Outline

Approximation Algorithms

- We have a $\mathcal{NPC}$ problem Q.

Approximation Algorithms

- We have a $\mathcal{NPC}$ problem Q.
- Then it is extremely unlikely Q can be solved in polynomial time.
- What to do? Just give up?

Approximation Algorithms

- We have a $\mathcal{NPC}$ problem Q.
- Then it is extremely unlikely Q can be solved in polynomial time.
- What to do? Just give up?
- Many $\mathcal{NPC}$ problems are natural problems with important applications. We cannot afford to just give up!

Approximation Algorithms

- We have a $\mathcal{NPC}$ problem Q.
- Then it is extremely unlikely Q can be solved in polynomial time.
- What to do? Just give up?
- Many $\mathcal{NPC}$ problems are natural problems with important applications. We cannot afford to just give up!
- Many $\mathcal{NPC}$ problems are hard to solve if we insist on absolute optimal solution.

Approximation Algorithms

- We have a $\mathcal{NPC}$ problem Q.
- Then it is extremely unlikely Q can be solved in polynomial time.
- What to do? Just give up?
- Many $\mathcal{NPC}$ problems are natural problems with important applications. We cannot afford to just give up!
- Many $\mathcal{NPC}$ problems are hard to solve if we insist on absolute optimal solution.
- But if we settle for nearly optimal solutions (for example, within 50% of optimal), it might be possible to solve Q.

Approximation Algorithms

- We have a $\mathcal{NPC}$ problem Q.
- Then it is extremely unlikely Q can be solved in polynomial time.
- What to do? Just give up?
- Many $\mathcal{NPC}$ problems are natural problems with important applications. We cannot afford to just give up!
- Many $\mathcal{NPC}$ problems are hard to solve if we insist on absolute optimal solution.
- But if we settle for nearly optimal solutions (for example, within 50% of optimal), it might be possible to solve Q.
- In many applications, a nearly optimal solution might be good enough.

Approximation Algorithms

- We have a $\mathcal{NPC}$ problem Q.
- Then it is extremely unlikely Q can be solved in polynomial time.
- What to do? Just give up?
- Many $\mathcal{NPC}$ problems are natural problems with important applications. We cannot afford to just give up!
- Many $\mathcal{NPC}$ problems are hard to solve if we insist on absolute optimal solution.
- But if we settle for nearly optimal solutions (for example, within 50% of optimal), it might be possible to solve Q.
- In many applications, a nearly optimal solution might be good enough.
- This is the subject of Approximation Algorithms: Try to find solutions not too far from optimal.

Outline

- First we have to define what we mean by "nearly optimal".

Approximation Algorithms

- First we have to define what we mean by “nearly optimal”.
- If Q is a decision problem, the term approximation makes no sense: The answer is either yes or no. Nothing to be approximated.

Approximation Algorithms

- First we have to define what we mean by "nearly optimal".
- If Q is a decision problem, the term approximation makes no sense: The answer is either yes or no. Nothing to be approximated.
- So we now switch back to optimization problems.

Approximation Algorithms for Minimization Problems

Approximation Algorithm

- Q: a minimization problem.
- A: an algorithm for solving Q.
- I: an instance of Q.
- $Opt(I)$: the optimal solution of I.
- $|Opt(I)|$: the value of $Opt(I)$.
- $A(I)$: the solution found by A on input I.
- $|A(I)|$: the value of $A(I)$.
- If $\frac{|A(I)|}{|Opt(I)|} \leq r$ for some constant r and for ALL input instances I, then we say "A is an approximation algorithm for Q with performance ratio r."

Example: TSP

Example: Q is TSP

- I: an instance of TSP: Given a complete graph $G = (V, E)$ and weight function $w(*)$, find a HC C of G with minimum total length $w(C)$.

Example: TSP

Example: Q is TSP

- I: an instance of TSP: Given a complete graph $G = (V, E)$ and weight function $w(*)$, find a HC C of G with minimum total length $w(C)$.
- A: an algorithm for solving TSP. It finds a HC in G, not necessarily with minimum length.

Example: TSP

Example: Q is TSP

- I: an instance of TSP: Given a complete graph $G = (V, E)$ and weight function $w(*)$, find a HC C of G with minimum total length $w(C)$.
- A: an algorithm for solving TSP. It finds a HC in G, not necessarily with minimum length.
- $Opt(G, w)$: the optimal solution. Namely an HC of G with minimum length. (Caution: since TSP is $\mathcal{NPC}$, $Opt(G, w)$ is unknown!.)

Example: TSP

Example: Q is TSP

- I: an instance of TSP: Given a complete graph $G = (V, E)$ and weight function $w(*)$, find a HC C of G with minimum total length $w(C)$.
- A: an algorithm for solving TSP. It finds a HC in G, not necessarily with minimum length.
- $Opt(G, w)$: the optimal solution. Namely an HC of G with minimum length. (Caution: since TSP is $\mathcal{NPC}$, $Opt(G, w)$ is unknown!.)
- $|Opt(G, w)|$: the length of $Opt(G, w)$.

Example: TSP

Example: Q is TSP

- I: an instance of TSP: Given a complete graph $G = (V, E)$ and weight function $w(*)$, find a HC C of G with minimum total length $w(C)$.
- A: an algorithm for solving TSP. It finds a HC in G, not necessarily with minimum length.
- $Opt(G, w)$: the optimal solution. Namely an HC of G with minimum length. (Caution: since TSP is $\mathcal{NPC}$, $Opt(G, w)$ is unknown!.)
- $|Opt(G, w)|$: the length of $Opt(G, w)$.
- $A(G, w)$: the solution found by A. Namely, a HC of G found by A.

Example: TSP

Example: Q is TSP

- I: an instance of TSP: Given a complete graph $G = (V, E)$ and weight function $w(*)$, find a HC C of G with minimum total length $w(C)$.
- A: an algorithm for solving TSP. It finds a HC in G, not necessarily with minimum length.
- $Opt(G, w)$: the optimal solution. Namely an HC of G with minimum length. (Caution: since TSP is $\mathcal{NPC}$, $Opt(G, w)$ is unknown!.)
- $|Opt(G, w)|$: the length of $Opt(G, w)$.
- $A(G, w)$: the solution found by A. Namely, a HC of G found by A.
- $|A(G, w)|$: the length of the HC of G found by A.

Example: TSP

Example: Q is TSP

- I: an instance of TSP: Given a complete graph $G = (V, E)$ and weight function $w(*)$, find a HC C of G with minimum total length $w(C)$.
- A: an algorithm for solving TSP. It finds a HC in G, not necessarily with minimum length.
- $Opt(G, w)$: the optimal solution. Namely an HC of G with minimum length. (Caution: since TSP is $\mathcal{NPC}$, $Opt(G, w)$ is unknown!.)
- $|Opt(G, w)|$: the length of $Opt(G, w)$.
- $A(G, w)$: the solution found by A. Namely, a HC of G found by A.
- $|A(G, w)|$: the length of the HC of G found by A.
- If $\frac{|A(I)|}{|Opt(I)|} \leq 1.5$ for ALL input instances I, then we say "A is an approximation algorithm for TSP with performance ratio 1.5."

Example: TSP

Example: Q is TSP

- I: an instance of TSP: Given a complete graph $G = (V, E)$ and weight function $w(*)$, find a HC C of G with minimum total length $w(C)$.
- A: an algorithm for solving TSP. It finds a HC in G, not necessarily with minimum length.
- $Opt(G, w)$: the optimal solution. Namely an HC of G with minimum length. (Caution: since TSP is $\mathcal{NPC}$, $Opt(G, w)$ is unknown!.)
- $|Opt(G, w)|$: the length of $Opt(G, w)$.
- $A(G, w)$: the solution found by A. Namely, a HC of G found by A.
- $|A(G, w)|$: the length of the HC of G found by A.
- If $\frac{|A(I)|}{|Opt(I)|} \leq 1.5$ for ALL input instances I, then we say "A is an approximation algorithm for TSP with performance ratio 1.5."
- So, for any input G, w, A will always find a HC of G within 50% of the optimal length.

