Lecture 20: Approximate algorithms

Outline of this Lecture

- Introduction.
- Performance ratios for approximation algorithms.
- Vertex-cover problem.
- Traveling-salesman problem.

Introduction

Many important problems are NPC which are likely to be quite hard to solve exactly. We can not just forget those problems, as they are so important. There are several things we can do:

- Try exponential time algorithm. An optimal solution is found. Not feasible if problem size is large.

- Try general optimization methods. e.g., *branch-and-bound*, *genetic algorithms*, *neural nets*. Some is hard to show how *good* they are compared with the optimal solution.

- Try approximate algorithms. Generally fast, but may not get an optimal solution. But they can be proved to be *close* to the optimal solutions.

Here we discuss some examples of approximate algorithms.

Performance ratios

Suppose we work on an optimization problem where each solution carries a *cost*. An approximate algorithm returns a legal solution, but the cost of that legal solution may not be optimal.

For example, suppose we are looking for a minimum size vertex-cover (VC). An approximate algorithm returns a VC for us, but the size (cost) may not be minimum.

Another example is we are looking for a maximum size independent set (IS). An approximate algorithm returns an IS for us, but the size (cost) may not be maximum.

Performance ratios

Let C be the cost of the solution returned by an approximate algorithm, and C^* is the cost of the optimal solution.

We say the approximate algorithm has an *approximation ratio* $\rho(n)$ for an input size n, where

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

Intuitively, the *approximation ratio* measures how bad the approximate solution is compared with the optimal solution. A large (small) *approximation ratio* means the solution is much worse than (more or less the same as) an optimal solution.

Performance ratios

Observe that $\rho(n)$ is always ≥ 1; if the ratio does not depend on n, we may just write ρ or ϵ .

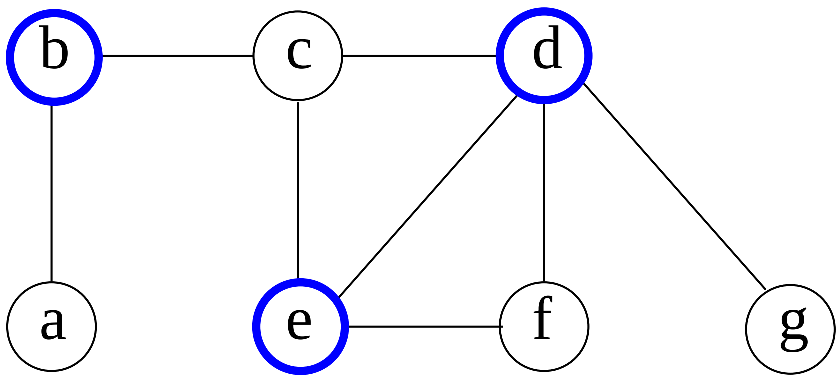
Therefore, a 1-approximation algorithm gives an optimal solution.

Some problems have polynomial-time approximation algorithms with small constant approximate ratios, while others have best known polynomial-time approximation algorithms whose approximate ratios grow with n.

Interestingly, some approximate algorithms also takes the performance ratio ϵ as input, such that the running time also depends on ϵ , e.g., $O(n^{\frac{2}{\epsilon}})$.

Vertex-cover

Vertex Cover: A vertex cover of a graph G is a set of vertices such that every edge in G is incident to at least one of these vertices.



The decision vertex-cover problem was proven NPC.

Now, we want to solve the optimal version of vertex-cover problem, i.e., we want to find a minimum size vertex cover of a given graph. We call such vertex cover an optimal vertex cover C^* .

