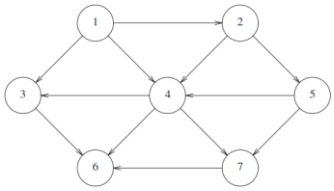
Chapter 9
Graph Algorithms

1

Introduction

- graph theory
  - useful in practice
  - represent many real-life problems
  - can be slow if not careful with data structures



2

2

Definitions

- an undirected graph $G = (V, E)$ is a finite set V of vertices together with a set E of edges
- an edge is a pair (v, w), where v and w are vertices

- this definition allows
  - self-loops, or edges that connect vertices to themselves
  - parallel edges, or multiple edges that connect the same pair of vertices

- a graph without self-loops is a simple graph
- a graph with parallel edges is sometimes called a multigraph

3

3

Definitions

- two vertices are adjacent if there is an edge between them
  - the edge is said to be incident to the two vertices

- if there are no parallel edges, the degree of a vertex is the number of edges incident to it
  - self-loops add only 1 to the degree

- a subgraph of a graph G is a subset of G's edges together with the incident vertices

4

4

Definitions

- a path in a graph is a sequence of vertices connected by edges
  - a simple path is a path with no repeated vertices, except possibly the first and last

- a cycle is a path of at least one edge whose first and last vertices are the same
  - a simple cycle is a cycle with no repeated edges of vertices other than the first and last

- the length of a path is the number of edges in the path

5

5

Definitions

- a graph is connected if every vertex is connected to every other vertex by a path through the graph

- a connected component G' of a graph G is a maximal connected subgraph of G: if G' is a subset of F and F is a connected subgraph of G, then $F = G'$

- a graph that is not connected consists of a set of connected components

- a graph without cycles is called acyclic

6

6

Definitions

- a tree is a connected, acyclic undirected graph
- a forest is a disjoint set of trees
- a spanning tree of a connected graph is a subgraph that is a tree and also contains all of the graph's vertices
- a spanning forest of a graph is the union of spanning trees of its connected components

7

7

Definitions

- if $|V|$ is the number of vertices and $|E|$ is the number of edges, then, in a graph without self-loops and parallel edges, there are $|V|\,(|V|-1)/2$ possible edges
- a graph is complete if there is an edge between every pair of vertices
- the density of a graph refers to the proportion of possible pairs of vertices that are connected
- a sparse graph is one for which $|E| \ll |V|(|V|-1)/2$
- a dense graph is a graph that is not sparse
- a bipartite graph is one whose vertices can be divided into two sets so that every vertex in one set is connected to at least one vertex in the other set

8

8

Definitions

- in a directed graph or digraph the pairs (v, w) indicating edges are ordered: the edge (v, w) goes from v (the tail) to w (the head)
- since edges have a direction, we use the notation $v \to w$ to denote an edge from v to w
- edges in digraphs are frequently called arcs
- the indegree of a vertex w is the number of arcs $v \to w$ (i.e., the number of arcs coming into w), while the outdegree of w is the number of arcs $w \to v$ (i.e., the number of arcs exiting w)
- we will call w a source if its indegree is 0
- an aborescence is a directed graph with a distinguished vertex u (the root) such that for every other vertex v there is a unique directed path from u to v

9

9

Definitions

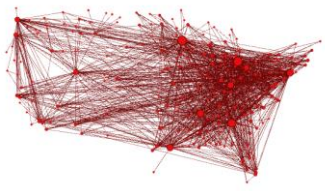
- in a directed graph, two vertices v and w are strongly connected if there is a directed path from v to w and a directed path from w to v
- a digraph is strongly connected if all its vertices are strongly connected
- if a digraph is not strongly connected but the underlying undirected graph is connected, then the digraph is called weakly connected
- a weighted graph has weights or costs associated with each edge
  - weighted graphs can be directed or undirected
  - a road map with mileage is the prototypical example

