Approximate Algorithms

Introduction:

An Approximate Algorithm is a way of approach **NP-COMPLETENESS** for the optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at the most polynomial time. Such algorithms are called approximation algorithm or heuristic algorithm.

- For the traveling salesperson problem, the optimization problem is to find the shortest cycle, and the approximation problem is to find a short cycle.
- For the vertex cover problem, the optimization problem is to find the vertex cover with fewest vertices, and the approximation problem is to find the vertex cover with few vertices.

Performance Ratios

Suppose we work on an optimization problem where every 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 considering for a **minimum size vertex-cover (VC)**. An approximate algorithm returns a VC for us, but the size (cost) may not be minimized.

Another Example is we are considering for a **maximum size Independent set (IS)**. An approximate Algorithm returns an IS for us, but the size (cost) may not be maximum. 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 approximate ratio P (n) for an input size n, where

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

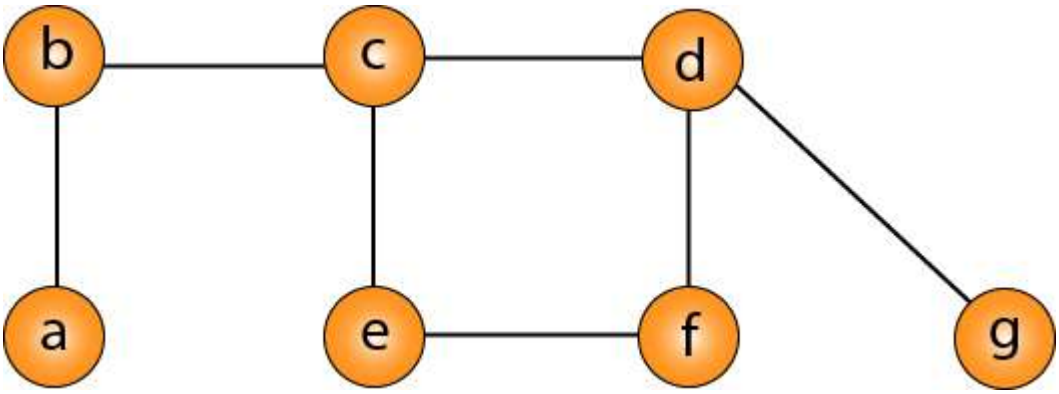
Intuitively, the approximation ratio measures how bad the approximate solution is distinguished with the optimal solution. A large (small) approximation ratio measures the solution is much worse than (more or less the same as) an optimal solution.

Observe that P (n) is always ≥ 1, if the ratio does not depend on n, we may write P. Therefore, a 1-approximation algorithm gives an optimal solution. Some problems have polynomial-time approximation algorithm with small constant approximate ratios, while others have best-known polynomial time approximation algorithms whose approximate ratios grow with n.

Vertex Cover

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

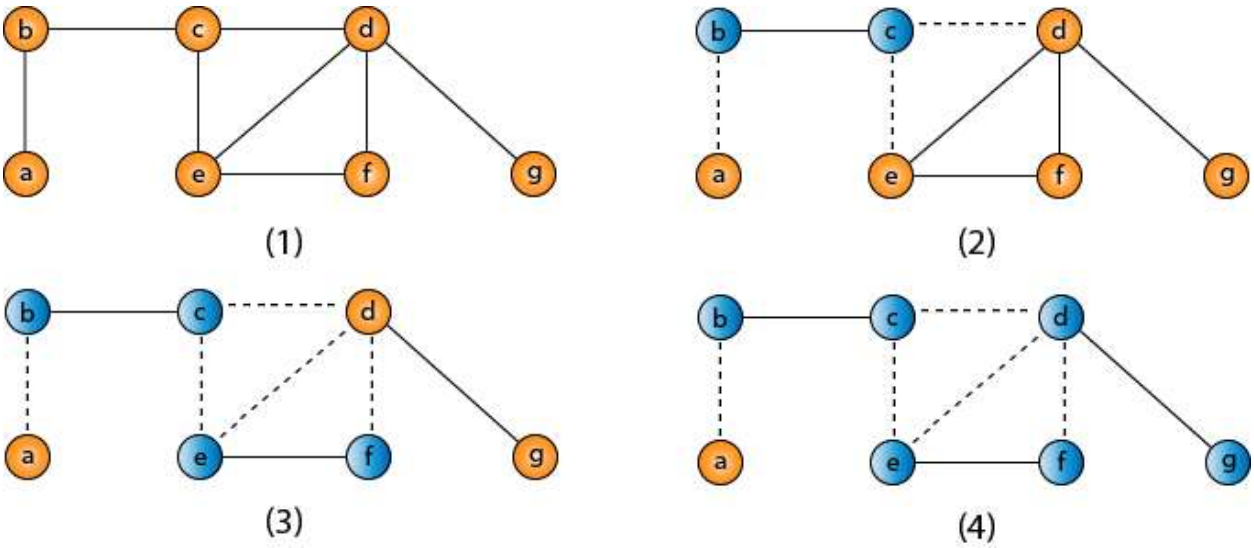
The decision vertex-cover problem was proven NPC. Now, we want to solve the optimal version of the vertex cover problem, i.e., we want to find a minimum size vertex cover of a given graph. We call such vertex cover an optimal vertex cover C*.



