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 some problems where it does not yield an optimal solution it 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

Total Weighted Time to Completion Problem

Total Weighted Time to Completion Problem: Given a set of jobs: job_1, job_2, …, job_n characterized by their lengths: l_1, l_2, … , l_n and their weights: w_1, w_2, … , w_n

- Output a schedule that minimizes the total weighted time to completion.
  - Schedule: An ordering of the jobs to be serviced by the resource
  - Completion time: C_i, of job l_i is the sum of the l_j's up to and including l_i
- Total Weighted Time to Completion $= \sum_{i=1 \text{ to } n} w_i\, C_i$

Cal Poly
Computer Science Department

3

Question?

- Given 3 jobs job_1, job_2, job_3 where the respective lengths are 1, 2, 3. What are the Completion times of the three jobs if done in the order given?
  - 1, 2, 3
  - 3, 5, 6
  - 1, 3, 6
  - 1, 4, 6
- If the weights are respectively 3, 2, 1. What is the Total Weighted Time to Completion for this ordering?

- Note: There are n! orderings.

Cal Poly
Computer Science Department

4

Total Weighted Time to Completion Problem (TWTC)

Finding a Greedy algorithm that solves this problem:
Focus on process

- Goal: Find a sequence of a set of jobs that **minimizes** the total weighted time to completion.
- There may be many candidate greedy algorithms!
- How do we find a greedy algorithm that works.
  - Look at special cases
  - Generate candidate greedy algorithms
  - Narrow down to single (if possible) candidate, usually by looking at simple cases to eliminate candidates, then prove correctness. (It may be that no greedy algorithm works.)
  -

Cal Poly
Computer Science Department

5

Discovery of candidates

- Explore special cases:
  - Equal lengths
  - Equal weights

- What order do you think you should schedule jobs?
  - Larger weights first; shorter lengths first
  - Smaller weights first; shorter lengths first
  - Larger weights first; longer lengths first
  - Smaller weights first; longer lengths first

- Hint: Look at small examples to test your ideas

Cal Poly
Computer Science Department

6

What are reasonable candidates?

- How can we combine the information we have into a greedy algorithm
- Two simple options, order the jobs by either:
  - Difference: $w_i - l_i$ Ratio: w_i / l_i
- How to rule one of these out?
  - Find, if possible, an example where the two different algorithms give two different answers.
  - Clearly the larger solution for the example is not optimal.
  - This rules out that algorithm but does not prove anything about the algorithm that gives the smaller solution.

Cal Poly
Computer Science Department   7

TWTC example problem

Consider 2 jobs $l_1 = 1, w_1 = 5$ $l_2 = 4, w_2 = 12$

- What is the weighted total time to completion given by the two algorithms?

Cal Poly
Computer Science Department   8

2 jobs $l_1 = 1, w_1 = 5$ $l_2 = 4, w_2 = 12$

- What is the weighted total time to completion given by the two algorithms?

	$W_i - L_i$	W_i / L_i
Job$_1$	4	5
Job$_2$	8	3
Ordering	Job$_2$, Job$_1$	Job$_1$, Job$_2$
TWTC	48+25 = 73	5+60 =65

- $W_i - L_i$ does not always yield the optimal (smallest) TWTC.

Proving correctness: exchange argument

Plan: Assume the greedy algorithm does not produce the optimal solution.

- Assume the w/l ratios are all distinct so there is only one possible ordering given by the algorithm.
- Assume that the jobs are labeled according to the decreasing w/l ratio, e.g. w_1/l_1 is the largest ratio, w_n/l_n is smallest
- Assume that there is an ordering Σ^* that is the optimal solution but is different from Σ the ordering given by the algorithm

Now show that Σ^* that is not the optimal solution.

This contradicts our assumption that it is optimal and different from Σ. Hence the optimal solution must be Σ.

Proving correctness: exchange argument details

the jobs are labeled according to the decreasing ratios

If Σ^* is not Σ, it must have two consecutive jobs where i > j but i comes before j in Σ^*. Why??

Order of jobs in Σ^*

Jobs before i	Job i	Job j	Jobs after j

If switch jobs i and j, what happens to the completion times of jobs?

