# Chapter 11. More Greedy Algorithms

- Dijkstra’s algorithm (single source shortest paths)
- Fractional knapsack problem
- Finding optimal code

# Dijkstra’s Algorithm

Single source shortest paths for a directed graph with no negative edges

# Single-Source Shortest Paths

- We want to find the shortest paths between Binghamton and New York City, Boston, and Washington DC.  Given a US road map with all the possible routes how can we determine our shortest paths?

# Single Source Shortest Paths Problem

- To solve this problem, we may use Floyd's algorithm that finds all pairs shortest paths via dynamic programming. But, this is an overkill, because we have *a single source* now.

- Floyd's algorithm is $\Theta(n^3)$. Can we solve the single source shortest paths problem faster than $\Theta(n^3)$?

# Dijkstra’s algorithm

- Given a weighted digraph and a vertex *s* in the graph, find a shortest path from *s* to an arbitrary node t
- Both for directed and undirected graphs
- No negative edges
- Graph must be connected

# Dijkstra's shortest path algorithm

- **Dijkstra's algorithm solves the single source shortest path problem in 2 stages.**
  - Stage 1: A greedy algorithm computes the shortest *distance* from *s* to all other nodes in the graph and saves a *data structure*.
  - Stage 2 : Uses the *data structure* to find a shortest path from *s* to *t*.

# Main idea

- Assume that the shortest distances from the starting node *s* to the rest of the nodes are
  $d(s, s) \le d(s, s_1) \le d(s, s_2) \le \ldots \le d(s, s_{n-1})$
- In this case a shortest path from s to $s_i$ may include any of the vertices $\{s_1, s_2 \ldots s_{i-1}\}$ but cannot include any $s_j$ where $j > i$.
- Dijkstra's main idea is to select the nodes and compute the shortest distances in the order $s, s_1, s_2, \ldots, s_{n-1}$

# Example

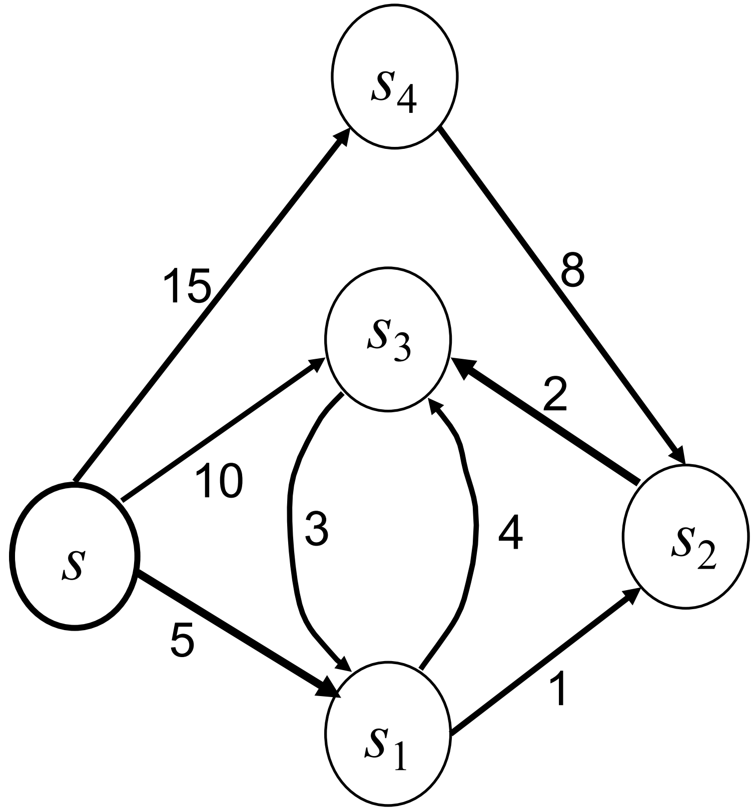
$d(s, s) = 0 \leq$

$d(s, s_1) = 5 \leq$

$d(s, s_2) = 6 \leq$

$d(s, s_3) = 8 \leq$

$d(s, s_4) = 15$




Note: The shortest path from $s$ to $s_2$ includes $s_1$ as an intermediate node but cannot include $s_3$ or $s_4$.