Approximation Algorithms

- Since we are dealing with minimization problem, the value of $A(I)$ is always $\geq$ the value of $Opt(I)$.

Approximation Algorithms

- Since we are dealing with minimization problem, the value of $A(I)$ is always $\geq$ the value of $Opt(I)$.
- So we always have $\frac{|A(I)|}{|Opt(I)|} \geq 1$.

Approximation Algorithms

- Since we are dealing with minimization problem, the value of $A(I)$ is always $\geq$ the value of $Opt(I)$.
- So we always have $\frac{|A(I)|}{|Opt(I)|} \geq 1$.
- Hence the performance ratio r is always ≥ 1. (If the ratio $r = 1$, then $A(I)$ is an optimal solution!).

Approximation Algorithms

- Since we are dealing with minimization problem, the value of $A(I)$ is always $\geq$ the value of $Opt(I)$.
- So we always have $\frac{|A(I)|}{|Opt(I)|} \geq 1$.
- Hence the performance ratio r is always ≥ 1. (If the ratio $r = 1$, then $A(I)$ is an optimal solution!).
- If r is closer to 1, then the solution found by A is closer to the optimal solution. (if $r = 1.01$, then A always finds a solution within 1% of optimal.)

Approximation Algorithms

- Since we are dealing with minimization problem, the value of $A(I)$ is always $\geq$ the value of $Opt(I)$.
- So we always have $\frac{|A(I)|}{|Opt(I)|} \geq 1$.
- Hence the performance ratio r is always ≥ 1. (If the ratio $r = 1$, then $A(I)$ is an optimal solution!).
- If r is closer to 1, then the solution found by A is closer to the optimal solution. (if $r = 1.01$, then A always finds a solution within 1% of optimal.)
- The goals of approximation algorithm design:
  - Reduce the performance ratio r.
  - Reduce the run time.

- The two goals often conflict:
  - If we allow $\Theta(n^2)$ time, we might find an algorithm with performance ratio $r = 1.5$.

Approximation Algorithms

- The two goals often conflict:
  - If we allow $\Theta(n^2)$ time, we might find an algorithm with performance ratio $r = 1.5$.
  - If we allow $\Theta(n^3)$ time, we might be able to find an algorithm with performance ratio $r = 1.2$.

Approximation Algorithms

- The two goals often conflict:
  - If we allow $\Theta(n^2)$ time, we might find an algorithm with performance ratio $r = 1.5$.
  - If we allow $\Theta(n^3)$ time, we might be able to find an algorithm with performance ratio $r = 1.2$.
  - If we allow $\Theta(2^n)$ time, we may find an algorithm with performance ratio $r = 1$, namely the algorithm always find an optimal solution.

Approximation Algorithms

- The two goals often conflict:
  - If we allow $\Theta(n^2)$ time, we might find an algorithm with performance ratio $r = 1.5$.
  - If we allow $\Theta(n^3)$ time, we might be able to find an algorithm with performance ratio $r = 1.2$.
  - If we allow $\Theta(2^n)$ time, we may find an algorithm with performance ratio $r = 1$, namely the algorithm always find an optimal solution.
- Depending on applications, one goal might be more important than the other goal.

Approximation Algorithms

- The two goals often conflict:
  - If we allow $\Theta(n^2)$ time, we might find an algorithm with performance ratio $r = 1.5$.
  - If we allow $\Theta(n^3)$ time, we might be able to find an algorithm with performance ratio $r = 1.2$.
  - If we allow $\Theta(2^n)$ time, we may find an algorithm with performance ratio $r = 1$, namely the algorithm always find an optimal solution.
- Depending on applications, one goal might be more important than the other goal.
- For most approximation algorithm research, the primary goal is to reduce the performance ratio as long as we stay within polynomial time.

Approximation Algorithm For Maximization Problems

Maximization Problem

- Let Q be a maximization problem (such as Maximum Independent Set problem.)

Approximation Algorithm For Maximization Problems

Maximization Problem

- Let Q be a maximization problem (such as Maximum Independent Set problem.)
- The definitions of I, $A(I)$, $Opt(I)$ etc are exactly the same as before, except:

Approximation Algorithm For Maximization Problems

Maximization Problem

- Let Q be a maximization problem (such as Maximum Independent Set problem.)
- The definitions of I, $A(I)$, $Opt(I)$ etc are exactly the same as before, except:
- If $\frac{|Opt(I)|}{|A(I)|} \leq r$ for some constant r and for ALL input instances I, then we say "A is an approximation algorithm for Q with performance ratio r."

Approximation Algorithm For Maximization Problems

Maximization Problem

- Let Q be a maximization problem (such as Maximum Independent Set problem.)
- The definitions of I, $A(I)$, $Opt(I)$ etc are exactly the same as before, except:
- If $\frac{|Opt(I)|}{|A(I)|} \leq r$ for some constant r and for ALL input instances I, then we say "A is an approximation algorithm for Q with performance ratio r."
- Since we are dealing with maximization problems, the value of $A(I)$ is always $\leq$ the value of $Opt(I)$.

Maximization Problem

- Let Q be a maximization problem (such as Maximum Independent Set problem.)
- The definitions of I, $A(I)$, $Opt(I)$ etc are exactly the same as before, except:
- If $\frac{|Opt(I)|}{|A(I)|} \leq r$ for some constant r and for ALL input instances I, then we say "A is an approximation algorithm for Q with performance ratio r."
- Since we are dealing with maximization problems, the value of $A(I)$ is always $\leq$ the value of $Opt(I)$.
- By our definition, the performance ratio r is always ≥ 1.

Approximation Algorithm For Maximization Problems

Maximization Problem

- Let Q be a maximization problem (such as Maximum Independent Set problem.)
- The definitions of I, $A(I)$, $Opt(I)$ etc are exactly the same as before, except:
- If $\frac{|Opt(I)|}{|A(I)|} \leq r$ for some constant r and for ALL input instances I, then we say "A is an approximation algorithm for Q with performance ratio r."
- Since we are dealing with maximization problems, the value of $A(I)$ is always $\leq$ the value of $Opt(I)$.
- By our definition, the performance ratio r is always ≥ 1.
- Our goal is still to reduce r.

Outline

Heuristic Algorithms

Heuristic Algorithm is another approach for solving hard optimization problems:

Heuristic Algorithms

Heuristic Algorithm is another approach for solving hard optimization problems:

- Using intuition, try to achieve the optimization goal.

Heuristic Algorithms

Heuristic Algorithm is another approach for solving hard optimization problems:

- Using intuition, try to achieve the optimization goal.
- Easy to design and understand.

Heuristic Algorithms

Heuristic Algorithm is another approach for solving hard optimization problems:

- Using intuition, try to achieve the optimization goal.
- Easy to design and understand.
- No need to prove the performance ratio.

Heuristic Algorithms

Heuristic Algorithm is another approach for solving hard optimization problems:

- Using intuition, try to achieve the optimization goal.
- Easy to design and understand.
- No need to prove the performance ratio.
- The worst drawback: You never know how far is your solution from the optimal.

Heuristic Algorithms

Heuristic Algorithm is another approach for solving hard optimization problems:

- Using intuition, try to achieve the optimization goal.
- Easy to design and understand.
- No need to prove the performance ratio.
- The worst drawback: You never know how far is your solution from the optimal.
- In some cases, the solutions produced by heuristic algorithms can be very bad. And you don't know it!

Approximation Algorithms vs Heuristic Algorithms

In contrast, the approximation algorithm:

- Must prove the performance ratio.

