Advanced Algorithms

Approximation Algorithms

Pset guidelines

- The thinking phase.

This is when you think about the problem and try to understand how to solve it. This phase terminates when you understand how to answer the problems in the problem set.

During this phase, **unless otherwise stated**, you may:

- consult any source
- Work with 2-3 friends (encouraged!)
- you may not take anything with you other than your understanding.

Pset guidelines

- The Writing phase.

This is when you write your own solution.

- you must write your solution independently and on your own.
- you may not use any written material when writing your solution.
  E.g.:
  - Do not look at a website when writing your solution,
  - Do not use notes from your discussion with friends,
  - Do not look at last year's solutions when writing your solution

Approximation Algorithms

- The fact that a problem is NP-complete doesn't mean that we cannot find an approximate solution efficiently.
- We would like to have some guarantee on the performance – how far are we from the optimal?
- What is the best we can hope for (assuming $P \neq NP$)?

Approximation Algorithms (minimization)

- In minimization problems:

  Alg is r-approximation if
  $Alg(I) \leq r \cdot OPT(I)$ for any instance I.

- In maximization problems:

  Alg is r-approximation if
  $Alg(I) \geq (1/r) \cdot OPT(I)$ for any instance I.

Approximating Vertex Cover

VertexCover(G=(V,E)):

while (E≠∅)

1. select an arbitrary edge (u,v)
2. add both u and v to the cover
3. delete all edges incident to either u or v

1. This is a legal cover (why?)
2. This is a 2-approximation; its size is at most 2 times OPT (the size of a minimum vertex cover).

Proof: Let c be the number of iterations. The VC has size 2c. The edges selected in step 1 form a matching of size c (why?). Even if we only need to cover these edges we need at least c vertices.

Approximating Vertex Cover

A more natural algorithm: select in each iteration a vertex with maximum degree, add it to the cover and remove all its adjacent edges.

Looks promising!

However, the approximation ratio of this approach is not bounded: for any r there exists a graph for which the VC chosen by the algorithm is r-times larger than the optimal VC

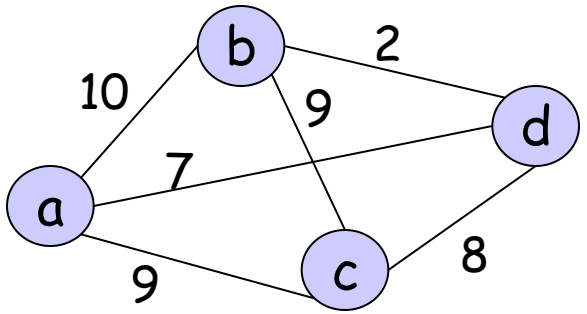
Proof: In Class

Example 2: Approximation Algorithm for Euclidean Traveling Salesman Problem

The Problem: Given n points in the plane (corresponding to locations of n cities) find a shortest traveling salesman tour – that passes exactly once in each of the points.

For each pair of cities a,b, we are given the distance dist(a,b) from a to b.

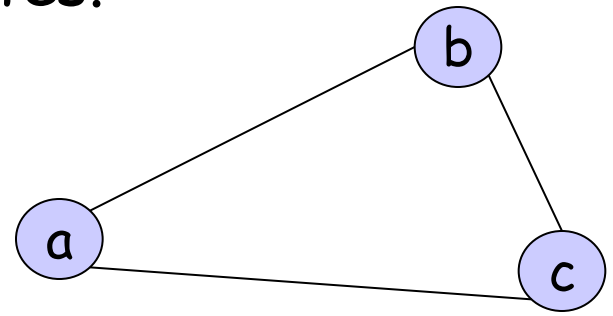
In other words, the input is a weighted complete graph.



Euclidean Traveling Salesman Problem

Distances in the plane satisfy the triangle inequality:
$\text{dist}(a,b) \leq \text{dist}(a,c) + \text{dist}(c,b)$

It means that direct routes are always shorter than indirect routes.



For this version, we will see a simple 2-approximation. we will find in poly-time a tour whose length is at most twice the optimal.

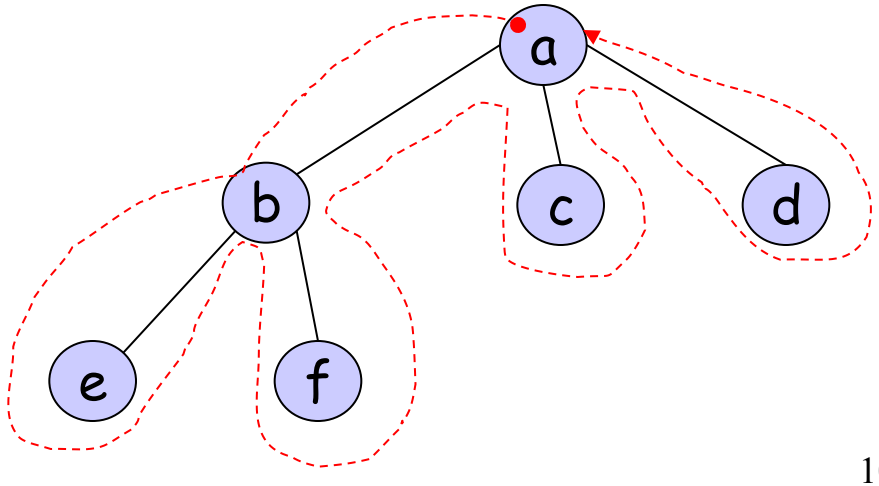
Approximating Euclidean TSP

Note: The weight of a minimum spanning tree is always less than the weight of the optimal tour.

