Constant-Factor Approximations of Asymmetric TSP

Zuzanna Skoczylas, Richard Xu
MIT 6.854

Fall 2020

Contents

1 Asymmetric Traveling Salesman Problem

This is a reading project that synthesizes the work of [STV20] and [TV20], two papers that provide constant-factor approximations for the asymmetric TSP problem. In addition to those papers, we consulted [Sve15] for intuitions behind the final reduction, presentations about the paper for more context as well as the full versions of each paper to understand details about the math.

1.1 Asymmetric TSP

Definition 1 (Asymmetric TSP)**.** *Let $G = (V, E)$ be a directed graph. For each edge $e \in E$, we associate with it some nonnegative cost $w(e)$. A* tour *on the graph G is a multiset F of edges such that (V, F) is connected and Eulerian.*

The Asymmetric Traveling Salesman Problem *(ATSP) asks us to find a minimum-cost tour on G.*

Compared to the metric TSP problem studied in class, the graph G here may not be complete. Our tour may need to use a given edge twice, so it needs to be a multiset of edges. The condition that (V, F) is connected ensures that the tour visits each vertex at least once.

1.2 Previous Approaches

From the definition of a tour, we see that ATSP requires an *integral, Eulerian, connected* solution. While ATSP is an NP-hard problem, we have a polynomial-time algorithm as soon as we drop one of the constraints:

1. Without the integral constraint, we get an LP relaxation that will be further explored in section 2.
2. Without the Eulerian constraint, we get the Minimum Spanning Tree problem.
3. Without the connected constraint, we get the Cycle Cover problem.

These relationships are illustrated in the graph below.

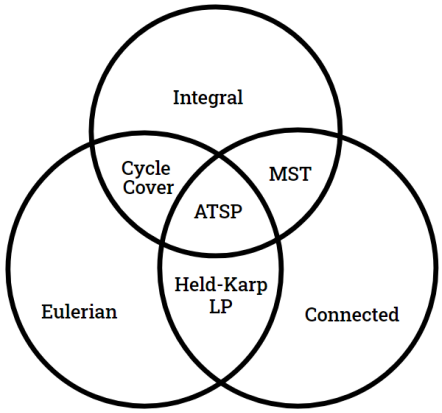


Figure 1: Different Constraints and the Resulting Graph Problem

Most approaches to finding an approximation algorithm for TSP starts with one of these three relaxations to construct a valid tour, sometimes incorporating ideas in another relaxation. For undirected TSP, the approximation algorithm we learned in class primarily used MST. The $O(\log n)$-approximation by [FGM82], which is adapted in our homework problem, used cycle cover.

In the past decade, [Asa+17] uses Held-Karp LP and thin trees, an idea inspired by MST, to achieve an $O(\frac{\log n}{\log \log n})$ approximation algorithm. The approach in [STV20] and [TV20] adapts the use of Held-Karp LP in the thin tree paper. However, they incorporate Cycle Cover instead of MST. An important paper in this line is [Sve15], achieving a constant-factor approximation for a specific type of TSP problem called *node-weighted TSP*. Then, [STV20] achieves a 506-approximation and [TV20] a $(22 + \epsilon)$-approximation.

1.3 Plan of Attack

The proof consists of three reductions, as highlighted below. In each section, we will first define the problem instance as well as how we measure the performance of an approximation algorithm for this problem. Each of these reductions adds extra structures to the traveling salesman problem we are working with, which eventually helps us find a constant factor approximation.

We will then demonstrate how to turn an approximation algorithm of the reduced problem to an approximation algorithm of the original problem. Finally, we will construct an approximation algorithm for subtour cover that, combined with the proofs above, give us a $(22+\epsilon)$-approximation.

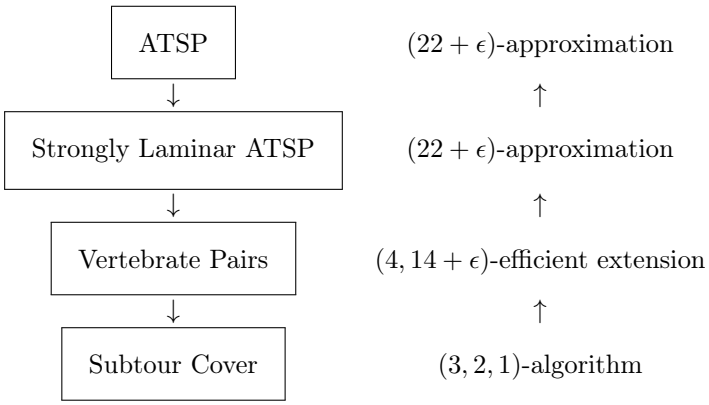


Figure 2: Plan of Attack

2 Strongly-Laminar Instances

2.1 Held-Karp Relaxation

One way to phrase the TSP problem is the Integer Program

$$\begin{aligned} &\min w^T x \ s.t. \\ &\sum_{e=(u,v)} x_e = \sum_{e=(v,u)} x_e (\text{flow conservation}) \\ &\sum_{e \text{ crosses cut } S} x_e \geq 2 \forall S (\text{subtour elimination}) \\ &x_e \in \mathbb{Z}^+. \end{aligned}$$

In the case that $x_e \in \mathbb{Z}^+$, the first constraint ensures that the resulting tour is Eulerian. Why is the second constraint called "subtour elimination"? Suppose that we remove the second constraint for some cut S. Then, we can have two subtours, one for S and one for $\bar{S}$, that satisfy all the constraints. Thus, we add the second constraint to eliminate this possibility.

