Course: Analysis of Algorithms
Code: CS33104
Branch: MCA -3rd Semester

Lecture 10 - Approximation Algorithms

Faculty & Coordinator : Dr. J Sathish Kumar (JSK)

Department of Computer Science and Engineering
Motilal Nehru National Institute of Technology
Allahabad, Prayagraj-211004

Approximation Algorithms

- ***NP-hard problems***
  - No known polynomial-time algorithms for these problems, and
  - there are serious theoretical reasons to believe that such algorithms do not exist.
- Approximation Algorithms allow for getting a solution close to the optimal solution of an optimization problem in polynomial time.
- A ***heuristic*** is a common-sense rule drawn from experience rather than from a mathematically proved assertion.

Approximation Algorithms

- If we use an algorithm whose output is just an approximation of the actual optimal solution, we would like to know how accurate this approximation is.
- We can quantify the accuracy of an approximate solution s_a to a problem of minimizing some function $\boldsymbol{f}$ by the size of the relative error of this approximation, where s^* is an exact solution to the problem, then

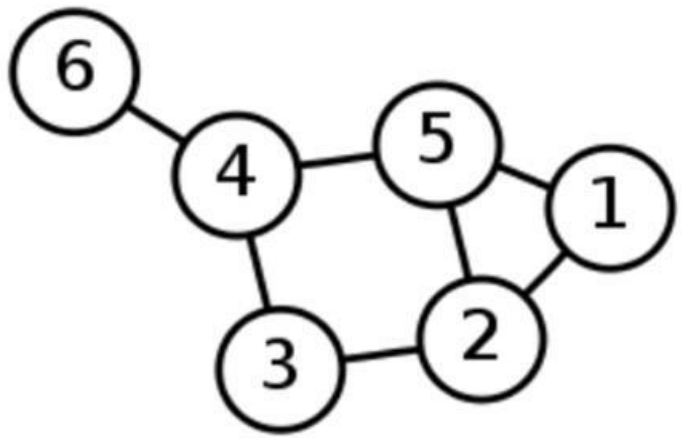
$$r(Sa) = \frac{f(Sa)}{f(S^*)}$$

- The accuracy ratio of approximate solutions to maximization problems is usually computed as

$$r(Sa) = \frac{f(S^*)}{f(Sa)}$$

Vertex Cover Problem

- The vertex cover problem is to find a vertex cover of minimum size in a given undirected graph.
- Such a vertex cover is called an optimal vertex cover.
- This problem is the optimization version of an NP-complete decision problem.
- Definition: Consider a graph G= (V, E) where V and E are accordingly vertex and edges. A vertex cover of an undirected graph is a subset V'⊆ V such that if (u, v) is an edge of G, Then either u ∈ V' or v ∈ V'or both.



vertex covers of size 3, such as {2, 4, 5} and {1, 2, 4}.

