PSU COMPUTER SCIENCE DEPARTMENT

CS584 Final Project

Ant Colony and State Space Search Optimizers For TSP

Matthew Bui

March 12, 2020

1 PROBLEM

1.1 INTRODUCTION

The traveling salesman problem (also called the traveling salesperson problem or TSP) is a problem asked to find the shortest path traveling through all the vertexes in a graph and return to the original vertex. TSP and its variants can be applied to many areas such as manufacture, logistic, DNA sequencing, and even astronomy. "As astronomers observing many sources will want to minimize the time spent moving the telescope between the sources" [1]. In complexity, TSP also is the generalization for other problems like the travelling purchaser problem and the vehicle routing problem.

1.2 MOTIVATION

TSP is a typical NP-Complete Problem. The problem is easy to understand and its solution is straightforward. Besides, TSP has many practical and theoretical benefits. Solving TSP will give me experience working with other NP-Complete problems. Therefore, TSP is the best choice for this project.

1.3 DESCRIPTION

We can use a graph to represent for TSP problem. The paths are the edges, the cost is the weight, and the cities are the vertexes. Usually, the graph in TSP fully connected. There are two kinds of TSP called symmetric and asymmetric. In symmetric TSP, the edges are undirected weighted. In the meanwhile, in asymmetric TSP, the edges are directed weighted.

Symmetric and asymmetric TSPs are used in different cases. Symmetric TSP can be used for flight scheduling problems where the time moving back and forth from one city to one other is often identical. On the other hand, asymmetric TSP can be used for car routing where one-way streets and traffic collisions make time going back and forth differently.

2 Algorithm

There are four possible ways to tackle NP-Complete problems:

- Exact algorithm such as brute force and dynamic programming for small problem sizes.
- Find the optimizers giving a good solution in a short time.
- Find any solvable special cases for the subproblems and solve it.
- Find the provable approximate solution.

In this project, we will solve TSP using the first two approaches. The exact algorithm will be used to analyze how good the optimizers perform.

2.1 Held-Karp

"The Held–Karp algorithm, also called Bellman–Held–Karp algorithm, is a dynamic programming algorithm proposed in 1962 independently by Bellman and by Held and Karp to solve the Traveling Salesman Problem (TSP)"[2]. This is known for the earliest algorithm solving TSP. It reduces the performance of brute-force which is $O(n!)$ to $O(2^n n^2)$. Held-Karp uses tabulation dynamic programming and is very efficient for small size problems. Although $O(2^n n^2)$ is still non-polynomial, Held-Karp still plays a vital role in real-world applications.

2.1.1 Pseudocode

```
function algorithm TSP (G, n) is
    for k := 2 to n do
        C({k}, k) := d_{1,k}
    end for

    for s := 2 to n-1 do
        for all S ⊆ {2, . . . , n}, |S| = s do
            for all k ∈ S do
                C(S, k) := min_{m≠k,m∈S} [C(S\{k}, m) + d_{m,k}]
            end for
        end for
    end for

    opt := min_{k≠1} [C({2, 3, . . . , n}, k) + d_{k, 1}]
    return (opt)
end function
```

Pseudocode 1: Held-Karl[2]

2.1.2 Description

- Set distances from the initial state to its neighbors.
- For each size i of the subset from 2 to n - 1, generate all subsets which has size i.
  - For each subset size i, calculate the minimum distance from the initial state to the subset.
- We calculate the minimum distance from the small size subsets to the big size subsets.
- Since the problem has the optimal substructure when we calculate the minimum distanced of bigger size subsets, we re-use the value from the smaller size subsets calculated.

2.1.3 Complexity

The worst-case time complexity: $O(2^n n^2)$[2].
The space complexity: $O(2^n n)$[2].
As we can see that a big disadvantage of this algorithm is space complexity. Although we use a ton of space, the time complexity is still not polynomial time. We need a more efficient algorithm for time and space. This brings us to the second approach to solve TSP.

