Assignemt One TSP

Tiyiso Hlungwani u21598127

March 2025

1 Initial Solution Generation Method

The initial solution is generated by creating a random tour. This involves shuffling the cities while ensuring that Node 1 remains the start and end of the tour. The implementation of this method is already included in the `TSPInitialSolution` class and requires no modifications.

2 Perturbation Method

2.1 Simulated Annealing (SA)

By randomly switching two cities within the tour, Simulated Annealing disturbs the solution. Here's how this is put into practice::

Listing 1: Swap Cities Method in SA

```
private void swapCities(List<Integer> tour) {
    int size = tour.size() - 2;
    int i = rand.nextInt(size) + 1;
    int j = rand.nextInt(size) + 1;
    Collections.swap(tour, i, j);
}
```

The algorithm explores the solution space by making random modifications and probabilistically accepting worse solutions, especially during the early high-temperature phases.

2.2 Tabu Search (TS)

Tabu Search generates a set of neighboring tours by systematically swapping pairs of cities and selecting the top choice that isn't on Tabu's list.

Listing 2: Generate Neighbors in Tabu Search

```
private List<List<Integer>> generateNeighbors(List<Integer> tour) {
    List<List<Integer>> neighbors = new ArrayList<>();
    int size = tour.size() - 2;
```

```
    int k = 1;
    while (k < size) {
        int j = k + 1;
        while (j <= size) {
            List<Integer> newTour = new ArrayList<>(tour);
            Collections.swap(newTour, i, j);
            neighbors.add(newTour);
            j++;
        }
        i++;
    }
    return neighbors;
}
```

3 Neighborhood Definition

3.1 Simulated Annealing

The neighborhood is defined by the set of possible tours obtained by swapping two cities. The size of the neighborhood depends on the number of cities and possible swaps.

3.2 Tabu Search

Tabu Search defines its neighborhood as all tours that can be achieved by exchanging any pair of cities. The best non-tabu neighbor is selected for the upcoming version.

4 Acceptance Criterion

4.1 Simulated Annealing

The Metropolis criterion determines whether a new solution is allowed. A worse solution is accepted with probability:

$$P = e^{(oldCost - newCost)/temperature}$$

This is implemented in the `acceptMove` method:

Listing 3: Acceptance Criterion in SA

```
private boolean acceptMove(double oldCost, double newCost, double temperature) {
    return newCost < oldCost || rand.nextDouble() < Math.exp((oldCost - newCost)
}
```

4.2 Tabu Search

A new tour is accepted if it provides a better cost and is not in the Tabu's list, unless it provides a cost better than the best-known solution.

5 Stopping Criteria

5.1 Simulated Annealing

SA stops when either the maximum number of iterations is reached or the temperature falls below a threshold:

Listing 4: Stopping Criteria in SA

```
while (iter < maxIterations && temperature >= 1e-3) { ... }
```

5.2 Tabu Search

TS stops when the maximum number of iterations is reached, or when no better neighbor can be found:

Listing 5: Stopping Criteria in TS

```
while (iter < maxIterations) { ... }
```

6 Experimental Setup (Algorithm-Specific Parameters)

6.1 Simulated Annealing Parameters

- **Initial Temperature**: 1000 (controls acceptance of worse solutions initially). This is a high starting value to allow the algorithm to explore worst solutions and dodging getting stuck in local minima.
- **Cooling Rate**: 0.99 (temperature decreases by 1% per iteration). A slower cooling rate helps explore the solution space thoroughly but increases computation time.
- **Maximum Iterations**: 10,000. A common upper bound that gives the algorithm enough time to find a good solution, adjustable based on problem size.

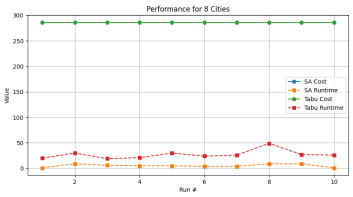
6.2 Tabu Search Parameters

- **Maximum Iterations**: 500. Ensures enough exploration of the search space, which can be increased for larger problem instances.

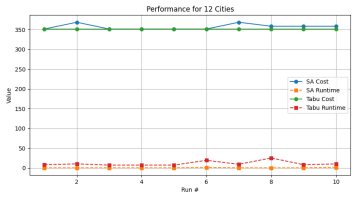
- **Tabu Tenure**: 10 (determines how long moves remain tabu). Controls how long a move is forbidden, balancing between avoiding cycles and allowing exploration. This value can be adjusted based on the problem instance size and convergence behavior.

