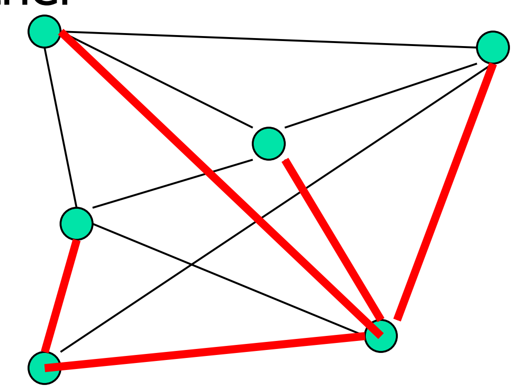
# Chapter 8
# Greedy Algorithms on Graphs

References:
[DPV 4.4-4.5]
[KT 4.4-4.6]
[CLRS 23, 24.3, 6]
[SSS 6.1, 6.3]

# Part 1: Minimum Spanning Trees

# The Problem

- Given a set of cities, how to connect them by road networks?
  - Result: all cities are reachable one-another
  - Shortest total length of roads
  - (does not necessarily give shortest
    length between a given pair of cities)



- Abstract model
  - Given: a connected undirected graph G
  - Edges have weights (e.g. distance)
  - Goal: a set of edges that connect all vertices with minimum total weight

# Minimum Spanning Trees

- A *spanning tree* of an undirected graph is a subset of edges such that the graph remains connected
  - Must be a tree (i.e. no cycles)
  - Reason: if there is a cycle, we can remove one edge in the cycle and all vertices remain connected
- A *minimum spanning tree* (MST) is a spanning tree with the minimum total edge weight
- How to find the MST?
  - Greedy algorithms?

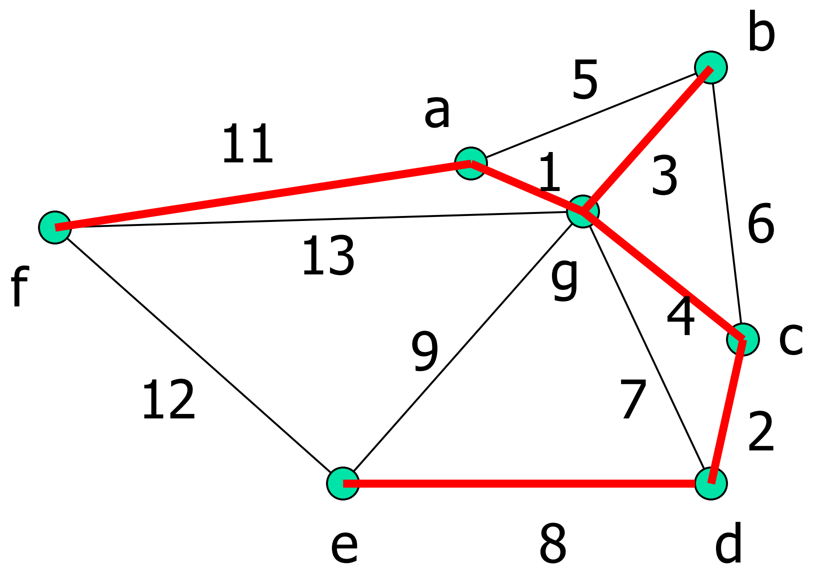
# Greedy Algorithm for MST

- Idea: repeatedly add the shortest unused edge
  - As long as it connects two different components, i.e. not redundant
- This is *Kruskal's algorithm*
  - High-level pseudocode: (many details to be filled in later)

```
MST-Kruskal(G) {
  E := sorted list of edges (by weight)
  T := G without edges
  while T has fewer than n-1 edges {
    Remove first edge e =(u,v) from E
    if (u, v in different components in T)
      add e to T
  }
}
```

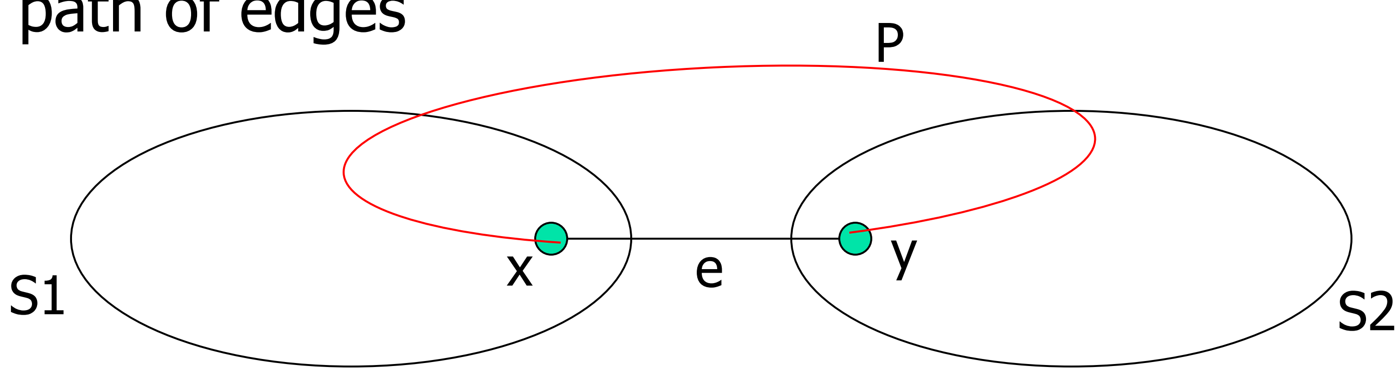
# Kruskal’s Algorithm: Example

- Edge labels = weights
  - (a, g): add
  - (c, d): add
  - (b, g): add
  - (c, g): add
  - (a, b): cycle, not add
  - (b, c): cycle, not add
  - (d, g): cycle, not add
  - (d, e): add
  - (e, g): cycle, not add
  - (a, f): add