2.2 Ant Colony Optimizer

The pure approach of Ant Colony Optimizer (ACO) to find the shortest path to the goal state and go back to the original state just like the behavior of biological ants finding food. There is a similarity between ACO and TSP, so ACO can be used to solve TSP without any modification. When we use ACO for other optimization problems, we need to modify the problem to become the shortest path problem on the weighted graph.

2.2.1 Pseudocode

```
procedure ACO_MetaHeuristic is
    while not_termination do
        generateSolutions()
        daemonActions()
        pheromoneUpdate()
    repeat
end procedure
```

Pseudocode 2: ACO[3]

2.2.2 Description

In an optimization problem, the termination condition usually is set to the timeout or the local maxima or minima point where the performance stops improving as the algorithm runs. The ACO is an iterate process of many artificial ants finding the shorstest path to the goal state and come back to the original state (the action) until the algorithm is terminated. An ant is just a simple computational agent searching for a good solution based on the pheromone. In the first step, the ants randomly generate the solution. Later, ants generate solutions based on the pheromone level. After having the solution, the ants take the action going to the goal state and come back home. The pheromone is deposited at each state by ants visiting and is updated in the graph.

2.2.3 Formula

- Pheromone update formula:

$$"\tau_{xy} \leftarrow (1-\rho)\tau_{xy} + \sum_{k} \Delta\tau_{xy}^{k} \tag{2.1}$$

$$\Delta\tau_{xy}^{k} = \begin{cases} \frac{Q}{L_K}, & \text{if ant } k \text{ go to the state} \\ 0, & \text{otherwise} \end{cases} \tag{2.2}$$

  where τ_{xy} is the amount of pheromone deposited for a state transition xy, ρ is the pheromone evaporation coefficient and $\Delta\tau_{xy}^{k}$ is the amount of pheromone deposited by kth ant, typically given for a TSP problem (with moves corresponding to arcs of the graph) by $\Delta\tau_{xy}^{k}$."[3]

  We can see the relationship between the number of ants visiting at a state with state pheromone level at a state in formula 2.1. The formula shows the pheromone at a state will increase if more ants going through this state. The bigger k is, the bigger $\sum_k \Delta\tau_{xy}^{k}$ is. This demonstrates learning property of ACO when the historical data affects the ants' decision.

  The most popular approach for the optimization problems is the greedy heuristic. ACO also uses this approach. The formula 2.2 shows the greedy choice for $\Delta\tau$. L_k is the weight of the edge in the graph. Larger L_k makes $\Delta\tau$ smaller which gives less update to τ.

- Select edge formula:

$$"p_{xy}^{k} = \frac{(\tau_{xy}^{\alpha})(\eta_{xy}^{\beta})}{\sum_{z \in \text{allowed}_x} (\tau_{xz}^{\alpha})(\eta_{xz}^{\beta})} \tag{2.3}$$

  where τ_{xy} is the amount of pheromone deposited for transition from state x to $y, 0 \leq \alpha$ is a parameter to control the influence of τ_{xy}, η_{xy} is the desirability of state transition xy (a priori knowledge, typically $1/d_{xy}$, where d is the distance) and $\beta \geq 1$ is a parameter to control the influence of η_{xy}. τ_{xz} and η_{xz} represent the trail level and attractiveness

for the other possible state transitions."[3]

The formula 2.3 which is based on Bayes Theory shows the probability of an ant going to the next state from the current state. When generating the solution, the greatest p does not guarantee that the ants will always go to this state, it is just likely. This property helps us combat to local minimum problem where the optimal solution might be costly early but efficient later. To make a decision choosing the next states, we can use the "roulette wheel"[4] method base on p.

2.3 State Space Search Optimizer

State Space Search Optimizer (SSSO) is a straight forward approach solving NP-Problems but widely applicable. SSSO is capable of solving any optimization problem with a few modifications. The whole iterate process can be simplified in three steps: make a decision for the next move, move to the next state, and score the move. To use SSSO solving different problems, we need to transform the problem into state representation.

