Computer Science 5400
Artificial Intelligence

Spring 2022

Puzzle Assignment #2

Version 22.02.27

DUE DATE : Monday, March 7th, 11:59:59pm

Assignment:

Create a program that implements **Breadth First Search** (**BFS**) over the Pengu puzzle 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 Pengu) that achieves a score of at least **8**.

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 a Pengu game board specification, in the same format used for the *Puzzle Assignment #1*. Your program shall write to the output file a sequence of actions for Pengu that achieves the required score of at least **8** and the final conditions of the grid board.
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 for Pengu to achieve the desired score from the initial input game board.

Input File:

The Pengu game specification input file will have the same format as the one from *Puzzle Assignment #1*.

Input File Example:

```
6 7
#######
#P# *0#
#*0*U*#
#0 S*##
#***S*#
#######
```

Output File:

Your output file should follow the following format:

- The first line shall contain a sequence of valid moves that if executed by Pengu in the input file initial game board achieves a score of at least **8**.
- Moves will be encoded using *numeric keypad* order.

- The next line should contain a single integer: The score attained by Pengu. after executing the sequence of moves.
- The next lines should display, in a format similar to the input file, the final configurations of the game board after executing the sequence of moves.
- If Pengu is **dead** at the end of the sequence of moves, indicate Pengu's final

location with an 'X' character

- You **must** follow the input/output format presented or else your submission will not be graded.

Output File Example:

```
2924931
8
#########
# #  0#
# 0 U #
#0 S ##
#  PS*#
#########
```

In the above example output file the first line is the sequence of moves executed by Pengu[1], the next line shows that a score of 8 is attained, and the next 6 lines display the final configurations of the game board after the execution of the shown sequence of moves

Submission:

Place all code/scripts in **this assignment's** git repository in the course's GitLab server, [link] (If you still do not have a repo by next Friday, let a TA know).
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, in your "`ReadyForGrading.txt`" file, you will put the language of which you are coding in. Only put **one** of the strings below:

```
c
c++
python
```

[1] "229314" is a shorter plan that also achieves score 8

```
javascript
java
typescript
rust
c#
```

Copy **one** of these words exactly into "`ReadyForGrading.txt`"

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 hw1.ex
./hw1.ex $1 $2
```

An example '`run.sh`' script for **Python3** if the program is called '`hw1.py3`' :

```
#!/bin/bash
python3 hw1.py3 $1 $2
```

An example '`run.sh`' script for **Java** if the program is called '`hw1.java`' :and the main class is called '`hw1`'.

```
#!/bin/bash
javac hw1.java
java hw1 $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:

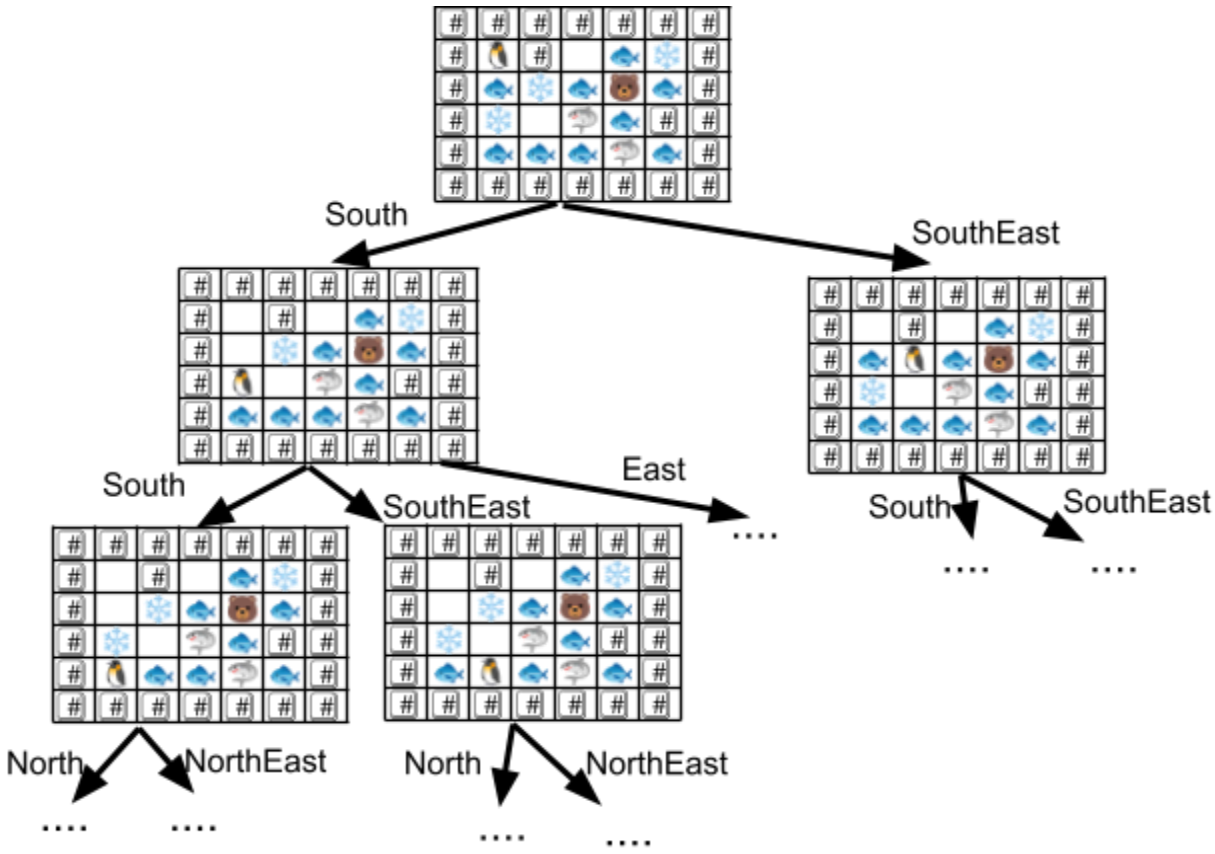
Your program will be evaluated and graded on the Computer Science department's Linux machines so your program needs to be compatible with the current system, compilers and environment.

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 a game of Pengu and its evolution as a tree, where nodes of the tree are Pengu game board configurations and edges of the tree are Pengu's 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.



A sample section of a game tree.

Implementation:

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

> **GameBoard** : A "snapshot" of an Pengu game, including location of the walls, snow cells, hazards and fish, the score, the location of Pengu.

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

TransitionFunction (s, p) → s'
INPUT: s : a GameBoard
p : A sequence of Pengu actions
OUTPUT: s' : the GameBoard that results from executing p in s.

Refining BFS for Pengu:

The GameBoard data structure and the TransistionFunction() can be used to develop a first draft of an adaptation of the generic search algorithm for BFS and for the Pengu 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 Pengu-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, some things never change no matter Pengu's actions (the location of the walls, the snow cells and hazards) so they may not need to be copied on every instance of a `GameBoard`.
Also, `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. An extra data structure `Set of GameBoards` may be used to detect and avoid this kind of repetition.

Good Luck!