Why? because by removing any edge from the optimal tour we get a spanning tree.

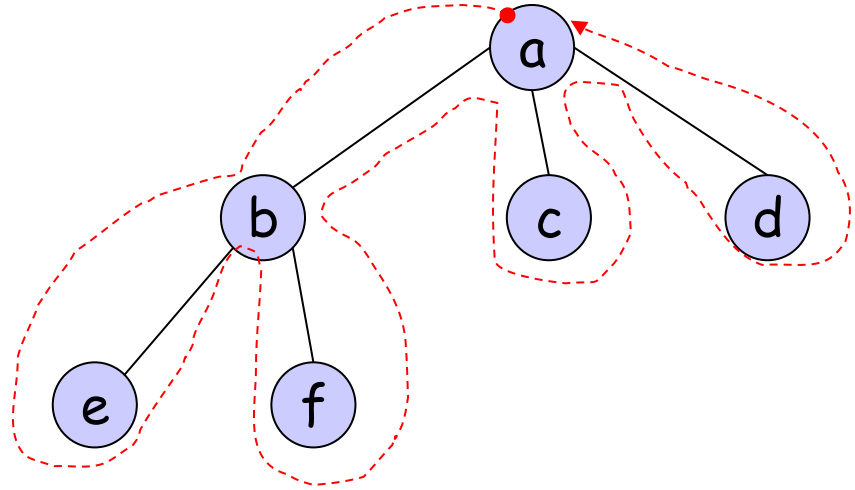
We will use this property to obtain an approximate solution.

Assume that this is our MST. Consider a DFS of this tree, let it be (w.l.o.g)
a-b-e-b-f-b-a-c-a-d-a

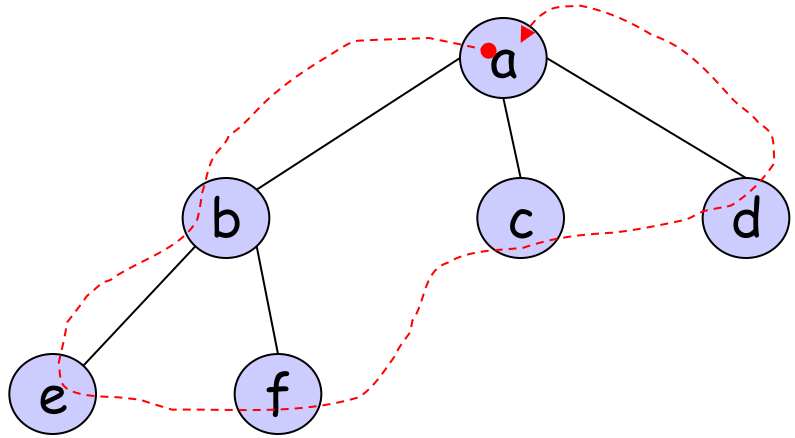


Approximating Euclidean TSP

- The DFS tour defines a spanning cycle of the graph. Each edge of the MST is used exactly twice. Hence, the length of this tour is twice the weight of the MST. However, we might visit some cities more than once.



- To get a legal solution we make shortcuts (move in the next step to the next unvisited vertex). This can only reduce the total length of the path.



Approximating Euclidean TSP

The resulting algorithm:

1. Find a minimum spanning tree of points
2. Visit the vertices in pre-order of DFS.

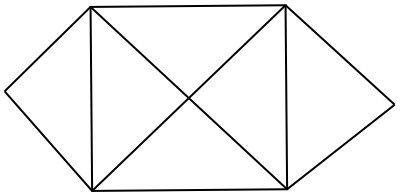
The length of the resulting tour is at most 2 times the optimal (why?)→ this is a 2-approximation algorithm.

Why 2-approximation

- The DFS tour defines a spanning cycle of the graph. Each edge of the MST is used exactly twice.
- Can we turn MST into a cycle without taking every edge of the MST twice?

Euler Tours

- Euler tours "draw without lifting your pen from the paper"
- An Euler tour in a graph is a tour of the graph that visits each edge exactly once. An Euler cycle begins and ends at the same vertex.
- Well known that an undirected graph contains an Euler cycle iff (1) it is connected and (2) each vertex has even degree.
- Easy to construct Euler tours efficiently.



Why 2-approximation

- The DFS tour defines a spanning cycle of the graph. Each edge of the MST is used exactly twice.
- Can we turn MST into a cycle without taking every edge of the MST twice?
- Taking each edge twice guarantees that the degree of every vertex is even, so there a spanning Euler cycle.
- Why double all edges? To guarantee an Euler cycle, we only need to fix vertices whose degree is odd.
- How many such vertices? An even number!
- Just pair odd-degree MST vertices in the cheapest way. Hopefully, this weighs much less than OPT. How to pair?

Maximum Matchings

- a **matching** in a graph $G=(V,E)$: a set of edges M from E such that each vertex in V is incident to at most one edge of M.
- a **maximum** matching in G: a matching of maximum cardinality
- a **minimum weight matching** in a weighted graph: a maximum matching of minimum total weight.
- Finding a minimum weight matching can be done in poly-time.