Dropping the constraint $x_e \in \mathbb{Z}^+$ gives us a linear program known as the *Held-Karp Relaxation*.

2.2 Dual LP: Tolls

When we have an LP, it is natural to consider the dual. Define the variables y_S for each set S and α_v for each vertex $v \in V$. Using the cookbook, the dual is

$$\max 2 \sum_S y_S \ s.t. \sum_{S \text{ crossed by } (u,v)} y_S + \alpha_u - \alpha_v \leq w(u,v).$$

How do we make sense of this? First, define the *reduced cost* of an edge (u,v) to be $w'(u,v) = w(u,v) + \alpha_v - \alpha_u$, then the constraint states that $w'(u,v) \geq \sum_{S \text{ crossed by } (u,v)} y_S$.

Imagine that you are a salesman driving across the world. However, there are different countries in the world, and they require you pay a toll! In particular, each time you enter or leave country S, you have to pay a toll equal to y_S. Notice that tour of the world must enter and leave each country at least once, so the total toll you pay is $2\sum_S y_S$. The constraint that $w'(u,v) \geq \sum_{S \text{ crossed by } (u,v)} y_S$ ensures that the toll cost of a given edge is always less than the original cost of the edge. Some countries may be nested, which we can think of as states and counties.

This construction is beneficial because it gives us another method to measure the cost of a tour. First, notice that a path that does not cross a boundary S does not need to pay a toll. Then, traveling along that path would be free. Next, let $LP_y = 2\sum_S y_S$. Then,

Theorem 2. *Suppose A constructs a tour whose total cost is $\leq \alpha \cdot LP_y$. Then, A is an α-approximation algorithm for ATSP.*

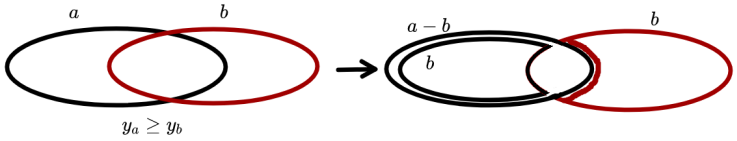
Proof. Let w_{OPT} be the cost of the optimal tour, then the optimum for the Held-Karp LP is at most w_{OPT}. By Strong Duality, $LP_y \leq w_{OPT}$. □

We now impose some extra restriction on y, which gives us more structure to work with in future reductions.

2.3 Laminar Instance: No Border Disputes

Let $\mathcal{L}$ be all the sets U where $y_U > 0$. An instance y is *laminar* if for every $U, V \in \mathcal{L}$, either U, V are nested or disjoint. If we go back to our toll analogy, a violation of the laminar property is a border dispute.

An important subroutine we invoke in future sections is *contracting* a set $U \in \mathcal{L}$ to a single vertex. When there are border disputes, we cannot perform a contraction. To resolve border disputes, we pick out sets that have disputes disputes and untangle them using the construction shown below.



Notice that the toll for crossing the original a border is $(a-b)+b = a$, and the toll for crossing the original b border is b. Therefore, the tolls are still the same and we have resolved the border conflict.

Because we need to create an extra set, we may be concerned that this causes extra border disputes. However, there exists an order to perform the untangle algorithm that would resolve this issue while creating only polynomially many sets U with $y_U > 0$. A full detail of this procedure can be found in [CFN85].

2.4 Strongly Laminar Instance: No Alaska

Let $G[U]$ be the subgraph of G containing only the vertices of U. An instance y is *strongly laminar* if for every $U \in \mathcal{L}$, $G[U]$ is strongly connected. This means that each country is contiguous: you can drive between any two towns without leaving the country itself.

Suppose some country U is not contiguous, then the salesman needs to enter and exit the country multiple times just to reach all the towns in the country. This would cause extra casework in future reductions, so we add the constraint to remove this casework. To summarize, we make the following definition:

Definition 3. *A* Strongly Laminar ATSP instance *to be a tuple $\mathcal{I} = (G, \mathcal{L}, x, y)$, where y is strongly laminar and optimal. A set $U \in \mathcal{L}$ is called a* laminar set*. Finally, let $LP_I = LP_y$.*

We now show that such an instance can be constructed.

