CS171 Introduction to Computer Science II

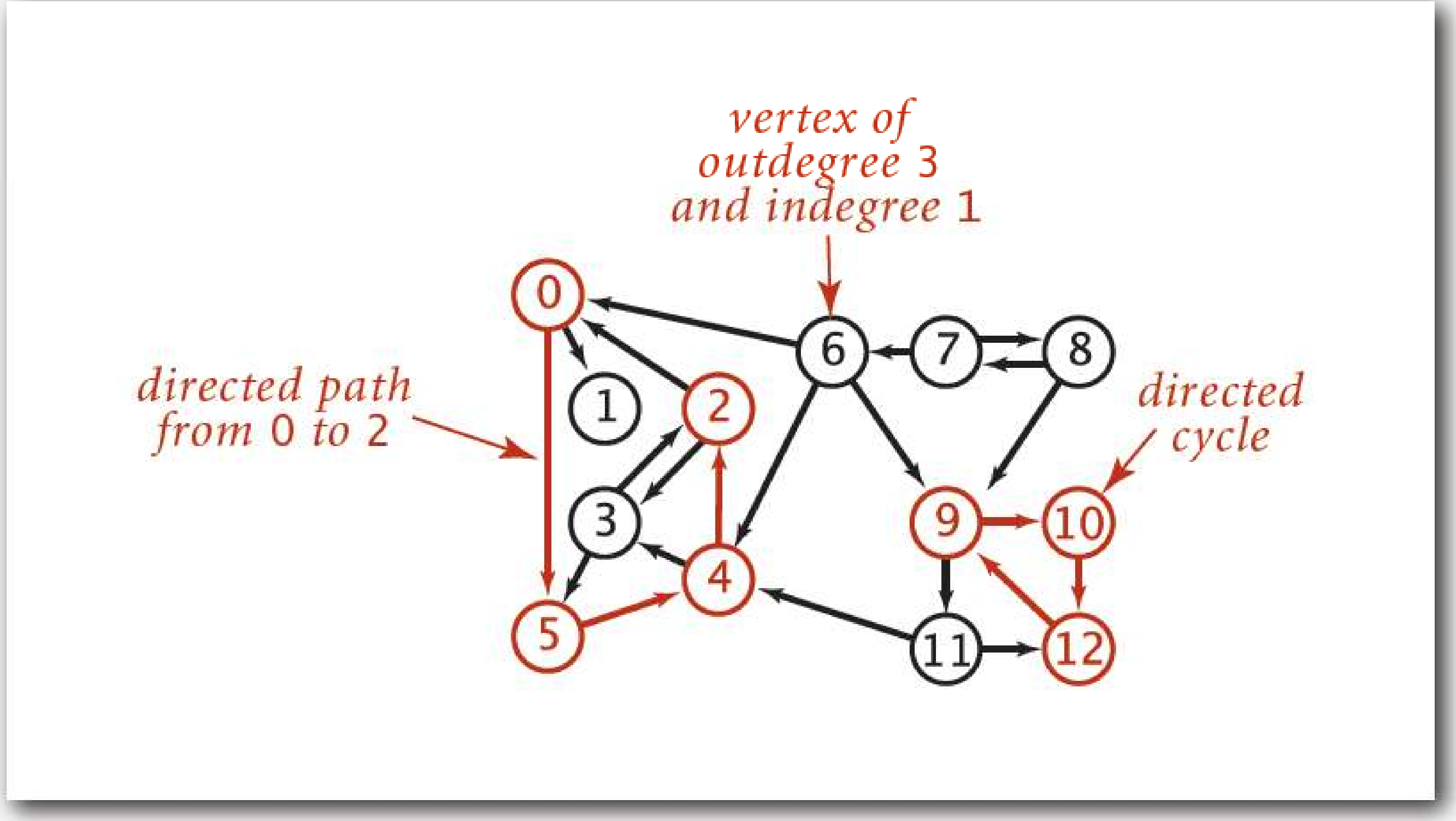
Graphs

Graphs

- Simple graphs
- Algorithms
  - Depth-first search
  - Breadth-first search
  - shortest path
  - Connected components
- Directed graphs
- Weighted graphs
- Minimum spanning tree
- Shortest path

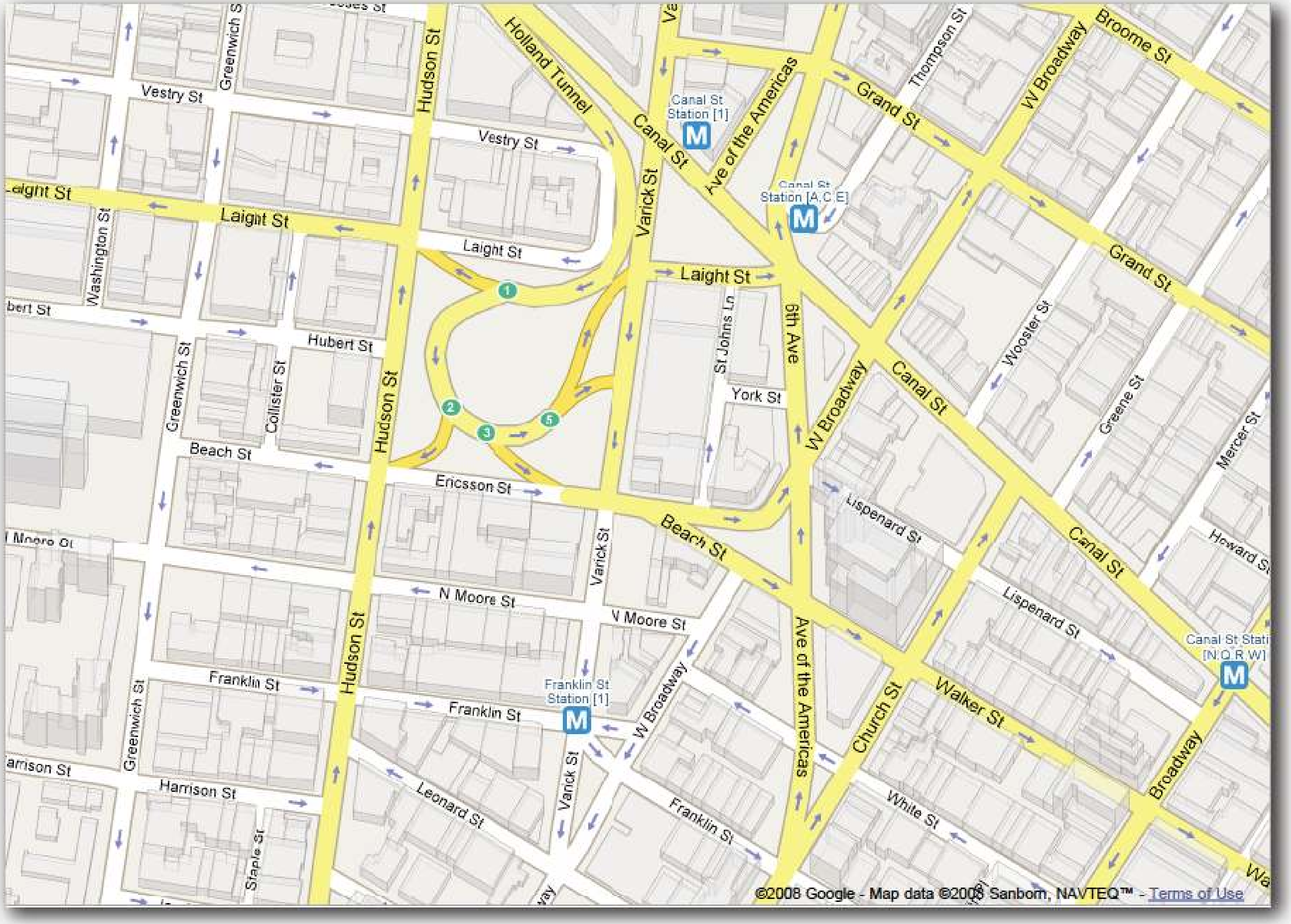
Directed graphs

Digraph. Set of vertices connected pairwise by directed edges.



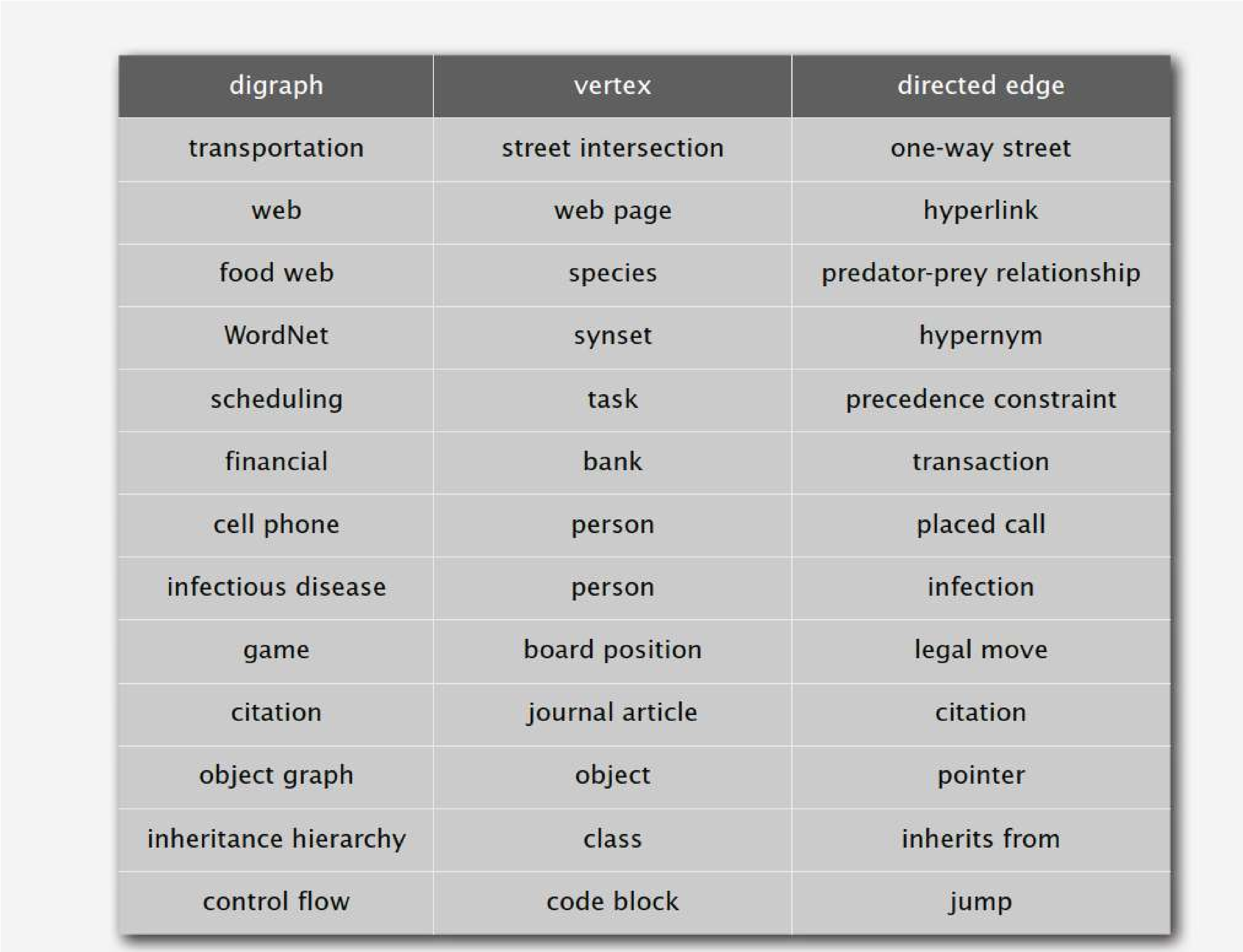
Road network

Vertex = intersection; edge = one-way street.



