

# Questions

---

- Office hours today 2-3 & 4-5
- HW 6 due today
- HW 7 posted
- Quiz 7 open

# Approximation Algorithms

---

# Coping With NP-Completeness

---

## Stick with small problems

- Develop clever enumeration strategies.
- Guaranteed to find optimal solution.
- No guarantees on running time, but problem is small.

## Special Cases

- Look at specific types of input
- Find an algorithm that runs in polynomial time
- Gives correct solution.
  - Example: vertex cover in bipartite graphs, perfect graphs.

## Heuristics.

- Develop intuitive algorithms.
- Can solve the problem exactly for a subset of inputs
- Guaranteed to run in polynomial time.
- No guarantees on quality of solution for all inputs.

# Coping With NP-Completeness

---

## Approximation algorithms.

- Guaranteed to run in polynomial time.
- Guaranteed to find "high quality" solution, say within 5% of optimum.

Obstacle: need to prove a solution's value is close to optimum, without even knowing what optimum value is!

## Unwilling to relax correctness.

- Solve in exponential time but faster than brute force..
- Example:
  - Dynamic Programming for Knapsack  $O(nW)$
  - Brute Force  $O(2^n)$

## Average Case .

- Find an algorithm which works well on average.
- Average running time is polynomial.

# Approximation Algorithm

---

- An algorithm that returns near-optimal solutions is called an ***approximation algorithm***.
- **We need to find an *approximation ratio bound* for an approximation algorithm.**

# Approximation Ratio bound

---

We say an approximation algorithm for the problem has a ratio bound of  $\rho(n)$  if for any input size  $n$ , the cost  $C$  of the solution produced by the approximation algorithm is within a factor of  $\rho(n)$  of the  $C^*$  of the optimal solution:

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

This definition applies for both minimization and maximization problems.

# Approximate Ratio

---

- If  $\rho(n)=1$ , then the algorithm is an optimal algorithm
- The larger  $\rho(n)$ , the worse the algorithm
- If  $\rho(n)=2$  the algorithm is sometimes called a 2OPT or 2-Approximation solution

# A 2-Approximation Algorithm for Metric TSP

---

- The key to designing approximation algorithm is to obtain a bound on the optimal value OPT.
- In the case of TSP, the minimum spanning tree gives a lower bound for OPT the minimum TSP tour.

Note: In metric TSP

$$d(a,b) \leq d(a,c) + d(c,b)$$

*Always as fast to go “directly” from a to b.*

Euclidean TSP is an example of a metric TSP.

*\*\* Another metric would be the Manhattan (taxi-cab) distance.*



# Approximation Algorithms for Metric TSP

---

- Claim: The cost of a Minimum Spanning Tree is no greater than the cost of an optimal TSP tour.

$$\text{cost(MST)} \leq \text{OPT}$$

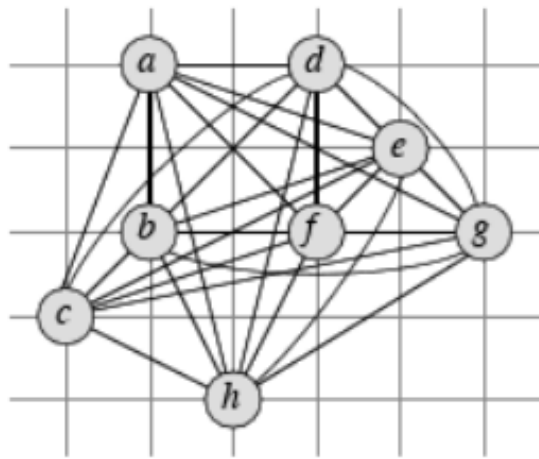
- Proof: Notice that deleting an edge from a TSP tour results in a spanning tree. Therefore, the minimum spanning tree give us a lower bound on OPT, the cost of the optimal TSP tour.
- This gives us several simple 2-approximation algorithms for metric TSP.

# Approx Algorithm - 1

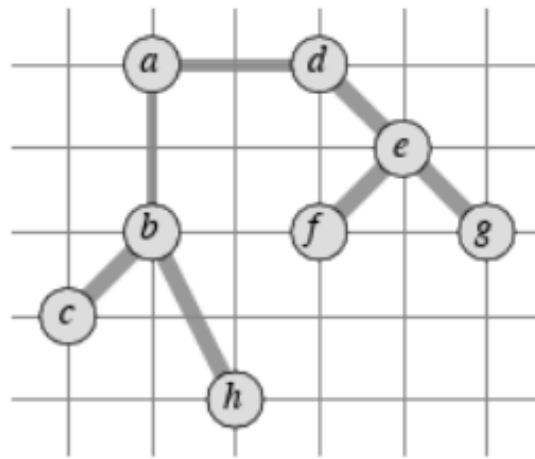
---

APPROX-TSP-TOUR( $G, c$ )

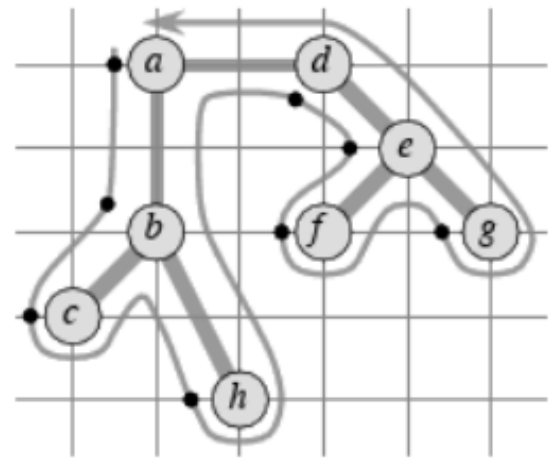
- 1 select a vertex  $r \in G.V$  to be a “root” vertex
- 2 compute a minimum spanning tree  $T$  for  $G$  from root  $r$   
using MST-PRIM( $G, c, r$ )
- 3 let  $H$  be a list of vertices, ordered according to when they are first visited  
in a preorder tree walk of  $T$
- 4 **return** the hamiltonian cycle  $H$



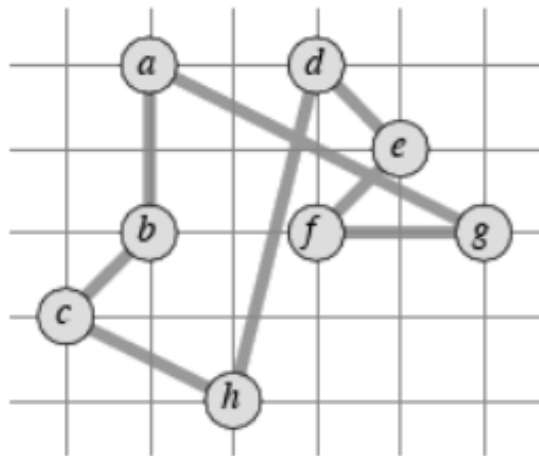
(a)



