While we have good algorithms for many optimization problems, the unfortunate reality elucidated by theoretical computer science is that so very many important real-world optimization problems are **NP**-hard. What do we do? Suppose we are given an **NP**-hard problem to solve. Assuming $\mathbf{P} \neq \mathbf{NP}$, we can't hope for a polynomial-time algorithm for these problems. But can we get polynomial-time algorithms that always produce a "pretty good" solution? (a.k.a. *approximation algorithms*)

Objectives of this lecture

In this lecture, we will

- define and motivate **approximation algorithms**
- derive two **greedy algorithms** for scheduling jobs on multiple machines to minimize their makespan (a.k.a. stacking blocks to minimize the height of the tallest stack)
- see how **rounding linear programs** can give us an approximate vertex cover
- analyze an approximation algorithm for metric traveling salesperson that works by just solving some graph problems that we already know how to solve

1 Introduction

Given an **NP**-hard problem, we don't hope for a fast algorithm that always gets the optimal solution — if we had such a polynomial algorithm, we would be able to use it to solve everything in NP, and that would imply that $\mathbf{P} = \mathbf{NP}$, something we expect is false. But when faced with an **NP**-hard problem, giving up is not the only reasonable solution! There are several ways that we might try to move forward

- First approach: find a polynomial-time algorithm that guarantees to get at least a "pretty good" solution? E.g., can we guarantee to find a solution that's within 10% of optimal? If not that, then how about within a factor of 2 of optimal? Or, anything non-trivial?
- Second approach: find a faster algorithm than the naïve one. There are a bunch of ideas that you can use for this, often related to dynamic programming. You already saw an example: in Lecture II on Dynamic programming, we gave an algorithm for TSP that ran in time $O(n^2 2^n)$, which is much faster than the naive brute force $O(n \cdot n!) \geq n^{n/2}$ time algorithm.

Today's lecture focuses on the first idea, to derive polynomial-time *approximation algorithms*.

1.1 Formal definition

Definition: Approximation Algorithm

Given some optimization problem with optimal solution value OPT, and an algorithm which produces a feasible solution with value ALG, we say that the algorithm is a *c-approximation algorithm* if ALG is *always* within a factor of c of OPT. The convention differs depending on whether the optimization problem is a minimization or maximization problem.

Minimization An algorithm is a c-approximation ($c > 1$) if for all inputs, $\text{ALG} \leq c \cdot \text{OPT}$.

Maximization An algorithm is a c-approximation ($0 < c < 1$) if for all inputs, $\text{ALG} \geq c \cdot \text{OPT}$.

2 Scheduling Jobs on Multiple Machines to Minimize Load

Problem: Scheduling jobs on multiple machines to minimize the makespan

You have m identical machines on which you want to schedule some n jobs. Each job $i \in \{1, 2, \ldots, n\}$ has a processing time $p_i > 0$. You want to partition the jobs among the machines to minimize the load of the most-loaded machine. In other words, if S_j is the set of jobs assigned to machine j, define the *makespan* of the solution to be

$$\max_{1 \leq j \leq m} \left(\sum_{i \in S_i} p_j \right)$$

You want to minimize the makespan of the solution you output.

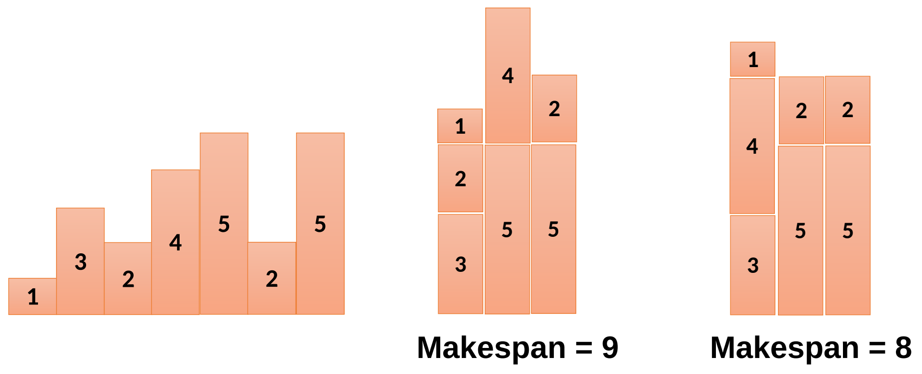
This is the formal definition of the problem that is usually used in textbooks, but here's a nicer (subjectively) more intuitive way to describe the problem.