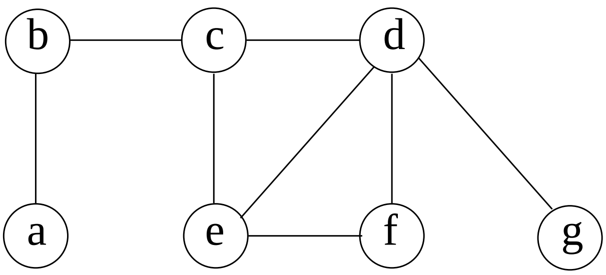
Vertex-cover

An exact polynomial time algorithm to find an optimal vertex cover C^* depends on your future hard work! So we seek help from the following approximate algorithm:

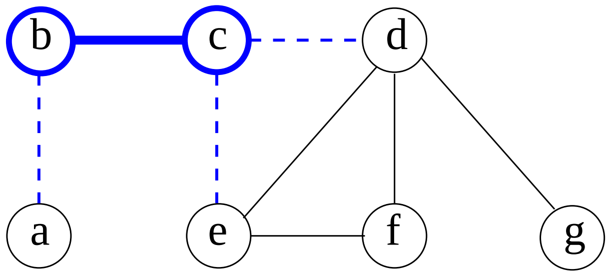
```
Approx-Vertex-Cover(G=(V, E)) {
 C = empty-set;
 E'= E;
 while E' is not empty do {
  let (u, v) be any edge in E'; (*)
  add u and v to C;
  remove from E' all edges incident to
  u or v;
 }
 return C;
}
```

The idea is to take an edge (u, v) one by one, put BOTH vertex to C, and remove all the edges incident to u or v. We carry on until all edges have been removed. Obviously, C is a VC. But how good is C ?

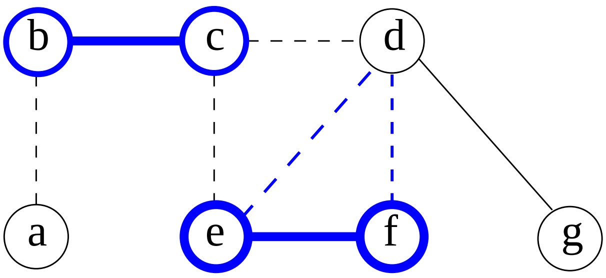
Vertex-cover



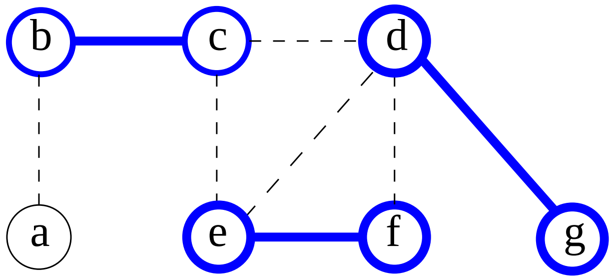
(1)



(2)



(3)



(4)

VC={b, c, d, e, f, g}

Approximate vertex-cover

Claim : `Approx-Vertex-Cover` is a 2-approximation algorithm, i.e.,

$$\frac{|C|}{|C^*|} \leq 2.$$

It means the number of vertices in C returned by `Approx-Vertex-Cover` guarantees to be at most twice of the optimal value.

Approximate vertex-cover

Proof :

1. Let A be the edge set selected by line `(*)`. Observe that $|C| = 2|A|$.

2. Observe the that the edges in A does not have any common vertex between them. It means for $e = (x, y) \in A$, either x or y must be selected to the optimal vertex cover C^* . It follows $|C^*| \geq |A|$.

3. Now we have

$$\begin{aligned} \frac{|C|}{2} &= |A| \leq |C^*| \\ \Rightarrow \quad \frac{|C|}{|C^*|} &\leq 2. \end{aligned}$$

Traveling-salesman problem

Imagine your are a salesman, and you need to visit n cities. You want to start a *tour* at a city and visit every city *exactly one time*, and finish the tour at the city from where you start. There is a non-negative cost $c(i, j)$ to travel from city i to city j. The goal is to find a tour (which is a Hamiltonian cycle) of minimum cost. We assume every two cities are connected. Such problem is called *Traveling-salesman problem (TSP)*.

We can model the cities as a complete graph of n vertices, where each vertex represents a city.

It can be shown that TSP is NPC. (CLRS pp.1012-1013).

Traveling-salesman problem

An exact polynomial time algorithm to find an optimal tour H^* depends on your future hard work!

If we assume the cost function c satisfies the *triangle inequality*, then we can use the following approximate algorithm.

Triangle inequality: Let u, v, w be any three vertices, we have

$$c(u, w) \leq c(u, v) + c(v, w).$$

Intuitively, it is always better (or not worse) to make a short-cut.