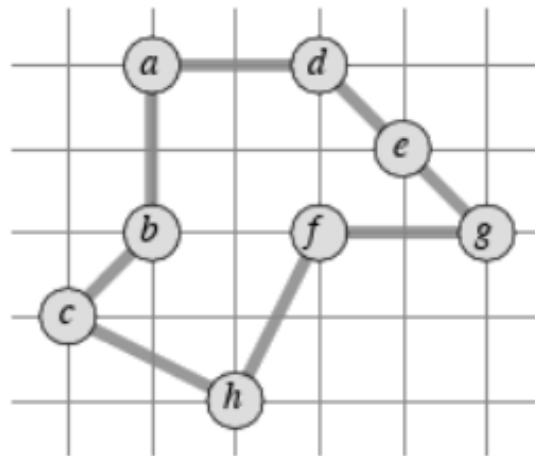
(b)



(c)



(d)



(e)

# Questions

---

# Definition

---

We need the following definition to design the algorithm.

- Definition: A (multi-) graph is **Eulerian** if it has a walk that uses every edge exactly once.
- Fact: A connected graph is Eulerian if and only if every vertex has even degree.

# Approx Algorithm - 2

---

1. Find a minimum spanning tree of  $G$
2. Duplicate each edge in the minimum spanning tree to obtain a Eulerian graph
3. Find a Eulerian tour  $E$  of the Eulerian graph
4. Convert  $E$  to a tour  $T$  by going through the vertices in the same order of  $E$  , skipping vertices that were already visited

Claim: The above algorithm is a 2-approximation algorithm for TSP

# Proof

---

Claim: The above algorithm is a 2-approximation algorithm for Metric TSP

Proof: As noted above,  $\text{cost}(\text{MST}) \leq \text{OPT}$ . Since  $E$  contains two copies of each edge in  $\text{MST}$ ,  
$$\text{cost}(E) = 2\text{cost}(\text{MST}).$$

Finally, by triangle inequality, shortcutting previously visited vertices does not increase the cost. Hence we have

$$\text{cost}(T) \leq 2\text{cost}(\text{MST}) \leq 2\text{OPT}.$$

# Christofides' Algorithm for TSP

Given an instance for TSP problem,

---

1. Find a minimum spanning tree **T** for that instance.
2. Find a **min-cost perfect matching M** for odd-degree nodes of **T**.
3. **T**  $\cup$  **M** has an Euler tour. Find the tour.
4. List the vertices as visited by the Euler tour. Remove duplicates to get a **TSP tour H**.



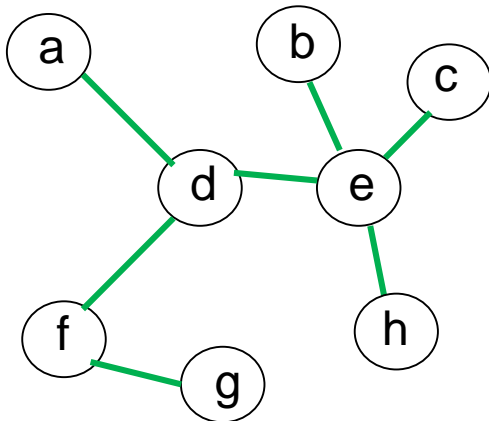
# Christofides' Algorithm for TSP

Given an instance for TSP problem,

---

1. Find a minimum spanning tree **T** for that instance.
2. Find a **min-cost perfect matching M** for odd-degree nodes of **T**.
3. **T**  $\cup$  **M** has an Euler tour. Find the tour and list vertices.
4. Remove duplicates to get a **TSP tour H**.

Step 1: MST



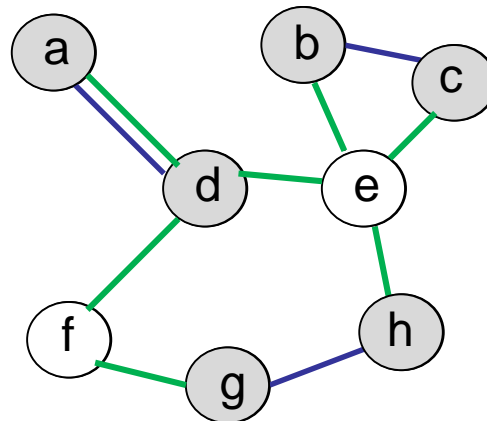
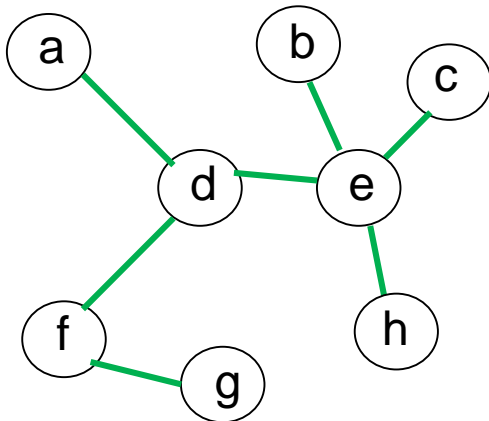
# Christofides' Algorithm for TSP

Given an instance for TSP problem,

---

1. Find a minimum spanning tree **T** for that instance.
2. Find a **min-cost perfect matching M** for odd-degree nodes of **T**.
3. **T**  $\cup$  **M** has an Euler tour. Find the tour and list vertices.
4. Remove duplicates to get a **TSP tour H**.

