The travelling salesperson problem: A genetic approach

Isaac Hamm, George Li, Will Wardinsky, Keiffer da Silva

CS 312: Algorithm Design and Analysis

Abstract

We implemented a genetic algorithm (GA) in an attempt to solve the travelling salesperson problem (TSP). The genetic algorithm relies heavily on randomization. The randomness provided our algorithm a chance to improve on previous iterations, but it can be problematic as well. The genetic algorithm requires three functions to be effective: crossover, mutation, and survival. The crossover function takes two parents and splices them together to create a child. The mutation function creates a new child by switching two cities at random. Our survival method deletes children probabilistically according to our fitness function. The big-O of our algorithm is O(n^7) on an average case, although that fluctuated as we experimented with our implementation. The genetic algorithm was inefficient in solving the traveling salesperson problem, due to its high run time and overdependence on randomization. Like any other algorithm, the genetic method is just one tool that when implemented well, provides clarity to the traveling salesperson problem and a basis of understanding for further research.

Introduction

For our TSP project, our group decided that we would use a genetic algorithm. In this paper we will discuss our algorithm, its complexity, and our empirical results. Then we will consider why our algorithm gave us those results. Finally, we will look at future work to make our algorithm run better.

Why we chose GA

We decided on the GA because we felt that we had a good understanding of how to implement the different parts of the algorithm, based on what we had learned in class. We also saw opportunities where we felt like we could improve the algorithm to make it run swiftly and return good results. Although the notes from class were helpful, we still found that there were a lot of problems in our implementation that we needed to fix. Most of these came down to optimizing the algorithm for our use.

Problems we encountered

The problems that we had to fix were mostly—if not all—optimization problems. We needed to figure out the optimal number of generations to find a good solution while keeping the runtime to a minimum. This also applied to the number of children that we wanted to make, and our survival rate through each generation. The more children that we made, the longer the runtime. The problem is that we need a sufficient number of children to find a good solution. In the end, we found that the optimal number of children depended on the number of cities that we had to traverse through. Explanation of how we found this number is explored in greater detail in the “Analysis of results” section below.

Pros and cons of GA

Genetic algorithms rely heavily on randomization. This is both a pro and a con. The randomness allows the algorithm to find new solutions very quickly, but it also means that our algorithm could simulate an exhaustive search before it returns a solution that is better than our initial BSSF. We also experimented with running our algorithm until it found no better solutions

within a specified number of generations, so that it wouldn't loop endlessly or get stuck in local optima. The other con involved in randomness is that it can't replicate solutions that it finds: in other words, it is not a deterministic algorithm. Because of the randomness, it will return a unique solution each time it is run; old solutions die with each rerun.

One pro of our algorithm is that it does regularly beat the greedy algorithm, but that's balanced out by the con of the relative runtime it takes. Our runtime grows exponentially (between n^2 and n^5) with the size of the problem, meaning that although we can handle larger problems than B&B (which grows at 2^n), we still run out of time and space very quickly as the problem grows. Note that beating the greedy algorithm on average really isn't that impressive, as we start off with a greedy solution for our initial state. This is a pro because we never do *worse* than the greedy algorithm, but it can give a false sense of how well our algorithm *actually* performs. Without that starting point, it could run many generations and never come close to beating the greedy algorithm by itself.

The other pro of our algorithm is that it works regardless of the nature of the problem. We only dealt with relatively dense graphs in our case, but the algorithm works exactly the same on sparse graphs. Unfortunately, its runtime wouldn't improve because we treat infinite cost paths as low-fitness and still create those children; but it would be able to finish the problem and return a better solution than the greedy algorithm, given enough time to generate a sufficient number of generations.

Algorithm explanation

To run the algorithm, we implemented the following steps: generation of initial states, creation of children states, and survival/killing of non-surviving states. Note that our fitness rating is simply the cost of the entire tour, where non-complete tours have a fitness of infinity.

Generation of Initial States

For the generation of initial states, we started by generating ncities^2 number of random states using the default random algorithm included in our code. This worked well for problem sizes <= 30 and allowed us to frequently beat the greedy algorithm. However, when testing this method of initial state-generation for size 60, we found that it failed to generate 10 states within the allotted 10-minute reasonable time period; not a good method of initial state generation.

Since the problem wasn't the *number* of states that were being generated, but rather the method of generation, we sought for a different way of generating initial states. After considering a few different options, we decided that the number of initial states didn't matter quite as much as the number of children we generate thereafter, especially if we start with a good initial state and generate good children. Thus, we opted to generate greedy initial states, using a modified greedy algorithm (see included code snippet) that generates a greedy solution for each city acting as a start city (Note: we had to drop this back to just one greedy state for larger problem sizes). Many of these solutions can be the same, but many are unique as well. This can be performed in O(n^3) time where n is the number of cities, and we use a different start city each time. This allows us to generate cities in O(n^3) time because each run of the greedy algorithm takes

O(n^2) time worst-case (O(n) to find the minimum edge for O(n) cities) and we're running it O(n) times, once for each city to act as the start city. Thus, although not particularly fast, it far outpaces any other method of solution generation, and at the same time comes up with a number of fairly good parent solutions.

```
# first generation -- greedy
for i in range(ncities):
   greedy_solution = TSPSolver.greedy(self, time_allowance, i)['soln']
   if greedy_solution != math.inf:
       new_solution = GA_Solution(greedy_solution.route, greedy_solution.cost)
       solutions.append(new_solution)
```

Creation of children states

For the creation of child states, we use two methods: crossover and mutation. These methods are commonly used in genetic algorithms and are not difficult to adapt to the TSP. See the code for these two methods below.

Our crossover method is performed first on our initial/surviving states. This method takes two states at random and performs a crossover between them by choosing a random index to splice the two lists of cities (i.e., the list is split into two). We take the parent list from index 0 to the splice index and use that to generate a new route, using the following method: we add the cities in order from parent2 to a new list, skipping over the cities that are present in the spliced section of parent1, and vice-versa for the spliced section from parent2. We then add the spliced section of the list *back* into the new list generated by skipping those cities. This always guarantees that we never visit the same city twice, and can give us new orientations far away from what we originally let survive. It doesn't guarantee a complete solution, but we deal with this by letting this solution's fitness equal infinity if it does not complete a tour, and then pruning all of these solutions prior to our survival function. Note that we add these crossover lists to our solutions array *before* making mutations, so that we can consider pre- and post-mutation crossover solutions as possibilities to beat our BSSF.