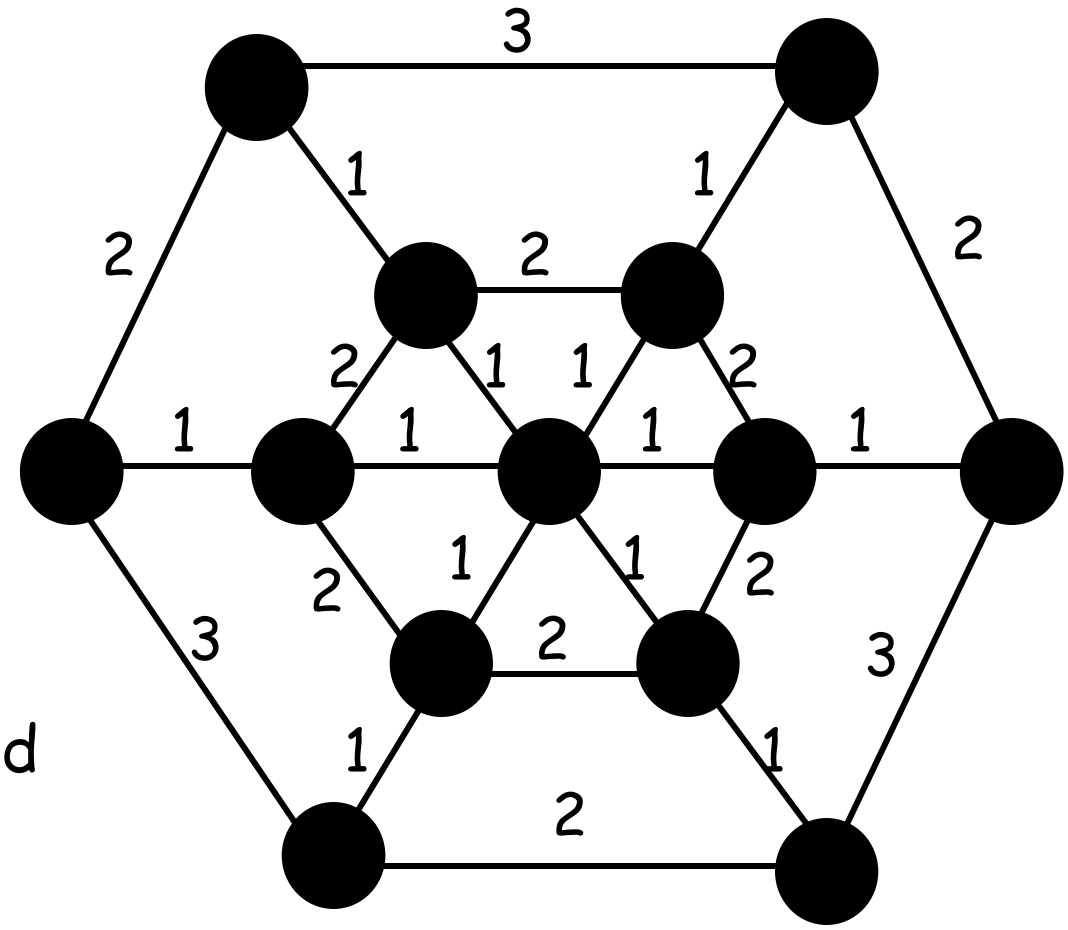
Why 2-approximation

- The DFS tour defines a spanning cycle of the graph. Each edge of the MST is used exactly twice.
- Can we construct a TSP cycle that does not use every edge of the MST twice?
- Taking each edge twice guarantees that the degree of every vertex is even, so there a spanning Euler cycle.
- Why double all edges? To guarantee an Euler cycle, we only need to fix vertices whose degree is odd.
- How many such vertices? An even number!
- Just pair odd-degree MST vertices in the lightest way. Hopefully, M weighs much less than optimal solution. How? Use a minimum weight maximum matching.

Christofides' Algorithm

1. Find an MST T.
2. Find a minimum weight matching M in the subgraph induced by odd-degree vertices in T.
3. Find an Eulerian tour of T U M.
   does one always exist?
4. Make shortcuts.

