Greedy & Dynamic Programming

Learning Objectives

1. **Understand and apply the Greedy approach to solving problems**
   - Prim's Algorithm (find minimum spanning tree)
   - Dijkstra's Algorithm (find shortest path distances)
2. **Understand and apply Dynamic Programming techniques to solving problems**
   - Knapsack Problem

Agenda

1. **Greedy Approach**
   - Prim's Algorithm (minimum spanning tree)
   - Dijkstra's Algorithm (shortest path distance)
2. **Dynamic Programming**
   - Knapsack Problem

1. Greedy Approach

Greedy Algorithms

- Greedy Algorithms build up a solution piece by piece, always choosing the next piece that offers the most immediate and obvious benefit.

- Sometimes such an approach can be lead to an inferior solution, but in other cases it can lead to a simple and optimal solution.

1a. Prim's Algorithm

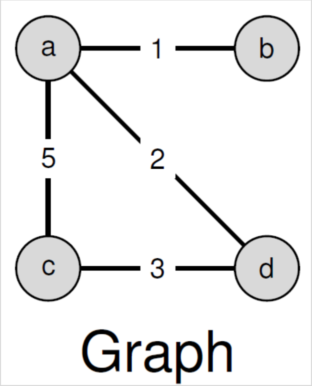
Spanning Tree Problem

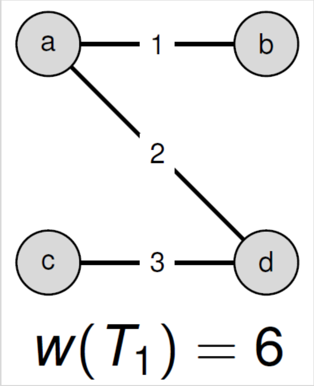
A spanning tree of a connected graph is a connected acyclic subgraph (i.e., a tree) which contains

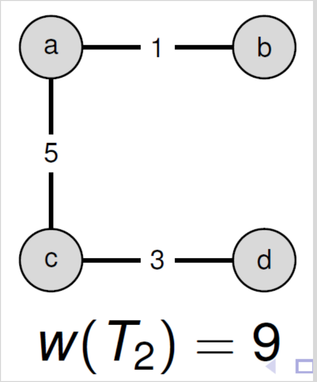
- all the vertices of the graph, and
- a subset of edges from the original graph.

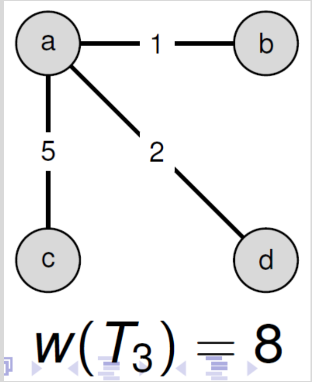
Minimum Spanning Tree Problem

A minimum spanning tree of a weighted connected graph is the spanning tree of the smallest total weight (sum of the weights on all of the tree's edges).









Applications of Minimum Spanning Tree

- Designing networks (phones, computers etc.): Want to connect up a series of offices with telephone or wired lines, but want to minimise cost.
- Approximate solutions to hard problems: travelling salesman
    - “Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city"

Prim’s Algorithm – Sketch

Prim’s Algorithm is one approach to find minimum spanning tree.

Idea: Select one vertex at a time and add to tree.

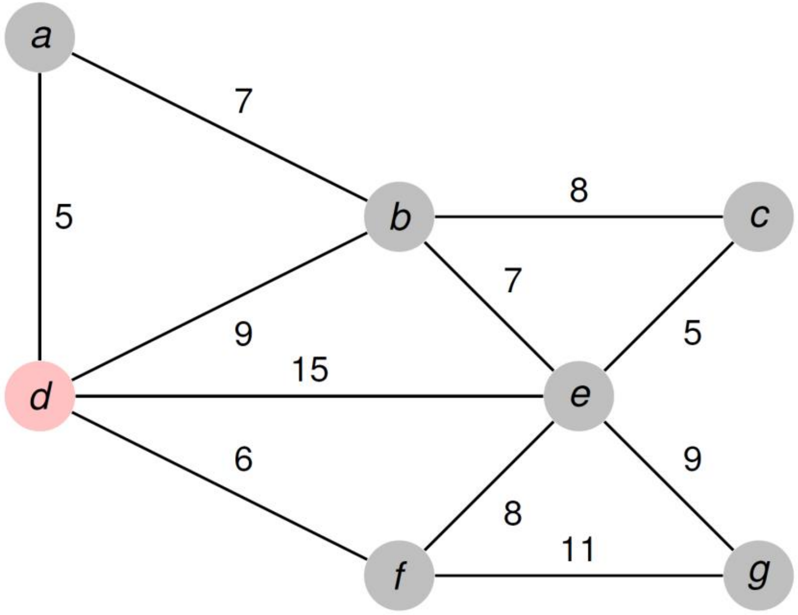
1. Start with one randomly selected vertex and add this to tree.
2. Then at each iteration, add a neighbouring vertex to the tree that has minimum edge weight to one of the vertices in the current tree. It must not be in the tree.
3. Use a min priority queue to quickly find this neighbouring vertex with minimum edge weight (*in literature, the neighbour set is sometimes called the frontier set*).

Prim's Algorithm – Sketch

4. When adding, we may need to update the smallest edge weight to a vertex in neighbour set, as there may be a smallest edge weight from updated tree to new neighbour set.

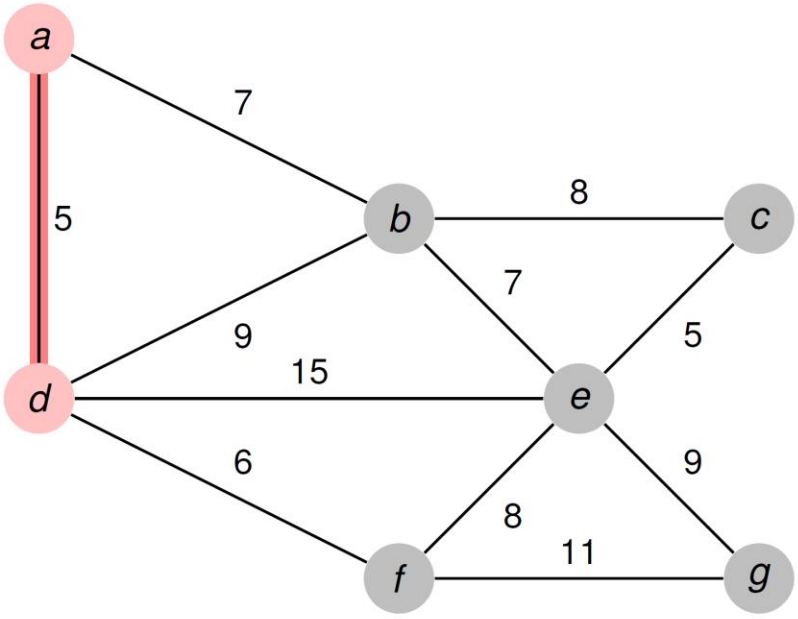
5. When all vertices added to tree, we are done.