2.3.1 Representation

"In state space search, a state space is formally represented as a tuple S

$$S : \langle S, A, Action(s), Result(s,a), Cost(s,a) \rangle$$

, in which: S is the set of all possible states; A is the set of possible action, not related to a particular state but regarding all the state space; $Action(s)$ is the function that establish which action is possible to perform in a certain state; $Result(s,a)$ is the function that return the state reached performing action a in state s;$Cost(s,a)$ is the cost of performing an action a in state s. In many state spaces is a constant, but this is not true in general."[5]

Let us see one way how TSP can be transformed into state representation. Consider in TSP problem, we have a graph with vertexes, edges, and weights. A state is a single solution path, the result is the sum of all weights in the path, the action is the transformation from this solution to the neighbor solutions and since there is no cost when doing the action, the cost is 0. Our solution paths are the sequences starting with initial vertex v_0, following a sequence of vertexes from the set $\{v_1, v_2, ..., v_{n-1}\}$, and ending with v_0. The transformation can be defined by moving the second vertex in the sequence of the current state to other positions while the rest vertexes are unchanged. For example, [1, 2, 3, 4] has two neighbors which are [1, 3, 2, 4] and [1, 4, 3, 2].

2.3.2 Pseudocode

```
Data: S: initial state
opt = inf
result = []
Result: The optimal state
while not termination condition do
    read current;
    if opt < S.result then
        opt = S.result;
        result = S;
    end
    S = S.decideNextState();
end
return result
```

2.3.3 Decide Next State

In pseudocode SSSO, the deciding next state is the main strategy of SSSO. It has two main strategies called local search and complete search. Both strategies are useful. Depending on the problem, we can select an appropriate strategy.

2.3.4 Complete Search

Complete Search is an approach just like brute-force when we need to search the whole state space, so we can always find the best optimization with Complete Search. The termination condition is when the whole space is completely searched. This leads to a problem as same as NP-Complete problems. Since the space is too big, we can not finish searching in polynomial time. However, we can use complete search as verification for local search in a small space. The popular complete search techniques are traditional depth-first search, breadth-first search, iterative deepening, and lowest-cost-first search.

2.3.5 Local Search

The problem of complete search brings us to a faster optimizer called the local search. Local search strategy improves the result by using heuristics, not searching the whole space, so the local optimizer only produces good solutions but the best optimization. This problem called local minima/maxima is known as the biggest problem of local search optimizer. The termination condition in local search can be set when the algorithm reaches the local optimization or the timeout. The popular local search techniques are heuristic depth-first search, greedy best-first search, and A* search.

2.3.6 Local Search Problem

When the optimizing score stops improving in local search, it means that we are facing to the local max/min problem. To combat it, we can use some techniques called Jump, Restart, and

Beam Search[6]. In Jump technique, instead of moving one step to the next neighbors, we jump over n steps to avoid local max/min. Even after we jump and find new optimization, we are not sure if the result is optimized. When n steps are set either too high or too small, we skip the global max/min or cannot jump over local max/min. To fix this, we can apply Restart. We can restart the Local Search and compare with the old optimal result. This is the trade-off for time complexity to get a better result. Even if Jump and Restart are used, we are still able to get local max/min because the distance between the initial state and the goal state is too big. To deal with this, we can use Beam Search techniques where we search in multiple directions instead of one direction. Even if we use Beam Search, we can not avoid local max/min. If the beam is set too high, we do not have enough space or time to complete. If the beam is set too low, we easily get local max/min. No matter how hard we try to avoid local max/min, it is still a part of local search.

3 Experiment

The main purpose of this experiment is to compare the time and score performances of Held Karl, ACO, SSSO solving TSP. The implementation of Held Karl and ACO are inherited from Carl Ekerot[7] and Ruipeng Zhang[8]. I will focus on improving Local Search using Search Beam and Restart. We will see the effects of using Restart and Search to Local Search. In the experiment, we will examine the speed growth of Held-Karl and the performance comparison in the same timeout setting of ACO and SSSO. Besides, we will see the effect of problem sizes to the solvers. The experiment is implemented in Python and uses Python time library to measure time-consuming. We will solve symmetric TSP.