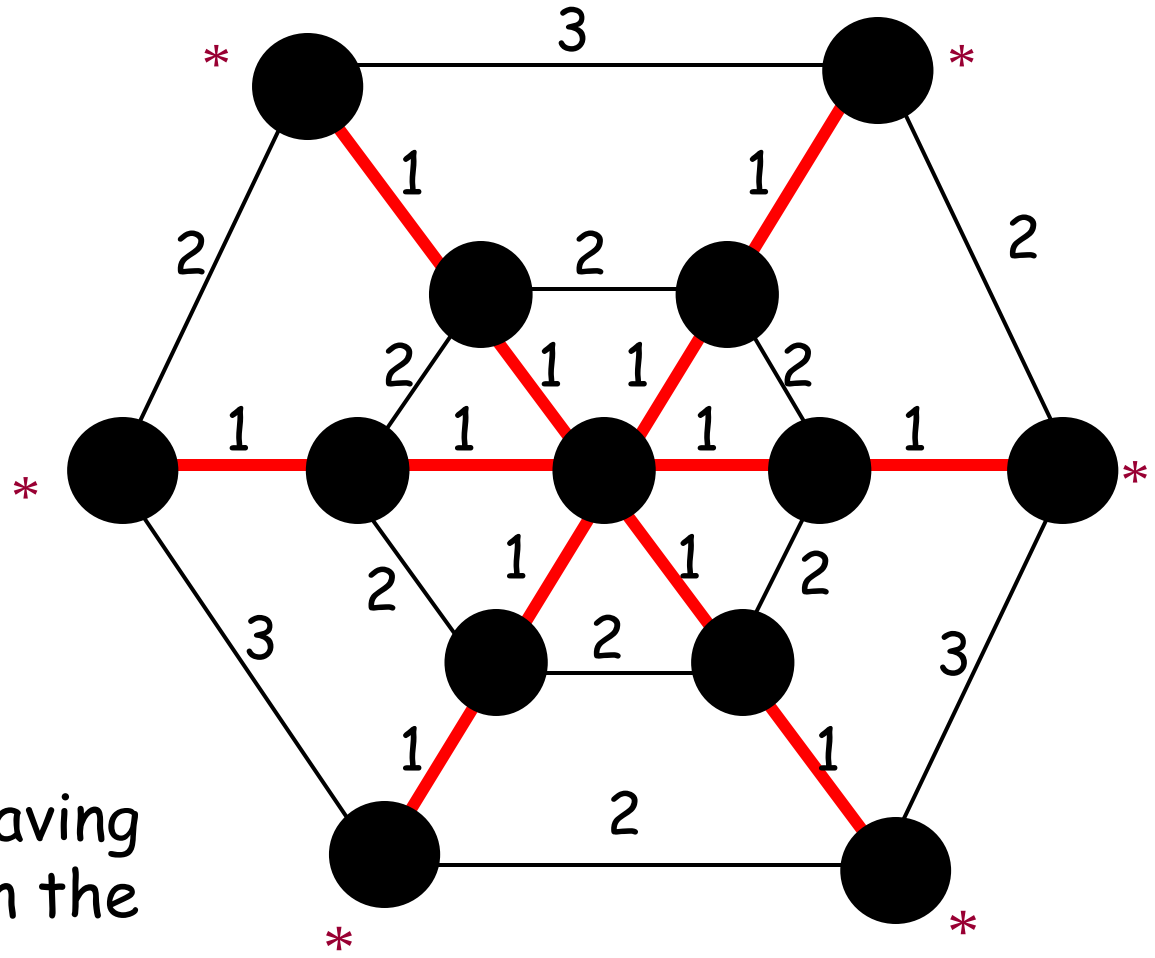
Christofides, Example (1)



The weighted graph G.

Christofides, Example (2)