Prim's Algorithm – Example



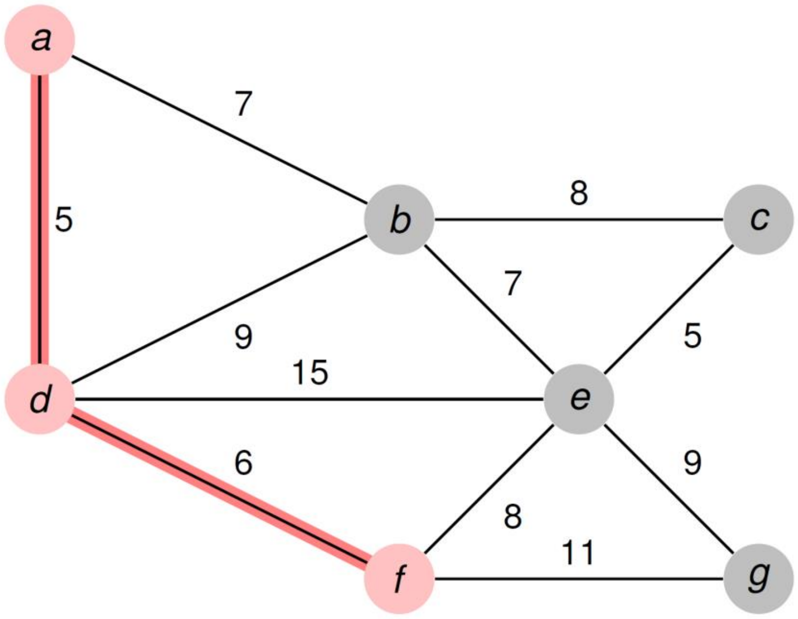
$V_T = \{d\}, PQ = \{(a, 5), (f, 6), (b, 9), (e, 15)\}$

Prim's Algorithm – Example



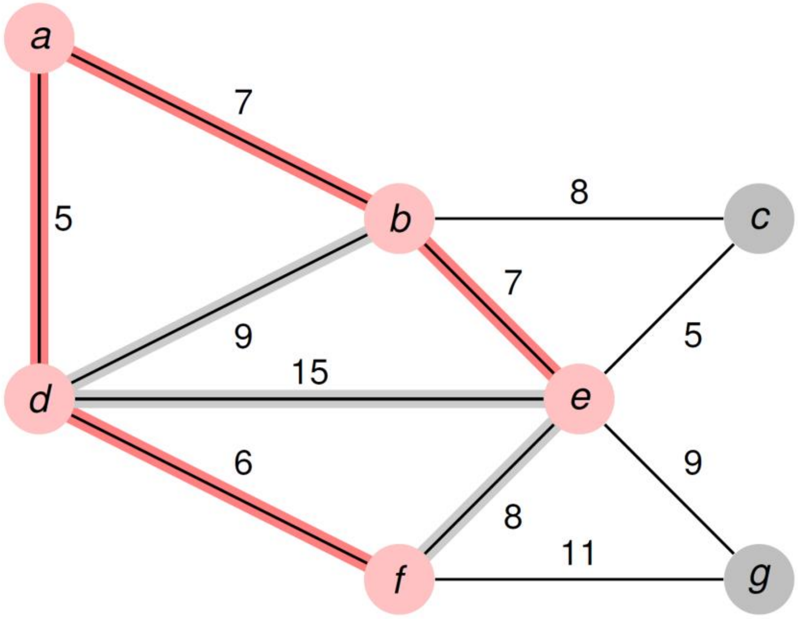
$V_T = \{d, a\}$, $PQ = \{(f, 6), (b, 7), \cancel{(b, 9)}, (e, 15)\}$

Prim’s Algorithm – Example



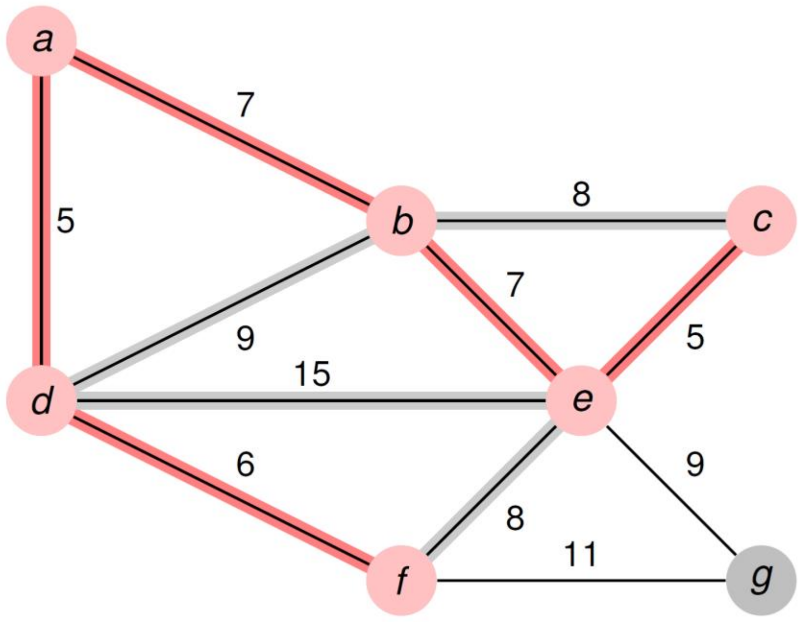
$V_T = \{d, a, f\}, PQ = \{(b, 7), (e, 8), (g, 11), \cancel{(e, 15)}\}$