©2008 Google - Map data ©2008 Sanborn, NAVTEQ™ - Terms of Use

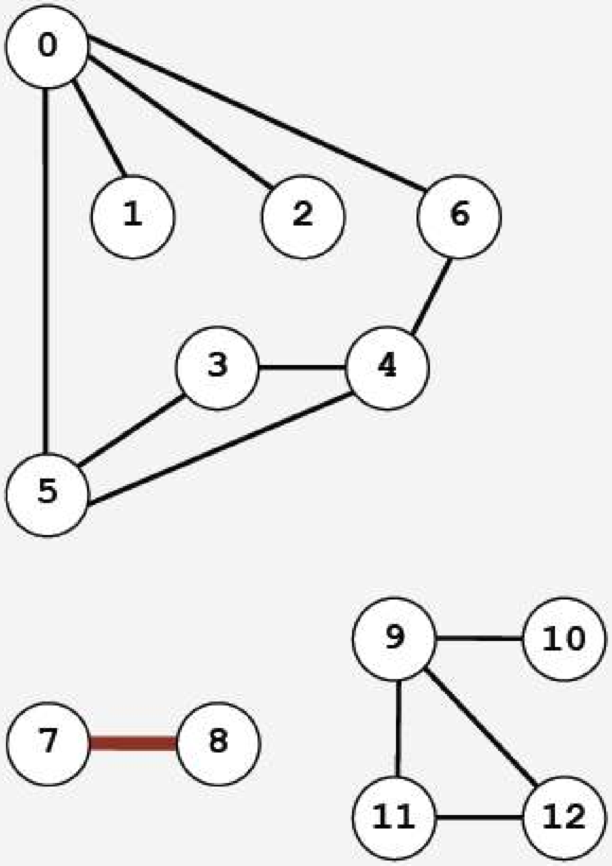
Digraph applications

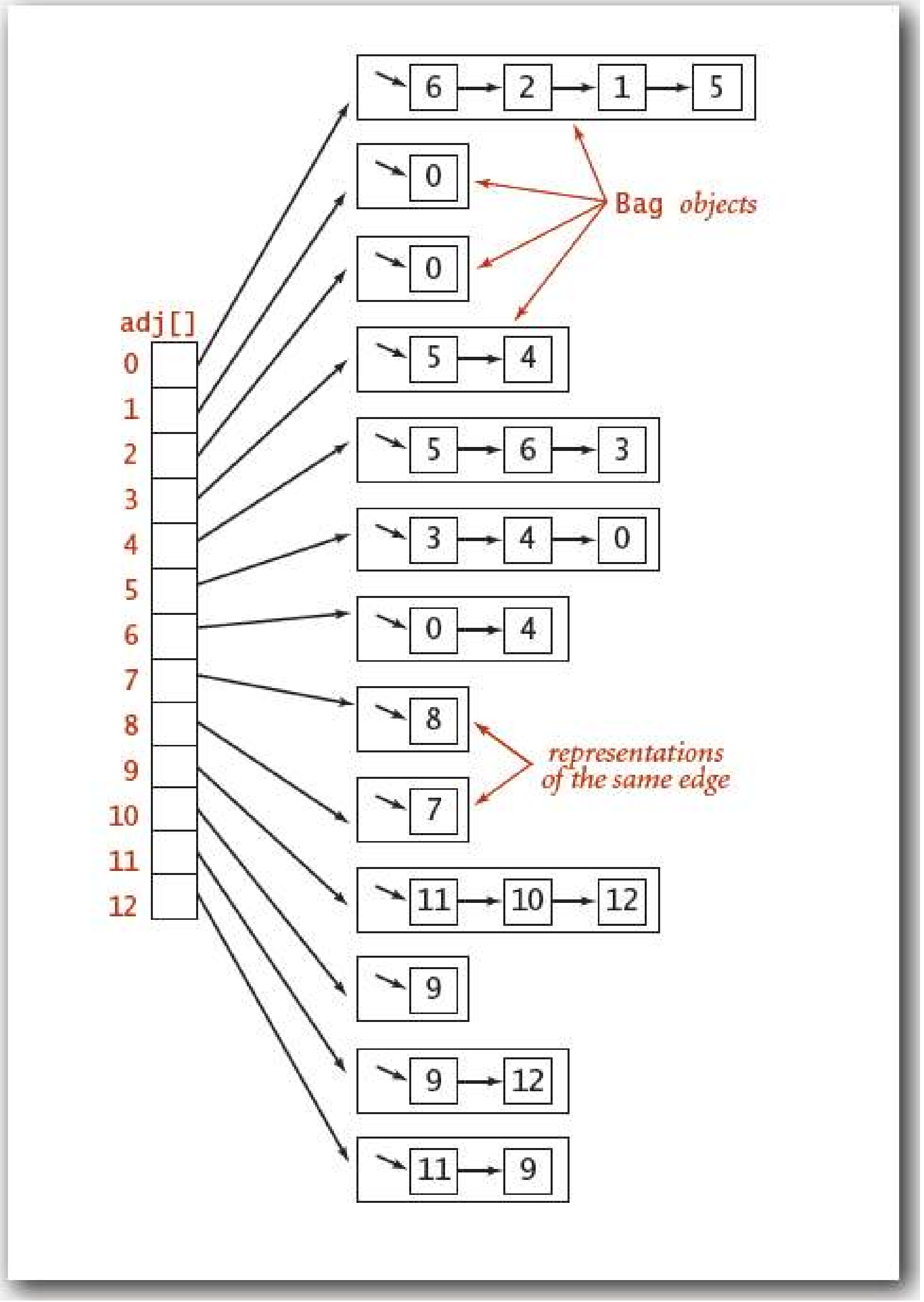


digraph	vertex	directed edge
transportation	street intersection	one-way street
web	web page	hyperlink
food web	species	predator-prey relationship
WordNet	synset	hypernym
scheduling	task	precedence constraint
financial	bank	transaction
cell phone	person	placed call
infectious disease	person	infection
game	board position	legal move
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

Adjacency-list graph representation

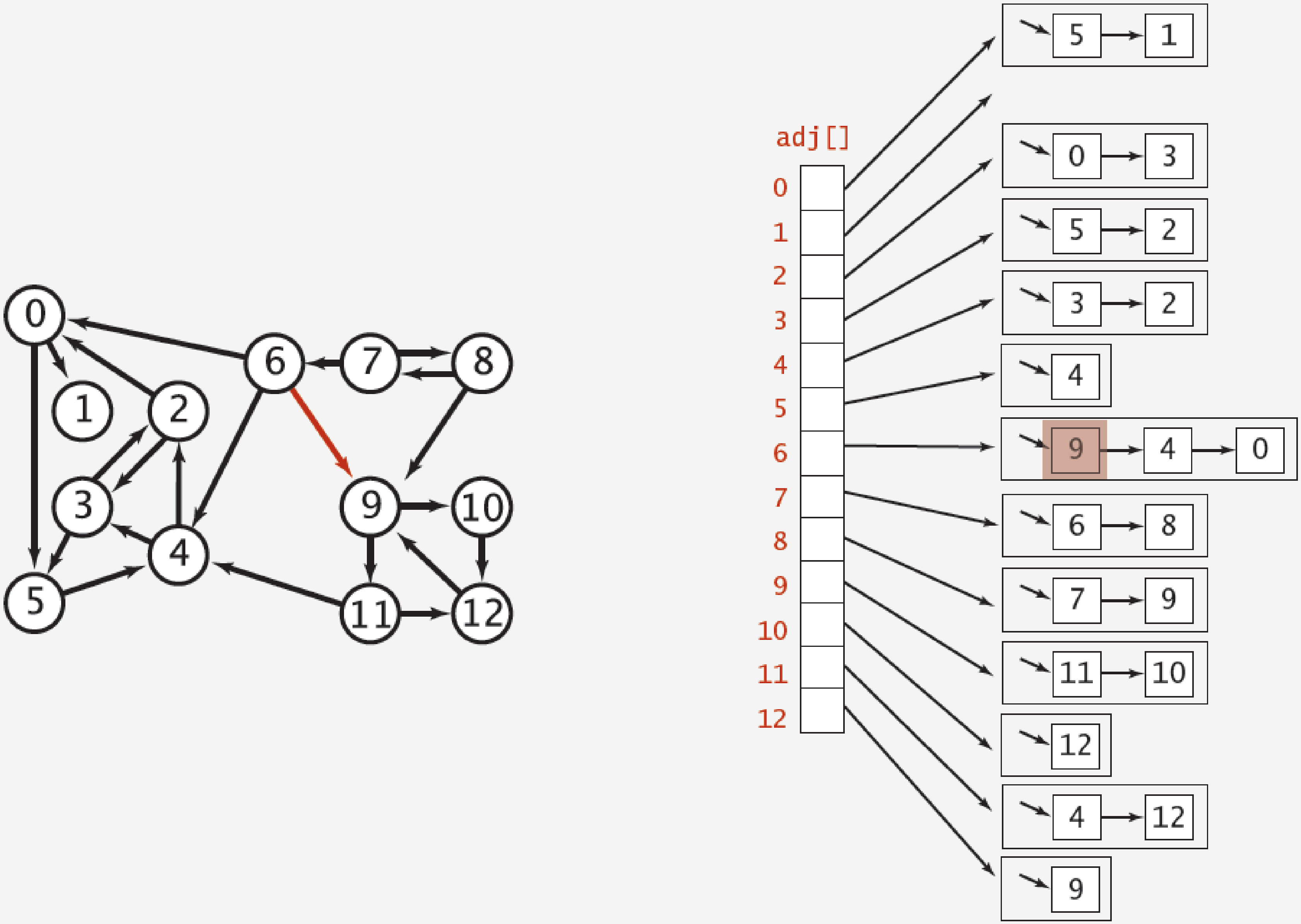
Maintain vertex-indexed array of lists.





Adjacency-lists digraph representation

Maintain vertex-indexed array of lists (use Bag abstraction).



Digraph API

	public class Digraph	
	Digraph(int V)	*create an empty digraph with V vertices*
	Digraph(In in)	*create a digraph from input stream*
void	addEdge(int v, int w)	*add a directed edge v→w*
Iterable<Integer>	adj(int v)	*vertices pointing from v*
int	V()	*number of vertices*
int	E()	*number of edges*
Digraph	reverse()	*reverse of this digraph*
String	toString()	*string representation*

Adjacency-lists digraph representation: Java implementation

```
public class Digraph
{
   private final int V;
   private final Bag<Integer>[] adj;

   public Digraph(int V)
   {
      this.V = V;
      adj = (Bag<Integer>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Integer>();
   }

   public void addEdge(int v, int w)
   {
      adj[v].add(w);

   }

   public Iterable<Integer> adj(int v)
   {  return adj[v];   }
}
```

adjacency lists

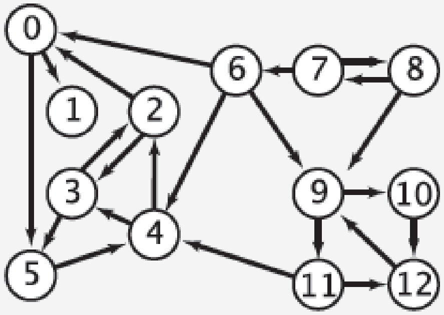
create empty digraph with V vertices

add edge v→w

iterator for vertices pointing from v

Digraph API

```
V → 13
    22 ← E
     4  2
     2  3
     3  2
     6  0
     0  1
     2  0
    11 12
    12  9
     9 10
     9 11
     8  9
    10 12
    11  4
     4  3
     3  5
     7  8
     8  7
     5  4
     0  5
     6  4
     6  9
     7  6
```



```
% java TestDigraph tinyDG.txt
0->5
0->1
2->0
2->3
3->5
3->2
4->3
4->2
5->4
…
11->4
11->12
12-9
```

```
In in = new In(args[0]);
Digraph G = new Digraph(in);

for (int v = 0; v < G.V(); v++)
   for (int w : G.adj(v))
      StdOut.println(v + "->" + w);
```

read digraph from input stream

print out each edge (once)

