570 Re-implementation of H-TSP Heuristic Solver

Anonymous submission

Abstract

This project aims to provide a re-implementation of a paper proposing a novel approach to heuristically solving the Traveling Salesman Problem (TSP) called Hierarchical TSP solving. (Pan et al. 2023) The authors have developed a framework they call H-TSP for heuristically solving large traveling salesman (TSP) problems quickly. This framework uses an encoder to extract information about the network using a Convolutional Neural Network (CNN) similar to those used in image processing. Then, a Markov Decision Process, (MDP) referred to as the "upper model" in the paper, splits the problem into several parts. A Transformer Network referred to as the "lower model," then generates solutions to each subproblem. These two processes take place together in a "joint training process," so that the MDP can change its approach over time and gradually produce better subproblems for the solver. The paper's results show that the framework is faster and just as accurate as other methods when solving small problems. As the problem size grows, the H-TSP framework sometimes runs two orders of magnitude faster than its contemporaries, though sometimes sacrificing some accuracy. This re-implementation doesn't do the original justice, as the overall process wasn't fully implemented in this time frame.

Code Repository

Anonymous re-implementation GitHub Repository — https://github.com/JohnPurdue/570-H-TSP-reimplementation

Original Project GitHub Repository — https://github.com/Learning4Optimization-HUST/H-TSP

Introduction

The Traveling Salesman Problem (TSP) is an infamous NP-hard problem in Computer Science. The premise is that a merchant wants to plan his route to visit all the cities on a particular map exactly once and end up back home, where he started. The key is that he wants to do this in the shortest route possible. In situations with only a few cities, this is trivial but as the number of cities grows into dozens, hundreds, and hundreds of thousands, the number of potential routes to check explodes very quickly. This makes for a difficult problem in practice.

Solving such problems is useful in a theoretical sense for developing solving methods for similar NP-hard problems, but it also has significant real-world applications. The obvious example is delivery route management. Delivery vehicles often have a depot where they will pick up deliveries and stop at the end of the day, and the goal of routing is to get them to as many different drop-off locations as possible during the day. This may get even more complex when one considers other factors such as traffic patterns and driver breaks. A similar program could be used to route autonomous vehicles, plan public transportation routes, or plan routes inside warehouses for delivering supplies to where they're needed optimally. In addition to these more obvious examples, there are potential uses in chip manufacturing and network design where the length that a signal needs to be transmitted (whether across a circuit board or several routing nodes) may pose significant time delay and signal degradation.

The main metrics we're concerned with in this paper and implementation are the runtime and gap percent of a given model. Runtime is a measure in seconds of how long a program takes to return a valid solution to our problem. Which of course changes depending on the machine it's run on. In this project, we're only comparing runtimes from the same device then, to enforce standardization. The Gap percentage is a measure of how far off the heuristic solution is from the actual best solution.

$$Gap\% = \frac{L_{optimal} - L_{heuristic}}{L_{optimal}} * 100$$

This particular project will use two separate models to split one very large TSP into sub-problems, and then solve each one separately. Then, they are all merged to form one solution. This is accomplished using Markov Decision Processes, (MPDs) a Transformer Neural Network, and an encoder utilizing CNN architecture to gather spatial data on each particular network.

Related Work

The most efficient exact TSP solvers that currently exist use linear optimization techniques and apply integer programming to solve them. (Helsgaun 2017) However, these branch-and-cut algorithms as they are called, become computationally expensive with large problems. To solve this,

heuristic solvers also exist. These alternative models measure their effectiveness in how quickly they reach a feasible solution and how close this solution is to the optimal one. (This difference is referred to as the Gap Percentage in this paper.)

Some exact solvers are complex and impressive in how they handle even the largest of problems in TSBLIB. However, Concord, one of, if not the most runtime efficient of these solvers, (Applegate et al. 2009) was published in a paper in 2008. Since then, there have been few improvements to exact solvers, despite the TSP being a well-known and well-studied premise.

Greedy algorithms (such as Prim's and Kruskal's algorithms) can be somewhat effective heuristic algorithms, but they pay no attention to long-term route planning. This is where an MDP specifically becomes very useful. MDPs allow the specification of a discount factor, which affects how much the model prioritizes current rewards with the potential for more in the future. (Rewards here generally refer to the negative length traveled, as we'll see later.) Notably, the paper this project is based on uses a discount factor of 1.0. This means the MDP prioritizes rewards in every future step it can see equally with rewards now. However, if we trained and ran an MDP to decide which node to travel to for every step of the process our runtime would suffer significantly. This phenomenon is the primary motivation behind the structure of this project.