3.1 Procedure

3.1.1 Experiment 1

Procudure: Solve TSP in a small size which is n = 10, 15, 20 using Held-Karp.

Expectation: We will see the time growth of Held-Karp.

3.1.2 Experiment 2

Procudure: Solve TSP in a small size which is n = 10 and timeout is set to 10 seconds 100 times using: Held-Karp, Local Best First Search, Local Best First Search with Restart, Local Beam Search, and Local Beam Search With Restart.

Expectation: We will see the performance of different local search approaches as well as the effect of using Restart and Beam

3.1.3 Experiment 3

Procudure: Solve TSP in a small size which is n = 15 and timeout is set to 10 seconds for 10 times using Held-Karp, Local Beam Search with Restart, and ACO.

EXPECTATION: We want to see who is the winner ACO or SSSO by comparing the differences of the results and Held-Karl's solution

3.1.4 EXPERIMENT 4

PROCUDURE: Solve TSP in a small size which is n = 20 using and timeout is set to 30 seconds using: Held-Karl, Local Beam Search with Restart, ACO.

EXPECTATION: We want to see the effect of problem size to the solvers. Help-Karl solves n = 20 very slow but its result is very useful to verify for ACO and SSSO.

3.1.5 EXPERIMENT 5

PROCUDURE: Solve TSP in a small size which is n = 50, 100, 500 using and timeout is set to 60 seconds using: Held-Karl, Local Beam Search with Restart, ACO.

EXPECTATION: We want to test the score and time performance of ACO and SSSO in very large sizes and see if they still perform well.

4 REPORT

4.1 EXPERIMENT 1

Problem size n	10	15	20	30
Time (sec(s))	0.024	3.227	90.446	> 1000

Figure 4.1: A Result Table of Experiment 1

In small space n = 10, 15. Held-Karp solves the problem efficiently. However, after 20, the time grows very fast and it can not solve problem size n = 30 with a normal computer.

4.2 EXPERIMENT 2

The scores in the histogram 4.1 are the differences between the actual scores and the Held-Karp's score. We can see the Best First having the problem with local minima. $P[optimal - score \leq 50] = 0.1$ which is really bad. Beam Search has better performance than Best First but it still has local minima. $P[optimal - score \leq 50] \approx 0.3$ which is okay. Beam Search With Restart has better performance than Best First Search. However, the performance is not improved from the regular Beam Search. $P[optimal - score \leq 50] \approx 0.4$. The restart method does not affect Beam Search too much. It can be explained by the stableness of Beam Search. The score range of beam search is small, when we do Restart, the score just improves a little bit. Surprisingly, Best First Search With Restart works with Best First Search.

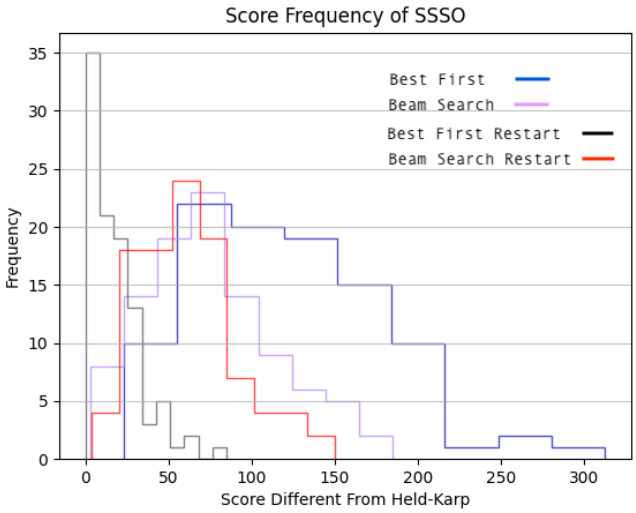

Figure 4.2: A Histogram of SSSO performances with size = 10 and trail = 100

$P[optimal - score \leq 50] \approx 0.9$ which is excellent. It helps in boosting the performances of Best First Search significantly. As we can see, Restart and Beam Search help tackle the local minima but no matter how hard we try to avoid local minima, it still exists in SSSO.

