CST207
DESIGN AND ANALYSIS OF ALGORITHMS

Lecture 12: Approximation Algorithms

Lecturer: Dr. Yang Lu

Email: luyang@xmu.edu.my

Office: A1-432

Office hour: 2pm-4pm Mon & Thur

Why Need Approximation Algorithms?

- Many problems are NP-complete, but are too important to give up merely because obtaining an optimal solution is intractable.
- If a problem is NP-complete, we are unlikely to find a polynomial-time algorithm for solving it exactly, but even so, there may be hope.

Why Need Approximation Algorithms?

- There are at least three approaches to getting around NP-completeness:
  - **Approach 1:** If the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory.
  - **Approach 2:** We may be able to isolate important special cases that are solvable in polynomial time.
  - **Approach 3:** It may still be possible to find *near-optimal* solutions in polynomial time (either in the worst case or on average).
- In practice, near-optimality is often good enough. An algorithm that returns near-optimal solutions is called an *approximation algorithm*.

Approximation Algorithms

- For example, if you only have 5 days to prepare final exams for 5 courses, you have two strategies:
  - Spend 4 days to make 1 course get A and 1 day to make all the other 4 courses get C.
  - Evenly spend 5 days to 5 courses to make each course get B.
- It is same for engineering, sometimes we don't have to pursue perfect solution for a problem due to high cost, because the resource (e.g. hardware, computational time, labour) is limited.
  - A relative good result is enough and we can focus on something else such that the total return is maximized. (GPA for 5 Bs is higher than that of 1 A and 4 Cs).
- In economics, it is call profit maximization, which is achieved when marginal revenue equals marginal cost.

Approximation Ratio

- The optimization problem may be either a maximization or a minimization problem.
- We say that an algorithm for a problem has an *approximation ratio* of $\rho(n)$ if, for **any** input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

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

- We also call an algorithm that achieves an approximation ratio of $\rho(n)$ a $\rho(n)$ *approximation algorithm*.

Approximation Ratio

- For a minimization problems, we have $0 < C^* \leq C$.
- For a maximization problems, we have $0 < C \leq C^*$.
- The approximation ratio of an approximation algorithm is never less than 1.
- The smaller the approximation ratio, the better the approximation algorithm.
  - A 1-approximation algorithm produces an optimal solution.

Approximation Algorithms

Now, we look at four problems that can be solved by approximation algorithms:

- The vertex-cover problem
- The set-covering problem
- The travel-salesman problem
- MAX-CNF satisfiability problem

