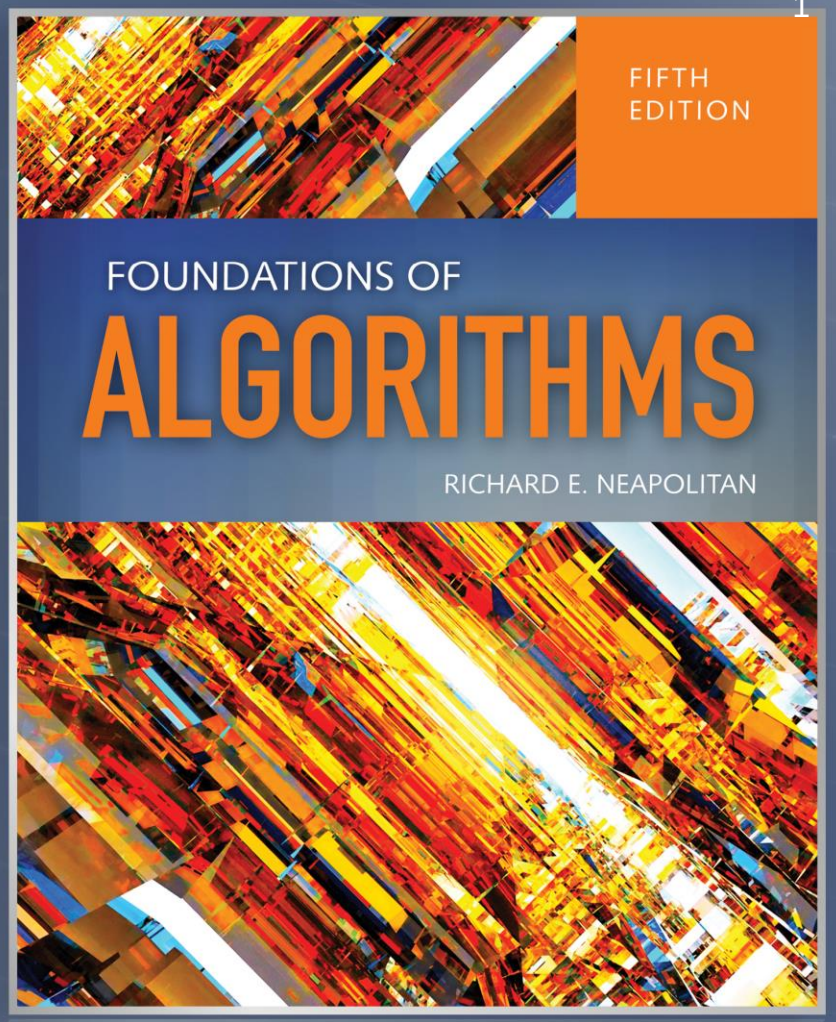
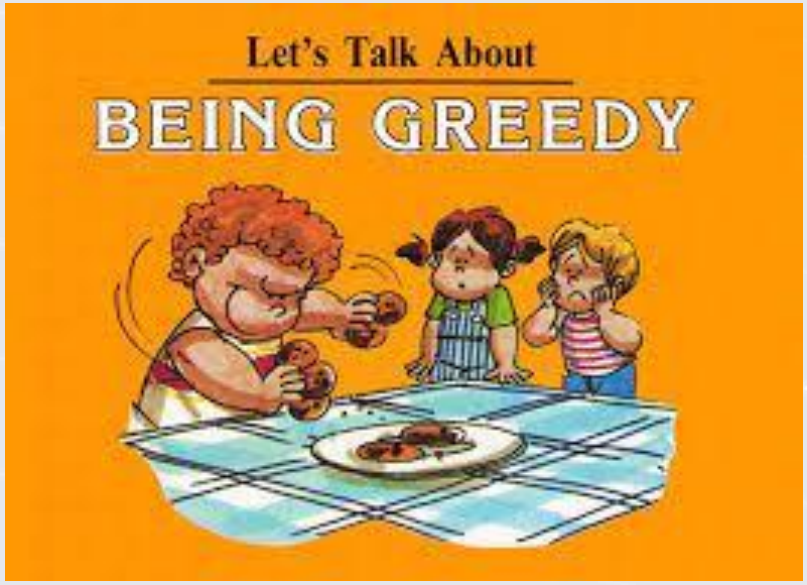
The Greedy Approach

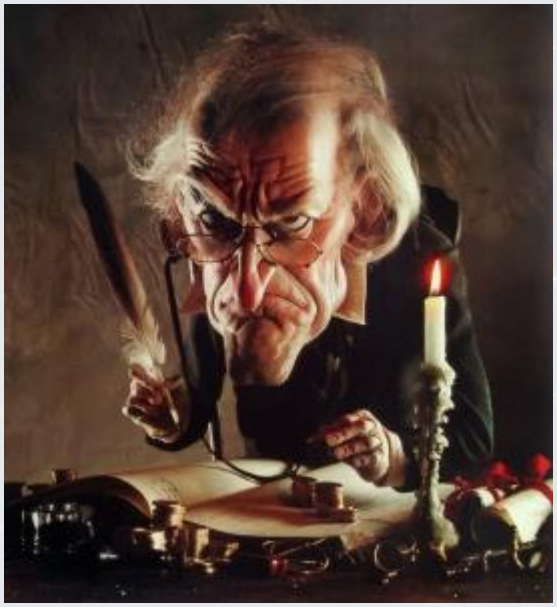
Chapter 4



Objectives

- Describe the Greedy Programming Technique
- Contrast the Greedy and Dynamic Programming approaches to solving problems
- Identify when greedy programming should be used to solve a problem
- Prove/disprove greedy algorithm produces optimal solution
- Solve optimization problems using the greedy approach

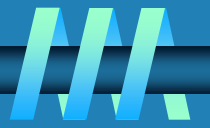




	Greedy Approach	Dynamic Programming Approach
Approach	In Greedy Technique, we make a decision/choice that seems best at a particular moment hoping it would lead to an optimal solution.	In Dynamic Programming, we make decisions/choices by keeping in mind the current state and the solution to the previously solved subproblems in order to calculate the optimal solution.
Optimality	There is no guarantee that we will always arrive at a solution using the Greedy Technique.	There is a guarantee of arriving at a solution because this technique considers all possible cases and arrives at the most optimal solution.
Memory	It is more efficient in terms of memory as it doesn't look back or update the previously calculates values.	It requires a table for updating solutions at each step and looks back at previously calculates values, increasing its memory complexity
Time Complexity	Greedy approach is comparitively faster than Dynamic Programming Approach	Dynamic Programming is generally slower than the Greedy Approach.

