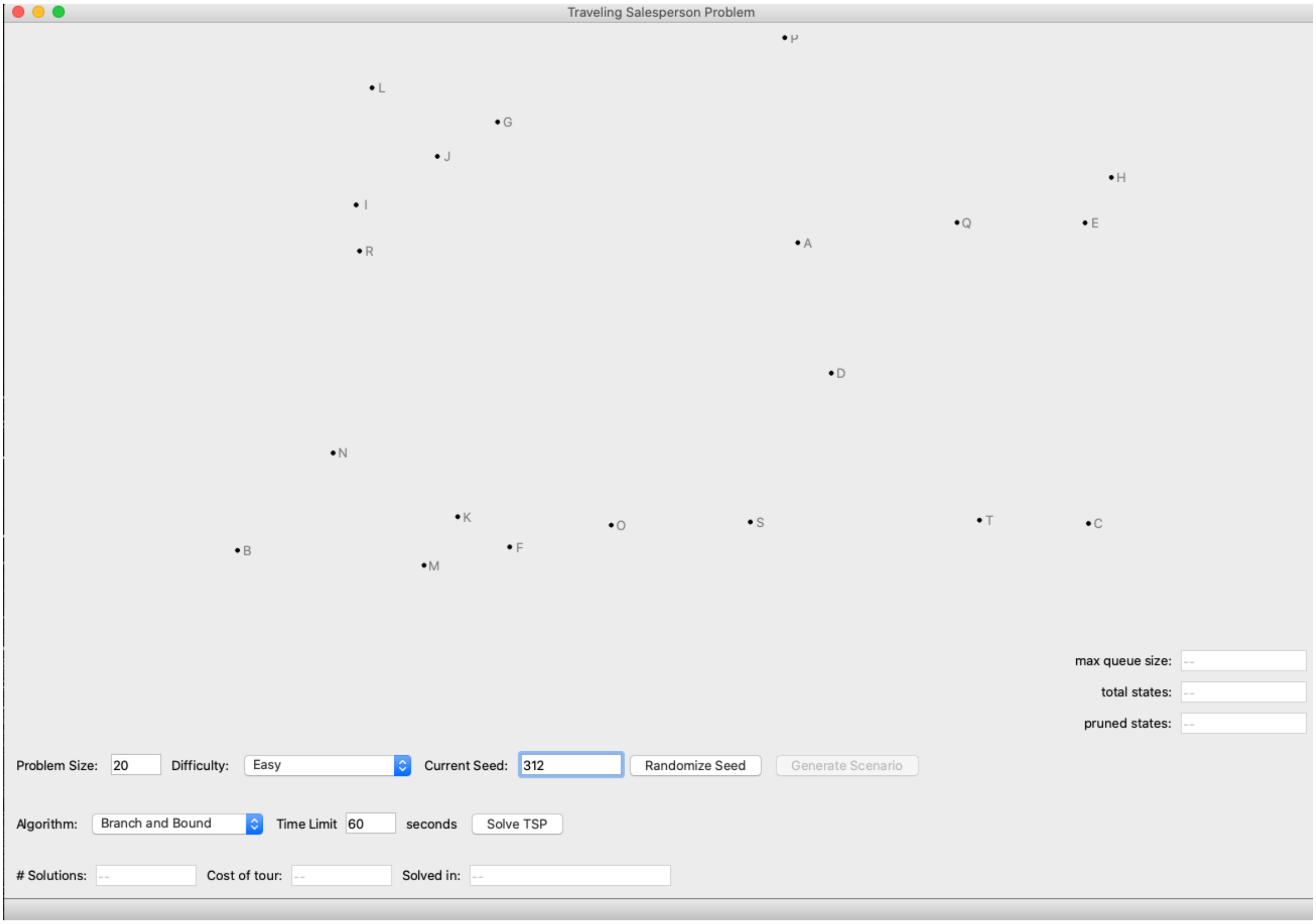
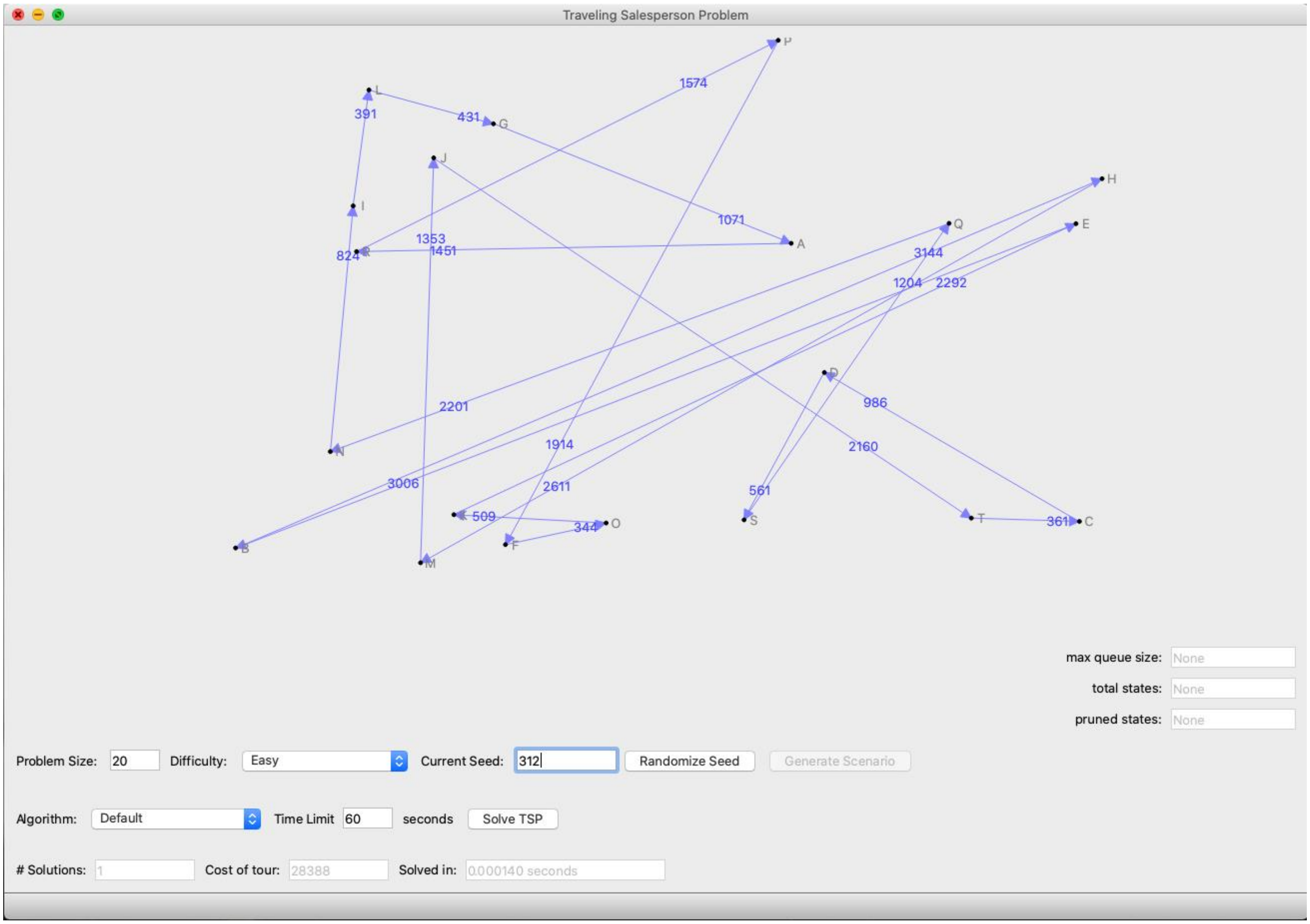
Project #5: Travelling Salesperson (Intelligent Search)