THE VERTEX-COVER PROBLEM

The Vertex-Cover Problem

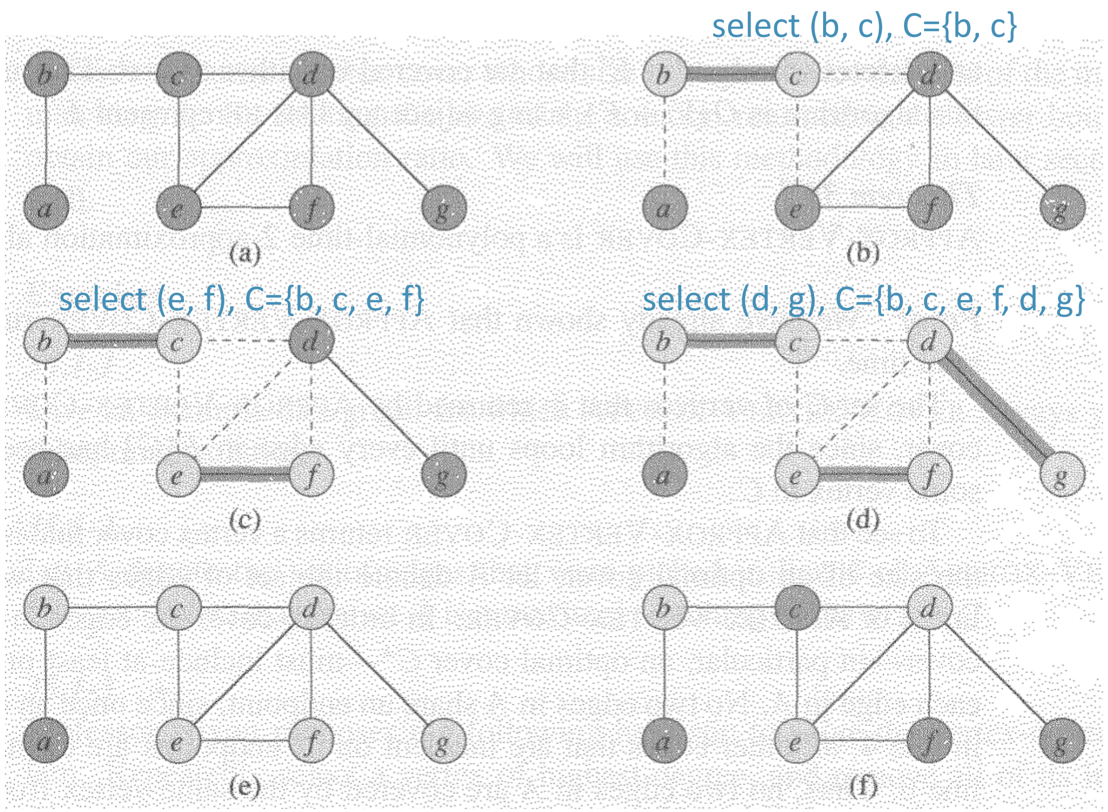
- A *vertex cover* of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge of G, then either $u \in V'$ or $v \in V'$ (or both).
- The size of a vertex cover is the number of vertices in it.
- The vertex-cover problem is to find a vertex cover of minimum size in a given undirected graph.
- This problem is NP-hard and its corresponding decision problem is NP-complete.
  - For the decision problem with parameter k, a straightforward solution is to check all subsets $V' \subseteq V$ of size k.
  - The time complexity is $|V|^k$ (can't be bounded by a polynomial function).

A vertex cover

An minimum vertex cover

Image source: https://en.wikipedia.org/wiki/Vertex_cover

Approximation Algorithm for the Vertex-Cover Problem




```
vertex_set approx_vertex_cover (graph G)
{
    vertex_set C;
    edge_set E;

    C = Ø;
    E = all the edges of G;
    while (!empty(E)){
        let (u, v) be an arbitrary edge in E;
        C = C ∪ {u, v};
        remove from E every edge whose endpoint is either u or v;
    }
    return C;
}
```

The time complexity of this algorithm is $O(|V| + |E|)$

XIAMEN UNIVERSITY MALAYSIA 厦門大學馬來西亞分校 厦门大学信息学院 SCHOOL OF INFORMATICS XIAMEN UNIVERSITY 厦门大学计算机科学系 Computer Science Department of Xiamen University

Image source: Figure 35.1, Thomas H. Cormen, Introduction to Algorithms, Second Edition.

Approximation Algorithm for the Vertex-Cover Problem

Theorem 1

`approx_vertex_cover` is a polynomial-time 2-approximation algorithm.

Proof:

- We have already shown that `approx_vertex_cover` runs in polynomial time.
- let A denote the set of edges that were picked in the while loop.
- An optimal cover C^* must include at least one endpoint of each edge in A, because C^* covers every edge in A.
- No two edges in A share an endpoint, since once an edge is picked, all other edges that share the same endpoints with the picked edge are deleted from E.

Approximation Algorithm for the Vertex-Cover Problem

Proof (cont'd):

- Thus, no two edges in A are covered by the same vertex in C^*.
- In other words, one vertex in C^* can at most cover one edge in A.
  - It is possible that two vertex in C^* covers one edge in A.
- Therefore, we have the lower bound

$$|C^*| \geq |A|.$$

- Each edge pick puts two new endpoints in C, we have:

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

THE SET-COVERING PROBLEM

The Set-Covering Problem

- An instance (X, F) of the *set-covering problem* consists of a finite set X and a family F of subsets of X, such that every element of X belongs to at least one subset in F:

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

- The problem is to find a minimum number of subsets $C \subseteq F$, which include all elements of X:

$$X = \bigcup_{S \in C} S\,.$$

- This problem is NP-hard and its corresponding decision problem is NP-complete.
  - Similar to the vertex-cover problem, the time-complexity of a brute-force algorithm for the decision problem is $|F|^k$.