- The **goal** is to produce a **globally optimal solution**
- Optimal – must be proven

Greedy Technique

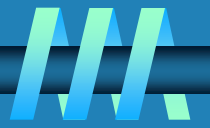


Constructs a solution to an *optimization problem* piece by piece through a sequence of choices that are:

- ***feasible***
- ***locally optimal***
- ***irrevocable***

For some problems, yields an optimal solution for every instance.
For most, does not but can be useful for fast approximations.

Applications of the Greedy Strategy



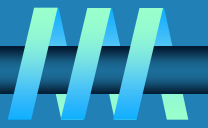
- **Optimal solutions:**
  - **change making for "normal" coin denominations**
  - **minimum spanning tree (MST)**
  - **single-source shortest paths**
  - **simple scheduling problems**
  - **Huffman codes**

- **Approximations:**
  - **traveling salesman problem (TSP)**
  - **knapsack problem**
  - **other combinatorial optimization problems**

Greedy Algorithm

- ***Selection procedure***: Choose the next item to add to the solution set according to the greedy criterion satisfying the locally optimal consideration
- ***Feasibility Check***: Determine if the new set is feasible by determining if it is possible to complete this set to provide a solution to the problem instance
- ***Solution Check:*** Determine whether the new set produced is a solution to the problem instance.

Change-Making Problem



Given unlimited amounts of coins of denominations $d_1 > \ldots > d_m$, give change for amount n with the least number of coins

Example: $d_1 = 25c$, $d_2 = 10c$, $d_3 = 5c$, $d_4 = 1c$ and $n = 48c$

Greedy solution:

Greedy solution is

- optimal for any amount and "normal" set of denominations
- may not be optimal for arbitrary coin denominations

Make Change Algorithm

- while (there are more coins and the instance is not solved)

```
{
  grab the largest coin;
  if (adding the coin makes the change exceed amount owed)
  {
    reject coin;
  } else
    {
       add the coin to the change;
    }
  if (total value of the change equals the amount owed)
  {
    the instance is solved;
  }
}
```

d1 = 25c, d2 =10c, d3 = 5c, d4 = 1c and n = 48c

1. 25 (added 25, sum 25)
2. 10 (rejected 25, sum 25)
3. 10 (added 10, sum 35)
4. 10 (added 10, sum 45)
5. 10 (rejected 10, sum 45)
6. 5 (rejected 5, sum 45)
7. 1 (added 1, sum 46)
8. 1 (added 1, sum 47)
9. 1 added 1, sum 48)

10. result

    25 + 10 + 10 + 1 + 1 + 1

Optimal Solution? Prove

- Set of coins finite – {H,Q,D,N,P}
- Brute force, show greedy algorithm produces an optimal solution to be made for $.01 - $.50
- Any amount of change > $.50 would be a multiple of what was shown (use induction)
- Include a 12-cent coin: coins .50, .25, .12, .10, .05, .01
  - Produce $.16 in change: not optimal

Spanning Tree

- Assume: Connected, weighted, undirected graph G
- **Spanning tree** of a connected graph G: a connected acyclic subgraph of G that includes all of G's vertices
- **Minimum spanning tree (MST)** of a weighted, connected graph G: a spanning tree of G of minimum total weight

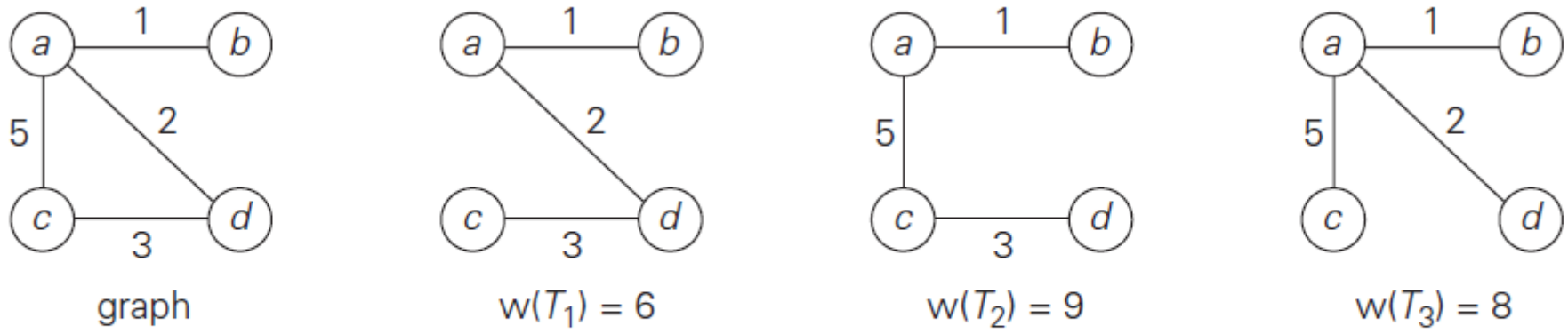
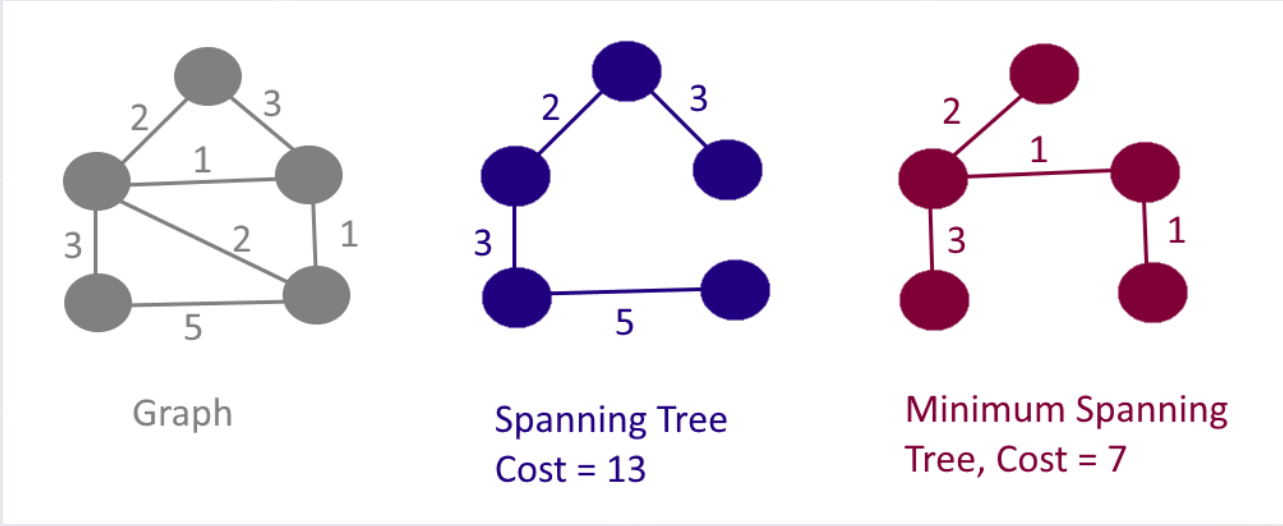