Framework: The [framework]() includes a user interface that generates a specified number of random points (the cities) on a 2-D canvas. The problems are generated based on a random seed, which you can set for debugging purposes. Clicking the "Generate Scenario" button resets the problem instance for the given "Current Seed". Clicking the "Randomize Seed" button chooses a new 3-digit random seed and updates the "Current Seed" field. You can control the problem size using the "Problem Size" field, as shown in the following figure:



The "Difficulty" drop-down menu allows you to select from one of three problem difficulty levels: Easy, Normal and Hard. The "Algorithm" pop-up menu allows you to select different algorithms for solving the problem. A simple random tour "Default" algorithm is already implemented. In the following figure, that default algorithm is run on a random problem of size 20 of difficulty level Normal with a random seed of 312.

A word of clarification: the GUI currently has text by each city indicating the city "name". The *n* cities are A,B,...,Z,AA,BB, etc. On the solution returned by the solver, the order that the cities were visited is indicated by the arrows on each segment between cities and the corresponding length of each segment is indicated close to the arrowhead. The cost of the tour, the time spent finding the tour, and the number of solutions found are reported in the respective text boxes.



You will implement an additional algorithm (Branch and Bound) for this lab by writing code for the `TSPSolver.branchAndBound()` method. You will also use this framework for the group TSP project, in which you will implement two additional algorithms: a greedy approach and some other approach of your choosing.

To Do

1. Write a branch and bound algorithm (your TSP solver) to find the shortest complete simple tour through the `City` objects in the array `Cities`. You will use the reduced cost matrix for your lower bound function and "partial path" as your state space search approach. Implement your solver in the following method: `TSPSolver.branchAndBound()`.
2. Your solver should include a time-out mechanism so that it will terminate and report the best solution so far (BSSF) after 60 seconds of execution time. You can use the "private" member `TSPSolver._time_limit`, which is set to the default value 60 and automatically updates whenever the `Time Limit field` is edited in the application form. Note that it is not critical that you use precisely 60 seconds. Running a timer and checking the time on every iteration through your branch and bound algorithm is sufficient, if slightly imprecise. You can use timers to interrupt your search if you want to be more precise about ending exactly at 60 seconds.
3. Assign the "private" member `TSPSolver._bssf` to a `TSPSolution` object that contains the path you have discovered. You should be creative with your initial BSSF value as it can have a significant impact on early pruning.
4. To display your solution, populate the results array with the cost of the discovered tour, the elapsed time that it took you to discover it and the number of intermediate solutions considered, respectively. For an example of how to do this, you can look at the default algorithm `ProbemAndSolver.defaultRandomTour()` method.

When counting intermediate solutions, do not include your initial BSSF. Thus, the intermediate count will be 0 if the BSSF is optimal or time expires.

5. For this project, the performance analysis will focus on both time and space. You will need a mechanism to report the total number of child states generated (whether they are put on the queue or not), and also the number of generated states pruned due to your evolving BSSF. This includes all child states generated that never get expanded, either because they are not put on the queue, pruned when dequeued, or because they never get dequeued before termination. You do not need to report the even larger number of potential sub-states of your pruned states which are implicitly pruned. Report the maximum size of the queue which is the upper bound of memory used.
6. There are three difficulty levels that govern city connectivity: Easy (symmetric), Normal (asymmetric), and Hard (asymmetric and some infinite distances). You can play with all of them during testing but just use the Hard level for all of your reporting below. With the Easy level, the optimal tour cannot have crossed paths; however, in the case of the harder levels, the optimal tour may have crossed paths.
7. Most of your results should include multiple BSSF updates (# of Solutions reported in GUI), especially for smaller numbers of cities. Because this can only happen when the search reaches a leaf node (finds a complete tour), and because each state can add many children states to the queue, you must think carefully about your search strategy and come up with a priority key for the queue that implements it. While it is tempting (and probably useful) to visit states with a low bound, it is also important to find complete tours (so the BSSF can be updated and the tree can be pruned more), so your prioritization of states should consider both bound and tree depth (and anything else you can think of that improves performance---be creative).