# Dijkstra's greedy selection rule

- Assume $s_1$, $s_2$ … $s_{i-1}$ have been selected, and their shortest distances have been stored in ***Solution***

- Select node $s_i$ and save d($s$, $s_i$) if $s_i$ has the shortest distance from s on a path that may include only $s_1$, $s_2$ … $s_{i-1}$ as intermediate nodes. We call such paths ***special***

- To apply this selection rule efficiently, we need to maintain for each unselected node *v* the *distance of the **shortest special path*** from *s* to *v*, *D*[*v*].

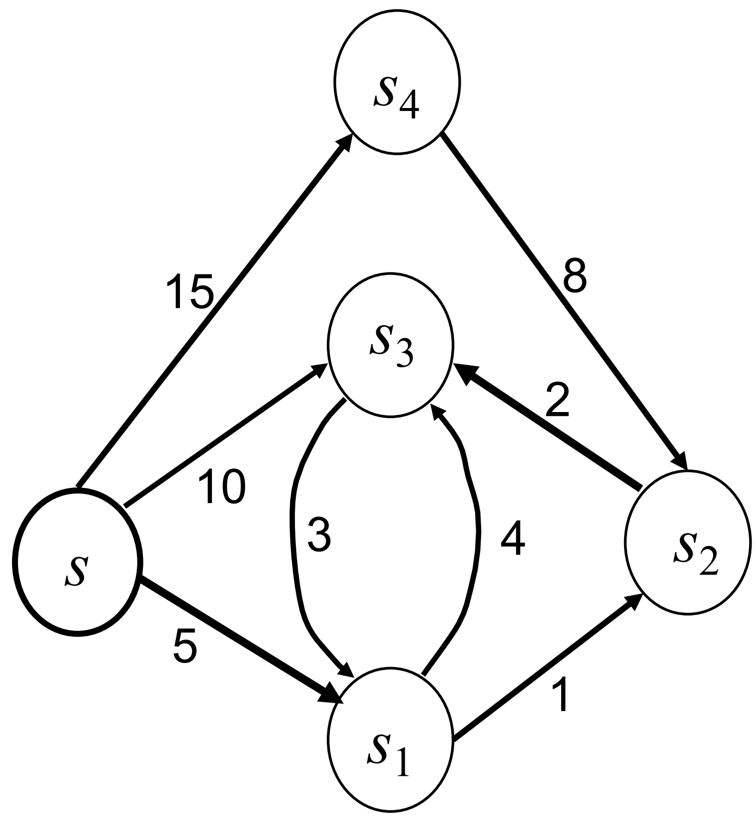
# Application Example

*Solution* = {($s$, 0)}
D[$s_1$]=5 for path [$s$, $s_1$]
D[$s_2$]= $\infty$ for path [$s$, $s_2$]
D[$s_3$]=10 for path [$s$, $s_3$]
D[$s_4$]=15 for path [$s$, $s_4$].

*Solution* = {($s$, 0), ($s_1$, 5) }
D[$s_2$]= 6 for path [$s$, $s_1$, $s_2$]
D[$s_3$]=9 for path [$s$, $s_1$, $s_3$]
D[$s_4$]=15 for path [$s$, $s_4$]

*Solution* = {($s$, 0), ($s_1$, 5), ($s_2$, 6) }
D[$s_3$]=8 for path [$s$, $s_1$, $s_2$, $s_3$]
D[$s_4$]=15 for path [$s$, $s_4$]

*Solution* = {($s$, 0), ($s_1$, 5), ($s_2$, 6),($s_3$, 8), ($s_4$, 15) }



# Implementing the selection rule

- Node *near* is selected and added to *Solution* if D(*near*) $\leq$ D(*v*) for any $v \notin$ *Solution*.