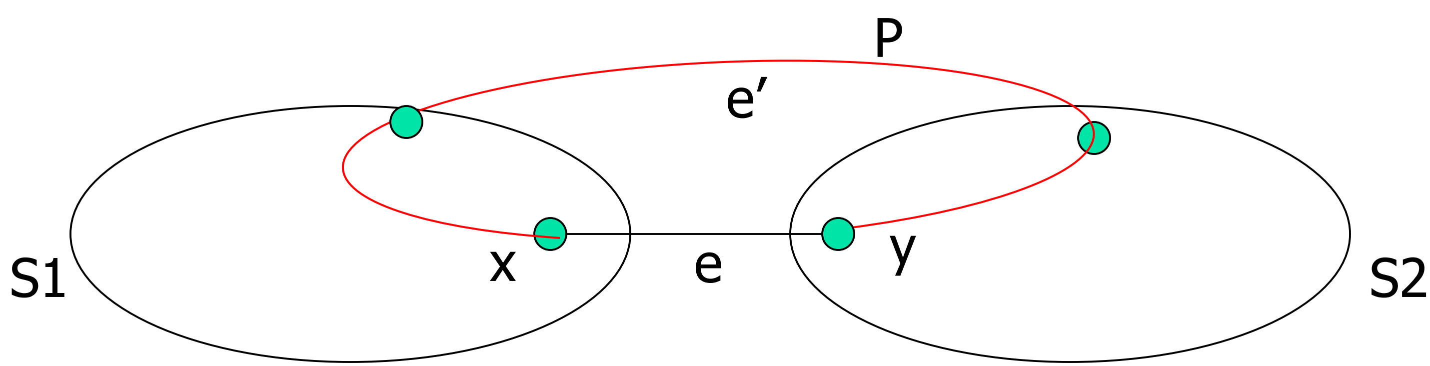
# Kruskal's Algorithm: Optimality

- Does it always produce the MST?
  - Certainly it is a spanning tree; is it minimum?
- Proof by contradiction. Sketch:
- Assume there is an edge e in the true MST T* but not in the tree produced by Kruskal's algorithm $T^K$
- Edge e separates T* into two parts, S1 and S2
- Since e is not in $T^K$, $T^K$ connects x and y by another path of edges



# Proof of Optimality (cont'd)

- In this path P, there is at least one edge e' connecting S1 and S2
- e' must have smaller weight than e since $T^K$ adds edges by increasing order of weights
- Hence we can remove e from T* and add e' to T* to get another tree with smaller weight
- This contradicts optimality of T*

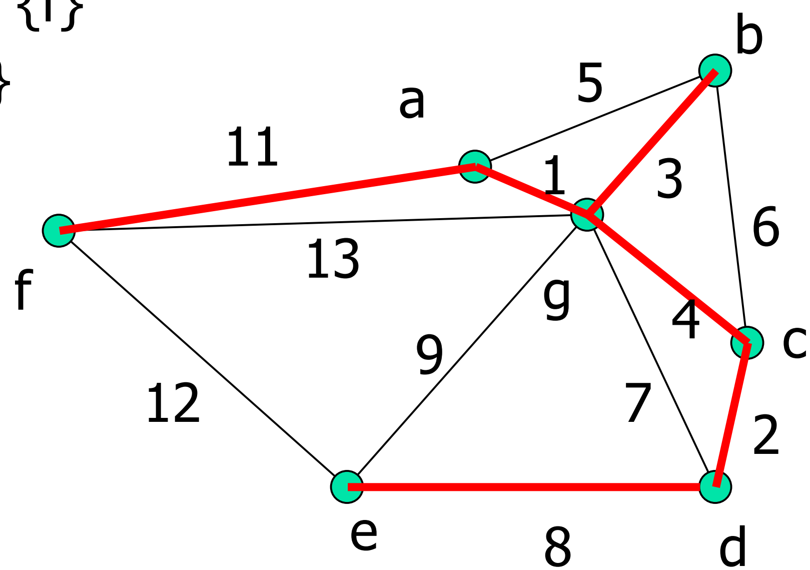


# Kruskal’s Algorithm: Running Time

- Sorting edge weights: $O(m \log m)$
  - $= O(m \log n)$ (recall $m = O(n^2)$)
- Inside while loop: depends on two important operation
  - 1) Checking whether two vertices are already connected (i.e. in the same component)
    - At most $O(m)$ times in total, over all executions of loop
  - 2) Connecting two vertices (components) together when an edge is added
    - At most $O(n)$ times in total, over all executions of loop (since the final MST has $n - 1$ edges only)
- How to perform these operations efficiently?

# Kruskal's using Disjoint Sets

- Representing connected components as *disjoint sets*
  - Initially: {a} {b} {c} {d} {e} {f} {g}
  - Add (a,g): {a,g} {b} {c} {d} {e} {f}
  - Add (c,d): {a,g} {b} {c,d} {e} {f}
  - Add (b,g): {a,b,g}, {c,d} {e} {f}
  - Add (c,g): {a,b,c,d,g} {e} {f}
  - Do not add (a,b) [same set]
  - Do not add (b,c)
  - Do not add (d,g)
  - Add (d,e): {a,b,c,d,e,g} {f}
  - Do not add (e,g)
  - Add (a,f): {a,b,c,d,e,f,g}



# Disjoint Set Data Structure

- We want a data structure for storing:
  - A number of disjoint sets
  - Each set contains elements from the same ground set
- And support the following operations efficiently:
  - Find(u): given an element u, return a "name" of the set containing that element
    - A simple name: some element in the set ("set representatives")
  - Union(u, v): merge two sets containing u and v
    - Often, u and v are already the set representatives

# Using Union-find in MST Algorithms

- Find() and Union() operations support the MST algorithm:
  - Each vertex begins as an individual set initially
  - Edge joining vertices $\rightarrow$ merge into same component $\rightarrow$ merge the disjoint sets
  - To check whether two vertices u, v belong to same component: just check whether Find(u) = Find(v)
  - To merge two vertices u and v, find their set representatives, and then merge: Union(Find(u), Find(v))
- We need data structures supporting the union and find operations

# Union-find Try #1: Array

- Use an array to keep track of the set names
  - A[i] = set name of element i
  - Example: {1,2,7} {3,4} {5} {6}