Cal Poly
Computer Science Department

11

Proving correctness: exchange argument details

Jobs before i	Job i	Job j	Jobs after j

If switch jobs i and j, what happens to the completion times of jobs?

    k $\neq$ i, j   i   j

– unaffected   up   down

- Now what does switching i and j do to the total cost?
  - Since i > j we know that w_i / l_i < w_j / l_j so
  - $w_i * l_j$ < $w_j * l_i$ BUT
  - the ordering after making the switch has a lower total weighted time to completion! CONTRADICTION that Σ^* was the minimum total weighted time to completion

Cal Poly
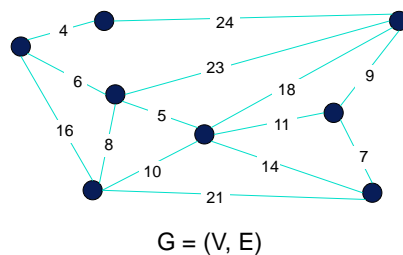Computer Science Department

12

Minimum Spanning Tree Problem

- *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 (sum of the edge weights in the tree)

- Minimum Spanning Tree Problem: For a weighted graph, find the MST

Minimal Spanning Tree Example

G = (V, E)

Minimal Spanning Tree Example

G = (V, E)   T, $\Sigma_{e \in T} c_e = 50$
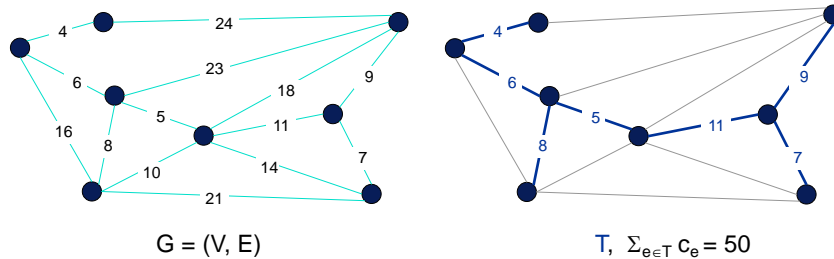
3 Greedy Approaches – these all work!

- Build a spanning tree by
    - Successively adding the lightest edge that does not create a cycle (Kruskal's)

    - Start with a root node and grow the tree outward without creating a cycle (Prim's)

    - Start with the full graph and delete edges in order of decreasing cost as long as we do not disconnect the tree (Reverse-delete?)
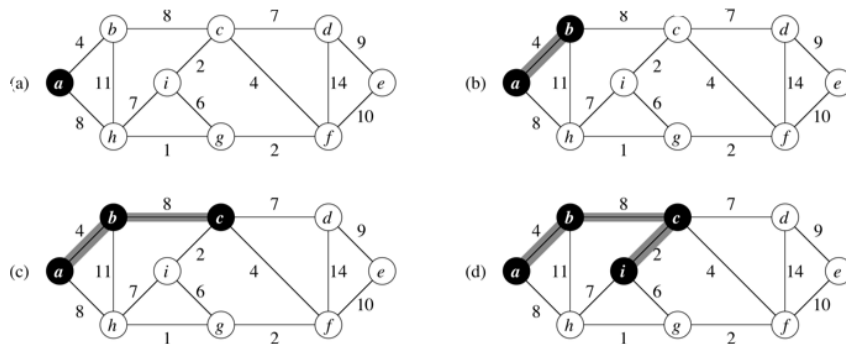
Prim's MST algorithm

- 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, ..., 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!)
  - Key Question – how do we do this efficiently

- Stop when all vertices are included

Cal Poly
Computer Science Department

17

Example of Prim's Algorithm in action

Cal Poly
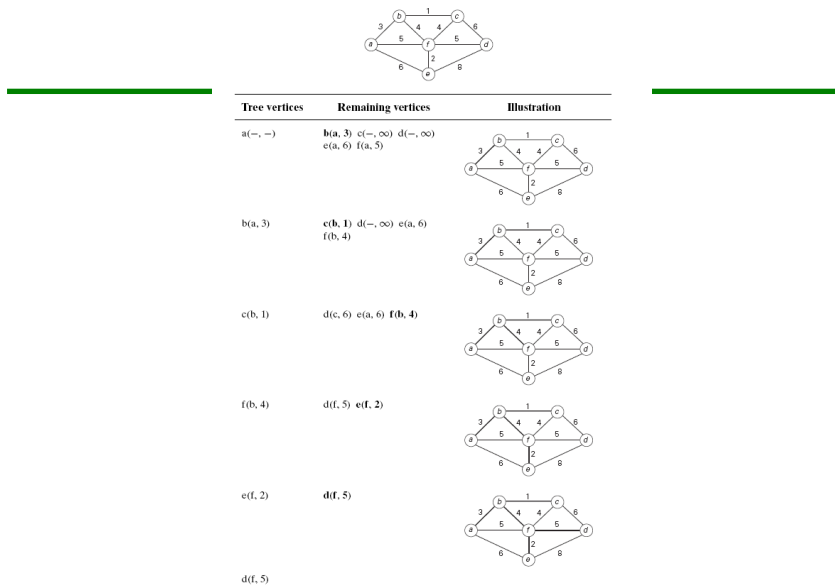Computer Science Department

18

Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

FIGURE 9.3 Application of Prim's algorithm. The parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are shown in bold.