The Set-Covering Problem

- X consists of 12 black points, and F is a family of subsets of X.

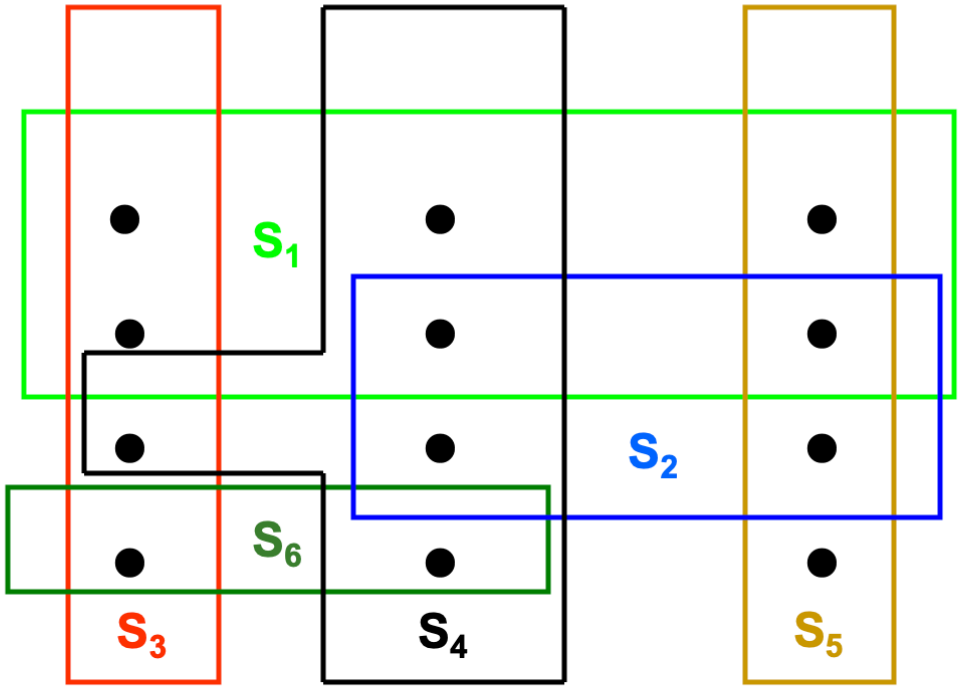
$$F\colon \{S_1, S_2, S_3, S_4, S_5, S_6\}.$$

- An optimal solution $C^* \subseteq F$ is:

$$C^* = \{S_3, S_4, S_5\}.$$

- A solution produced by the greedy algorithm $C \subseteq F$ is:

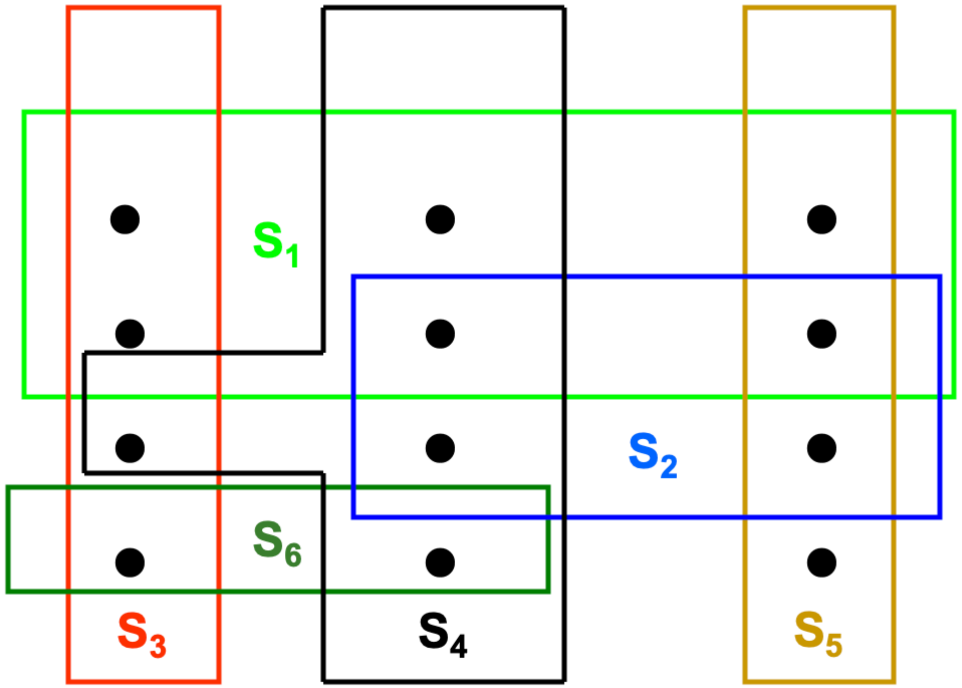
$$C = \{S_1, S_3, S_4\ S_5\}.$$



Approximation Algorithm for the Set-Covering Problem

```
set greedy_set_cover (set X, set F)
{
    set U, C;
    U = X;
    C = Ø;

    while (!empty(U)){
        select an S from F that maximizes |S ∩ U|;
        U = U − S;
        C = C ∪ {S};
    }
    return C;
}
```



- At each stage, pick the set S that covers the greatest number of remaining elements that are uncovered.
- Result: Add to C the sets S_1, S_4, S_5, S_3 in order.

Approximation Algorithm for the Set-Covering Problem

- The number of iterations of the loop is bounded from above by $\min(|X|, |F|)$.
  - If $|X| < |F|$, the size of $|U|$ is reduced in each iteration. Therefore there are at most $|X|$ loops.
  - If $|X| > |F|$, we will not repeat selecting the same S from F. Therefore there are at most $|F|$ loops.
- The loop body can be implemented to run in time $O(|X||F|)$.
- Total time complexity: $O(|X||F|\min(|X|, |F|))$, which is polynomial in $|X|$ and $|F|$.

```
set greedy_set_cover (set X, set F)
{
    set U, C;
    U = X;
    C = Ø;

    while (!empty(U)){
        select an S from F that maximizes |S ∩ U|;
        U = U − S;
        C = C ∪ {S};
    }
    return C;
}
```

Approximation Algorithm for the Set-Covering Problem

Theorem 2

`greedy_set_cover` is a polynomial-time $(\ln|X| + 1)$-approximation algorithm.

- The proof is skiped here due to high complexity.
- In this example, the approximation ratio $\rho(n)$ is not a constant but a logarithm function of the size of input X.
  - As the size of the instance gets larger, the size of the approximate solution may grow, relative to the size of an optimal solution.
  - Because the logarithm function grows rather slowly, however, this approximation algorithm may nonetheless give useful results.

THE TRAVEL-SALESMAN PROBLEM

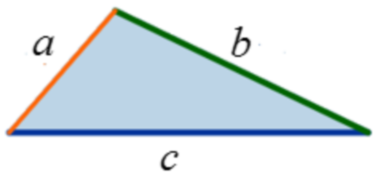
The Travel-Salesman Problem

- Given a complete undirected graph $G = (V, E)$ that has a nonnegative integer cost $c(u, v)$ associated with each edge $(u, v) \in E$, and we must find a Hamiltonian cycle (i.e. a tour) of G with minimum cost.
- This problem is NP-hard and its corresponding decision problem is NP-complete.
  - Worst-case time complexity of dynamic programming solution is $\Theta(n^2 2^n)$.
  - The state space tree in the branch-and-bound algorithm has $(n-1)!$ leaves. The worst-case is that the optimal solution is found on the last leaf, i.e. no node is pruned.

The Travel-Salesman Problem

- In many practical situations, it is always cheapest to go directly from a place u to a place w; going by way of any intermediate stop v can't be less expensive.
  - Usually true if the cost is distance you walk.
  - Sometimes not true if the cost is the flight price.
- Reversely, cutting out an intermediate stop never increases the cost.
- We formalize this notion by saying that the cost function c satisfies the *triangle inequality* if for all vertices $u, v, w \in V$,