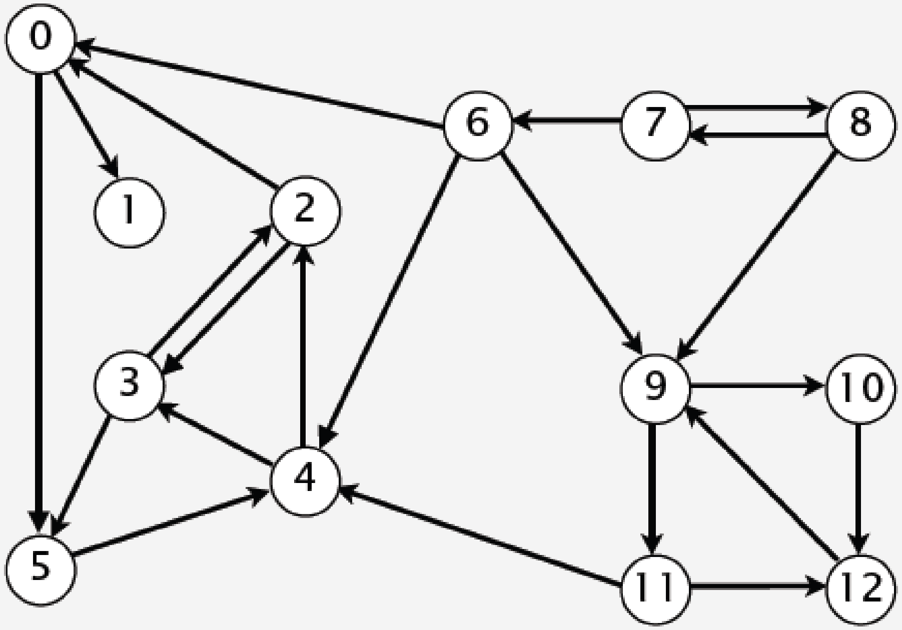
Depth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- DFS is a digraph algorithm.

```
DFS (to visit a vertex v)

Mark v as visited.
Recursively visit all unmarked
        vertices w pointing from v.
```

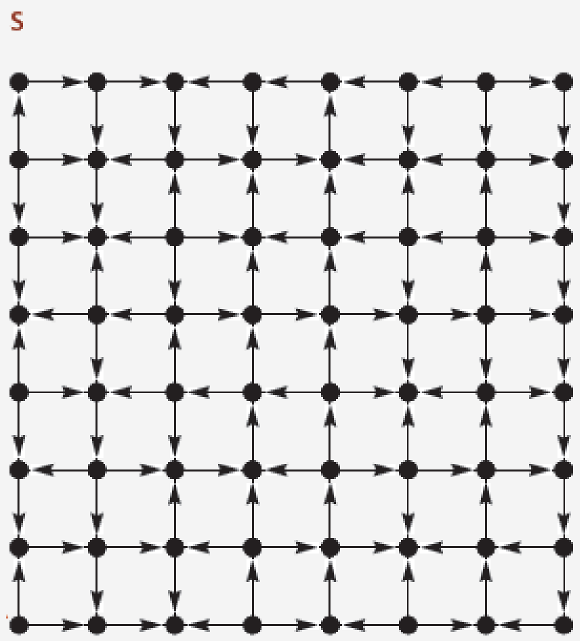


Breadth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- BFS is a digraph algorithm.

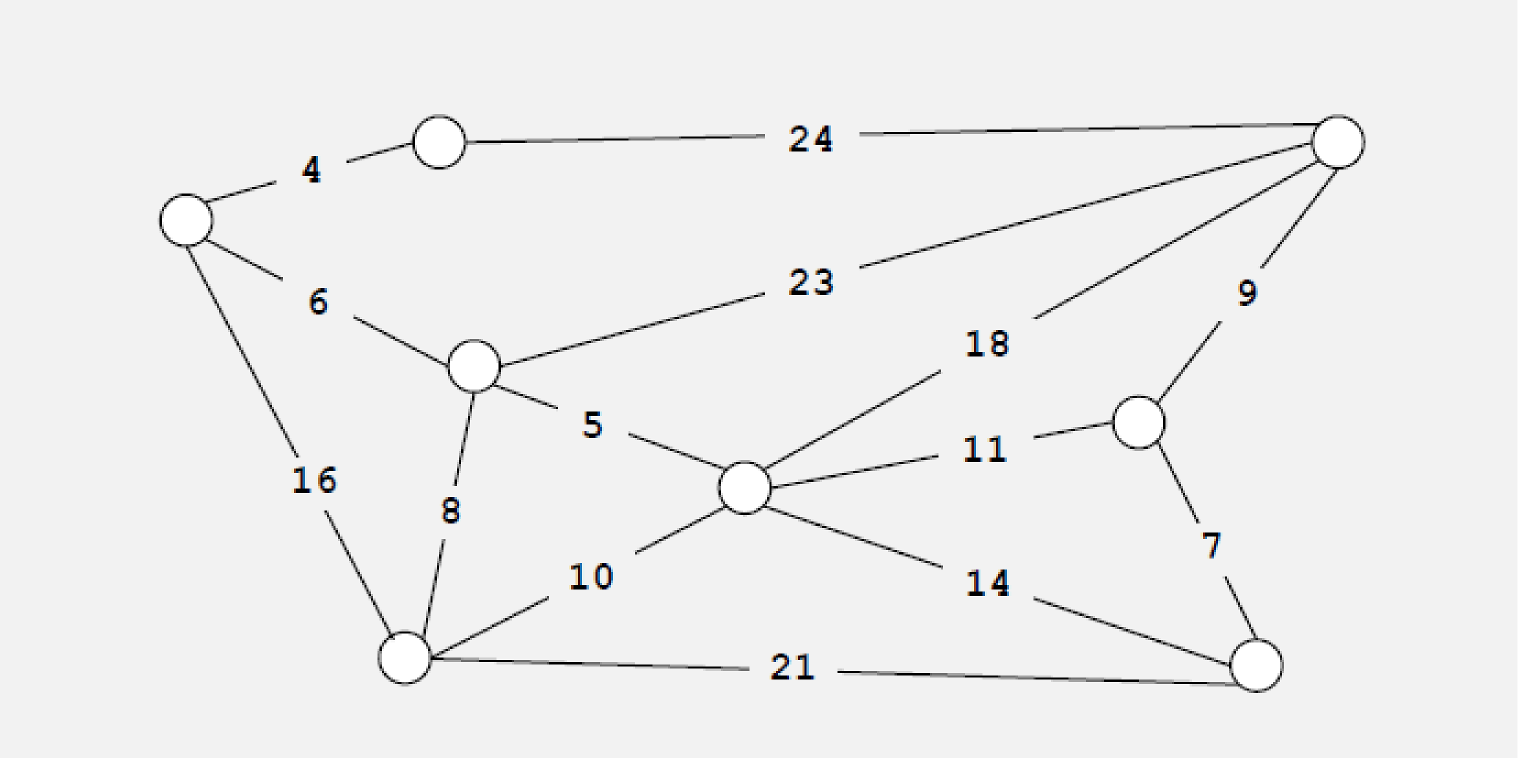
```
BFS (from source vertex s)
Put s onto a FIFO queue, and mark s as visited.
Repeat until the queue is empty:
  - remove the least recently added vertex v
  - for each unmarked vertex pointing from v:
    add to queue and mark as visited.
```



Proposition. BFS computes shortest paths (fewest number of edges).

Edge-weighted graphs

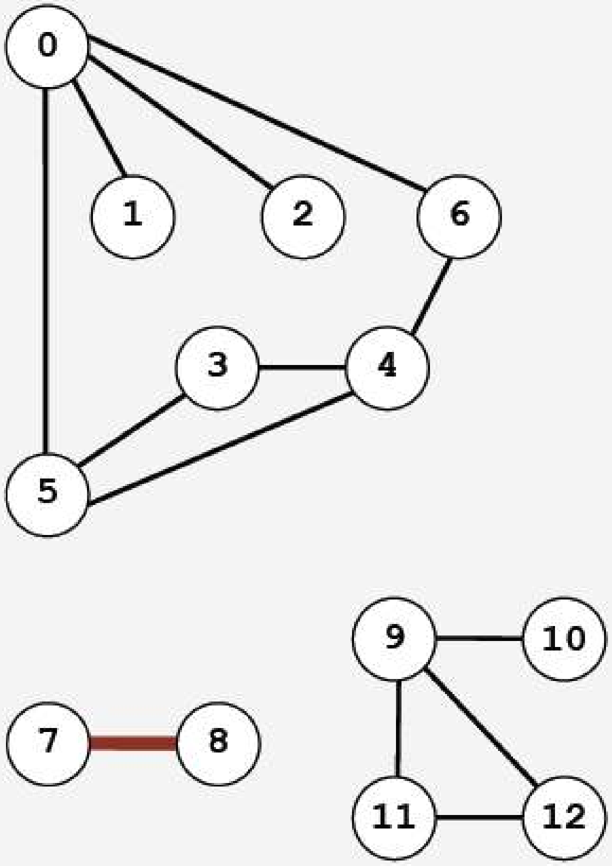
- Each connection has an associated weight