10

10

Example

- airport connections



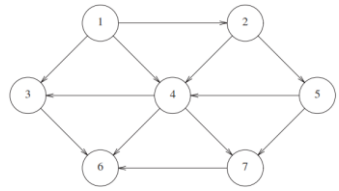
http://allthingsgraphed.com/2014/08/09/us-airlines-graph/

11

11

Graph Representation

- two concerns: memory and speed
- we'll consider directed graphs, though undirected graphs are similar
- the following graph has 7 vertices and 12 edges

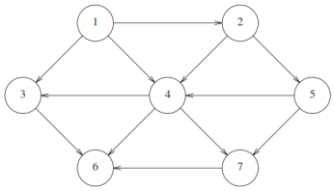


12

12

Graph Representation

- adjacency matrix
  - 2D matrix where an element is 1 if $(u, v) \in A$ and 0 otherwise



	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[1]	0	1	1	1	0	0	0
[2]	0	0	0	1	1	0	0
[3]	0	0	0	0	0	1	0
[4]	0	0	1	0	0	1	1
[5]	0	0	0	1	0	0	1
[6]	0	0	0	0	0	0	0
[7]	0	0	0	0	0	1	0

13

13

Graph Representation

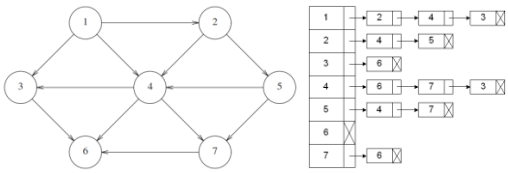
- adjacency matrix
  - alternatively, could use costs ∞ or $-\infty$ for non-edges
  - not efficient if the graph is sparse (number of edges small)
    - matrix $O(|V|^2)$
    - e.g., street map with 3,000 streets results in intersection matrix with 9,000,000 elements
- adjacency list
  - standard way to represent graphs
  - undirected graph edges appear twice in list
  - more efficient if the graph is sparse (number of edges small)
    - matrix $O(|E| + |V|)$

14

14

Graph Representation

- adjacency list
  - for each vertex, keep a list of adjacent vertices

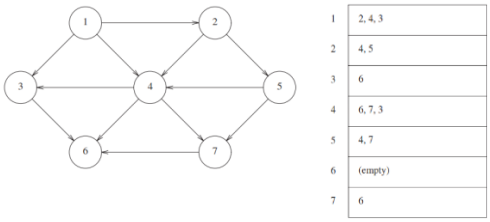


15

15

Graph Representation

- adjacency list alternative
  - for each vertex, keep a vector of adjacent vertices



16

16

Topological Sort

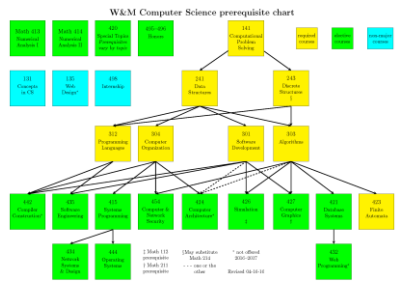
- a directed acyclic graph (DAG) is a digraph with no directed cycles
  - a DAG always has at least one vertex
- topological sort
  - an ordering of the vertices in a directed graph such that if there is a path from v to w, then v appears before w in the ordering
  - not possible if graph has a cycle

17

17

Topological Sort

- example directed acyclic graph



18

18

Topological Sort

- topological sort
  - determine the indegree for every $v \in V$
    - place all source vertices in a queue
  - while there remains a $v \in V$
    - find a source vertex
    - append the source vertex to the topological sort
    - delete the source and its adjacent arcs from G
    - update the indegrees of the remaining vertices in G
    - place any new source vertices in the queue
- when no vertices remain, we have our ordering, or, if we are missing vertices from the output list, the graph has no topological sort