Approximation Algorithms vs Heuristic Algorithms

In contrast, the approximation algorithm:

- Must prove the performance ratio.
- Harder to design. (How do you compare the solution constructed by the algorithm with the optimal solution which is UNKNOWN?)

Approximation Algorithms vs Heuristic Algorithms

In contrast, the approximation algorithm:

- Must prove the performance ratio.
- Harder to design. (How do you compare the solution constructed by the algorithm with the optimal solution which is UNKNOWN?)
- Sometimes, they are counter-intuitive. (They must consider the worst cases.)

Approximation Algorithms vs Heuristic Algorithms

In contrast, the approximation algorithm:

- Must prove the performance ratio.
- Harder to design. (How do you compare the solution constructed by the algorithm with the optimal solution which is UNKNOWN?)
- Sometimes, they are counter-intuitive. (They must consider the worst cases.)
- However, because of the existence of the performance ratio, we know how the solution constructed by the algorithm compares with the optimal solution. (If $r = 1.5$, our solution is at most within 50% of optimal.)

Outline

Approximation Algorithms: Minimum Vertex Cover (MVC)

Minimum Vertex Cover (MVC)

Input: An undirected graph $G = (V, E)$.
Find: A vertex cover $C \subseteq V$ of G such that the size $|C|$ is minimum.

Approximation Algorithms: Minimum Vertex Cover (MVC)

Minimum Vertex Cover (MVC)

Input: An undirected graph $G = (V, E)$.
Find: A vertex cover $C \subseteq V$ of G such that the size $|C|$ is minimum.

- The goal of the problem is: cover all edges of G by using as few vertices as possible.

Approximation Algorithms: Minimum Vertex Cover (MVC)

Minimum Vertex Cover (MVC)

Input: An undirected graph $G = (V, E)$.
Find: A vertex cover $C \subseteq V$ of G such that the size $|C|$ is minimum.

- The goal of the problem is: cover all edges of G by using as few vertices as possible.
- So the intuition is: Include in C the vertices that cover many edges.

Approximation Algorithms: Minimum Vertex Cover (MVC)

Minimum Vertex Cover (MVC)

Input: An undirected graph $G = (V, E)$.
Find: A vertex cover $C \subseteq V$ of G such that the size $|C|$ is minimum.

- The goal of the problem is: cover all edges of G by using as few vertices as possible.
- So the intuition is: Include in C the vertices that cover many edges.
- If $deg(v) = k$, then the vertex v will cover k edges.

Approximation Algorithms: Minimum Vertex Cover (MVC)

Minimum Vertex Cover (MVC)

Input: An undirected graph $G = (V, E)$.
Find: A vertex cover $C \subseteq V$ of G such that the size $|C|$ is minimum.

- The goal of the problem is: cover all edges of G by using as few vertices as possible.
- So the intuition is: Include in C the vertices that cover many edges.
- If $deg(v) = k$, then the vertex v will cover k edges.
- Thus, the heuristic algorithm should include in C the vertices with high degrees.

MVC: Heuristic Algorithm

Heuristic-MVC(G)

1. $C \leftarrow \emptyset$
2. **while** $E \neq \emptyset$ **do**
3. pick a vertex v with the highest $deg(v)$ (break ties arbitrarily)
4. $C \leftarrow C \cup \{v\}$
5. delete all edges incident to v from E
6. **output** C

MVC: Heuristic Algorithm

Heuristic-MVC(G)

1. $C \leftarrow \emptyset$
2. **while** $E \neq \emptyset$ **do**
3. pick a vertex v with the highest $deg(v)$ (break ties arbitrarily)
4. $C \leftarrow C \cup \{v\}$
5. delete all edges incident to v from E
6. **output** C

- The algorithm is very simple

MVC: Heuristic Algorithm

Heuristic-MVC(G)

1. $C \leftarrow \emptyset$
2. **while** $E \neq \emptyset$ **do**
3. pick a vertex v with the highest $deg(v)$ (break ties arbitrarily)
4. $C \leftarrow C \cup \{v\}$
5. delete all edges incident to v from E
6. **output** C

- The algorithm is very simple
- It's also very intuitive: Since the goal is to cover all edges using as few vertices as possible, we should include the vertex that covers most edges first. What else can you do?

MVC: Heuristic Algorithm

Heuristic-MVC(G)

1. $C \leftarrow \emptyset$
2. **while** $E \neq \emptyset$ **do**
3. pick a vertex v with the highest $deg(v)$ (break ties arbitrarily)
4. $C \leftarrow C \cup \{v\}$
5. delete all edges incident to v from E
6. **output** C

- The algorithm is very simple
- It's also very intuitive: Since the goal is to cover all edges using as few vertices as possible, we should include the vertex that covers most edges first. What else can you do?
- How well it performs?

MVC: Heuristic Algorithm

Heuristic-MVC(G)

1. $C \leftarrow \emptyset$
2. **while** $E \neq \emptyset$ **do**
3. pick a vertex v with the highest $deg(v)$ (break ties arbitrarily)
4. $C \leftarrow C \cup \{v\}$
5. delete all edges incident to v from E
6. **output** C

- The algorithm is very simple
- It's also very intuitive: Since the goal is to cover all edges using as few vertices as possible, we should include the vertex that covers most edges first. What else can you do?
- How well it performs?
- It can be infinitely bad!

Heuristic Algorithm: Minimum Vertex Cover (MVC)

Fact

For any $r > 0$, there exists graphs G so that $\frac{|\text{Heuristic-MVC}(G)|}{|\text{Opt}(G)|} \geq r$.

Heuristic Algorithm: Minimum Vertex Cover (MVC)

Fact

For any $r > 0$, there exists graphs G so that $\frac{|\text{Heuristic-MVC}(G)|}{|\text{Opt}(G)|} \geq r$.

- Fix an integer k, (we will set the value of k later).

Heuristic Algorithm: Minimum Vertex Cover (MVC)

Fact

For any $r > 0$, there exists graphs G so that $\frac{|\text{Heuristic-MVC}(G)|}{|\text{Opt}(G)|} \geq r$.

- Fix an integer k, (we will set the value of k later).
- Pick an integer n so that each of $2, 3, \ldots, k$ divides n. ($n = k!$ would do. But actual n could be much smaller than $k!$.)

Heuristic Algorithm: Minimum Vertex Cover (MVC)

Fact

For any $r > 0$, there exists graphs G so that $\frac{|\text{Heuristic-MVC}(G)|}{|\text{Opt}(G)|} \geq r$.

- Fix an integer k, (we will set the value of k later).
- Pick an integer n so that each of $2, 3, \ldots, k$ divides n. ($n = k!$ would do. But actual n could be much smaller than $k!$.)
- We will construct a bipartite graph $G = (X, Y, E)$.

Heuristic Algorithm: Minimum Vertex Cover (MVC)

Fact

For any $r > 0$, there exists graphs G so that $\frac{|\text{Heuristic-MVC}(G)|}{|\text{Opt}(G)|} \geq r$.

- Fix an integer k, (we will set the value of k later).
- Pick an integer n so that each of $2, 3, \ldots, k$ divides n. ($n = k!$ would do. But actual n could be much smaller than $k!$.)
- We will construct a bipartite graph $G = (X, Y, E)$.
- $|Y| = n$.

Heuristic Algorithm: Minimum Vertex Cover (MVC)

Fact

For any $r > 0$, there exists graphs G so that $\frac{|\text{Heuristic-MVC}(G)|}{|\text{Opt}(G)|} \geq r$.

- Fix an integer k, (we will set the value of k later).
- Pick an integer n so that each of $2, 3, \ldots, k$ divides n. ($n = k!$ would do. But actual n could be much smaller than $k!$.)
- We will construct a bipartite graph $G = (X, Y, E)$.
- $|Y| = n$.
- $X = X_1 \cup X_2 \cup \cdots \cup X_k$.

