Note that for many problems, multiple answers may be correct. Solutions are provided to give examples of correct solutions, not to indicate that all other possible solutions are wrong.

Work on the following problems in groups of three. Make sure everyone in your group understands the problem before moving on to the next; you’re not necessarily expected to finish all the problems.

I. Search Space Formulation

The $n$-queens problem is a popular problem originally proposed by a chess player in 1848. It requires putting $n$ queens on an $n \times n$ chess board such that none of the queens is attacking any of the other queens. In case you’re unfamiliar with chess, a queen can move in any direction for any number of squares:

Concretely, the picture on the right, above, shows a solution to the 4-queens problem. Finding a solution to the $n$-queens problem can be thought of as a search problem.

1) Formulate $n$-queens as a search problem. This means you should specify:
   a. Start state: empty board
   b. Successor function: add a queen to an empty square if doing so results in $n$ or fewer queens
   c. Search space (what does a valid state in the search tree look like): chess board with $n$ or fewer queens, each on a different square
   d. Goal test: a state is the goal if it has exactly $n$ queens and no queen is attacking any other queen (obviously, this can be stated more formally)

2) Like many search problems, the $n$-queens problem can be parameterized in a number of different ways to create a valid search problem. Come up with a different way of formulating it as a search problem, specifying the same four items as in (1).
Your new formulation shouldn’t just change the way you encode the problem but should also change the successor function.

Many solutions possible: could make successor function add queens more selectively, creating a more expensive successor function but with fewer valid states.

4) Does either of your search formulations have any advantages over the other? Think about this in terms of memory and time requirements, and the number of nodes you’d expect to expand before finding a solution.

Answers will vary – different ways of formulating the search will result in different branching factors; large branching factors are problematic for search time, especially for BFS.

5) *N*-queens can also be solved using recursion. For a little Python practice, write a recursive program to solve *n*-queens. (**Come back to this one after you’ve done the rest of the worksheet or solve it on your own after class.**)

We assume that we have a function isValid checks if we can add a queen at (row, col) given a list of current placements of queens (abstracting away the chess-rule specific part of the problem). We then return a list of solutions, where a solution is a list of length n, where the ith item in the list is the column number of the queen in row i.

```python
def nQueens(n, boardWidth):
    if n==0: # we're done
        return [[]]
    else:
        return addQueen(n-1, boardWidth, nQueens(n-1, boardWidth))

def addQueen(row, boardWidth, solutions):
    newSolutions = []
    for curSol in solutions:
        for col in range(boardWidth):
            if isValid(row, col, curSol): # safe to add a queen at this place in the board
                newSolutions.append(curSol + [col])
    return newSolutions
```
II. Search techniques

Maia just moved to a new apartment, and has to figure out the shortest paths to various places from her new place.

1.) Draw the first two levels of the search tree for creating a route from Home to Soda. Assume for all parts of this problem that when we expand a node, successors are ordered alphabetically from left to right.

2.) What happens if we naively try to do depth-first tree search for this problem?

*Infinitely go back and forth between states (search tree has infinite depth)*
3.) How can we get around the problem in (2)?

Use graph search; keep a list of nodes we've already expanded and don't expand them again.

4.) Draw the search tree for the following approaches for finding a route from Home to Soda, assuming a graph search algorithm, and show only the nodes expanded by each search. How many nodes are expanded in each search? What is the cost of the route found by each search?

a. Breadth-first search

Nodes expanded: 6

Route cost: 8

b. Depth-first search

Nodes expanded: 4

Route cost: 12
c. Greedy search

This problem was a bit ill-defined... it should have had a heuristic given but we hadn’t really learned heuristics. The solution below shows expanding the node that is closest to an already expanded node – this isn’t strictly a real heuristic since heuristics must reference the goal. The takeaway point for this is that following the shortest next path can lead you in to bad spaces in the search space and doesn’t guarantee that you’ll find an optimal path.

Nodes expanded: 4

Route cost: 12
d. Uniform-cost search

Nodes expanded: 6

Route cost: 8

(in this case, UCS is same as BFS – this isn’t guaranteed!)

5.) Which of the above methods techniques found the shortest route? Did any find the shortest possible route? Are any of them guaranteed to find the shortest possible route?

Uniform cost search and breadth first search; yes, this is the shortest route; UCS is if all costs are > ε

6.) For each search technique, give one pro and one con of using the technique. Hint: think about things like optimality, time and space complexity, whether the technique is guaranteed to terminate, etc.

a. Breadth first search

Pro: complete if amount of branching is finite

Con: exponential in time and space

b. Depth first search

Pro: linear space complexity

Con: not complete

c. Uniform cost search

Pro: optimal for costs greater than epsilon

Con: like BFS, exponential in time and space

III. Formulating problems as search

Think of a problem that doesn’t involve route-planning but that can be formulated as search; for example, can your favorite game be thought of as a search problem? How about planning your schedule? With your group, pick a problem that seems interesting and come up with:

a. The start state

b. Successor function

c. State space

d. How to determine if you’re in the goal state