$Solution = \{(s, 0)\}$

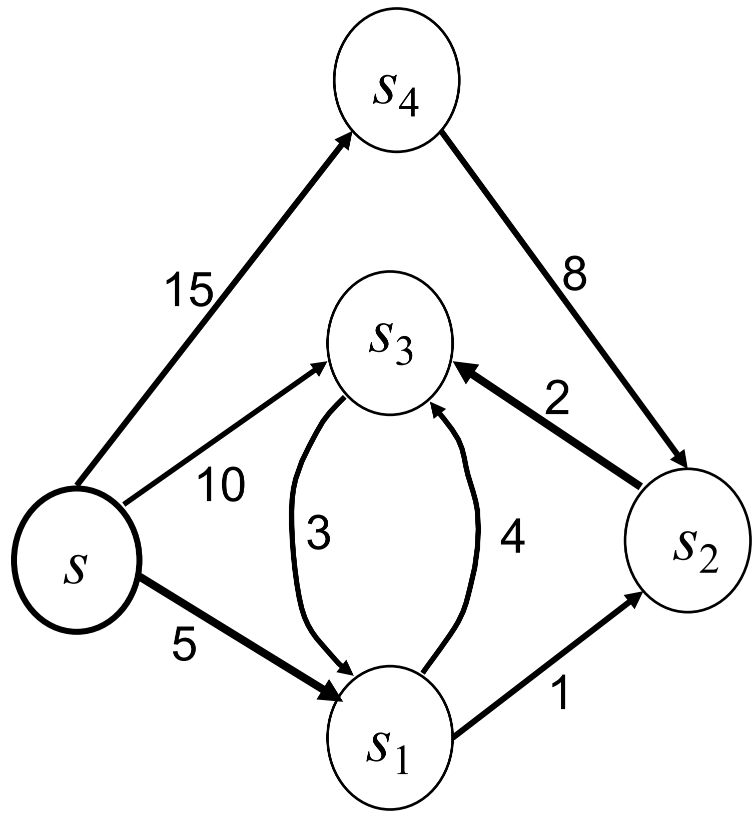
$D[s_1]=5 \leq D[s_2]=\infty$

$D[s_1]=5 \leq D[s_3]=10$

$D[s_1]=5 \leq D[s_4]=15$

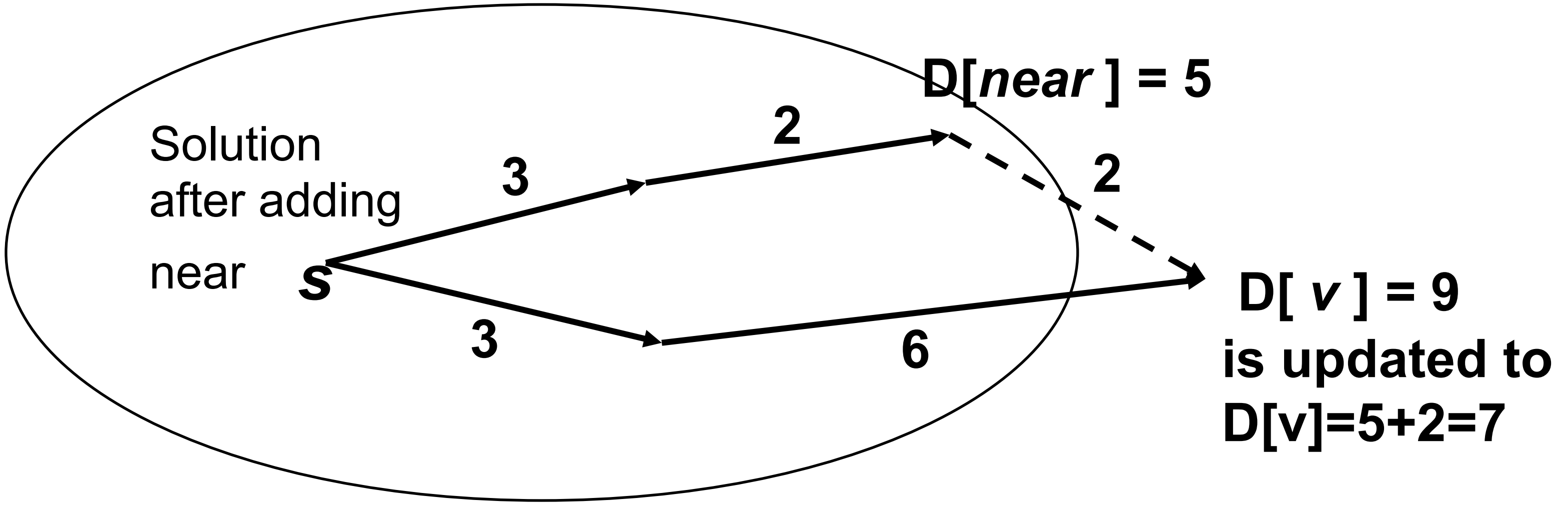
Node $s_1$ is selected

$Solution = \{(s, 0), (s_1, 5)\}$



# Updating D[ ]

- *After adding near to Solution, D[v] of all nodes v $\notin$* ***Solution*** *are updated if there is a* ***shorter special path*** *from s to v that contains node* ***near****, i.e., if (D[near] + w(near, v) < D[v]) then*
  *D[v] = D[near] + w(near, v )*

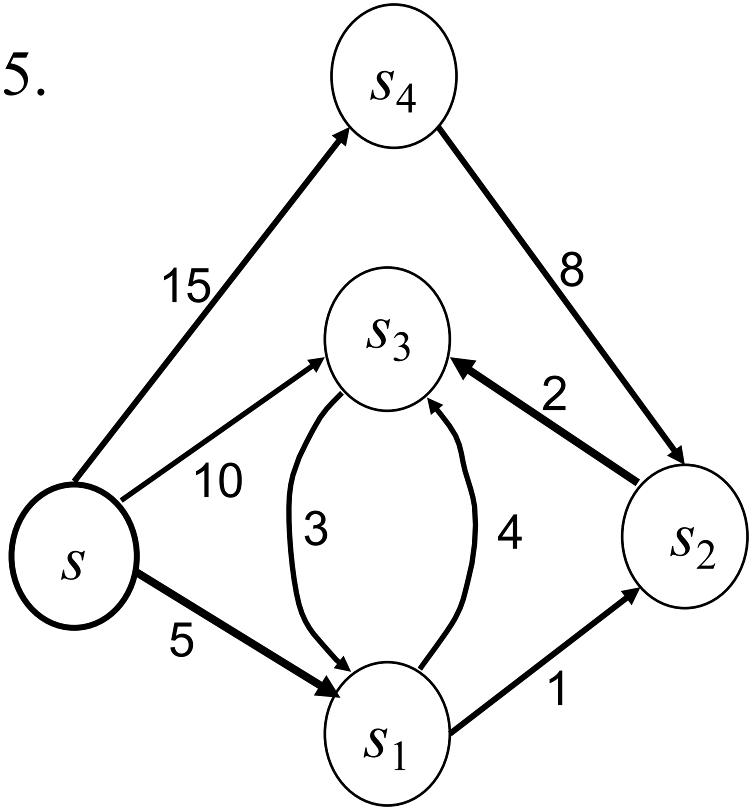


# Example: Updating D

$Solution = \{(s, 0)\}$
$D[s_1]=5$, $D[s_2]=\infty$ , $D[s_3]=10$, $D[s_4]=15$.

$Solution = \{(s, 0), (s_1, 5)\}$
$D[s_2]= D[s_1]+w(s_1, s_2)=5+1=6$,
$D[s_3]= D[s_1]+w(s_1, s_3)=5+4=9$,
$D[s_4]=15$

$Solution = \{(s, 0), (s_1, 5), (s_2, 6)\}$
$D[s_3]=D[s_2]+w(s_2, s_3)=6+2=8$,
$D[s_4]=15$




$Solution = \{(s, 0), (s_1, 5), (s_2, 6), (s_3, 8), (s_4, 15)\}$

# Dijkstra's Algorithm for Finding the Shortest Distance from a Single Source

```
Dijkstra(G, s)
    for each v ∈ V
        do D [ v ] ← ∞ Use adjacency matrix or list?
    D [ s ] ← 0
    MH ←make-MH(D,V ) // MH: MinHeap
    while MH ≠ ∅
            near ← MH.extractMin ()
            for each v ∈ Adj(near)   // Use adjacency list or matrix?
                if D[v] > D[near] + w(near, v)
                then D[v] ← D [near] + w(near, v)
                MH.decreaseDistance (D[v], v )
    return the label D[u] of each vertex u
```