19

Prim's Algorithm: More details

Initialize prev(v), dist(v) $\forall$ v∈V; T= {start vertex}; U = V-T; dist(start vertex) = 0
while(T is missing a vertex)
   pick the vertex, v1, in U with the shortest edge, dist(), to the group of vertices in
      the spanning tree add v1 to T
   /* this loop looks through every neighbor of v and checks to see if that
    * neighbor could reach the minimum spanning tree more cheaply through v1 */
   for each edge of v1 (v1, v2)
      if(length(v1, v2) < dist[v2])
         dist[v2] = length(v1, v2)
         prev[v2] = v1
      end if
   end for
end while

Cal Poly
Computer Science Department

20

Algorithm: Prim-MST (G)
Input: Graph G=(V,E) with edge-weights.
// Initialize priorities and place in priority queue.
1. priority[i] = infinity for each vertex i
2. Insert vertices and priorities into priorityQueue;
 // Set the priority of vertex 0 to 0.
3. priorityQueue.decreaseKey (0, 0) //

// Process vertices one by one in order of priority

4. while priorityQueue.notEmpty()
      // Get "best" vertex out of queue.
5.   v = priorityQueue.extractMin()
6.   Add v to MST;
         // Explore edges from v.
7.   for each edge e=(v, u) in adjList[v]
8.   w = weight of edge e=(v, u);

// If it's shorter to get to MST via v, then update.
9.   if priority[u] > w
10.   priorityQueue.decreaseKey (u, w)
11.   predecessor[u] = v
12.   endif
13.   endfor
14. endwhile

15. Build MST;
16. return MST

Algorithm: Dijkstra-SPT (G, s)
Input: Graph G=(V,E) with non-negative edge weights
// Initialize priorities and place in priority queue.
1. priority[i] = infinity for each vertex i;
2. Insert vertices and priorities into priorityQueue;
   // Source s has priority 0
3. priorityQueue.decreaseKey (s, 0)

// Process vertices one by one in order of priority

4. while priorityQueue.notEmpty()
      // Get "best" vertex out of queue.
5.   v = priorityQueue.extractMin()
6.   Add v to SPT;
         // Explore edges from v.
7.   for each edge e=(v, u) in adjList[v]
8.   w = weight of edge e=(v, u);

// If it's shorter to get to u from s via v, update.
9.   if priority[u] > priority[v] + w
10.   priorityQueue.decreaseKey (u, priority[v]+w)
11.   predecessor[u] = v
12.   endif
13.   endfor
14. endwhile

15. Build SPT;
16. return SPT

21

Cut property – proving MST algorithms correct

Assume all the edge costs are distinct. Let
- S be non-empty proper subset of V
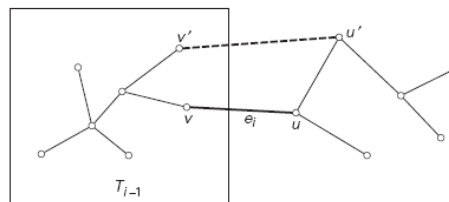- e = (v,w) the minimal cost edge between S and V-S
- then every MST contains e

FIGURE 9.4 Correctness proof of Prim's algorithm.

Cut property – proving MST algorithms correct

Assume all the edge costs are distinct. Let
- S be non-empty proper subset of V
- e = (v,w) the minimal cost edge between S and V-S
- then every MST contains e

Proof: (sketch – fill in the details)
- Let T be a spanning tree that does not contain e
- Use an exchange argument
- Find an e' so that exchanging e for e' reduces the cost
  » There must be a path from v to w in T
  » Follow the path to find an edge that goes from S to S-V
  » Let that be e', exchange to get T'
  » Finally show: T' is a tree (acyclic) and has lower cost

Correctness and Analysis: Prim's Algorithm

- Correctness : apply the cut property at each stage of the algorithm
- Analysis: Using a heap implementation of a **priority queue** the algorithm will run:
  - O(|E|) since each edge will eventually be placed in the queue
  - |V| * extractMin and |E| * changePriority operations
  - A priority queue can be implemented using a heap to so that both extractMin and changePriority are O(log |V|)
  - Therefore the overall running time is |E| * (log |V|)
- Why is it not proper to use the priority queue operations as the basic operations?