Problem: Stacking blocks to minimize the height

You have n blocks, the i^{th} of which has height p_i. You want to arrange the blocks into m stacks such that the height of the tallest stack is as short as possible.

Observe that these two problems are exactly the same, just with a different story, but the later (I think) is easier to visualize and think about.

Example Here's an example input to the job scheduling / block stacking problem. Say we have $p = \{1, 3, 2, 4, 5, 2, 5\}$ and $m = 3$. The blocks are shown on the left, and two possible ways to stack them are shown on the right. The makespan is the height of the tallest stack, which is 9 for the first example, and 8 for the second example.



The second example turns out to be optimal. Can you think of a proof of why?

Exercise

Give a concise argument that a makespan of 8 is optimal for the block stacking example above.

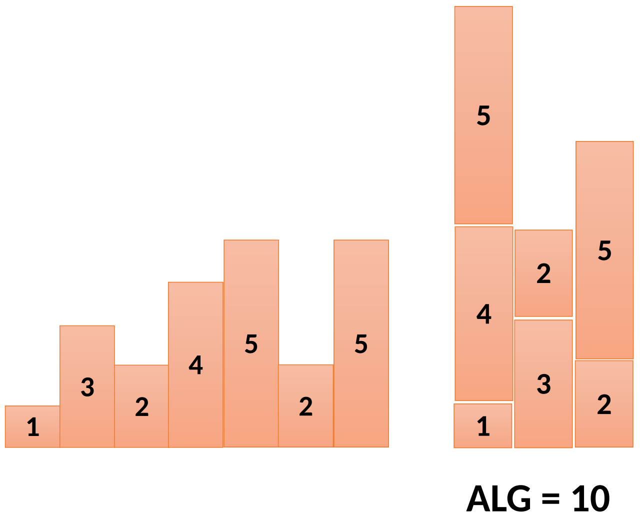
2.1 Algorithms for job scheduling / block stacking

Our first approach to the solve the block stacking problem is a *greedy algorithm*. Recall that greedy algorithms are those that just look for some locally best choice and make that at each step, rather than planning ahead in any way. Greedy algorithms are often very good for producing approximations.

Algorithm: Greedy job scheduling / block stacking

Start with m empty stacks, then, for each block, place it on the current shortest stack.

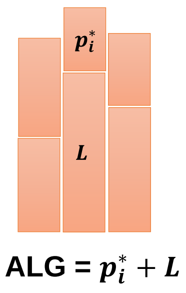
Applying this to the example above, we would get the following configuration, which has a makespan of 10, which is only 25% more than the optimal, so not too bad.



But that was just one example, how bad can it get in general? We need to prove that this algorithm always gives us something that is pretty good, or near optimal.

Theorem 1: Quality of greedy job scheduling

The greedy approach outputs a solution with makespan at most 2 times the optimum, i.e., it is a 2-approximation algorithm.



Proof. Let's start by looking at the height of the tallest stack in our solution, since this is what defines the makespan (the answer). Call the last block added to the tallest stack i^*, so its height is p_{i^*}. Now call the remaining height of the tallest stack L. So we have by definition ALG $= L + p_{i^*}$. The hardest part of these greedy algorithm proofs is relating the value of ALG to the value of OPT.