Space complexity for crossover is constant as we are only ever doubling the number of states from 2 to 4 (note that this is only for one run of the function). Time complexity is O(n) where n is the size of the parent, which should match the size of the problem (each solution only holds the number of cities visited). This is demonstrated in the for loop below, where it loops through and checks the cities visited for each parent as described above. Note that it also takes O(n) time to calculate the cost of each new child, but since these are equal, O(2n) simplifies to O(n).

Our mutation method is very simple: we choose two random cities from a complete solution and switch their order in that solution, then recalculate the cost based on this new ordering. This allows us to quickly generate a lot of new, randomized solutions. Each of these can be computed in O(n) time because it takes O(n) time to calculate the cost, and switching cities is constant. Space complexity is constant for the same reasons as crossover, because we are only ever doubling from one parent to a second child.

One implementation that we hoped to code but didn't have time to implement was including probability randomness such that better (shorter) paths were more likely to be chosen for both crossovers and mutations. We outline this in more detail in the "Future work" section below, but mention here that we were unable to implement this probability randomization due to the limited amount of time that we had to work on the project.

```
def crossover(self, parent1, parent2):
   splice_index = random.randint(0, len(parent1.cities_visited) - 1)

   spliced_route1 = parent1.cities_visited[:splice_index]
   spliced_route2 = parent2.cities_visited[:splice_index]

   child_route1 = GA_Solution()
   child_route2 = GA_Solution()

   for i in range(len(parent1.cities_visited)): #n
       if parent2.cities_visited[i] not in spliced_route1:
           child_route1.cities_visited.append(parent2.cities_visited[i])
       if parent1.cities_visited[i] not in spliced_route2:
           child_route2.cities_visited.append(parent1.cities_visited[i])

   child_route1.cities_visited.extend(spliced_route1)
   child_route2.cities_visited.extend(spliced_route2)

   child_route1.calculate_cost()
   child_route2.calculate_cost()

   return child_route1, child_route2
```

```
def mutation(self, route, generation, generation_limit):

      index1 = random.randint(0, len(route.cities_visited) - 1)
      index2 = random.randint(0, len(route.cities_visited) - 1)
      route.cities_visited[index1], route.cities_visited[index2] =
          route.cities_visited[index2],route.cities_visited[index1]
      route.calculate_cost()
      return route
```

Survival

Our method for survival was designed to keep the best solutions without getting stuck in local optima. To do this, we split the list of generated solutions into equal-sized buckets. The number of buckets is equal to the number of children that we want to survive (which is the same as the number of children we generate). Then, we sort each bucket by solution cost and create weights for that new sorted list such that lower-cost (or better fit) solutions are more likely — but not guaranteed — to be chosen. We then choose one child from each bucket to survive and let the rest die off.

This method works quite well, and gives us a good balance of continued randomness with probability of choosing better solutions each time, but we're not sure that our weights are appropriate. We think that we could further optimize our survival by experimenting with our weighting system and making it more likely to choose better solutions; right now, we still end up with a lot of bad solutions that we have to sift through.

Time complexity for survival is O(n^6). n^5 is the number of children we let survive based on our generation ratio (variable `number_of_wanted_children` in the code below), and in a worst case, the for loop inside of the while loop could also be the same as the city size, leading to another n. Space complexity for this is weird, as we are only cutting things out in this method. We have to hold a new survivors array, which will be equal to n^5 where n is the problem size (this comes from our generation ratio). Weights and players array will never surpass this n^5 size as our buckets are divisions of this, because we are dividing our list of parents into small bucket sizes, and weights and players will equal the size of the buckets.

```
def survival(self, list_of_children, num_survive):
    number_of_wanted_children = num_survive

    survivors = []

    while len(survivors) != number_of_wanted_children:

        weights = []
        player = []
        wanted_num = len(list_of_children) // number_of_wanted_children

        for i in range(wanted_num):
            j = random.randint(0, len(list_of_children) - 1)
            player.append(list_of_children[j])
            list_of_children[j] = list_of_children[-1]
            list_of_children.pop()

        player.sort(key=lambda x: x.cost_of_solution)

        counter = len(player) * 5
        for i in range(len(player)):
            if counter != 0:
                counter -= 5
            weights.append(counter)

        if len(weights) == 1:
            survivors.extend(player)
        else:
            survivors.extend(random.choices(player, weights))

    return survivors
```

Complexity

Time Complexity

The big-O complexity for our code fluctuates depending on our implementation. Our final version is theoretically a worst case of O(n!), but typically runs far less than this. This worst case comes from the nature of our implementation, where we break only after finding a local optima; this is demonstrated in the pseudocode below this paragraph. Because of this, it is realistic to believe that we could perform an exhaustive-level of search in the worst case. Actual case is highly dependent on the specific problem, but in practice the while loop is typically not performed more than n times, leading to a realistic average case of O(n^7) due to crossover, mutation, and survival all being O(n^6).

```
while not no_change_in_ten and time.time() - start_time < time_allowance:
    did_change = False

    for i in range(num_children #n5): #n6
        mutation #n

    for i in range(num_children - 1 #n5): #n6
        crossover #n

    for solution in solutions: #n5
        update bssf

    if did_change:
        gens_not_changed = 0
    else:
        gens_not_changed += 1

    if gens_not_changed == 10:
        no_change_in_ten = True

    solutions = self.survival(solutions, num_children) #n6
    generations += 1
```

Other implementations run in similar O(n^7) complexity — the difference being that we can guarantee O(n^7) because our while loop was limited to a bounded number of generations (theoretically, we can simplify that constant bound down to O(n^6), but our generation count almost always exceeded n, so realistically it is O(n^7)). Our worst time-complexity comes out to about O(n^11), when we use our generation ratio to calculate how many generations we make. This would make the while loop run n^5 times, leading to n^11 because we have to call the n^6 survival function each time.

