Approximation Algorithms

Math 482, Lecture 36

Misha Lavrov

May 4, 2020

Approximate solutions

Integer programming is often really hard, and we'd like to be able to say "here is a method that is not optimal, but is good enough."

Approximate solutions

Integer programming is often really hard, and we'd like to be able to say "here is a method that is not optimal, but is good enough." But what does "good enough" mean?

Approximate solutions

Integer programming is often really hard, and we'd like to be able to say "here is a method that is not optimal, but is good enough." But what does "good enough" mean?

Some possible definitions, in terms of the optimal value z^*:

- "approximation difference of δ": find a solution with objective value z such that $|z - z^*| < \delta$.

Approximate solutions

Integer programming is often really hard, and we'd like to be able to say "here is a method that is not optimal, but is good enough." But what does "good enough" mean?

Some possible definitions, in terms of the optimal value z^*:

- "approximation difference of δ": find a solution with objective value z such that $|z - z^*| < \delta$.

  This doesn't work well: it's sensitive to units, and usually not any easier than finding the optimal solution.

Approximate solutions

Integer programming is often really hard, and we'd like to be able to say "here is a method that is not optimal, but is good enough." But what does "good enough" mean?

Some possible definitions, in terms of the optimal value z^*:

- "approximation difference of δ": find a solution with objective value z such that $|z - z^*| < \delta$.

  This doesn't work well: it's sensitive to units, and usually not any easier than finding the optimal solution.

- *approximation ratio* of ρ: find a solution with objective value z such that $\frac{1}{\rho}z^* \leq z \leq \rho z^*$.

Approximate solutions

Integer programming is often really hard, and we'd like to be able to say "here is a method that is not optimal, but is good enough." But what does "good enough" mean?

Some possible definitions, in terms of the optimal value z^*:

- "approximation difference of δ": find a solution with objective value z such that $|z - z^*| < \delta$.

  This doesn't work well: it's sensitive to units, and usually not any easier than finding the optimal solution.

- *approximation ratio* of ρ: find a solution with objective value z such that $\frac{1}{\rho}z^* \leq z \leq \rho z^*$.

  This is the approach we'll use!

Approximation algorithms

Definition

A *ρ-approximation algorithm* for a problem is an algorithm that always finds a solution whose objective value is within a factor of ρ of the optimal objective value.

Approximation algorithms

Definition

A *ρ-approximation algorithm* for a problem is an algorithm that always finds a solution whose objective value is within a factor of ρ of the optimal objective value.

Today, we'll look at:

- Several 2-approximation algorithms for **vertex cover**.

Approximation algorithms

Definition

A *ρ-approximation algorithm* for a problem is an algorithm that always finds a solution whose objective value is within a factor of ρ of the optimal objective value.

Today, we'll look at:

- Several 2-approximation algorithms for **vertex cover**.

  This means we find a vertex cover whose size is at most twice the optimal size.

Approximation algorithms

Definition

A *ρ-approximation algorithm* for a problem is an algorithm that always finds a solution whose objective value is within a factor of ρ of the optimal objective value.

Today, we'll look at:

- Several 2-approximation algorithms for **vertex cover**.

  This means we find a vertex cover whose size is at most twice the optimal size.

- A 2-approximation algorithm for the **traveling salesman problem**.

  This means we find a tour whose cost is at most twice the optimal cost.

The vertex cover problem

In the vertex cover problem, we are given a graph. We want to choose the smallest set of vertices S, such that every edge has at least one endpoint in S.

The vertex cover problem

In the vertex cover problem, we are given a graph. We want to choose the smallest set of vertices S, such that every edge has at least one endpoint in S.

- We saw that for bipartite graphs, this can be solved by solving a max-flow problem and taking the minimum cut.

The vertex cover problem

In the vertex cover problem, we are given a graph. We want to choose the smallest set of vertices S, such that every edge has at least one endpoint in S.

- We saw that for bipartite graphs, this can be solved by solving a max-flow problem and taking the minimum cut.
- For general graphs, the best known exact algorithms take exponential time.

The vertex cover problem

In the vertex cover problem, we are given a graph. We want to choose the smallest set of vertices S, such that every edge has at least one endpoint in S.

- We saw that for bipartite graphs, this can be solved by solving a max-flow problem and taking the minimum cut.
- For general graphs, the best known exact algorithms take exponential time.