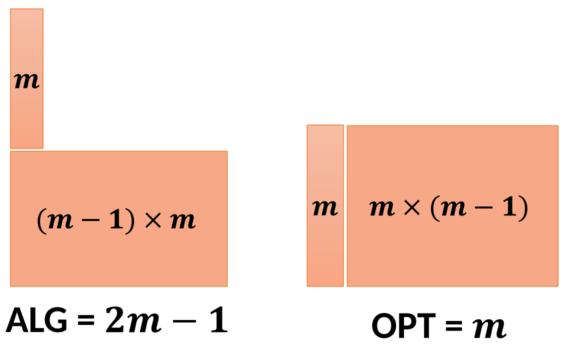
First, note that since every block must be placed somewhere, OPT $\geq p_i$ for all i and specifically, OPT $\geq p_{i^*}$. What can we say about L? Remember that the algorithm always chooses the *shortest stack* to place the next block, so when it decided to place i^* on the stack, it was because L was the height of the shortest stack at the time. This means that every stack has height at least L, which means that OPT $\geq L$.

So, combining these two inequalities, we get

$$\text{ALG} = L + p_{i^*} \leq \text{OPT} + \text{OPT} = 2\text{OPT}$$

2.2 A worst-case example

Is this analysis tight? Sadly, yes. Suppose we have $m(m-1)$ jobs of size 1, and 1 job of size m, and we schedule the small jobs before the large jobs. The greedy algorithm will spread the small jobs equally over all the machines, and then the large job will stick out, giving a makespan of $2m-1$, whereas the right thing to do is to spread the small jobs over $m-1$ machines and get a makespan of m.



The approximation ratio in this case is $\frac{2m-1}{m} = 2(1-\frac{1}{m}) \approx 2$, so this looks almost tight. It can actually be made tight with a bit more analysis.

Exercise

Improve the analysis slightly to show that the approximation ratio of the greedy algorithm is actually $2(1-\frac{1}{m})$, which makes the above example tight.

Can we get a better algorithm? The worst-case example helps us see the problem: when small jobs come before big jobs they can cause big problems! So lets prevent this...

2.3 An improved greedy algorithm

Algorithm: Sorted greedy job scheduling / block stacking

Sort the blocks from biggest to smallest, then do the greedy algorithm.

This algorithm prevents the worst-case example that we showed before, but what does it do in general? Let's prove that it is in fact an improvement.

Theorem 2: Quality of sorted greedy job scheduling

The sorted greedy approach outputs a solution with makespan at most 1.5 times the optimum, i.e., it is a 1.5-approximation algorithm.

Proof. Let's use the same setup as before and say that i^* is the last block added to the tallest stack, and L is the height of the rest of the stack underneath i^*, so the makespan (tallest stack) is $L + p_{i^*}$. We still have the facts that OPT $\geq L$ and OPT $\geq p_{i^*}$. We need to make some new observation in order to get the approximation ratio lower.

First, suppose $L > 0$, which means there are some blocks underneath i^*. Since the blocks were processed in sorted order (**important**), all of the blocks underneath are at least as large. Furthermore, since L was the height of the shortest stack at the time that i^* was added, every other stack is also non-empty and contains blocks that are at least as large as i^*. From this, we can deduce that there exists at least $m+1$ blocks of size at least p_{i^*} because there was at least one in each of the m stacks before we processed i^*. So, by the pigeonhole principle, since there are only m stacks, for any possible configuration, there must always be a stack that contains two blocks of size at least p_{i^*}. Therefore $\text{OPT} \geq 2p_{i^*}$, or equivalently $p_{i^*} \leq \frac{1}{2}\text{OPT}$.

Combining this with our original inequality, we get

$$\text{ALG} = L + p_{i^*} \leq \text{OPT} + \tfrac{1}{2}\text{OPT} = 1.5\text{OPT}.$$

Now we have an edge case to deal with. What if $L = 0$? Then we can't say that there are any blocks underneath i^* or make any argument about the number of large blocks. In this case, we just have $\text{ALG} = p_{i^*}$, but we know that $\text{OPT} \geq p_{i^*}$, so actually $\text{ALG} = \text{OPT}$, so we get the exact answer in this case. $\square$

Exercise

It is possible to show that the makespan of Sorted Greedy is at most $\frac{4}{3}\text{OPT}$. Try it.