Heuristic Algorithm: Minimum Vertex Cover (MVC)

Fact

For any $r > 0$, there exists graphs G so that $\frac{|\text{Heuristic-MVC}(G)|}{|\text{Opt}(G)|} \geq r$.

- Fix an integer k, (we will set the value of k later).
- Pick an integer n so that each of $2, 3, \ldots, k$ divides n. ($n = k!$ would do. But actual n could be much smaller than $k!$.)
- We will construct a bipartite graph $G = (X, Y, E)$.
- $|Y| = n$.
- $X = X_1 \cup X_2 \cup \cdots \cup X_k$.
- $|X_1| = n$. Each vertex in X_1 is adjacent to 1 vertex in Y.

Heuristic Algorithm: Minimum Vertex Cover (MVC)

Fact

For any $r > 0$, there exists graphs G so that $\frac{|\text{Heuristic-MVC}(G)|}{|\text{Opt}(G)|} \geq r$.

- Fix an integer k, (we will set the value of k later).
- Pick an integer n so that each of $2, 3, \ldots, k$ divides n. ($n = k!$ would do. But actual n could be much smaller than $k!$.)
- We will construct a bipartite graph $G = (X, Y, E)$.
- $|Y| = n$.
- $X = X_1 \cup X_2 \cup \cdots \cup X_k$.
- $|X_1| = n$. Each vertex in X_1 is adjacent to 1 vertex in Y.
- $|X_2| = n/2$. Each vertex in X_2 is adjacent to 2 vertices in Y.

Heuristic Algorithm: Minimum Vertex Cover (MVC)

Fact

For any $r > 0$, there exists graphs G so that $\frac{|\text{Heuristic-MVC}(G)|}{|\text{Opt}(G)|} \geq r$.

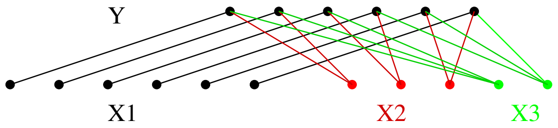
- Fix an integer k, (we will set the value of k later).
- Pick an integer n so that each of $2, 3, \ldots, k$ divides n. ($n = k!$ would do. But actual n could be much smaller than $k!$.)
- We will construct a bipartite graph $G = (X, Y, E)$.
- $|Y| = n$.
- $X = X_1 \cup X_2 \cup \cdots \cup X_k$.
- $|X_1| = n$. Each vertex in X_1 is adjacent to 1 vertex in Y.
- $|X_2| = n/2$. Each vertex in X_2 is adjacent to 2 vertices in Y.
- $|X_3| = n/3$. Each vertex in X_3 is adjacent to 3 vertices in Y.

Heuristic Algorithm: Minimum Vertex Cover (MVC)

- $\ldots$
- $|X_k| = n/k$. Each vertex in X_k is adjacent to k vertices in Y.

Heuristic Algorithm: Minimum Vertex Cover (MVC)

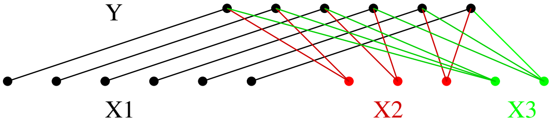
- ...
- $|X_k| = n/k$. Each vertex in X_k is adjacent to k vertices in Y.



- The above graph is an example with $k = 3$ and $n = 6$.

Heuristic Algorithm: Minimum Vertex Cover (MVC)

- $\ldots$
- $|X_k| = n/k$. Each vertex in X_k is adjacent to k vertices in Y.



- The above graph is an example with $k = 3$ and $n = 6$.
- Every vertex in X_1 has degree 1.
- Every vertex in X_2 has degree 2.
- $\ldots$
- Every vertex in X_k has degree k.
- Every vertex in Y has degree k

Heuristic Algorithm: Minimum Vertex Cover (MVC)

- The Heuristic-MVC algorithm include all vertices in X_k into C.

Heuristic Algorithm: Minimum Vertex Cover (MVC)

- The Heuristic-MVC algorithm include all vertices in X_k into C.
- Then all vertices in X_{k-1} into C.

Heuristic Algorithm: Minimum Vertex Cover (MVC)

- The Heuristic-MVC algorithm include all vertices in X_k into C.
- Then all vertices in X_{k-1} into C.
- $\cdots$
- Then all vertices in X_1 into C.

Heuristic Algorithm: Minimum Vertex Cover (MVC)

- The Heuristic-MVC algorithm include all vertices in X_k into C.
- Then all vertices in X_{k-1} into C.
- $\cdots$
- Then all vertices in X_1 into C.
- So Heuristic-MVC$(G) = X_1 \cup \cdots X_k = X$.

Heuristic Algorithm: Minimum Vertex Cover (MVC)

- The Heuristic-MVC algorithm include all vertices in X_k into C.
- Then all vertices in X_{k-1} into C.
- $\cdots$
- Then all vertices in X_1 into C.
- So Heuristic-MVC$(G) = X_1 \cup \cdots X_k = X$.
- However, Opt$(G) = Y$.

$$\begin{aligned}\frac{|\text{Heuristic-MVC}(G)|}{|\text{Opt}(G)|} &= \frac{n+n/2+n/3+\cdots+n/k}{n} \\ &= 1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{k} \sim \ln k\end{aligned}$$

Heuristic Algorithm: Minimum Vertex Cover (MVC)

- The Heuristic-MVC algorithm include all vertices in X_k into C.
- Then all vertices in X_{k-1} into C.
- $\cdots$
- Then all vertices in X_1 into C.
- So Heuristic-MVC$(G) = X_1 \cup \cdots X_k = X$.
- However, Opt$(G) = Y$.

$$\begin{aligned} \frac{|\text{Heuristic-MVC}(G)|}{|\text{Opt}(G)|} &= \frac{n+n/2+n/3+\cdots+n/k}{n} \\ &= 1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{k} \sim \ln k \end{aligned}$$

- The last sum is the Harmonic series (remember?), and it approaches to $\ln k$.

Heuristic Algorithm: Minimum Vertex Cover (MVC)

- The Heuristic-MVC algorithm include all vertices in X_k into C.
- Then all vertices in X_{k-1} into C.
- $\cdots$
- Then all vertices in X_1 into C.
- So Heuristic-MVC$(G) = X_1 \cup \cdots X_k = X$.
- However, Opt$(G) = Y$.

$$\begin{aligned}\frac{|\text{Heuristic-MVC}(G)|}{|\text{Opt}(G)|} &= \frac{n+n/2+n/3+\cdots+n/k}{n} \\ &= 1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{k} \sim \ln k\end{aligned}$$

- The last sum is the Harmonic series (remember?), and it approaches to $\ln k$.
- Since $\ln k \to \infty$ when $k \to \infty$, we can chose k so that $\ln k > r$ for any r.

Approximation Algorithm: MVC

Appr-MVC$(G = (V, E))$

1. $C \leftarrow \emptyset$ (C will be a VC of G)
2. $M \leftarrow \emptyset$ (M will be a matching of G. It is not really needed by the algorithm. However, it will help to prove the performance ratio.)

Approximation Algorithm: MVC

Appr-MVC$(G = (V, E))$

1. $C \leftarrow \emptyset$ (C will be a VC of G)
2. $M \leftarrow \emptyset$ (M will be a matching of G. It is not really needed by the algorithm. However, it will help to prove the performance ratio.)
3. **while** $E \neq \emptyset$ **do:**
4. pick any edge $e = (u, v)$ in G
5. $C \leftarrow C \cup \{u, v\}$
6. $M \leftarrow M \cup \{e\}$
7. delete all edges that are incident to u or v from E
8. **output** C

Approximation Algorithms: MVC

- The algorithm clearly takes polynomial time.