Theorem 4. *Given an ATSP instance (G, w), we can construct a strongly laminar instance $(G, \mathcal{L}, x, y)$ in polynomial time.*

Proof. We first solve the Held-Karp LP and the dual LP. While there are exponentially many constraints in the primal LP, we can solve it in polynomial time with the ellipsoid algorithm and a separation oracle. Let the solution by x, y. The resulting solution y only has polynomially many nonzero terms as well.

Then, we resolve border disputes following the process in section 2.3.

If some set $U \in \mathcal{L}$ is not strongly connected, let S be one of its connected components. We transfer the toll cost from U to S. Notice that we did not change the objective value, so y is still optimal. Furthermore, every toll price is now cheaper because there is no edge going between S and $U \setminus S$, so the constraints are all still satisfied. □

3 Vertebrate Pairs

We now introduce the notion of a *vertebrate pair*, whose name derives a construction called a "backbone". First, we motivate the definition of vertebrate pairs by looking at previous works in the field.

3.1 Node-Weighted ATSP

In [Sve15], Svensson found a constant approximation algorithm for a special case of TSP known as node-weighted ATSP.

Definition 5. *In* node-weighted ATSP *we have a graph G and costs $h(v)$ for each $v \in V$. The cost of an edge is defined as $w(u, v) = h(u) + h(v)$.*

In other words, if we convert the instance into laminar ATSP $(G, \mathcal{L}, x, y)$, then $\mathcal{L}$ consists solely of singleton sets $\{v\}$.

We hope to generalize this algorithm to the case of general TSP. Naively, we can try to contract every laminar set to a single point so that we can apply Svensson's result.

Algorithm 1 Given $(G, \mathcal{L}, x, y)$, construct a tour on G.

Contract every maximal laminar set in $\mathcal{L}$.
Let $(G', \mathcal{L}', x', y')$ = resulting instance, $\mathcal{L}'$ consists of only singletons.
Run Svensson's Algorithm on $(G', \mathcal{L}', x', y')$ to construct a tour T.
for all maximal set $U \in \mathcal{L}$ we contracted **do**
  Uncontract U, recurse on $G[U]$ and attach the resulting tour to T.
end for

However, suppose we have a graph whose laminar sets are all nested. For example, suppose $\mathcal{L} = \{\{1\}, \{1, 2\}, \ldots, \{1, 2, \ldots, n\}\}$. Then, the recursion we perform can be n levels deep. Even if we get a constant factor approximation at each level, the result is an $O(n)$-approximation, which is terrible!

We want to be more careful with the count so that we can bound each level of the recursion effectively. However, this gives us the intuition the singleton laminar sets are easy to deal with. They contribute only a constant to the final approximation factor. So, our goal is to pay for the non-singleton laminar sets.

3.2 Backbone

We now define the notion of a backbone.

Definition 6. *A tour B is a* backbone *if for every non-singleton set $L \in \mathcal{L}_{\geq 2}$, $V(B) \cap L \neq \emptyset$.*

This definition hopes to utilize our intuition that singleton laminar sets are easy to work with. Since the backbone crosses every non-singleton set $L \in \mathcal{L}_{\geq 2}$, the only remaining sets to visit are singleton sets. We suspect that it is easy to extend a backbone to a tour using the ideas from [Sve15]. Then, if there is a way to efficiently construct a backbone, we are done.

3.3 Recursive Algorithm

Before we introduce the recursive algorithm, we first explore some of the tools used in the recursive algorithm. First, let us first define efficiency of an algorithm in the context of backbone extension.

Definition 7. *Let $(G, \mathcal{L}, x, y)$ be a strongly laminar instance and let B be a backbone. A* backbone extension *F is (κ, η)-efficient if $B + F$ is a tour and*

$$c(F) \leq \kappa \cdot LP_y + \eta \cdot \sum_{v \notin B, \{v\} \in \mathcal{L}} 2y_{\{v\}}.$$

The two terms represent two different sources of cost. The extension F needs to visit every singleton set, which is contained in the second term. Furthermore, the extension may need to cross additional sets, and those costs are contained in the first term. The perfect extension would be $(0, 1)$-efficient, and we will find a $(4, 14 + \epsilon)$-efficient algorithm in section 4. Next, we define nice paths and the length of a path.

Definition 8. *A path $P_{u,v}$ is* nice *if it enters and exits each laminar set at most once. The* length *of a path is equal to the total amount of toll the path pays.*

We first show that a nice path exists for any two vertices in a laminar set.

Theorem 9. *Let $(G, \mathcal{L}, x, y)$ be a strongly laminar instance, $U \in \mathcal{L}$ a laminar set, and $u, v \in U$. Then, there exists a nice path between u and v.*

Proof. Since the instance is strongly laminar, $G[U]$ is strongly connected. Then, there exists some path $P_{u,v}$ between them.