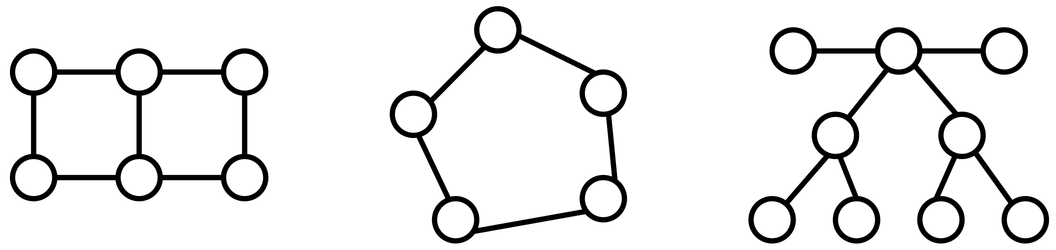
3 Vertex Cover via LP Rounding

Recall that a *vertex cover* in a graph is a set of vertices such that every edge is incident to (covers) at least one of them. The minimum vertex cover problem is to find the smallest such set of vertices.

Definition: Minimum Vertex Cover

Given a graph G, find a smallest set of vertices such that every edge is incident to at least one of them.

For instance, this problem is like asking: what is the fewest number of guards we need to place in a museum in order to cover all the corridors.



Exercise

Find a vertex cover in the graphs above of size 3. Argue that there is no vertex cover of size 2.

3.1 A linear-programming-based algorithm

We don't expect to find the optimal solution in polynomial-time, but we will show in this section that we can use *linear programming* to obtain a 2-approximate solution. That is, if the graph G has a vertex cover of size k^*, we can return a vertex cover of size at most $2k^*$. Let's remind ourselves of the LP *relaxation* of the vertex cover problem:

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} w_v & \\ \text{s.t.} \quad & w_u + w_v \geq 1 & \forall \{u, v\} \in E \\ & w_v \geq 0 & \forall v \in V \end{aligned}$$

Remember that in the integral version of the problem, the variables denote that a vertex v is in the cover if $x_v = 1$ and not in the cover if $x_v = 0$. Solving the integral version is NP-hard, so we settle for a relaxation, where we allow fractional values of x_v, so a vertex can be "half" in the cover. This is called an "LP relaxation" because any true vertex cover is a feasible solution, but we've made the problem easier by allowing fractional solutions too. So, the value of the optimal solution now will be at least as good as the smallest vertex cover, maybe even better (i.e., smaller), but it just might not be legal any more.

Since, in an actual vertex cover we can not take half of a vertex, our goal is to convert this fractional solution into an actual vertex cover. Here's a natural idea.

Algorithm: Relax-and-round for vertex cover

Solve the LP relaxation for x_v for each $v \in V$, then pick each vertex for which $x_v \geq \frac{1}{2}$

This is called *rounding* the linear program (which literally is what we are doing here by rounding the fraction to the nearest integer — for other problems, the "rounding" step might not be so simple).

Theorem: Relax-and-round is a 2-approximation

The above algorithm is a 2-approximation to VERTEX-COVER.

Proof. We need to prove two things. First, that we actually obtain a valid vertex cover (every edge must be incident to a vertex that we pick) and that the size of the resulting cover is not too large.

Feasibility Suppose for the sake of contradiction that there exists an edge (u, v) that is not covered. Then that means we did not pick either u or v, which means that $x_u < \frac{1}{2}$ and $x_v < \frac{1}{2}$. Therefore, $x_u + x_v < 1$, but this contradicts the LP constraint that $x_u + x_v \geq 1$. So the algorithm always outputs a vertex cover.

Approximation ratio Let LP denote the objective value of the relaxation. Since it is a relaxation, its solution can not be worse than the optimal integral solution (the actual minimum vertex cover), so LP $\leq$ OPT. Furthermore, when we round the solution, in the worst case, we increase variables from $\frac{1}{2}$ to 1, so we double their value. Since the objective is $\sum x_v$, this at most doubles the objective value, so ALG $\leq$ 2LP. Combining these inequalities, we get ALG $\leq$ 2LP $\leq$ 2OPT. □

3.2 Hardness of Approximation

Interesting fact: nobody knows any approximation algorithm for vertex cover with approximation ratio 1.99. Best known is $2 - O(1/\sqrt{\log n})$, which is $2 - o(1)$.