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

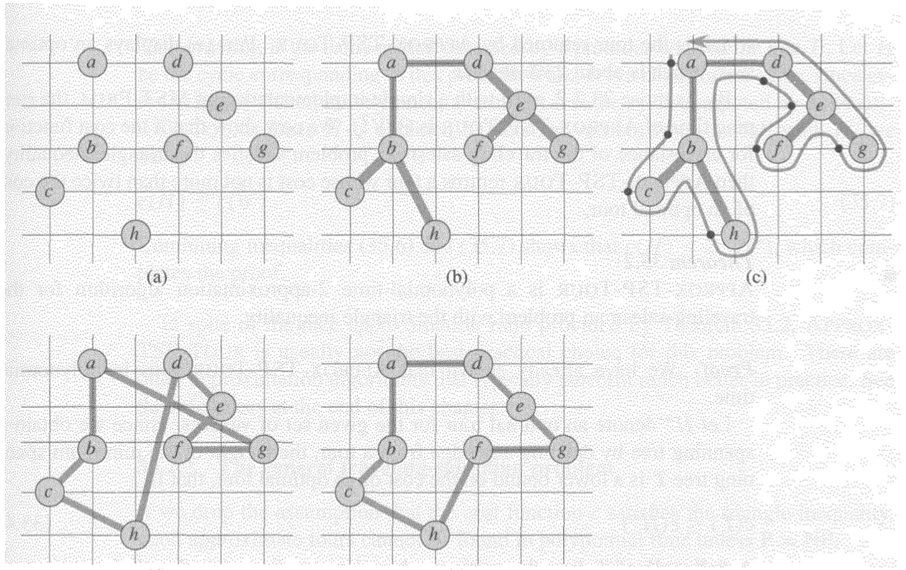


$$a+b>c$$
$$a+c>b$$
$$b+c>a$$

Approximation Algorithm for the Travel-Salesman Problem

- We will first use Prim’s algorithm to compute a minimum spanning tree (MST), whose weight is a lower bound on the length of an optimal TSP tour.
  - Recall that the every-case time complexity for Prim’s algorithm is $T(n^2)$.
  - The optimal cost for TSP must be less than the one for MST (removing any edge from the tour is a spanning tree).
- We will then use the MST to create a tour whose cost is no more than **twice** that of the MST's weight, as long as the cost function satisfies the triangle inequality.
  - Thus, it is a 2-approximation algorithm.

Approximation Algorithm for the Travel-Salesman Problem



Approx. tour Optimal tour

```
cycle approx_tsp_tour (graph G,
                       const number W[][])
{
    vertex r;
    tree T;
    vertex_list L;

    r = a vertex in G; // select the root
    T = prim(G, W, r);
    L = the list of vertices visited in a preorder tree walk of T;
    return the Hamiltonian cycle that visits the vertices in the order L;
}
```

- Assume that each two vertices are connected in the undirected graph.
- Actually, Prim's algorithm doesn't need to specify the root. However, here we need a root to do traversal.
- Full walk of the tree: a, b, c, b, h, b, a, d, e, f, e, g, e, d, a, shown in (c).
- Preorder walk of the tree: a, b, c, h, d, e, f, g , shown in (d).

Image source: Figure 35.2, Thomas H. Cormen, Introduction to Algorithms, Second Edition.

Approximation Algorithm for the Travel-Salesman Problem

- Now you may ask: what if $c(h, d)$ is super high, can the total cost of this tour still be at most twice of that of the optimal tour?
- No worry. The triangle inequality helps us dispel worries.