FIGURE 9.2 Graph and its spanning trees, with T_1 being the minimum spanning tree.

Minimum Spanning Tree for G

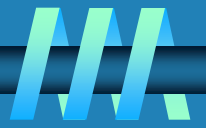
- Let G = (V , E)
- Let T be a spanning tree for G: T = (V, F) where $F \subseteq E$
- Find T such that the sum of the weights of the edges in F is minimal



PRIMS ALGORITHM
VERSUS
KRUSHAL ALGORITHM

PRIMS ALGORITHM	KRUSHAL ALGORITHM
A greedy algorithm that finds a minimum spanning tree for a weighted undirected graph	A minimum spanning tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest
Generates the minimum spanning tree starting from the root vertex	Generates the minimum spanning tree starting from the least weighted edge
Selects the root vertex	Selects the shortest edge
Selects the shortest edge connected to the root vertex	Selects the next shortest edge

Prim's MST algorithm



- Start with tree T_1 consisting of one (any) vertex and "grow" tree one vertex at a time to produce MST through a series of expanding subtrees $T_1, T_2, \ldots, T_n$

- On each iteration, construct T_{i+1} from T_i by adding vertex not in T_i that is closest to those already in T_i (this is a "greedy" step!)

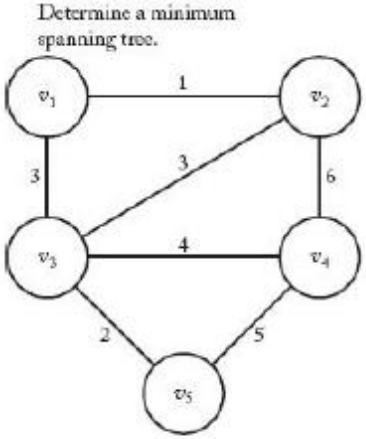
- Stop when all vertices are included

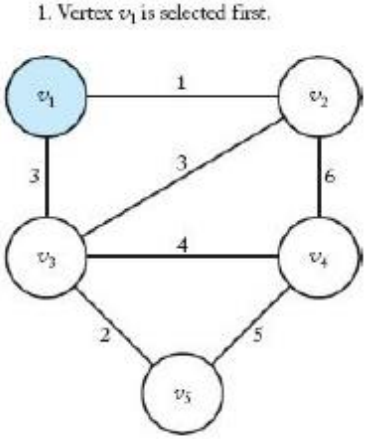
High-Level - Prim's Algorithm

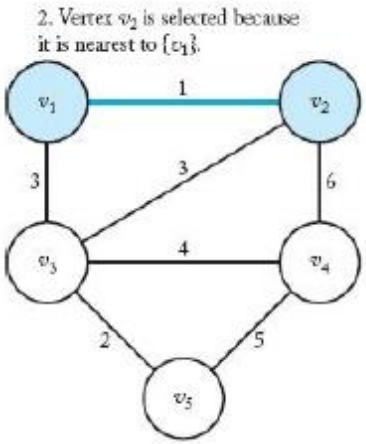

```
F = ∅;                                          // Initialize set of edges
                                                // to empty.
Y = {v1};                                       // Initialize set of vertices to
                                                // contain only the first one.
while (the instance is not solved){

  select a vertex in V − Y that is              // selection procedure and
  nearest to Y;                                 // feasibility check

  add the vertex to Y;
  add the edge to F;

  if (Y == V)                                   // solution check
      the instance is solved;
}
```