Approximation Algorithms: MVC

- The algorithm clearly takes polynomial time.
- C is a VC of G: Any edge e deleted from E at line 7, has at least one end vertex is included in C.

Approximation Algorithms: MVC

- The algorithm clearly takes polynomial time.
- C is a VC of G: Any edge e deleted from E at line 7, has at least one end vertex is included in C.
- We will show the performance ratio is $r \leq 2$.

Approximation Algorithms: MVC

- The algorithm clearly takes polynomial time.
- C is a VC of G: Any edge e deleted from E at line 7, has at least one end vertex is included in C.
- We will show the performance ratio is $r \leq 2$.
- M is a matching of G: When we add an edge e into M at line 6, all edges that share a common end vertex with e are deleted from E at line 7.

Approximation Algorithms: MVC

- The algorithm clearly takes polynomial time.
- C is a VC of G: Any edge e deleted from E at line 7, has at least one end vertex is included in C.
- We will show the performance ratio is $r \leq 2$.
- M is a matching of G: When we add an edge e into M at line 6, all edges that share a common end vertex with e are deleted from E at line 7.
- Clearly, $|C| = 2 \cdot |M|$ (whenever we include 1 edge into M, we include 2 vertices into C).

Approximation Algorithms: MVC

- The algorithm clearly takes polynomial time.
- C is a VC of G: Any edge e deleted from E at line 7, has at least one end vertex is included in C.
- We will show the performance ratio is $r \leq 2$.
- M is a matching of G: When we add an edge e into M at line 6, all edges that share a common end vertex with e are deleted from E at line 7.
- Clearly, $|C| = 2 \cdot |M|$ (whenever we include 1 edge into M, we include 2 vertices into C).
- Let Opt(G) be an optimal VC of G (which is unknown).

Approximation Algorithms: MVC

Lemma

Let $G = (V, E)$ be a graph. Let C be any vertex cover of G. Let M be any matching of G. Then $|C| \geq |M|$.

Approximation Algorithms: MVC

Lemma

Let $G = (V, E)$ be a graph. Let C be any vertex cover of G. Let M be any matching of G. Then $|C| \geq |M|$.

Proof.

- Any edge $e \in M$ must be covered by a vertex in C.
- No two edges $e_1, e_2 \in M$ can be covered by the same vertex in C, because e_1 and e_2 have no common end vertex.
- Therefore $|C| \geq |M|$.

□

Approximation Algorithms: MVC

- So $|\text{Opt}(G)| \geq |M|$.

Approximation Algorithms: MVC

- So $|\text{Opt}(G)| \geq |M|$.
- Hence: $\frac{|\text{Appr-MVC}(G)|}{|\text{Opt}(G)|} = \frac{|C|}{|\text{Opt}(G)|} \leq \frac{2 \cdot |M|}{|M|} = 2$, as to be shown.

Approximation Algorithms: MVC

- So $|\text{Opt}(G)| \geq |M|$.
- Hence: $\frac{|\text{Appr-MVC}(G)|}{|\text{Opt}(G)|} = \frac{|C|}{|\text{Opt}(G)|} \leq \frac{2 \cdot |M|}{|M|} = 2$, as to be shown.
- This algorithm is counter-intuitive: In order to cover the edge $e = (u, v)$ picked at line 4, we only need to include one end vertex (u or v) into C. But we included both of them into C at line 5.

Approximation Algorithms: MVC

- So $|\text{Opt}(G)| \geq |M|$.
- Hence: $\frac{|\text{Appr-MVC}(G)|}{|\text{Opt}(G)|} = \frac{|C|}{|\text{Opt}(G)|} \leq \frac{2\cdot|M|}{|M|} = 2$, as to be shown.
- This algorithm is counter-intuitive: In order to cover the edge $e = (u, v)$ picked at line 4, we only need to include one end vertex (u or v) into C. But we included both of them into C at line 5.
- Nevertheless, it works well.

Approximation Algorithms: MVC

- So $|\text{Opt}(G)| \geq |M|$.
- Hence: $\frac{|\text{Appr-MVC}(G)|}{|\text{Opt}(G)|} = \frac{|C|}{|\text{Opt}(G)|} \leq \frac{2 \cdot |M|}{|M|} = 2$, as to be shown.
- This algorithm is counter-intuitive: In order to cover the edge $e = (u, v)$ picked at line 4, we only need to include one end vertex (u or v) into C. But we included both of them into C at line 5.
- Nevertheless, it works well.
- This simple algorithm is the best approximation algorithm for MVC for general graphs.

Outline

Traveling Salesman Problem (TSP)

TSP

Input: A complete graph $G = (V, E)$ with edge weight function $w(*) \geq 0$.
Find: A HC C of G so that the total weight $w(C)$ is minimum.

Traveling Salesman Problem (TSP)

TSP

Input: A complete graph $G = (V, E)$ with edge weight function $w(*) \geq 0$.
Find: A HC C of G so that the total weight $w(C)$ is minimum.

We consider a special case:

Traveling Salesman Problem (TSP)

TSP

Input: A complete graph $G = (V, E)$ with edge weight function $w(*) \geq 0$.
Find: A HC C of G so that the total weight $w(C)$ is minimum.

We consider a special case:

Δ-TSP

It is the same as the general TSP problem, except that the weight function $w(*)$ must satisfy the triangle inequality: For any three vertices $x, y, z \in V$ we have:

$$w(x, y) + w(y, z) \geq w(x, z)$$

Traveling Salesman Problem (TSP)

TSP

Input: A complete graph $G = (V, E)$ with edge weight function $w(*) \geq 0$.
Find: A HC C of G so that the total weight $w(C)$ is minimum.

We consider a special case:

Δ-TSP

It is the same as the general TSP problem, except that the weight function $w(*)$ must satisfy the triangle inequality: For any three vertices $x, y, z \in V$ we have:

$$w(x, y) + w(y, z) \geq w(x, z)$$

- For most applications, we have the Δ-TSP. (Flying directly from city x to z is cheaper than flying from x to y, then from y to z.)

Traveling Salesman Problem (TSP)

TSP

Input: A complete graph $G = (V, E)$ with edge weight function $w(*) \geq 0$.
Find: A HC C of G so that the total weight $w(C)$ is minimum.

We consider a special case:

Δ-TSP

It is the same as the general TSP problem, except that the weight function $w(*)$ must satisfy the triangle inequality: For any three vertices $x, y, z \in V$ we have:

$$w(x, y) + w(y, z) \geq w(x, z)$$

- For most applications, we have the Δ-TSP. (Flying directly from city x to z is cheaper than flying from x to y, then from y to z.)
- Δ-TSP is still $\mathcal{NPC}$.

TSP: Heuristic Algorithm

Heuristic-TSP(G)

1. start at the beginning vertex v_1
2. **for** $k = 2$ **to** n **do:**
3. goto the next vertex v_k = the un-visited vertex that is closest to the current vertex v_{k-1}
4. go back from v_n to v_1

TSP: Heuristic Algorithm

Heuristic-TSP(G)

1. start at the beginning vertex v_1
2. **for** $k = 2$ **to** n **do:**
3. goto the next vertex v_k = the un-visited vertex that is closest to the current vertex v_{k-1}
4. go back from v_n to v_1

Fact

For any $r > 0$, there exists instance $I = \langle G, w(*)\rangle$ of Δ-TSP so that $\frac{|\text{Heuristic-TSP}(I)|}{|\text{Opt}(I)|} \geq r$.

TSP: Heuristic Algorithm

Heuristic-TSP(G)

1. start at the beginning vertex v_1
2. **for** $k = 2$ **to** n **do:**
3. goto the next vertex $v_k =$ the un-visited vertex that is closest to the current vertex v_{k-1}
4. go back from v_n to v_1

Fact

For any $r > 0$, there exists instance $I = \langle G, w(*)\rangle$ of Δ-TSP so that $\frac{|\text{Heuristic-TSP}(I)|}{|\text{Opt}(I)|} \geq r$.