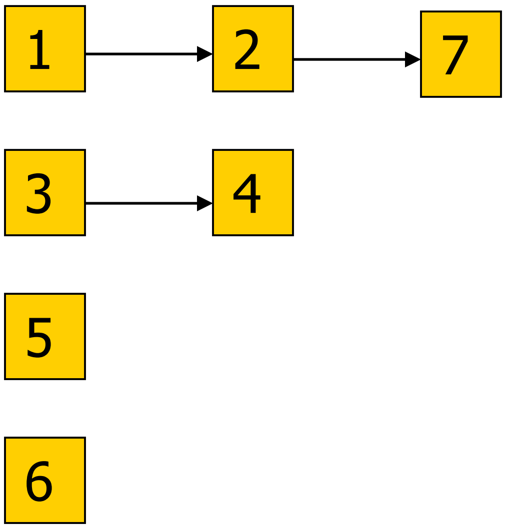
| A[i] | 1 | 1 | 3 | 3 | 5 | 6 | 1 |
|---|---|---|---|---|---|---|---|

- Find(i): just return A[i]
  - O(1) time
- Union(i, j): find the set names of i and j, change all array entries with those two names to the same name
  - O(n) time
  - Example: union(2, 5)

| A[i] | 1 | 1 | 3 | 3 | 1 | 6 | 1 |
|---|---|---|---|---|---|---|---|

# Union-find Try #2: Linked List

- Put all objects in same disjoint set (same name) as a list. Head of list is set representative
- Find(u): O(n) time (go through all lists)
- Union(u,v): can be done in O(1) time if use doubly-circular linked list



# Union-find #3: Array-and-Linked-List

- For each element, keep track of:
  - Set name
  - Pointer of next element in the set
  - Size of the set
- Size of set is kept because we use the *weighted-union heuristic:* always modify the smaller set
- Example:

Initially: {1} {2} {3} {4} {5} {6} {7}

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | ← Set name |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | ← Next element of the set, 0 = null |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | ← Size of set |

# Union-find Example (Cont'd)

Union(1,7): {1,7} {2} {3} {4} {5} {6}

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 1 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 1 | 1 | 1 | 1 | - |

7's name is now 1

1's next element is 7

Size of {1,7} is 2

Don't need to keep size anymore

Union(3,4): {1,7} {2} {3,4} {5} {6}

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 3 | 5 | 6 | 1 |
| 7 | 0 | 4 | 0 | 0 | 0 | 0 |
| 2 | 1 | 2 | - | 1 | 1 | - |

# Union-find Example (Cont'd)

Union(2,7): {1,2,7} {3,4} {5} {6}

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 1 | 3 | 3 | 5 | 6 | 1 |
| 2 | 7 | 4 | 0 | 0 | 0 | 0 |
| 3 | - | 2 | - | 1 | 1 | - |

{1,7} has size 2 while {2} has size 1, so only change set name of {2}

Union(3,7): {1,2,3,4,7} {5} {6}

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 5 | 6 | 1 |
| 3 | 7 | 4 | 2 | 0 | 0 | 0 |
| 5 | - | - | - | 1 | 1 | - |

Follow this virtual "linked list" to find entries to modify

Union(1,2): no change (Set names of 1,2 the same)

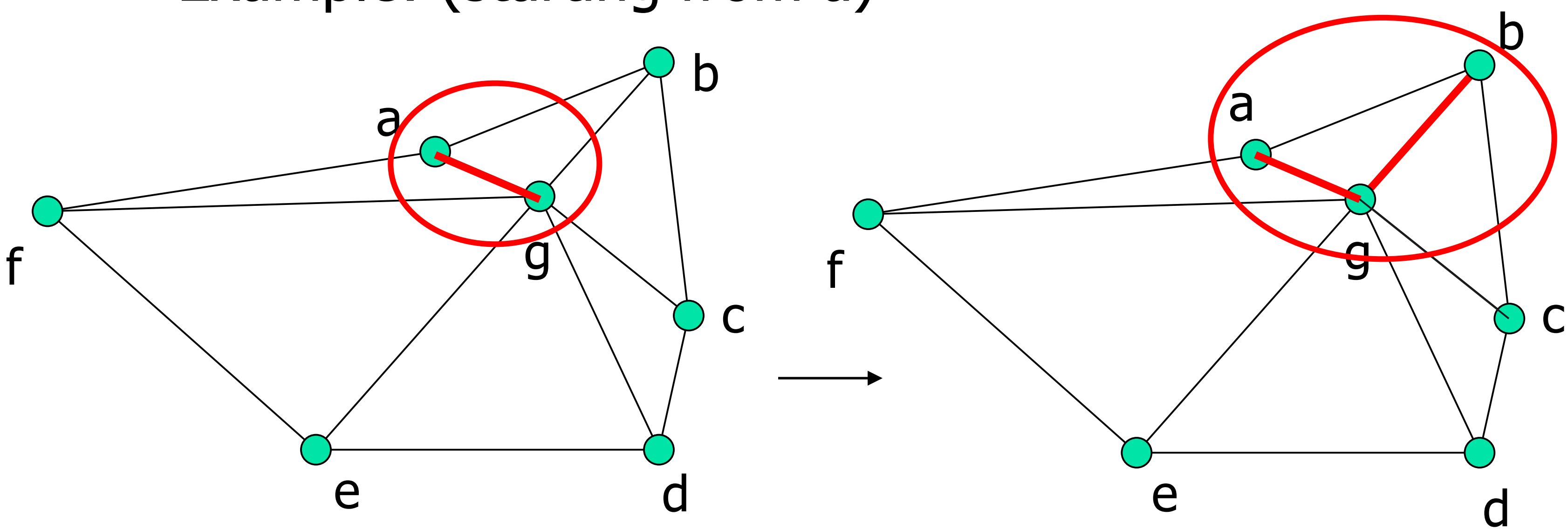
Union(2,3): no change… and so on

# Running Time of Kruskal's

- Using the array-and-linked-list data structure:
  - Find(u) in O(1) time
  - Union(u, v) is still worst-case O(n) time (update entries)
- Better analysis for Union():
  - Every time a union occurs, only the smaller set is modified
  - After modification, size of modified set at least doubled
  - Therefore, each entry can be modified at most log n times
  - Total time for n Union() = O(n log n)
- Overall running time
  - O(m log n) for sorting, O(m) for Find, O(n log n) for Union
  - Total O(m log n)
  - There are better data structures with faster running times

# Another Approach?

- Is there another way of “greedily” adding edges?
- Idea: “grow” the component from a vertex
- Only one partial MST when the algorithm is running, not a forest (as in Kruskal’s)
- Example: (starting from a)