Prim's Algorithm – Example



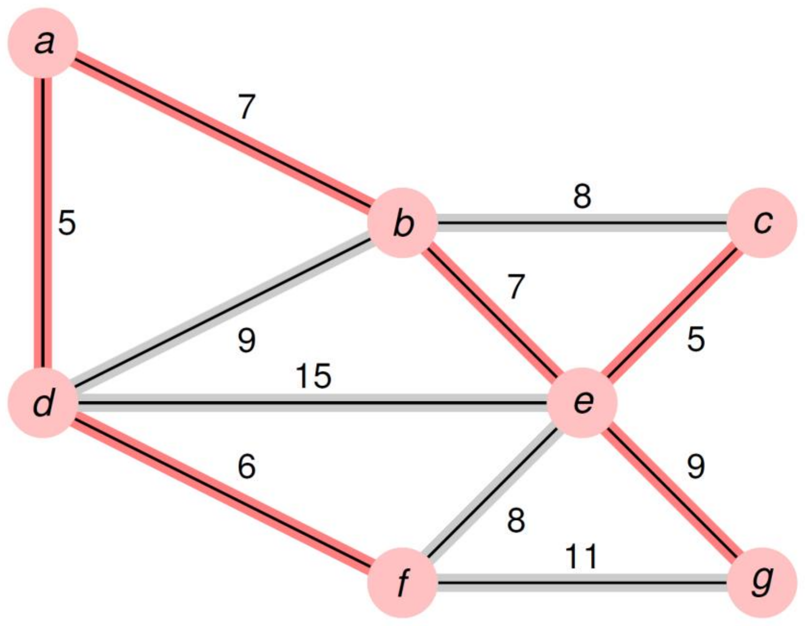
$V_T = \{d, a, f, b, e\}$, $PQ = \{(c, 5), \cancel{(c, 8)}, (g, 9), \cancel{(g, 11)}\}$

Prim's Algorithm – Example



$V_T = \{d, a, f, b, e, c\}$, $PQ = \{(g, 9)\}$

Prim’s Algorithm – Example



$V_T = \{d, a, f, b, e, c, g\}$, $PQ = \{\}$

Prim's Algorithm – Summary

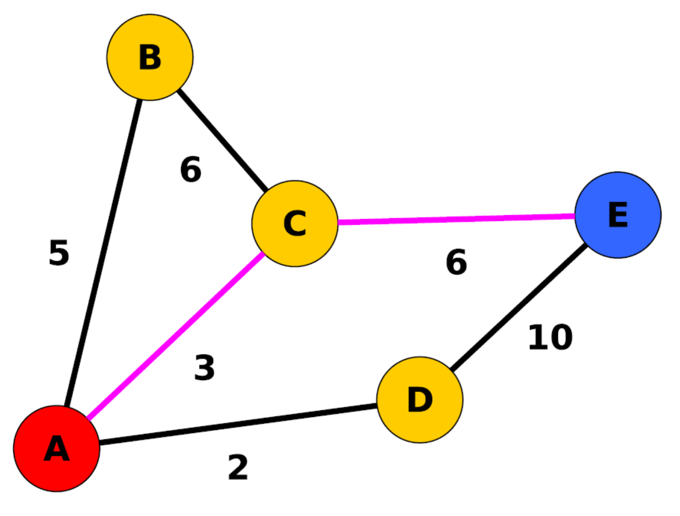
- The efficiency of the algorithm depends on the underlying data structure used
  - Adjacency matrix: O(|V|^2)
  - Adjacency list and min-heap: O((|V| + |E|)lg|V|) = O(|E|lg|V|)

1b. Dijkstra's Algorithm

RMIT UNIVERSITY

Shortest Paths in Graphs

Problem: Given a weighted connected graph, the shortest-path problem asks to find the shortest path from a starting source vertex to a destination target vertex.



Dijkstra’s Algorithm

- **Problem:** Given a weighted connected graph, the single-source shortest-paths problem asks to find the shortest path to all vertices given a single starting source vertex.

Dijkstra's Algorithm

Idea:

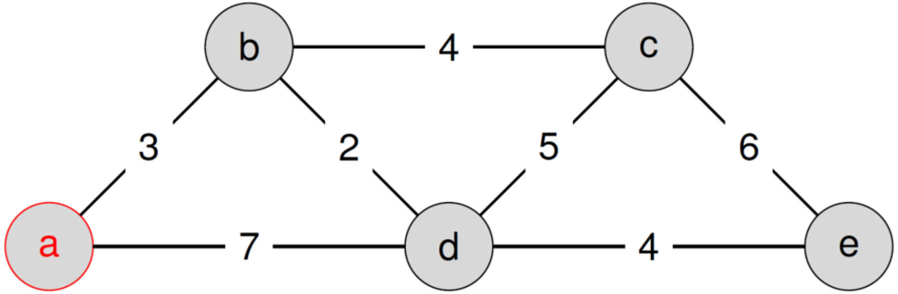
- At all times, we maintain our best estimate of the shortest-path distances from source vertex to all other vertices.
- Initially we don't know, so all distance estimates are ∞.
- But as the algorithm explores the graph, we update our estimates, which converges to the true shortest path distance.

Dijkstra's Algorithm – Sketch

Maintain a set S of vertices whose final shortest-path weights from the source s have already been determined.

1. Initially S is empty. Initialise distance estimates to ∞ for all non-source vertices. Distance of source vertex is 0.
2. Select the vertex v not in S with the minimum shortest-path estimate.
3. Add v to S.
4. Update our distance estimates to neighbouring vertices that are not in S.
5. Repeat from step 2, until all vertices have been added to S.