The greedy algorithm is to add nodes to S until we have a vertex cover. If we're clever, we can try to add the node that covers the most still-uncovered edges.

The vertex cover problem

In the vertex cover problem, we are given a graph. We want to choose the smallest set of vertices S, such that every edge has at least one endpoint in S.

- We saw that for bipartite graphs, this can be solved by solving a max-flow problem and taking the minimum cut.
- For general graphs, the best known exact algorithms take exponential time.

The greedy algorithm is to add nodes to S until we have a vertex cover. If we're clever, we can try to add the node that covers the most still-uncovered edges. This is not going to find the optimal solution, but does it get a good approximation?

Approximation ratios for the greedy algorithm

Answer: No! Even for bipartite graphs, the greedy algorithm can have an arbitrarily bad approximation ratio.

Approximation ratios for the greedy algorithm

Answer: No! Even for bipartite graphs, the greedy algorithm can have an arbitrarily bad approximation ratio.

We will construct a bipartite graph $G_{n,k}$ as follows:

Approximation ratios for the greedy algorithm

Answer: No! Even for bipartite graphs, the greedy algorithm can have an arbitrarily bad approximation ratio.

We will construct a bipartite graph $G_{n,k}$ as follows:

- On one side, a set of vertices A with $|A| = n$.

Approximation ratios for the greedy algorithm

Answer: No! Even for bipartite graphs, the greedy algorithm can have an arbitrarily bad approximation ratio.

We will construct a bipartite graph $G_{n,k}$ as follows:

- On one side, a set of vertices A with $|A| = n$.
- On the other side, $k - 1$ sets of vertices $B_2, B_3, \ldots, B_k$ with $|B_i| = \lfloor \frac{n}{i} \rfloor$. (Let $B = B_2 \cup B_3 \cup \cdots \cup B_k$.)

Approximation ratios for the greedy algorithm

Answer: No! Even for bipartite graphs, the greedy algorithm can have an arbitrarily bad approximation ratio.

We will construct a bipartite graph $G_{n,k}$ as follows:

- On one side, a set of vertices A with $|A| = n$.
- On the other side, $k - 1$ sets of vertices $B_2, B_3, \ldots, B_k$ with $|B_i| = \lfloor \frac{n}{i} \rfloor$. (Let $B = B_2 \cup B_3 \cup \cdots \cup B_k$.)
- For every i, each vertex in A has one edge to B_i, evenly distributed so that each vertex in B_i gets i or $i + 1$ edges to A.

Approximation ratios for the greedy algorithm

Answer: No! Even for bipartite graphs, the greedy algorithm can have an arbitrarily bad approximation ratio.

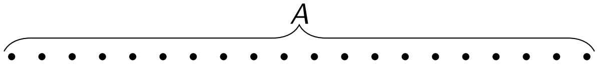
We will construct a bipartite graph $G_{n,k}$ as follows:

- On one side, a set of vertices A with $|A| = n$.
- On the other side, $k-1$ sets of vertices $B_2, B_3, \ldots, B_k$ with $|B_i| = \lfloor \frac{n}{i} \rfloor$. (Let $B = B_2 \cup B_3 \cup \cdots \cup B_k$.)
- For every i, each vertex in A has one edge to B_i, evenly distributed so that each vertex in B_i gets i or $i+1$ edges to A.

There is a vertex cover of size n: take all of A. But we'll see that the greedy algorithm finds the vertex cover B instead!

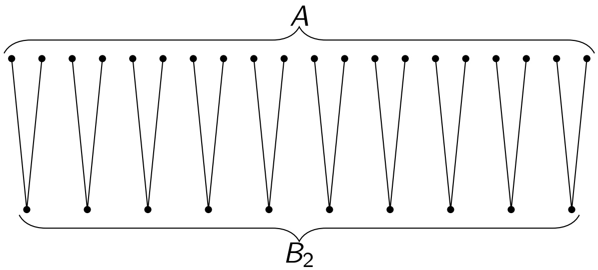
The construction

Here is the graph $G_{20,5}$ as an example:



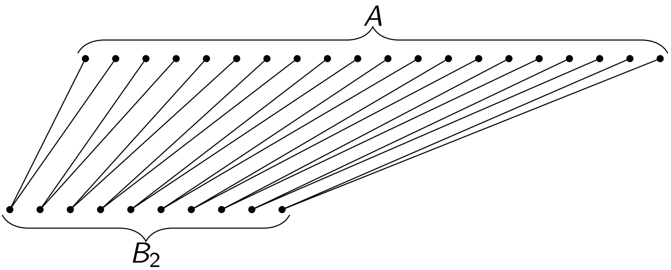
The construction

Here is the graph $G_{20,5}$ as an example:



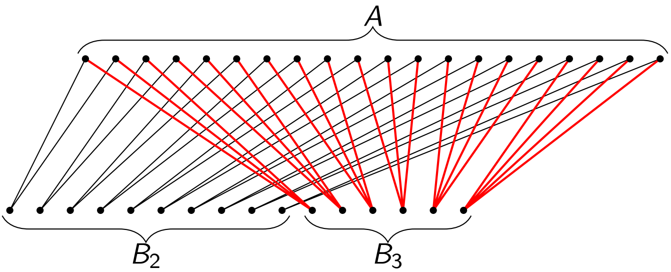
The construction

Here is the graph $G_{20,5}$ as an example:



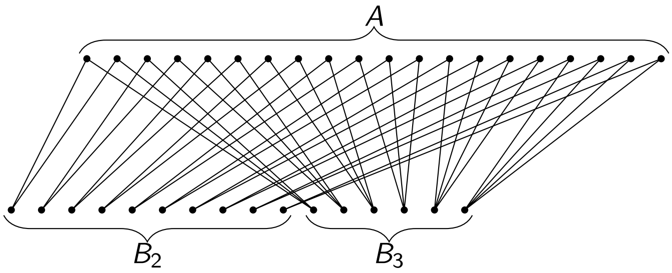
The construction

Here is the graph $G_{20,5}$ as an example:



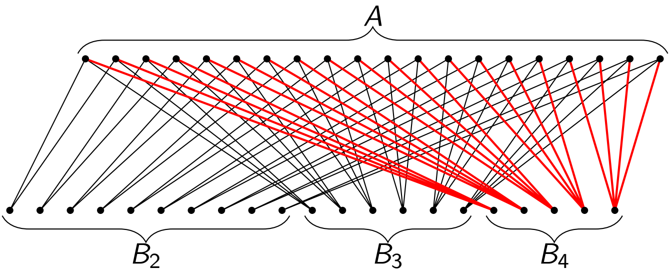
The construction

Here is the graph $G_{20,5}$ as an example:



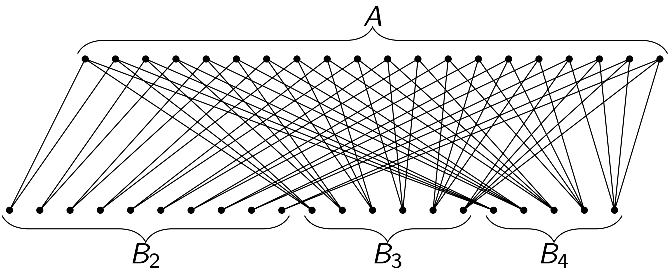
The construction

Here is the graph $G_{20,5}$ as an example:



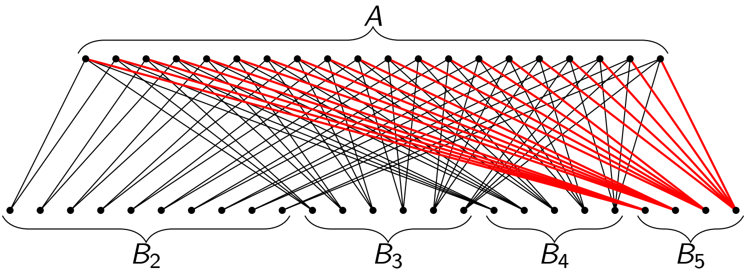
The construction

Here is the graph $G_{20,5}$ as an example:



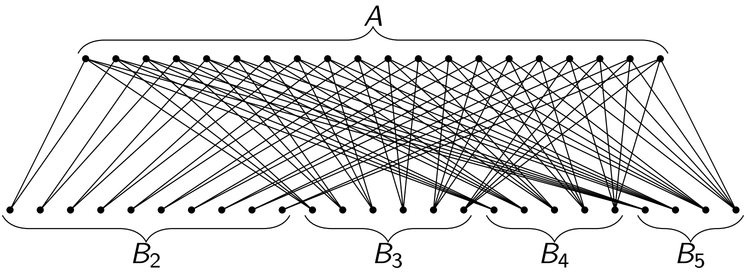
The construction

Here is the graph $G_{20,5}$ as an example:



The construction

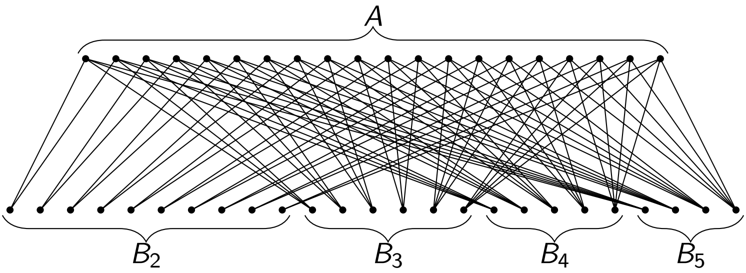
Here is the graph $G_{20,5}$ as an example:



In general, $G_{n,k}$ has n vertices in A and about $(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k})n$ vertices in B.

The construction

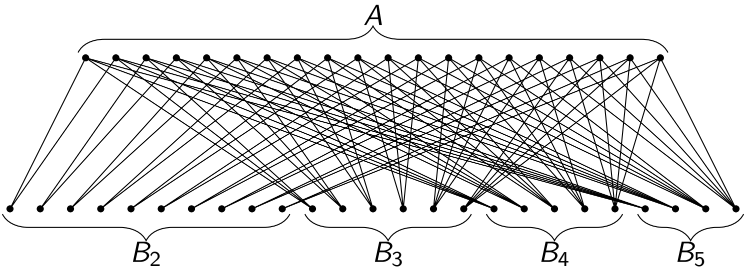
Here is the graph $G_{20,5}$ as an example:



In general, $G_{n,k}$ has n vertices in A and about $(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k})n$ vertices in B.

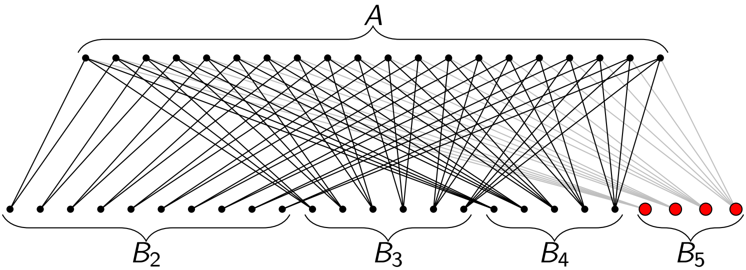
Each vertex in A has $k-1$ neighbors. But the vertices in B_k have k or $k+1$ neighbors, so the algorithm will choose them first.

The greedy algorithm on $G_{n,k}$



At first, vertices in B_k look most promising, and they get picked.

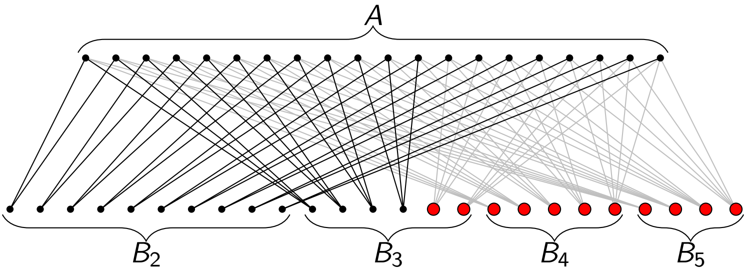
The greedy algorithm on $G_{n,k}$



At first, vertices in B_k look most promising, and they get picked.

But now, each vertex in A can cover one fewer uncovered edge. They look worse than vertices in B_{k-1}, so those will be picked next.