# Prim’s Algorithm

- Algorithm:
  - At every step, find minimum-weight outgoing edge
  - Enlarge component
- Finding minimum outgoing edge:
  - Naïve approach: check all edges, O(mn) time in total
  - Better approach: keep track of distance of each node outside the growing component (S) from within S
  - Update when new vertex is added to S: only check edges going out from this new vertex

# Prim's Algorithm

```
Prim-MST(G, s) {
  /* starting vertex s
     assume d(,) stores edge weights */
  for each vertex v
    D[v] := d(s, v) // initialise

  S := {s}
  while (S != V) {
    find u in V - S with minimum D[u]
    add u to S
    for each v in V - S
      D[v] := min(D[v], d(u, v))
  }
}
```

# Example Operation of Prim's

- Suppose start at f:

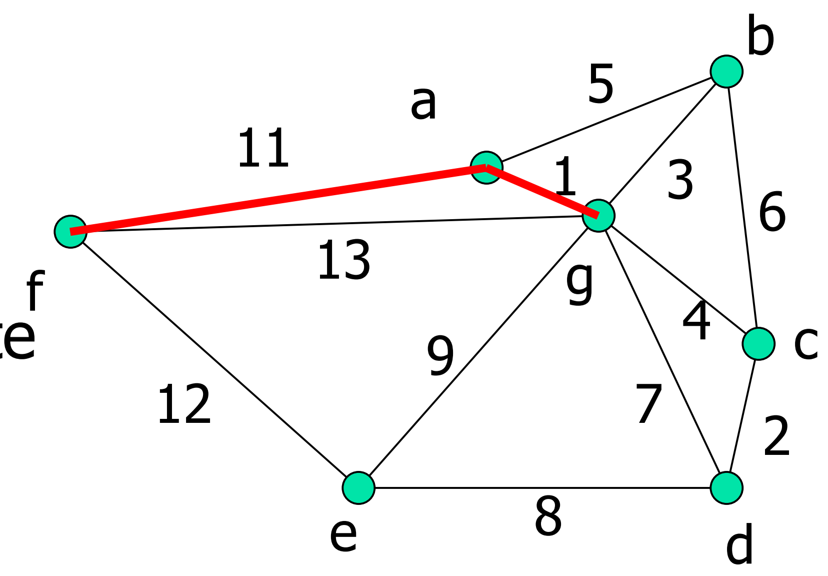
| v | a | b | c | d | e | g |
|---|---|---|---|---|---|---|
| D[v] | 11 | ∞ | ∞ | ∞ | 12 | 13 |

- Min. edge to a, add, update

| v | a | b | c | d | e | g |
|---|---|---|---|---|---|---|
| D[v] | / | 5 | ∞ | ∞ | 12 | 1 |

- Min. edge to g, add, update

| v | a | b | c | d | e | g |
|---|---|---|---|---|---|---|
| D[v] | / | 3 | 4 | 7 | 9 | / |



# Prim’s Example Operation (cont’d)

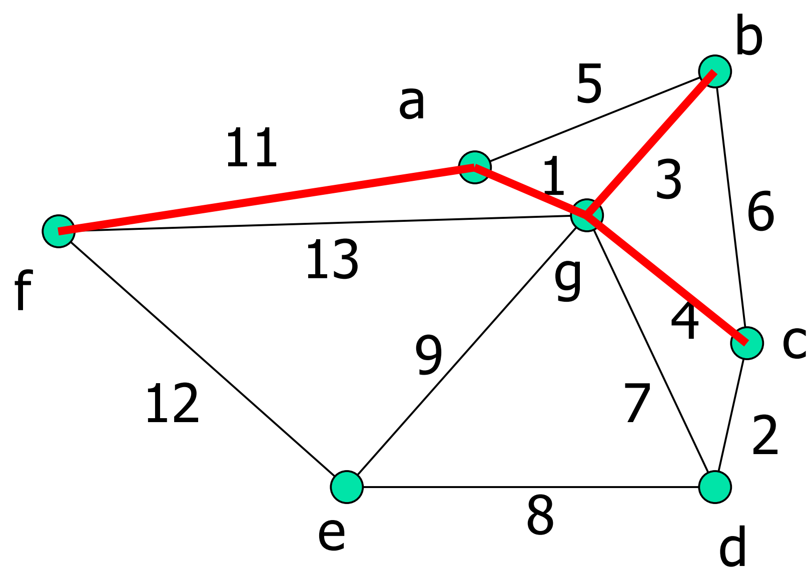
- Min. edge to b, add, update:

| v | a | b | c | d | e | g |
|---|---|---|---|---|---|---|
| D[v] | / | / | 4 | 7 | 9 | / |

- Min. edge to c, add, update:

| v | a | b | c | d | e | g |
|---|---|---|---|---|---|---|
| D[v] | / | / | / | 2 | 9 | / |

- … and so on…



# Running Time of Prim's

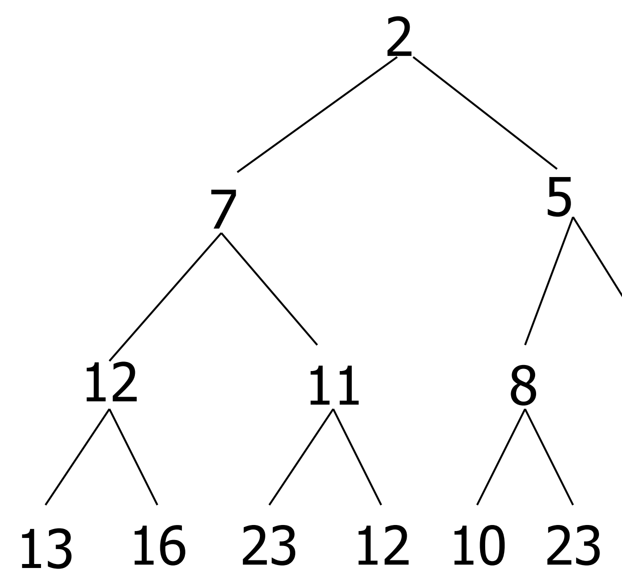
- Need two important operations:
  - *Find minimum:* O(n) times
  - *Change value:* O(m) times. Only consider the outgoing edges when a new vertex is added. O(m) edges, and edges would not be considered again
- Using an array
  - O(n) for finding minimum
  - O(1) for change value
  - Total $O(m + n^2) = O(n^2)$ time
- Using a *heap*
  - O(log n) time for both operations
  - Total O(m log n + n log n) = O(m log n) time