Developing and remembering Prim's Algorithm - 1

Big idea: grow a tree adding one vertex (closest) at a time -- Refinements

1. How find next vertex? -- what needs to happen to set up for finding the next vertex?
2. What are the operations needed?
3. What are possible data structures?
4. What should control flow look like?

Cal Poly
Computer Science Department

25

Developing and remembering Prim's Algorithm - 2

Try to make it concrete
- Keep track of vertices in a list and associate with each the "edge connecting it to the tree and the edge weight"
- Store the graph either as an adjacency matrix or an adjacency list
- Initialize the list of vertices, mark the start vertex with nil, 0 since it has no edge and no weight
- Traverse the adjacency list of the start vertex and update distances and edges
- Loop till all the vertices are connected – n-1
  - remove the nearest vertex and mark it in the tree
  - update the distances and edges of vertices in the adjacency list

Cal Poly
Computer Science Department

26

Single Source Shortest Paths Problem

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

- Variants:
  - Single destination shortest paths problem
  - Single pair shortest paths problem
  - All pairs shortest paths problem
- Issues
  - Negative edge weights
  - Cycles

Single Source Shortest Paths Problem
Dijksta's algorithm

- Doesn't BFS solve shortest paths?
  Yes, but only if the edge costs are all 1

- Why not just add edges to get the weight? (Good Question, can you use what you already know.)

Dijksta's algorithm pseudo-code
single source all shortest paths

- Doesn't BFS solve shortest paths?
  Yes, but only if the edge costs are all 1
- Why not just add edges to get the weight? (Good Question, can you use what you already know.)

- But problem is you may increase the number of edges significantly

Cal Poly
Computer Science Department

29

Three Shortest Path Algorithms (& Prim MST)
Depend on idea of relaxation

- Dijkstra
  - Edge weights must be non-negative
  - Relaxes each edge exactly once. O ((log |V|)*|E|)
- Bellman-Ford
  - Works with negative edge weights (Of course not cycles!)
  - Relaxes edges multiple times O (|V|* |E|)
- DAG - assumes no cycles!
  - Uses topological sort of vertices
  - Guarantees O(|E|+|V|) performance
  - Works with negative edge weights

Cal Poly
Computer Science Department

30

Relaxation:

- For each vertex, v, maintain an upper bound on the length (stored in dist(v)) of the shortest path from the source vertex to that vertex and the previous vertex on that shortest path
- Test to see if some new path is shorter than the current upper bound. If so then reduce the upper bound and update the previous vertex.

Relax(e=(u,v), w)

if dist(v)>dist(u)+w(u,v)

dist(v) = dist(u)+w(u,v)

prev(v) = u

Cal Poly
Computer Science Department

31

Dijkstra's Algorithm

- Dijkstra's algorithm.
  - Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
  - Initialize S = { s }, d(s) = 0.
  - Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e,$$

shortest path to some u in explored part, followed by a single edge (u, v)

add v to S, and set d(v) = π(v).
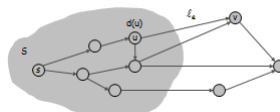
32

Cal Poly
Computer Science Department

Dijkstra's Algorithm

- Dijkstra's algorithm.
  - Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
  - Initialize S = { s }, d(s) = 0.
  - Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v)\,:\,u \in S} d(u) + \ell_e,$$

  shortest path to some u in explored part, followed by a single edge (u, v)

  add v to S, and set d(v) = π(v).

33

Cal Poly
Computer Science Department

Dijkstra's Algorithm: Implementation

- For each unexplored node, explicitly maintain $\pi(v) = \min_{e = (u,v)\,:\,u \in S} d(u) + \ell_e$.
  - Next node to explore = node with minimum π(v).
  - When exploring v, for each incident edge e = (v, w), update

$$\pi(w) = \min \{ \; \pi(w), \; \pi(v) + \ell_e \}.$$

- Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by π(v).

PQ Operation	Array	Binary heap	d-way Heap	Fib heap †
Insert	n	log n	d log $_d$ n	1
ExtractMin	n	log n	d log $_d$ n	log n
ChangeKey	1	log n	log $_d$ n	1
IsEmpty	1	1	1	1
Total	n^2	m log n	m log $_{m/n}$ n	m + n log n

34

Cal Poly
Computer Science Department

† Individual ops are amortized bounds

17

Representing Shortest Paths

- For each vertex u maintain the previous vertex from the source to that vertex, prev(u)
- Can reconstruct the shortest path to any vertex by walking backwards from the vertex to the source

```
Initialize_Single_Source(V)
    for v ∈ V
        dist(v) = ∞
        prev(v) = null
    dist(s) = 0
```

Dijksta's algorithm pseudo-code
single source all shortest paths

For every vertex dist [v]← ∞ as distance; prev[v]= null;

Dist[source] = 0;

Q = V (Q will be the set of unfinished vertices)

$V_T \neq \varnothing$

while Q is not empty.