We can see the complexity of our code by looking through all of the functions that we use to create our children. Starting with our crossover which would be O(n * parents), where the length of parents would be the equal to the number of children that we want to keep after each generation: this is either bounded to between 10-10,000, or calculated using the n^5 generation ratio (described in greater detail below). Therefore, parents should be O(n^5), leading to an

overall O(n^6) for crossover for our unbounded version, or O(n * *some constant*) for bounded versions. For the same reason, mutation will also be O(n * parents).

Next we have the part of our code that dictates which children survive to create the next generation. The first thing that we do is check all solutions for a new BSSF. Because we have to do this to all of our created children which would be the size of our original list of children or parents plus what was created in crossover and mutation. During crossover and mutation, we generate n^5 children for each of these, meaning that the big O would be O(n^5 + n^5 + n^5) which we can simplify to O(n^5). This loop will therefore run n^5 times, or the number of children we choose for bounded functions. The second part of our survival process, which as we have shown above is an O(n^6) function.

Space Complexity

Space complexity is actually a lot easier to figure out. We have a list that holds O(n^5) solutions. Then we would add the number of children created to that list. As stated before in the explanation for the first part of our survival method our crossover and mutation methods both create n^5 number of children then append it to the existing list. This would mean that the space complexity would simplify to n^5. Since we never hold more than this, the space complexity of our entire genetic algorithm would be O(n^5) for unbounded versions.

Analysis of results

We did some extra work in the table to show how our GA works to beat the greedy algorithm, and why it might not be a great choice above size 35 or so.

Random, greedy, and B&B algorithm performance

Our random, greedy, and branch and bound (B&B) are all typical. Unfortunately, our B&B algorithm was unable to complete anything greater than size 30 in 10 minutes (note that in table 1 for B&B instead of putting TB for values up to 200, we just ran it for 10 minutes to see how it would perform – on additional trials, we just put TB for B&B). Nonetheless, even when we let it run and then timeout for 10 minutes, B&B often beat the greedy algorithm by finding a better BSSF; we can't guarantee that it's an *optimal* solution in these cases. Note that it also often beats our implementation of the GA (see table 2), demonstrating its effectiveness at finding good solutions, even if it can't run long enough to find the optimal one.

Typical solutions for all algorithms are found in the appendix (p. 15) at the end of the paper. We kept problem sizes small and similar for visual comparison purposes.

Table 1

	Random		Greedy			Branch & Bound		
# Cities	Time (sec)	Path Length	Time (Sec)	Path Length	% of Random	Time (sec)	Path Length	% of Greedy
15	0.008	24057	0.0003	15630	64.97%	3.28	8353	53.44%
30	0.009	48540	0.0007	20639	42.51%	600	12960	62.79%
40	0.26	50,914	0.002	21,986	43.18%	600	17839	81.14%
60	52.95	86344	0.01	26,897	31.15%	600	22971	85.40%
100	TB	TB	0.04	42433	?	600	31681	74.66%
200	TB	TB	0.04	58939	?	600	57569	97.68%
500	TB	TB	1.11	103,863	?	TB	TB	?
750	TB	TB	2.92	131,981	?	TB	TB	?
1000	TB	TB	3.68	160,020	?	TB	TB	?

The generation ratio

This is where our creativity and unique approach to the problem really come into play. To optimize our GA, we calculated what we're calling the "generation ratio." This ratio is designed to optimize the likelihood of beating the greedy algorithm while minimizing the amount of time it takes to run the algorithm. Because our GA deals mostly with randomness, the odds of finding a solution are unpredictable; however, assuming that this unpredictability grows with the size of the problem, we *can* predict how often we will beat the greedy algorithm based on two factors: 1) the size of the problem, and 2) the numbers that we input into our greedy algorithm, including how many generations we make, how many children we make per generation, and how many children we allow to survive from generation to generation. Thus, our generation ratio is the ratio between the size of the problem and these input numbers.

We calculated this initially by performing trial and error on smaller problem sets. Our goal was to beat the greedy algorithm at least 50% of the time while having the algorithm also run in a reasonable amount of time (i.e., not exhaustive). Thus, we created the table and graph in Figure 1.1 (below) where x-values indicate the size of the city, and y-values indicate the number of generations required to beat the greedy algorithm for that problem size. Note that we make the number of children and the survival rate equal, double the number of generations; so if we need 45 generations (as with problem size 15), the number of children we create each generation would be 90, and the number of children that we let survive each generation is also never greater than 90.

After calculating these initial numbers through trial and error, we ran regression analyses using different types of polynomial functions to see which magnitude would be the best fit (we didn't explore exponential functions, as that would just recreate an exhaustive search time complexity, unable to beat B&B). Figure 1.2 shows these analyses, and figure 1.3 shows the best fit of an n^5 order of magnitude. We then multiplied the size of the problem by n^5, m, and b and used the accompanying coefficients (m = 0.0000233385, b = 23.0863) to calculate the number of generations and children/survival rates. This is the generation ratio.