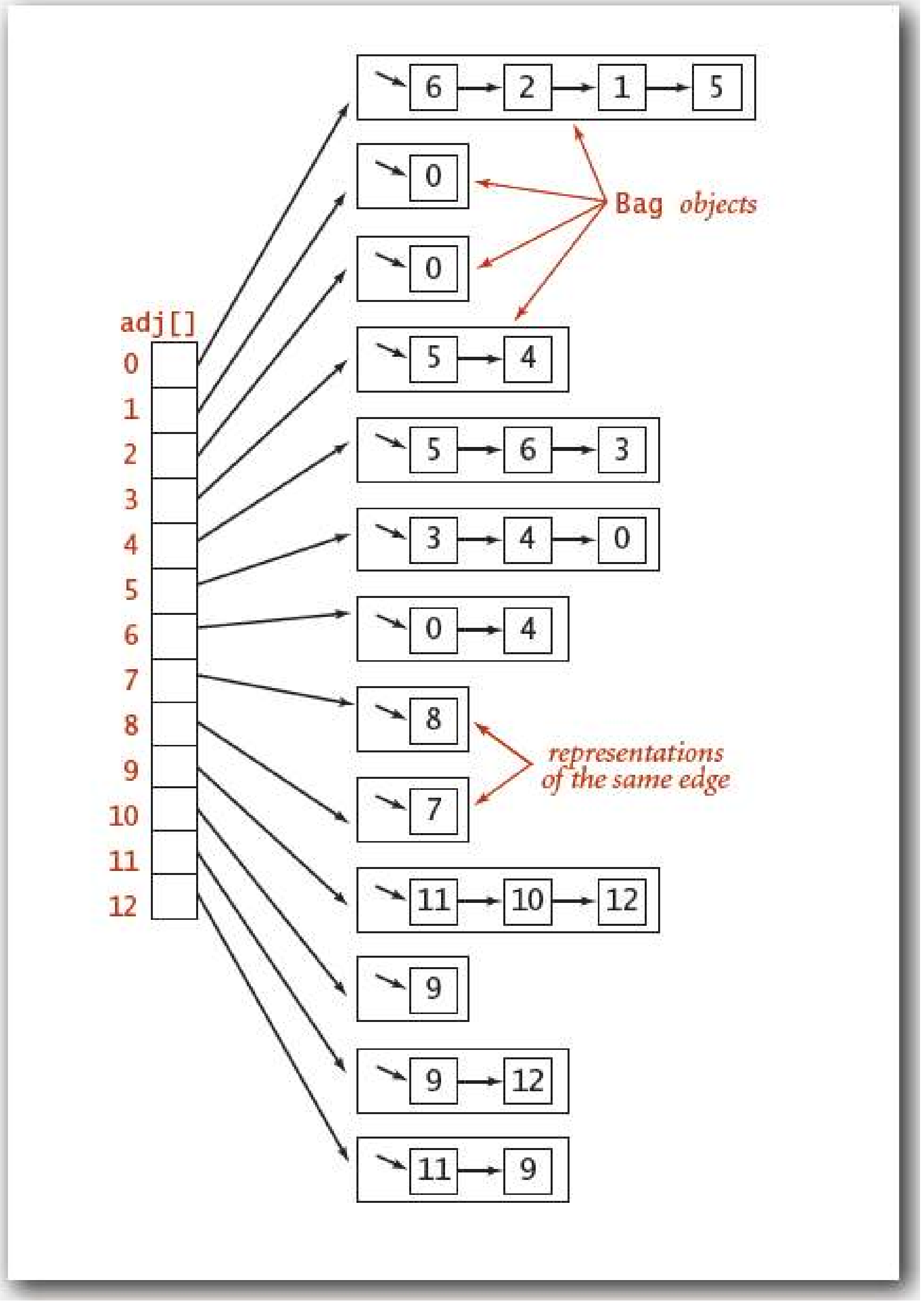


graph G

Adjacency-list graph representation

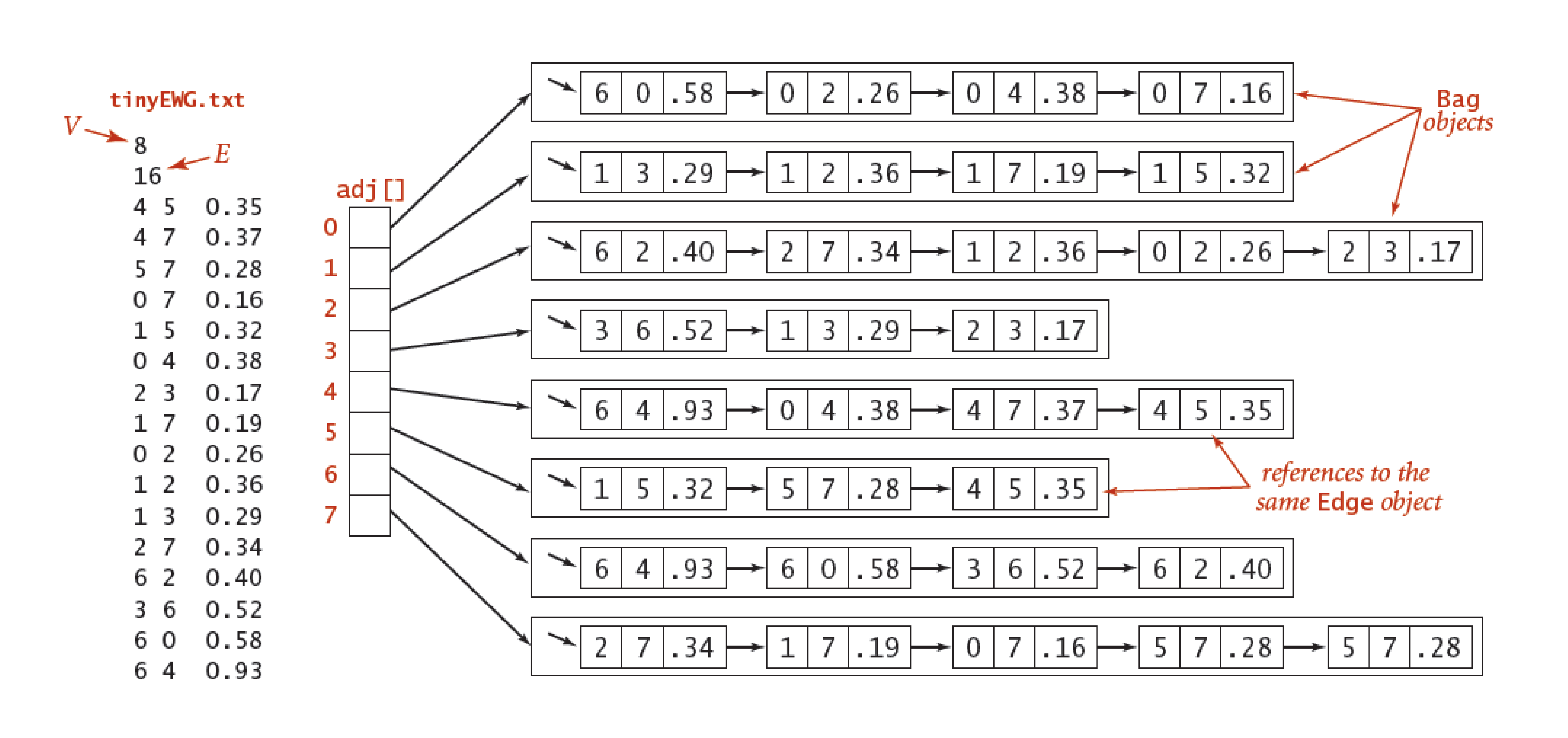
Maintain vertex-indexed array of lists.





Edge-weighted graph: adjacency-lists representation

Maintain vertex-indexed array of `Edge` lists (use `Bag` abstraction).



tinyEWG.txt

```
8
16
4 5  0.35
4 7  0.37
5 7  0.28
0 7  0.16
1 5  0.32
0 4  0.38
2 3  0.17
1 7  0.19
0 2  0.26
1 2  0.36
1 3  0.29
2 7  0.34
6 2  0.40
3 6  0.52
6 0  0.58
6 4  0.93
```

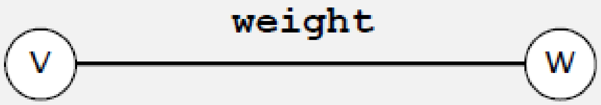
```
0  6 0 .58 → 0 2 .26 → 0 4 .38 → 0 7 .16
1  1 3 .29 → 1 2 .36 → 1 7 .19 → 1 5 .32
2  6 2 .40 → 2 7 .34 → 1 2 .36 → 0 2 .26 → 2 3 .17
3  3 6 .52 → 1 3 .29 → 2 3 .17
4  6 4 .93 → 0 4 .38 → 4 7 .37 → 4 5 .35
5  1 5 .32 → 5 7 .28 → 4 5 .35
6  6 4 .93 → 6 0 .58 → 3 6 .52 → 6 2 .40
7  2 7 .34 → 1 7 .19 → 0 7 .16 → 5 7 .28 → 5 7 .28
```

Weighted edge API

Edge abstraction needed for weighted edges.

`public class Edge implements Comparable<Edge>`

	`Edge(int v, int w, double weight)`	*create a weighted edge v-w*
`int`	`either()`	*either endpoint*
`int`	`other(int v)`	*the endpoint that's not v*
`int`	`compareTo(Edge that)`	*compare this edge to that edge*
`double`	`weight()`	*the weight*
`String`	`toString()`	*string representation*



Idiom for processing an edge `e`: `int v = e.either(), w = e.other(v);`

Weighted edge: Java implementation

```
public class Edge implements Comparable<Edge>
{
   private final int v, w;
   private final double weight;

   public Edge(int v, int w, double weight)
   {
      this.v = v;
      this.w = w;
      this.weight = weight;
   }

   public int either()
   {  return v;  }

   public int other(int vertex)
   {
      if (vertex == v) return w;
      else return v;
   }

   public int compareTo(Edge that)
   {
      if      (this.weight < that.weight) return -1;
      else if (this.weight > that.weight) return +1;
      else                                return  0;
   }
}
```

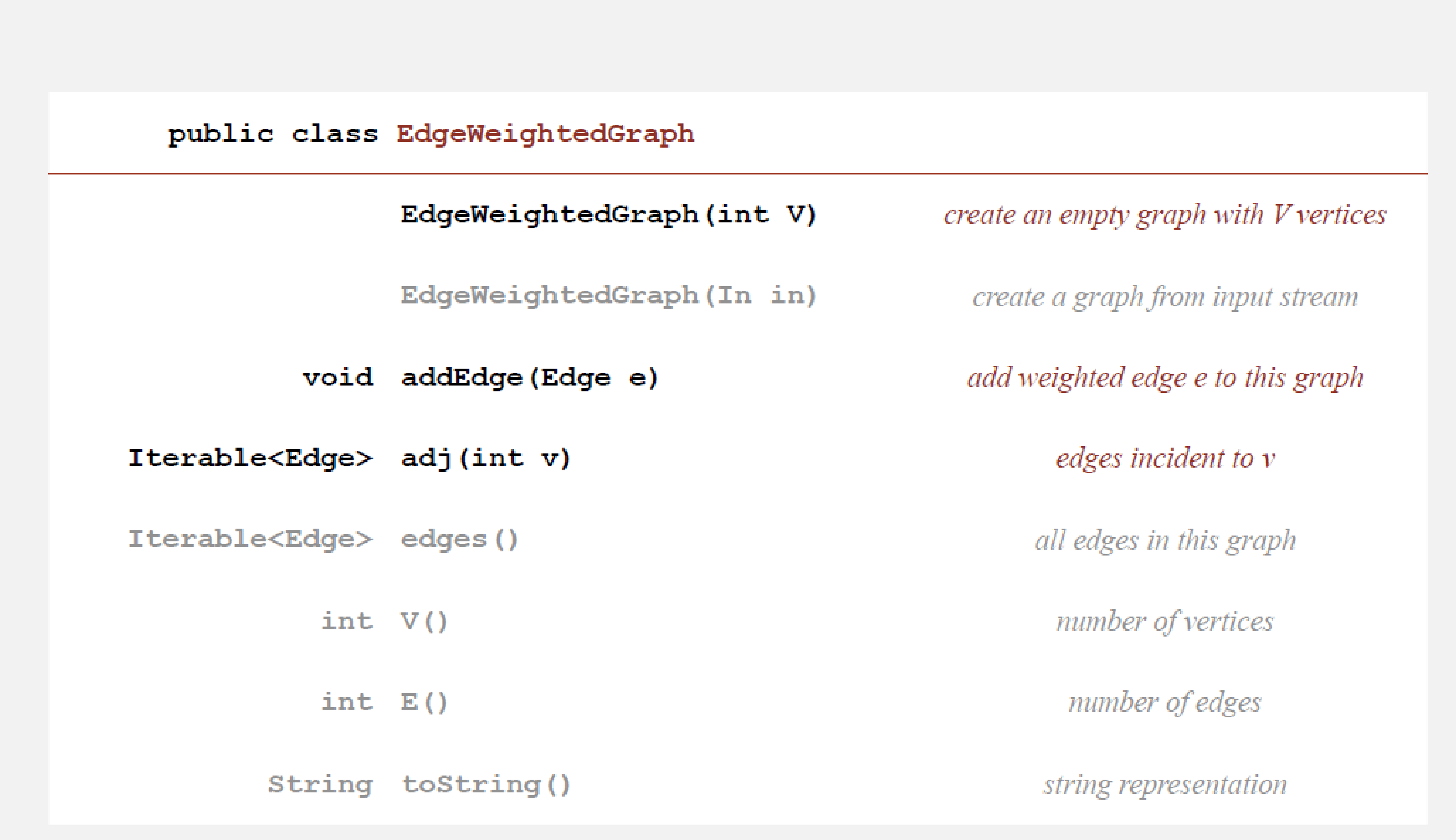
constructor

either endpoint

other endpoint

compare edges by weight

Edge-weighted graph API



public class	EdgeWeightedGraph	
	EdgeWeightedGraph(int V)	*create an empty graph with V vertices*
	EdgeWeightedGraph(In in)	*create a graph from input stream*
void	addEdge(Edge e)	*add weighted edge e to this graph*
Iterable<Edge>	adj(int v)	*edges incident to v*
Iterable<Edge>	edges()	*all edges in this graph*
int	V()	*number of vertices*
int	E()	*number of edges*
String	toString()	*string representation*