An MST in G

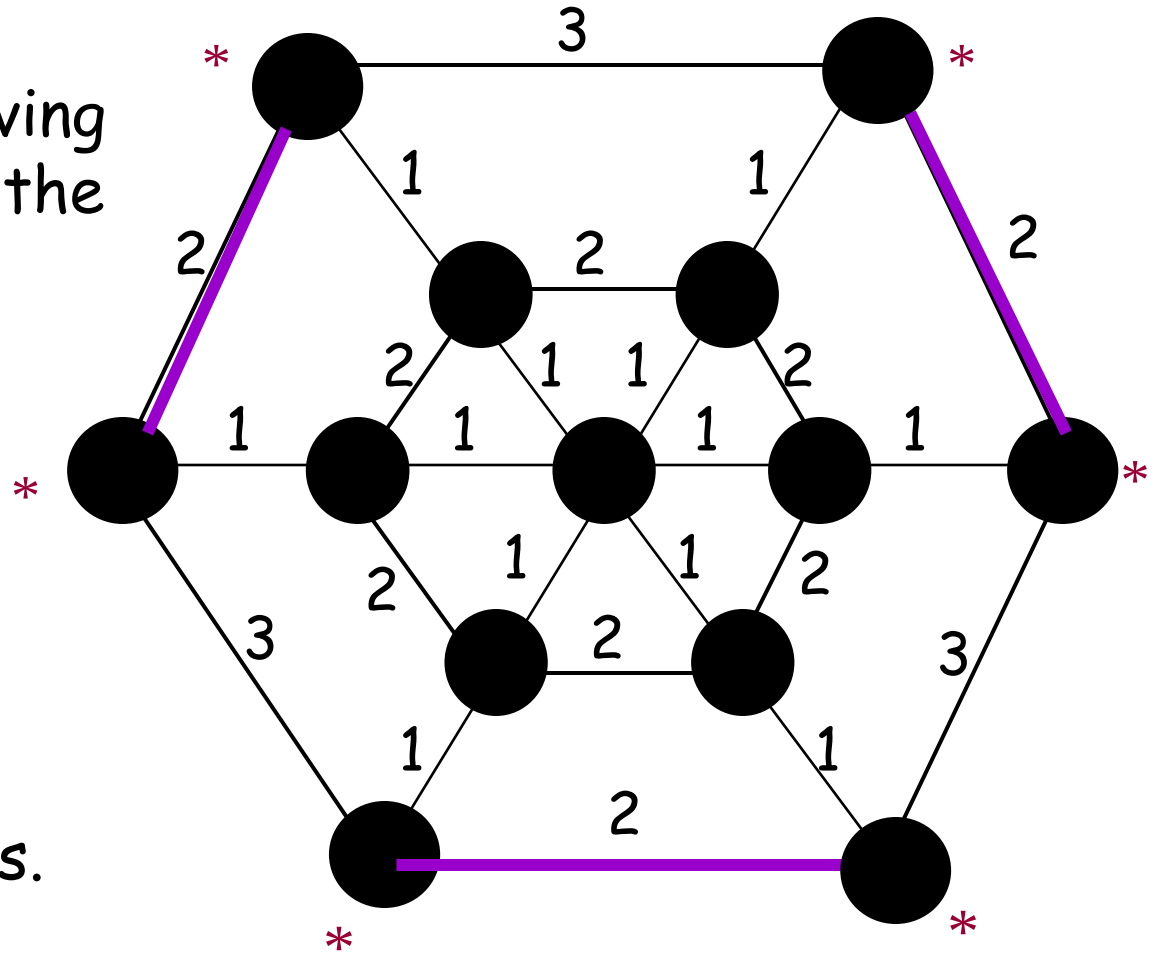


*- vertices having odd degree in the MST

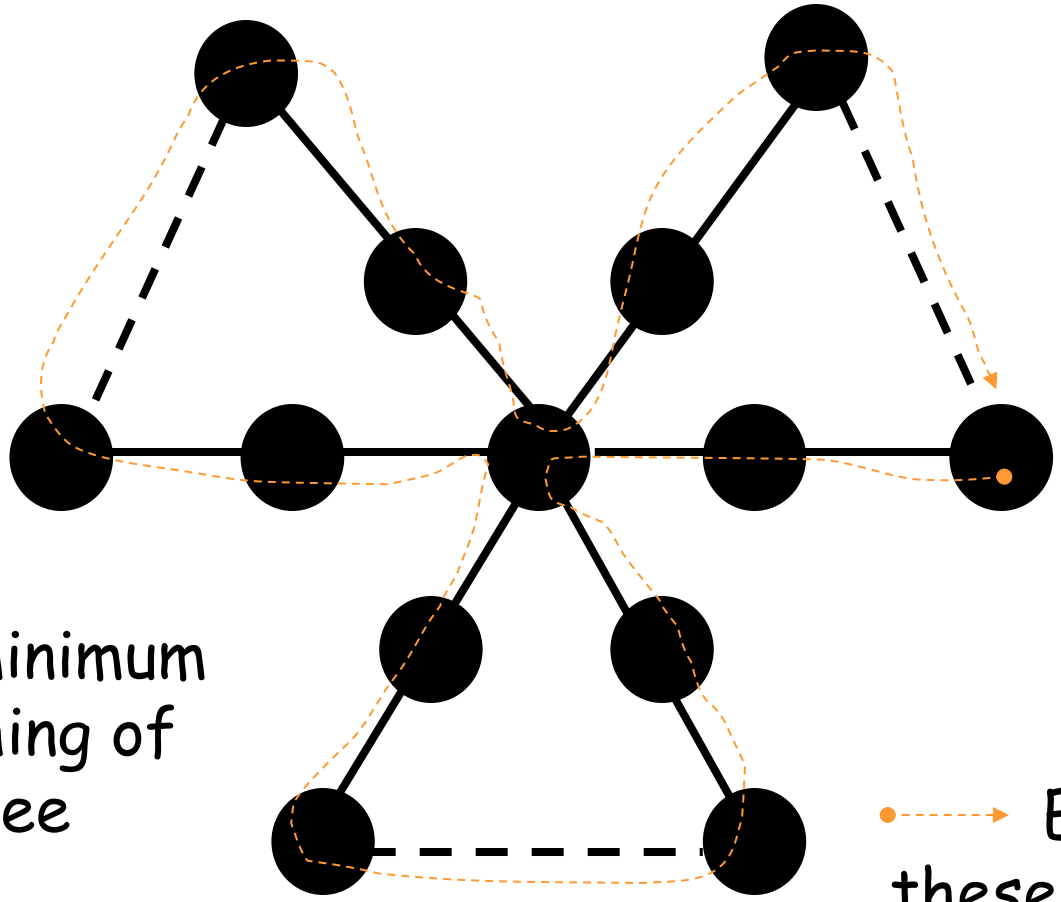
Christofides, Example (3)

*- vertices having odd degree in the MST

A minimum weight matching of the * vertices.



Christofides, Example (4)



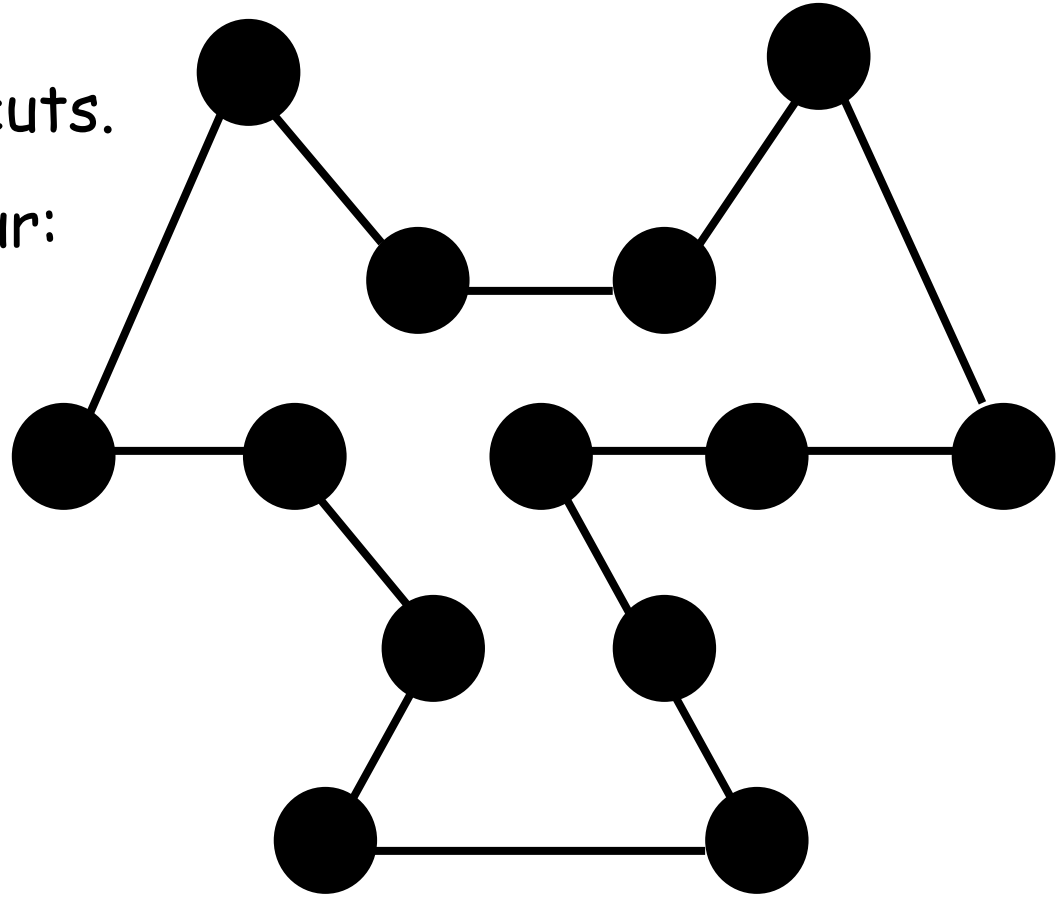
The MST + minimum weight matching of the odd-degree vertices.