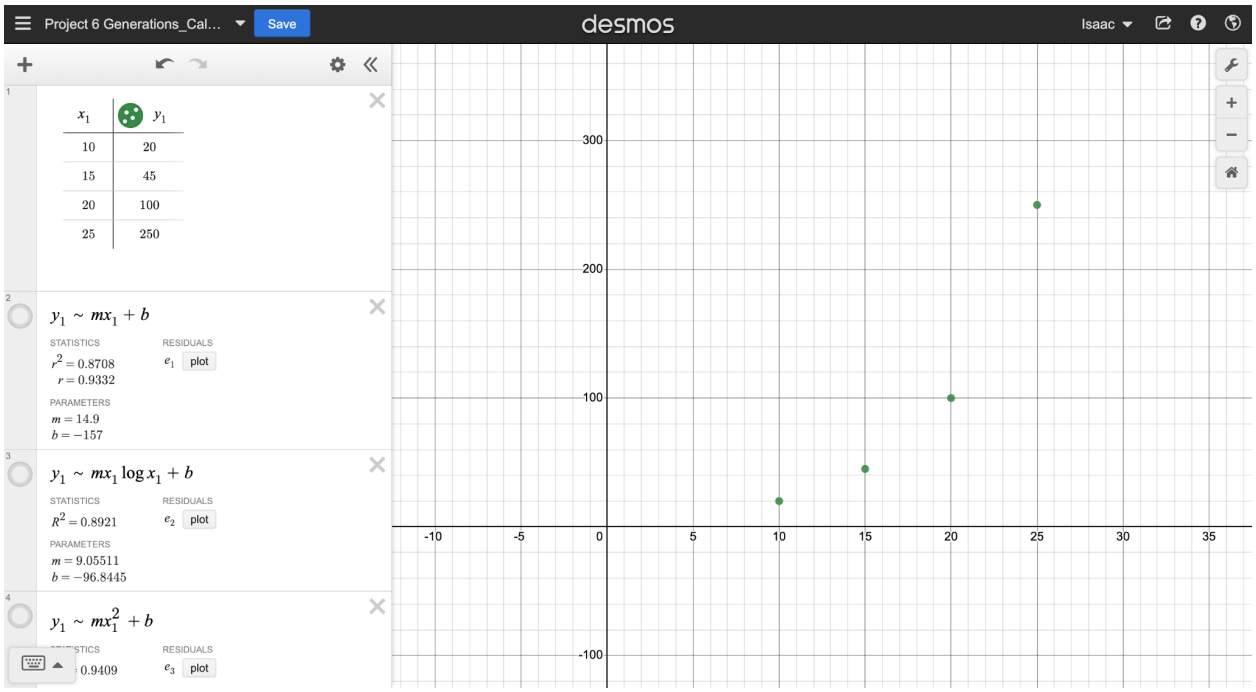


Figure 1.1

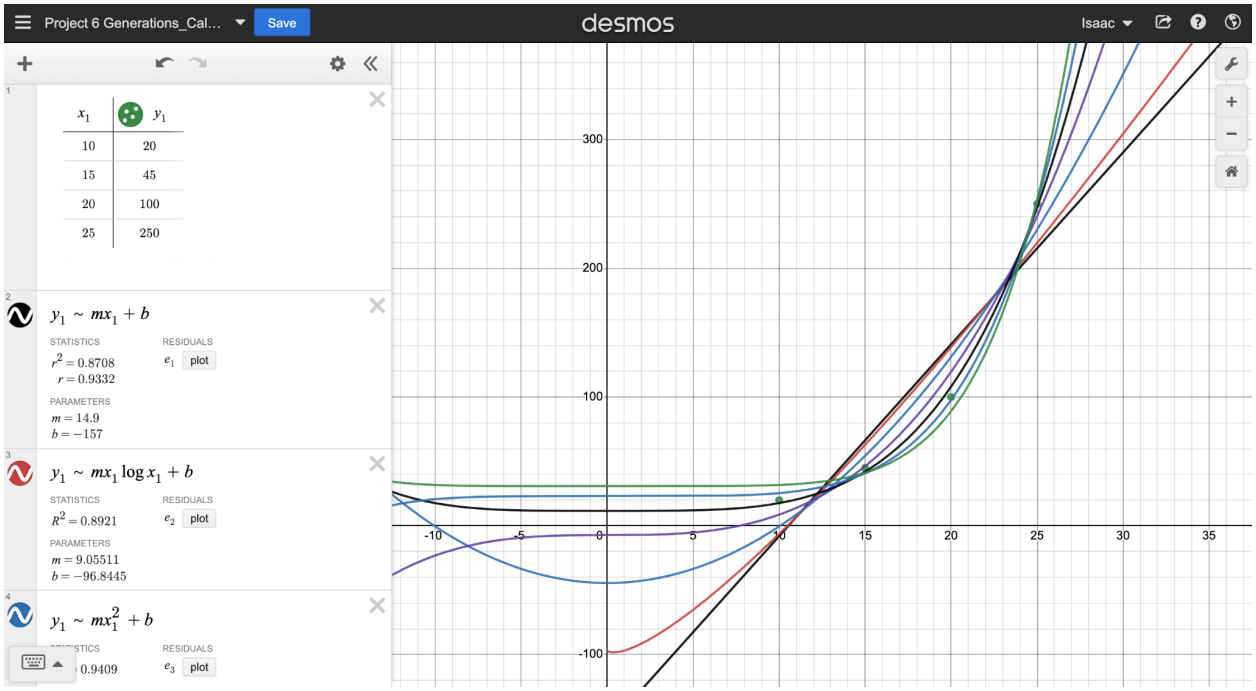


Figure 1.2

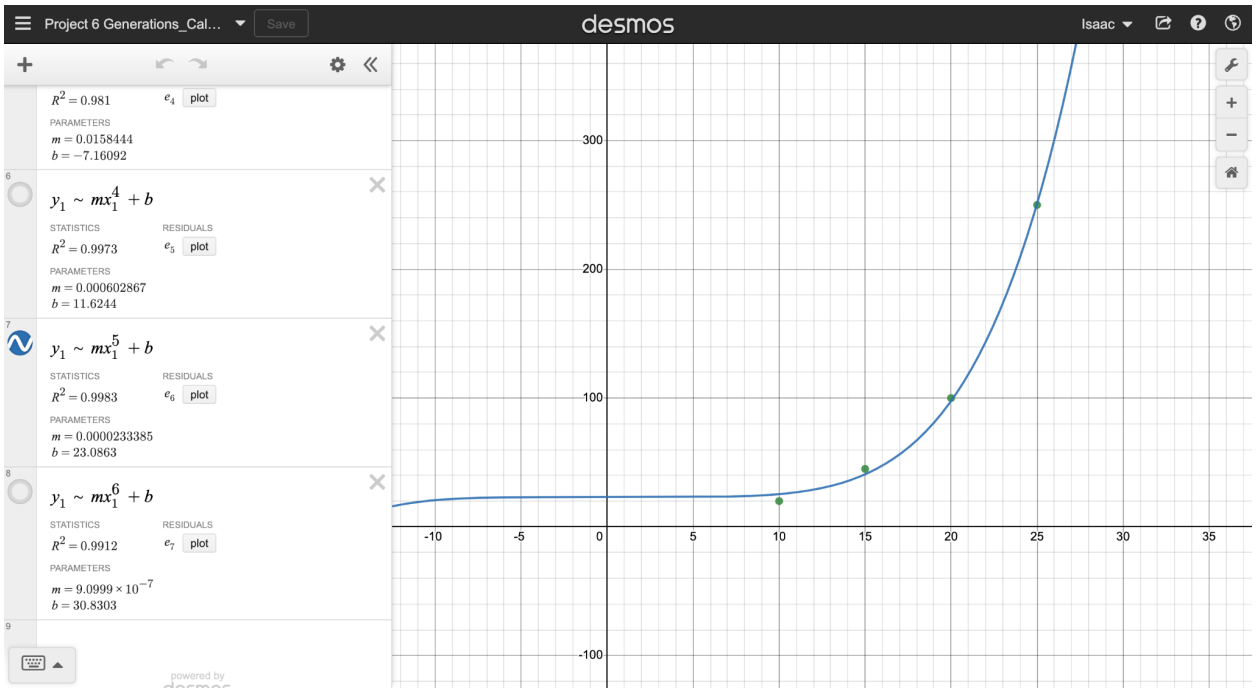


Figure 1.3

The genetic algorithm

For our implemented GA, you'll notice that there are 3 different versions in the table below: with random initialization, without random initialization unbounded, and without random initialization bounded. These three versions come from our experimenting with different ways of implementing the algorithm. Random initialization is using the included random function to generate n^2 initial states equal to the square of the problem size (n). Unbounded vs. bounded is whether we use our generation ratio to calculate the number of children based on the size of the problem (unbounded); or if we limit the generations run and children created to arbitrary numbers between 10-10,000 (bounded). Note here that we don't deal with bounded or unbounded with random initialization, because the random initialization is already too big to compute once n = 60. This is because the algorithm gets stuck creating random initial states for this problem size, and can barely even generate 10 initial states within 10 minutes; thus, we only ran the random algorithm unbounded to see how it run at its best.

The best version of the algorithm in terms of returning the solution with the lowest cost is with random initialization; the obvious tradeoff is that this also runs the longest. Using this version, we beat the greedy algorithm >50% of the time for sizes 15, and 30, but time out at 40 because of how quickly the problem grows. Anything greater than this is too big to calculate in a reasonable amount of time. This is, as we mention, because it takes too long to generate initial random states, and too long to run unbounded. We estimate that it could run around n = 35 under 10 minutes.

We run into a similar problem even when we don't generate initial random states, but we run it unbounded. In this case, we generate random states using the modified greedy algorithm

described in the algorithm explanation section above. The difference between this one and initialization with random states is that this version is more likely to actually finish after around 10 minutes and return *a* solution, although it is almost always just the initial greedy solution. This is because it doesn't get stuck just generating the initial random states, and instead runs out of time because of the large number of children it has to make and generations it has to run. The n^5 growth rate is just too big when the number of edges is also n^2.