Edge-weighted graph: adjacency-lists implementation

```
public class EdgeWeightedGraph
{
   private final int V;
   private final Bag<Edge>[] adj;

   public EdgeWeightedGraph(int V)
   {
      this.V = V;
      adj = (Bag<Edge>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Edge>();
   }

   public void addEdge(Edge e)
   {
      int v = e.either(), w = e.other(v);
      adj[v].add(e);
      adj[w].add(e);
   }

   public Iterable<Edge> adj(int v)
   {  return adj[v];  }
}
```

same as **`Graph`**, but adjacency lists of **`Edge`**s instead of integers

constructor

add edge to both adjacency lists

Graphs

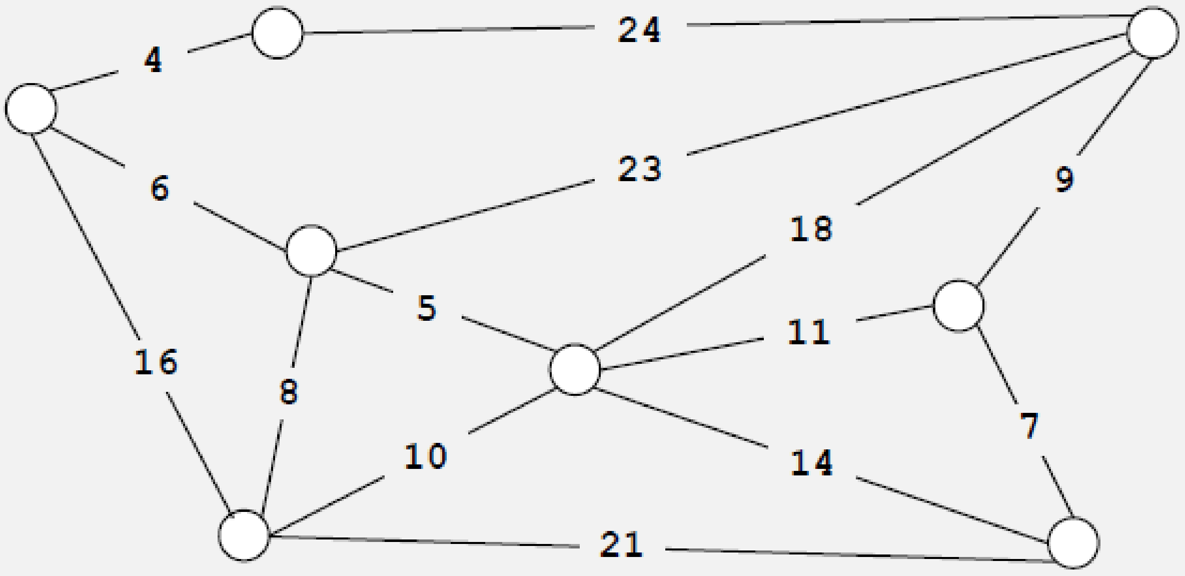
- Simple graphs
- Algorithms
  - Depth-first search
  - Breadth-first search
  - shortest path
  - Connected components
- Directed graphs
- Weighted graphs
- Minimum spanning tree
- Shortest path

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A spanning tree of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.



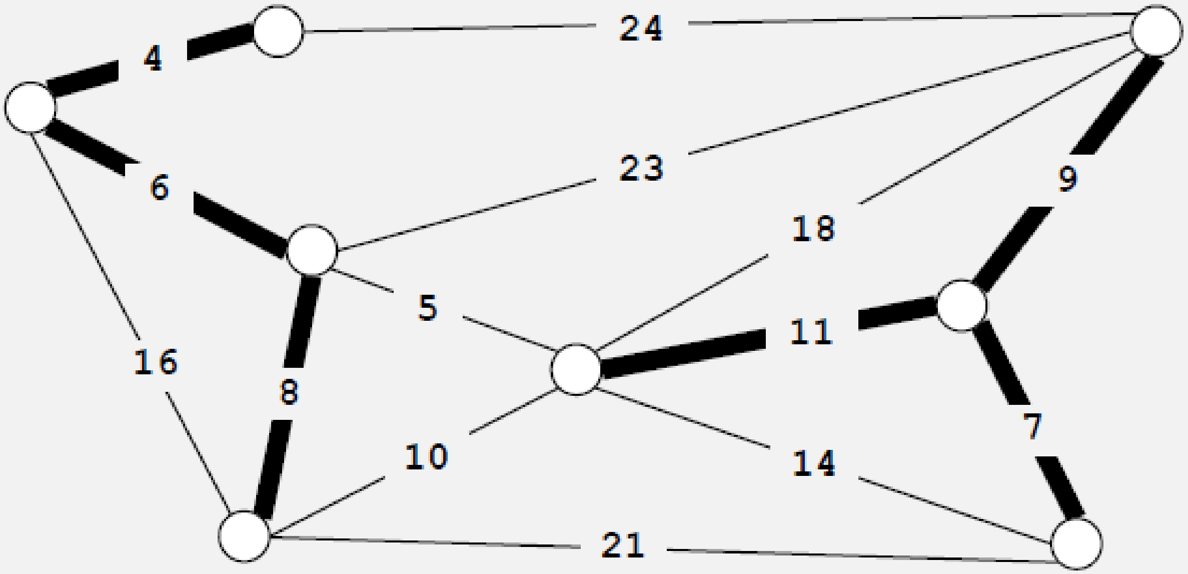
graph G

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A spanning tree of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.

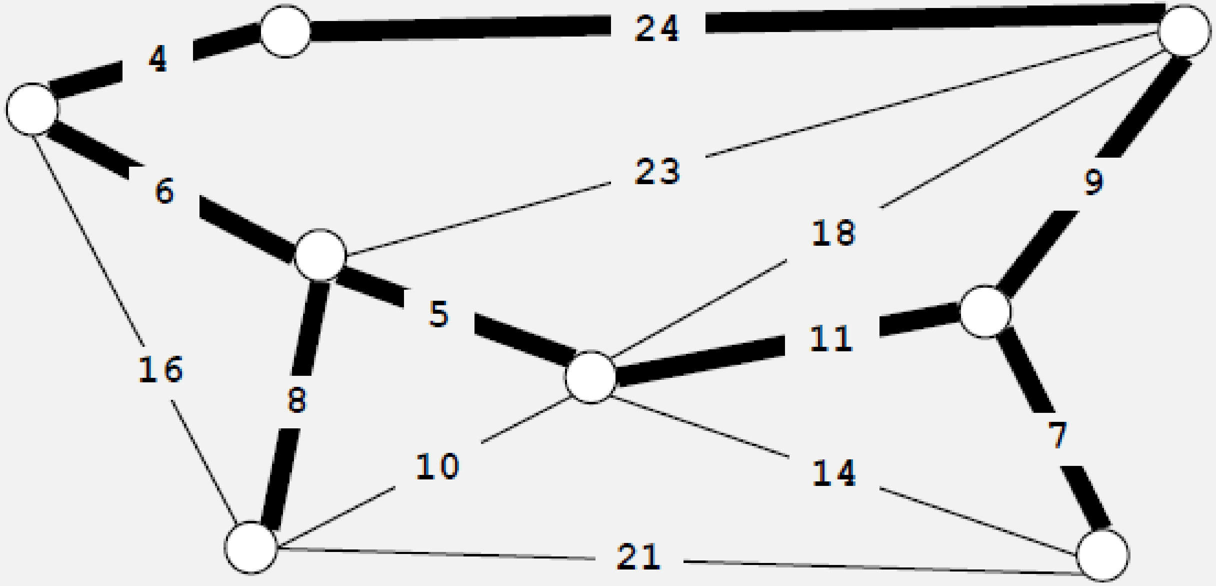


Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A spanning tree of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.

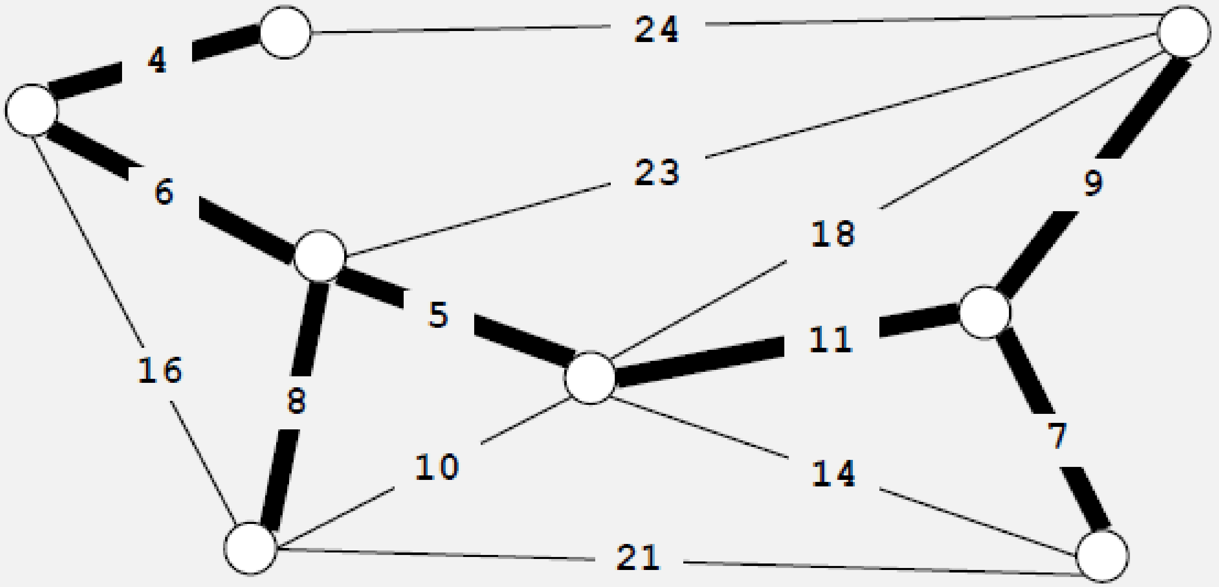


Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A spanning tree of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.



spanning tree T: cost = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7

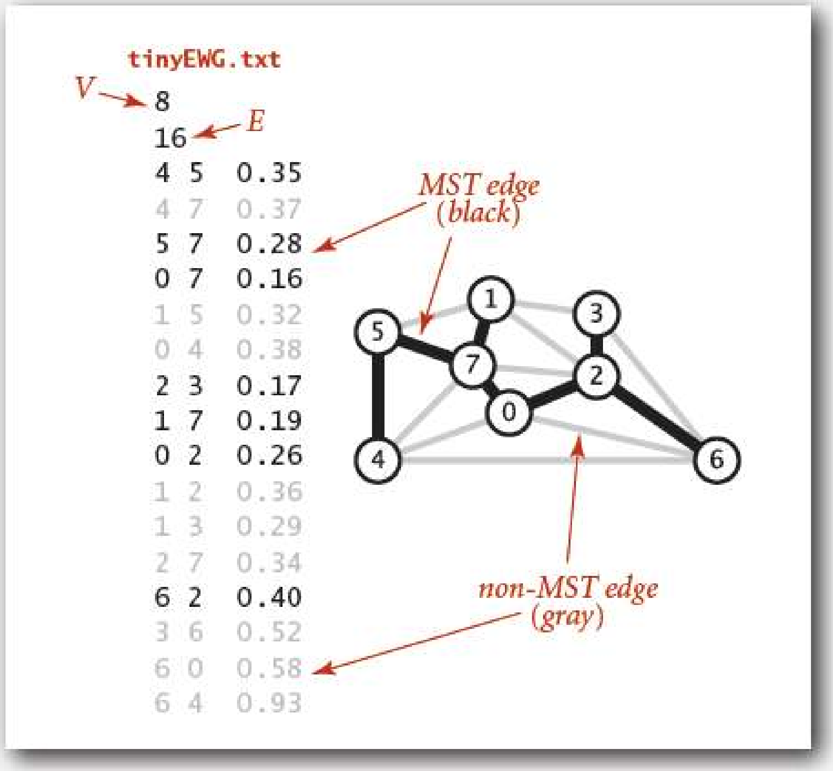
Applications

- Phone/cable network design – minimum cost
- Approximation algorithms for NP-hard problems

Minimum spanning tree API

Q. How to represent the MST?

public class MST		
	MST(EdgeWeightedGraph G)	*constructor*
Iterable<Edge>	edges()	*edges in MST*
double	weight()	*weight of MST*