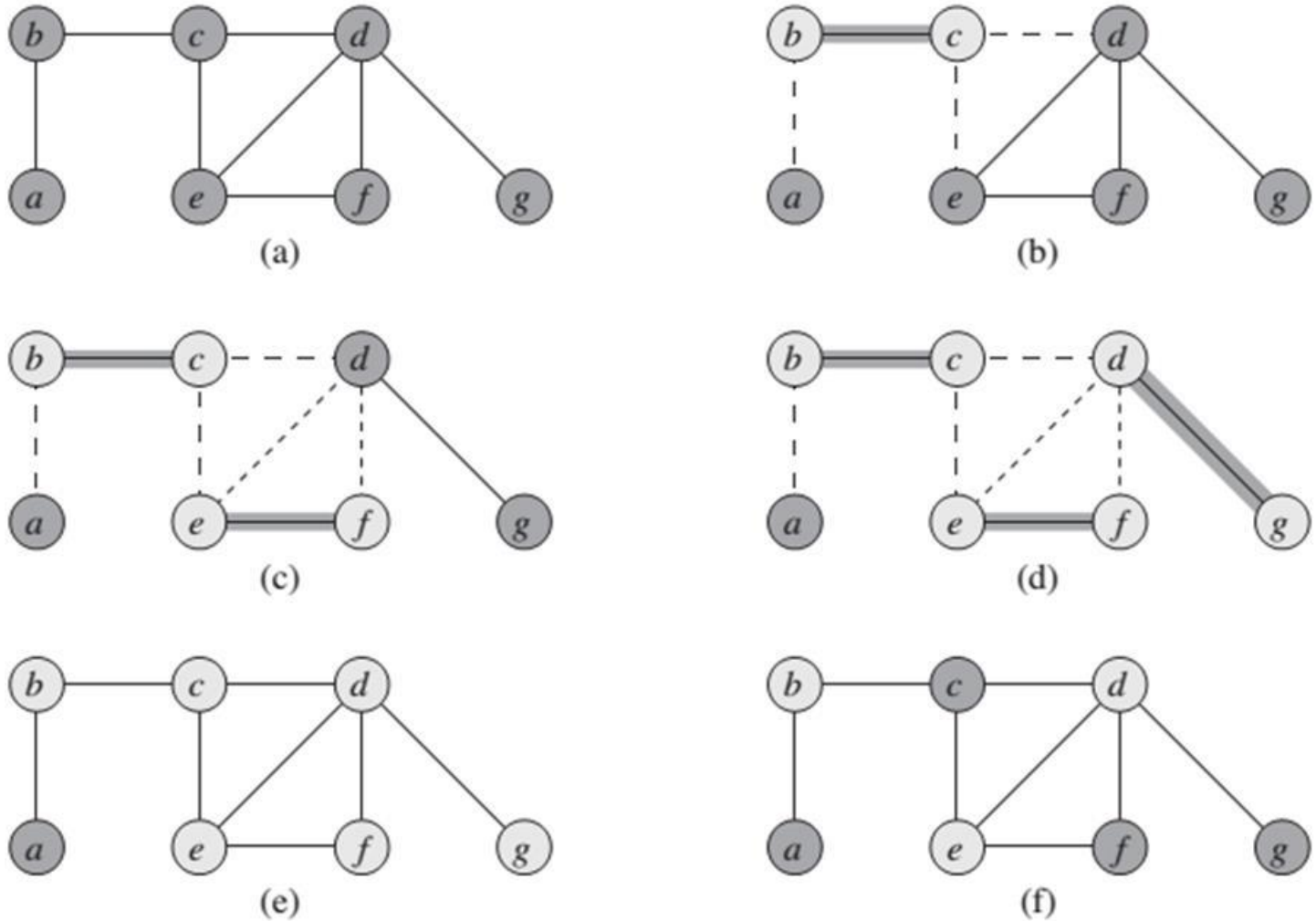
Applications

- Performance of a computer network, for collecting statistics on packets being transmitted. Draw a graph where the vertices are computers/routers, and edges are communication links.
- Installing security cameras covering all the places.

Algorithms for Vertex Cover Problem

- Brute Force
  - All possible subsets – N nodes => O(2^N)
- Approximation Algorithm
- Greedy algorithm
- Dynamic Programming algorithm

Approximation Algorithm



Approximation Algorithm

```
APPROX-VERTEX-COVER(G)
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4       let (u, v) be an arbitrary edge of E'
5       C = C ∪ {u, v}
6       remove from E' every edge incident on either u or v
7  return C
```

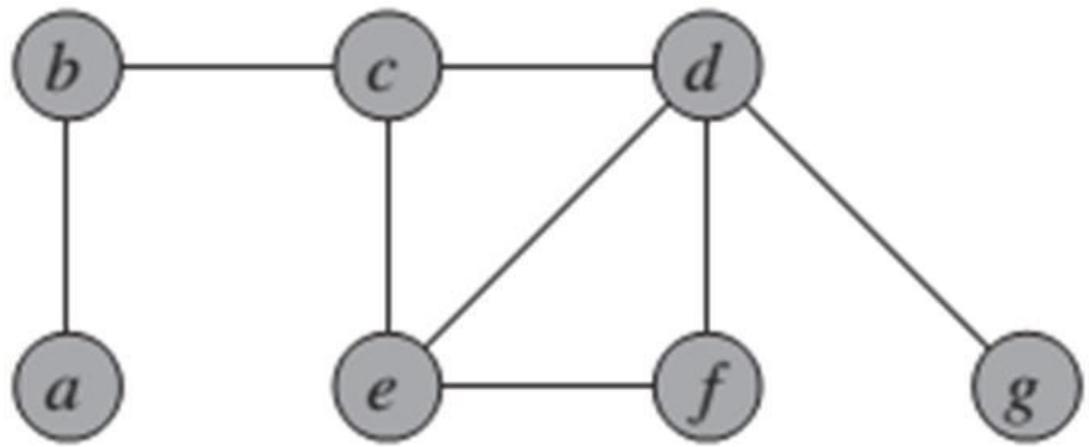
Approximation Algorithm

- The variable C contains the vertex cover being constructed.
- Line 1 initializes C to the empty set. Line 2 sets E’ to be a copy of the edge set G.E of the graph.
- The loop of lines 3–6 repeatedly picks an edge (u, v) from E’, adds its endpoints u and v to C, and deletes all edges in E’ that are covered by either u or v.
- Finally, line 7 returns the vertex cover C.
- The running time of this algorithm O(V + E), using adjacency lists to represent E’.

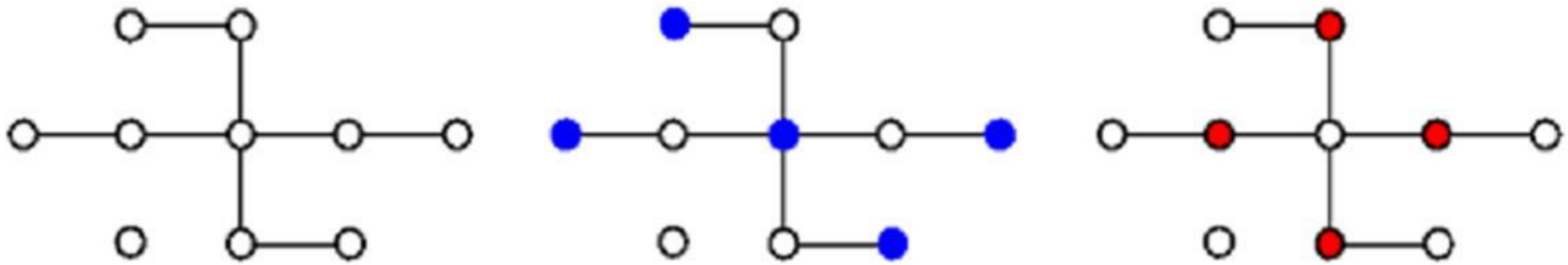
Greedy algorithm

1. $C \leftarrow \emptyset$
2. **while** $E \neq \emptyset$
3. Pick a vertex $v \in V$ of maximum degree in the *current* graph
4. $C \leftarrow C \cup \{v\}$
5. $E \leftarrow E \setminus \{e \in E : v \in e\}$
6. **return** C