      // Loop invariant: V_T = { v: know shortest path}
      // each time through finds shortest path to vertex
      // and adds edge to V_T

Dijksta's algorithm pseudo-code
single source all shortest paths

- For every vertex dist [v]← ∞ as distance; prev[v]= null;
- Dist[source] = 0;
- Q = V (Q will be the set of unfinished vertices)

- $V_T \neq \varnothing$
- while Q is not empty. // Loop invariant: V_T = { v: know shortest path}
  // each time through finds shortest path to vertex and adds to V_T
  - u be vertex with smallest distance to source in Q, remove u* from Q
  - $V_T = V_T \cup \{u^*\}$
  - for all of its unfinished neighbors v
    » calculate their tentative distance to source through current node --- u*.
    » If this distance is less than the previously recorded tentative distance of v, then overwrite dist[v] and update prev[v]

Cal Poly
Computer Science Department

37
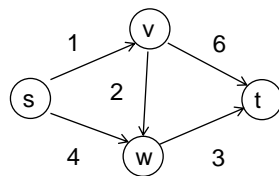
Simple Examples

What if negative edges lengths?
-- What if run Dijkstra's on this?
-- Add constant to each edge?

Works!
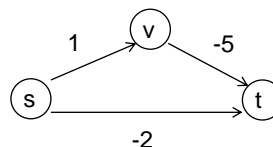
Cal Poly
Computer Science Department

38

Dijksta's algorithm – proof of correctness

- Theorem: For every directed graph with non-negative edge lengths, Dijkstra's algorithm correctly computes all shortest path distances

Proof by induction:

Picture

Dijksta's algorithm – proof of correctness

Proof by induction: mark nodes with a * once shortest path has been found

base case: - empty path is optimal

Inductive case: all previous iterations correct → next iteration correct

1. Dijkstra picks next node to add to the shortest path tree w by finding the w with the smallest length path given by dist(v*) + l(v*, w) where it has already found the shortest path to v*
   ----   must show this is the shortest path of all possible paths from s to w*.
   Know the shortest path to v* has been found in a previous iteration by the inductive hypothesis

2. Suppose there is a shorter path, P_2, s to w that has not been found yet. Any path from s to w must cross the frontier between nodes whose shortest paths are known and other nodes. Let the edge where P_2 crosses the frontier be (y*, z) thus there is a path from s to y* followed by the edge (y*, z), then a path from z to w.

3. But the the last section of P_2 must be ≥0 (since all edge lengths are non-negative).

4. Finally consider the path length of the part of P_2 from s to z given by dist(s to y*)+ l(y*, z). But the path from s to z that is part of the proposed better path from s to W* must be ≤ shortest path from s to w. Why?. This contradicts that the vertex chosen by Dijkstra is the closest vertex to s among all the vertices one edge away from the vertices for which the shortest path is already known.

Copyright ©The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
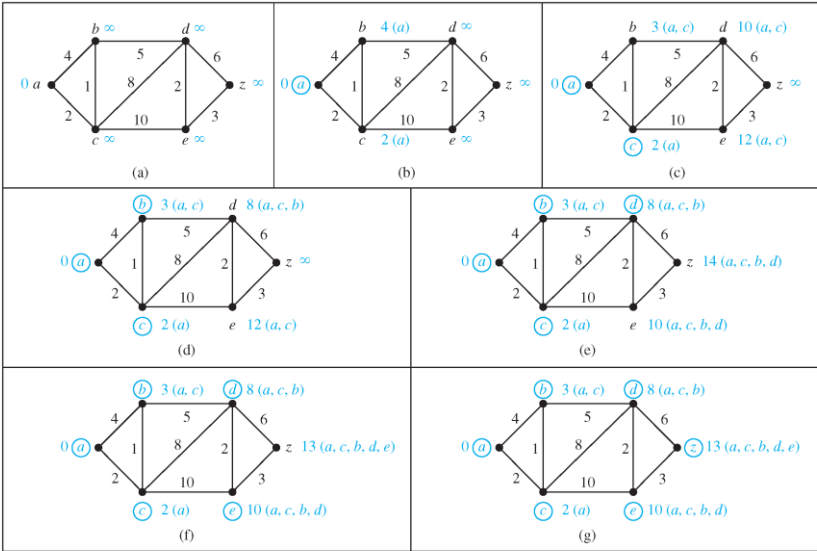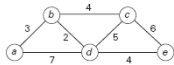Kenneth H. Rosen, *Discrete Mathematics and its Applications*, 7e

FIGURE 4   Using Dijkstra's Algorithm to Find a Shortest Path from *a* to *z*.

Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

from *a* to *b* :   *a − b*   of length 3
from *a* to *d* :   *a − b − d*   of length 5
from *a* to *c* :   *a − b − c*   of length 7
from *a* to *e* :   *a − b − d − e*   of length 9

FIGURE 9.11 Application of Dijkstra's algorithm. The next closest vertex is shown in bold.

42

Notes on Dijkstra's algorithm

- Doesn't work for graphs with negative weight
- 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
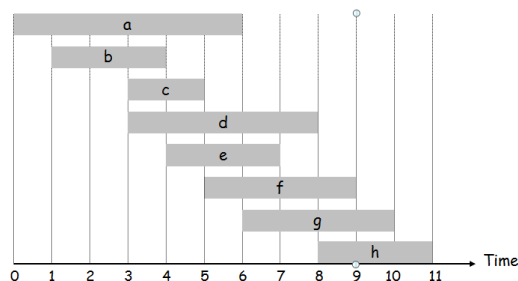- Don't mix up Dijkstra's algorithm with Prim's algorithm!

Cal Poly
Computer Science Department

43

Interval Scheduling

- Interval scheduling.
    - Job j starts at sj and finishes at fj.
    - Two jobs compatible if they don't overlap.
    - Goal: find maximum subset of mutually compatible jobs.

Cal Poly
Computer Science Department

44

Interval Scheduling: Greedy Algorithms

- Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.
  - [Earliest start time] Consider jobs in ascending order of start time s_j.
  - [Earliest finish time] Consider jobs in ascending order of finish time f_j.
  - [Shortest interval] Consider jobs in ascending order of interval length $f_j - s_j$.
  - [Fewest conflicts] For each job, count the number of conflicting jobs c_j. Schedule in ascending order of conflicts c_j.

Cal Poly
Computer Science Department

45

Interval Scheduling: Greedy Algorithms

- Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

breaks earliest start time

breaks shortest interval

breaks fewest conflicts

Cal Poly
Computer Science Department

46

Interval Scheduling: Greedy Algorithm

- Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken. (Earliest finish time first)

```
Sort jobs by finish times so that f₁ ≤ f₂ ≤ ... ≤ fₙ.

   jobs selected

A ← φ
for j = 1 to n {
    if (job j compatible with A)
        A ← A ∪ {j}
}
return A
```

- Implementation. O(n log n).
  - Remember job j* that was added last to A.
  - Job j is compatible with A if sj ≥ fj*.

Cal Poly
Computer Science Department

47

Interval Scheduling: Analysis

- Theorem. Greedy algorithm is optimal.
- Pf. (by contradiction)
  - Assume greedy is not optimal, and let's see what happens.
  - Let i_1, i_2, ... i_k denote set of jobs selected by greedy.
  - Let j_1, j_2, ... j_m denote set of jobs in the optimal solution with $i_1 = j_1$, $i_2 = j_2$, ..., $i_r = j_r$ for the largest possible value of r.
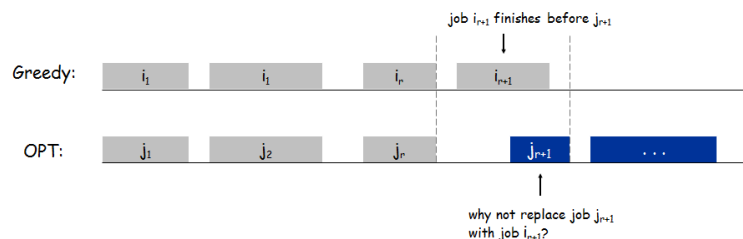
Cal Poly
Computer Science Department

48

24

Interval Scheduling: Analysis

- Theorem. Greedy algorithm is optimal.
- Pf. (by contradiction)
  - Assume greedy is not optimal, and let's see what happens.
  - Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
  - Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.
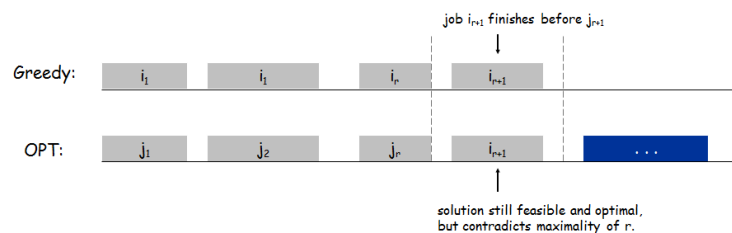
Computer Science Department   49