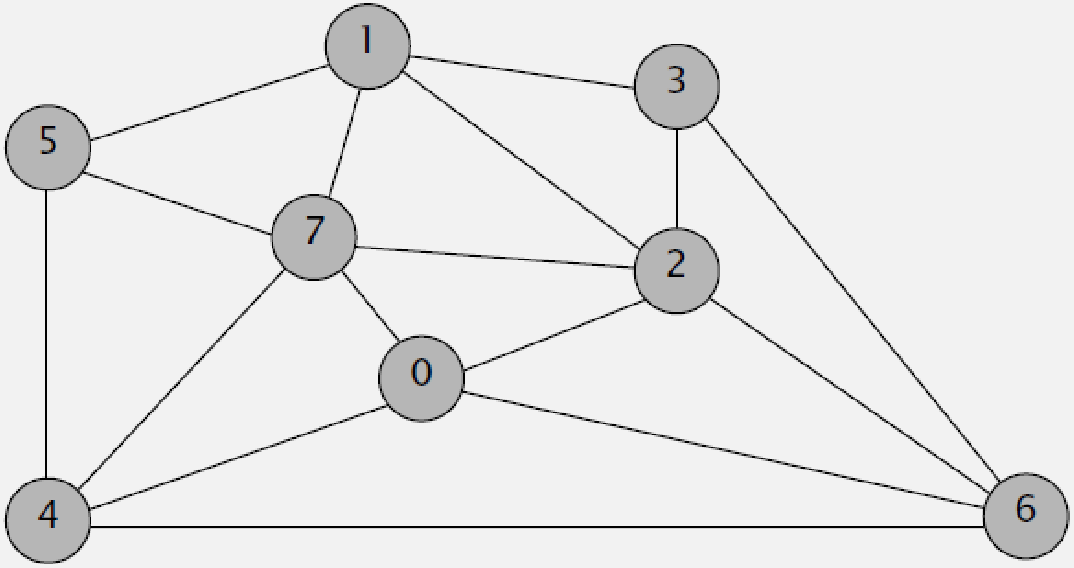
tinyEWG.txt

```
8
16
4 5  0.35
4 7  0.37
5 7  0.28
0 7  0.16
1 5  0.32
0 4  0.38
2 3  0.17
1 7  0.19
0 2  0.26
1 2  0.36
1 3  0.29
2 7  0.34
6 2  0.40
3 6  0.52
6 0  0.58
6 4  0.93
```

```
% java MST tinyEWG.txt
0-7 0.16
1-7 0.19
0-2 0.26
2-3 0.17
5-7 0.28
4-5 0.35
6-2 0.40
1.81
```

Prim's algorithm

- Start with vertex 0 and greedily grow tree T.
- At each step, add to T the min weight edge with exactly one endpoint in T.



an edge-weighted graph

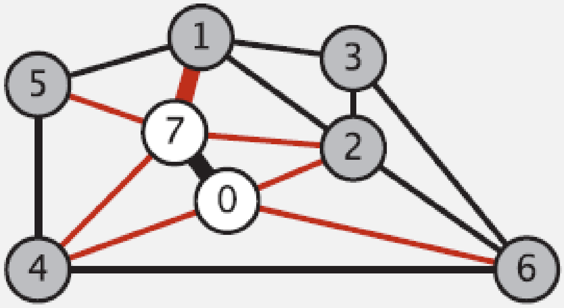
0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

Prim's algorithm: implementation challenge

Challenge. Find the min weight edge with exactly one endpoint in T.

How difficult?

- $O(E)$ time. ← try all edges
- $O(V)$ time.
- $O(\log E)$ time. ← use a priority queue !
- $O(\log^* E)$ time.
- Constant time.



1-7 is min weight edge with exactly one endpoint in T

priority queue of crossing edges

1-7	0.19
0-2	0.26
5-7	0.28
2-7	0.34
4-7	0.37
0-4	0.38
6-0	0.58

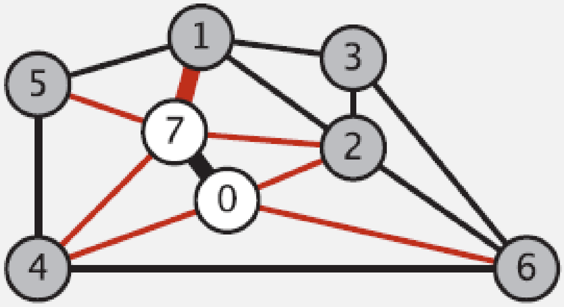
Prim's algorithm: lazy implementation

Challenge. Find the min weight edge with exactly one endpoint in T.

Lazy solution. Maintain a PQ of edges with (at least) one endpoint in T.

- Delete min to determine next edge $e = v–w$ to add to T.
- Disregard if both endpoints v and w are in T.
- Otherwise, let v be vertex not in T:
  - add to PQ any edge incident to v (assuming other endpoint not in T)
  - add v to T

1-7 is min weight edge with exactly one endpoint in T



priority queue of crossing edges

1-7	0.19
0-2	0.26
5-7	0.28
2-7	0.34
4-7	0.37
0-4	0.38
6-0	0.58

Prim's algorithm demo: lazy implementation

Use `MinPQ`: key = edge, prioritized by weight.

(lazy version leaves some obsolete edges on the PQ)

Prim's algorithm: lazy implementation

```
public class LazyPrimMST
{
   private boolean[] marked;    // MST vertices
   private Queue<Edge> mst;     // MST edges
   private MinPQ<Edge> pq;      // PQ of edges

    public LazyPrimMST(WeightedGraph G)
    {
        pq = new MinPQ<Edge>();
        mst = new Queue<Edge>();
        marked = new boolean[G.V()];
        visit(G, 0);

        while (!pq.isEmpty())
        {
           Edge e = pq.delMin();
           int v = e.either(), w = e.other(v);
           if (marked[v] && marked[w]) continue;
           mst.enqueue(e);
           if (!marked[v]) visit(G, v);
           if (!marked[w]) visit(G, w);
        }
   }
}
```

assume G is connected

repeatedly delete the
min weight edge e = v–w from PQ

ignore if both endpoints in T

add edge e to tree

add v or w to tree

Prim's algorithm: lazy implementation

```
private void visit(WeightedGraph G, int v)
{
   marked[v] = true;
   for (Edge e : G.adj(v))
      if (!marked[e.other(v)])
         pq.insert(e);
}

public Iterable<Edge> mst()
{  return mst;  }
```

add v to T

for each edge e = v–w, add to PQ if w not already in T

Lazy Prim's algorithm: running time

Proposition. Lazy Prim's algorithm computes the MST in time proportional to $E \log E$ and extra space proportional to E (in the worst case).

Pf.

operation	frequency	binary heap
delete min	E	log E
insert	E	log E

Graphs

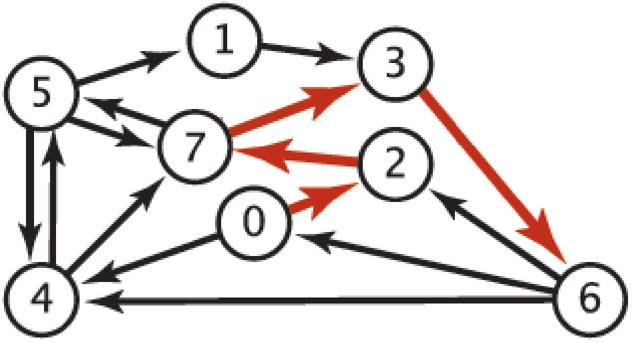
- Simple graphs
- Algorithms
  - Depth-first search
  - Breadth-first search
  - shortest path
  - Connected components
- Directed graphs
- Weighted graphs
- Minimum spanning tree
- Shortest path

Shortest paths in a weighted digraph

Given an edge-weighted digraph, find the shortest (directed) path from s to t.

edge-weighted digraph

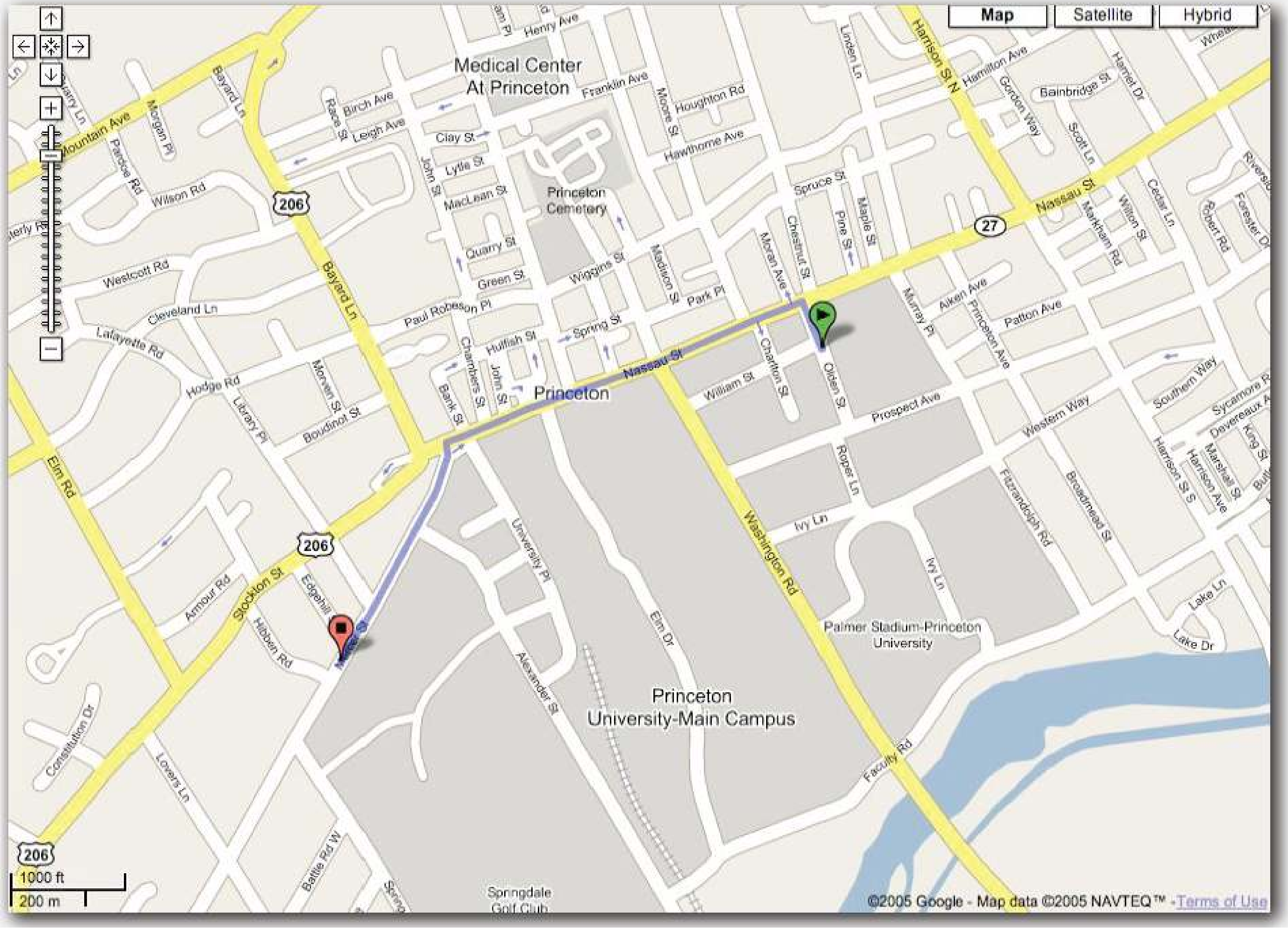
4->5	0.35
5->4	0.35
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93