An approximate algorithm for vertex cover:

```
1.  Approx-Vertex-Cover (G = (V, E))
2.  {
3.        C = empty-set;
4.     E'= E;
5.     While E' is not empty do
6.       {
7.     Let (u, v) be any edge in E': (*)
8.     Add u and v to C;
9.     Remove from E' all edges incident to
10.       u or v;
11.     }
12.    Return C;
13. }
```

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



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

Traveling-salesman Problem

In the traveling salesman Problem, a salesman must visits n cities. We can say that salesman wishes to make a tour or Hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is a non-negative cost c (i, j) to travel from the city i to city j. The goal is to find a tour of minimum cost. We assume that every two cities are connected. Such problems are 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.

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

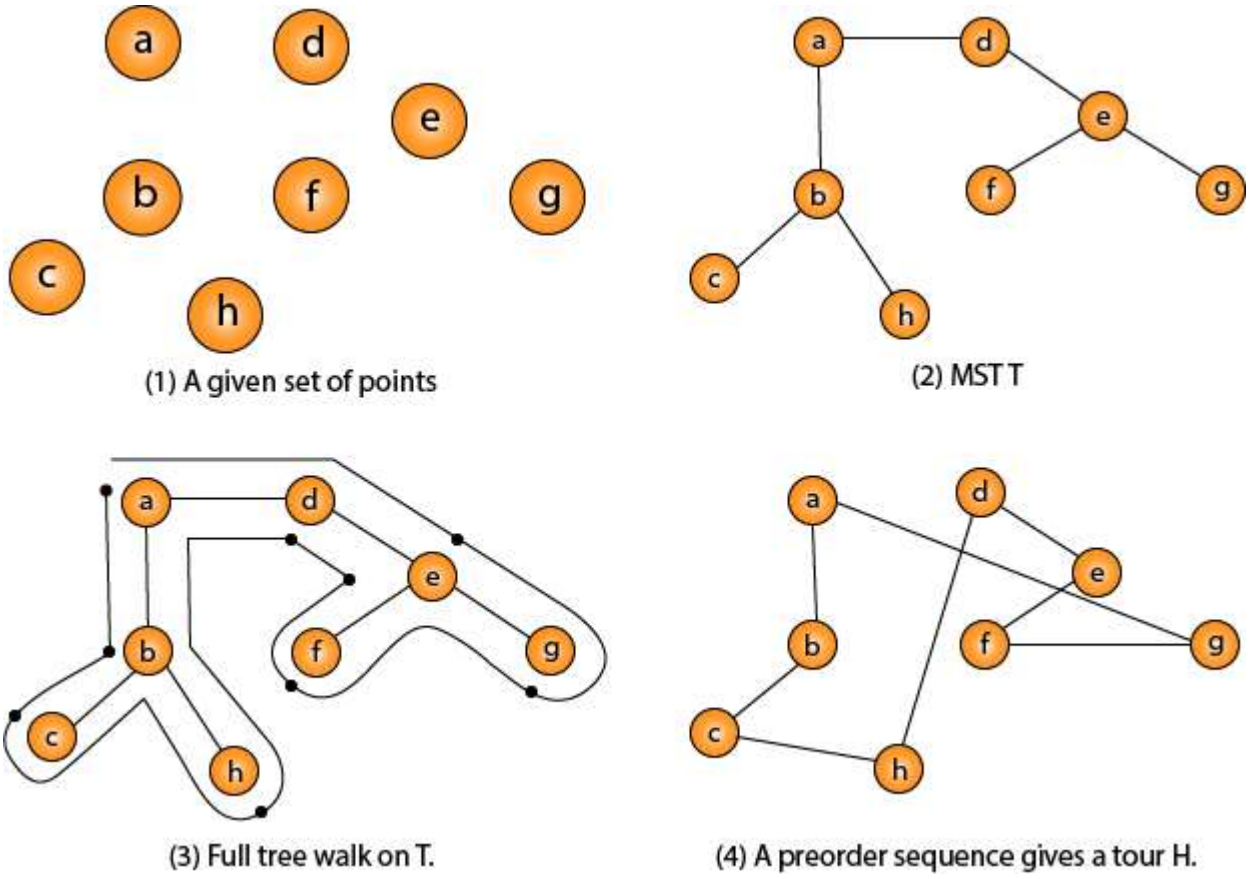
$$c(u, w) \leq c(u, v) + c(v, w)$$

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

1. Approx-TSP (G= (V, E))
2. {

3. Compute a MST T of G;
4. Select any vertex r is the root of the tree;
5. Let L be the list of vertices visited in a preorder tree walk of T;
6. Return the Hamiltonian cycle H that visits the vertices in the order L;
7. }

