Guidelines – Read Carefully! Please check each problem for problem-specific instructions. You are provided a zip file containing a single folder named xyz007 with the following folders and files:

xyz007
xyz007/mp440.py
xyz007/main.py

You should first rename the folder name xyz007 to be your NETID. If you are working in a team, the folder should be named with both group members' NETIDs, in the format of NETID1_NETID2. The folder name letters can be either upper or lower case. When you are ready to submit, remove any extra files (e.g., some python interpreter will create .pyc files) that are not required and zip the entire folder. The zip file should also be named with your NETID as NETID.zip (or NETID1_NETID2.zip for groups with two members). For this MP, main.py may be removed at the time of submission.

You are required to write your program adhering to Python 2.7 standards. In particular, DO NOT use Python 3.x. Beside the default libraries supplied in the standard Python distribution, you may use ONLY numpy and matplotlib libraries.

You should only work on mp440.py and do not create additional python files. The file main.py is for you to test your implementations. Note that during grading, your implementation will be called in different ways than what are in main.py, which serve only as minimal sanity checks. You should not use any variable or functions from main.py in your code. All the functions that you should implement are specified in the skeleton mp440.py file.

As mentioned in class, you may form groups of up to two students. Only a single student should submit the solution.

Problem 1 [30 points]. Implementation of a standard hill-climbing algorithm for n-queens. You need to implement four functions, first of which is one that creates a random state

`get_random_state(`n`)`

where the returned state is a list of n values, with the i-th value in the list corresponding to the $(i+1)$-th queen's row location in the $(i+1)$-th column. You should use some random number generators to do this (5 points). The second function you need to implement is for computing the number of attacking queens in a given state:

`compute_attacking_pairs(state)`

The computation should be done in a way identical to what was covered in our class (5 points). With these implemented, you can proceed to implement

`hill_descending_n_queens(state, comp_att_pairs)`

where the argument to be passed in is a state plus the function for computing the number of conflicting pairs. This should be a standard hill-climbing (in this case, descending) algorithm without random restarts nor side moves (10 points). It should return the final state by the search.

Finally, implement

`n_queens(`n`, get_rand_st, comp_att_pairs, hill_descending)`

that will be called with the previous three functions as parameters. This function should try random restarts if a randomly generated state state fails to yield a feasible n-queens configuration (10 points). Note that your code should finish running in a few second for up to $n = 20$ queens.

Problem 2 [70 points]. Search algorithms. In this problem you are to implement the queue management functionality for DFS, BFS, Uniform-cost, and A* best-first search. Only graph search is required. In main.py, there is an illustration of the data structure used to encode a graph, which is a dictionary indexed by numerical node index numbers from 1 to n for an n-vertex graph. For each node, a multi-type list is used to store *(i)* the node index (node_id), *(ii)* the visited flag, *(iii)* the parent after the cost-to-come of the node becomes final, *(iv)* the cost-to-come for the node, *(v)* the heuristic value for the node, *(vi)* the neighbors as a list of id-cost tuples, and *(vii)* the id-cost tuple stored in a dictionary.

For each X where X $\in$ {DFS, BFS, UC, ASTAR}, you are to implement three functions

```
add_to_queue_X(node_id, parent_node_id, cost, initialize)
is_queue_empty_X()
pop_front_X()
```

Where an item in a queue contains a tuple of the form `(node_id, parent_node_id)`. For DFS and BFS, there is no need to maintain the cost. You need to use one or more global variables in your code to hold the queue between calls to these functions. In `add_to_queue_X`, the last parameter `initialize` is set to true only when it is called the first time in a search when the start node is inserted into the queue. You should use this chance to initialize your internal queue data structure.

For each of the search methods, correct implementation will get you 15–20 points. For DFS and BFS (15 points each), there should be a deterministic path and for UC and A* (20 points each), the solution should also be mostly deterministic (there may be some variance due to tie-breaking). For UC and A* with consistent heuristics, the solution path should always be optimal. Note that the optimal cost is maintained for you already; there is nothing you need to do regarding the final cost if your implementation is correct.

Keep in mind that for actual grading, we will use larger graphs with tens of nodes.

A sample run of main.py, if you have correctly implemented your side, should produce the results shown as follows.

```
Graph search result dump
DFS path:  [1, 3, 4, 5], cost:  9, #expansions:  4
BFS path:  [1, 2, 5], cost:  13, #expansions:  5
UC path:  [1, 2, 3, 4, 5], cost:  8, #expansions:  5
A* (admissible) path:  [1, 3, 4, 5], cost:  9, #expansions:  4
A* (consistent) path:  [1, 2, 3, 4, 5], cost:  8, #expansions:  5

The n-queens problem
A random state:  [4, 5, 7, 5, 2, 5, 1], conflicting pairs:  5
Final state after hill-climbing:  [4, 2, 7, 5, 2, 5, 1], conflicting pairs:  2
A valid solution:  [2, 4, 6, 1, 3, 5, 7]
```