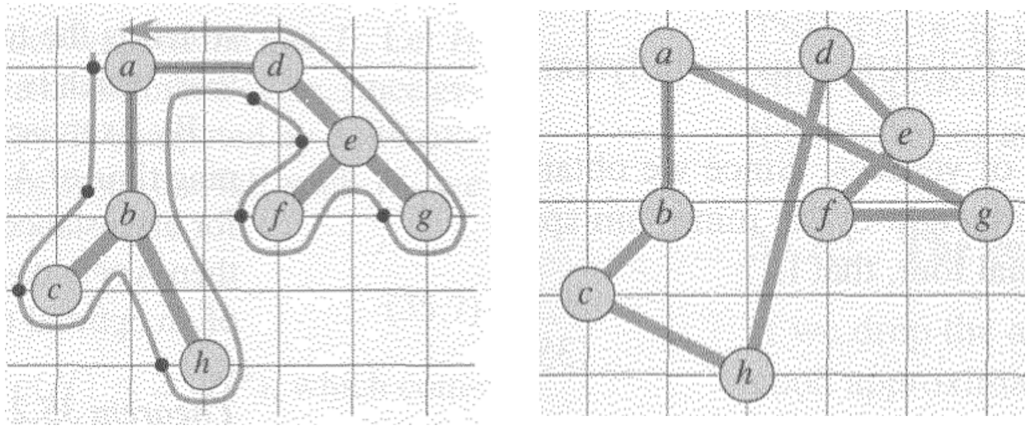


Image source: Figure 35.2, Thomas H. Cormen, Introduction to Algorithms, Second Edition.

Approximation Algorithm for the Travel-Salesman Problem

Theorem 3

`approx_tsp_tour` is a polynomial-time 2-approximation algorithm for TSP with the triangle inequality.

Proof:

- `approx_vertex_cover` is simply a call to Prim's algorithm with a preorder traversal, that is obviously in polynomial time.
- Let H^* denote an optimal tour for the given set of vertices.
- Since we can obtain a spanning tree by deleting any edge from the optimal tour, the cost of the MST T must be a lower bound on the cost of an optimal tour, i.e.

$$c(T) \leq c(H^*),$$

where $c(\cdot)$ denotes the total cost of the edges in the tree/tour.

Approximation Algorithm for the Travel-Salesman Problem

Proof (cont'd):

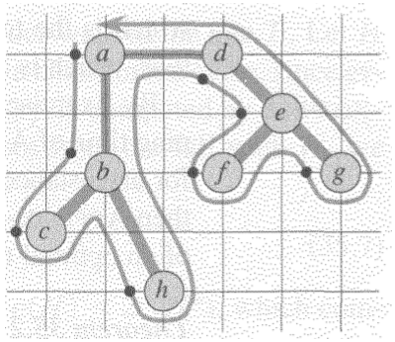
- Since the full walk of T (let us call this walk W) traverses every edge of T exactly twice, we have

$$c(W) = 2c(T).$$

- Hence, we have

$$c(W) \leq 2c(H^*).$$

- That is, the cost of W is within a factor of 2 of the cost of an optimal tour.



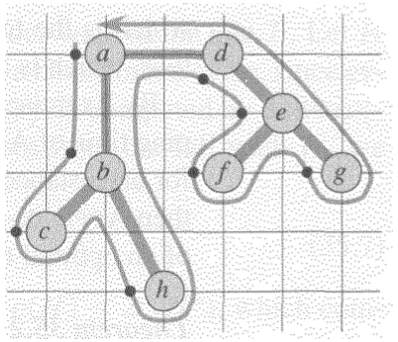
Full walk W of T

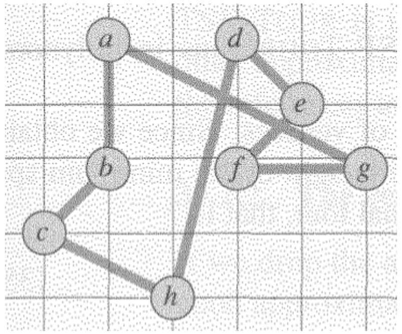
Image source: Figure 35.2, Thomas H. Cormen, Introduction to Algorithms, Second Edition.

Approximation Algorithm for the Travel-Salesman Problem

Proof (cont'd):

- However, you may notice that a very important problem: W is generally not a tour.
  - It visits each internal nodes twice in T.
- By the triangle inequality, we can delete a visit to any vertex from W and the cost does not increases.
  - If a vertex v is deleted from W between visits to u and w, the resulting ordering specifies going directly from u to w.
- By repeatedly applying this operation, we can remove from W all but the first visit to each vertex, i.e. we obtain the preorder walk of the tree finally.
  - a, b, c, ~~b~~, h, ~~b~~, ~~a~~, d, e, f, ~~e~~, g, ~~e~~, ~~d~~, a.