Suppose $P_{u,v}$ enters and exits a laminar set V more than once. Let s be the location of the first entrance and t the location of the last exit. Then, there exists a path $P_{s,t}$ containing only vertices in V. Replace the original $s - t$ path with $P_{s,t}$, then the new path only enters and exits V once. □

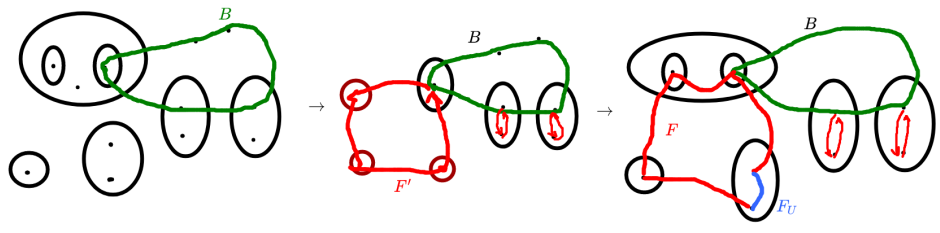
For each pair of vertices (u, v), we fix some path $P_{u,v}$ between them. We are now ready to introduce the algorithm for strongly laminar ATSP. The algorithm, given an instance $(G, \mathcal{L}, x, y)$ and a set W, computes the tour on $G[W]$. Our desired tour would set $W = V$.

Algorithm 2 Recursive ATSP Algorithm

Require: $(G, \mathcal{L}, x, y)$, $W \in \mathcal{L} \cup \{V\}$, and an (κ, η)-efficient extension algorithm A.

1. Choose $u, v \in W$ such that $P_{u,v}$ is the longest path. Let B be the cycle $P_{u,v} + P_{v,u}$.
2. Let $\mathcal{L}_{\bar{B}}$ be all the sets B does not cross. Contract each set in $\mathcal{L}_{\bar{B}}$ to a single vertex.
3. Use algorithm A to create a tour F' of the new graph.
4. Uncontract each set in $\mathcal{L}_{\bar{B}}$ to get an incomplete tour F.
5. For each set $U \in \mathcal{L}_{\bar{B}}$, recursively call this algorithm to construct a tour in U.
6. Add all the tours to F and remove any extraneous edges. Return F.

Below is an illustration of this algorithm in action.



The left graph represents step 1. Graph 2 represents steps 2 and 3: the crimson-colored sets are the contracted laminar sets, and the red cycles are the result of the extension. On step 3, notice that B is a backbone in

the contracted graph so we can run the extension algorithm. Graph 3 represents the final 3 steps. Notice that only the bottom contracted set needs a recursive call, indicated by the blue-colored line segment.

This algorithm shares a similar flavor to our first algorithm, except that we use a backbone. In the first step of the algorithm, we chose the *longest* nice path in the current graph. Why do we choose the longest path, when we are trying to minimize cost? First, notice that the length of a nice path is at most LP_y by definition. Then, a long path means that it crosses many laminar sets, and it is almost a backbone already. There are two possible scenarios:

1. The longest nice path in the graph is short. This tells us that the distance between any two nodes in the graph is small, helping us bound the remaining costs.

2. The longest nice path in the graph is long. Then, this path crosses many laminar sets and the recursive step is going to be cheap.

3.4 Efficiency of Recursion

Next, we prove the efficiency of the recursive algorithm above. We included more detail for this proof compared to the rest since it is, in our opinion, the most subtle proof in [TV20]. It also highlights how the backbone setup from the past few pages pays off.

Theorem 10. *For a set W, let D_W be the length of the longest nice path $P_{u,v}$. Given a (κ, η)-efficient extension algorithm and a strongly laminar instance $\mathcal{I}$, the resulting tour F has cost at most*

$$c(F) \leq (2\kappa + 2) \cdot LP_I + (\kappa + \eta) \cdot (LP_I - D_W).$$

Given this theorem, we notice that $c(F) \leq (3\kappa + \eta + 2)LP_I - (\kappa + \eta)D_W \leq (3\kappa + \eta + 2)OPT$. Therefore, our recursion algorithm is a $(3\kappa + \eta + 2)$-approximation.

Proof. The proof is based on the intuition described in the previous section: if $P_{u,v}$ is long then we have visited many laminar sets. If $P_{u,v}$ is short there is always a cheap walk between any two vertices in W. When a backbone extension is costly, we are able to save costs from the fact that there are fewer sets in $\mathcal{L}_{\bar{B}}$.

Argue by induction from the smallest laminar set L to the largest. The base case is a singular laminar set, where both sides of the inequality are 0 (using an empty tour). We now look at the inductive step for some set L. The total cost of a tour on L is the sum of 1. the cost of the backbone extension, and 2. the cost of the recursive step. Proceed by bounding the cost of each term.

Since the algorithm is (κ, η) efficient, $c(F') \leq \kappa \cdot LP_{I'} + \eta \cdot \sum_{L \in \mathcal{L}_B} 2y_{\{v_L\}}$. Next, we look at each term in this formula:

1. We can count the cost of each set in I' to see that

$$LP_{I'} \leq D_W + 2 \sum_{L \in \mathcal{L}, L \cap V(B) \neq \emptyset} y_L + 2 \sum_{L \in \mathcal{L}_B} y_{\{v_L\}}.$$

2. To enter and leave the node v_L, we pay the tolls for L. We also may enter and leave at different vertices, paying at most D_L extra. Therefore, $2y_{\{v_L\}} = 2y_L + D_L$.