# Time Complexity Analysis

1. **for each** $v \in V$
2. **do** $D[\, v\,] \leftarrow \infty$
3. $D[\, s\,] \leftarrow 0$
4. MH $\leftarrow$make-MH(D,$V$)

5. **while** MH $\neq \varnothing$
6. **do** *near* $\leftarrow$ MH.extractMin ()
7. **for each** $v \in$ Adj(*near* )
8. **if** $D[\, v\,] > D[\, near\,] + w(near, v)$
9. **then** $D[\, v\,] \leftarrow$ $D[near] + w(near, v)$
10. MH.decreaseDistance (D[$v$ ], $v$ )
11. **return** the label $D[u]$ of each vertex $u$

Assume a node in *MH* can be accessed in $O(1)$

Using Heap implementation
Lines 1 - 4 run in $O(V)$

Max Size of *MH* is | V |

**(5)** Loop = $O(V)$

**(6)** O(lg V)

**(5+6)** $O(V \lg V)$

**8, 9)** are O(1) and executed O(E) times in total

**(10)** Decrease- Key operation on the heap takes $O(\lg V)$ time, and is executed O(E) times in total

→ O(E lg V)

So total time is $O(V \lg V + E \lg V)$ = O(E lg V)

# Alternative way to implement Dijkstra's algorithm

- Use an array instead of a MinHeap
- Time Complexity
  - O(V) to extract min
  - O(1) for decreaseDistance
  - Thus, $O(V^2)$ in total

# Knapsack Problem

- There are *n* different items in a store
- Item *i* :
  - weighs $w_i$ pounds
  - worth $\$v_i$

- A thief breaks in
- Can carry up to *W* pounds in his knapsack
- What should he take to maximize the value of his haul?

# 0-1 vs. Fractional Knapsack

- 0-1 Knapsack Problem:
  - the items cannot be divided
  - thief must take entire item or leave it behind
- Fractional Knapsack Problem:
  - thief can take partial items
  - for instance, items are liquids or powders
  - solvable with a greedy algorithm…

# Greedy Fractional Knapsack Algorithm

- Sort items in decreasing order of value per pound
- While still room in the knapsack (limit of W pounds) do
  - consider next item in sorted list
  - take as much as possible (all there is room or as much as will fit)
- $O(n \log n)$ running time (for sorting)

# Greedy 0-1 Knapsack Alg?

- 3 items:
  - item 1 weighs 10 lbs, worth \$60 (\$6/lb)
  - item 2 weighs 20 lbs, worth \$100 (\$5/lb)
  - item 3 weighs 30 lbs, worth \$120 (\$4/lb)
- knapsack can hold 50 lbs
- greedy strategy:
  - take item 1
  - take item 2
  - no room for item 3

suboptimal!

# 0-1 Knapsack Problem

- Taking item 1 is a big mistake globally although looks good locally
- Use dynamic programming to solve this in pseudo-polynomial time
- Knapsack problem will be discussed in depth later on (a separate chapter)

# Finding Optimal Code

# Finding Optimal Code

- Input:
  - data file of characters and
  - number of occurrences of each character
- Output:
  - a binary encoding of each character so that the data file can be represented as efficiently as possible
  - "optimal code"

# Huffman Code

- Idea: use short codes for more frequent characters and long codes for less frequent

| char | a | b | c | d | e | f | total bits |
|---|---|---|---|---|---|---|---|
| # | 45 | 13 | 12 | 16 | 9 | 5 | |
| fixed | 000 | 001 | 010 | 011 | 100 | 101 | 300 |
| variable | 0 | 101 | 100 | 111 | 1101 | 1100 | 224 |