The greedy algorithm on $G_{n,k}$

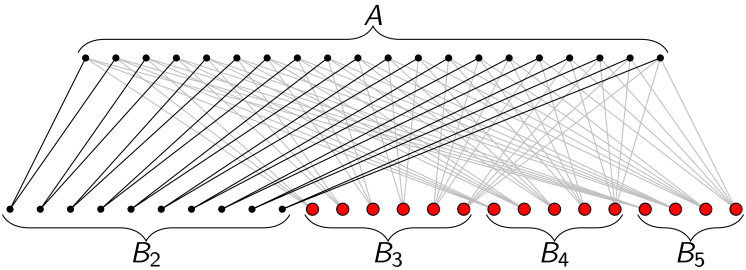


At first, vertices in B_k look most promising, and they get picked.

But now, each vertex in A can cover one fewer uncovered edge.
They look worse than vertices in B_{k-1}, so those will be picked next.

This continues; we pick vertices from B at each step.

The greedy algorithm on $G_{n,k}$

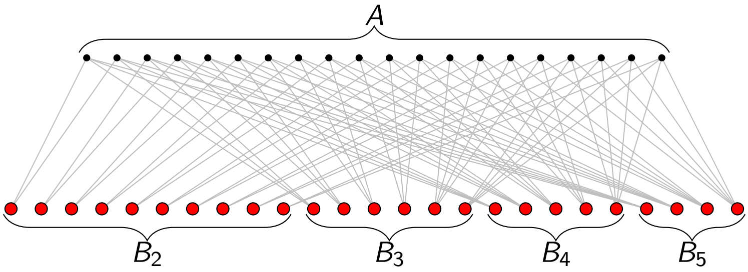


At first, vertices in B_k look most promising, and they get picked.

But now, each vertex in A can cover one fewer uncovered edge. They look worse than vertices in B_{k-1}, so those will be picked next.

This continues; we pick vertices from B at each step.

The greedy algorithm on $G_{n,k}$



At first, vertices in B_k look most promising, and they get picked.

But now, each vertex in A can cover one fewer uncovered edge. They look worse than vertices in B_{k-1}, so those will be picked next.

This continues; we pick vertices from B at each step.

Eventually, we'll have picked all of B. The approximation ratio is $\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k}$, which can be arbitrarily bad.

Another algorithm

Here's another, better algorithm:

Another algorithm

Here's another, better algorithm:

1. Start by greedily picking a matching M: start with $M = \varnothing$, then as long as there's an edge e that shares no endpoints with edges of M, add e to M.

Another algorithm

Here's another, better algorithm:

1. Start by greedily picking a matching M: start with $M = \varnothing$, then as long as there's an edge e that shares no endpoints with edges of M, add e to M.

   (This is not necessarily the largest matching. But it will be "maximal": once we stop, all edges share an endpoint with at least one edge of M.)

Another algorithm

Here's another, better algorithm:

1. Start by greedily picking a matching M: start with $M = \varnothing$, then as long as there's an edge e that shares no endpoints with edges of M, add e to M.

   (This is not necessarily the largest matching. But it will be "maximal": once we stop, all edges share an endpoint with at least one edge of M.)

2. Take all $2|M|$ endpoints of the edges of M to be our vertex cover.

Another algorithm

Here's another, better algorithm:

1. Start by greedily picking a matching M: start with $M = \varnothing$, then as long as there's an edge e that shares no endpoints with edges of M, add e to M.

   (This is not necessarily the largest matching. But it will be "maximal": once we stop, all edges share an endpoint with at least one edge of M.)

2. Take all $2|M|$ endpoints of the edges of M to be our vertex cover.

Any vertex cover needs to contain at least $|M|$ vertices: it needs to cover all edges in M, and each vertex can only cover one of those edges. (This bound also follows from weak duality.)

Another algorithm

Here's another, better algorithm:

1. Start by greedily picking a matching M: start with $M = \varnothing$, then as long as there's an edge e that shares no endpoints with edges of M, add e to M.

   (This is not necessarily the largest matching. But it will be "maximal": once we stop, all edges share an endpoint with at least one edge of M.)

2. Take all $2|M|$ endpoints of the edges of M to be our vertex cover.

Any vertex cover needs to contain at least $|M|$ vertices: it needs to cover all edges in M, and each vertex can only cover one of those edges. (This bound also follows from weak duality.)

But we've only taken $2|M|$ vertices, so we have a 2-approximation algorithm!

Weighted vertex cover

A generalization of this is weighted vertex cover: here, every vertex i has a weight w_i, and we want to choose the vertex cover with the least total weight.