Greedy algorithm



Greedy algorithm



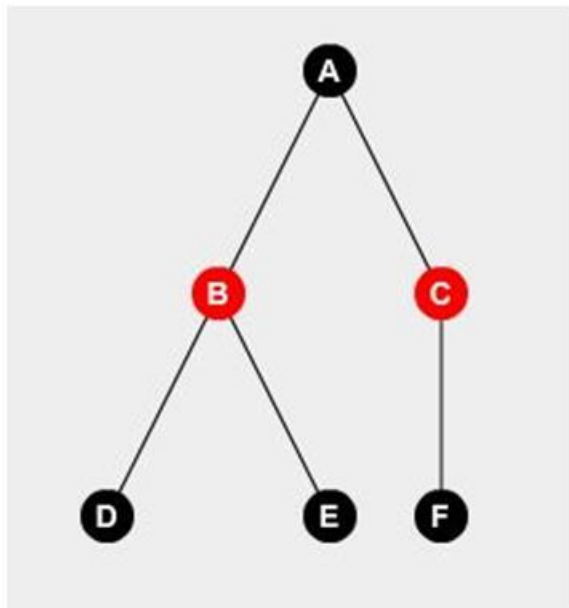
(a) A graph instance

(b) a vertex cover of size 5 obtained by the greedy algorithm

(c) a vertex cover of size 4 optimal solution

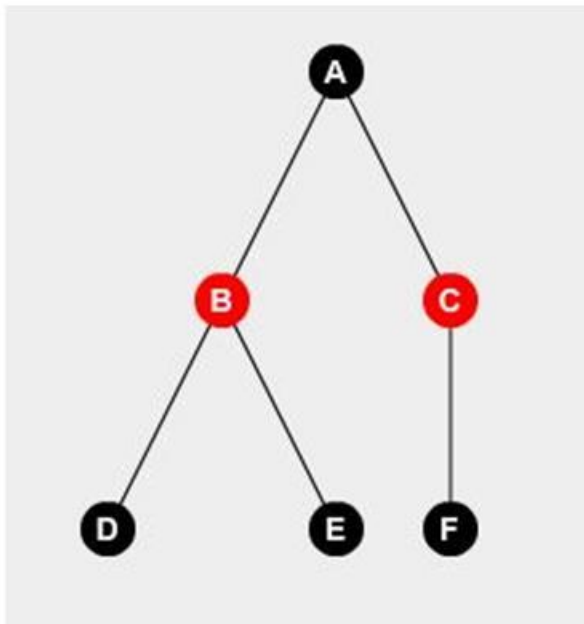
Dynamic Programming algorithm

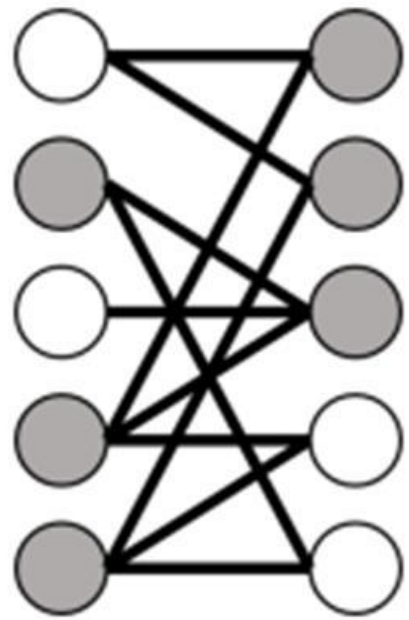
- If the graph was a **Tree**, that means if it had **(n-1)** nodes where **n** is the number of edges and there are no cycle in the graph, we can solve it using dynamic programming.



Dynamic Programming algorithm

- If the graph was a Bipartite Graph or tree, we can solve it using dynamic programming.