Step 2: Min Perfect Matching for odd vertices

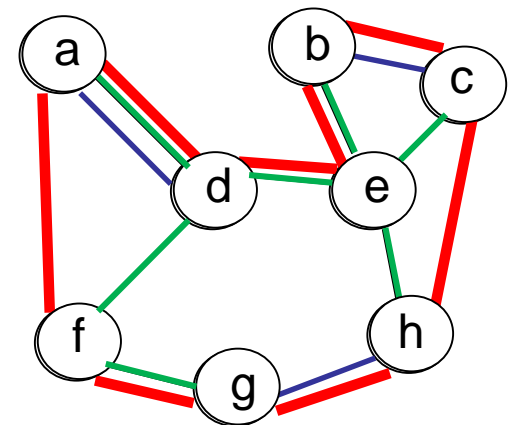
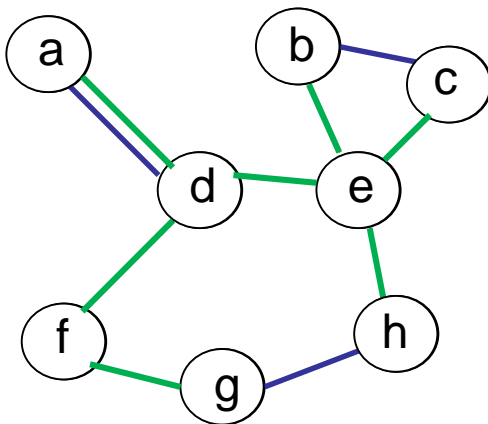


# Christofides' Algorithm for TSP

Given an instance for TSP problem,

1. Find a minimum spanning tree **T** for that instance.
2. Find a **min-cost perfect matching M** for odd-degree nodes of **T**.
3. **T**  $\cup$  **M** has an Euler tour. Find the tour and list vertices.
4. Remove duplicates to get a **TSP tour H**.

Step 3: All degrees are even so find an Euler tour and list vertices:  
a,d,e,b,c,e,h,g,f,d,a

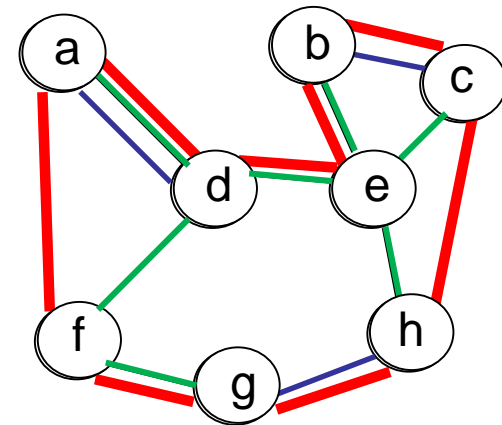
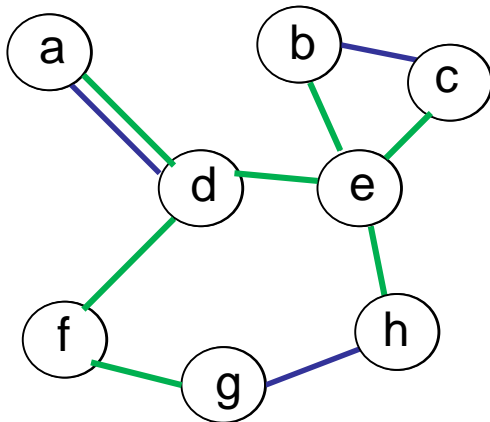


# Christofides' Algorithm for TSP

Given an instance for TSP problem,

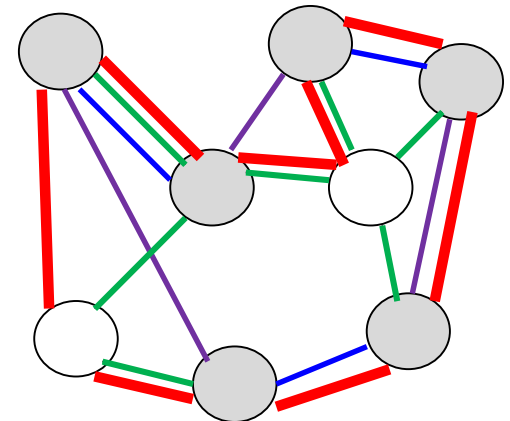
1. Find a minimum spanning tree **T** for that instance.
2. Find a **min-cost perfect matching M** for odd-degree nodes of **T**.
3.  $T \cup M$  has an Euler tour. Find the tour and list vertices.
4. Remove duplicates to get a **TSP tour H**.

Step 4: Remove duplicates: a,d,e,b,c,e,h,g,f,d,a



# Approximation Bound for Christofides' algorithm

- **Theorem:** *If the cost function satisfies the triangle inequality,*  
*then the Christofides' algorithm is a 1.5-approximation algorithm.*
- **Proof:** Let  $H^*$  be an optimal TSP solution.  
 $\text{cost}(\text{MST } T) \leq \text{cost}(H^*)$  (based on our previous arguments)  
Get a cycle  $C$  on the set of odd-degree nodes of  $T$ ,  
by joining them in the order they appear in  $H$ .  
 $C$  partitions into two perfect matchings for odd-degree nodes:  $M_1$  and  $M_2$ .  
 $\text{cost}(M_1) + \text{cost}(M_2) = \text{cost}(C) \leq \text{cost}(H^*)$  (based on triangle inequality)  
For at least one of the matchings, say  $M_1$ ,  $\text{cost}(M_1) \leq 0.5 * \text{cost}(H^*)$ .  
Since  $M$  is a min-cost perfect matching,  $\text{cost}(M) \leq \text{cost}(M_1) \leq 0.5 * \text{cost}(H^*)$ .  
So  $\text{cost}(H) \leq \text{cost}(T) + \text{cost}(M) \leq 1.5 * \text{cost}(H^*)$ .





# Greedy Approximations

---

- Use a greedy algorithm to solve the given problem
  - Repeat until a solution is found:
  - Among the set of possible next steps:  
Choose the current best-looking alternative and commit to it
- Usually fast and simple
- Works in some cases...(always finds optimal solutions)
  - Dijkstra's single-source shortest path algorithm
  - Prim's and Kruskal's algorithm for finding MSTs
- but not in others...(may find an approximate solution)
  - TSP – always choosing current least edge-cost node to visit next

# Knapsack

---

- Given: a set  $S$  of  $n$  objects with nonnegative weights and values, and a weight bound:
  - $w_1, w_2, \dots, w_n, W$  (weights, weight bound).
  - $v_1, v_2, \dots, v_n$  (values).
  - for all  $i$ ,  $w_i \leq W$
- Find: subset of  $S$  with total weight at most  $W$ , and maximum total value.
- Define:  $OPT$  = The optimal solution.
- Problem is known to be NP-hard





# Knapsack Approximation

---

## Greedy1 Algorithm

- Define the density ratio:  $d_i = \frac{v_i}{w_i}$  for item  $i$
- Sort items in non-increasing order such that  
 $d_{(1)} \geq d_{(2)} \geq \dots \geq d_{(n)}$ .
- Greedily pick items in above order until  
 $A = \{\text{item}(1), \text{item}(2), \dots, \text{item}(j)\}$  with  $\text{weight}(A) \leq W$  and  
 $\text{weight}(A) + w_{(j+1)} > W$