Weighted vertex cover

A generalization of this is weighted vertex cover: here, every vertex i has a weight w_i, and we want to choose the vertex cover with the least total weight.

The integer program for this is:

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{Z}^{|V|}}{\text{minimize}} & \displaystyle\sum_{i\in V} w_i x_i & \\ \text{subject to} & x_i + x_j \geq 1 & \text{for all } ij \in E \\ & \mathbf{0} \leq \mathbf{x} \leq \mathbf{1} & \end{array}$$

Weighted vertex cover

A generalization of this is weighted vertex cover: here, every vertex i has a weight w_i, and we want to choose the vertex cover with the least total weight.

The integer program for this is:

$$\begin{array}{lll} \underset{\mathbf{x}\in\mathbb{Z}^{|V|}}{\text{minimize}} & \displaystyle\sum_{i\in V} w_i x_i & \\ \text{subject to} & x_i + x_j \geq 1 & \text{for all } ij \in E \\ & \mathbf{0} \leq \mathbf{x} \leq \mathbf{1} & \end{array}$$

Our previous 2-approximation algorithm doesn't work anymore: if one endpoint of an edge has weight 1 and the other has weight 99, then choosing both endpoints is 100 times as bad as choosing one endpoint!

LP rounding

Instead, we can use a technique that's very common in approximation algorithms.

1. Solve the linear relaxation of this integer program.

LP rounding

Instead, we can use a technique that's very common in approximation algorithms.

1. Solve the linear relaxation of this integer program.
2. Round the solution $\mathbf{x}$ to an integer solution $\mathbf{x}'$: when $x_i \geq \frac{1}{2}$, set $x_i' = 1$, and when $x_i < \frac{1}{2}$, set $x_i' = 0$.

LP rounding

Instead, we can use a technique that's very common in approximation algorithms.

1. Solve the linear relaxation of this integer program.
2. Round the solution $\mathbf{x}$ to an integer solution $\mathbf{x}'$: when $x_i \geq \frac{1}{2}$, set $x_i' = 1$, and when $x_i < \frac{1}{2}$, set $x_i' = 0$.

Why is $\mathbf{x}'$ still a feasible solution?

LP rounding

Instead, we can use a technique that's very common in approximation algorithms.

1. Solve the linear relaxation of this integer program.
2. Round the solution $\mathbf{x}$ to an integer solution $\mathbf{x}'$: when $x_i \geq \frac{1}{2}$, set $x_i' = 1$, and when $x_i < \frac{1}{2}$, set $x_i' = 0$.

Why is $\mathbf{x}'$ still a feasible solution? Because if $x_i + x_j \geq 1$, then either $x_i \geq \frac{1}{2}$ or $x_j \geq \frac{1}{2}$ (or both), so either $x_i' = 1$ or $x_j' = 1$.

LP rounding

Instead, we can use a technique that's very common in approximation algorithms.

1. Solve the linear relaxation of this integer program.
2. Round the solution $\mathbf{x}$ to an integer solution $\mathbf{x}'$: when $x_i \geq \frac{1}{2}$, set $x_i' = 1$, and when $x_i < \frac{1}{2}$, set $x_i' = 0$.

Why is $\mathbf{x}'$ still a feasible solution? Because if $x_i + x_j \geq 1$, then either $x_i \geq \frac{1}{2}$ or $x_j \geq \frac{1}{2}$ (or both), so either $x_i' = 1$ or $x_j' = 1$.

How good is this approximation algorithm?

LP rounding

Instead, we can use a technique that's very common in approximation algorithms.

1. Solve the linear relaxation of this integer program.
2. Round the solution $\mathbf{x}$ to an integer solution $\mathbf{x}'$: when $x_i \geq \frac{1}{2}$, set $x_i' = 1$, and when $x_i < \frac{1}{2}$, set $x_i' = 0$.

Why is $\mathbf{x}'$ still a feasible solution? Because if $x_i + x_j \geq 1$, then either $x_i \geq \frac{1}{2}$ or $x_j \geq \frac{1}{2}$ (or both), so either $x_i' = 1$ or $x_j' = 1$.

How good is this approximation algorithm? We always have $x_i' \leq 2x_i$, so the weight of $\mathbf{x}'$ is at most twice the weight of $\mathbf{x}$. Since $\mathbf{x}$ is better than the best integer solution, we have a 2-approximation.

