cmput396 **asn1** **due in class 0800 Sep 20** **24 marks**

1. **If you leave any part of this question blank, your assignment will not be marked and its weight will be transferred to the final exam.** Print the name and ID number of each group member (at most 4) for this assignment:

   _____

   _____

   _____

   _____

   Acknowledge **all** sources, including all references and all people not in your group with whom you discussed any part of any question (for each discussion, list the relevant questions) (continue on the back of this page if there is insufficient space):

   Each group member must read, agree to, and sign this statement:

   I am familiar with the Code of Student Behaviour. I understand that are significant penalties for any infraction of this Code.

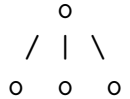
   _____

   _____

   _____

   _____

2. A search tree is *uniform* if all the leaves are at the same depth, and all non-leaves have the same number of children (called the *branching factor*). Below is a uniform search tree with depth 1, branching factor 3. (a) Draw a search tree with depth 2, branching factor 4.



   (c) Give the number of leaves of a uniform search tree with depth k, branching factor d. Explain.

3. Here is the output from `rmaze.py` with input `m10.txt`. For each time-step interval (i.e., between the printing of 2 successive diagrams), explain what search operations occur.

   E.g.: in the first interval, the algorithm picks the left neighbour (how do we know this?) of the starting cell, finds that cell empty, so goes there.

```
X  X  X  X  X  X  X
X        +  !     X
X  X  X  X  X  X  X

X  X  X  X  X  X  X
X     ?  +  !     X
X  X  X  X  X  X  X

X  X  X  X  X  X  X
X  ?     +  !     X
X  X  X  X  X  X  X

X  X  X  X  X  X  X
X     ?  +  !     X
X  X  X  X  X  X  X

X  X  X  X  X  X  X
X        +  !     X
X  X  X  X  X  X  X

finish at location (1, 4)
```

4. For `rmaze.py`, comment out the line that shuffles the neighbour offsets, and then run the program on `m10.txt`. What happens? Explain carefully.

5. In `maze.py`, the tuple (0,-1) of `nbr_offsets` corresponds to moving left: explain why.

   Which tuple corresponds to moving down? explain. Hint: run `maze.py` on `m12.txt` after commenting out the line that shuffles `nbr_offsets`.

6. For `maze.py`, comment out the line that shuffles `nbr_offsets`, and run with input `m12.txt`. On the left diagram, In order, show the first 6 cells on which the tilde symbol appears. Explain briefly. On the right diagram, repeat for the other version of `maze.py`, i.e. when `pop()` is replaced with `popleft()`.

```
X X X X X X X
X . . . . . X
X . . . . . X
X . . + . . X
X . . . . . X
X . . . . ! X
X X X X X X X
```

```
X X X X X X X
X . . . . . X
X . . . . . X
X . . + . . X
X . . . . . X
X . . . . ! X
X X X X X X X
```

7. Below is an iterative function `wander( )` that can replace the recursive function `rwander( )` in the program `/simple/maze/rmaze.py`, The code has been scrambled, and indentation removed.

   Beside it, rearrange, and properly indent, this function.

   Hint. This function behaves similarly to `rwander( )`, except that it is iterative and it has an iterations counter. Once the destination is found, execution breaks from the main `while` loop with the `return` statement. Whenever a character that is not the destination and not a wall is found, then — after the necessary computation — `break` is called, causing execution to leave the for loop and go to the start of the body of the while loop. After you have written your function, you can test it on `m.txt` to make sure that it works.

```
break
def wander(self,psn):
elif new_ch != wall_ch:
for shift in nbr_offsets:
if new_ch == dest_ch:
if self.char_at(psn) == curr_ch:
if self.char_at(psn) == empt_ch:
new_ch = self.char_at(new_psn)
new_psn = psn[0]+shift[0],psn[1]+shift[1]
num_iterations += 1
num_iterations = 0
print(num_iterations,'looks')
psn = new_psn
return new_psn
self.mark_location(psn,curr_ch)
self.mark_location(psn,empt_ch)
self.showpretty()
shuffle(nbr_offsets)
while True:
```

8. Draw the associated cell adjacency graph for this maze. Label nodes row-by-row in alphabetic order (as in the next question).

```
X  X  X  X  X  X
X  .  X  .  .  X
X  .  X  .  .  X
X  .  .  .  X  X
X  X  X  X  X  X
```

9. Here is the cell adjacency graph of a maze. For a search from `f`, assuming neighbour lists are stored in alphabetic order, show the order in which cells are first seen with (i) breadth-first-search (ii) depth-first-search (iii) iterative depth-first-search

```
a - b - c       d
|   |           |
e - f       g - h
    |       |
    i - j - k
```

10. In the following questions, *state* means *state of the sliding tile puzzle.*

    Show how the start state can be transformed into the finish, by drawing intermediate states.

```
start 4 x x
      1 x _
```

```
finish   1 x x
         4 x _
```

11. (a) For the diagram in the diagram labelled **search sliding tile space** of the webnotes, draw the next level of the search diagram (i.e. the next level of the search tree).

    (b) Draw the top 4 levels (depth 0, 1, 2, 3) of the bfs tree for the state below.

    The root node is the start state. Label children in this order: created by a *tile slide* (not a blank move) left, right, up, down respectively. In the tree, circle the nodes that are in the queue immediately before the first state of level 5 (depth 4) is removed from the queue.

```
4 7 3
2 5
1 8 6
```

12. On my office desktop, running `simple/stile/stile_search.py` on an unsolvable 3×3 state takes about 2.57 seconds for 181440 iterations, so about 70600 iterations per second.

    (a) Give the total number of 3×3 states. Explain.

(b) Give the number of unsolvable 3×4 states. Explain.

(c) Estimate the runtime, in hours, of the above algorithm on an unsolvable 3×4 state. Explain.

(d) Repeat (c) for an unsolvable 4×4 state. Give the answer in years. Explain.

13. An *inversion* of a sequence is a pair of elements x, y such that x precedes y in the sequence, but $x > y$. E.g. (3 1 2) has 2 inversions: 31 is an inversion, 32 is an inversion, 12 is not an inversion. E.g. (3 5 1 2 4) has 5 inversions. The number of inversions of a sliding tile state is the number of inversions of the associated sequence *that do not involve the blank*.

```
state S 3 1 5    state T  3 11  9  6
        6 _ 2             _  2 10  8
        7 4 8             7  4  5  1
```

E.g. S has sequence `(3 1 5 6 _ 2 7 4 8)`, with 7 inversions: 31, 32, 52, 54, 62, 64, 74.

(a) For each of the 4 slide transitions from S, give the new state, and its number of inversions.

(b) Is S solvable? Explain carefully. Is T solvable? Explain carefully.

(c) Consider a 9×9 state, with the blank in the center. When you write the state as a sequence, with sequence position 1, in what position is the blank?

For each of the 4 states reachable from this state by one tile slide, give the new position of the blank in the sequence, and give the change in the number of inversions.