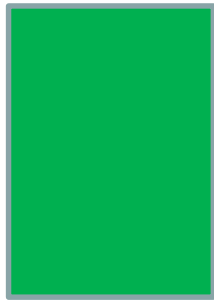
# Knapsack Approximation

---

- Seems like a quick approximation algorithm but it can be very “bad”
- Consider the small example:



Item 1  
 $w=1, v=2$   
 $d=2$



Item 2  
 $w=10, v=10$   
 $d=1$



Knapsack  $W = 10$

Greedy picks Item 1 with  $v = 2$   
Optimal Item 2 with  $v = 10$

$\rho = 10/2 = 5 = W/2$   
Error grows with  $W$

# Knapsack Approximation

---

- Seems like a quick approximation algorithm but it can be very “bad”
- Consider the small example
  - Item 1: weight 1 and value 2, density  $d = 2/1 = 2$
  - Item 2: weight  $W$  and value  $W$ , density  $d = W/W = 1$

The greedy algorithm picks the small Item 1 with value 2 since it has the larger density ratio.

This leaves  $W-1$  empty space in the backpack.

However there is no room for the large item which would have filled the entire backpack and given us total value  $W$ .

This is  $W/2$  off from optimal.  $W/2$  is not a constant.



# Knapsack 2-Approximation

---

## Greedy2 Algorithm

- Sort items in non-increasing order such that
$$d_{(1)} \geq d_{(2)} \geq \dots \geq d_{(n)}.$$
- Greedily pick items in above order until
$$A = \{\text{item}(1), \text{item}(2), \dots, \text{item}(j)\}$$
 with  $\text{weight}(A) \leq W$  and  $\text{weight}(A) + w_{(j+1)} > W$
- Pick the better of
$$A = \{\text{item}(1), \text{item}(2), \dots, \text{item}(j)\}$$
 and  $\{\text{item}(j+1)\}$ 
  - if  $\text{TotalValue}(A) > v_{(j+1)}$  return  $A$
  - else return  $\text{item}(j+1)$

*\*\* Note: If  $\text{weight}(A) = W$  then optimal.*

# Knapsack 2-Approximation

---

## Greedy2 Algorithm

- Pick the better of

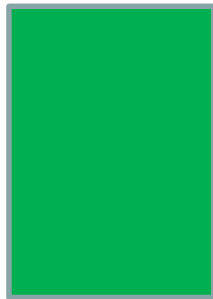
$A = \{\text{item}(1), \text{item}(2), \dots, \text{item}(j)\}$  and  $\{\text{item}(j+1)\}$

if  $\text{TotalValue}(A) > b_{(j+1)}$  return  $A$

else return  $\text{item}(j+1)$



Item 1  
 $w=1, v=2$   
 $d=2$



Item 2  
 $w=10, v=10$   
 $d=1$



Knapsack  $W = 10$

Greedy2 picks Item 2 with  $v = 10$

Optimal Item 2 with  $v = 10$

# Greedy2 is 2-approximation Knapsack

---

Proof: We used a greedy algorithm so if the solution is suboptimal then we must have some leftover space at the end. Let  $S$  be the total weight of the solution  $A$ , then the left over space is  $W - S$ . Imagine we were able to take a fraction of an item. Then by adding  $\frac{W-S}{w_{j+1}} v_{j+1}$  to our knapsack's total benefit, we would either match or exceed  $\text{OPT}$  (remember  $\text{OPT}$  is unable to use fractions).

So  $\text{OPT} \leq \sum_{i=1}^j v_i + \frac{W-S}{w_{j+1}} v_{j+1} \leq \sum_{i=1}^j v_i + v_{j+1}$  since we couldn't fit the entire  $(j+1)$ st item in the knapsack  $\frac{W-S}{w_{j+1}} < 1$ .

$$\text{OPT} \leq \sum_{i=1}^j v_i + v_{j+1} \leq 2 \max(\sum_{i=1}^j v_i, v_{j+1})$$

$$\text{OPT} \leq 2 \max(\sum_{i=1}^j v_i, v_{j+1})$$

The Greedy2 solution is  $\max(\sum_{i=1}^j v_i, v_{j+1})$

$$\text{OPT} \leq 2 * \text{Greedy2}$$



# Greedy2 is 2-approximation Knapsack

---

$$OPT \leq 2 * Greedy2$$

$$\frac{1}{2} OPT \leq Greedy2$$

$$Greedy2 \geq \frac{1}{2} OPT.$$

Knapsack is a maximization problem so we want the largest possible solution. In the worse case  $Greedy2 = \frac{1}{2} OPT$

So we have a guaranteed approximation ratio of

$$\rho = \frac{OPT}{Greedy2} = \frac{OPT}{\frac{1}{2}OPT} = 2$$

# Integer Programming

---

- An optimization method similar to linear programming but all variables must be integers.
- A technique that can be used to solve many problems including Knapsack and Bin packing
- Unfortunately, there is no known polynomial algorithm for integer programming.
- Integer Programming is NP-complete
- Software and libraries to solve IP but slow for large problems.

# Integer Programming

---

## Example of Integer Programming

maximize:  $2x + y + z$  (objective)  
subject to:  $3x + y \leq 10$  (constraints)  
 $2y + z \leq 5$   
 $x, y, z \geq 0$  and integers

If  $x=3, y=1, z=3$  then objective  $2(3)+1+3 = 10$

If  $x=3, y=0, z=5$  then objective  $2(3)+0+5 = 11$

# Knapsack as IP problem

---

Let  $x_i = 1$  if item  $i$  is in knapsack  
 $= 0$  if item  $i$  is NOT in knapsack

Max  $V_1x_1 + V_2x_2 + \dots + V_nx_n$

Subject to:

$$W_1x_1 + W_2x_2 + \dots + W_nx_n \leq W$$

$$x_1, x_2, x_n \leq 1$$

$$x_1, x_2, x_n \geq 0$$

$$x_1, x_2, x_n \text{ integers}$$

# The Bin Packing Problem

---

Definition (Bin Packing Problem). Given a list of objects and their weights, and a collection of bins of fixed size, find the smallest number of bins so that all of the objects are assigned to a bin.

We consider packing problems of one dimension, though there is no conceptual difficulty extending the problem to  $p$  dimensions. In the one-dimensional problem objects have a single dimension (cost, time, size, weight, or any number of other measures). Problems of higher dimension have objects with more measures under consideration (cost and weight, length, width, and depth, etc.). The object is to pack a set of items into as few bins as possible. As the number of dimensions (measures) under consideration increases, the complexity increases tremendously, so we limit ourselves in this discussion to one dimension.