Traveling salesman problem

Our final topic: a 2-approximation algorithm for the traveling salesman problem.

Traveling salesman problem

Our final topic: a 2-approximation algorithm for the traveling salesman problem.

- Well, actually, this is a 2-approximation algorithm for *metric* TSP: we assume that costs are symmetric ($c_{ij} = c_{ji}$) and obey the triangle inequality ($c_{ij} + c_{jk} \geq c_{ik}$).

Traveling salesman problem

Our final topic: a 2-approximation algorithm for the traveling salesman problem.

- Well, actually, this is a 2-approximation algorithm for *metric* TSP: we assume that costs are symmetric ($c_{ij} = c_{ji}$) and obey the triangle inequality ($c_{ij} + c_{jk} \geq c_{ik}$).
- It's possible to do better and get a $\frac{3}{2}$-approximation: this is Christofides's algorithm (also discussed in section 17.2 of **PS**).

  Christofides's algorithm uses similar ideas, but is a bit more complicated, so we'll skip it.

Traveling salesman problem

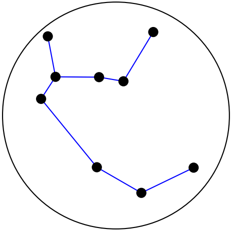
Our final topic: a 2-approximation algorithm for the traveling salesman problem.

- Well, actually, this is a 2-approximation algorithm for *metric* TSP: we assume that costs are symmetric ($c_{ij} = c_{ji}$) and obey the triangle inequality ($c_{ij} + c_{jk} \geq c_{ik}$).
- It's possible to do better and get a $\frac{3}{2}$-approximation: this is Christofides's algorithm (also discussed in section 17.2 of **PS**).

  Christofides's algorithm uses similar ideas, but is a bit more complicated, so we'll skip it.
- We'll assume without proving it that it's possible to quickly find a minimum-cost *spanning tree* (it is).

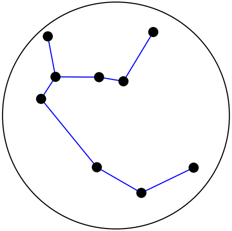
Spanning trees

Suppose we have the min-cost spanning tree.



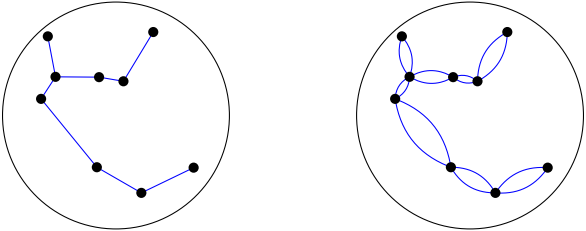
Spanning trees

Suppose we have the min-cost spanning tree. We can use it to find an almost-tour, which will visit some cities multiple times.



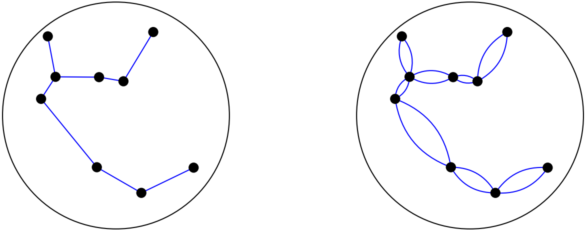
Spanning trees

Suppose we have the min-cost spanning tree. We can use it to find an almost-tour, which will visit some cities multiple times.



Spanning trees

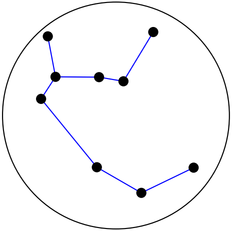
Suppose we have the min-cost spanning tree. We can use it to find an almost-tour, which will visit some cities multiple times.

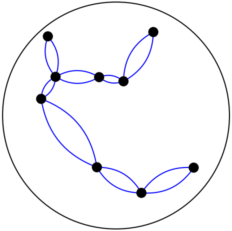


- The almost-tour has twice the cost of the spanning tree.

Spanning trees

Suppose we have the min-cost spanning tree. We can use it to find an almost-tour, which will visit some cities multiple times.