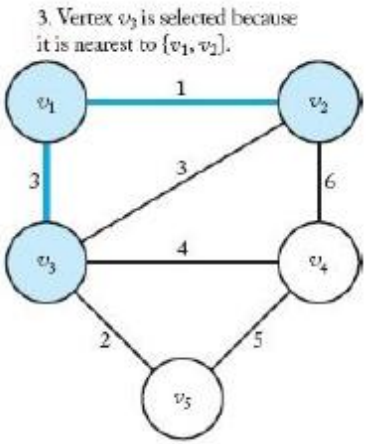
Prim's MST Algorithm

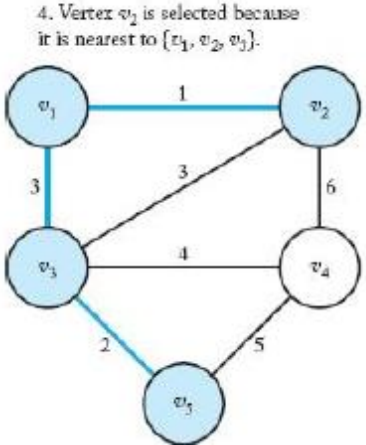
```
void prim (int n,
           const number W[][],
           set_of_edges& F)
{
  index i, vnear;
  number min;
  edge e;
  index nearest[2..n];
  number distance[2..n];

  F = ∅;
  for (i = 2; i <= n; i++){
     nearest[i] = 1;              // For all vertices, initialize v1
     distance[i] = W[1][i];       // to be the nearest vertex in
  }                               // Y and initialize the distance
                                  // from Y to be the weight
                                  // on the edge to v1.
  repeat (n - 1 times){
                                                // Add all n - 1 vertices to
     min = ∞;
     for (i = 2; i <= n; i++)                   // Check each vertex for
        if (0 ≤ distance[i] < min){             // being nearest to Y.
           min = distance[i];
           vnear = i;
        }
     e = edge connecting vertices indexed
         by vnear and nearest[vnear];
     add e to F;
     distance[vnear] = -1;                      // Add vertex indexed by
     for (i = 2; i <= n; i++)                   // vnear to Y.
        if (W[i][vnear] < distance[i]){         // For each vertex not in
           distance[i] = W[i][vnear];           // Y, update its distance
           nearest[i] = vnear;                  // from Y.
        }
  }
}
```

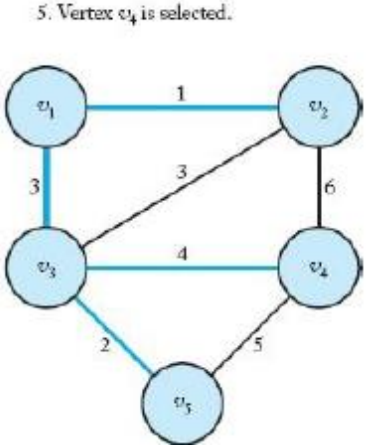












Every-Case Time Complexity of Prim's Algorithm 4.1

- Input Size: n (the number of vertices)
- Basic Operation: Two loops with n – 1 iterations inside a repeat loop

- Proof by induction that this construction actually yields MST
- Needs priority queue for locating the closest fringe vertex
- Time complexity:

$$T(n) = 2(n - 1)(n - 1) \in \theta(n^2)$$

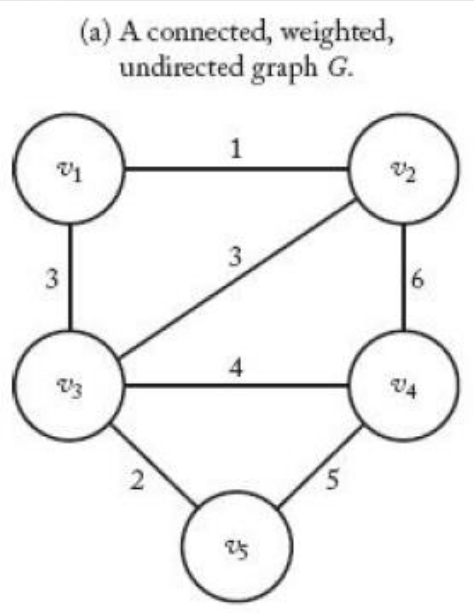
- Efficiency
  - $O(n^2)$ for weight matrix representation of graph and array implementation of priority queue
  - O(m log n) for adjacency list representation of graph with n vertices and m edges and min-heap implementation of priority queue

Spanning Tree Produced by Prim's Algorithm **Minimal**?

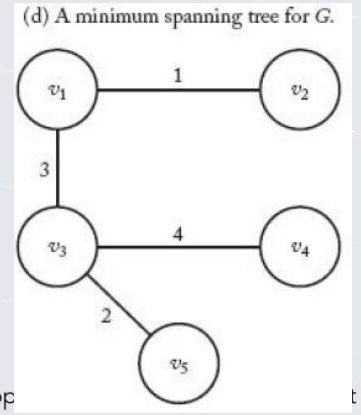
- **Dynamic Programming Algorithm** – show principle of optimality applies
- **Greedy Algorithm** – easier to develop – **must formally prove optimal solution always produced**
- Two parts to proof:
  - Lemma 4.1
  - Theorem 4.1

Promising

- Undirected graph G = (V, E)
- Subset F $\subseteq$ E
- F is called *promising* if edges can be added to it to form a **minimum spanning tree**



Subset {(v1, v2), (v1, v3)} is promising
Subset {(v2, v4)} is not promising



Lemma 4.1

Let:

G = (V, E) connected weighted, undirected graph

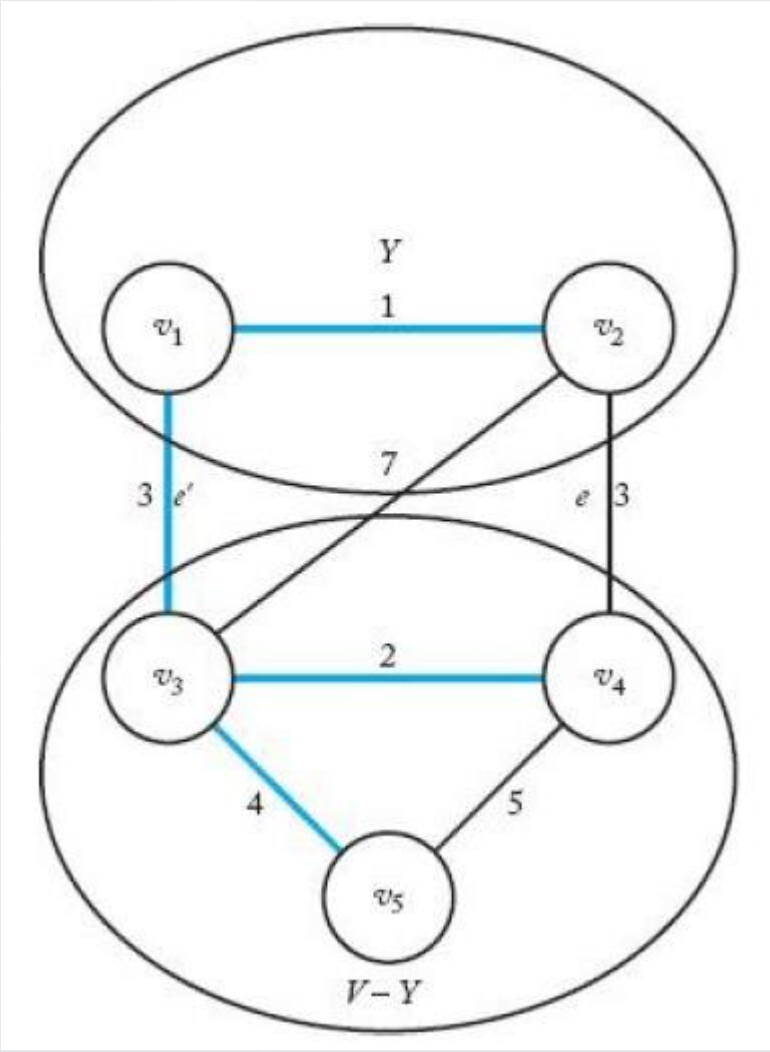
F $\subseteq$ E be promising

Y $\subseteq$ V be the set of vertices connected by edges in F

If e is a minimum edge connecting some $v_y \in$ Y to $v_x \in$ V-Y, then F $\cup$ {e} is promising

Proof Lemma 4.1

- F is promising – must be a minimum spanning tree (v, F') such that $F \subseteq F'$
- if $e \in F'$, $F \cup \{e\} \subseteq F'$ => $F \cup \{e\}$ is promising (proof complete)
- If $e \notin F'$, $F' \cup \{e\}$ must contain a cycle containing e => there must be another $e' \in F'$ in the cycle connecting some $v_x \in Y$ to $v_y \in V - Y$
- Cycle disappears if $F' \cup \{e\} - \{e'\}$ => spanning
- Since e is minimum, e <= e' => $F' \cup \{e\} - \{e'\}$ must be a spanning tree
- $F \cup \{e\} \subseteq F' \cup \{e'\} - \{e'\}$
- Since F connects only vertices in Y, $e' \notin F$
- i.e. e was selected
  - adding e does not create a cycle => **$F \cup \{e\}$ is promising**



Theorem 4.1 – Proof by Induction

Prim's Algorithm always produces a minimum spanning tree

- Induction Base: F = empty set is promising
- Induction Hypothesis: Assume after i iterations of the repeat loop, F is promising
- Induction Step: Show F $\cup$ {e} is promising where e is the edge selected in the next iteration
- Since e was selected, e is a minimum weight edge connecting a vertex in Y to a vertex in V – Y
- By Lemma 4.1, F $\cup$ {e} is promising

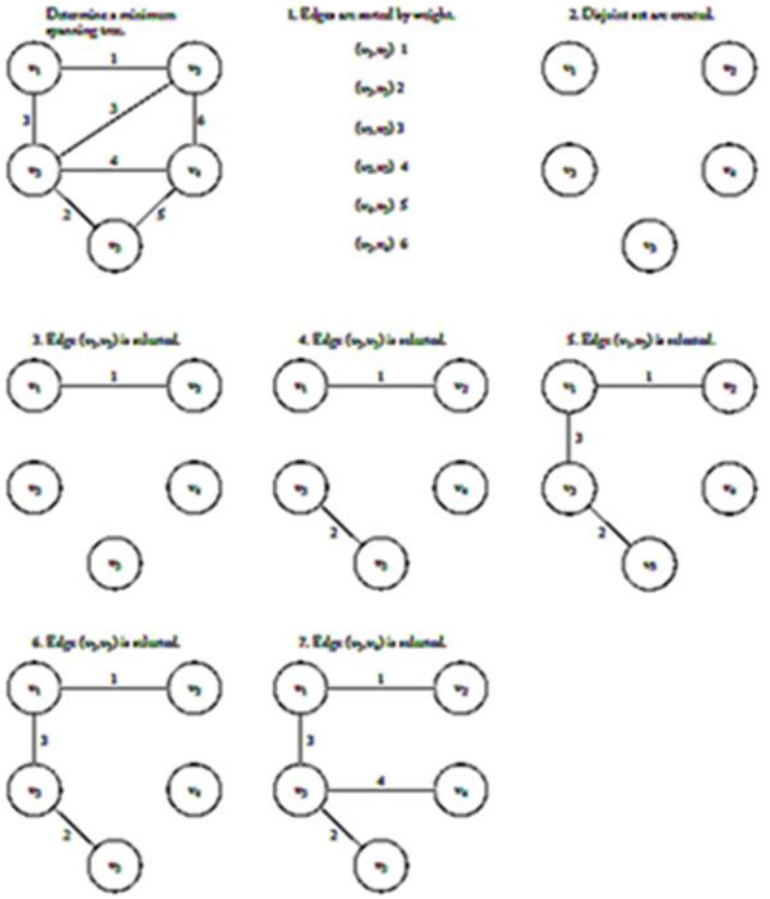
Kruskal's Minimum Spanning Tree Algorithm

- Sort the edges in nondecreasing order of lengths
- "Grow" tree one edge at a time to produce MST through a series of expanding forests F1, F2, …, Fn-1
- On each iteration, add the next edge on the sorted list unless this would create a cycle. (If it would, skip the edge.)