This heuristic algorithm can perform very badly, and you would not know it!

Outline

TSP: Approximation Algorithm

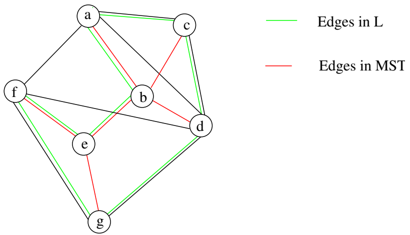
Appr-TSP(G)

1. construct a minimum spanning tree T of G.
2. let L be the list of the vertices in the preorder of T.
3. return L

TSP: Approximation Algorithm

Appr-TSP(G)

1. construct a minimum spanning tree T of G.
2. let L be the list of the vertices in the preorder of T.
3. return L



In the above example, the HC returned by the algorithm is: $abefgdca$.

TSP: Approximation Algorithm

Theorem

The performance ratio of Appr-TSP is $r = 2$.

TSP: Approximation Algorithm

Theorem

The performance ratio of Appr-TSP is $r = 2$.

- The output L can can be viewed as constructed as follows:

TSP: Approximation Algorithm

Theorem

The performance ratio of Appr-TSP is $r = 2$.

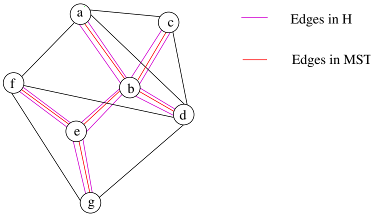
- The output L can can be viewed as constructed as follows:
- Start with the MST T.

TSP: Approximation Algorithm

Theorem

The performance ratio of Appr-TSP is $r = 2$.

- The output L can can be viewed as constructed as follows:
- Start with the MST T.
- Travel around T, using each edge of T exactly twice. This is a tour H of G. (It is not a HC of G, since some vertices are traveled more than once).
- So $w(H) = 2 \cdot w(T)$.



TSP: Approximation Algorithm

Short-Cut Operation

Suppose that a vertex v is visited by H more than once. Let $u \to v \to w$ be a section of H containing v. A short-cut at v is the operation that replaces $u \to v \to w$ by $u \to w$.

Note: A short-cut operation cannot increase the length of the tour because of the triangle inequality.

TSP: Approximation Algorithm

Short-Cut Operation

Suppose that a vertex v is visited by H more than once. Let $u \to v \to w$ be a section of H containing v. A short-cut at v is the operation that replaces $u \to v \to w$ by $u \to w$.

Note: A short-cut operation cannot increase the length of the tour because of the triangle inequality.

- L is obtained from H by a sequence of short-cut operations.

TSP: Approximation Algorithm

Short-Cut Operation

Suppose that a vertex v is visited by H more than once. Let $u \to v \to w$ be a section of H containing v. A short-cut at v is the operation that replaces $u \to v \to w$ by $u \to w$.

Note: A short-cut operation cannot increase the length of the tour because of the triangle inequality.

- L is obtained from H by a sequence of short-cut operations.
- So $w(L) \leq w(H) = 2 \cdot w(T)$.

TSP: Approximation Algorithm

Short-Cut Operation

Suppose that a vertex v is visited by H more than once. Let $u \to v \to w$ be a section of H containing v. A short-cut at v is the operation that replaces $u \to v \to w$ by $u \to w$.

Note: A short-cut operation cannot increase the length of the tour because of the triangle inequality.

- L is obtained from H by a sequence of short-cut operations.
- So $w(L) \leq w(H) = 2 \cdot w(T)$.
- Let O be an optimal HC of G. Namely $w(O)$ is the minimum among all HCs of G. O is not known. Since TSP is $\mathcal{NPC}$, we cannot find O in polynomial time.

TSP: Approximation Algorithm

Short-Cut Operation

Suppose that a vertex v is visited by H more than once. Let $u \to v \to w$ be a section of H containing v. A short-cut at v is the operation that replaces $u \to v \to w$ by $u \to w$.

Note: A short-cut operation cannot increase the length of the tour because of the triangle inequality.

- L is obtained from H by a sequence of short-cut operations.
- So $w(L) \leq w(H) = 2 \cdot w(T)$.
- Let O be an optimal HC of G. Namely $w(O)$ is the minimum among all HCs of G. O is not known. Since TSP is $\mathcal{NPC}$, we cannot find O in polynomial time.
- However, in order to prove the performance ratio, we must find a lower bound for $w(O)$.

TSP: Approximation Algorithm

- Let e be any edge on O.

TSP: Approximation Algorithm

- Let e be any edge on O.
- Consider $P = O - \{e\}$. P is a path and passing through all vertices of G. So P is a spanning tree of G.

TSP: Approximation Algorithm

- Let e be any edge on O.
- Consider $P = O - \{e\}$. P is a path and passing through all vertices of G. So P is a spanning tree of G.
- However, since T is a MST, we have $w(T) \leq w(P)$.

TSP: Approximation Algorithm

- Let e be any edge on O.
- Consider $P = O - \{e\}$. P is a path and passing through all vertices of G. So P is a spanning tree of G.
- However, since T is a MST, we have $w(T) \leq w(P)$.
- Hence: $w(O) = w(P) + w(e) \geq w(P) \geq w(T)$.

TSP: Approximation Algorithm

- Let e be any edge on O.
- Consider $P = O - \{e\}$. P is a path and passing through all vertices of G. So P is a spanning tree of G.
- However, since T is a MST, we have $w(T) \leq w(P)$.
- Hence: $w(O) = w(P) + w(e) \geq w(P) \geq w(T)$.
- Therefore:

$$\frac{w(L)}{w(O)} \leq \frac{2 \cdot w(T)}{w(T)} = 2$$

TSP: Approximation Algorithm

- Let e be any edge on O.
- Consider $P = O - \{e\}$. P is a path and passing through all vertices of G. So P is a spanning tree of G.
- However, since T is a MST, we have $w(T) \leq w(P)$.
- Hence: $w(O) = w(P) + w(e) \geq w(P) \geq w(T)$.
- Therefore:
$$\frac{w(L)}{w(O)} \leq \frac{2 \cdot w(T)}{w(T)} = 2$$
- So the performance ratio of Appr-TSP is $r = 2$.

TSP: Approximation Algorithm

- Let e be any edge on O.
- Consider $P = O - \{e\}$. P is a path and passing through all vertices of G. So P is a spanning tree of G.
- However, since T is a MST, we have $w(T) \leq w(P)$.
- Hence: $w(O) = w(P) + w(e) \geq w(P) \geq w(T)$.
- Therefore:
$$\frac{w(L)}{w(O)} \leq \frac{2 \cdot w(T)}{w(T)} = 2$$
- So the performance ratio of Appr-TSP is $r = 2$.

We can do better.

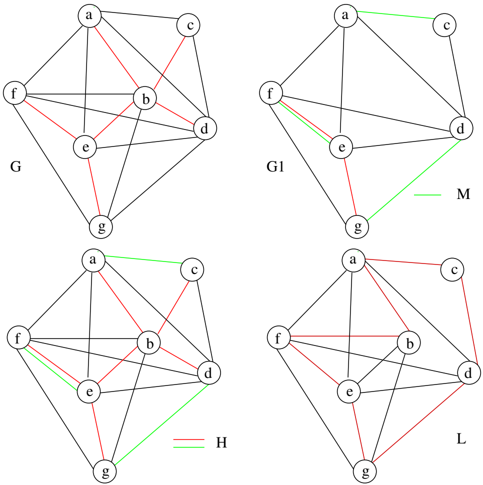
Outline

TSP: Christofides Algorithm

Christofides$(G = (V, E))$

1. construct a minimum spanning tree T of G.
2. let $V_1 \subseteq V$ be the set of vertices that have odd degrees in T.
3. let $G_1 = (V_1, E_1)$ be the subgraph of G consisting of all vertices in V_1 and all edges connecting them.
4. find a minimum weight perfect matching M in G_1. (Namely, M is a perfect matching of G_1 so that $w(M)$ is the minimum among all perfect matchings of G_1.)
5. consider the graph consisting of the edges $T \cup M$. Every vertex has even degree in this graph.
6. find an Euler tour H in $T \cup M$. (H visits each vertex of G at least once.)
7. take short-cuts on H, until it becomes a HC L of G.
8. return L

TSP: Christofides Algorithm



TSP: Christofides Algorithm

Theorem

The performance ratio of Christofides algorithm is $r = 1.5$.