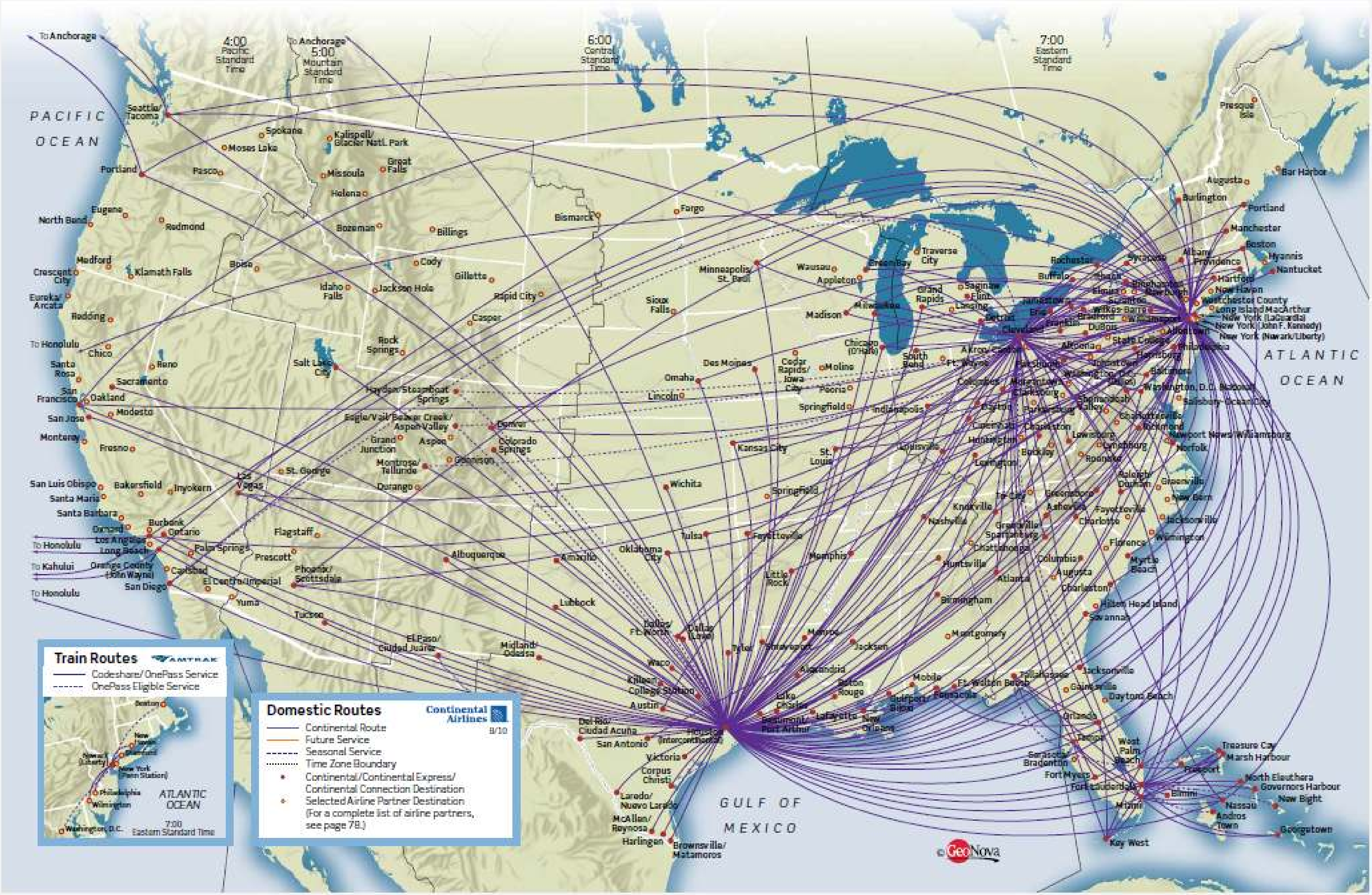
shortest path from 0 to 6

0->2	0.26
2->7	0.34
7->3	0.39
3->6	0.52

Google maps


Map
Satellite
Hybrid
Medical Center
At Princeton
Henry Ave
Franklin Ave
Birch Ave
Leigh Ave
Race St
Clay St
Lytle St
MacLean St
John St
Princeton
Cemetery
Quarry St
Green St
Wiggins St
Moore St
Houghton Rd
Hawthorne Ave
Linden Ln
Harrison St N
Hamilton Ave
Gordon Way
Bainbridge St
Harriet Dr
Scott Ln
Spruce St
Chestnut St
Pine St
Maple St
Moran Ave
Madison St
Park Pl
Nassau St
27
Wilson St
Cedar Ln
Markham Rd
Robert Rd
Forester Dr
Mountain Ave
Morgan Pl
Bayard Ln
Parkside Rd
Wilson Rd
206
Westcott Rd
Cleveland Ln
Lafayette Rd
Hodge Rd
Library Pl
Morven St
Boudinot St
Paul Robeson Pl
Chambers St
Hulfish St
Spring St
Bank St
Princeton
William St
Charlton St
Olden St
Prospect Ave
Murray Pl
Aiken Ave
Princeton Ave
Patton Ave
Southern Way
Western Way
Sycamore Rd
Devereaux Ave
Harrison St S
Harrison Ave
Marshall St
King St
Fitzrandolph Rd
Broadmead St
Roper Ln
Ivy Ln
Washington Rd
University Pl
Elm Dr
Alexander St
Palmer Stadium-Princeton
University
Lake Ln
Lake Dr
Princeton
University-Main Campus
Faculty Rd
Elm Rd
Armour Rd
Stockton St
Edgehill St
Hibben Rd
Constitution Dr
Lovers Ln
Battle Rd W
Springdale
Golf Club
1000 ft
200 m
©2005 Google - Map data ©2005 NAVTEQ™ - Terms of Use

Continental U.S. routes (August 2010)



http://www.continental.com/web/en-US/content/travel/routes

Dijkstra’s Algorithm

- Finds all shortest paths given a source
- Solves single-source, single-destination, single-pair shortest path problem
- Intuition: grows the paths from the source node using a greedy approach

Shortest Paths – Dijkstra's Algorithm

- Assign to every node a distance value: set it to zero for source node and to infinity for all other nodes.
- Mark all nodes as unvisited. Set source node as current.

- For current node, consider all its unvisited neighbors and calculate their ***tentative* distance**. If this distance is **less than the previously** recorded distance, **overwrite** the distance (edge relaxation). Mark it as visited.
- Set the unvisited node with **the smallest distance** from the **source node** as the next "current node" and repeat the above

- Done when all nodes are visited.

Data structures

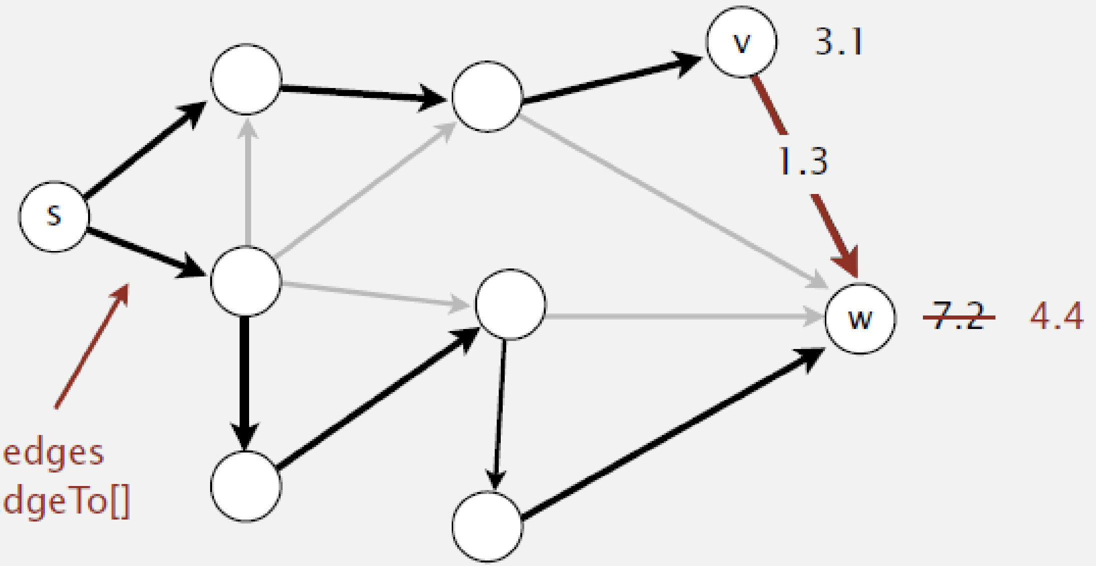
- Distance to the source: a vertex-indexed array distTo[] such that distTo[v] is the length of the shortest known path from s to v
- Edges on the shortest paths tree: a parent-edge representation of a vertex-indexed array edgeTo[] where edgeTo[v] is the parent edge on the shortest path to v

Edge relaxation

Relax edge $e = v \to w$.

- `distTo[v]` is length of shortest known path from s to v.
- `distTo[w]` is length of shortest known path from s to w.
- `edgeTo[w]` is last edge on shortest known path from s to w.
- If $e = v \to w$ gives shorter path to w through v, update `distTo[w]` and `edgeTo[w]`.

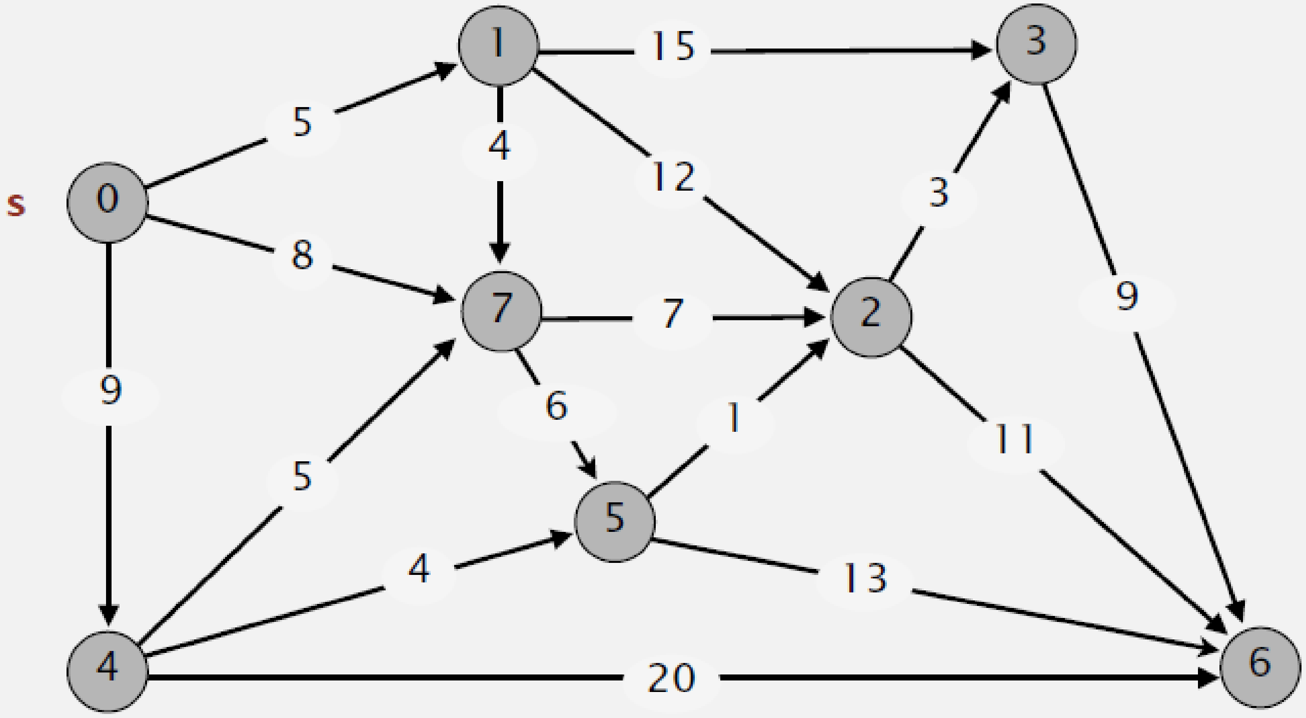
v→w successfully relaxes



black edges are in edgeTo[]

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges pointing from that vertex.



an edge-weighted digraph

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

Dijkstra's algorithm: Java implementation

```
public class DijkstraSP
{
   private DirectedEdge[] edgeTo;
   private double[] distTo;
   private IndexMinPQ<Double> pq;

   public DijkstraSP(EdgeWeightedDigraph G, int s)
   {
      edgeTo = new DirectedEdge[G.V()];
      distTo = new double[G.V()];
      pq = new IndexMinPQ<Double>(G.V());

      for (int v = 0; v < G.V(); v++)
         distTo[v] = Double.POSITIVE_INFINITY;
      distTo[s] = 0.0;

      pq.insert(s, 0.0);
      while (!pq.isEmpty())
      {
          int v = pq.delMin();
          for (DirectedEdge e : G.adj(v))
             relax(e);
      }
    }
 }
```

relax vertices in order of distance from s

Dijkstra's algorithm: Java implementation

```
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
       distTo[w] = distTo[v] + e.weight();
       edgeTo[w] = e;
       if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
       else                pq.insert     (w, distTo[w]);
   }
}
```

update PQ

Priority-first search

Insight. Four of our graph-search methods are the same algorithm!