# Bin Packing is NP-complete

---

Bin Packing problem: Given  $n$  items of sizes  $a_1, a_2, \dots, a_n$  ( $0 < a_i \leq 1$ ), pack these items in at most  $k$  bins of size 1.

## 1. Bin packing in NP

### – To verify a solution

- Add the weights of the items in each bin.
- Each bin must contain  $\leq 1$  unit.
- Check that each item is in a bin
- There are at most  $k$  bins used.

### – This can be done in $O(n)$ .

## 2. SET-PARTITION reduces to Bin Packing

# SET-PARTITION $\leq_p$ Bin Packing

---

SET-PARTITION: Given a set of numbers  $S = \{s_1, s_2, \dots, s_n\}$ . Is there a subset of  $S$ ,  $X$ , such that the sum of the elements in  $X$  is equal to the sum of the elements in  $S-X$ .

Bin Packing: Given  $n$  items of sizes  $a_1, a_2, \dots, a_n$  ( $0 < a_i \leq 1$ ), pack these items in at most  $k$  bins of size 1.

Let  $\text{sum}(S) = \sum_{i=1}^n s_i$ . Define  $A = \{a_1, a_2, \dots, a_n\}$

where  $a_i = \frac{2s_i}{\text{sum}(S)}$  for  $i = 1, \dots, n$ . This can be done in  $O(n)$ .

Now we must show that  $S$  can be partitioned into 2 sets if and only if  $A$  can be packed into  $K=2$  bins.

Note:  $\text{Sum}(A) = \sum_{i=1}^n \frac{2s_i}{\text{sum}(S)} = 2 \frac{\sum_{i=1}^n s_i}{\text{sum}(S)} = 2$ .

If there exists a partition  $X$  for  $S$  then  $\text{sum}(X) = \text{sum}(S-X) = \text{sum}(S)/2$ .

If we put the elements in  $X$  into Bin 1 then the sum of the corresponding elements

$$\text{Sum}(\text{Bin1}) = \sum_{i \in X} \frac{2s_i}{\text{sum}(S)} = 2 \frac{\sum_{i \in X} s_i}{\text{sum}(S)} = 2 \frac{\text{sum}(X)}{\text{sum}(S)} = 2 \frac{\frac{\text{sum}(S)}{2}}{\text{sum}(S)} = 1.$$

# SET-PARTITION $\leq_p$ Bin Packing

---

SET-PARTITION: Given a set of numbers  $S = \{s_1, s_2, \dots, s_n\}$ . Is there a subset of  $S$ ,  $X$ , such that the sum of the elements in  $X$  is equal to the sum of the elements in  $S-X$ .

Bin Packing: Given  $n$  items of sizes  $a_1, a_2, \dots, a_n$  ( $0 < a_i \leq 1$ ), pack these items in at most  $k$  bins of size 1.

If there exists a partition  $X$  for  $S$  then  $\text{sum}(X) = \text{sum}(S-X) = \text{sum}(S)/2$ .

Likewise, if we put the elements from  $S-X$  into Bin 2 then the sum of the corresponding elements

$$\text{Sum}(\text{Bin2}) = \sum_{i \in S-X} \frac{2s_i}{\text{sum}(S)} = 2 \frac{\sum_{i \in S-X} s_i}{\text{sum}(S)} = 2 \frac{\text{sum}(S-X)}{\text{sum}(S)} = 2 \frac{\frac{\text{sum}(S)}{2}}{\text{sum}(S)} = 1.$$

Therefore if  $S$  has a partition we can pack the items of  $A$  into two bins.

If  $A$  can be packed into two bins then  $S$  has a set-partition. By the assignment of values to  $A$ . Each bin must have weight of exactly 1 and the corresponding items would create a partition of  $S$ .

Therefore, Bin-Packing is in NP-complete.



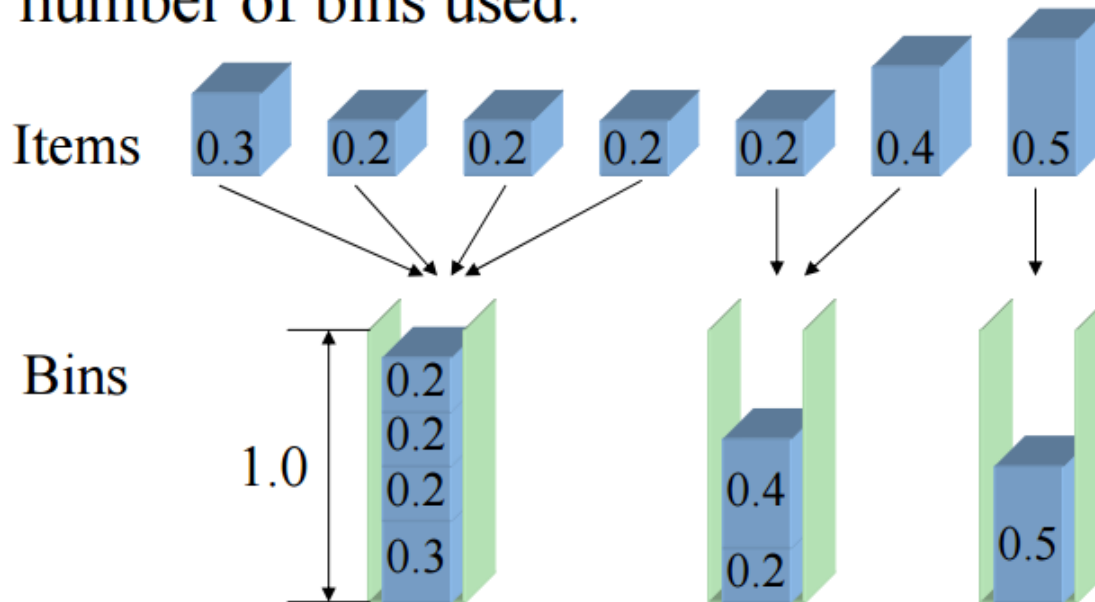
# Bin Packing Optimization

- Input:

- $n$  items with sizes  $a_1, \dots, a_n$  ( $0 < a_i \leq 1$ ).