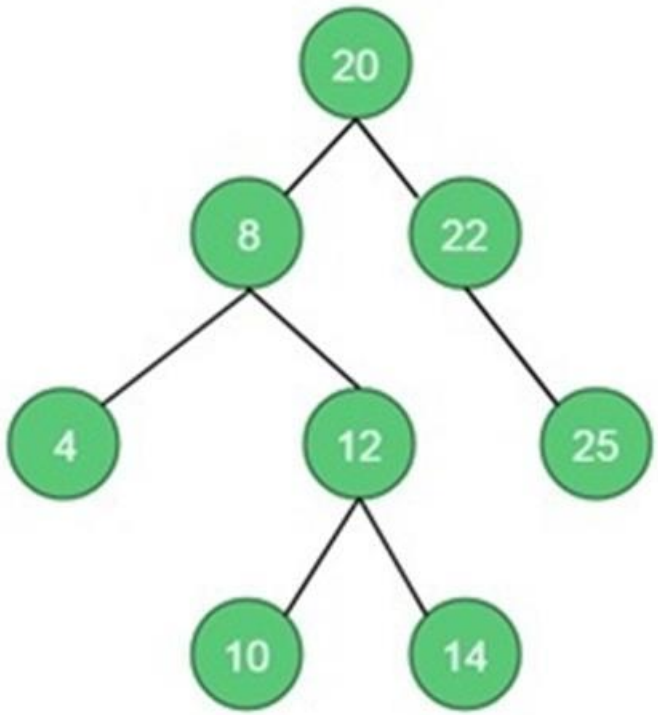


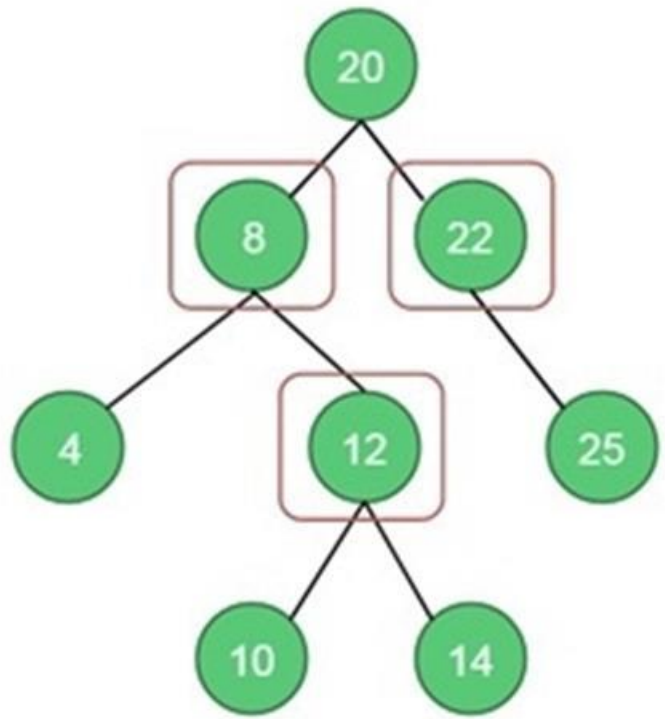


Dynamic Programming algorithm

- The idea of the approach to find minimum vertex cover in a tree is to consider following two possibilities for root and recursively for all the nodes down the root.
  - Root is a part of vertex cover:
    - In this case root covers all children edges. We recursively calculate size of vertex covers for left and right subtrees and add 1 to the result (for root).
  - Root is not a part of vertex cover:
    - In this case, both children of root must be included in vertex cover to cover all root to children edges. We recursively calculate size of vertex covers of all grandchildren and number of children to the result (for two children of root).
- Finally we take the minimum of the two, and return the result.
- The brute force time complexity of this approach is exponential i.e. $O(2^n)$ because we recursively solve the same subproblems many times.

Dynamic Programming algorithm





Dynamic Programming algorithm

- Solve the vertex cover problem in bottom up manner and store the solutions of the sub-problems to calculate the minimum vertex cover of the problem.
- In bottom up manner we start from the leaf nodes, compute its minimum vertex cover and store it to that node.
- Now for the parent node we recursively compute the vertex cover from its children nodes and store it to that node.
- In this approach we calculate the vertex cover of any node only once and reuse the stored value if required.

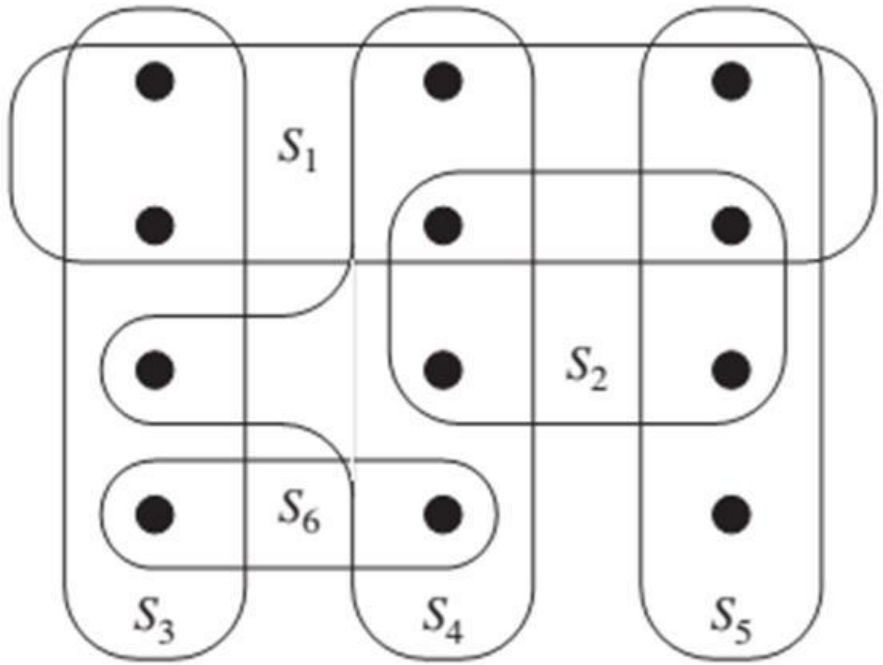
Summary

- The vertex cover (VC) problem belongs to the class of NP-complete graph theoretical problems.
- We are unlikely to find a polynomial-time algorithm for solving vertex-cover problem exactly.
- Minimum vertex cover is one of the Karp‘s 21 diverse combinatorial and graph theoretical problems (Karp, 1972), which were proved to be NP-complete.

The set-covering problem

- The set-covering problem is an optimization problem that models many problems that require resources to be allocated.
- Its corresponding decision problem generalizes the NP-complete vertex-cover problem and is therefore also NP-hard.
- Examine a simple greedy heuristic approach.

The set-covering problem



An instance $(X, \mathcal{F})$ of the set-covering problem, where X consists of the 12 black points and $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$. A minimum-size set cover is $\mathcal{C} = \{S_3, S_4, S_5\}$, with size 3. The greedy algorithm produces a cover of size 4 by selecting either the sets S_1, S_4, S_5, and S_3 or the sets S_1, S_4, S_5, and S_6, in order.

