions *candidate plans*.

PROCEDURE Act-Man-2-BFS

INPUT s0 : the initial GameBoard

goal(s) : boolean function that tests if the goal is true in
GameBoard s.

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

VAR frontier : FIFO Queue of sequences of actions.

BEGIN

enqueue [] (empty sequence) into frontier

WHILE frontier is not empty

 dequeue sequence of actions p = [a0, a1, a2, ..., ak] from 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

 enqueue [a0, a1, a2, ..., ak, a] into frontier

 ENDW

END.

This is, of course, just a first draft towards implementing *Puzzle Assignment #2*.

There are plenty of other details still to be taken care of.

Further refinement:

You can of course modify these suggestions to better suit your design ideas. (This is a sketch after all). For example, TransitionFunction() repeats a lot of its computations during the search because every new candidate plan is an extension of a previously processed candidate plan. The elements of the frontier could be made more complex in order to avoid this repetition.

It may also be possible for two different sequences of actions to lead to the exact same game configuration. A clever ordering of the actions or an extra data structure `SetOfGameBoards` may be used to detect and avoid this kind of repetition.

Good Luck!