4.3 Experiment 3

Trail	1	2	3	4	5	6	7	8	9	10	Average
HK	160	170	115	168	244	220	250	139	166	227	185.9
ACO	230	192	163	252	381	259	312	141	176	272	237.8
Local Search	256	234	186	238	327	278	310	206	205	283	252.3

Figure 4.3: A Result Table of Experiment 3

In size = 15, ACO has better performance than my Local Search. As you can see, the bigger size problems give us bigger gaps between the score and the perfect score. Comparing with the gap of Local Search in experiment 2, the gap in experiment 3 is bigger. The gap growth depends on how good the optimizers are. ACO is considered a more complex optimizer than SSSO, the score gap of ACO is better than Local Search. We will examine the score gap growth of ACO and Local Search in experiments 4 and 5.

Solver	Held Karp	ACO	SSSO
Score	234	339	361

Figure 4.4: A Result Table of Experiment 4

4.4 Experiment 4

Figures 4.4 confirms again, ACO beats SSSO at solving TSP. It can be explained by learning from the history property of ACO. ACO keeps tracking the pheromone as the algorithm goes. In the meanwhile, the heuristic I have used for SSSO is greedy which is very straight forward. As a result, the score of SSSO increases slowly.

4.5 Experiment 5

Solver	ACO	Beam Search Restart
size 50	331	515
size 100	448	1177
size 500	903.0	25877

Figure 4.5: A Score Result Table of Experiment 5

Figure 4.5 shows us clearly a big difference between ACO and SSSO. ACO outperforms SSSO in the bigger sizes. ACO's performance is very stable in different problem sizes. In meanwhile, SSSO starts giving a really bad score in a bigger space because SSSO's optimal score increases super slow.

5 Conclusion

In this project, we experience solving an NP-Complete problem using heuristic optimizer and exact algorithm. The exact algorithm is very useful for small size problems. In some real applications, this is acceptable because the users sometimes do not need to solve big size problems. This motivates computer scientists to find the better not polynomial time to solve NP-Complete problems. Heuristic approaches can give us good solutions in a short time but they can not avoid the local maxima/minima problem. The more complex heuristic gives us more power in optimization. We experience this via experiments. SSSO's heuristic is straight forward and works for many problems but it has problems with bigger size problems. In the meanwhile, ACO heuristic is more complicated and we need to modify the problem to work it. However, ACO performs outstandingly in different sizes.

6 FUTURE WORKS

This project, I have experienced two approaches solving NP-Complete and there are two more left. I expect to experience two more in the future. After finishing four approaches, I expect to be familiar with solving the NP-Complete problem and be able to pick the right approach for different problems.

REFERENCES

[1] Wikipedia Contributor, *Travelling salesman problem*, Mar 8 2020, From:
https://en.wikipedia.org/wiki/Travelling_salesman_problem

[2] Wikipedia Contributor, *Held–Karp algorithm*, Mar 8 2020, From:
https://en.wikipedia.org/wiki/Held–Karp_algorithm

[3] Wikipedia Contributor, *Ant colony optimization algorithms*, Mar 8 2020, From:
https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms

[4] Ali Mirjalili, *How the Ant Colony Optimization algorithm works*, Mar 8 2020, From:
*https://youtu.be/*783*ZtAF*4*j*5*g*

[5] Wikipedia Contributor, *State space search*, Mar 8 2020, From:
https://en.wikipedia.org/wiki/State_space_search

[6] Wikipedia Contributor, *Beam search*, Mar 8 2020, From:
https://en.wikipedia.org/wiki/Beam_Search

[7] Carl Ekerot, *Held-Karp*, Mar 8 2020, From:
https://github.com/CarlEkerot/held–karp

[8] Ruipeng Zhang, *ant-colony-optimization*, Mar 8 2020, From:
https://github.com/pjmattingly/ant–colony–optimization