The set-covering problem

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

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

We say that a subset $S \in \mathcal{F}$ ***covers*** its elements. The problem is to find a minimum-size subset $\mathcal{C} \subseteq \mathcal{F}$ whose members cover all of X:

$$X = \bigcup_{S \in \mathcal{C}} S \, . \tag{35.8}$$

A greedy approximation algorithm

```
GREEDY-SET-COVER(X, F)
1  U = X
2  C = ∅
3  while U ≠ ∅
4      select an S ∈ F that maximizes |S ∩ U|
5      U = U − S
6      C = C ∪ {S}
7  return C
```

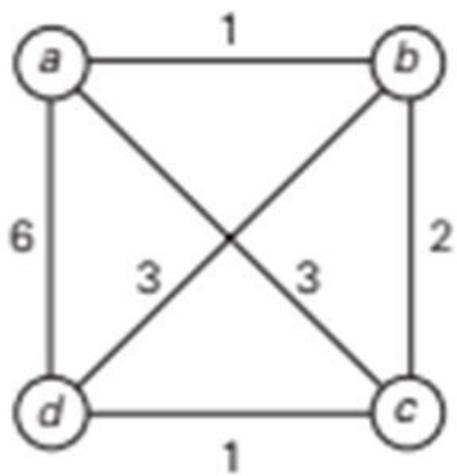
We can easily implement GREEDY-SET-COVER to run in time polynomial in $|X|$ and $|\mathcal{F}|$. Since the number of iterations of the loop on lines 3–6 is bounded from above by $\min(|X|, |\mathcal{F}|)$, and we can implement the loop body to run in time $O(|X|\,|\mathcal{F}|)$, a simple implementation runs in time $O(|X|\,|\mathcal{F}|\min(|X|, |\mathcal{F}|))$.

Greedy Algorithms for the TSP

- **Nearest-neighbor algorithm**
- The following well-known greedy algorithm is based on the ***nearest-neighbor*** heuristic: always go next to the nearest unvisited city.
  - **Step 1** Choose an arbitrary city as the start.
  - **Step 2** Repeat the following operation until all the cities have been visited: go to the unvisited city nearest the one visited last (ties can be broken arbitrarily).
  - **Step 3** Return to the starting city.

Greedy Algorithms for the TSP

- **Nearest-neighbor algorithm**



- **EXAMPLE 1** For the instance represented by the graph in Figure 12.10, with *a* as the starting vertex, the nearest-neighbor algorithm yields the tour (Hamiltonian circuit) s_a: *a* - *b* - *c* - *d* - *a* of length 10.
- The optimal solution, as can be easily checked by exhaustive search, is the tour $s*$: *a* - *b* - *d* - *c* - *a* of length 8.
- Thus, the accuracy ratio of this approximation is

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{10}{8} = 1.25$$

(i.e., tour s_a is 25% longer than the optimal tour $s*$).

Greedy Algorithms for the TSP

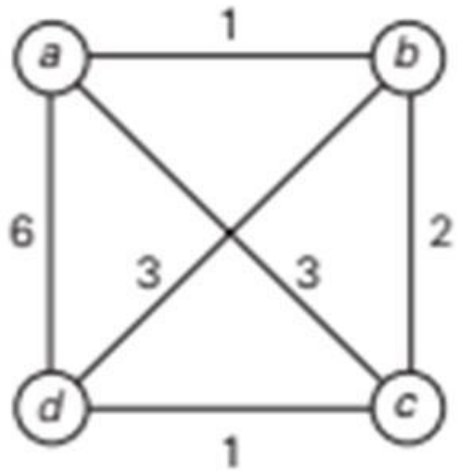
- **Multifragment-heuristic algorithm**
- Another natural greedy algorithm for the traveling salesman problem considers it as the problem of finding a minimum-weight collection of edges in a given complete weighted graph so that all the vertices have degree 2
- **Step 1** Sort the edges in increasing order of their weights. (Ties can be broken arbitrarily.) Initialize the set of tour edges to be constructed to the empty set.

  Step 2 Repeat this step *n* times, where *n* is the number of cities in the instance being solved: add the next edge on the sorted edge list to the set of tour edges, provided this addition does not create a vertex of degree 3 or a cycle of length less than *n*; otherwise, skip the edge.

  Step 3 Return the set of tour edges

Greedy Algorithms for the TSP

- **Multifragment-heuristic algorithm**



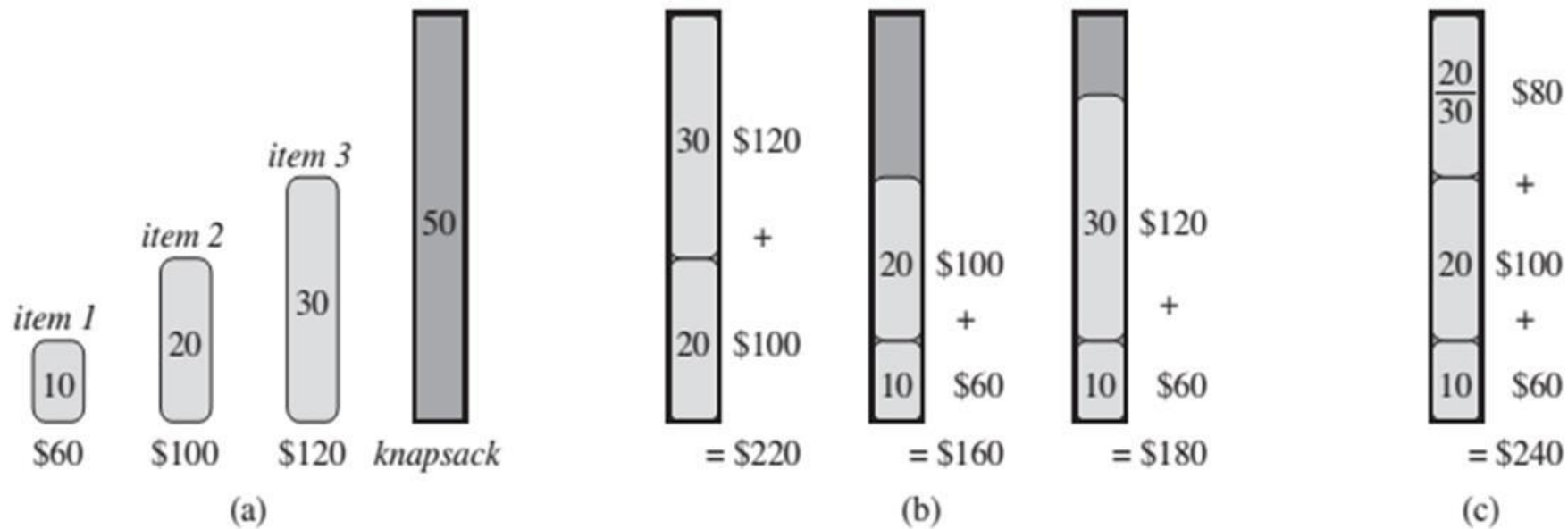
- As an example, applying the algorithm to the graph in Figure yields {*(a, b), (c, d), (b, c), (a, d)*}.
- This set of edges forms the same tour as the one produced by the nearest-neighbor algorithm.
- In general, the multifragment-heuristic algorithm tends to produce significantly better tours than the nearest-neighbor algorithm