Plugging in and doing some grueling math gives us

$$c(F') \leq \kappa \cdot (D_W + \sum_{L \in \mathcal{L}, L \cap V(B) \neq \emptyset} 2y_L) + (\kappa + \eta) \cdot \sum_{L \in L_B} (2y_L + D_L). \qquad (1)$$

Next, we bound the cost of the recursion step. Let the *value* of a set $L \in \mathcal{L}$ be the cost of entering and leaving *every* Laminar subset of L. We find three properties of value:

1. $LP_I \leq \text{value}(L)$. Combined with the inductive hypothesis, we find that

$$c(F_L) \leq (2\kappa + 2)\text{value(L)} + (\kappa + \eta)(\text{value(L)} - D_L). \qquad (2)$$

2. The backbone is a nice path, so

$$\text{value}(W) \geq \sum_{L \cap V(B) \neq \emptyset} 2y_L + \sum_{L \in L_B} \text{value}(L) \qquad (3)$$

3. The backbone has cost D_W. Since all the sets in $L_{\bar{B}}$ does not contain the backbone,

$$\text{value}(W) \geq \sum_{L \in L_B} \text{value}(L) + D_W.$$

To make sense of these inequalities, notice that when D_W is high we know that value(L) is small. This agrees with our intuition: whenever the backbone cost D_W is high, we can save cost because the remaining sets L have to be simpler.

Combine all the inequalities above to get

$$\begin{aligned} c(F') + \sum_{L \in \mathcal{L}} c(F_L) &\leq \kappa(D_W + \sum_{L \in \mathcal{L}, L \cap V(B) \neq \emptyset} 2y_L) + \sum_{L \in L_B} ((2\kappa+2) \cdot \text{value(L)} + (\kappa+\eta) \cdot (2y_L + \text{value(L)})) \\ &\leq \kappa \cdot D_W + \kappa \cdot \text{value(W)} + (\kappa+2) \cdot (\text{value(W)} - D_W) + (\kappa+\eta) \cdot (\text{value(W)} - D_W) \\ &= (2\kappa+2) \cdot \text{value(W)} - 2 \cdot D_W + (\kappa+\eta) \cdot (\text{value(W)} - D_W) \end{aligned}$$

The first inequality comes from expanding (1) and (2) and rearranging. The second inequality comes from (3) and rearranging. Adding the cost of B, which is at most $2 \cdot D_W$, the result follows. □

We removed most of the grueling math from this proof, but some were unavoidable. Next, we highlight how the final inequalities in the proof relate to our intuition from the start.

Looking at the first line of the inequality, we see that it is *cheaper* to have a set crossed by the backbone compared to when it is solved in the recursion (since $\kappa \leq 2\kappa+2$). While a long backbone means D_W is high, we get to cross laminar sets cheaply and save costs. These factors allow us to contain the cost of the backbone extension within its own recursion level, which ultimately gets us the constant factor approximation.

4 Subtour Cover

An important subroutine for the backbone extension algorithm is *subtour cover*, originally used in [Sve15]. The subtour cover problem is similar in spirit to the cycle cover approach in [FGM82]: given a set of cycles H, we find some cycles F that connect different cycles together and repeat this process. However, there are two major differences:

1. The algorithm should be "backbone aware", not crossing many non-singleton laminar sets since the backbone already paid for it.

2. Not only do we need a global bound for the subtour cover cost, we need a *local* bound as well to ensure that the cost does not spread between recursion levels.

We introduce subtour cover algorithm and its construction before the backbone extension algorithm since it is otherwise extremely difficult to understand the backbone extension.

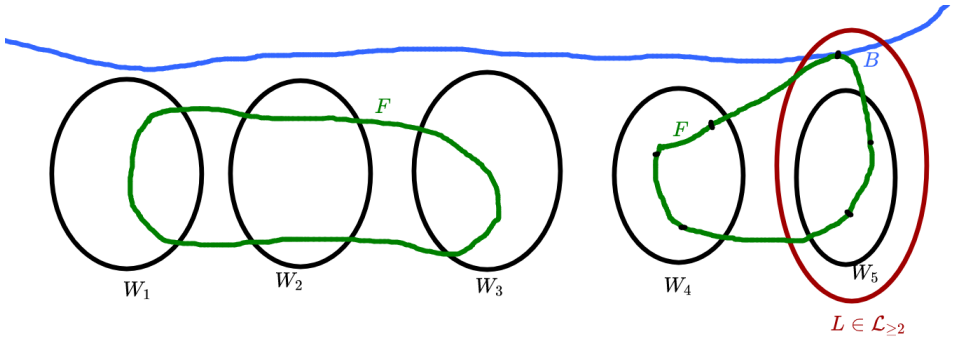
Definition 11 (Subtour Cover)**.** *Let $\mathcal{I} = (G, \mathcal{L}, x, y; B)$ be a vertebrate pair instance, and let $G' = G[V \setminus V(B)]$ be the graph of vertices not visited by the backbone. Let H be a set of subtours (cycles) in G' such that H never crosses a nonsingleton laminar set.*