# Heaps

- Store elements in a *complete binary tree*
  - Left to right, do not start a new level unless it is full
  - *Balanced* (height = O(log n))
  - Parent ≤ both children (for min-heaps; max-heap is opposite)
- Can easily be represented as an array A[1..n], with all these in O(1) time:
  - Parent(A[i]) = $A[\lfloor i/2 \rfloor]$
  - Left-child(A[i]) = A[2i]
  - Right-child(A[i]) = A[2i+1]

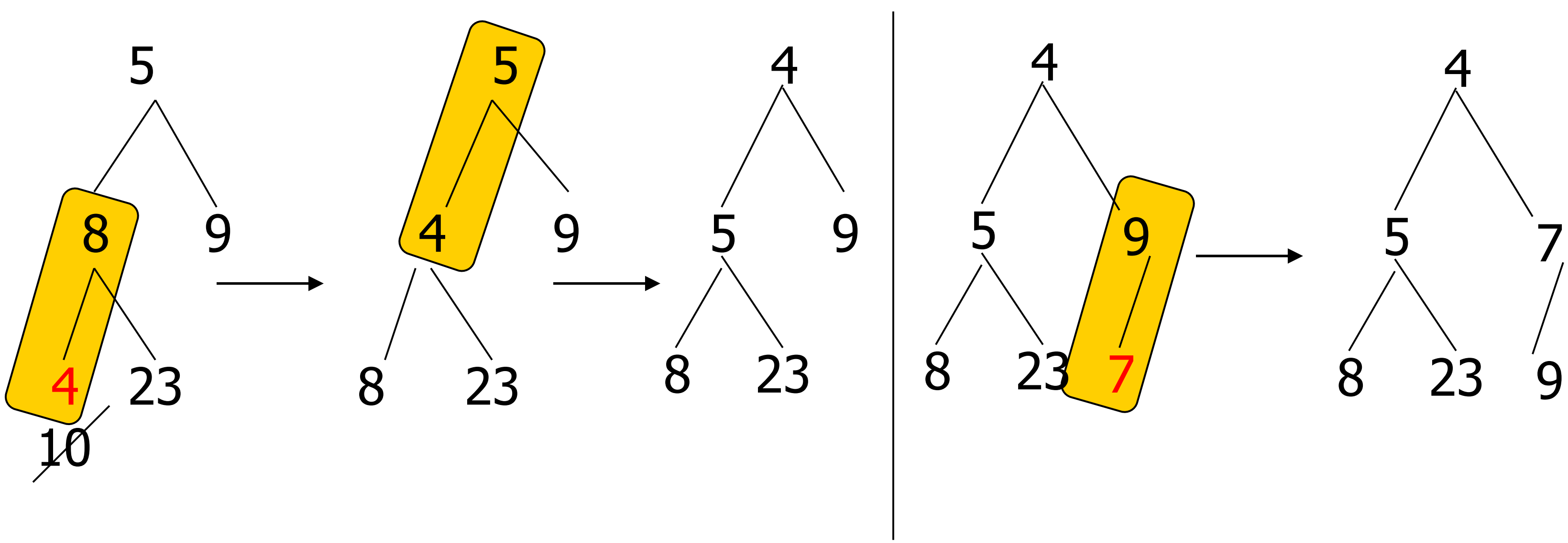
A

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 7 | 5 | 12 | 11 | 8 | 9 | 13 | 16 | ... |



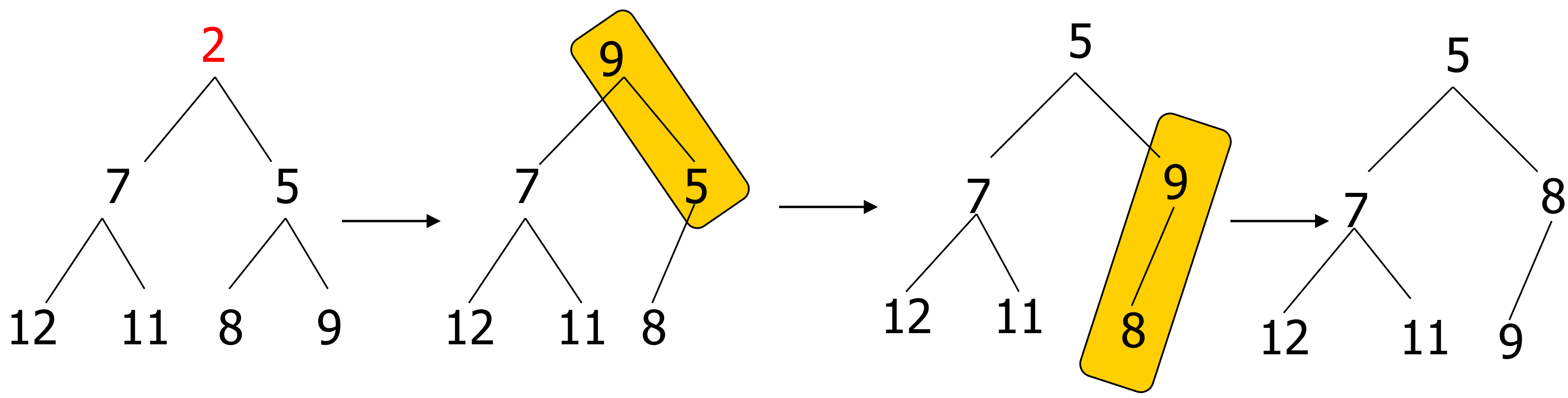
# Heap Operations

- Each in O(log n) time:
- (1) Decrease value
  - "Bubble up" (compare with parent and swap)
- (2) Insertion
  - Place at last available position, then similarly bubble up



# Heap Operations

- (3) Delete minimum
  - Always at root
  - Remove root and replace it with last element in heap
  - "Sift down" the heap