Approximation Algorithms for the Knapsack Problem

- **Greedy Algorithms for the Knapsack Problem**
- **Step 1** Compute the value-to-weight ratios $r_i = v_i/w_i$, $i = 1, \ldots, n$, for the items given.
- **Step 2** Sort the items in nonincreasing order of the ratios computed in Step 1. (Ties can be broken arbitrarily.)
- **Step 3** Repeat the following operation until no item is left in the sorted list: if the current item on the list fits into the knapsack, place it in the knapsack and proceed to the next item; otherwise, just proceed to the next item.

Approximation Algorithms for the Knapsack Problem

Let us consider the instance of the knapsack problem with the knapsack capacity and the item information as follows



Approximation Algorithms for the Knapsack Problem

Let us consider the instance of the knapsack problem with the knapsack capacity 10 and the item information as follows

item	weight	value
1	7	$42
2	3	$12
3	4	$40
4	5	$25

item	weight	value	value/weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

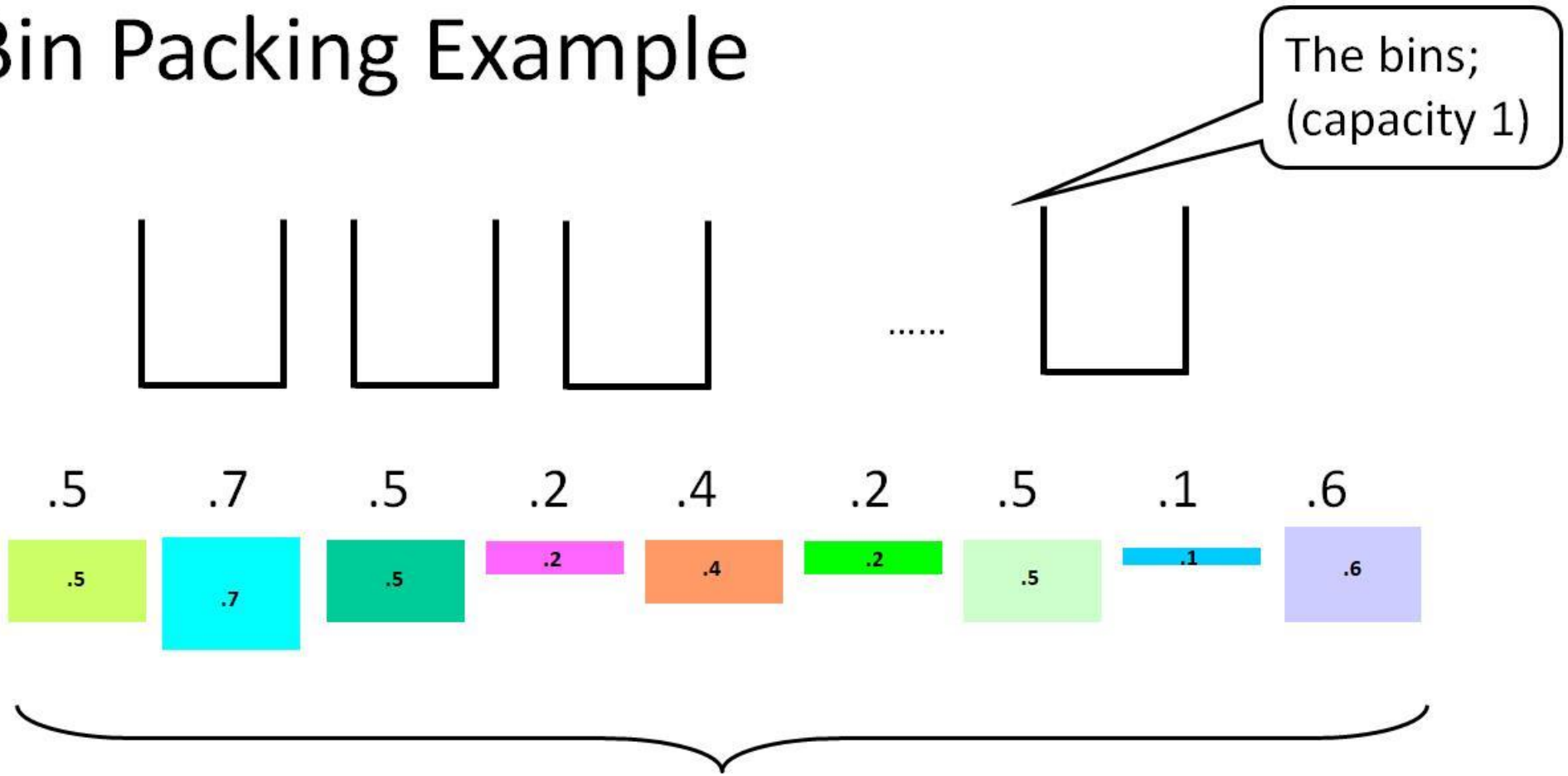
- The greedy algorithm will select the first item of weight 4, skip the next item of weight 7, select the next item of weight 5, and skip the last item of weight 3.
- The solution obtained happens to be optimal for this instance.
- Does this greedy algorithm always yield an optimal solution? The answer, of course, is no:

Bin Packing

- Given n items with sizes s_1, s_2, ..., s_n such that $0 \leq s_i \leq 1$ for $1 \leq i \leq n$, pack them into the fewest number of unit capacity bins.
- Problem is NP-hard (NP-Complete for the decision version).
- There is no known polynomial time algorithm for its solution, and it is conjectured that none exists.

Bin Packing

- Bin Packing Example



Example Applications

Filling recycle bins

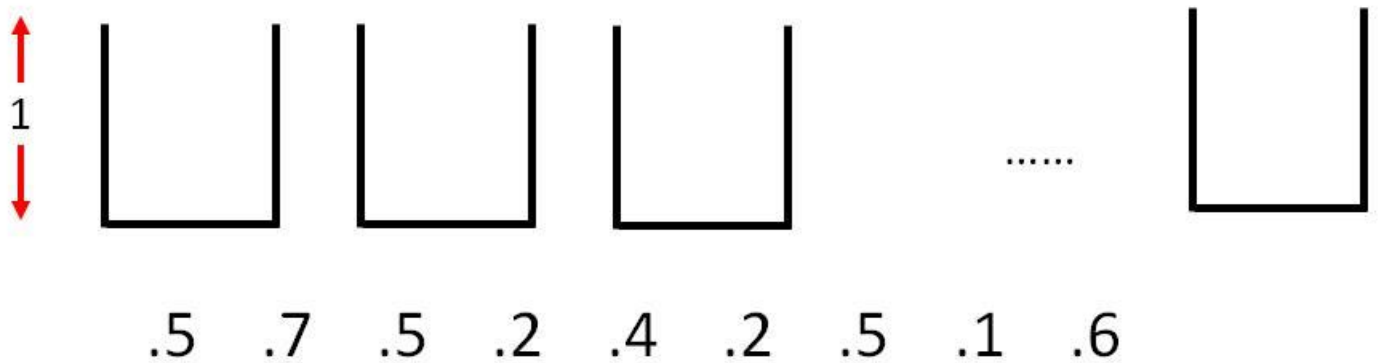
Loading trucks

Historical Application

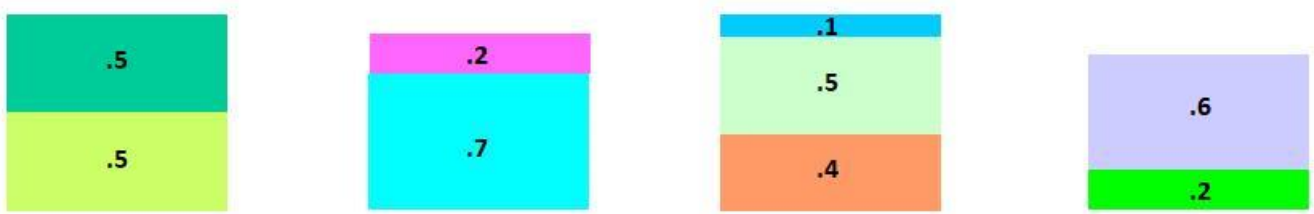
Mix tapes

Bin Packing Optimal Solution

Bin Packing Problem



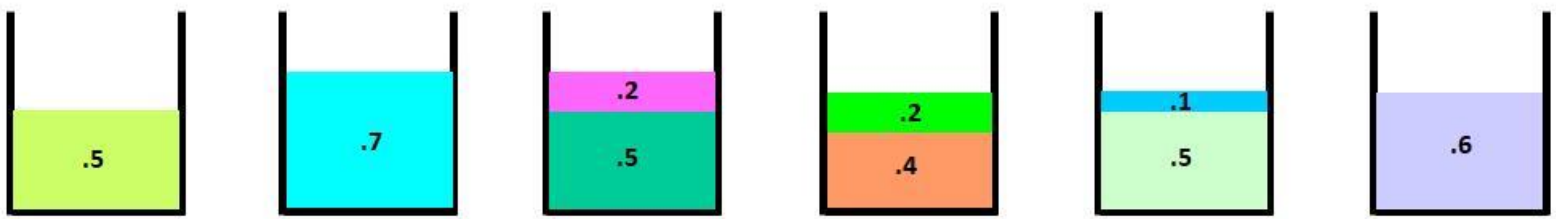
Optimal Packing



$M_{Opt} = 4$

Bin Packing

- Next-Fit (NF) Algorithm
- Check to see if the current item fits in the current bin. If so, then place it there, otherwise start a new bin.