Traveling-salesman Problem



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

Set Cover

Definition

An Instance (X, F) of the set-covering problem consists of a finite set X and a family F of subset of X, such that every elemennt of X belongs to at least one subset of F :

$$X = \bigcup_{S \in F} S$$

We say that a subset $S \in F$ covers all elements in X. Our goal is to find a minimum size subset $C \subseteq F$ whose members cover all of X.

$$X = \bigcup_{S \in C} S \tag{1}$$

The cost of the set-covering is the size of C, which defines as the number of sets it contains, and we want $|C|$ to be minimum. An example of set-covering is shown in Figure 1. In this Figure, the minimum size set cover is $C = \{T_3, T_4, T_5\}$ and it has the size of 3.

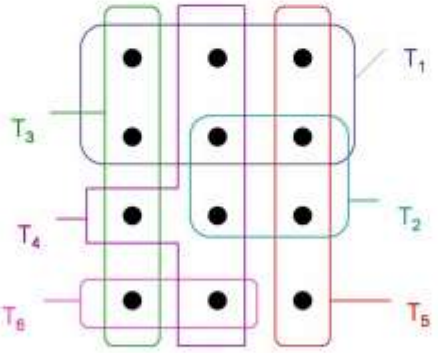


Figure 1: An instance (X, F) of set-covering problem. Here, X consists of 12 vertices and $F = \{T_1, T_2, T_3, T_4, T_5, T_6\}$. A minimum-size set cover is $C = \{S_3, S_4, S_5\}$. The greedy algorithm produces set cover of size 4 by selecting the sets S_1, S_4, S_5, S_3 in order.

Set-covering problem is a model for many resource covering problems. As mentioned earlier in the previous lecture, set-covering is an NP-Hard problem. We will now examine a greedy algorithm that gives logarithmic approximation solution.

A Greedy Approximation Algorithm

IDEA: At each stage, the greedy algorithm picks the set $S \in F$ that covers the greatest numbers of elements not yet covered.
For the example in Figure 1, the greedy algorithm will first pick T_1 because T_1 covers the maximum number of uncovered elements, which is 6. Then, it will pick T_4 since it covers maximum number uncovered elements, which is 3, leaving 3 more elements uncovered. Then it will select T_5 and T_3 , which cover 2 and 1 uncovered elements, respectively. At this point, every element in X will be covered.
By greedy algorithm, $C = \{T_1, T_4, T_5, T_3\} \Rightarrow cost = |C| = 4$

Where Optimum solution, $C = \{T_3, T_4, T_5\} \Rightarrow cost = |C| = 3$

Algorithm 1: GREEDY-SET-COVER (X, F)

```
1 U ← X
2 C ← ∅
3 While U ≠ 0
4    do select an S ∈ F that maximizes |S ∩ U|
5       U ← U − S
6       C ← C ∪ {S}
7 return C
```

The description of this algorithm are as following. First, start with an empty set C. Let C contains a cover being constructed. Let U contain, at each stage, the set of remaining uncovered elements. While there exists remaining uncovered elements, choose the set S from F that covers as many uncovered elements as possible, put that set in C and remove these covered elements from U. When all element are covered, C contains a subfamily of F that covers X and the algorithm terminates.

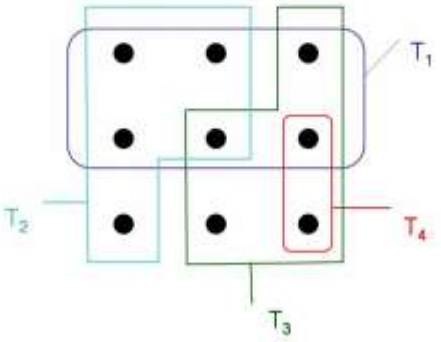


Figure 2: Another instance (X, F) of set-covering problem. Here, X consists of 9 vertices and $F = \{T_1, T_2, T_3, T_4\}$. The greedy algorithm produces set cover of size 3 by selecting the sets T_1, T_3 and T_2 in order.