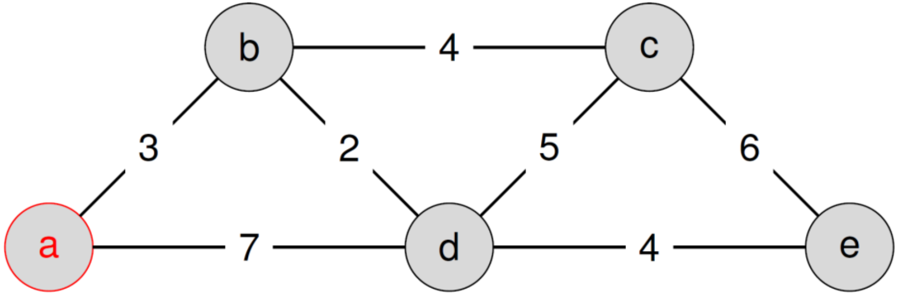
Dijkstra’s Algorithm – Example



a(a,0) b(-,∞) c(-,∞) d(-,∞) e(-,∞)

S = { }

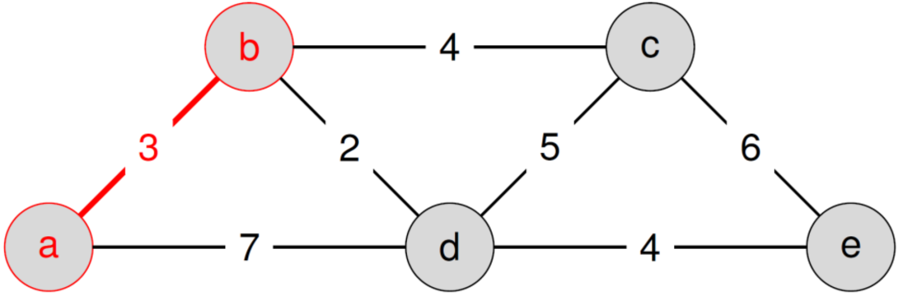
Dijkstra’s Algorithm – Example



b(a,3) c(-,∞) d(a,7) e(-,∞)

S = {a(a,0)}

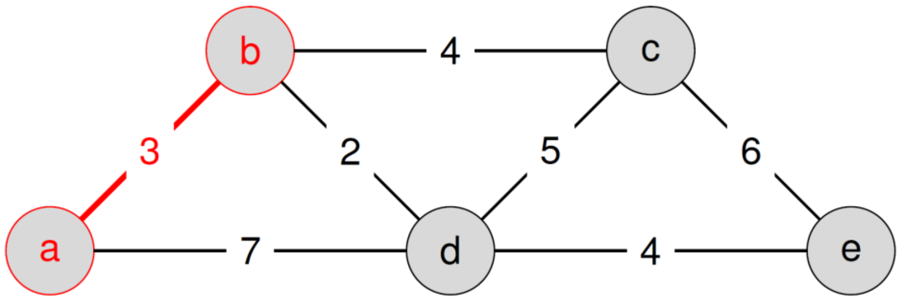
Dijkstra’s Algorithm – Example



b(a,3)   c(-,∞)   d(a,7)   e(-,∞)

S = {a(a,0)}

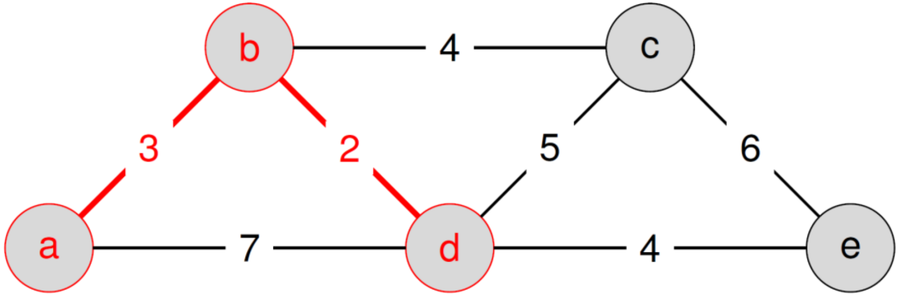
Dijkstra's Algorithm – Example



c(b,3 + 4)   d(b,3 + 2)   e(-,∞)

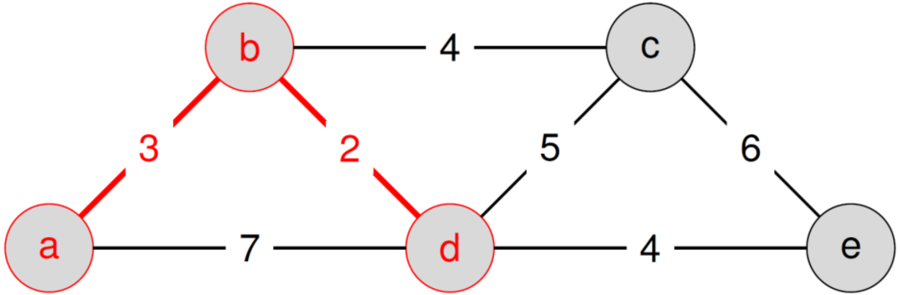
S = {a(a,0), b(a,3)}

Dijkstra's Algorithm – Example



c(b,7) d(b,5) e(-,∞)
S = {a(a,0), b(a,3)}

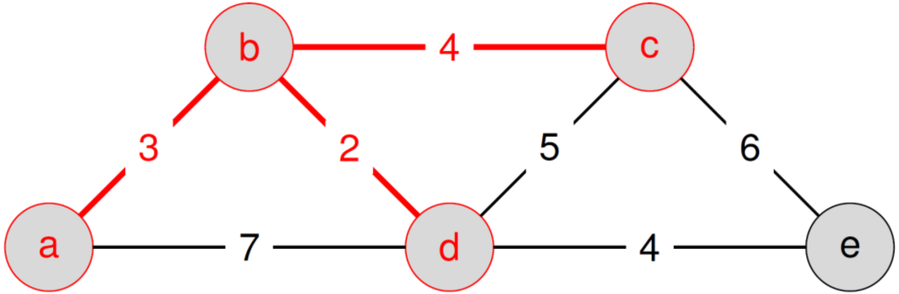
Dijkstra's Algorithm – Example



c(b,7) e(d,5+4)

S = {a(a,0), b(a,3), d(b,5)}

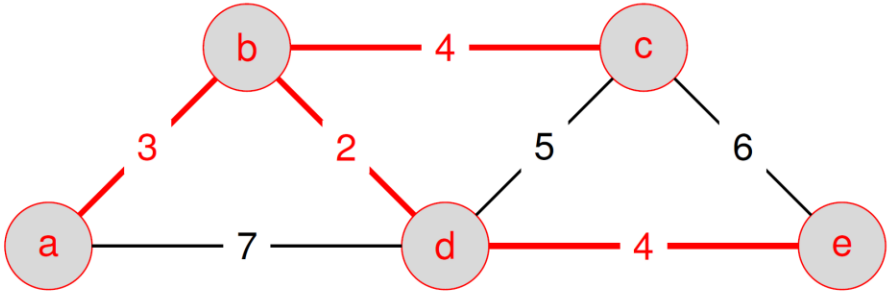
Dijkstra’s Algorithm – Example



e(d,9)