Another greedy algorithm for MST: Kruskal's

- Sort the edges in non-decreasing order of lengths

- "Grow" tree one edge at a time to produce MST through a series of expanding forests $F_1, F_2, \ldots, F_{n-1}$

- On each iteration, add the next edge on the sorted list unless this would create a cycle. (If it would, skip the edge.)

Cal Poly
Computer Science Department   50
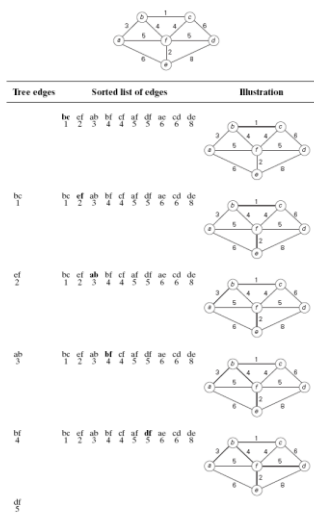
FIGURE 9.5 Application of Kruskal's algorithm. Selected edges are shown in bold.

Cal Poly
Computer Science Department

51

Notes about Kruskal's algorithm

- Algorithm looks easier than Prim's but is harder to implement (checking for cycles!)

- Cycle checking: a cycle is created if and only if added edge connects vertices in the same connected component

- *Union-find* algorithms – see section 9.2

Cal Poly
Computer Science Department

52

Correctness and Analysis: Kruskal's Algorithm

- Correctness : apply the cut property at each stage of the algorithm
- Analysis: Assuming n vertices and m edges. Using Union, Find and MakeSet operations discussed in the text with an efficient implementation
  - the algorithms order of complexity is dominated by the sorting operation on the edges hence it is $O(|E| * (\log |E|)$

Bellman Ford Algorithm

```
Bellman-Ford (G, w, s)// w: weights
   Initialize_Single_Source(V)
   For i = 1 to |V|-1
        for each edge (u,v) in E
           Relax((u,v),w)
   for each edge (u,v) in E
     if dist(v) > dist(u) + w(u,v)
        return false  //negative cycle detected
```

Correctness of Bellman Ford Algorithm

- May want to defer till get to dynamic programming. Good lab to write out as a dynamic programming algorithm
- Proof of correctness is proof of optimal substructure

DAG Algorithm

- DAG implies no cycles – no need to worry about negative weight cycles
- If we topologically sort the vertices, then any path must be consistent with the topological sort, that is the u comes before v in the path ↔ u comes before v in the topological sort
- By making one pass over the vertices in the topological sort and relaxing the values of the vertices in their adjacency list we are guaranteed to find the shortest paths.

DAG algorithm pseudo code and efficiency

```
DAG-Shortest-Paths (G,w,s)
    Topologically sort vertices in G
    Initialize-Single-Source (G,s)
    For each vertex u // in topological sorted order
        For vertex v in the adjacency list of u
            Relax(u,v,w)
```

- Running time is $\theta(|E|+|V|)$ since topological sorting is $\theta(|E|+|V|)$ and the Relax function is called $\theta(|E|+|V|)$ times

Cal Poly
Computer Science Department

57

Shortest Path Problem

- Shortest path network.
  - Directed graph G = (V, E).
  - Source s, destination t.
  - Length ℓ_e = length of edge e.
  - cost of path = sum of edge costs in path
- Shortest path problem: find shortest directed path from s to t.
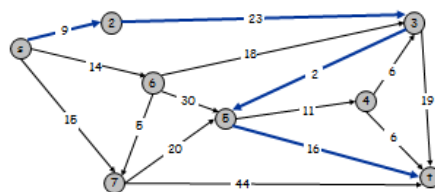
Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 48.

58

Cal Poly
Computer Science Department
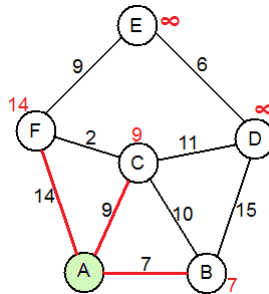
Cal Poly
Computer Science Department

A* (heuristic): Brief comparison

- A* uses finds a least-cost path from a given initial node to one goal node (out of one or more possible goals).
- It uses a distance-plus-cost heuristic function
  f(x) = g(x) + h(x)) to determine the order in which the search visits nodes in the tree.
  - g(x) – path cost function which is the cost from the starting node to the current node
  - h(x) an ***admissible*** "heuristic estimate" of the distance to the goal (usually denoted). It must not over-estimate the distance to the goal

Cal Poly
Computer Science Department   60

Coding Problem

- *Coding*: assignment of bit strings to alphabet characters
- *Codeword*: bit string assigned to character
- 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?