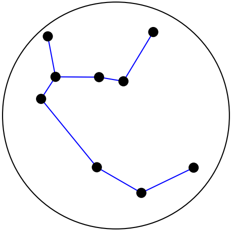


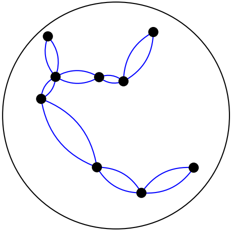


- The almost-tour has twice the cost of the spanning tree.
- The optimal tour contains a spanning tree: delete any edge and you'll have $n - 1$ edges connecting all cities.

Spanning trees

Suppose we have the min-cost spanning tree. We can use it to find an almost-tour, which will visit some cities multiple times.





- The almost-tour has twice the cost of the spanning tree.
- The optimal tour contains a spanning tree: delete any edge and you'll have $n - 1$ edges connecting all cities.
- So the optimal tour has at least the cost of the min-cost spanning tree.

Shortcuts

We have an almost-tour that's a 2-approximation:

(cost of almost-tour) $=2\cdot$(cost of tree) $\leq 2\cdot$(cost of optimal tour).

Shortcuts

We have an almost-tour that's a 2-approximation:

(cost of almost-tour) $=$ 2$\cdot$(cost of tree) $\leq$ 2$\cdot$(cost of optimal tour).

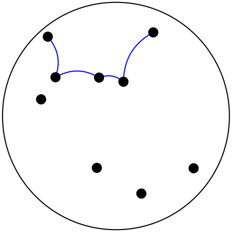
To turn the almost-tour into a tour, we take shortcuts! (This is where the triangle inequality comes in.)

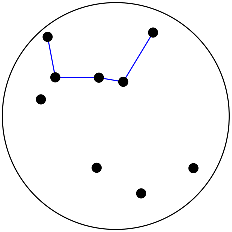
Shortcuts

We have an almost-tour that's a 2-approximation:

(cost of almost-tour) $=2\cdot$(cost of tree) $\leq 2\cdot$(cost of optimal tour).

To turn the almost-tour into a tour, we take shortcuts! (This is where the triangle inequality comes in.)



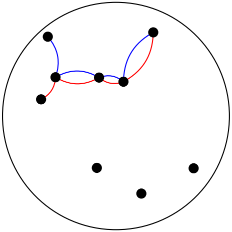


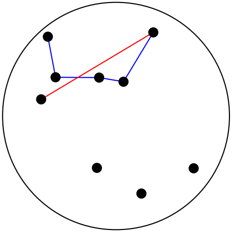
Shortcuts

We have an almost-tour that's a 2-approximation:

(cost of almost-tour) $=$ 2·(cost of tree) $\leq$ 2·(cost of optimal tour).

To turn the almost-tour into a tour, we take shortcuts! (This is where the triangle inequality comes in.)



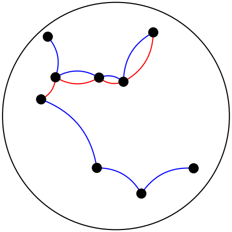


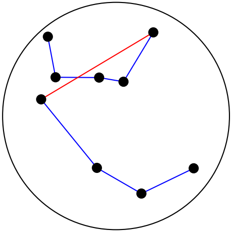
Shortcuts

We have an almost-tour that's a 2-approximation:

(cost of almost-tour) $=$ 2·(cost of tree) $\leq$ 2·(cost of optimal tour).

To turn the almost-tour into a tour, we take shortcuts! (This is where the triangle inequality comes in.)



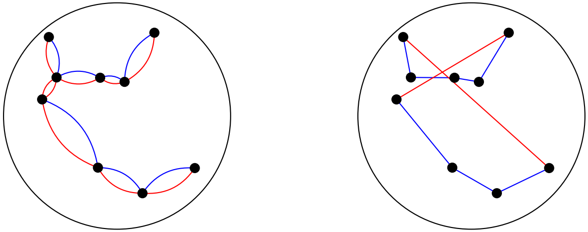


Shortcuts

We have an almost-tour that's a 2-approximation:

(cost of almost-tour) $=$ 2$\cdot$(cost of tree) $\leq$ 2$\cdot$(cost of optimal tour).

To turn the almost-tour into a tour, we take shortcuts! (This is where the triangle inequality comes in.)



Taking shortcuts only decreases the total cost. So we end up with a tour that's a 2-approximation of the optimal tour.