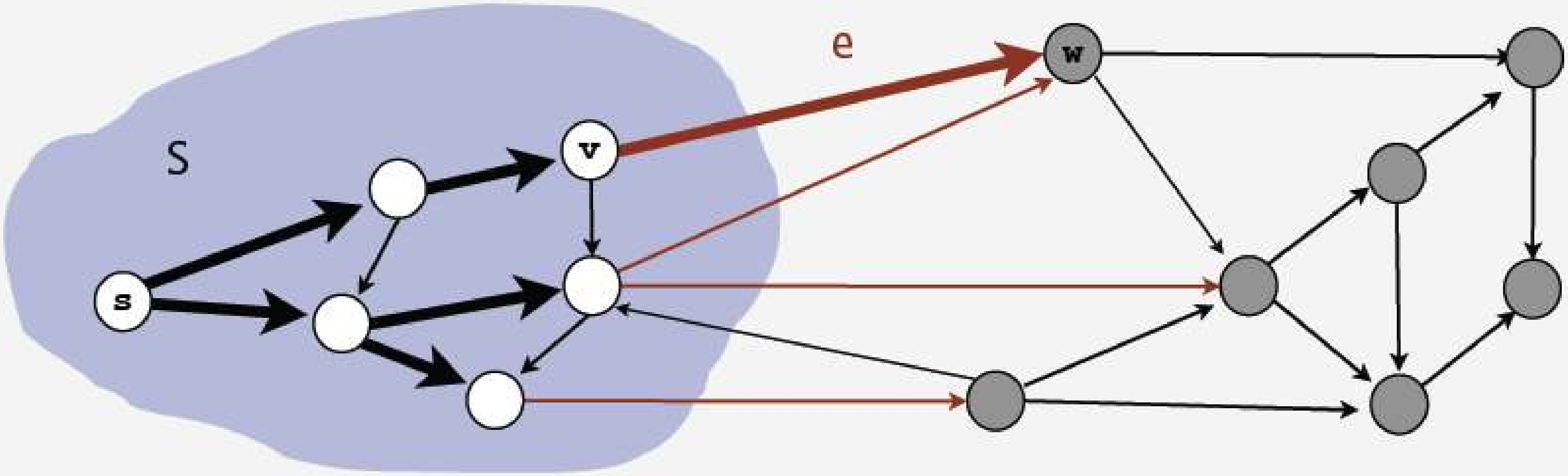
- Maintain a set of explored vertices S.
- Grow S by exploring edges with exactly one endpoint leaving S.

DFS. Take edge from vertex which was discovered most recently.

BFS. Take edge from vertex which was discovered least recently.

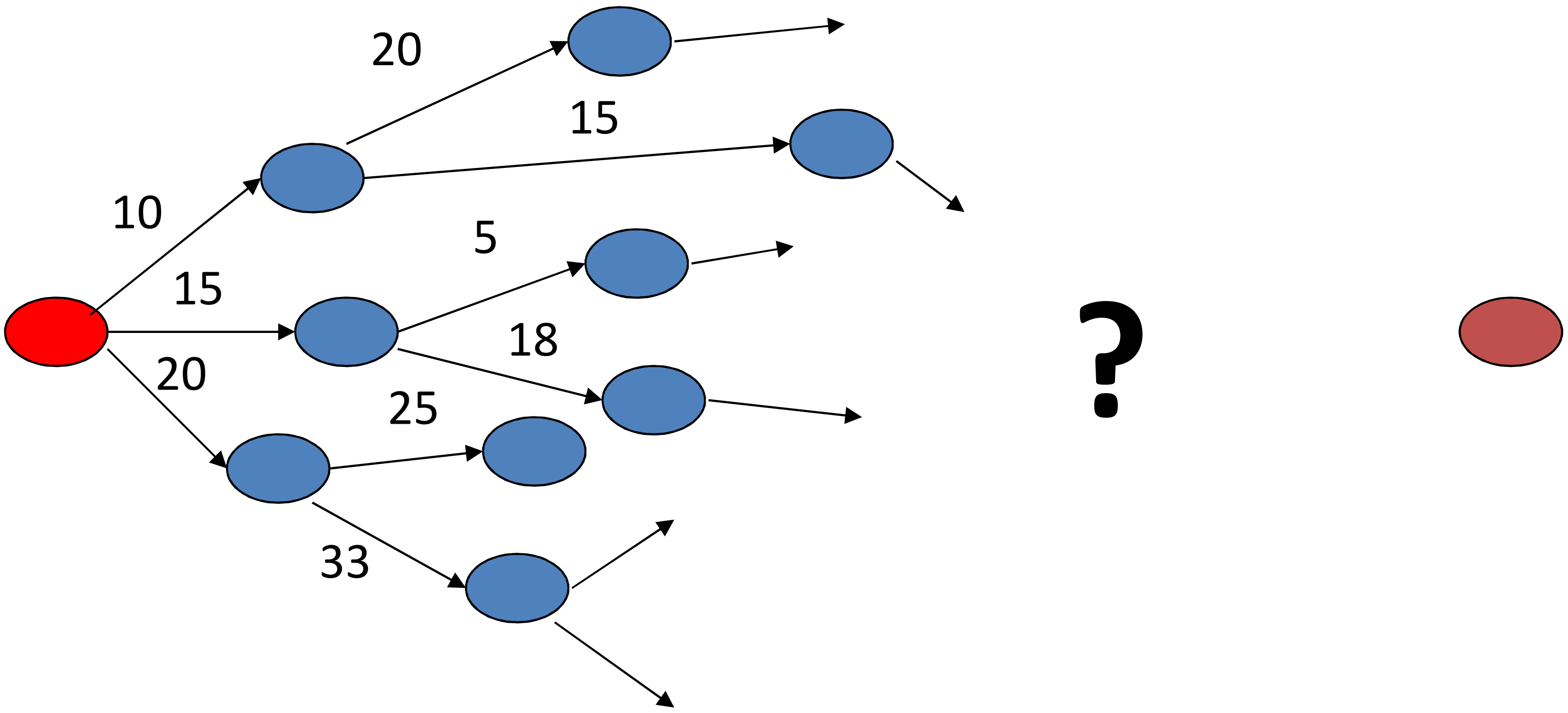
Prim. Take edge of minimum weight.

Dijkstra. Take edge to vertex that is closest to S.



MapQuest

- Shortest path for a single source-target pair
- Dijkstra algorithm can be used



Better Solution: Make a ‘hunch”!

- Use ***heuristics*** to guide the search
  - **Heuristic**: estimation or “hunch” of how to search for a solution
- We define a heuristic function:

  h(n) = “estimate of the cost of the cheapest path from the starting node to the goal node”

The A* Search

- A* is an algorithm that:
  - Uses heuristic to guide search
  - While ensuring that it will compute a path with minimum cost

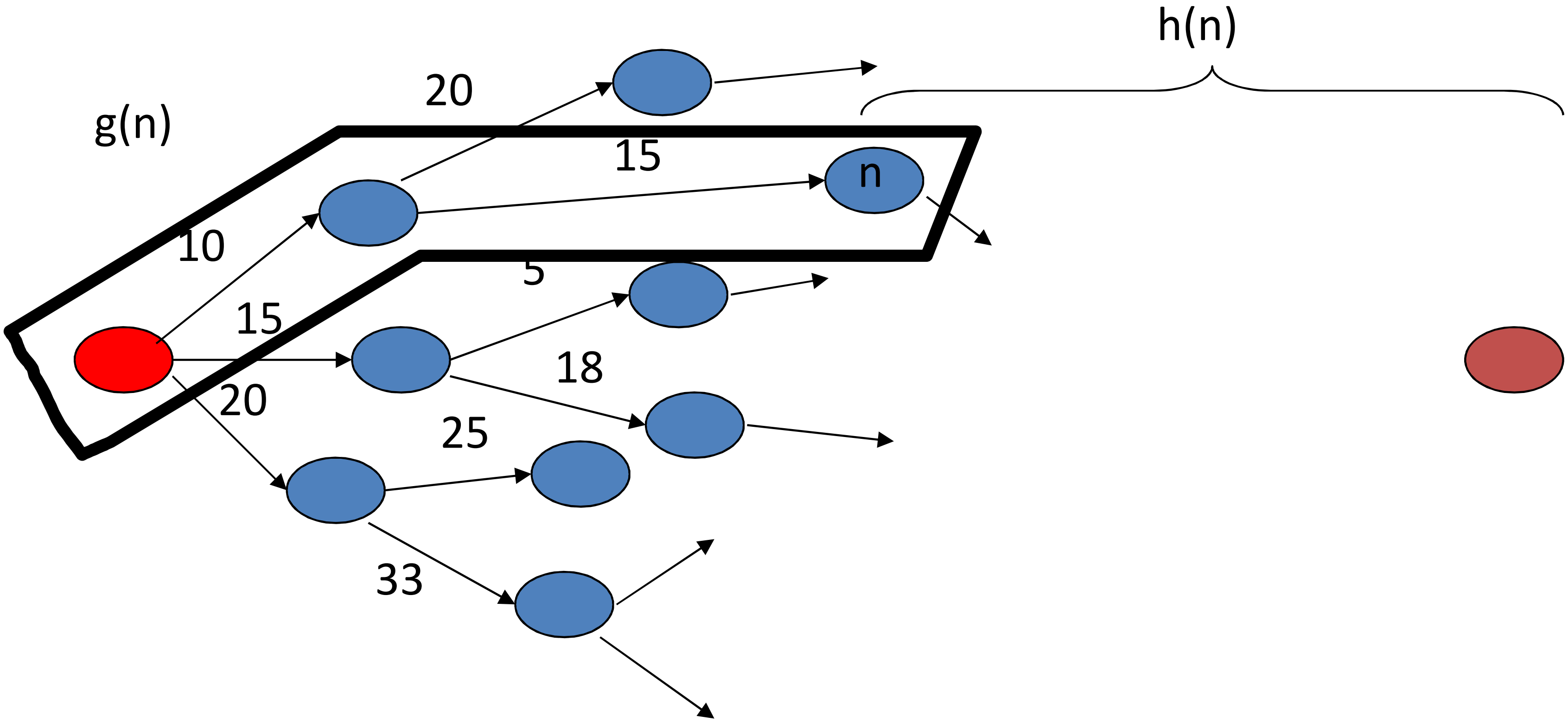
“estimated cost”

- A* computes the function f(n) = g(n) + h(n)

“actual cost”

A*

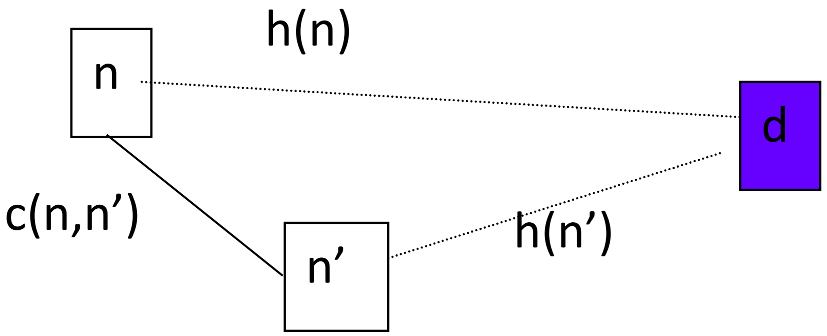
- f(n) = g(n) + h(n)
  - g(n) = “cost from the starting node to reach n”
  - h(n) = “estimate of the cost of the cheapest path from n to the goal node”



Properties of A*

- A* generates an optimal solution if h(n) is an admissible heuristic and the search space is a tree:
  - h(n) is **admissible** if it never overestimates the cost to reach the destination node
- A* generates an optimal solution if h(n) is a consistent heuristic and the search space is a graph:
  - h(n) is **consistent** if for every node n and for every successor node n’ of n:

    $h(n) \leq c(n,n') + h(n')$



- If h(n) is consistent then h(n) is admissible
- Frequently when h(n) is admissible, it is also consistent

Admissible Heuristics

- A heuristic is admissible if it is optimistic, estimating the cost to be smaller than it actually is.

- MapQuest:

h(n) = “Euclidean distance to destination”

is admissible as normally cities are not connected by roads that make straight lines