Thus, the most realistically useful version of the algorithm is the bounded version with greedy initialization of states. This version runs *fast*, even for large problem sizes, but is more likely to return less optimal solutions, and will often just return the greedy solution. The clear tradeoff, then, is time vs. optimality—as is always the case with NP-hard optimization problems.

Overall, the *best* version of the algorithm in terms of returning the best fit (or lowest cost) is the unbounded version of the algorithm *with* initialization of random states, although the unbounded version with greedy initialization should also return pretty good results, and can be run on larger problem sizes than random initialization.

The other takeaway from the tables above and below is that the greedy algorithm is actually pretty good; certainly not optimal, but *much* better than random solutions, and not trivial to beat. Using a greedy algorithm for an initial BSSF is often a good choice because it is a fairly good solution, but also because it can be generated so fast compared to other possible solutions – O(n^2) time.

Table 2

	Our Algorithm (W/ random init)			Our Algorithm (W/out random init, unbounded)			Our Algorithm (W/out random init, bounded)		
# Cities	Time (sec)	Path Length	% of Greedy	Time (sec)	Path Length	% of Greedy	Time (sec)	Path Length	% of Greedy
15	0.14	12,082	77.30%	3.15	8,869	55.59%	0.32	10,045	64.26%
30	17.9	17,930	86.87%	142	13,805	66.89%	0.4	17,062	82.67%
40	TB	TB	TB	TB	TB	?	0.67	20,313	92.24%
60	TB	TB	TB	TB	TB	?	0.81	26,345	97.95%
100	TB	TB	TB	TB	TB	?	1.81	35,712	84.16%
200	TB	TB	TB	TB	TB	?	12.44	54,815	93.01%
500	TB	TB	TB	TB	TB	?	23.93	102,726	98.79%
750	TB	TB	TB	TB	TB	?	49.94	130,782	99.10%
1000	TB	TB	TB	TB	TB	?	94.73	155,444	97.14%

Future work

One of the first ways that we could continue to improve on our algorithm is by creating better initial states. At first, we just tried generating random initial states, and this worked well for smaller problem sizes. For smaller problems, random initial states were more likely to approach a good or optimal solution, and they weren't costly to create. However, when the problem size reached ncities ≈ 60, using the random tour generator to create initial states became unfeasible. At this point, we switched to generating as many greedy states as possible (using each city as a different starting point – described in greater detail above) for our initial states. This was effective in that it gave us a number of fairly good solutions, but we were limited by the number of initial states we could create. The problem was that this often landed us in local optima where the greedy states were, as they were often very similar paths. Future work could

seek to generate better initial states and thereby create better solutions throughout the entire algorithm.