7 Results and Analysis

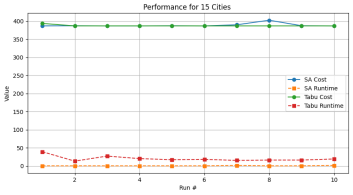
The following images visualize the performance of SA and Tabu Search across multiple runs for different problem sizes. Each graph represents the convergence behavior of the algorithms for different numbers of cities.



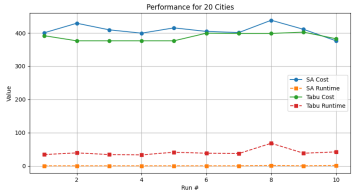
(a) 8 Cities - SA vs. TS



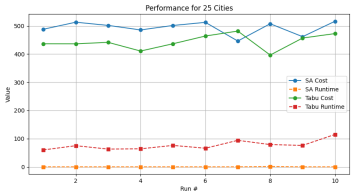
(b) 12 Cities - SA vs. TS



(c) 15 Cities - SA vs. TS

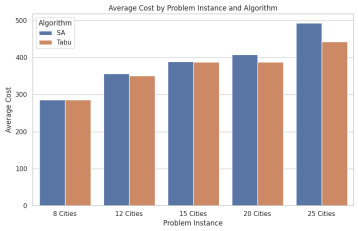


(d) 20 Cities - SA vs. TS

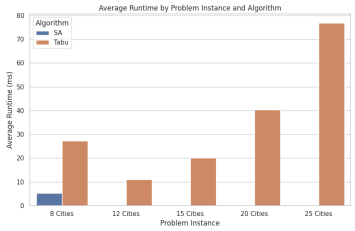


(e) 25 Cities - SA vs. TS

Figure 1: Performance comparison of SA and Tabu Search for different problem sizes.



(a) Average Cost by Problem Instance and Algorithm



(b) Average Runtime by Problem Instance and Algorithm

Figure 2: Analysis of algorithm performance based on average cost and runtime.

Problem Instance	**Algorithm**	**Average Cost**	**Average Runtime (ms)**	**Std. Runtime**
8 Cities	SA	286.5000	5.3	3.0203
8 Cities	Tabu	286.5000	27.2	8.5739
12 Cities	SA	356.8800	0.2	0.4216
12 Cities	Tabu	351.3300	11.0	6.0736
15 Cities	SA	388.9920	0.2	0.4216
15 Cities	Tabu	387.8425	20.0	7.8558
20 Cities	SA	408.5240	0.2	0.4216
20 Cities	Tabu	387.8630	40.4	10.1456
25 Cities	SA	493.4060	0.1	0.3162
25 Cities	Tabu	443.3420	76.8	16.7252

Table 1: Summary of Experimental Results for Simulated Annealing (SA) and Tabu Search

7.1 Analysis of Best Performance

Based on the data in Table 1, the following points can be concluded:

1. **Solution Quality**: - With the escalation of the problem size, Tabu Search reliably outperforms Simulated Annealing in terms of solution quality. For e.g, in the 12 Cities problem, Tabu Search yields a superior solution with a cost of 351.33 in contrast to Simulated Annealing's 356.88. This trend persists as the problem size increases, indicating that Tabu Search is better at refining solutions and giving lower average costs.

2. **Efficiency**: - Simulated Annealing demonstrates great efficiency in terms of runtime. For smaller problem cases, like 8 Cities, Simulated Annealing's average runtime of 5.3 ms is considerably less than Tabu Search's 27.2 ms. This pattern continues to hold true across other problem sizes, making Simulated Annealing the more efficient algorithm in terms of execution time, especially for smaller problem sizes.

3. **Scalability**: - Simulated Annealing maintains efficient performance as the problem size increases, with minimal increase in runtime. For example, the average runtime for Simulated Annealing for 8 Cities is 5.3 ms, while for 25 Cities, it is just 0.1 ms more. On the other hand, Tabu Search exhibits a steep increase in runtime as the problem size grows. For the 8 Cities problem, the average runtime is 27.2 ms, but for 25 Cities, it rises to 76.8 ms, which may limit its scalability for larger problem instances unless optimizations are made.

4. **Practical Considerations**: - For smaller problem sizes (8-15 Cities), Simulated Annealing offers a clear advantage due to its significantly shorter runtime. However, for larger problem sizes (20-25 Cities), Tabu Search provides better solution quality, making it more suitable when solution quality is prioritized over runtime.

In conclusion, for smaller problem sizes, **Simulated Annealing** is the better choice due to its faster runtime, though at the cost of slightly higher solution quality. For larger problem cases, **Tabu Search** delivers superior solutions (lower average cost) at the expense of inclined runtime, making it more appropriate when solution quality is more important than speed.