High-Level Kruskal's MST Algorithm

```
F = ∅;                                          // Initialize set of
                                                // edges to empty.
create disjoint subsets of V, one for each
vertex and containing only that vertex;

sort the edges in E in nondecreasing order;

while (the instance is not solved){

  select next edge;                             // selection procedure

  if (the edge connects two vertices in         // feasibility check
                    disjoint subsets){
      merge the subsets;
      add the edge to F;
  }
  if (all the subsets are merged)               // solution check
     the instance is solved;
}
```

```
void kruskal (int n, int m,
              set_of_edges E,
              set_of_edges& F)
{
  index i, j;
  set_pointer p, q;
  edge e;
  Sort the m edges in E by weight in nondecreasing order;
  F = ∅;
  initial(n);                    // Initialize n disjoint s
  while (number of edges in F is less than n - 1){
    e = edge with least weight not yet considered;
    i, j = indices of vertices connected by e;
    p = find(i);
    q = find(j);
    if (! equal(p, q)){
      merge(p, q);
      add e to F;
    }
  }
}
```



- initial(n) ∈ θ(n)
- p = find(i) sets p to point at the set containing index i
  - find ∈ θ(lg m) where m is the depth of the tree representing the disjoint data sets
- merge(p,q) merges 2 sets into 1 set
  - merge ∈ θ(c) where c is a constant
- equal(p,q) where p and q point to sets returns true p and q point to the same set
  - equal ∈ θ(c) where c is a constant

Worst-case Time Complexity Kruskal

- Basic operation: a comparison instruction
- Input size: n, number of vertices
  m, number of edges

3 considerations of Kruskal

1. Time to sort edges: $W(m) \in \theta(m \lg m)$
2. Time in the while loop $W(m) \in \theta(m \lg m)$
3. Time to initialize n disjoint data sets: $T(n) \in \theta(n)$

while loop: manipulation of disjoint data sets

- Worst case, every edge is considered

  $W(m,n) \in \theta(m \lg m)$

To connect n nodes requires at least n-1 edges:

$m >= n-1$

G fully connected $m = n(n-1)/2 \in \theta(n^2)$

$W(m,n) \in \theta(n^2 \lg n^2) = \theta(n^2 2\lg n) = \theta(n^2 \lg n)$

Spanning Tree Produced by Kruskal's Algorithm Minimal?

- Lemma 4.2
- Theorem 4.2

Lemma 4.2

- Let G = (V, E) is a connected, weighted, undirected graph
- F is a promising subset of E
- Let e be an edge of minimum weight in E – F

then

- F ∪ {e} has no cycles
- F ∪ {e} is promising

Proof of Lemma 4.2 is similar to the proof of Lemma 4.1

Theorem 4.2

Kruskal's Algorithm always produces a minimum spanning tree

Proof: Use induction to show the set F is promising after each iteration of the repeat loop

- Induction base: F = ∅ empty set is promising
- Induction hypothesis: assume after the ith iteration of the repeat loop, the set of edges F selected so far is promising
- Induction step: Show F ∪ {e} is promising where is the selected edge in the (i+1) th iteration
- e selected in the next iteration, it has a minimum weight
- e connects vertices in disjoint sets
- Because e is selected, it is minimum and connects two vertices in disjoint sets
- By Lemma 4.2 F ∪ {e} is promising

Prim vs Kruskal

- **Sparse graph**
  - m close to n – 1
  - Kruskal θ(n lg n) faster than Prim
- **Highly connected graph**
  - Kruskal θ(n^2 lg n)
  - Prim's faster

Shortest paths – Dijkstra's algorithm

Single Source Shortest Paths Problem: Given a weighted connected graph G, find shortest paths from source vertex *s* to each of the other vertices

Dijkstra's algorithm: Similar to Prim's MST algorithm, with a different way of computing numerical labels: Among vertices not already in the tree, it finds vertex *u* with the smallest sum