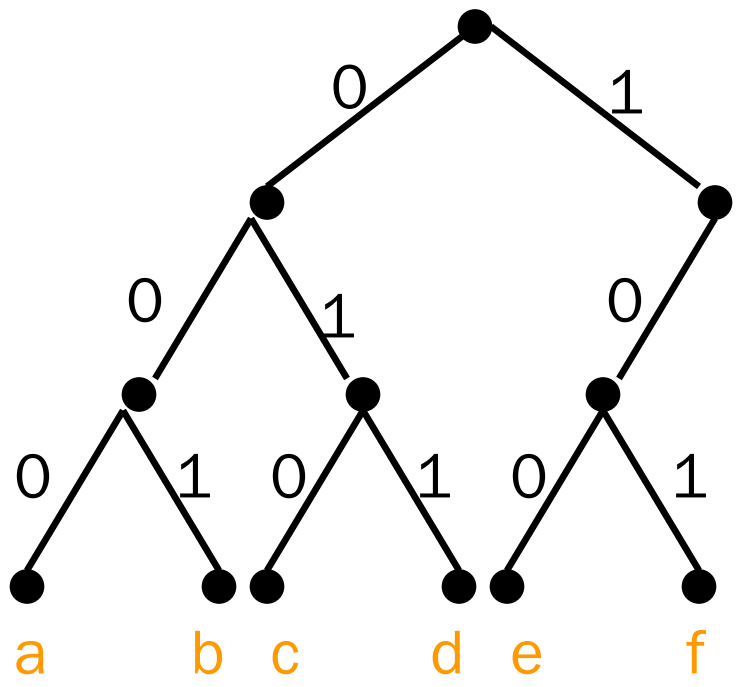
# How to Decode?

- With fixed length code, easy:
  - break up into 3's, for instance
- For variable length code, ensure that no character's code is the prefix of another
  - no ambiguity

101111110100

b d e a a

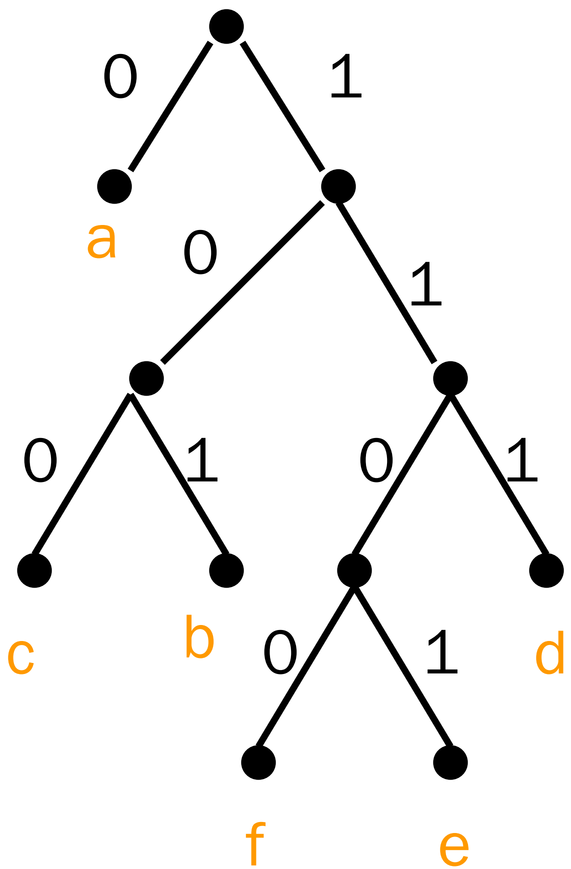
# Binary Tree Representation




fixed length code

cost of code is sum, over all chars *x*, of number of occurrences of *x* times depth of *x* in the tree

# Binary Tree Representation




variable length code

cost of code is sum, over all chars *c*, of number of occurrences of *c* times depth of *c* in the tree

# Algorithm to Construct Tree Representing Huffman Code

- Given set *C* of *n* chars, *c* occurs *f[c]* times
- insert each *c* into priority queue *Q* using *f[c]* as key
- for *i* := 1 to *n-1* do
  - x := extract-min(*Q*)
  - y := extract-min(*Q*)
  - make a new node *z* with left child *x* (label edge 0), right child *y* (label edge 1), and *f[z]* = *f[x]* + *f[y]*
  - insert *z* into *Q*
- Time Complexity: O(nlgn)