We calculated a generation ratio (described above) that allowed us to beat the greedy algorithm at least 50% of the time in the problem size in which we worked. The problem with this ratio is that it created an overhead that was too high to realistically handle for larger problems. Future work would continue optimizing this ratio, for both its ability to beat the greedy algorithm, and its ability to handle big problem sizes. Questions that we were unable to answer but that would be interesting to explore include: What kind of ratio would be necessary for the GA to *always* beat the greedy solution? Is that possible? Does there exist a ratio such that the GA can beat the greedy and still run in a reasonable amount of time for large problem sizes?

One of the core components of a genetic algorithm is its randomness; and one of the key ways to make GAs run better is to predict, direct, and properly handle such randomness. Our GA was mostly random, although we did include certain methods of probability for children's survival and the likelihood of performing mutations. As we played around with these probabilities, we noticed that they could have a significant impact on whether or not our algorithm beat the greedy algorithm. Future work could focus on optimizing these probabilities in order to better guide the randomness of the GA. If the randomness were better guided, this could also help to optimize our generation ratio, as it may not be necessary to create as many children in a case where *better* children are more frequently made.

In this same line of thinking, one of the reasons that we chose the GA was that we wanted to "cross-contaminate" it with other algorithms. Unfortunately, due to the nature of the project, we were unable to implement any of these mixed algorithms due to limited time. However, one of the mixes that we were most excited about was implementing ant colony optimization-like features in our GA. Without digging into ant colony optimizations too much, what we imagined was creating a probability for each edge in the graph, such that edges with lower weights were *more likely* to be chosen for crossovers and/or mutations. This may significantly increase the runtime of each individual crossover or mutation; but if it ended up giving better solutions more frequently, then it may be worth it. In other words, if it took double or triple the time to create a mutation, but better paths were created 10x as frequently, then it would be a good implementation. Future work should seek to analyze the possibility of creating such mixed algorithms, the probability of these mixes outperforming already useful algorithms, and empirically and theoretically analyzing these mixed algorithms to see if they could actually outperform current algorithm implementations.

All of the optimizations mentioned in this section have one commonality: seeking to eliminate the randomness of the GA. This is obviously a tricky game to play, as randomness is one of the pros of a GA. Said better, these optimizations seek to *guide* rather than eliminate randomness, in a way that better solutions are more frequently found, thereby decreasing the amount of time necessary for the GA to run.

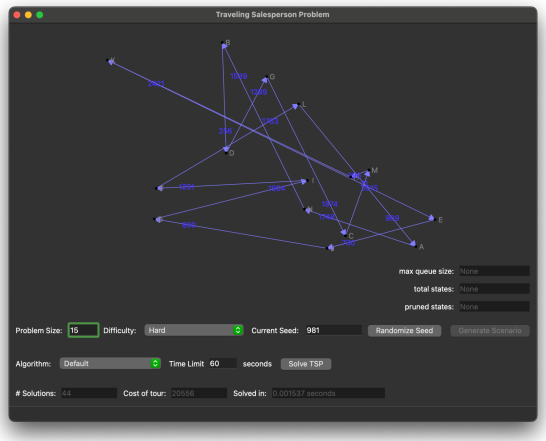
Conclusion

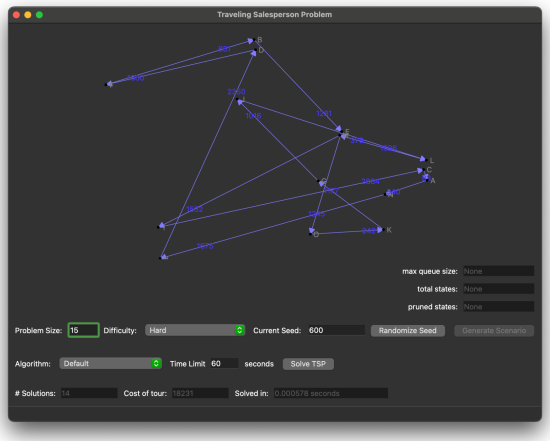
Overall, the genetic algorithm can provide solutions that are good, but it is often not the best way to approach an algorithm. For novel problems, it can be a good initial approach, but other methods of solving a problem should be sought for. For this reason, we recommend using genetic algorithms only under certain conditions, including: 1) problems for which there exists no “shortcut” or other pre-established method of solving (i.e., new problems that don’t fit old schemas), 2) problems for which high overhead (long runtimes and high memory needs) is not an issue, and 3) problems that can be solved effectively with randomness, or that don’t have obvious or implementable heuristics.