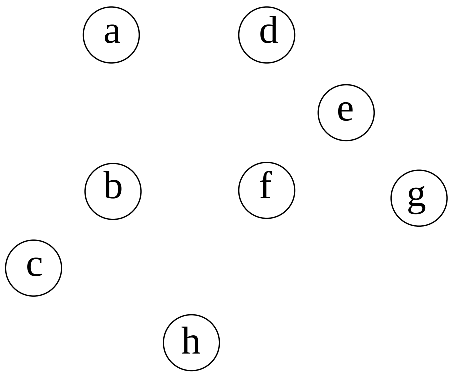
Traveling-salesman problem

One important observation to develop an approximate solution is if we remove an edge from H^* , the tour becomes a *spanning tree* .

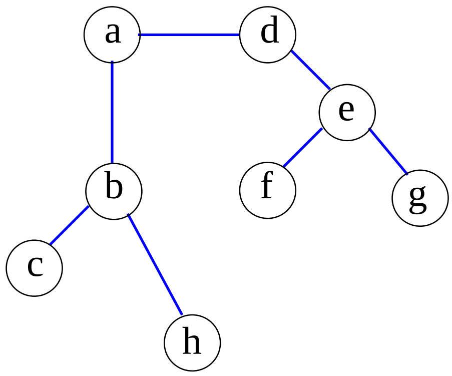
But we know how to compute a *spanning tree* of minimum cost (MST). Given a MST T, how can we convert it to a tour H ?

```
Approx-TSP(G=(V, E)) {
 compute a MST T of G;
 select any vertex r be the root of
 the tree;
 let L be the list of vertices
 visited in a preorder tree walk
 of T;
 return the hamiltonian cycle H that
 visits the vertices in the order L;
}
```

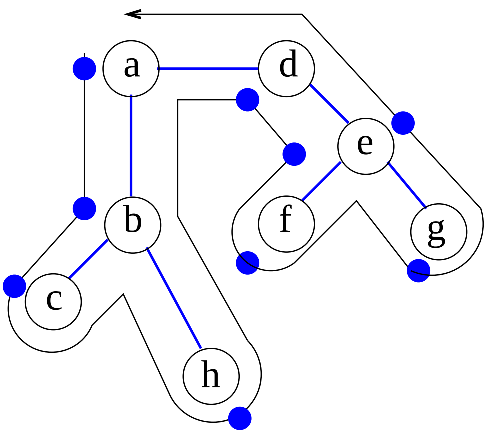
Traveling-salesman problem



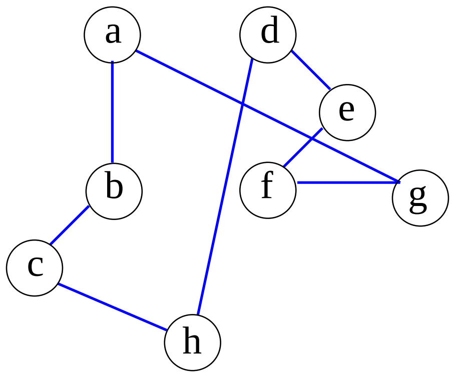
(1)A given set of points.



(2) MST T.



(3) Full tree walk on T.



(4) A preorder sequence gives a tour H.

Traveling-salesman problem

Intuitively, `Approx-TSP` first makes a full walk of MST T, which visits every edge exactly two times. To create a hamiltonian cycle from the full walk, it by-passes some vertices (which corresponds to making a shortcut).

Claim : `Approx-TSP` is a 2-approximation algorithm, i.e.,

$$\frac{c(H)}{c(H^*)} \leq 2.$$

Traveling-salesman problem

Proof :

1. Observe that if we remove any edge from H^* , then it becomes to a spanning tree, hence we have
$$c(T) \le c(H^*).$$

2. The cost of a full walk on T is $2c(T)$; since H makes a short-cut on the full walk, by *triangle inequality*, we have
$$c(H) \le 2c(T).$$

3. Combining, we have
$$\frac{c(H)}{2} \le c(T) \le c(H^*)$$
$$\Rightarrow \quad \frac{c(H)}{c(H^*)} \le 2.$$