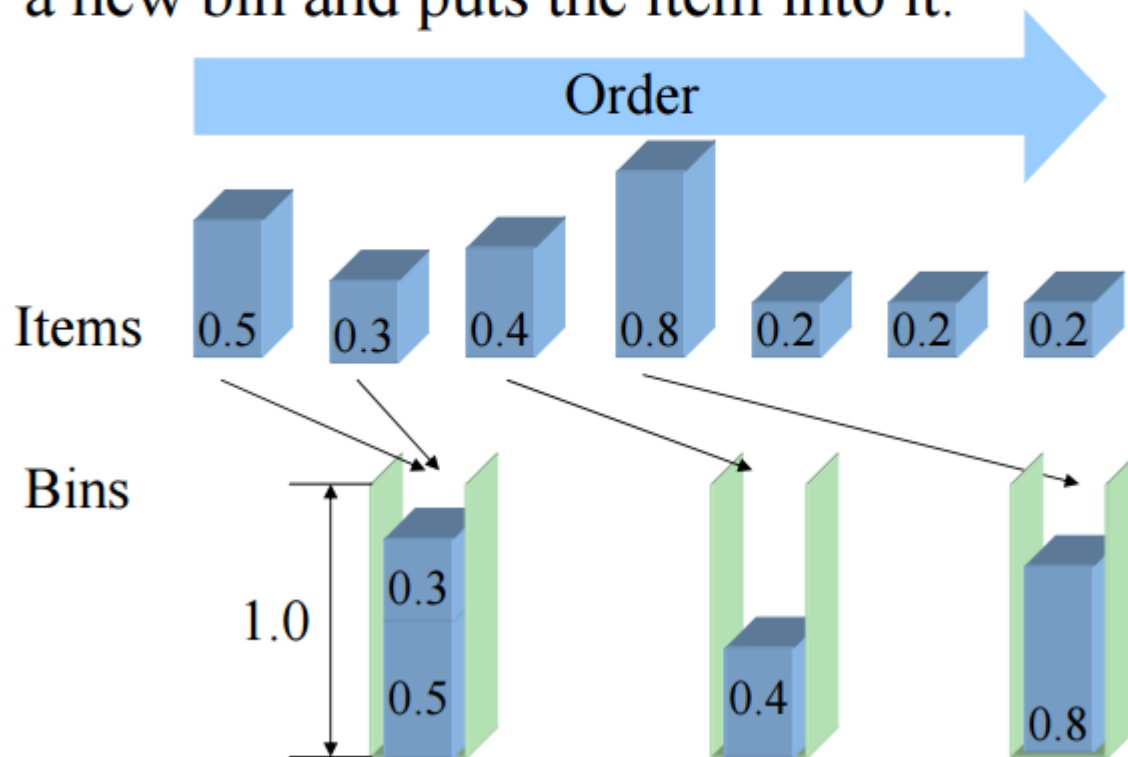
- Task:

- Find a packing in unit-sized bins that minimizes the number of bins used.



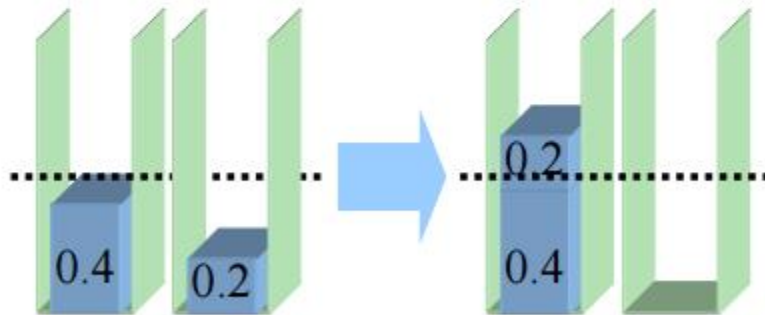
# First-Fit Algorithm

- This algorithm puts each item in one of partially packed bins.
  - If the item does not fit into any of these bins, it opens a new bin and puts the item into it.



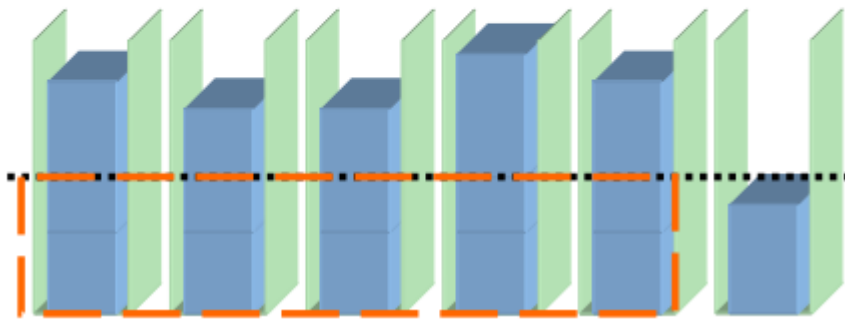
# First-Fit finds 2OPT solution

- OPT: # bins used in the optimal solution.
- [Proof]
  - Suppose that First-Fit uses  $m$  bins.
  - Then, at least  $(m-1)$  bins are more than half full.
    - We never have two bins less than half full.
      - If there are two bins less than half full, items in the second bin can be substituted into the first bin by First-Fit.



# First-Fit finds 2OPT solution


- Suppose that First-Fit uses  $m$  bins.
- Then, at least  $(m-1)$  bins are more than half full.



Sum of sizes  
of the items

$$\text{OPT} \geq \sum_{i=1}^n a_i > \frac{m-1}{2}$$

$$2\text{OPT} > m-1$$

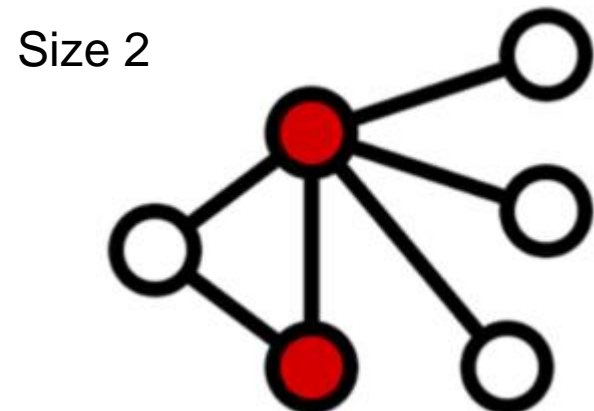
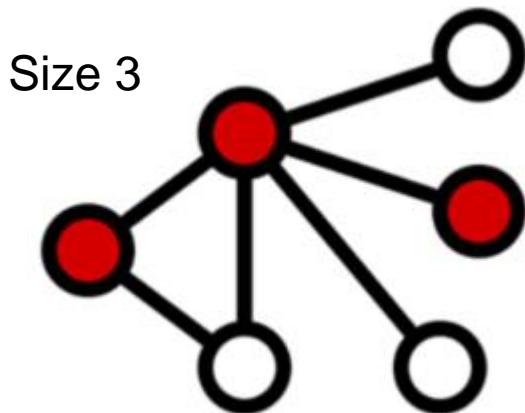
The size  
of 

Since  $m$  and  $\text{OPT}$  are integers.  $\longrightarrow 2\text{OPT} \geq m$

# Vertex-cover problem

---

- Vertex cover: Given an undirected graph  $G=(V,E)$ , then a subset  $C \subseteq V$  is a vertex cover if for all  $(u,v) \in E$ , then  $u \in C$  or  $v \in C$  (or both).
- Size of a vertex cover  $C$ : is the number of vertices in it.
- Vertex-cover Optimization Problem: Find a vertex-cover of minimum size.