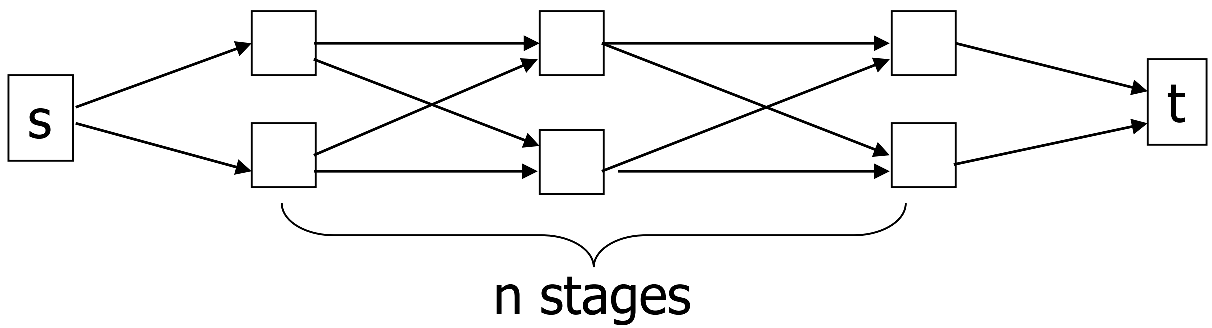
# Part 2: Shortest Paths

# The Shortest Path Problem

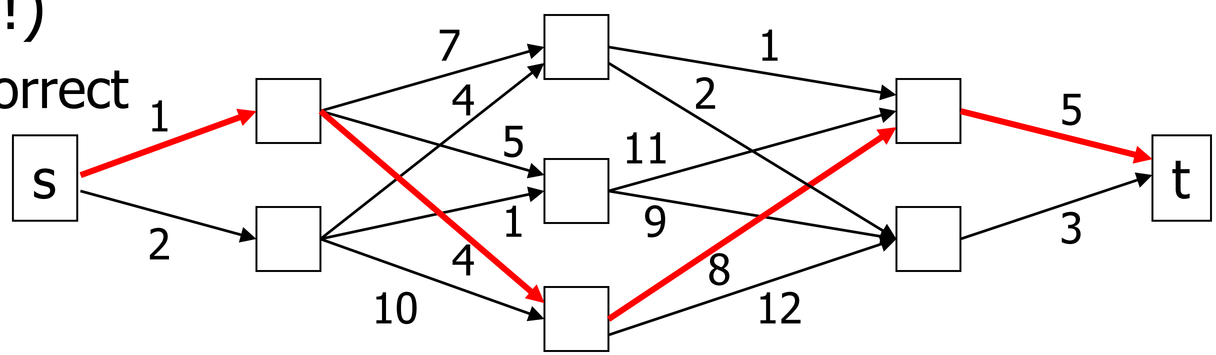
- Many problems have similar nature
  - What is the shortest way to go from Leicester to London?
  - What is the quickest way to send a packet between two computers?
  - …
- Abstract model
  - Given: a directed graph G with edge weights (e.g. distance), and a starting vertex s
  - Goal: find the shortest paths from s to all other vertices (*Single source shortest path*)
    - Path length = sum of edge weights
  - Not more difficult than just one s-t pair; same algorithms

# Naïve Approaches

- Try #1: try all possible paths, calculate distance and find the minimum one
  - Problem: there can be exponentially many paths! ($2^n$)

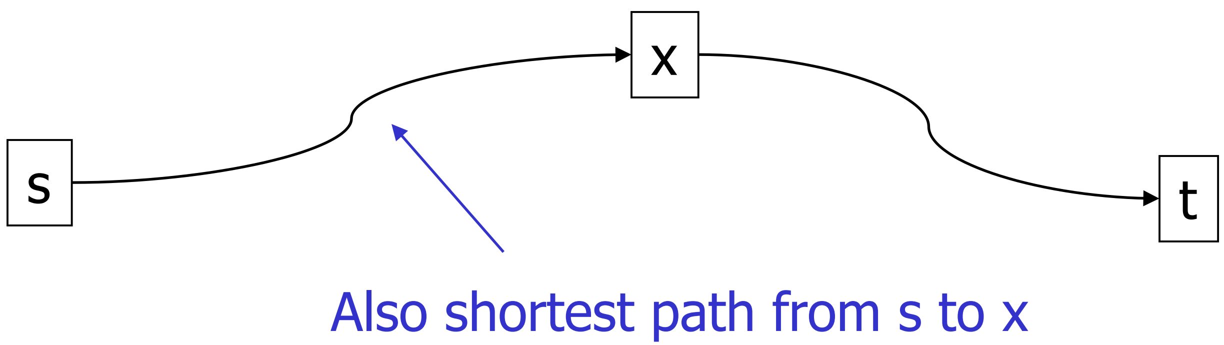



- Try #2: just pick the shortest edge at each step (greedy!)
  - Not correct



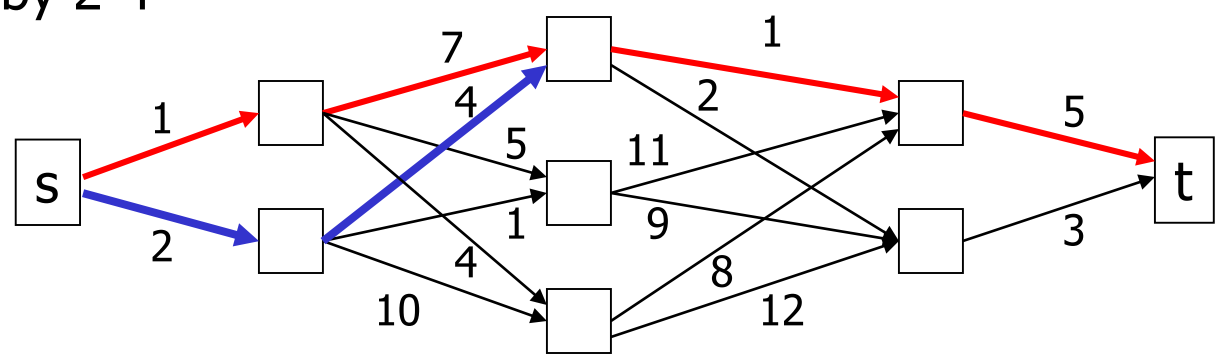
# Optimal Substructure

- Consider the shortest path from s to t
  - Suppose it goes through x
  - Then this path from s to x must be a shortest path from s to x, too
  - (otherwise, we can replace it with a shorter path, the whole s-t path would also be shorter)