19

19

Topological Sort

```
void Graph::topsort( )
{
    for( int counter = 0; counter < NUM_VERTICES; counter++ )
    {
        Vertex v = findNewVertexOfIndegreeZero( );
        if( v == NOT_A_VERTEX )
            throw CycleFoundException( );
        v.topNum = counter;
        for each Vertex w adjacent to v
            w.indegree--;
    }
}
```

- since finding vertex with 0 indegree must look at all vertices, and this is performed $|V|$ times, $O(|V|^2)$

20

20

Topological Sort

- instead, we can keep all the vertices with indegree 0 in a list and choose from there
- $O(|E| + |V|)$

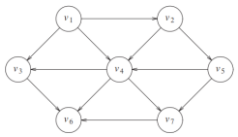
```
void Graph::topsort( )
{
    Queue<Vertex> q;
    int counter = 0;

    q.makeEmpty( );
    for each Vertex v
        if( v.indegree == 0 )
            q.enqueue( v );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );
        v.topNum = ++counter;  // Assign next number

        for each Vertex w adjacent to v
            if( --w.indegree == 0 )
                q.enqueue( w );
    }

    if( counter != NUM_VERTICES )
        throw CycleFoundException( );
}
```

21

21

Topological Sort

- use a table to keep track of the source vertices



	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v_1	0	0	0	0	0	0	0
v_2	1	0	0	0	0	0	0
v_3	2	1	1	1	0	0	0
v_4	3	2	1	0	0	0	0
v_5	1	1	0	0	0	0	0
v_6	3	3	3	3	2	1	0
v_7	2	2	2	1	0	0	0
Enqueue	v_1	v_2	v_5	v_4	v_3, v_7		v_6
Dequeue	v_1	v_2	v_5	v_4	v_3	v_7	v_6

22

22

Shortest-Path Algorithms

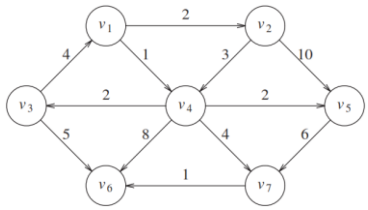
- shortest-path problems
  - input is a weighted graph with a cost on each edge
  - weighted path length: $\sum_{i=1}^{N-1} c_{i,i+1}$
- single-source shortest-path problem
  - given as input a weighted graph, $G = (V, E)$ and a distinguished vertex s, find the shortest weighted path from s to every other vertex in G

23

23

Shortest-Path Algorithms

- example
  - shortest weighted path from v_1 to v_6 has cost of 6
  - no path from v_6 to v_1

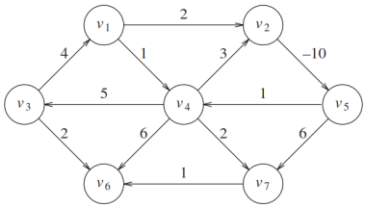


24

24

Shortest-Path Algorithms

- negative edges can cause problems
  - path from v_5 to v_4 has cost of 1, but a shorter path exists by following the negative loop, which has cost -5
  - shortest paths thus undefined



25

25

Shortest-Path Algorithms

- many examples where we want to find shortest paths
  - if vertices represent computers and edges connections, the cost represents communication costs, delay costs, or combination of costs
  - if vertices represent airports and edges costs to travel between them, shortest path is cheapest route
- we find paths from one vertex to all others since no algorithm exists that finds shortest path from one vertex to one other faster

26

26

Shortest-Path Algorithms

- four problems
  - unweighted shortest-path
  - weighted shortest-path with no negative edges
  - weighted shortest-path with negative edges
  - weighted shortest-path in acyclic graphs

27

27

Unweighted Shortest Paths

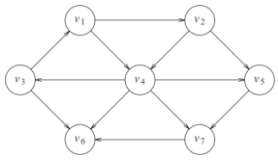
- unweighted shortest-path
  - find shortest paths from s to all other vertices
  - only concerned with number of edges in path
    - we will not actually record the path elements