The problem of subtour cover *aims to find a set of subtours F such that*

1. *F enters and leaves every connected component of H.*

2. *If a subtour $D \in F$ crosses a nonsingleton laminar set, then D and B share a vertex.*

An example instance and solution of subtour cover is given in the graph below. The blue line represents the backbone, and each set W_i is a connected component of H. The crimson circle on the right represents a nonsingleton laminar set. We see that F is a set of cycles connecting the different components of H together. Furthermore, when F has to cross a laminar set as in the right, it makes sure to also visit a vertex on the backbone B.



Why do we require the second condition? Notice that B already pays for the cost of crossing every non-singleton laminar set. If a cycle $D \in F$ crosses the set as well, we want to reroute B so that we only pay the toll once. This is easiest when D and B share a vertex. Next, we discuss how to measure the efficiency of a subtour cover algorithm.

Definition 12. *Let (I, B, H) be an instance of subtour cover. An (α, κ, β)-algorithm finds a subtour cover F such that*

$$c(F) \leq \kappa \cdot LP_I + \beta \cdot \sum_{v \in V \setminus V(B)} 2y_v,$$

and if for each connected component D of F that does not intersect the backbone*,*

$$c(D) \leq \alpha \cdot \sum_{v \in V(D)} 2y_v.$$

The first inequality is a global bound on the cost of our subtour cover F. The second inequality is a *local* bound on the cost of this cover. It ensures that *each cycle pays for itself locally.*

4.1 Strategy for Subtour Cover

Recall from class that the minimum cycle cover can be solved using min-cost flow. We will adopt those techniques to solve subtour cover. Next, notice that we almost have a solution already. The solution x to the Held-Karp LP has 1 unit of flow entering and leaving every component H. Furthermore, the cost of x is LP_I, which is very cheap. What makes x not a valid subtour cover?

1. x is a fractional solution. We may try to fix this using the fact that all the capacities are integral. However, this naive approach often causes not enough flow to enter/leave each component of H.
2. x is missing the laminar set condition.

We describe how to fix each issue by modifying the instance of min-cost flow we are working with. These fixes eventually result in a $(3, 2, 1)$-algorithm.

4.2 Fixing Fractional Solution: Rerouting

To fix fractional flow, we will add auxiliary vertices for each component and reroute flow to ensure that each component is integral and satisfies the first condition.

For each component W_i, add an extra vertex a_i. For every incoming edge to (u, w_i) where $w_i \in W_i$, create an edge (u, a_i) with the same cost. For every outgoing edge (w_i, v), create an edge (a_i, v) with the

same cost. Finally, add the constraint that the inflow and outflow from a_i is at least 1. As we have learned in the max-flow unit, this can be enforced by adding supply/demand onto the node.

We have a solution to this flow problem with the cost at most $2LP_I$. Start with the solution x. For each edge (u, v) where $u \in W_i$ and $v \in W_j$, create edges (u, a_j) and (a_i, v) instead. Notice that each vertex a_i receives at least 1 unit of in-flow and out-flow.

The reason adding a node is helpful is that finding an integral solution is easier using this approach. Recall that we make a fractional solution integral by finding an unsaturated cycle of minimum cost and add to that cycle until it is saturated. Since every cycle goes through the extra vertices a_i, each a_i will still end with at least 1 flow.

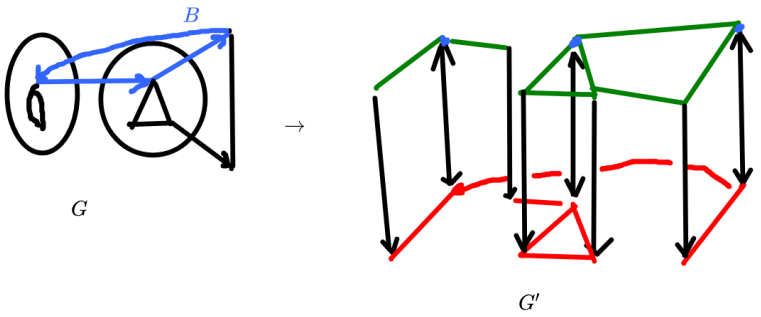
After finding an integral solution, we can map the one unit of flow through a_i to flow through one of the vertices in W_i. Hence, we can resolve the first issue while doubling the cost.