Report: 90 points total. The other 10 come from your design experience.

1. [20] Include your well-commented code.
2. [10] Explain both the **time** and **space** complexity of your algorithm by showing and summing up the complexity of each subsection of your code. Keep in mind the following things:
   - Priority Queue
   - SearchStates
   - Reduced Cost Matrix, and updating it
   - BSSF Initialization
   - Expanding one SearchState into others
   - The full Branch and Bound algorithm
3. [5] Describe the data structures you use to represent the states.
4. [5] Describe the priority queue data structure you use and how it works.
5. [5] Describe your approach for the initial BSSF.
6. [25] Include a table containing the following columns.

# Cities	Seed	Running time (sec.)	Cost of best tour found (*=optimal)	Max # of stored states at a given time	# of BSSF updates	Total # of states created	Total # of states pruned
15	20	412	710	615	600	412	710
16	902	710	412	615	412	710	600
412	710	4.2	156*	600	1	700	40
600	615	2.8	58*	412	3	600	324
710	412	60	213	710	2	800	120
412	710	60	265	615	0	800	155

Note that the numbers in the above table are completely made up and may or may not have any correlation with reality. Your table must include at least 10 rows of results, each for a different problem ranging between 10 and 50 cities. The first two rows should report your results on the specific cities/seeds shown above (15/20 and 16/902). Of the 10 problems, 4 must run for the full 60 seconds (before timing out and returning the best solution found so far). # of BSSF updates is the number of times a solution was found which was better than the current BSSF. A value of 0 means the final solution was just the initial BSSF. Pruned states include a) those which are not put on the queue because their initial bound is greater than the current BSSF and also b) any states that are put on the priority queue, but when taken off the queue, their bound is now greater than the updated BSSF, thus allowing the state to be immediately pruned without expansion, or that are never taken off the queue at all. Just count the states actually pruned (not the many potential sub-states of those states which are also implicitly pruned).

7. [10] Discuss the results in the table and why you think the numbers are what they are, including how time complexity and pruned states vary with problem size.
8. [10] Discuss the mechanisms you tried and how effective they were in getting the state space search to dig deeper and find more solutions early.

To aid in your debugging here are some of our results you can compare against. For each we ran 5 trials in hard mode (a different set of missing edges each time) and report the medians (ranges in brackets for times and total nodes).

- problem size: 14; random seed: 1; cost of tour: 9844 ; time: ~7.23 secs [2.27-9.51s]; total nodes created: ~38K [13K-51K]
- problem size: 14; random seed: 2; cost of tour: 9549 ; time: ~1.43 secs [0.39s-67.53s]; total nodes created: ~8K [2K-286K]
- problem size: 14; random seed: 3; cost of tour: 9031 ; time: ~2.71 secs [1.11-4.93s]; total nodes created: ~14K [6K-27K]

Note that because each "hard" scenario is randomly generated with a different set of edges, the cost of the optimal tour is different, and the inherent difficulty varies (as exemplified by a couple of trials on seed 2 with 57.69 and 67.53 seconds -- the other 3 were super fast).

To make things a bit more predictable, we added a "Hard (Deterministic)" mode which uses the random seed to make sure that, just like the cities being generated in the same locations for a give size/seed pair, the edges removed will be consistent as well. So, under "Hard (Deterministic)" mode, we get the following results:

- problem size: 14; random seed: 1; cost of tour: 10573* ; time: ~8.45 secs *(median)*; total nodes created: ~46.5K
- problem size: 14; random seed: 2; cost of tour: 10061; time: 60 secs; total nodes created: ~290.2K
- problem size: 14; random seed: 3; cost of tour: 8638*; time: 2.10 secs *(median)*; total nodes created: ~11.9K