28

28

Unweighted Shortest Paths

- example
  - start with v_3

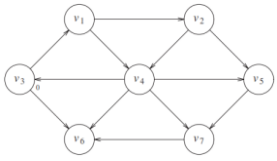


29

29

Unweighted Shortest Paths

- example
  - mark 0 length to v_3

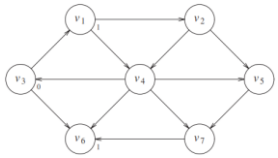


30

30

Unweighted Shortest Paths

- example
  - mark 1 length for v_1 and v_6

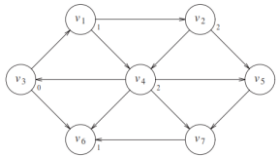


31

31

Unweighted Shortest Paths

- example
  - mark 2 length for v_2 and v_4

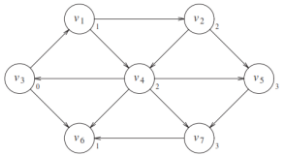


32

32

Unweighted Shortest Paths

- example
  - final path assignments



33

33

Unweighted Shortest Paths

- searching an unweighted shortest-path uses a breadth-first search
  - processes vertices in layers, according to distance
  - begins with initializing path lengths

v	known	d_v	p_v
v_1	F	∞	0
v_2	F	∞	0
v_3	F	0	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

  - a vertex will be marked known when the shortest path to it is found

34

34

Unweighted Shortest Paths

```
void Graph::unweighted( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
        for each Vertex v
            if( !v.known && v.dist == currDist )
            {
                v.known = true;
                for each Vertex w adjacent to v
                    if( w.dist == INFINITY )
                    {
                        w.dist = currDist + 1;
                        w.path = v;
                    }
            }
}
```

35

35

Unweighted Shortest Paths

- with this algorithm
  - path can be printed
  - running time: $O(|V|^2)$
  - bad case

  - can reduce time by keeping vertices that are unknown separate from those known
  - new running time: $O(|E| + |V|)$

36

36

Unweighted Shortest Paths

```
void Graph::unweighted( Vertex s )
{
    Queue<Vertex> q;

    for each Vertex v
        v.dist = INFINITY;

    s.dist = 0;
    q.enqueue( s );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );

        for each Vertex w adjacent to v
            if( w.dist == INFINITY )
            {
                w.dist = v.dist + 1;
                w.path = v;
                q.enqueue( w );
            }
    }
}
```

37

37

Unweighted Shortest Paths

	Initial State			v_3 Dequeued			v_1 Dequeued			v_6 Dequeued		
v	known	d_v	p_v	known	d_v	p_v	known	d_v	p_v	known	d_v	p_v
v_1	F	∞	0	F	1	v_3	T	1	v_3	T	1	v_3
v_2	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_3	F	0	0	T	0	0	T	0	0	T	0	0
v_4	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_5	F	∞	0	F	∞	0	F	∞	0	F	∞	0
v_6	F	∞	0	F	1	v_3	F	1	v_3	T	1	v_3
v_7	F	∞	0	F	∞	0	F	∞	0	F	∞	0
Q:		v_3			v_1, v_6			v_6, v_2, v_4			v_2, v_4	

	v_2 Dequeued			v_4 Dequeued			v_5 Dequeued			v_7 Dequeued		
v	known	d_v	p_v	known	d_v	p_v	known	d_v	p_v	known	d_v	p_v
v_1	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_2	T	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_3	T	0	0	T	0	0	T	0	0	T	0	0
v_4	F	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_5	F	3	v_2	F	3	v_2	T	3	v_2	T	3	v_2
v_6	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_7	F	∞	0	F	3	v_4	F	3	v_4	T	3	v_4
Q:		v_4, v_5			v_5, v_7			v_7			empty	

38

38

Dijkstra's Algorithm

- weighted shortest-path – Dijkstra's algorithm
  - more difficult, but ideas from unweighted algorithm can be used
  - keep information as before for each vertex
    - known
    - set distance $d_w = d_v + c_{v,w}$ if $d_w = \infty$ using only known vertices
    - p_v the last vertex to cause a change to d_v
- greedy algorithm
  - does what appears to be best thing at each stage
  - e.g., counting money: count quarters first, then dimes, nickels, pennies
    - gives change with least number of coins

39

39

Dijkstra's Algorithm

- pseudocode
  - assumption: no negative weights
  - origin s is given

Initialization: $S \leftarrow \{s\}$ and $D \leftarrow V - \{s\}$.
Set dist[s] ← 0 and dist[v] ← ∞ for all other v.