4.3 Laminar Set Condition: Split Graph

We now need to resolve the laminar property, which is done using a technique known as a *split graph.*

Definition 13. *The split graph G' contains two copies of the original graph G, labeled as an* upper *and* lower *copy. For each vertex v, we have a downward edge (v_u, v_l). However, the only upward edges (v_l, v_u) occur on vertices of the backbone.*

All edges that enter a non-singular laminar set L are on the bottom layer, while all edges that leave the set is on the top layer. For edges that leave and enter at the same time, we create an extra vertex to facilitate the transition.



The graph above illustrates the transition to a split graph. Notice that edges entering a laminar set is removed from the top, and edges leaving a laminar set is removed from the bottom. We also added an auxiliary vertex to for the edge that crosses two laminar sets at once.

Notice that we can project a flow on G' onto a flow on G by merging the upper and lower flow together. Suppose that the projected flow on G crosses a laminar set L. Then, the original flow on G' must have entered L through a lower edge and left L on an upper edge. Since the only upward edges occur on the laminar vertices, we ensure that every cycle crossing L must also visit a laminar vertex.

4.4 Local Efficiency: Witness Flow

Careful analysis of the two modifications made above, we will find that we create an integral subtour cover with cost $c(F) \leq 2LP(I)$. However, we no longer have the local bound on the subtour cover, which states that $c(D) \leq \alpha \sum_{v \in V(D)} 2y_v$ for some α. This is because changing a flow to be integral can change the set of connected components of the graph. The fix turns out to be intimately related to the split graph constructed in the previous section.

Consider the flow on the bottom level of the split graph, which all end on the backbone. Recall that the locality bound is only required for components that are not attached to the backbone. As a result, we can delete any cycle that features the bottom part of the split graph. The amount of flow at the bottom graph is called a *witness flow.* This allows us to only consider cycles contained in only the top or only the bottom, even though we have a split graph.

Because we removed all cycles featuring the bottom part, the remaining cycles only cross singleton laminar sets. Notice that for every vertex v in a cycle, it originally has flow at most 1. After rerouting, it has flow at

most 2. Finally, combining the top and bottom graph gives us a flow of at least 4. As a result, the cost of a connected component D on the top is at most $4\sum_{v\in D} 2y_v$. Hence, we have a $(4, 2, 1)$-algorithm for subtour cover.

There are some complications between the witness flow and creating an integral solution, but they are omitted. The proof for $\alpha = 3$ involved requiring that the witness flow be acyclic, hence reducing the in- and out-degree of each vertex. However, maintaining an acyclic witness flow after rerouting is highly technical and therefore omitted as well.

5 Backbone Extension with Subtour Cover

From the previous section, we found a $(3, 2, 1)$-algorithm for subtour cover. We now construct a $(2, 15 + \epsilon)$-efficient extension algorithm. The general theorem is as follows:

Theorem 14. *Suppose there exists an (α, κ, β)-algorithm for Subtour Cover and a fixed constant $\epsilon > 0$. Then, there is a polynomial-time $(\kappa, 4\alpha + \beta + 1 + \epsilon)$-efficient extension algorithm.*

The proof of the theorem involves many highly complicated math expressions, which we will omit in favor of more intuition. To start, we define the notion of a *light* subtour.

Definition 15. *Let H be a subtour cover of $(\mathcal{I}, B)$. Then, H is* α-light *if for every subtour D,*

$$c(D) \leq \sum_{v\in D} l(v).$$

The function $l(v)$ is tedious to define, but it satisfies the following properties:

1. As y_v increases, $l(v)$ increases as well.
2. A vertex $v \in V(B)$ has a much higher value of $l(v)$ than a vertex not on the backbone.
3. If the constants α, β, κ are larger, $l(v)$ is larger as well.
4. $\sum_{v\in V} l(v) \leq \kappa LP_I + (4\alpha + \beta + 1 + \epsilon)\sum_{v\notin V(B)} 2y_v$.

Intuitively, every cycle/subtour containing a vertex v must pay the toll y_v. This should be accounted for in our calculations. Furthermore, a subtour that visits the backbone is much easier to attach than a subtour not visiting the backbone. Therefore, a subtour that visits the backbone should be considered lighter.

Svensson's algorithm starts with some arbitrary subtour cover H. Then, it repeatedly improves H using the subtour cover algorithm until it is 1-light. Then, it attaches H onto the backbone to complete the extension.

This construction gives us intuition for the third property of $l(v)$. A worse subtour cover algorithm means that it is more difficult to improve the cover H. In order for a 1-light algorithm to be feasible, we need to increase $l(v)$. Finally, the fourth property ensures that we get an efficient backbone extension.

Theorem 16. *Suppose that H is 1-light, then attaching H to B gives us a $(\kappa, 4\alpha + \beta + 1 + \epsilon)$-efficient backbone extension.*

Proof. Since H is a subtour cover, every vertex v is in one of the components of H. Since H is 1-light,

$$c(H) = \sum_{D \text{ component}} c(D) \leq \sum_{v\in V} l(v) \leq \kappa LP_I + (4\alpha + \beta + 1 + \epsilon) \sum_{v\notin V(B)} 2y_v.$$

Therefore, we have a $(\kappa, 4\alpha + \beta + 1 + \epsilon)$-efficient backbone extension. □

5.1 Improving H with Local Improvements

We now examine how Svensson's Algorithm proceeds to improve H.