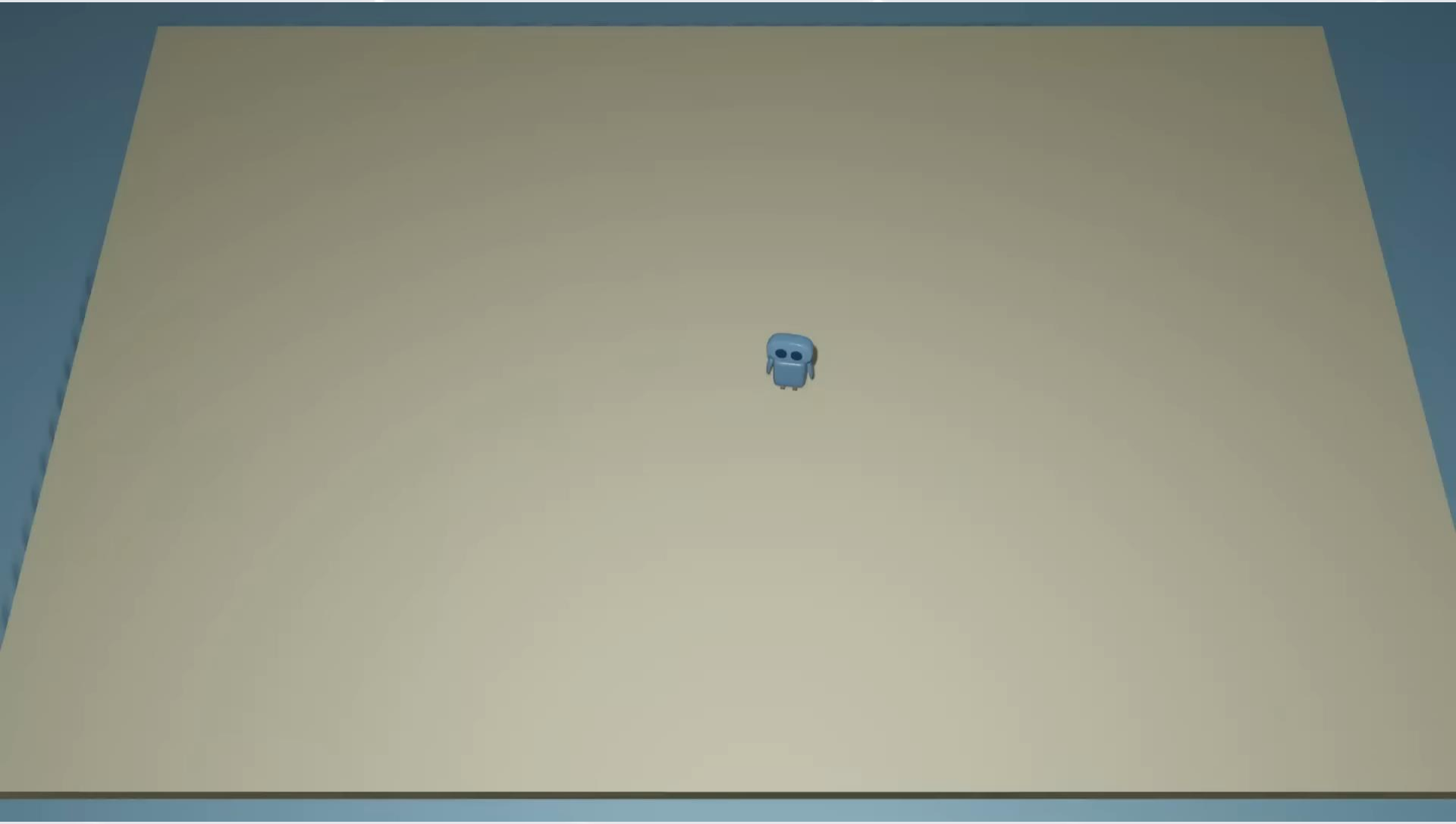
$$d_v + w(v,u)$$

where

v is a vertex for which shortest path has been already found on preceding iterations (such vertices form a tree)

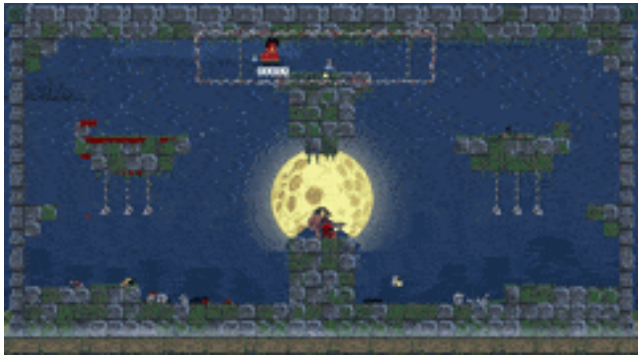
d_v is the length of the shortest path form source to *v*

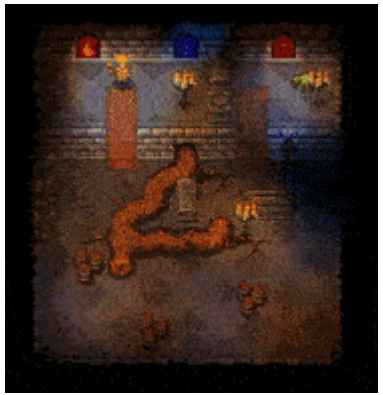
$w(v,u)$ is the length (weight) of edge from *v* to *u*




WIRED


GamerGuppy
2x speed
Concurrent Dijkstra Algorithm
-Understands 3d terrain
-Configure max radius of square
-Highly fast closeby, slow far away in open spaces





Notes on Dijkstra's algorithm

- Doesn't work for graphs with negative weights
- Applicable to both undirected and directed graphs
- Efficiency
  - $O(|V|^2)$ for graphs represented by weight matrix and array implementation of priority queue
  - $O(|E|\log|V|)$ for graphs represented by adj. lists and min-heap implementation of priority queue
- Don't mix up Dijkstra's algorithm with Prim's algorithm!

Coding Problem

Coding: assignment of bit strings to alphabet characters

Codewords: bit strings assigned for characters of alphabet

Two types of codes:

- *fixed-length encoding* (e.g., ASCII)
- *variable-length encoding* (e,g., Morse code)

Prefix-free codes: no codeword is a prefix of another codeword

Problem: If frequencies of the character occurrences are known, what is the best binary prefix-free code?

Huffman codes

- Any binary tree with edges labeled with 0's and 1's yields a prefix-free code of characters assigned to its leaves

- Optimal binary tree minimizing the expected (weighted average) length of a codeword can be constructed as follows

Huffman's algorithm

Initialize *n* one-node trees with alphabet characters and the tree weights with their frequencies.

Repeat the following step *n*-1 times: join two binary trees with smallest weights into one (as left and right subtrees) and make its weight equal the sum of the weights of the two trees.

Mark edges leading to left and right subtrees with 0's and 1's, respectively.

Example

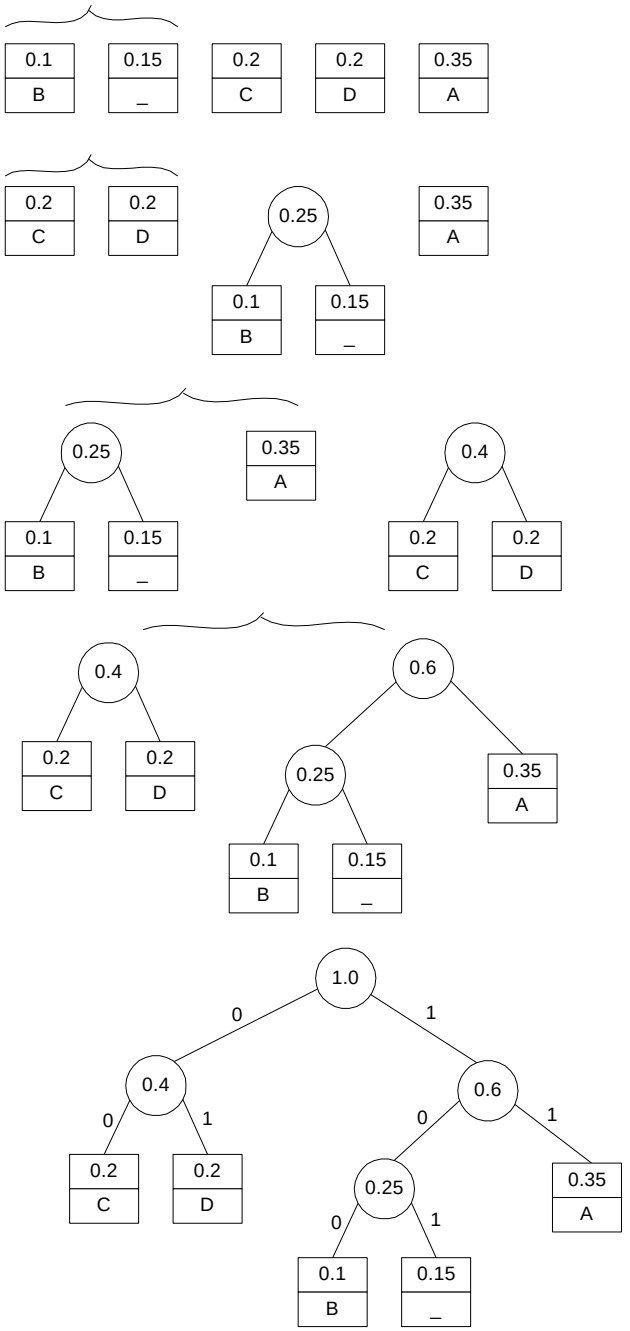
character	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