TSP: Christofides Algorithm

Theorem

The performance ratio of Christofides algorithm is $r = 1.5$.

- Let O be an optimal HC of G (which is unknown.)

TSP: Christofides Algorithm

Theorem

The performance ratio of Christofides algorithm is $r = 1.5$.

- Let O be an optimal HC of G (which is unknown.)
- Same as before, we have $w(O) \geq w(T)$.

TSP: Christofides Algorithm

Theorem

The performance ratio of Christofides algorithm is $r = 1.5$.

- Let O be an optimal HC of G (which is unknown.)
- Same as before, we have $w(O) \geq w(T)$.
- Suppose we can show $w(M) \leq \frac{w(O)}{2}$ then:

$$\begin{aligned} w(L) &\leq w(H) && (L \text{ is obtained from } H \text{ by short-cuts.}) \\ &= w(T) + w(M) && (\text{because } H = T \cup M) \\ &\leq w(O) + w(O)/2 \\ &= 1.5w(O) \end{aligned}$$

Then we will have $\frac{w(L)}{w(O)} \leq 1.5$ (this is all we want to show.)

The only thing remains to be shown: $w(M) \leq w(O)/2$

TSP: Christofides Algorithm

The only thing remains to be shown: $w(M) \leq w(O)/2$

- Suppose $V_1 = \{i_1, i_2, \ldots, i_k\}$. (Note: k must be even. Why?)
- Suppose that the vertices in V_1 appear in O in this order. (If not, we can just rename them. So we can do this without loss of generality.)

TSP: Christofides Algorithm

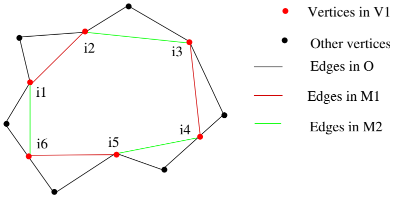
The only thing remains to be shown: $w(M) \leq w(O)/2$

- Suppose $V_1 = \{i_1, i_2, \ldots, i_k\}$. (Note: k must be even. Why?)
- Suppose that the vertices in V_1 appear in O in this order. (If not, we can just rename them. So we can do this without loss of generality.)
- Let $O' = \langle i_1 \to i_2 \to \cdots \to i_k \to i_1 \rangle$. ($O'$ is a sub-cycle obtained from O by taking short-cuts.)

TSP: Christofides Algorithm

The only thing remains to be shown: $w(M) \leq w(O)/2$

- Suppose $V_1 = \{i_1, i_2, \ldots, i_k\}$. (Note: k must be even. Why?)
- Suppose that the vertices in V_1 appear in O in this order. (If not, we can just rename them. So we can do this without loss of generality.)
- Let $O' = \langle i_1 \rightarrow i_2 \rightarrow \cdots \rightarrow i_k \rightarrow i_1 \rangle$. ($O'$ is a sub-cycle obtained from O by taking short-cuts.)
- Let $M_1 = \{(i_1, i_2), (i_3, i_4), \ldots (i_{k-1}, i_k)\}$ and $M_2 = \{(i_2, i_3), (i_4, i_6) \ldots (i_k, i_1)\}$



TSP: Christofides Algorithm

- Both M_1 and M_2 are perfect matchings of G_1.

TSP: Christofides Algorithm

- Both M_1 and M_2 are perfect matchings of G_1.
- However, M is a minimum weight perfect matching of G_1.

TSP: Christofides Algorithm

- Both M_1 and M_2 are perfect matchings of G_1.
- However, M is a minimum weight perfect matching of G_1.
- So we have: $w(M) \leq w(M_1)$ and $w(M) \leq w(M_2)$.

TSP: Christofides Algorithm

- Both M_1 and M_2 are perfect matchings of G_1.
- However, M is a minimum weight perfect matching of G_1.
- So we have: $w(M) \leq w(M_1)$ and $w(M) \leq w(M_2)$.
- Hence:

$$\begin{array}{rcll} w(O) & \geq & w(O') & \text{(because } O' \text{ is obtained from } O \text{ by short-cuts)} \\ & = & w(M_1) + w(M_2) & \text{(because } O' = M_1 \cup M_2\text{.)} \\ & \geq & w(M) + w(M) & \\ & = & 2 \cdot w(M) & \end{array}$$

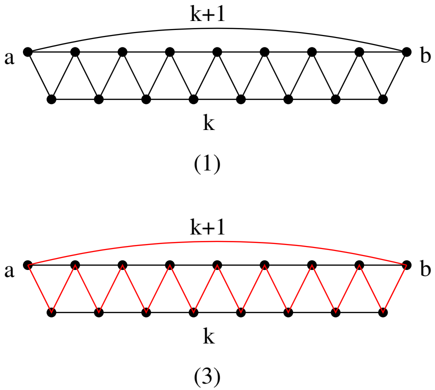
- This implies $w(M) \leq w(O)/2$, as to be shown.

TSP: Christofides Algorithm

- Christofides Algorithm was discovered in 1976. It remains the best algorithm for solving the ΔTSP problem (with the best performance ratio).

TSP: Christofides Algorithm

- Christofides Algorithm was discovered in 1976. It remains the best algorithm for solving the ΔTSP problem (with the best performance ratio).
- The following is an example that shows the ratio is actually 1.5.



TSP: Christofides Algorithm

- Fig (1) is a graph:
  - It has $k+1$ vertices on the lower line and $k+2$ vertices on the upper line.
  - Each edge shown has length 1, except the long arc has length $k+1$.
  - The length of all un-shown edges $e = (x, y)$ is the length of the shortest path between x and y.
  - It is easy to check the length function $w(*)$ defined this way satisfies the triangle inequality.

TSP: Christofides Algorithm

- Fig (1) is a graph:
  - It has $k+1$ vertices on the lower line and $k+2$ vertices on the upper line.
  - Each edge shown has length 1, except the long arc has length $k+1$.
  - The length of all un-shown edges $e = (x, y)$ is the length of the shortest path between x and y.
  - It is easy to check the length function $w(*)$ defined this way satisfies the triangle inequality.
- Figure (2) shows a MST T of G (in red color). a and b are the only vertices in T with odd degree.

TSP: Christofides Algorithm

- Fig (1) is a graph:
  - It has $k+1$ vertices on the lower line and $k+2$ vertices on the upper line.
  - Each edge shown has length 1, except the long arc has length $k+1$.
  - The length of all un-shown edges $e=(x,y)$ is the length of the shortest path between x and y.
  - It is easy to check the length function $w(*)$ defined this way satisfies the triangle inequality.
- Figure (2) shows a MST T of G (in red color). a and b are the only vertices in T with odd degree.
- Figure (3) shows the HC L constructed by Christofides algorithm. Note that $w(L)=2(k+1)+(k+1)=3k+3$.