# Examples

---

# Vertex-cover problem

---

- Decision Vertex-cover problem is NP-complete. Does there exist a vertex cover for graph  $G$  with size  $\leq K$ .
  - Vertex-cover belongs to NP.
  - Vertex-cover is NP-Complete ( $\text{CLIQUE} \leq_p \text{Vertex-cover}$ .)
    - Reduce  $\langle G, k \rangle$  where  $G = \langle V, E \rangle$  of a CLIQUE instance to  $\langle G', |V| - k \rangle$  where  $G' = \langle V, E' \rangle$  where  $E' = \{(u, v) : u, v \in V, u \neq v \text{ and } \langle u, v \rangle \notin E\}$  of a vertex-cover instance.
- So find an approximate algorithm for vertex-cover.

# Greedy Approximate Vertex-Cover algorithm

---

## APPROX-VERTEX-COVER(G)

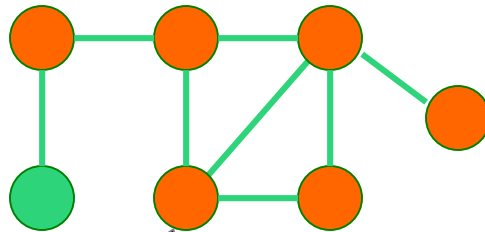
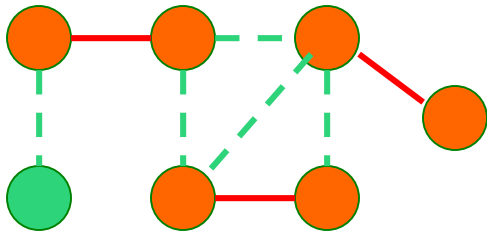
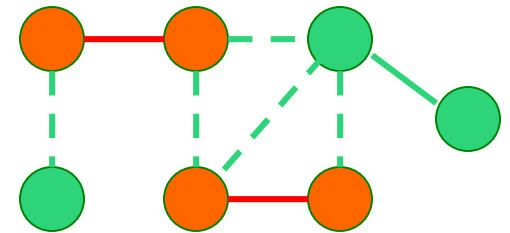
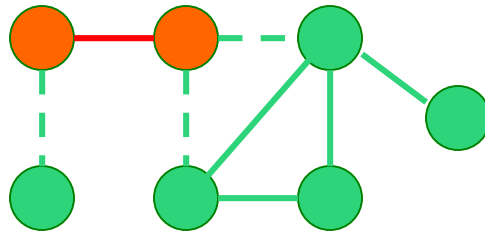
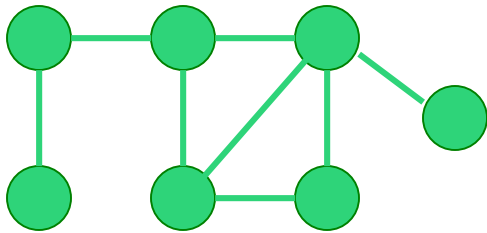
```
1  C ← ∅
2  E' ← E[G]
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C ← C ∪ (u, v)
6      remove every edge in E' incident on u or v
7  return C
```

Running time using an adjacency list is  $O(V+E)$ .

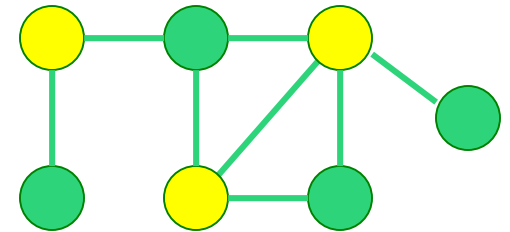


# Greedy Approximation for Vertex Cover

---



**Near Optimal  
size=6**



**Optimal  
Size=3**

# 2-approximate Vertex-Cover

---

The Greedy APPROXIMATE-VERTEX-COVER is a 2-approximate algorithm, i.e., the size of returned vertex cover set is at most twice of the size of optimal vertex-cover.

Proof:

Let  $A$  be the set of edges picked in line 4 and  $C^*$  be the optimal vertex-cover.

For each edge  $(u,v)$  selected for  $A$ ,  $C^*$  must include at least one of these vertices.

No two edges in  $A$  share a vertex so no two edges in  $A$  are covered by the same vertex in  $C^*$ . So

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

*Note:  $C^*$  is a set of vertices  $A$  a set of edges*

– Moreover,  $|C|=2|A|$ , so  $|C| \leq 2|C^*|$ .

# 2-approximate Vertex-Cover

---

$$|C^*| \geq |A| \quad \text{or} \quad 2|C^*| \geq 2|A|$$

Recall:  $C^*$  is a set of vertices and  $A$  is a set of edges selected in the algorithm.

The vertex cover  $C$  produced by the Greedy algorithm contains two vertices for each edge in  $A$ .