Algorithm 3 Given H, find a lighter subtour cover H'.

1. Run the subtour cover algorithm to create a cover F.
2. Look at each connected component $D \in F$, which links together multiple components of H. Try attaching D to H and remove any extraneous edges from the attachment.
3. If the resulting component is lighter than any individual component, keep the change. Otherwise, do not add D to the graph H.

Compared to the cycle cover algorithm we did on the homework, Svensson's algorithm does not always attach sets together. Instead, it tries to merge sets together only if the resulting component is lighter. We now show that the algorithm is effective.

Theorem 17. *Let H be a subtour cover, H' the result of Svensson's algorithm. Let $\Phi(H)$ be the largest α such that H is α-light. If $\Phi(H) > 1$, then there exists constants C, p such that*

$$\Phi(H) - \Phi(H') > (\frac{1}{Cn} \sum_{v \notin B} l(v))^{1+p}.$$

Proof. Let F be the result of the subtour cover. Since we have an (α, κ, β) algorithm for subtour cover, we know that F is globally cheap and each component of F is light.

Since $\Phi(H) > 1$, there exists some component of H that is heavy. Then, argue that if every component $D \in F$ makes the result heavier, then F cannot be globally cheap. Therefore, there exists some set D that becomes lighter from this.

From here, we perform some gruesome math involving $l(v)$ to bound the size of one improvement. □

Since the initial lightness of H is polynomial, we find that after $poly(n)$ many iterations H will become 1-light. Therefore, we can compute a $(\kappa, 4\alpha + \beta + 1 + \epsilon)$-efficient extension. Then, we plug in the $(3, 2, 1)$ algorithm we have for subtour cover to get a $(22 + \epsilon)$-extension for ATSP, as desired.

6 Conclusion

In this project, we combined three different reductions to create a $(22 + \epsilon)$-approximation for the ATSP problem. We start by adding the structure of strongly laminar ATSP, then introduced the concept of a backbone to contain the cost of an extension in its own recursion level. Finally, we explored how to perform a backbone extension by using subtour cover to make local improvements.

How would a researcher come up with such a series of reduction? We offer the following plausible scenario for constructing this proof.

1. On Laminar instances: Both thin-tree based approach and [Sve15] used the Held-Karp LP, and taking the dual is a natural thing to do when given an LP. Furthermore, strongly laminar instances allow us to phrase the condition for node-weighted TSP very easily, suggesting that they are related. This also suggests us to extend the work of [Sve15].

2. The naive algorithm in section 3 highlighted how many recursion levels is a major issue for this approach. Then, we either need to reduce the number of recursion steps or perform some analysis to ensure that each level of the recursion "pays for itself".

3. The idea of a backbone combines the insight that singleton sets are easy to deal with and the need for a recursion level to "pay for itself". We think this step is the stroke of genius in the [STV20] construction, but it is clear why such a construction would help generalize node-weighted TSP to the general case.

References

[Asa+17] Arash Asadpour et al. "An O (log n/log log n)-approximation algorithm for the asymmetric traveling salesman problem". In: *Operations Research* 65.4 (2017), pp. 1043–1061.

[CFN85] Gérard Cornuéjols, Jean Fonlupt, and Denis Naddef. "The Traveling Salesman Problem on a Graph and Some Related Integer Polyhedra". In: *Math. Program.* 33.1 (Sept. 1985), pp. 1–27. ISSN: 0025-5610. DOI: 10.1007/BF01582008. URL: https://doi.org/10.1007/BF01582008.

[FGM82] A. M. Frieze, G. Galbiati, and F. Maffioli. "On the worst-case performance of some algorithms for the asymmetric traveling salesman problem". In: *Networks* 12.1 (1982), pp. 23–39. DOI: https://doi.org/10.1002/net.3230120103. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/net.3230120103. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/net.3230120103.

[STV20] Ola Svensson, Jakub Tarnawski, and László A. Végh. "A Constant-Factor Approximation Algorithm for the Asymmetric Traveling Salesman Problem". In: *J. ACM* 67.6 (Nov. 2020). ISSN: 0004-5411. DOI: 10.1145/3424306. URL: https://doi.org/10.1145/3424306.

[Sve15] Ola Svensson. "Approximating ATSP by Relaxing Connectivity". In: *CoRR* abs/1502.02051 (2015). arXiv: 1502.02051. URL: http://arxiv.org/abs/1502.02051.

[TV20] Vera Traub and Jens Vygen. "An Improved Approximation Algorithm for ATSP". In: *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2020. Chicago, IL, USA: Association for Computing Machinery, 2020, pp. 1–13. ISBN: 9781450369794. DOI: 10.1145/3357713.3384233. URL: https://doi.org/10.1145/3357713.3384233.