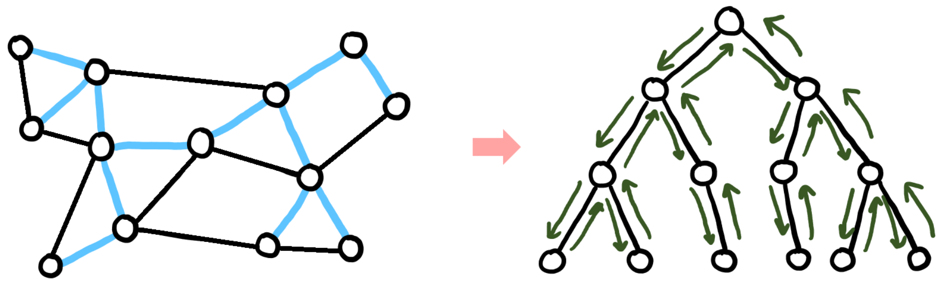
There are results showing that a good-enough approximation algorithm will end up showing that **P**=**NP**. Clearly, a 1-approximation would find the exact vertex cover, and show this. Håstad showed that if you get a 7/6-approximation, you would prove **P**=**NP**. This 7/6 was improved to 1.361 by Dinur and Safra. Beating $2 - \varepsilon$ has been related to some other problems (it is "unique games hard"), but is not known to be **NP**-hard.

4 Metric Traveling Salesperson

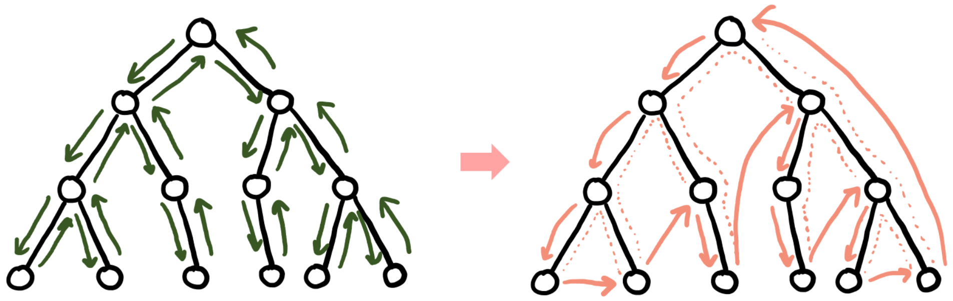
The metric traveling salesperson (Metric TSP) problem asks you to find the shortest path to visit n cities, each exactly once, returning to where you started. You are given distances d_{ij} between any pair of cities i and j (assume the graph is complete, you have all the distances). The "metric" part of the problem is that these distances are symmetric and obey the triangle inequality. That is, for any pair of vertices u, v and any intermediate vertex k, we always have $d(u, v) \leq d(u, k) + d(k, v)$. In other words, its never worse to go somewhere directly rather than via some other vertex.

To solve the problem, we'd like to re-use some previous problems that we already know how to solve and relate their solution to the solution of the TSP problem. To gain some intuition here, remember that the TSP problem is essentially asking for a cycle that visits every vertex of the graph and has minimum possible weight. That sounds quite similar to another problem we know... A problem where we want to pick a bunch of edges that connects the whole graph but minimizes the total weight. It sounds a lot like the minimum spanning tree (MST) problem! The difference is that we now have to build a cycle rather than a tree so we are a bit more constrained. Lets think about how we might pull a cycle out of an MST.

Here's an idea. Root the MST arbitrarily, now just do a depth-first traversal, and record each time we go down or up each edge. This uses each edge in the MST twice, but gives us a clean cycle that starts at the root, visits each vertex (at least) once and returns to the root. Since it uses the edges of the MST, it is hopefully not too expensive. Here's a diagram to illustrate. Say the MST of the graph is in blue, then we lay it out with an arbitrary root on the right and perform the depth-first traversal.