Therefore  $|C| = 2|A|$ , substituting into the inequality above yields,

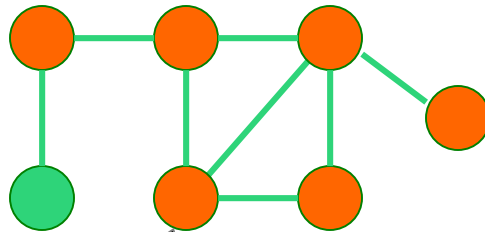
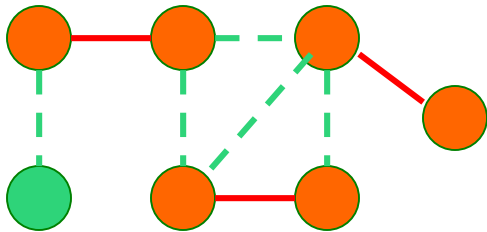
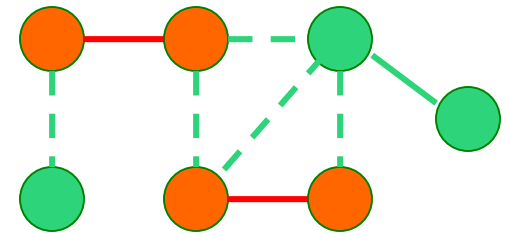
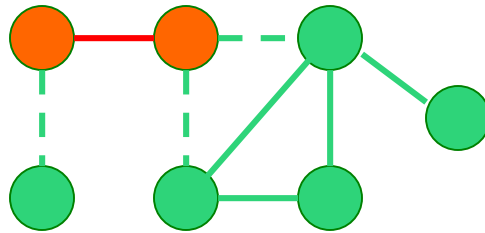
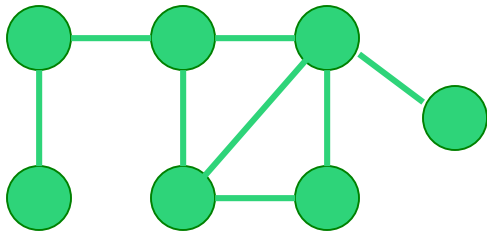
$$2|C^*| \geq |C| \quad \text{or} \quad |C| \leq 2|C^*|.$$

Since we are minimizing in the worst case  $|C| = 2|C^*|$ .  
So

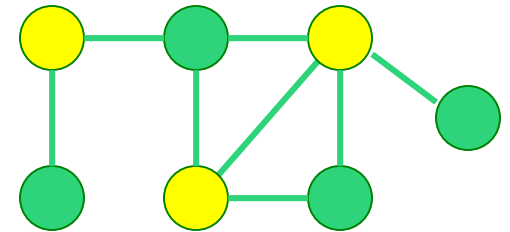
$$\rho = \frac{\text{Greedy}}{\text{OPT}} = \frac{2C^*}{C^*} = 2$$

# Greedy Approximation for Vertex Cover

---



**Near Optimal  
size=6**



**Optimal  
Size=3**

# What might be better?

---

# Vertex Cover Selecting Vertices

---

A natural modification is to repeatedly choose vertices which are incident to the largest number of *currently* uncovered edges.

```
GREEDY2 ( $G$ )
```

```
   $C \leftarrow \emptyset$ 
```

```
  while  $E \neq \emptyset$ 
```

```
    Pick a vertex  $v \in V$  of
```

```
      maximum degree in the current graph
```

```
     $C \leftarrow C \cup \{v\}$ 
```

```
     $E \leftarrow E \setminus \{e \in E : v \in e\}$ 
```

```
  return  $C$ 
```

# Greedy2 is bad for some graphs

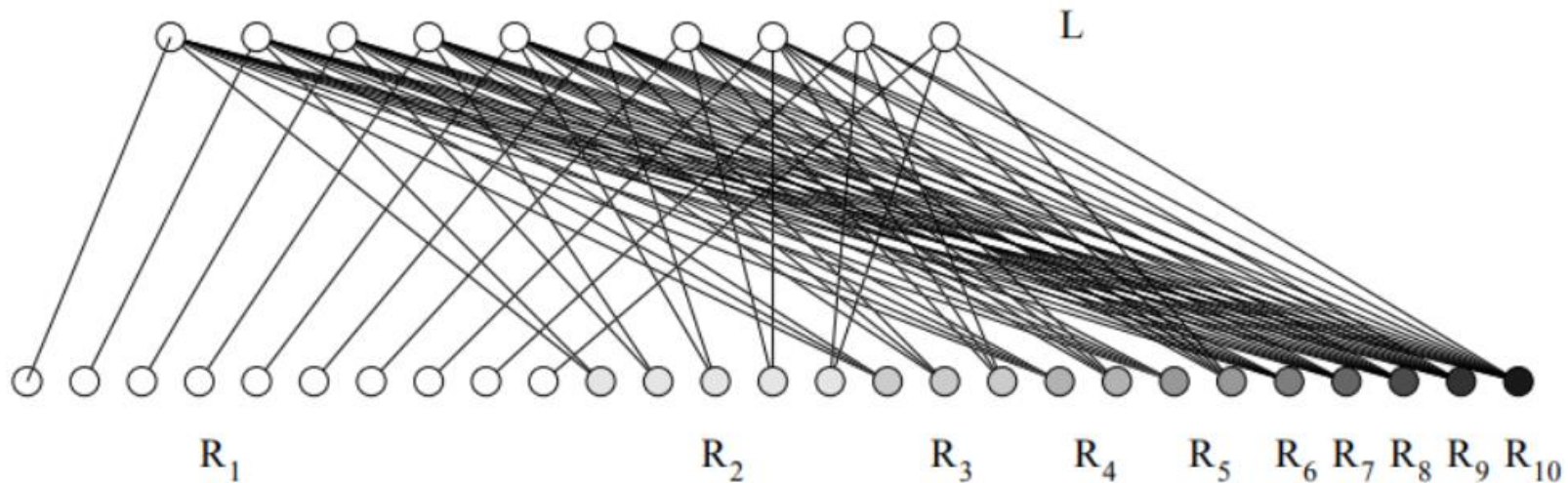
---

## Algorithm analysis

We claim that algorithm GREEDY2 does not achieve any bounded ratio. To see this, consider the following bipartite graph  $B = (L, R, E)$ . The vertex set  $L$  consists of  $r$  vertices. The vertex set  $R$  is further sub-divided into  $r$  sets called  $R_1, \dots, R_r$ . Each vertex in  $R_i$  has an edge to  $i$  vertices in  $L$  and no two vertices in  $R_i$  have a common neighbour in  $L$ ; thus,  $|R_i| = \lfloor r/i \rfloor$ . It follows that each vertex in  $L$  has degree at most  $r$  and each vertex in  $R_i$  has degree  $i$ . The total number of vertices  $n = \Theta(r \log r)$ .

# Bad luck Graph for Greedy2

---



Suppose that (out of sheer bad luck) the algorithm considers an edge out of  $R_r$  first, choosing the end-point in  $R$  as the vertex to be placed in the cover. Then it picks an edge out of  $R_{r-1}$ , again choosing its end-point in  $R$  for the cover  $C$ ; and, so on. Therefore the vertex cover chosen is  $C = R$ . But  $L$  is itself a vertex cover since the graph is bipartite. It follows that the ratio achieved by this algorithm is no better than  $|R|/|L| = \Omega(\log n)$ .