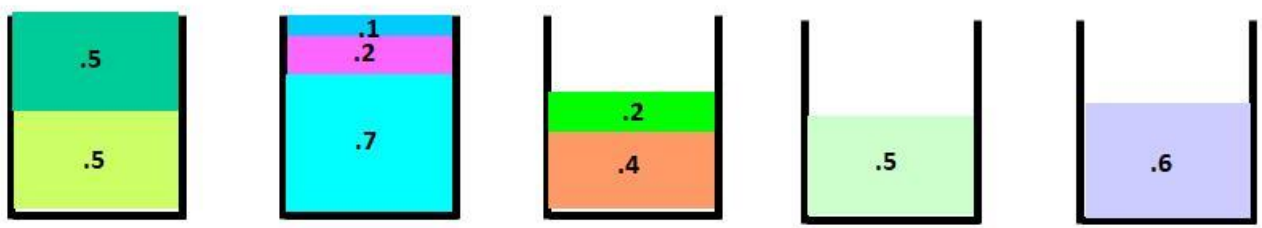
M = 6

Bin Packing

- Next Fit (NF) Approximation Ratio
- Theorem: Let M be the number of bins required to pack a list I of items optimally. Next Fit will use at most 2M bins.
- There exist sequences such that Next Fit uses 2M −2 bins, where M is the number of bins in an optimal solution.

Bin Packing

- First Fit (FF) Algorithm
- Scan the bins in order and place the new item in the first bin that is large enough to hold it.
- A new bin is created only when an item does not fit in the previous bins.



M = 5

Bin Packing

- Running Time for First Fit
- Easily implemented in $O(n^2)$ time
- Can be implemented in O(n log n) time:
  - Idea: Use a balanced search tree with height O(log n).
  - Each node has three values: index of bin, remaining capacity of bin, and best (largest) in all the bins represented by the subtree rooted at the node.
  - The ordering of the tree is by bin index.

Bin Packing

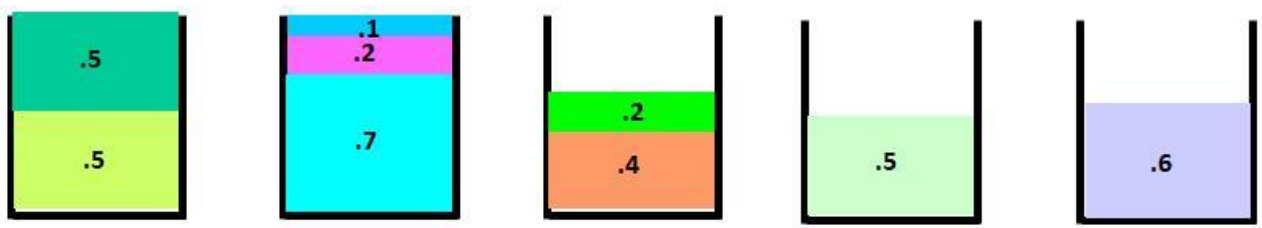
- First Fit Approximation Ratio
- Let M be the optimal number of bins required to pack a list I of items. Then First Fit never uses more than $\lceil 1.7M \rceil$.
- There exist sequences such that First Fit uses 1.6666…(M) bins.

Bin Packing

- Best Fit Algorithm (BF)
- New item is placed in a bin where it fits the tightest. If it does not fit in any bin, then start a new bin.
- Can be implemented in O(n log n) time, by using a balanced binary tree storing bins ordered by remaining capacity.

Bin Packing

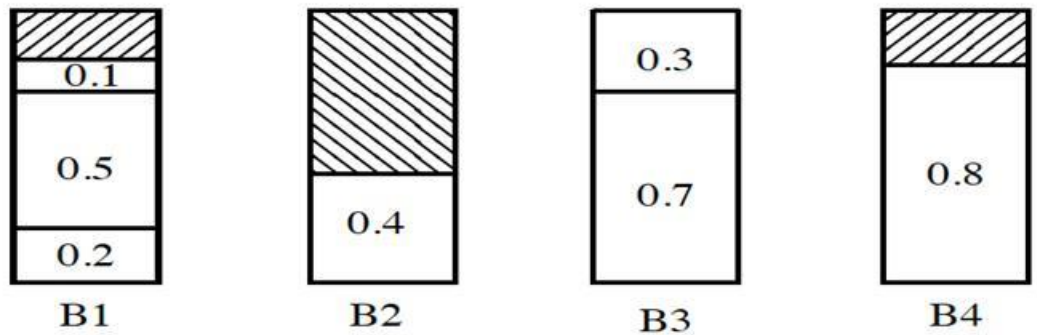
- Best Fit Algorithm (BF)
- New item is placed in a bin where it fits the tightest. If it does not fit in any bin, then start a new bin.
- Can be implemented in O(n log n) time, by using a balanced binary tree storing bins ordered by remaining capacity.



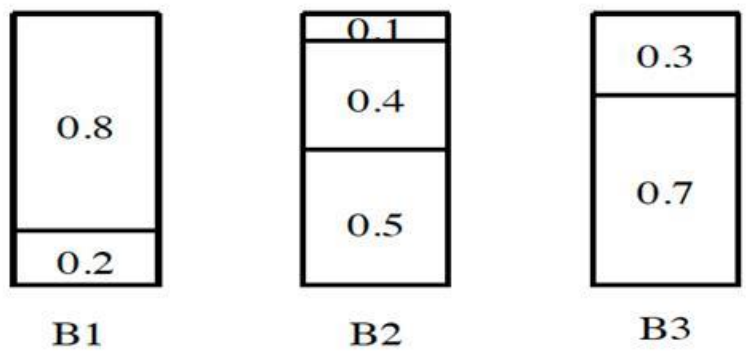
M = 5

Example for Best Fit (BF)

- I = (0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8)



Best Fit



Optimal Packing

Other Heuristics

- First Fit Decreasing (FFD): First order the items by size, from largest to smallest, then run the First Fit Algorithm.
- Best Fit Decreasing (BFD): First order the items by size, from largest to smallest, then run the Best Fit Algorithm.

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1.