TSP: Christofides Algorithm

- Fig (1) is a graph:
  - It has $k+1$ vertices on the lower line and $k+2$ vertices on the upper line.
  - Each edge shown has length 1, except the long arc has length $k+1$.
  - The length of all un-shown edges $e=(x,y)$ is the length of the shortest path between x and y.
  - It is easy to check the length function $w(*)$ defined this way satisfies the triangle inequality.
- Figure (2) shows a MST T of G (in red color). a and b are the only vertices in T with odd degree.
- Figure (3) shows the HC L constructed by Christofides algorithm. Note that $w(L)=2(k+1)+(k+1)=3k+3$.
- Fig (4) shows the optimal HC O. Note that $w(O)=2k+3$.

TSP: Christofides Algorithm

- Fig (1) is a graph:
  - It has $k+1$ vertices on the lower line and $k+2$ vertices on the upper line.
  - Each edge shown has length 1, except the long arc has length $k+1$.
  - The length of all un-shown edges $e=(x,y)$ is the length of the shortest path between x and y.
  - It is easy to check the length function $w(*)$ defined this way satisfies the triangle inequality.
- Figure (2) shows a MST T of G (in red color). a and b are the only vertices in T with odd degree.
- Figure (3) shows the HC L constructed by Christofides algorithm. Note that $w(L)=2(k+1)+(k+1)=3k+3$.
- Fig (4) shows the optimal HC O. Note that $w(O)=2k+3$.
- So the ratio on this graph is $\frac{3k+3}{2k+3} \rightarrow 1.5$ when $k \rightarrow \infty$.

Outline

Polynomial Time Approximation Scheme

Polynomial Time Approximation Scheme (PTAS)

Let Q be a given optimization problem. A PTAS for Q is a class of algorithms so that, for any $\epsilon > 0$, we have an algorithm A_ϵ in this class with following properties:

- The runtime of A_ϵ is polynomial in n (it may depend on ϵ in any way).
- The performance ratio of A_ϵ is at most $1 + \epsilon$.

Polynomial Time Approximation Scheme

Polynomial Time Approximation Scheme (PTAS)

Let Q be a given optimization problem. A PTAS for Q is a class of algorithms so that, for any $\epsilon > 0$, we have an algorithm A_ϵ in this class with following properties:

- The runtime of A_ϵ is polynomial in n (it may depend on ϵ in any way).
- The performance ratio of A_ϵ is at most $1 + \epsilon$.

- Example: The runtime of A_ϵ is $O(n^{\frac{1}{\epsilon}})$.

Polynomial Time Approximation Scheme

Polynomial Time Approximation Scheme (PTAS)

Let Q be a given optimization problem. A PTAS for Q is a class of algorithms so that, for any $\epsilon > 0$, we have an algorithm A_ϵ in this class with following properties:

- The runtime of A_ϵ is polynomial in n (it may depend on ϵ in any way).
- The performance ratio of A_ϵ is at most $1 + \epsilon$.

- Example: The runtime of A_ϵ is $O(n^{\frac{1}{\epsilon}})$.
- Let $\epsilon = 0.1$. Then we have an algorithm $A_{0.1}$ with performance ratio 1.1 (namely within 10% of optimal), with runtime $O(n^{10})$.

Polynomial Time Approximation Scheme

Polynomial Time Approximation Scheme (PTAS)

Let Q be a given optimization problem. A PTAS for Q is a class of algorithms so that, for any $\epsilon > 0$, we have an algorithm A_ϵ in this class with following properties:

- The runtime of A_ϵ is polynomial in n (it may depend on ϵ in any way).
- The performance ratio of A_ϵ is at most $1 + \epsilon$.

- Example: The runtime of A_ϵ is $O(n^{\frac{1}{\epsilon}})$.
- Let $\epsilon = 0.1$. Then we have an algorithm $A_{0.1}$ with performance ratio 1.1 (namely within 10% of optimal), with runtime $O(n^{10})$.
- Let $\epsilon = 0.01$. Then we have an algorithm $A_{0.01}$ with performance ratio 1.01 (namely within 1% of optimal), with runtime $O(n^{100})$.

Polynomial Time Approximation Scheme

Polynomial Time Approximation Scheme (PTAS)

Let Q be a given optimization problem. A PTAS for Q is a class of algorithms so that, for any $\epsilon > 0$, we have an algorithm A_ϵ in this class with following properties:

- The runtime of A_ϵ is polynomial in n (it may depend on ϵ in any way).
- The performance ratio of A_ϵ is at most $1 + \epsilon$.

- Example: The runtime of A_ϵ is $O(n^{\frac{1}{\epsilon}})$.
- Let $\epsilon = 0.1$. Then we have an algorithm $A_{0.1}$ with performance ratio 1.1 (namely within 10% of optimal), with runtime $O(n^{10})$.
- Let $\epsilon = 0.01$. Then we have an algorithm $A_{0.01}$ with performance ratio 1.01 (namely within 1% of optimal), with runtime $O(n^{100})$.
- Letting ϵ smaller and smaller, we can approximate the optimal solution with arbitrarily small error. But we pay a heavy price on run time.

Polynomial Time Approximation Scheme

Fully Polynomial Time Approximation Scheme (FPTAS)

Let Q be a given optimization problem. A FPTAS for Q is a class of algorithms so that, for any $\epsilon > 0$, we have an algorithm A_ϵ in this class with following properties:

- The runtime of A_ϵ is polynomial in both n and $1/\epsilon$.
- The performance ratio of A_ϵ is at most $1 + \epsilon$.

Polynomial Time Approximation Scheme

Fully Polynomial Time Approximation Scheme (FPTAS)

Let Q be a given optimization problem. A FPTAS for Q is a class of algorithms so that, for any $\epsilon > 0$, we have an algorithm A_ϵ in this class with following properties:

- The runtime of A_ϵ is polynomial in both n and $1/\epsilon$.
- The performance ratio of A_ϵ is at most $1 + \epsilon$.

- Example: The runtime of A_ϵ is $O(n^{\frac{1}{\epsilon}})$. It is not a FPTAS: The runtime is polynomial in n, but exp in $1/\epsilon$.
- Example: The runtime of A_ϵ is $O(n^3 \cdot (\frac{1}{\epsilon})^4)$. The runtime is polynomial in both n and $1/\epsilon$. This is a FPTAS.

Polynomial Time Approximation Scheme

- Baring the extremely unlikely event that $\mathcal{NP} = \mathcal{P}$, a FPTAS is the best we can hope for solving an $\mathcal{NPC}$ problem:
  - We can approximate the optimal solution within arbitrarily small error.
  - The runtime is polynomial
  - The runtime is polynomial in error rate $1/\epsilon$.

Polynomial Time Approximation Scheme

Euclidean TSP

This is a special case of the ΔTSP:

- The vertices are the points on the 2D plane (or high dimension space.)
- The weight function is $w(u, v)$ = the Euclidean distance between the point u and the point v.

Polynomial Time Approximation Scheme

Euclidean TSP

This is a special case of the ΔTSP:

- The vertices are the points on the 2D plane (or high dimension space.)
- The weight function is $w(u, v)$ = the Euclidean distance between the point u and the point v.

- Euclidean TSP is still $\mathcal{NPC}$.
- Aurora showed (1992): There is a FPTAS for solving Euclidean TSP.
- In contrast, Christofides algorithm is the best known algorithm for solving the ΔTSP.

Polynomial Time Approximation Scheme

- Can we find approximation algorithm for all $\mathcal{NPC}$ problems?
- Some are easier than others.
- Some are almost impossible.

Polynomial Time Approximation Scheme

- Can we find approximation algorithm for all $\mathcal{NPC}$ problems?
- Some are easier than others.
- Some are almost impossible.

Theorem

If there exists a polynomial time approximation algorithm for solving the Maximum Clique problem (or the Maximum Independent Set problem) for any constant performance ratio r, then $\mathcal{NP} = \mathcal{P}$.

- So unless $\mathcal{NP} = \mathcal{P}$, we cannot have an approximation algorithm for MC with performance ratio $r = 100$, $r = 10000$, any r!