S = {a(a,0), b(a,3), d(b,5), c(b,7)}

Dijkstra’s Algorithm – Example



S = {a(a,0), b(a,3), d(b,5), c(b,7), e(d,9)}

Dijkstra’s Algorithm – Example

So, we have the following distances from vertex a:

$$a(a,0) \quad b(a,3) \quad d(b,5) \quad c(b,7) \quad e(d,9)$$

Which gives the following shortest paths:

Length	Path
3	a – b
5	a – b – d
7	a – b – c
9	a – b – d – e

Dijkstra's Algorithm – Summary

- Dijkstra's algorithm is guaranteed to always return the optimal solution.
- Time complexity
  - Adjacency matrix: O(|V|^2)
  - Adjacency list and min-heap: O((|V| + |E|)lg|V|) = O(|E|lg|V|)

2. Dynamic Programming

Dynamic Programming

- **Dynamic Programming** is a general algorithm approach for solving problems using the solutions of **overlapping** subproblems.

Dynamic Programming – Idea

1. Setup a recurrence relating a solution of larger instances to the solutions of smaller instances.

2. Solve smaller instances **once**.

3. Record solutions in a table.

4. Extract solutions to the initial instance from the table, i.e., use solutions of smaller instances to construct solutions of initial larger problem instance.

Dynamic Programming

- Sounds familiar? ***Divide and Conquer***?
- What is the difference?
  - Dynamic programming can be thought of as (1) Divide and Conquer and (2) **storing sub-solutions**.
  - Why have both then?

Dynamic Programming

- **Divide-and-conquer algorithms** are preferred when the sub-problems/instances are **independent**, e.g., merge sort.

- **Dynamic programming approach** is better when the sub-problems/instances are **dependent**, i.e., the solution to a sub-problem **may be needed multiple times**.

Dynamic Programming

- Hence saving solutions allow them to be **reused rather than recomputed**.
- Trade-off space (more) for time (faster).
- “Programming” here means “planning”

Dynamic Programming Approaches

- Two basic approaches to Dynamic Programming:
  - Bottom-Up
  - Top-Down

Dynamic Programming Approaches

- **Bottom-Up**
  - Study a recursive divide and conquer algorithm and figure out the dependencies between the subproblems.
  - Solve all subproblems, and then use solutions to subproblems to construct solutions to larger problems.

Dynamic Programming Approaches

- **Top-Down**
  - Start with a divide and conquer algorithm, and begin dividing recursively.
  - Only solve/recurse on a subproblem if the solution is not available in the table (→ dependency)
  - Save solutions to subproblems in a table.

2a. Knapsack Problem (Bottom Up)

Knapsack Problem

- Given n items of known weights $w_1, \ldots, w_n$ and the values $v_1, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.
- Recall that the exact solution for all instances of this problem has been proven to be $O(2^n)$.
- We can solve the problem using dynamic programming in "**pseudo-polynomial**" time.

Knapsack Problem – DP Sketch

- Consider an instance of the knapsack problem defined by the first i items, $1 \le i \le n$, with weights $w_1, \ldots, w_n$, values $v_1, \ldots, v_n$, and capacity j, $1 \le j \le W$.
- Let $V[i, j]$ be an optimal value to the subproblem instance of having the first i items and a knapsack capacity of j.
  - If we can convert the current problem into a subproblem like this, we can ask the question: **"Should we put n to the bag or not?"**

Knapsack Problem – DP Sketch

- We can divide all the subsets of the first i items that fit into the knapsack of capacity j into two categories:
  - The subsets that do not include the i^{th} item (last item)
  - The subsets that include the i^{th} item (last item)

Knapsack Problem – DP Sketch

- Among the subsets that **do not** include the i^{th} item, the value of the optimal subset is, by definition, $V[i-1, j]$

- Among the subsets that **do include** the i^{th} item ($j - w_i \geq 0$), an optimal subset is made up of this item and an optimal subset of the first $i - 1$ items that fit into the knapsack of capacity $j - w_i$.
    - The value of such an optimal subset is $v_i + V[i-1, j-w_i]$.

Knapsack Problem – DP Sketch

- Whether we choose to include i^{th} item **depends on** whether the i^{th} item can fit into knapsack and if so, which leads to larger value ($V[i,j]$).
- This leads to the following recursion:

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0$$

Bottom-Up DP Algorithm

- **Bottom-up Dynamic Programming:** What we have been doing up to this point, computing solutions to all entries in the dynamic programming table.

- Use an example to illustrate the table filling process.

Bottom-Up DP Algorithm

Given the following problem, how do we solve it using a Bottom-Up Dynamic Programming algorithm?

- Knapsack capacity $W = 6$

i	1	2	3	4	5
weight (w_i)	3	2	1	4	5
value (v_i)	$25	$20	$15	$40	$50

Bottom-Up DP Algorithm

We record the solutions to each smaller problems in table.

$\downarrow i \quad W \rightarrow$	0	1	2	3	4	5	6
0							
1							
2							
3					?		
4							
5							**GOAL**

$V[3,4] = ?$ *stores the optimal value for a knapsack with only the first 3 items and has a capacity of 4*

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $\quad W \longrightarrow$		0	1	2	3	4	5	6
	0							
$w_1 = 3, v_1 = 25$	1							
$w_2 = 2, v_2 = 20$	2							
$w_3 = 1, v_3 = 15$	3							
$w_4 = 4, v_4 = 40$	4							
$w_5 = 5, v_5 = 50$	5							

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0						
$w_2 = 2, v_2 = 20$	2	0						
$w_3 = 1, v_3 = 15$	3	0						
$w_4 = 4, v_4 = 40$	4	0						
$w_5 = 5, v_5 = 50$	5	0						

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $\quad W \longrightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0					
$w_2 = 2, v_2 = 20$	2	0						
$w_3 = 1, v_3 = 15$	3	0						
$w_4 = 4, v_4 = 40$	4	0						
$w_5 = 5, v_5 = 50$	5	0						

Calculating value for $V[1,1]$; $i = 1, j = 1$

$j - w_i = 1 - w_1 = 1 - 3 = -2 < 0$

$\rightarrow V[1,1] = V[i-1,j] = V[1-1,1] = V[0,1] = 0$

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0					
$w_2 = 2, v_2 = 20$	2	0	0					
$w_3 = 1, v_3 = 15$	3	0						
$w_4 = 4, v_4 = 40$	4	0						
$w_5 = 5, v_5 = 50$	5	0						