Euler tour on these edges

Christofides, Example (5)

Apply shortcuts.

The final tour:



Christofides algorithm has a provable performance guarantee

Theorem: Christofides algorithm finds a tour of length at most 3/2 optimal.

Proof:

- weight of MST $\leq$ OPT
- weight of matching $\leq$ OPT/2 (next slide)
- shortcuts don't cost

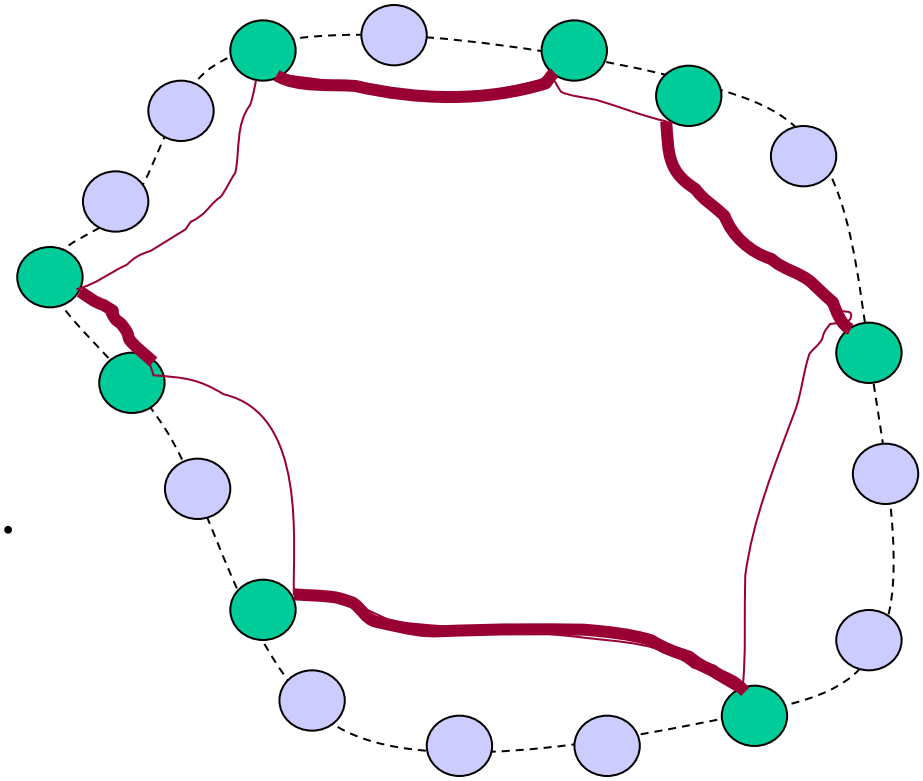
Christophedes analysis (cont')

green vertices: odd-degree vertices in MST. (their # is even)

Any Optimal TSP tour (dotted line) visits these vertices in some order.

In red: Shortcuts of optimal TSP through these vertices.

In **bold** red: The light half of the shortcuts.



A minimum weight matching is not heavier than the **bold** shortcuts $\leq \frac{1}{2}$ OPT.

Metric TSP

- Saw a 3/2-approx. algorithm for Euclidean TSP.
- Where did we use the fact that the instance is Euclidean?
- Only in triangle inequality!

- The algorithms we saw work for any metric, not just the Euclidean one.

Non-Metric TSP

When did we use the assumption that for all a,b,c $dist(a,b) \leq dist(a,c) + dist(c,b)$?

Is it really needed?

Yes – get ready for our first non-approximability result!

Theorem: For any constant c, if there is a polynomial time c-approximation algorithm for TSP then P=NP.

Non-Metric TSP

Proof: Reduction from the Hamiltonian cycle problem. Assume that TSP is c-approximable.

There exists an algorithm *A* that gets a TSP instance *G'* and returns a TS tour whose cost is at most c times the optimal.

In particular, if *G'* has a TS tour of cost n, *A* finds a tour of cost at most cn.

We will use algorithm *A* to solve the Hamiltonian-cycle decision problem.