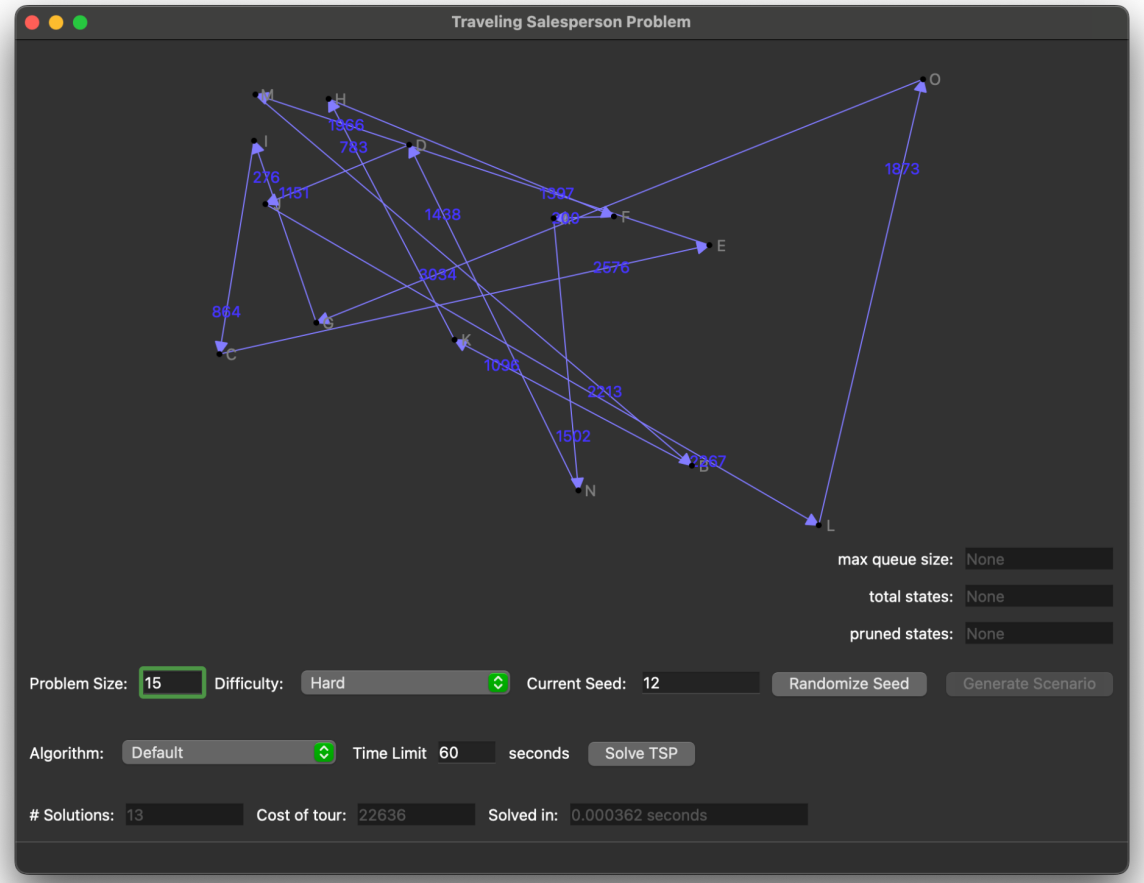
Genetic algorithms can never be, beat, or guarantee optimal solutions; they are too random for that. They can get good solutions, but even this is never a guarantee due to their highly randomized nature. This was an interesting exercise, but in practice, we would avoid genetic algorithms as much as possible. Better options exist for most problems, so while GAs are a good tool to have in the toolbox, they are one that is best saved for rainy days.

Appendix

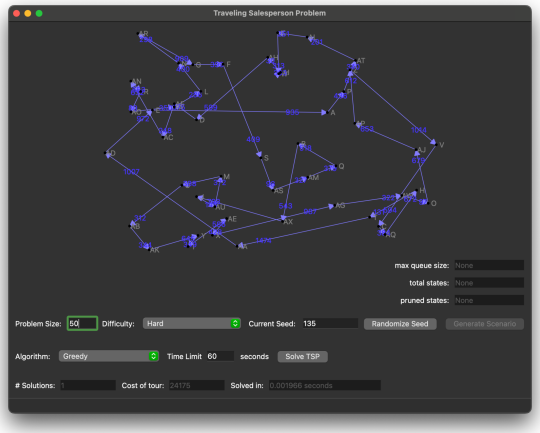
Typical default/random solutions:

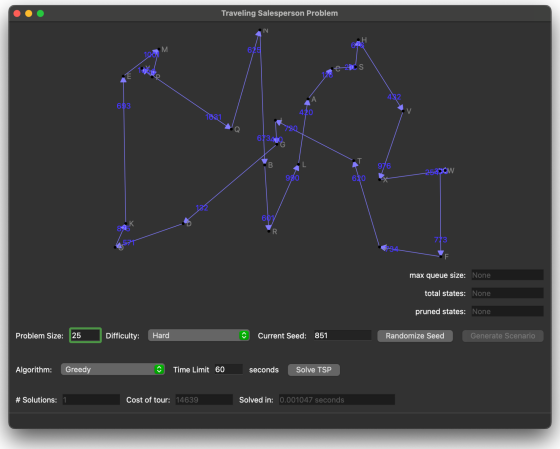


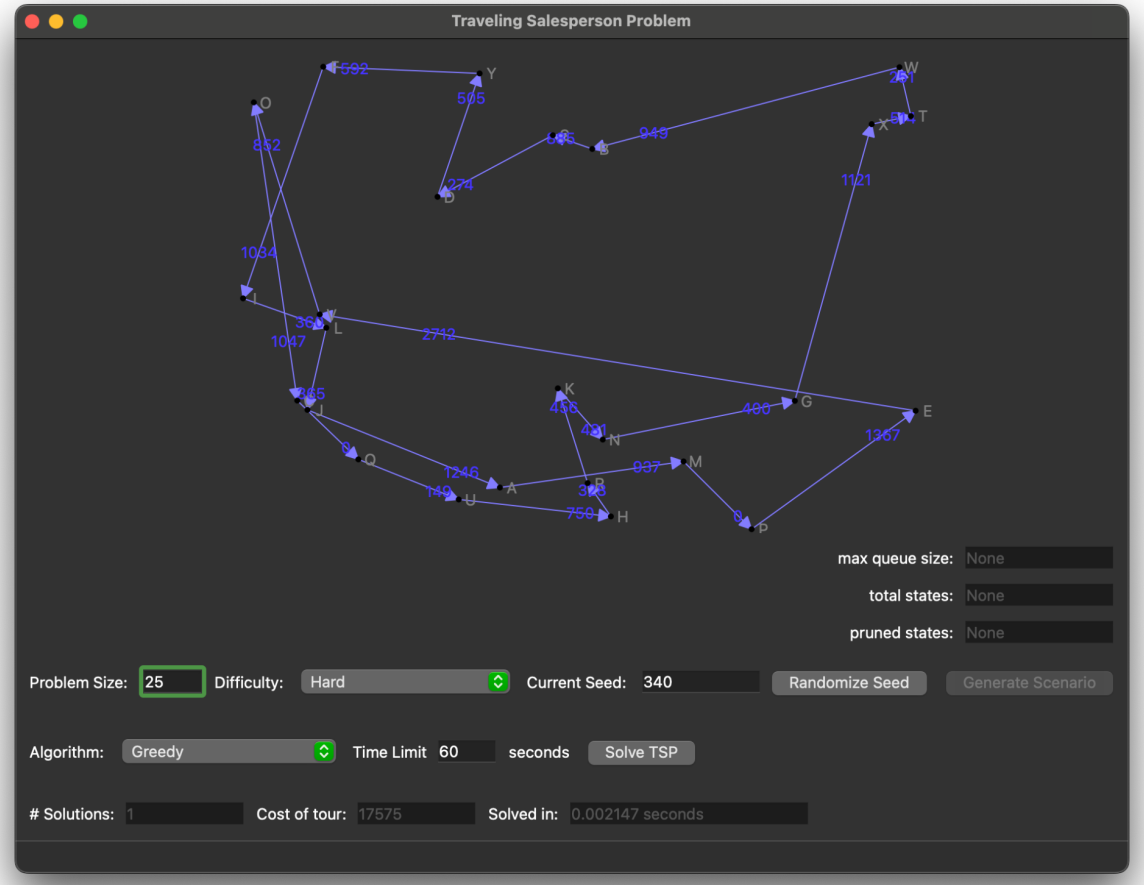




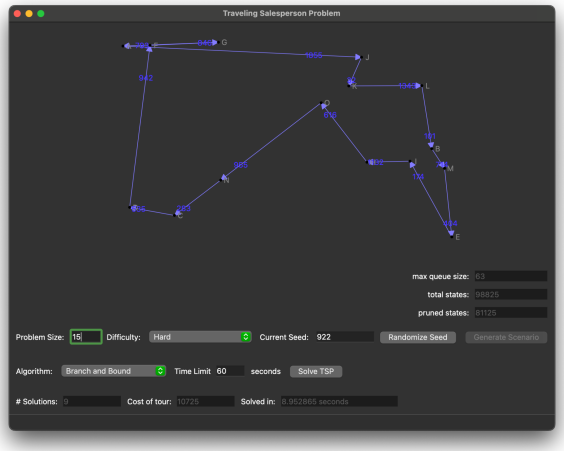
Typical Greedy Solutions

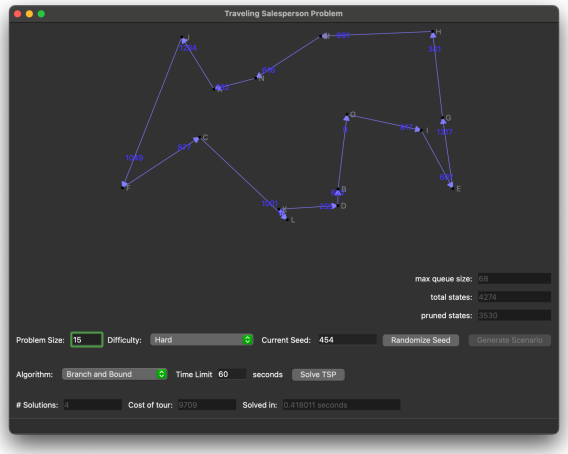


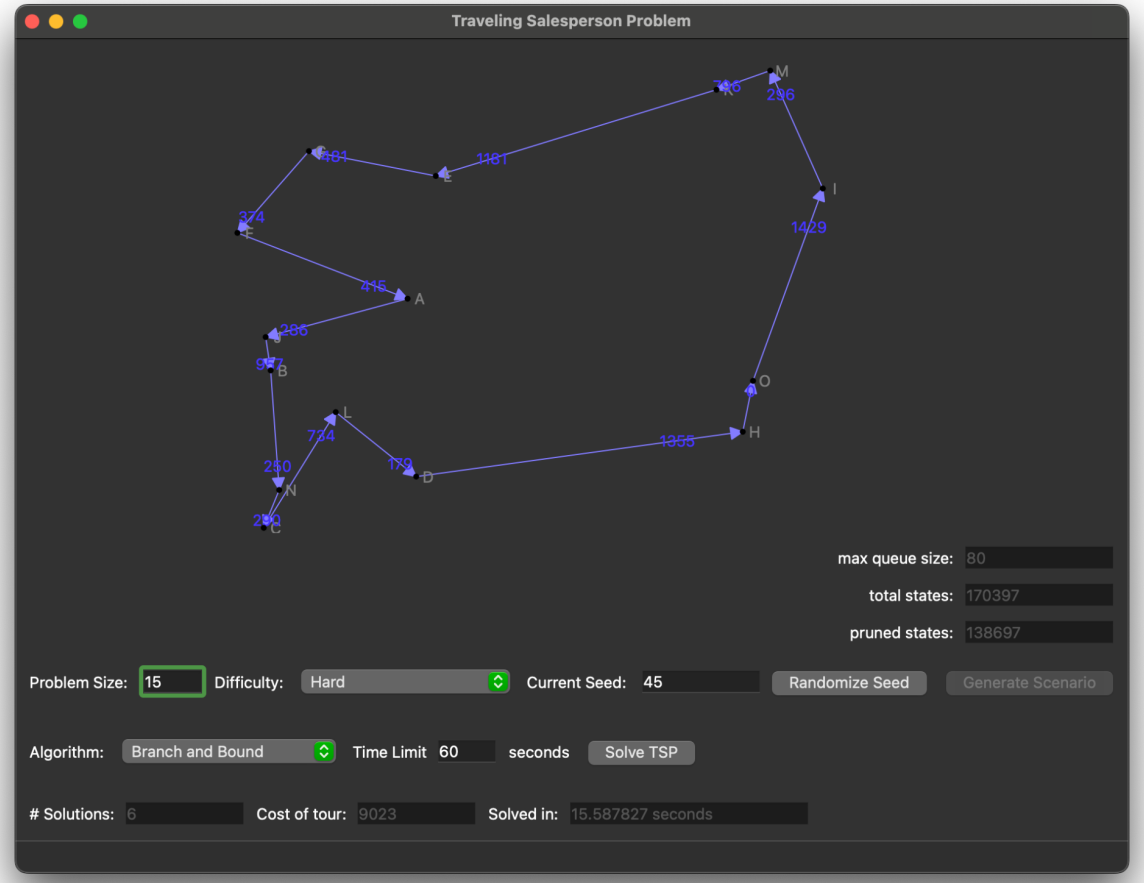




Typical B&B solutions







Typical genetic algorithm solutions