The Hamiltonian cycle H generated by full walk W

Approximation Algorithm for the Travel-Salesman Problem

Proof (cont'd):

- Since H is obtained by deleting vertices from the full walk W, we have

$$c(H) \leq c(W).$$

- We therefore have:

$$c(H) \leq 2c(H^*).$$

- That is, the theorem is proved.

MAX-CNF SATISFIABILITY PROBLEM

Randomized Approximation Algorithm

- Just as there are randomized algorithms that compute exact solutions, there are randomized algorithms that compute approximate solutions.
- We say that a randomized algorithm for a problem has an approximation ratio of $\rho(n)$ if, for any input of size n, the expected cost $E[C]$ of the solution produced by the randomized algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

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

- We call this kind of algorithm *randomized $\rho(n)$-approximation algorithm*.
  - It is like a deterministic approximation algorithm, except that the approximation ratio is for an expected value.

MAX-CNF Satisfiability Problem

- The input consists of n Boolean variables $x_1, \ldots, x_n$, each of which may be set to either $true$ or $false$.
- m clauses $C_1, \ldots, C_m$, each of which consists of an "OR" operator of some number of the variables and their negations
  - For example, $x_3 \vee \neg x_5 \vee x_{11}$, where $\neg x_i$ is the negation of x_i.
- A nonnegative weight w_j for each clause C_j.
- The objective of the problem is to find an assignment of $true/false$ to the x_i that maximizes the total weights of the satisfied clauses.

MAX-CNF Satisfiability Problem

- For example, we have:
  - $C_1: x_1 \vee \neg x_2$ with $w_j = 1$.
  - $C_2: x_1 \vee x_2$ with $w_j = 2$.
  - $C_3: \neg x_1 \vee \neg x_2$ with $w_j = 3$.
  - $C_4: \neg x_1 \vee x_2$ with $w_j = 4$.
- The optimal solution is $x_1 = false, x_2 = true$ with total weight 9.

Randomized Approximation Algorithm for MAX-CNF Satisfiability Problem

- Now, we have an extremely simple randomized algorithm:

Set each x_i *to* $true$ *independently with probability 1/2.*

- And we have the following theorem:

> **Theorem 4**
>
> The randomized algorithm gives a randomized 2-approximation algorithm for the maximum satisfiability problem.

Randomized Approximation Algorithm for MAX-CNF Satisfiability Problem

Proof:

- Consider a random variable Y_j such that Y_j is 1 if clause C_j is satisfied and 0 otherwise.
- Let $W = \sum_{j=1}^{m} w_j Y_j$ be a random variable that is equal to the total weight of the satisfied clauses.
- Then, recall the lemma for probabilistic analysis: $E[Y_j] = Pr[\text{clause } C_j \text{ satistied}]$.

$$E[W] = \sum_{j=1}^{m} w_j E[Y_j] = \sum_{j=1}^{m} w_j Pr[\text{clause } C_j \text{ satistied}].$$

- For each clause C_j, the probability that it is not satisfied is the probability of when
  - each unnegated literal in C_j is set to $false$;
  - each negated literal in C_j is set to $true$.

Randomized Approximation Algorithm for MAX-CNF Satisfiability Problem

Proof (cont'd):

- Because each of which happens with probability 1/2 independently, we have:

$$Pr[\text{clause } C_j \text{ satistied}] = \left(1 - \left(\frac{1}{2}\right)^{l_j}\right) \geq \frac{1}{2},$$

  where $l_j \geq 1$ the size of clause j.

- Let OPT denote the optimum value of the MAX-CNF instance:

$$E[W] = \sum_{j=1}^{m} w_j Pr[\text{clause } C_j \text{ satistied}] \geq \frac{1}{2}\sum_{j=1}^{m} w_j \geq \frac{1}{2} OPT,$$

  because the sum over all w_j is the upper bound of OPT.

Conclusion

After this lecture, you should know:

- Why do we need approximation algorithms.
- How to measure the gap between an approximate solution and an optimal solution.
- How to get a polynomial-time approximation algorithm and prove its approximation ratio $\rho(n)$.

Assignment

- No tutorial this week.
- Assignment 6 is released. The deadline is **18:00, 13th July**.

Thank you!

Reference:

- Chapter 35, Thomas H. Cormen, Introduction to Algorithms, Second Edition.