Calculating value for $V[2,1]$; $i = 2, j = 1$

$j - w_i = 1 - w_2 = 1 - 2 = -1 < 0$

$\rightarrow V[2,1] = V[i-1,j] = V[2-1,1] = V[1,1] = 0$

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i \qquad W \longrightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0					
$w_2 = 2, v_2 = 20$	2	0	0					
$w_3 = 1, v_3 = 15$	3	0	15					
$w_4 = 4, v_4 = 40$	4	0	Calculating value for $V[3,1]$; $i = 3, j = 1$					
$w_5 = 5, v_5 = 50$	5	0						

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i \qquad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0					
$w_2 = 2, v_2 = 20$	2	0	0					
$w_3 = 1, v_3 = 15$	3	0	15					
$w_4 = 4, v_4 = 40$	4	0	15					
$w_5 = 5, v_5 = 50$	5	0	15					

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0				
$w_2 = 2, v_2 = 20$	2	0	0					
$w_3 = 1, v_3 = 15$	3	0	15					
$w_4 = 4, v_4 = 40$	4	0	15					
$w_5 = 5, v_5 = 50$	5	0	15					

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0				
$w_2 = 2, v_2 = 20$	2	0	0	20				
$w_3 = 1, v_3 = 15$	3	0	15					
$w_4 = 4, v_4 = 40$	4	0	15					
$w_5 = 5, v_5 = 50$	5	0	15					

Calculating $V[2,2]$; $i = 2, j = 2$

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0				
$w_2 = 2, v_2 = 20$	2	0	0	20				
$w_3 = 1, v_3 = 15$	3	0	15	20				
$w_4 = 4, v_4 = 40$	4	0	15	20				
$w_5 = 5, v_5 = 50$	5	0	15	20				

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25			
$w_2 = 2, v_2 = 20$	2	0	0	20				
$w_3 = 1, v_3 = 15$	3	0	15	20				
$w_4 = 4, v_4 = 40$	4	0	15	20				
$w_5 = 5, v_5 = 50$	5	0	15	20				

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25			
$w_2 = 2, v_2 = 20$	2	0	0	20	25			
$w_3 = 1, v_3 = 15$	3	0	15	20				
$w_4 = 4, v_4 = 40$	4	0	15	20				
$w_5 = 5, v_5 = 50$	5	0	15	20				

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25			
$w_2 = 2, v_2 = 20$	2	0	0	20	25			
$w_3 = 1, v_3 = 15$	3	0	15	20	?			
$w_4 = 4, v_4 = 40$	4	0	15	20				
$w_5 = 5, v_5 = 50$	5	0	15	20				

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25			
$w_2 = 2, v_2 = 20$	2	0	0	20	25			
$w_3 = 1, v_3 = 15$	3	0	15	20	35			
$w_4 = 4, v_4 = 40$	4	0	15	20				
$w_5 = 5, v_5 = 50$	5	0	15	20				

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25			
$w_2 = 2, v_2 = 20$	2	0	0	20	25			
$w_3 = 1, v_3 = 15$	3	0	15	20	35			
$w_4 = 4, v_4 = 40$	4	0	15	20	35			
$w_5 = 5, v_5 = 50$	5	0	15	20	35			

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25	25		
$w_2 = 2, v_2 = 20$	2	0	0	20	25	25		
$w_3 = 1, v_3 = 15$	3	0	15	20	35			
$w_4 = 4, v_4 = 40$	4	0	15	20	35			
$w_5 = 5, v_5 = 50$	5	0	15	20	35			

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25	25		
$w_2 = 2, v_2 = 20$	2	0	0	20	25	25		
$w_3 = 1, v_3 = 15$	3	0	15	20	35	?		
$w_4 = 4, v_4 = 40$	4	0	15	20	35			
$w_5 = 5, v_5 = 50$	5	0	15	20	35			

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25	25		
$w_2 = 2, v_2 = 20$	2	0	0	20	25	25		
$w_3 = 1, v_3 = 15$	3	0	15	20	35	?		
$w_4 = 4, v_4 = 40$	4	0	15	20	35			
$w_5 = 5, v_5 = 50$	5	0	15	20	35			

Calc. $V[3,4]$; $i = 3, j = 4$

$\max(V[2,4], v_3 + V[2,3])$

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25	25		
$w_2 = 2, v_2 = 20$	2	0	0	20	25	25		
$w_3 = 1, v_3 = 15$	3	0	15	20	35	40		
$w_4 = 4, v_4 = 40$	4	0	15	20	35			
$w_5 = 5, v_5 = 50$	5	0	15	20	35			

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0 \text{ for } j \geq 0 \text{ and } V[i,0] = 0 \text{ for } i \geq 0$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25	25		
$w_2 = 2, v_2 = 20$	2	0	0	20	25	25		
$w_3 = 1, v_3 = 15$	3	0	15	20	35	40		
$w_4 = 4, v_4 = 40$	4	0	15	20	35	40		
$w_5 = 5, v_5 = 50$	5	0	15	20	35	40		

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i \qquad W \longrightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25	25	25	
$w_2 = 2, v_2 = 20$	2	0	0	20	25	25	45	
$w_3 = 1, v_3 = 15$	3	0	15	20	35	40	45	
$w_4 = 4, v_4 = 40$	4	0	15	20	35	40		
$w_5 = 5, v_5 = 50$	5	0	15	20	35	40		

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25	25	25	
$w_2 = 2, v_2 = 20$	2	0	0	20	25	25	45	
$w_3 = 1, v_3 = 15$	3	0	15	20	35	40	45	
$w_4 = 4, v_4 = 40$	4	0	15	20	35	40	55	
$w_5 = 5, v_5 = 50$	5	0	15	20	35	40		

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $\quad W \longrightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25	25	25	
$w_2 = 2, v_2 = 20$	2	0	0	20	25	25	45	
$w_3 = 1, v_3 = 15$	3	0	15	20	35	40	45	
$w_4 = 4, v_4 = 40$	4	0	15	20	35	40	55	
$w_5 = 5, v_5 = 50$	5	0	15	20	35	40	?	