This sounds pretty promising, but there's a technicality we have to solve. This traversal visits every vertex *at least* once, but it visits many vertices multiple times, which is not allowed in the final TSP. What can we do about that? Well, the simplest solution is just to *not* do that. Any time we encounter a duplicate vertex, lets just skip it and go to the next non-duplicate that the traversal would visit. This looks like the following.



We call the process of cutting out duplicates "shortcutting". We are taking a shortcut from the traversal and going straight to the next non-duplicate! Now here's the last cool observation we need. What did we just end up with? When we shortcutted the traversal, we ended up with a sequence of vertices in a depth-first traversal order... Hmmm, this is nothing but the *pre-order* of the vertices in the tree!

Algorithm: MST Approximation for Metric TSP

- Compute an MST M of the graph G
- Root M at an arbitrary vertex r
- Output a pre-order of M followed by r (return to the root)

Theorem: MST-TSP is a 2-approximation

This MST approach gives a 2-approximation to the METRIC TSP problem.

Proof. First, we want to compare the cost of the optimal TSP tour with the weight of the MST. Consider any TSP tour (i.e., a cycle that visits every vertex). If we remove any one edge of the tour, what are we left with? We will have $n - 1$ edges that connect all n vertices... Oh, that's just a spanning tree! Since its a spanning tree, by definition its weight can not be less than that of the MST, so

$$\text{weight}(MST) \le \text{weight}(TSP - \{e\}) \le \text{weight}(TSP).$$

Now how costly is our solution? We started by traversing the MST depth-first, using each edge twice, one down then once back up. Therefore the cost of this walk was 2weight(MST). Then we had to shortcut out the duplicates to obtain a valid TSP tour. However, remember that the triangle inequality says that going directly to a vertex is never worse than taking a detour, so therefore this shortcutting process can only make the cost lower, so

$$\text{ALG} \le 2 \cdot \text{weight}(MST) \le 2 \cdot \text{weight}(TSP) = 2\text{OPT}.$$

□

The second step is where we needed the assumption that the distances were a metric. Without it, we can not make the argument that the shortcuts do not increase the total cost.

5 Some more approximation algorithms

Optional content — Will not appear on the homeworks or the exams

5.1 Greedy Algorithm for Set Cover

The SET-COVER problem is defined as follows:

Definition 1: SET-COVER

Given a domain X of n points, and m subsets $S_1, S_2, \ldots, S_m$ of these points, find the fewest number of these subsets needed to cover all the points.

SET-COVER is NP-Hard. However, there is a simple algorithm that gets an approximation ratio of $\ln n$ (i.e., that finds a cover using at most a factor $\ln n$ more sets than the optimal solution).

Algorithm: Greedy set cover

Pick the set that covers the most points. Throw out all the points covered. Repeat.

What's an example where this algorithm *doesn't* find the best solution?

Theorem: Greedy set cover is a $\ln n$-approximation

If the optimal solution uses k sets, the greedy algorithm finds a solution with at most $O(k \ln n)$ sets.

Proof. Since the optimal solution uses k sets, there must some set that covers at least a $1/k$ fraction of the points. The algorithm chooses the set that covers the most points, so the first set it chooses covers at least that many. Therefore, after the first iteration of the algorithm, there are at most $n(1 - 1/k)$ points left. Again, since the optimal solution uses k sets, there must some set that covers at least a $1/k$ fraction of the remainder (if we got lucky we might have chosen one of the sets used by the optimal solution and so there are actually $k - 1$ sets covering the remainder, but we can't count on that necessarily happening). So, again, since we choose the set that covers the most points remaining, after the second iteration, there are at most $n(1 - 1/k)^2$ points left. Generally, after t rounds, there are at most $n(1 - 1/k)^t$ points left. After $t = k \ln n$ rounds, there are at most $n(1 - 1/k)^{k \ln n} < n(1/e)^{\ln n} = 1$ points left, which means we are done. $\square$

Also, you can get a slightly better bound by using the fact that after $k \ln(n/k)$ rounds, there are at most $n(1/e)^{\ln(n/k)} = k$ points left, and (since each new set covers at least one point) you only need to go k more steps. This gives the somewhat better bound of $k \ln(n/k) + k$. So, we have:

Theorem: Better bound for greedy set cover

If the optimal solution uses k sets, the greedy algorithm finds a solution with at most $k \ln(n/k) + k$ sets.