Cal Poly
Computer Science Department

61

Morse and ASCII codes

Appendix B: ASCII Codes

Printable 8-bit ASCII codes

191

University Publishing Online, hosted by Cambridge University Press © 2011

```
A ●—        J ●———      S ●●●
B —●●●       K —●—       T —
C —●—●       L ●—●●      U ●●—
D —●●        M ——        V ●●●—
E ●          N —●        W ●——
F ●●—●       O ———       X —●●—
G ——●        P ●——●      Y —●——
H ●●●●       Q ——●—      Z ——●●
I ●●         R ●—●
```

Cal Poly
Computer Science Department

62

Example

- Let Σ = { lower case letters, five punctuation marks and space}
- How can we encode this – 5 bits since $2^5 = 32$
- Is there a way to reduce the length not of the code for each symbol BUT the average length of a message.
- Use frequency of the occurrence of the symbols
  - e, t, a - most frequent   -- q, j, x, z – least frequent
  - Normalize so the sum of frequencies is = 1
  - Use frequencies to compute E(symbol length)
  - E.g. Morse code
- Prefix free codes are codes: for all symbols x and y – codeword(x) is not a prefix of codeword(y)
- Decoding is easy - - why?

Cal Poly
Computer Science Department   63

Huffman codes – key insights

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

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

.

Cal Poly
Computer Science Department   64

Example

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

codeword   11   100   00   01   101
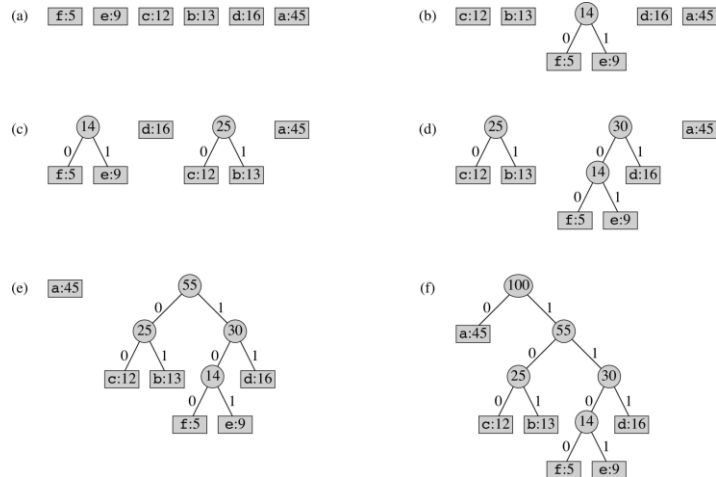
average bits per character: 2.25
for fixed-length encoding: 3
compression ratio: (3-2.25)/3*100% = 25%

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)
  - make the new tree weight equal the sum of the weights of the two subtrees.
- Mark edges leading to left and right subtrees with 0's and 1's, respectively

Constructing a Huffman code tree

(a) f:5 e:9 c:12 b:13 d:16 a:45

(b) c:12 b:13 (14) 0/ \1 f:5 e:9   d:16 a:45

(c) (14) 0/ \1 f:5 e:9   d:16   (25) 0/ \1 c:12 b:13   a:45

(d) (25) 0/ \1 c:12 b:13   (30) 0/ \1 c:12 b:13 (14) d:16 ... f:5 e:9   a:45

(e) a:45   (55) 0/ \1 (25) (30) ... c:12 b:13 (14) d:16 ... f:5 e:9

(f) (100) 0/ \1 a:45 (55) ... (25) (30) ... c:12 b:13 (14) d:16 ... f:5 e:9

Example: Build tree (smallest to left = 0)

character	A	B	C	D	E
frequency	0.32	0.25	0.2	0.18	0.05

Codewords:

H.C. average bits per character:

Fixed-length bits per character:

compression ratio =

Decode: 011110111011011

Extensions and issues

- Image compression
  - Fraction of a bit for a white pixel, higher for black pixel
  - Video/audio – only send changes
- Adaptive encoding

- Many schemes are more effective for particular applications

Cal Poly
Computer Science Department

69