codeword 11 100 00 01 101

average bits per character: 2.25

for fixed-length encoding: 3

compression ratio: (3-2.25)/3*100% = 25%



Greedy vs Dynamic

- Both solve optimization problems
- Shortest Path
  - Floyd – all pairs dynamic
  - Dijkstra – single source greedy
- Greedy algorithms usually simpler
- Greedy algorithms do not always produce optimal solution – must formally prove
- Dynamic Programming – show principle of optimality applies

0-1 Knapsack Problem

- Thief breaks into jewelry store carrying a knapsack in which to place stolen items
- Knapsack has a weight capacity W
- Knapsack will break if weight of stolen items exceeds W
- Each item has a value
- Thief's dilemma is to maximize the total value of items stolen while not exceeding the total weight capacity W

0-1 Knapsack Problem

Suppose there are n items. Let

$$S = \{item_1, item_2, ..., item_n\}$$
$$w_i = \text{ weight of } item_i$$
$$p_i = \text{ profit of } item_i$$
$$W = \text{ maximum weight the knapsack can hold,}$$

where w_i, p_i, and W are positive integers. Determine a subset A of S such that

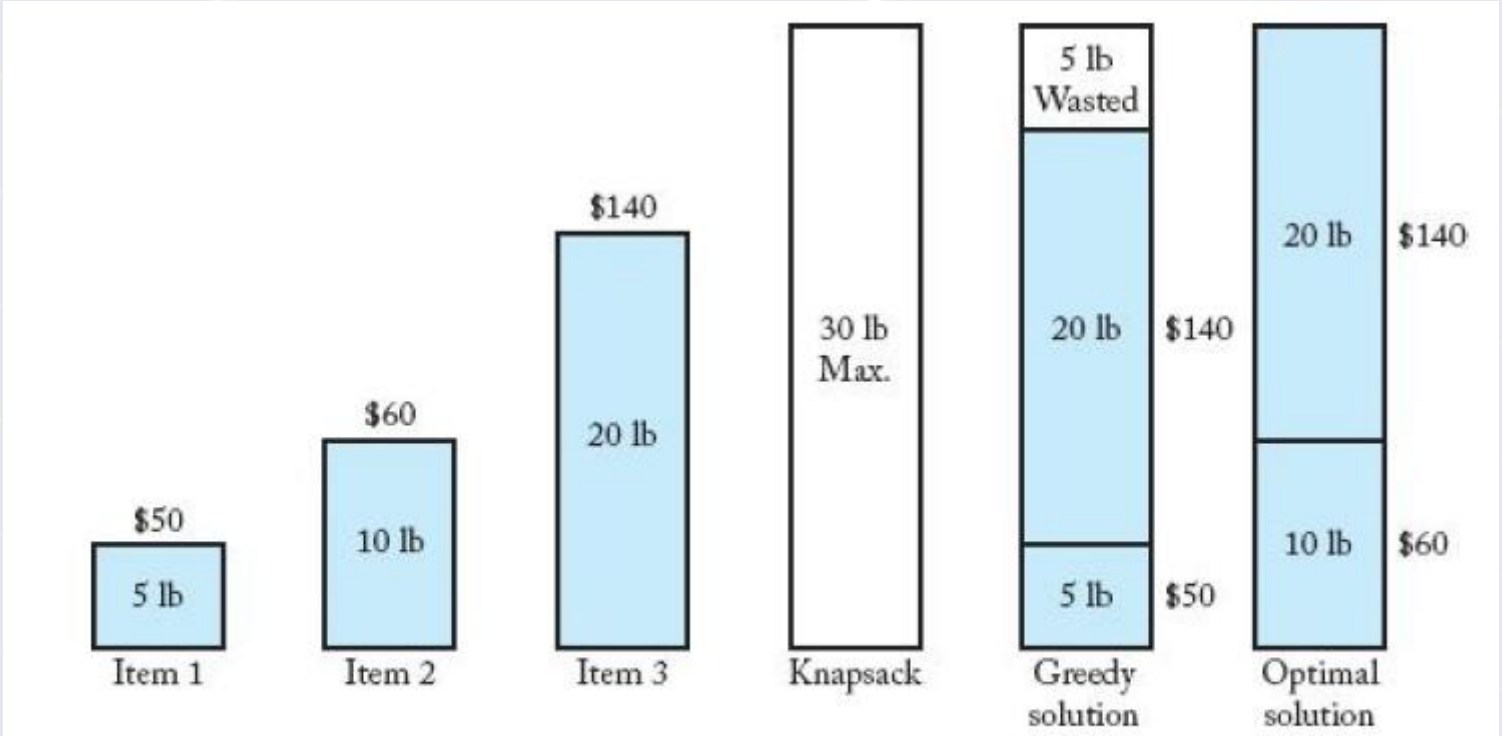
$$\sum_{item_i \in A} p_i \quad \text{is maximized subject to} \quad \sum_{item_i \in A} w_i \leq W.$$

Brute Force Solution

- Consider all subsets of the n items
- Discard subsets whose total weight exceeds W
- Of the remaining, take the one with maximum profit
- 2^n subsets of a set containing n items

Greedy Strategy

- Steal items with the largest profit first – stealing in non-increasing order according to profit
- It Can easily be shown by example **greedy strategy does not always produce an optimal solution**



$$item_1 : \frac{\$50}{5} = \$10 \qquad item_2 : \frac{\$60}{10} = \$6 \qquad item_3 : \frac{\$140}{20} = \$7.$$