This may be about the best you can do. Dinur and Steurer (improving on results of Feige) showed that if you get a $(1 - \epsilon) \ln(n)$-approximation algorithm for any constant $\epsilon > 0$, you will prove that **P=NP**.

5.2 Christofides' Algorithm for Metric TSP

We can improve on the MST-based 2-approximation algorithm by looking at what we did in a different way. What we did was: find the MST T, and create a new graph G' that contains 2 copies of each edge from T (that is, G' is a multigraph). We then found an Eulerian tour of G. We transform the Eulerian tour into a TSP tour by skipping cities we've already visited (shortcutting).

We can generalize this idea to other "spanning Eulerian multigraphs".

Definition: Spanning Eulierian Multigraph

A spanning Eulerian multigraph of G is a multigraph that

1. contains a single component with all of the cities of G,
2. contains some number (≥ 0) of copies of the edges between cities,
3. is Eulerian.

Theorem: Building TSP tours from Eulerian tours

If H is a spanning Eulerian multigraph of G and $cost(H)$ is the cost of the Eulerian tour on H, then we can efficiently find a TSP tour of cost $\leq cost(H)$.

Proof. Find the Eulerian tour on H (of $cost(H)$), turn it into a TSP tour via shortcuts of lower cost. □

For the MST-based algorithm, we used the "double MST" as the spanning Eulerian multigraph (SEM), but if we can find a lower cost SEM, we could potentially do better. That leads to Christofides'[1] algorithm.

Algorithm: Christofides' Algorithm

Compute the MST T of G. Compute a minimum weight perfect matching M between the vertices of *odd degree* in T. Construct the spanning Eulerian multigraph $G' = T \cup M$ (that is, G' contains the edges from T and the edges from the matching). Return the TSP tour constructed from shortcutting an Eulerian tour on G'.

Theorem

Christofides' gives a 3/2-approximation algorithm for METRIC TSP.

Proof. We can always construct G: there are always an even number of odd-degree vertices in T. (True for any undirected graph because $\sum_v degree(v) = 2|E|$).

G is a spanning Eulerian multigraph: it contains all the vertices. Vertices that had even degree in T have even degree in G. Vertices that have odd degree in T have 1 more edge incident on them (from the matching) so now they have even degree. Since all the vertices in G have even degree, G is Eulerian.

Let $cost(E)$ be the length of the Eulerian tour on G and $cost(C)$ be the cost of the TSP tour extracted from it by shortcutting. Note that $cost(C) \leq cost(E) = cost(T) + cost(M)$.

We have $cost(T) \leq cost(A^*)$ (same reason as before).

[1]This algorithm was proposed in 1976 by Nicos Christofides, who was at Carnegie Mellon University at the time. The result appeared as a technical report of the Graduate School of Industrial Administration (GSIA), which is now part of the Tepper School of Business.

Let $v_1, \ldots, v_k$ be the odd degree vertices, *listed in order that the appear in the optimal tour*. Form two matchings from these:

$$M_1 = (v_1, v_2), (v_3, v_4), \ldots, (v_{k-1}, v_k)$$
$$M_2 = (v_2, v_3), (v_4, v_5), \ldots, (v_k, v_1)$$

We then have $cost(M_1) + cost(M_2) \leq cost(A^*)$ since $M_1 \cup M_2$ is a shortcutting of the optimal tour. Since our matching M is the least weight, we have:

$$2cost(M) \leq cost(M_1) + cost(M_2) \leq cost(A^*)$$

This gives us:

$$cost(C) \leq cost(E) = cost(T) + cost(M) \leq cost(A^*) + (1/2)cost(A^*) = (3/2)cost(A^*)$$

$\square$

A paper from 2020 by Karlin, Klein, and Oveis Gharan presents an improved algorithm, which gives a 1.4999999999999999999999999999999999 approximation.[2]

[2] This is not an April fools joke. See `https://arxiv.org/abs/2007.01409`.