Problem Background

The traveling salesman problem is an NP-hard optimization problem to minimize the length of a route connecting several nodes, while only visiting each node once. Except for the depot node, which marks the starting point, but is only technically visited last. As mentioned previously, many approaches have been discussed to best solve this issue in a time-sensitive manner. These all rely on finding a suitable balance between the local best option (similar to how a greedy algorithm operates) and focusing on long-term planning to find the optimal route throughout the entire network. In addition, there are many considerations to make when optimizing runtime efficiency vs solution accuracy, each of which have their own tradeoffs.

Methodology

The framework outlined in the original paper follows the following pseudo code.

```
Input V = {v_1, v_2, ..., v_N}

Output Tau = {tau_1, tau_2, ... tau_N} # in order list of nodes visited

# Preprocess V with scalable Encoder
V = encode_and_pseudo_imagify(V)

# first node travelled from is depot
tau_1 = v_D
tau_2 = nearest node to v_D

while len(Tau) < N
    SubProb = GenerateSubProb(V, Tau)
    SubSol = SolveSubProb(SubProb)
    Tau = MergeSubSol(SubSol, Tau)
end
return Tau
```

As can be seen, the initial two nodes in tau are hard coded to provide initial data for subsequent decisions. Then, the initial data is processed by an encoder, before being split into segments by our upper model. Each of these segments is then solved by our lower model and merged into the partial solution. This partial solution is then considered by the upper model when creating the next problem subset, allowing the approach to change depending on the current location and number of nodes visited so far.

Encoder

The encoder for this project has the goal of capturing information about the overall structure of the graph network for the two models in the framework. This is done by first augmenting each data point into a vector. See, the initial data is a dictionary of nodes and their coordinates. That's essentially all the model has. In order to give more context, we add a "cluster" value corresponding to where the node falls in a grid. This re-implementation uses a 3x3 grid but this is arbitrary and could even be chosen dynamically depending on the number of nodes or their distribution. In addition, we add the relative coordinates to the center of the grid they fall into, and to the center of the node cluster for that grid. Finally, a Boolean value denotes whether that point is included in the current partial solution or not, followed by coordinates of their predecessor and successor if they have been. (These are initialized to zeros if the node hasn't been visited.) This new data is then processed by linear and CNN layers to find new information about the network.

Upper Model

After the model has access to information about the overall structure from the encoder we can begin to split the problem into its component parts. This is done by the upper model which utilized a MDP that decides whether a given node should be added to the current subproblem. The mechanism for these operations is somewhat more complex than other components, but it essentially follows this psuedocode. The added complexity with the actual code mostly comes from operations within the MDP.

```
Inputs: graph G, partial solution Tau, maxNumUnvisitedNodes

Outputs: subProblem P (v_s1, v_s2, ..., v_s_subLength), starting node v_s, target node v_t (both contained in P)

Initialize: P to 0s, Q is empty double ended Queue (Dequeue)

Push starting node to end of Q

while len(P) <= maxNumUnvisitedNodes and Q not empty
v_i = pop front of Q
```

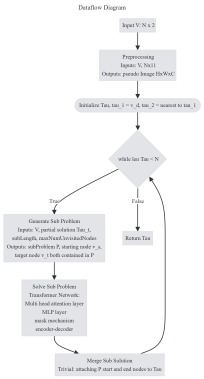

Figure 1: *Dataflow Diagram*

```
11  for v_j in neighbors in G
12      if v_j not in S_v
13          push v_j to end of Q
14          add v_j to S_v and P
15  oldLength = subLength - len(P)
16  P_start, P_end = setEndpoints(P)
17  return P, P_start, P_end
```

As can be seen from the psuedocode above, each node (up to a maxSubength parameter) requests to be added to the subproblem in order of their distance to the current node. (The last node of tau.) In the MDP, the reward given is determined by the length of the sub-route provided by the lower model according to the equation below.

$$Reward = L(\tau') - L(\tau)$$

Here τ is the previous length of the full route, and τ' is the length after the addition from the sub-problem solver. Effectively, it's just the length of the sub-problem generated, encouraging the generator to give the lower model more efficient options to solve for.

Lower Model

The lower level model is a Transformer network, trained to find the shortest route to solve an open loop TSP. An open loop TSP is similar to what we are solving with the larger program, but the trip does not return to where it started. We only start at one node and visit all others exactly once. This is necessary for the sub solution solver because we want to "stitch" these solutions together in the upper model to form one larger solution.

This Transformer Network uses a Multi-Head Attention layer and a Multi-Layer Perceptron section in order to choose the order that the given nodes should be visited in. In order to accommodate the different possible lengths of nodes we want to visit we utilize an encoder-decoder structure, similar to NLP where the translation of a sentence is not exactly the same in different languages. In this case the input size is equivalent to the outputs, but the length of that input changes as the upper model generates different sizes of subproblems.