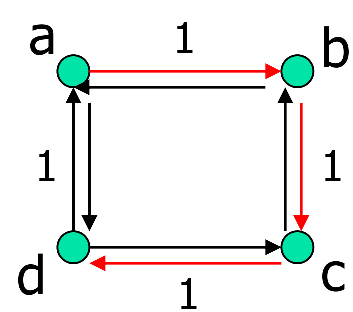


# Optimal Substructure

- Example:
  - The path 1-7-1-5 is not optimal, because 1-7 can be replaced by 2-4

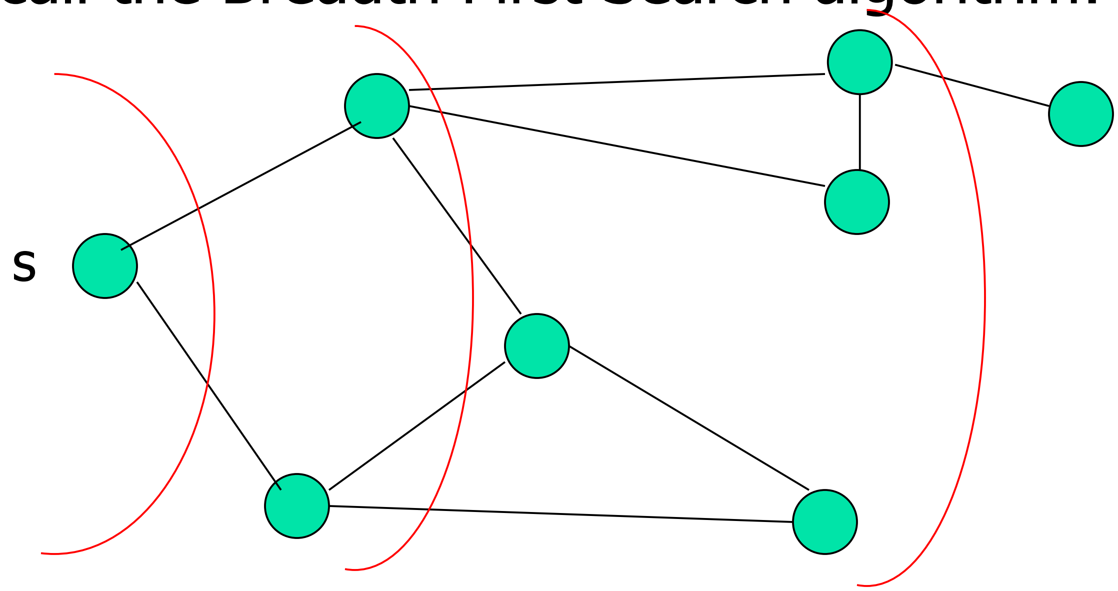



- Note that longest (simple) path does not have optimal substructure:
  - a-b-c-d is a longest path from a to d
  - but c-d is not longest path from c to d



# A Similar Idea to BFS

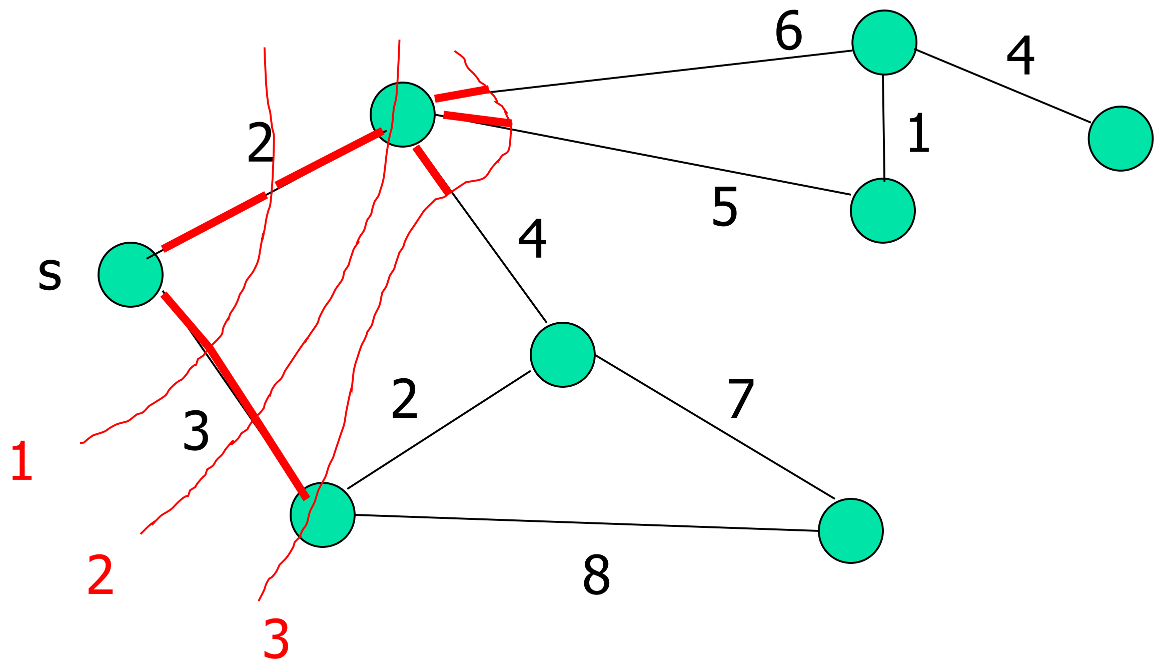
- Recall the Breadth First Search algorithm:




- In fact, BFS is a shortest path algorithm if all edges have weight 1
  - Extend wavefront by 1 unit each time until reach target

# Extending BFS…

- *Dijkstra's algorithm*: using a greedy idea
  - At every step, rather than extending the wavefront by 1 unit, we extend by the shortest "outgoing" edge
  - Example:



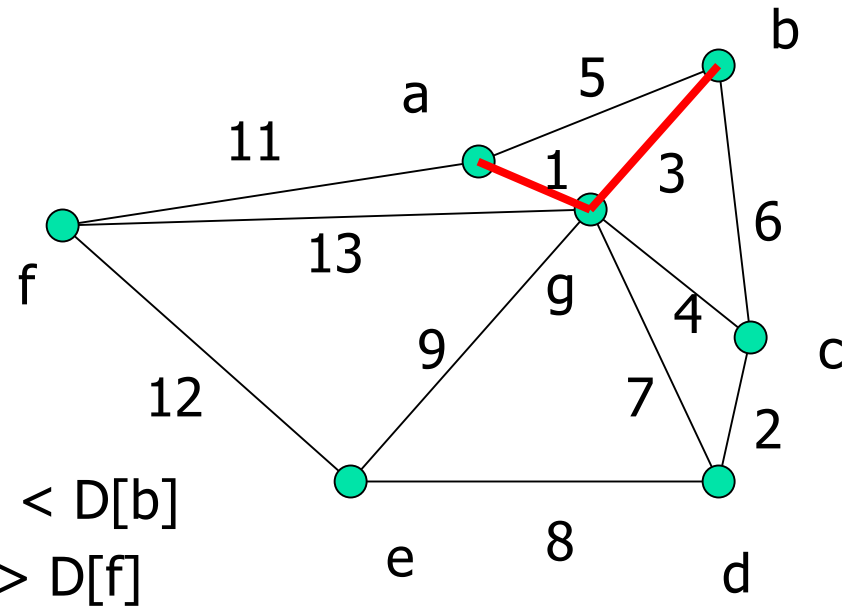
# The Algorithm in Pseudocode

```
Dijkstra(G, s)  // source vertex s
  for each vertex v {
    D[v] := d(s, v) // provisional dist.
    Pred[v] := s if v is neighbor of s,
      otherwise nil
  }
  S := {s}
  while (S != V) {
    find u in V - S with minimum D[u]
    add u to S
    for each v in V - S {
      if (D[u] + d(u,v) < D[v]) {
        D[v] := D[u] + d(u, v)
        Pred[v] := u
      }
    }
  }
```

# Example Operation of Dijkstra's

- Shortest path from a to all nodes
  - Initial

| v | b | c | d | e | f | g |
|---|---|---|---|---|---|---|
| D[v] | 5 | ∞ | ∞ | ∞ | 11 | 1 |
| P[v] | a | / | / | / | a | a |




  - Shortest is to g
    - Consider b: D[g] + d(gb) < D[b]
    - Consider f: D[g] + d(gf) > D[f]

| v | b | c | d | e | f | g |
|---|---|---|---|---|---|---|
| D[v] | 4 | 5 | 8 | 10 | 11 | 1 |
| P[v] | g | g | g | g | a | a |

# Dijkstra's Example (cont'd)

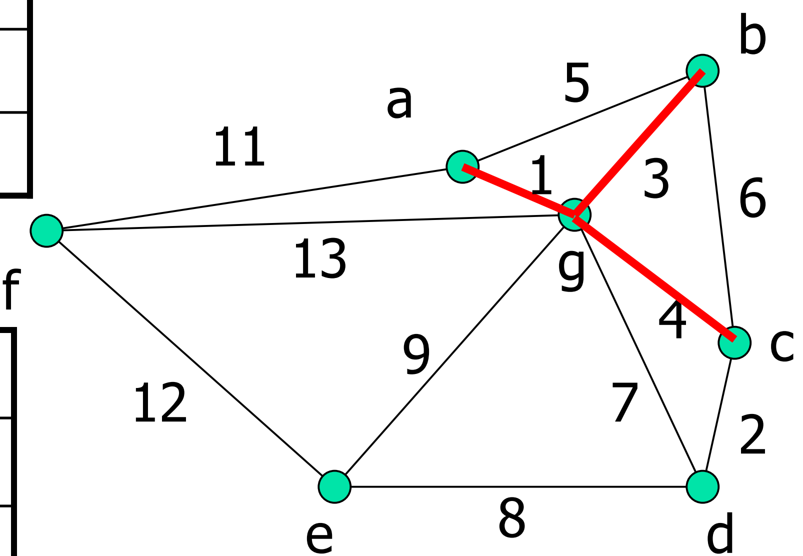
- Shortest is to b

| v | b | c | d | e | f | g |
|---|---|---|---|---|---|---|
| D[v] | 4 | 5 | 8 | 10 | 11 | 1 |
| P[v] | g | g | g | g | a | a |

- Shortest is to c

| v | b | c | d | e | f | g |
|---|---|---|---|---|---|---|
| D[v] | 4 | 5 | 7 | 10 | 11 | 1 |
| P[v] | g | g | c | g | a | a |

- … and so on…



# Getting the Actual Shortest Paths

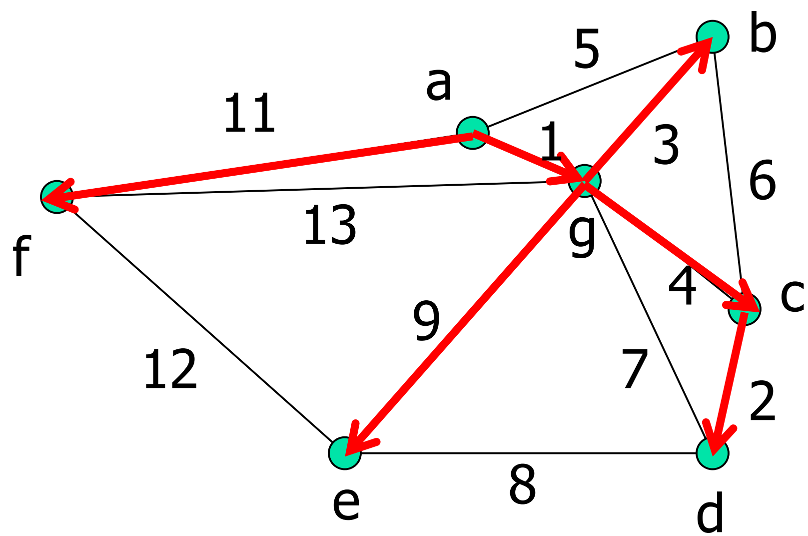
- At the end,

| v | b | c | d | e | f | g |
|---|---|---|---|---|---|---|
| D[v] | 4 | 5 | 7 | 10 | 11 | 1 |
| P[v] | g | g | c | g | a | a |

← Shortest distance

← predecessor

- Can retrieve shortest path from this Pred[] array
- Example:




Shortest path tree

# Running Time of Dijkstra's

- Note the similarity with Prim's algorithm for MST
  - Only difference is the update formula for D[v]
- Therefore, running time identical:
  - $O(n^2)$ for array
  - $O(m \log n)$ for heap
- Note that Dijkstra's algorithm only works for the case where *all edge weights are positive*