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25	25	25	
$w_2 = 2, v_2 = 20$	2	0	0	20	25	25	45	
$w_3 = 1, v_3 = 15$	3	0	15	20	35	40	45	
$w_4 = 4, v_4 = 40$	4	0	15	20	35	40	55	
$w_5 = 5, v_5 = 50$	5	0	15	20	35	40	55	

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2, v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1, v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4, v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5, v_5 = 50$	5	0	15	20	35	40	55	

Bottom-Up DP Algorithm

$$V[i,j] = \begin{cases} \max(V[i-1,j], v_i + V[i-1,j-w_i]) & \text{if } j - w_i \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ for $j \geq 0$ and $V[i,0] = 0$ for $i \geq 0$

$\downarrow i$ $\quad W\rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2, v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1, v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4, v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5, v_5 = 50$	5	0	15	20	35	40	55	65

Bottom-Up DP Algorithm – Backtrace

How to find the set of items to include? Use backtrace

1. From $V[n, W]$, trace back how we arrived at this table cell – either from $V[n-1, W]$ or $V[n-1, W-w_n]$.
2. Repeat this step until reach $V[0,0]$.
3. Items that were included in the backtrack form the final solution for knapsack problem.

Bottom-Up DP Algorithm – Backtrace

$\downarrow i$ $\quad W\rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2, v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1, v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4, v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5, v_5 = 50$	5	0	15	20	35	40	55	65

Let's do the backtrack!

Bottom-Up DP Algorithm – Backtrace

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2, v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1, v_3 = 15$	3	0	$\boxed{15}$	20	35	40	45	60
$w_4 = 4, v_4 = 40$	4	0	$\boxed{15}$	20	35	40	55	$\boxed{60}$
$w_5 = 5, v_5 = 50$	5	0	15	20	35	40	55	$\boxed{65}$

5th

Bottom-Up DP Algorithm – Backtrace

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2, v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1, v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4, v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5, v_5 = 50$	5	0	15	20	35	40	55	65

3rd 5th

Bottom-Up DP Algorithm – Backtrace

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	$\boxed{0}$	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	$\boxed{0}$	0	0	25	25	25	25
$w_2 = 2, v_2 = 20$	2	0	$\boxed{0}$	20	25	25	45	45
$w_3 = 1, v_3 = 15$	3	0	$\boxed{15}$	20	35	40	45	60
$w_4 = 4, v_4 = 40$	4	0	15	20	35	40	55	$\boxed{60}$
$w_5 = 5, v_5 = 50$	5	0	15	20	35	40	55	$\boxed{65}$

3rd $\qquad$ 5th

Bottom-Up DP Algorithm – Backtrace

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2, v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1, v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4, v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5, v_5 = 50$	5	0	15	20	35	40	55	65

Question: In general, using the dynamic programming table, how can we tell if there are multiple optimal solutions to a Knapsack problem?

DP Knapsack Problem

- The complexity of constructing the dynamic table is $\Theta(nW)$ in time and space (pretty expensive)
- The complexity of performing the backtrace to find the optimal subset is $\Theta(n + W)$.
  - **NOTE:** The running time of this algorithm is not a polynomial function of n; rather it is a polynomial function of n and W, the largest integer involved in defining the problem.
  - Such algorithms are known as pseudo-polynomial. They are efficient when the values $\{w_i\}$ are small, but less practical as these values grow large.

2b. Knapsack Problem (Top Down)

RMIT UNIVERSITY

DP Knapsack Problem – Top-Down

- *"Divide and conquer"* type of (top down) approach of solving knapsack generally recompute many previously computed sub-problems, hence inefficient.
- Bottom up dynamic programming approach avoids re-computation, but can compute many unnecessary solutions to sub-problems.
- Combine space saving of *"divide and conquer"* and speed up of bottom up approaches?

DP Knapsack Problem – Top-Down

ALGORITHM **MFKnapsack** (i, j)

/* Implement the memory function method (top-down) for the knapsack problem. */

/* INPUT : A non-negative integer i indicating the number of the first items being considered and a non-negative integer j indicating the knapsack capacity. */

/* OUTPUT : The value of an optimal, feasible subset of the first i items. */

/* NOTE: Requires global arrays $w[1 \ldots n]$ and $v[1 \ldots n]$ of weights and values of n items, and table $F[0 \ldots n, 0 \ldots W]$ initialized with -1s, except for row 0 and column 0 being all 0s. */

if $F[i, j] < 0$ **then**
    if $j < w[i]$ **then**
        $x =$ **MFKnapsack**$(i - 1, j)$
    else
        $x = \max($ **MFKnapsack**$(i - 1, j), v[i]+$ **MFKnapsack**$(i - 1, j - w[i]))$
    end if
    $F[i, j] = x$
end if
return $F[i, j]$

DP Knapsack Problem – Top-Down

```
if F[i, j] < 0 then
    if j < w[i] then
        x = MFKnapsack(i − 1, j)
    else
        x = max( MFKnapsack(i − 1, j), v[i]+ MFKnapsack(i − 1, j − w[i]))
    end if
    F[i, j] = x
end if
```

$\downarrow i \quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	–1	–1	–1	–1	–1	–1
$w_2 = 2, v_2 = 20$	2	0	–1	–1	–1	–1	–1	–1
$w_3 = 1, v_3 = 15$	3	0	–1	–1	–1	–1	–1	–1
$w_4 = 4, v_4 = 40$	4	0	–1	–1	–1	–1	–1	–1
$w_5 = 5, v_5 = 50$	5	0	–1	–1	–1	–1	–1	–1

Initially, set all values to -1 to indicate that the entries are not yet calculated

When a new value needs to be calculated, the method checks the table

DP Knapsack Problem – Top-Down

if $F[i,j] < 0$ **then**
  if $j < w[i]$ **then**
    $x =$ **MFKnapsack**$(i-1, j)$
  else
    $x = \max($ **MFKnapsack**$(i-1,j), v[i]+$ **MFKnapsack**$(i-1, j-w[i]))$
  end if
  $F[i,j] = x$
end if

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	–1	–1	–1	–1	–1	–1
$w_2 = 2, v_2 = 20$	2	0	–1	–1	–1	–1	–1	–1
$w_3 = 1, v_3 = 15$	3	0	–1	–1	–1	–1	–1	–1
$w_4 = 4, v_4 = 40$	4	0	–1	–1	–1	–1	–1	–1
$w_5 = 5, v_5 = 50$	5	0	–1	–1	–1	–1	–1	□

Let’s start with M(5,6)