Training

These upper and lower models are trained separately, as they are fundamentally doing different things. However, as they do work together, we can use sub-problems generated from the upper model (though they may be inefficient) to train the lower, and then these solutions are passed back into the upper to act as the reward for the previous actions. For this reason, the authors state that it is essential to warm up the lower model before this training. As if the lower model generates inefficient routes then the upper won't effectively learn how to partition route better. However, if the lower model is trained on lower quality routes in context of the larger graph context it has no impact on its long term performance since the lower model is never exposed to the larger graph context anyway.

As mentioned before the problem splitting process is controlled by a MDP. This process uses what's called Proximal Policy Optimization to minimize this objective function.

$$L(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

Here r_θ denotes the probability ratio of two policies

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}$$

$\hat{A}_t$ is the advantage function. This uses a Generalized Advantage Estimator (GAE) to choose between different policies.

$$\hat{A}_t = \sum_{l=1}^{\infty} (\gamma\lambda)^l (r_t + \gamma\hat{V}(s_t+l+1) - \hat{V}(s_t+l))$$

As previously discussed, the lower model is a Transformer Neural Network that is trained using the following policy gradient equation. The second calculation is approximately equivalent to the first and much more computationally efficient.

$$\nabla_\theta J(\theta) = E_{\pi\theta}[\nabla_\theta log\pi_\theta(\tau|s) A^{\pi\theta}(\tau)]$$

$$\approx \frac{1}{N}\sum_{i=1}^{N} (R(\tau^i) - b(s))\nabla_\theta log\pi_\theta(\tau^i|s)$$

In the above equations τ is a feasible solution to the given open loop TSP problem. $R(\tau^i)$ is the reward, which is the inverse of the length of the current subproblem, $-L(\tau^i)$ and the *shared baseline* component ($b(s)$) is defined as.

$$b(s) = \frac{1}{N}\sum_{i=1}^{N} R(\tau^i)$$

Experimental Results

The re-implementation of this paper posed significant obstacles to development. Namely, the data used from TSPLIB is inconsistently formatted, complex, and large as well as the MDP being a new process for me to learn about. Overall,

these challenges led to few significant results to show for this implementation of the proposed framework.

The original implementation however, achieved significant results. So I'll go through the briefly to showcase the potential effectiveness of the approach. The short version is that the H-TSP framework typically achieved a 7% gap accuracy across model sizes, where as other models ranged from 3% to several hundred, or ever thousand times that. In addition the time taken for each computation ranged from about 0.3 seconds to 3.3. This consistently was the shortest time across any models tested. In the smaller problems with only about 1,000 nodes, the differences are minimal, typically only a few seconds, but in the largest, with 10,000 nodes. the difference is of two orders of magnitude. (Three if you compare with the Concord exact solver mentioned previously.)

In addition to testing with their self-trained upper and lower models, they conducted some experiments with the lower model replaced by an LKH-3 solver. This is a different solver but it works in a very similar way to the one implemented in this paper. As such, it ends up getting a lower gap % in most of their tests that their pure model. However, this comes at the expense of longer run times, showcasing that there are numerous trade offs that can be made in this design process to reflect different use cases.

Conclusions and Future Research

The paper this project is based on shows significant promise that splitting a large TSP problem into smaller open loop problems can quickly solve those problems heuristically. This is especially applicable where time and computational efficiency are priorities, but a few percentage points of accuracy are not. However, by changing parameters within the lower level solver the accuracy and runtime can be effectively changed, as the original paper shows. This could allow the tailored creation of several algorithms, each for a specific purpose.

In addition, this could pose a useful framework for other routing and scheduling problems. Where one high level model splits a complex task into sub parts it may be much faster for a more traditional low level model to achieve good results.

Future research could also focus on the memory footprint of such algorithms as opposed to computational efficiency and runtime. This would allow the algorithms to be run more efficiently on smaller devices and could provide more real world applications. For example, TSP problems are often highly applicable to driver and vehicle routing, perhaps integrating similar algorithms into phones or cars cold provide local routing information that doesn't require internet connection or large computational power to run.

References

Applegate, D. L.; Bixby, R. E.; ChváTal, V.; Cook, W.; Espinoza, D. G.; Goycoolea, M.; and Helsgaun, K. 2009. Certification of an optimal TSP tour through 85,900 cities. *Oper. Res. Lett.*, 37(1): 11–15.

Helsgaun, K. 2017. An Extension of the Lin-Kernighan-Helsgaun TSP Solver for Constrained Traveling Salesman and Vehicle Routing Problems: Technical report.

Pan, X.; Jin, Y.; Ding, Y.; Feng, M.; Zhao, L.; Song, L.; and Bian, J. 2023. H-TSP: Hierarchically Solving the Large-Scale Traveling Salesman Problem. In *AAAI 2023*.