Non-Metric TSP

Given G=(V,E) and the question "Is there a HC in G" we construct a TSP instance G', such that if there is a HC in G then there is a TS tour of cost at most n in G', and if there is no HC in G then the minimum TS tour has cost greater than cn.

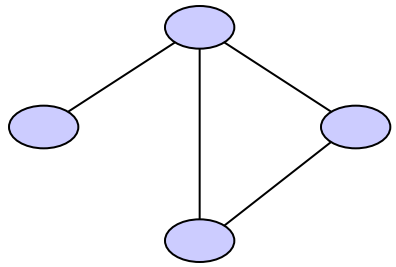
G'=(V,E') is a clique.

The weight of edges in E is 1

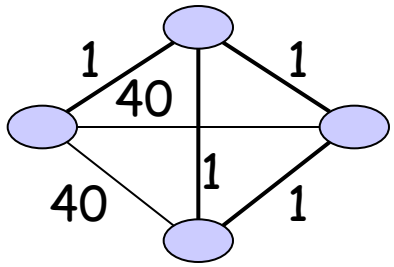
The weight of any additional edge is c·n.

- A HC in G corresponds to a TSP tour of weight n.
- Any tour that uses one or more additional edges has cost > cn.

G



G'



Example 3: Bin Packing

The input: A sequence of items (numbers), $a_1, a_2, \ldots, a_n$, such that for all i, $0 < a_i < 1$

The goal: 'pack' the items in bins of size 1. Use as few bins as possible.

Example: The input: 1/2, 1/3, 2/5, 1/6, 1/5, 2/5.

Optimal packing in two bins:

(1/2, 1/3, 1/6), (2/5, 2/5, 1/5).

Legal packing in three bins:

(1/2, 1/3), (2/5, 1/6, 1/5), (2/5)

Approximating Bin Packing

Next-fit Algorithm:

1. Open an *active* bin.
2. For all i=1,2,…,n :
   - If possible, place a_i in the current active bin;
   - Otherwise, open a new active bin and place a_i in it.

Example: The input: {0.3, 0.9, 0.2}.

Next-fit packing (three bins): (0.3), (0.9), (0.2).

Theorem: Next-fit is 2-approximation to BP

Approximating Bin Packing

Theorem: Let h be the number of bins used by Next Fit.
h < 2OPT.

Proof:

* $OPT \geq \Sigma_i a_i$ (why?).

** The sum of sizes of items in two consecutive bins is greater than 1 (otherwise, can put them together).

Suppose $h \geq 2OPT$. By **, the sum of items in the first 2OPT bins is greater than OPT, contradicting *.

Approximating Bin Packing

Is the analysis tight?

Consider an instance with 4n items {1/2, 1/2n, 1/2, 1/2n, …}.

Next-fit will put any two consecutive items in a bin.

Total number of bin used: 2n.

An optimal packing in n+1 bins: n bins, each with 1/2+1/2, one bin for the tiny items.

The ratio: $2n/(n+1) \rightarrow 2$ As n grows.

Approximating Bin Packing

First fit algorithm: place the next item in the first open bin that can accommodate it. Open a new bin only if no open bin has enough room.

Theorem: $h_{ff} \leq 1.7opt + 2$ (proof not here)

First fit Decreasing: sort the items from largest to smallest. Run FF according to the resulting order.

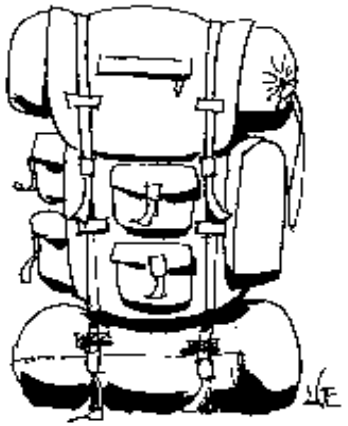
Theorem: $h_{ffd} \leq 1.222opt + 1$ (proof not here)

Example 4: The Knapsack problem

- You are about to go to a camp.
- There are many items you want to pack.
- You have one knapsack. The total weight you can carry is some fixed number W.
- Every item in your list has some weight, w_i, and some value (benefit), b_i, that measures how much you really need it.
- You need to pack the knapsack in a way that maximizes the total value of the packed items.

The Knapsack problem

Item #	Weight	Value
1	1	8
2	3	6
3	5	5
4	4	6



Max weight =8

A possible packing: Items 2 and 3. Value: 11

An optimal packing: Items 1,2,4. Value: 20

The Knapsack problem is NP-hard.

Greedy Algorithm for Knapsack

1. Consider the items in order of non-increasing b_i/w_i ratio
   $b_1/w_1 \geq b_2/w_2 \geq \ldots \geq b_n/w_n$
2. Add items to the knapsack as long as there is space.

Time Complexity:
$O(n \log n)$ (for sorting)
$O(n)$ for packing loop. → $O(n \log n)$

What's wrong with this algorithm?
Finds optimal packing if dividing items were OK.

Greedy Algorithm for Knapsack

Claim: The approximation ratio of Greedy is not bounded.

Proof: To get ratio c, consider the following instance:

There are two items:

$b_1 = 2, w_1 = 1$

$b_2 = 2c, w_2 = 2c$

The knapsack has volume $W = 2c$

Greedy packs only the first item, value = 2.

Optimal: Pack the second item, value=2c

Ratio = c.

Improved Algorithm for Knapsack

Take the maximum of Greedy and the largest value that fits by itself (the most profitable item).