DP Knapsack Problem – Top-Down

```
if F[i, j] < 0 then
    if j < w[i] then
        x = MFKnapsack(i − 1, j)
    else
        x = max( MFKnapsack(i − 1, j), v[i]+ MFKnapsack(i − 1, j − w[i]))
    end if
    F[i, j] = x
end if
```

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	–1	–1	–1	–1	–1	–1
$w_2 = 2, v_2 = 20$	2	0	–1	–1	–1	–1	–1	–1
$w_3 = 1, v_3 = 15$	3	0	–1	–1	–1	–1	–1	–1
$w_4 = 4, v_4 = 40$	4	0	□	–1	–1	–1	–1	□
$w_5 = 5, v_5 = 50$	5	0	–1	–1	–1	–1	–1	□

At M(5,6), $j > w_i$

We calculate M(4,6) and M(4,1)

DP Knapsack Problem – Top-Down

```
if F[i, j] < 0 then
    if j < w[i] then
        x = MFKnapsack(i − 1, j)
    else
        x = max( MFKnapsack(i − 1, j), v[i]+ MFKnapsack(i − 1, j − w[i]))
    end if
    F[i, j] = x
end if
```

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	–1	–1	–1	–1	–1	–1
$w_2 = 2, v_2 = 20$	2	0	–1	–1	–1	–1	–1	–1
$w_3 = 1, v_3 = 15$	3	0	□	–1	–1	–1	–1	–1
$w_4 = 4, v_4 = 40$	4	0	□	–1	–1	–1	–1	□
$w_5 = 5, v_5 = 50$	5	0	–1	–1	–1	–1	–1	□

M(4,1)
→ Calculate M(3,1)

DP Knapsack Problem – Top-Down

```
if F[i, j] < 0 then
    if j < w[i] then
        x = MFKnapsack(i − 1, j)
    else
        x = max( MFKnapsack(i − 1, j), v[i]+ MFKnapsack(i − 1, j − w[i]))
    end if
    F[i, j] = x
end if
```

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	□	–1	–1	–1	–1	–1
$w_2 = 2, v_2 = 20$	2	0	□	–1	–1	–1	–1	–1
$w_3 = 1, v_3 = 15$	3	0	□	–1	–1	–1	–1	–1
$w_4 = 4, v_4 = 40$	4	0	□	–1	–1	–1	–1	□
$w_5 = 5, v_5 = 50$	5	0	–1	–1	–1	–1	–1	□

DP Knapsack Problem – Top-Down

```
if F[i, j] < 0 then
    if j < w[i] then
        x = MFKnapsack(i − 1, j)
    else
        x = max( MFKnapsack(i − 1, j), v[i]+ MFKnapsack(i − 1, j − w[i]))
    end if
    F[i, j] = x
end if
```

$\downarrow i \qquad W\rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	□	–1	–1	–1	–1	–1
$w_2 = 2, v_2 = 20$	2	0	□	–1	–1	–1	–1	–1
$w_3 = 1, v_3 = 15$	3	0	□	□	–1	–1	–1	□
$w_4 = 4, v_4 = 40$	4	0	□	–1	–1	–1	–1	□
$w_5 = 5, v_5 = 50$	5	0	–1	–1	–1	–1	–1	□

M(4,6) → M(3,6), M(3,2)

DP Knapsack Problem – Top-Down

```
if F[i,j] < 0 then
    if j < w[i] then
        x = MFKnapsack(i − 1, j)
    else
        x = max( MFKnapsack(i − 1, j), v[i]+ MFKnapsack(i − 1, j − w[i]))
    end if
    F[i,j] = x
end if
```

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	□	□	□	□	□	□
$w_2 = 2, v_2 = 20$	2	0	□	□	–1	–1	□	□
$w_3 = 1, v_3 = 15$	3	0	□	□	–1	–1	–1	□
$w_4 = 4, v_4 = 40$	4	0	□	–1	–1	–1	–1	□
$w_5 = 5, v_5 = 50$	5	0	–1	–1	–1	–1	–1	□

DP Knapsack Problem – Top-Down

```
if F[i, j] < 0 then
    if j < w[i] then
        x = MFKnapsack(i − 1, j)
    else
        x = max( MFKnapsack(i − 1, j), v[i]+ MFKnapsack(i − 1, j − w[i]))
    end if
    F[i, j] = x
end if
```

$\downarrow i$ $\quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	□	□	□	□	□	□
$w_2 = 2, v_2 = 20$	2	0	□	□	–	–	□	□
$w_3 = 1, v_3 = 15$	3	0	□	□	–	–	–	□
$w_4 = 4, v_4 = 40$	4	0	□	–	–	–	–	□
$w_5 = 5, v_5 = 50$	5	0	–	–	–	–	–	□

Red squares indicate the possible items that we need to calculate

DP Knapsack Problem – Top-Down

```
if F[i, j] < 0 then
    if j < w[i] then
        x = MFKnapsack(i − 1, j)
    else
        x = max( MFKnapsack(i − 1, j), v[i]+ MFKnapsack(i − 1, j − w[i]))
    end if
    F[i, j] = x
end if
```

$\downarrow i \quad W \rightarrow$		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2, v_2 = 20$	2	0	0	20	–	–	45	45
$w_3 = 1, v_3 = 15$	3	0	15	20	–	–	–	60
$w_4 = 4, v_4 = 40$	4	0	15	–	–	–	–	60
$w_5 = 5, v_5 = 50$	5	0	–	–	–	–	–	**65**

No need to calculate every entry as done in the Bottom-Up approach

This approach also enables retrieving values rather than recomputing

Top-Down vs. Bottom-Up

In general, when to use top-down or bottom-up dynamic programming?

Top-down incurs additional space and time cost of maintaining stack space for storing recursive function calls. Hence:

- **Bottom-up:** When the final problem instance requires most or all of the sub-problem instances to be solved.
- **Top-down:** When the final problem instance only requires a subset of the sub-problem instances to be solved.

Wrapping things up

Learning Objectives

1. **Understand and apply the Greedy approach to solving problems**
   - Prim's Algorithm (find minimum spanning tree)
   - Dijkstra's Algorithm (find shortest path distances)
2. **Understand and apply Dynamic Programming techniques to solving problems**
   - Knapsack Problem

Thank you for a great and enjoyable semester!

See you again in other courses ☺

RMIT UNIVERSITY