Theorem: The above algorithm is a 2-approximation.

Proof: Suppose no weight of a single item exceeds W (these items can be removed in preprocessing), and that $b_1/w_1 \geq b_2/w_2 \geq \ldots \geq b_n/w_n$

Let B be the largest value, and let G be the value computed by the greedy algorithm.

Let j be the first item that the greedy algorithm rejects

Improved Algorithm for Knapsack

$ALG = \max(B, G) \geq (B + G)/2$

$G = \sum_{i=1}^{j-1} b_i$ (item j is the first to be rejected)

$B \geq b_j$ (B is the most profitable)

$G+B \geq \sum_{i=1}^{j} b_i > OPT$

Because the first j items have the largest 'profit density'

→ $ALG > OPT/2$

Is Knapsack really hard?

- The problem is NP-hard
- What if values were specified in unary? (equivalently, what if values are bounded by a polynomial in n, the number of items)
- We'll see we can solve the problem exactly!
- Such problems are called weakly NP-hard.
- A problem that is NP-hard even when all numbers are written in unary is called strongly NP-hard

Pseudo-polynomial algorithm for Knapsack

- Define $A[i,p]$ = minimum weight of a subset of items $1,\ldots,i$ whose total value is exactly p. ($A[i,p] = \infty$ if no such subset)
  $i=1,\ldots,n$; $p=1, \ldots, nB$

- Dynamic programming solution:
  - $A[1,p]$ is easy to compute for all p.
  - $A[i+1,p]$ = minimum of $A[i,p]$ and $w_{i+1} + A[i,p-b_{i+1}]$
- OPT = maximum p for which $A[n,p] \leq W$
- Running time? $O(n^2B)$ - poly(n) if B is poly(n)

Better than a const-factor

- So far we saw only approximation algorithms with a constant approximation ratio.
- For some problems we can get closer to the optimum. In fact – as close as we wish (but pay in running time).
- Welcome to APPROXIMATION SCHEMES

PTAS – Polynomial Time Approximation Schemes

A polynomial time approximation scheme is an algorithm which takes as input an additional parameter, ε.
For any fixed ε, it runs in polynomial time (in n, the size of the instance), and produces a solution whose value is at least $(1-\varepsilon)OPT$ for maximization problems
(at most $(1+\varepsilon)OPT$ for minimization problems).

Note: The dependency of the running time on ε is arbitrary (e.g. $n^{1/\varepsilon}$ is okay, since ε is considered a fixed constant)

This gives a clear trade-off between running time and quality of approximation.

PTAS for Knapsack

- OPT ≥ B.
- There are at most n items in solution
- Therefore, an optimal solution of an instance in which each value is off by at most $\varepsilon B/n$, will deviate from OPT by at most $\varepsilon B \leq \varepsilon OPT$.

Let $K = \varepsilon B/n$

Let $b_i^* = \lfloor b_i/K \rfloor$

Return optimal solution of b* instance

PTAS for Knapsack - Analysis

- b^* are integers bounded by $\lfloor B/K \rfloor = \lfloor n/\varepsilon \rfloor$
- So the running time is $O(n^3/\varepsilon)$.

- For any item i, $Kb_i^* = K\lfloor b_i/K \rfloor \geq K(b_i/K - 1) = b_i - K$
- So for any subset S,
  $K\sum_{i\in S} b_i^* \geq \sum_{i\in S}(b_i - K) \geq \sum_{i\in S} b_i - nK$

PTAS for Knapsack - Analysis

- $K \sum_{i \in S} b_i^* \geq \sum_{i \in S} (b_i - K) \geq \sum_{i \in S} b_i - nK$
- Consider the optimal subset O and the subset O* returned by the algorithm

$$\sum_{i \in O^*} b_i \geq K \sum_{i \in O^*} b_i^* \qquad \text{since } b_i \geq K b_i^*$$

$$\geq K \sum_{i \in O} b_i^* \qquad \text{since O* is optimal for } b_i^*$$

$$\geq \sum_{i \in O} b_i - nK \qquad \text{by ineq. at top}$$

$$= OPT - nK$$

$$\geq OPT - n\frac{\epsilon B}{n} \geq OPT - n\frac{\epsilon OPT}{n} = OPT(1 - \epsilon)$$

Problems that do not admit a PTAS the PCP Theorem

- MAX-3SAT: Given a 3CNF formula, determine the maximum number of clauses that can be satisfied simultaneously.
- There is a 7/8-approx. for MAX-3SAT. Can we do better?

- Theorem [PCP theorem]:
  There exists a polynomial-time algorithm that, given a 3CNF formula x, outputs a 3CNF formula f(x) such that:

  x satisfiable $\Rightarrow$ f(x) satisfiable

  x not satisfiable $\Rightarrow$ no assignment satisfies more than 90% of the clauses of f(x)

- Proof in advanced complexity classes

Problems that do not admit a PTAS the PCP Theorem

- Theorem [PCP theorem]:
  There exists a polynomial-time algorithm that, given a 3CNF formula x, outputs a 3CNF formula f(x) such that:

  x satisfiable $\Rightarrow$ f(x) satisfiable

  x not satisfieable $\Rightarrow$ no assignment satisfies more than 90% of the clauses of f(x)

- Consequently, unless P = NP, there is no 0.9-approximation algorithm for MAX-3SAT (why?)
- Unless P = NP, no PTAS for MAX-3SAT
- Basis for many hardness-of-approximation results