While there remains a $v \in D$:

1. Select a vertex $v \in D$ which has the shortest path length from s to v using only vertices in S (e.g., known vertices).
2. $S \leftarrow S \cup \{v\}$ and $D \leftarrow D - \{v\}$.

40

40

Dijkstra's Algorithm

- pseudocode (cont.)

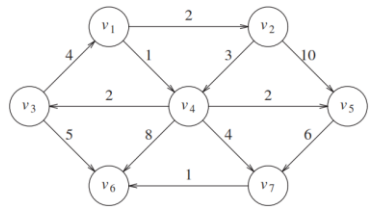
```
foreach vertex v {
  dist[v] = +∞
  known[v] = false
}
dist[s] = 0

while (there is a vertex w with known[w] == false) {
  v = argmin({ dist[w] | known[w] == false })
  known[v] = true
  foreach (w adjacent to v) {
    if (known[w] == false) { // edge relaxation
      dist[w] = min(dist[w], dist[v] + weight(v → w))
      from[w] = v
    }
  }
}
```

41

41

Dijkstra's Algorithm

- example: start at v_1



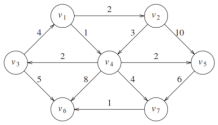
v	known	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

42

42

Dijkstra's Algorithm

- example



v	known	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	∞	0
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

v_1

v	known	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

v_4

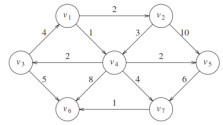
v	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

v_2

43

Dijkstra's Algorithm

- example



v	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	8	v_3
v_7	F	5	v_4

v_5, v_3

v	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	6	v_7
v_7	T	5	v_4

v_7

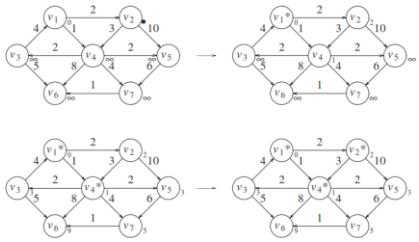
v	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	T	6	v_7
v_7	T	5	v_4

v_6

44

Dijkstra's Algorithm

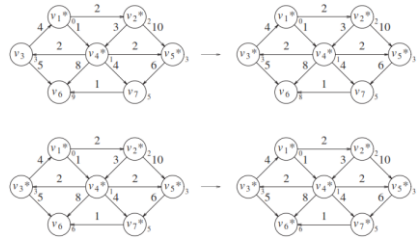
- example – stages shown on the graph



45

Dijkstra's Algorithm

- example – stages shown on the graph (cont.)



46

Dijkstra's Algorithm: Correctness

Proposition. Dijkstra's algorithm solves the single-origin shortest-paths problem in a weight digraph with nonnegative weights.

Proof. If v is reachable from s, then every edge v →w is relaxed exactly once, when v is relaxed, resulting in

$$\text{dist[w]} \leq \text{dist[v]} + \text{weight[v} \rightarrow \text{w]}.$$

This inequality holds until the algorithm terminates, since

1. dist[w] can only decrease, because relaxations can only decrease a dist[] value, and
2. dist[v] never changes, because edge weights are nonnegative and we choose the lowest dist[] value at each step, so no later relaxation can reduce dist[v].

Thus, after all vertices reachable from s have been added to the shortest paths tree, the shortest paths optimality conditions hold.

47

Dijkstra's Algorithm

- complexity
  - sequentially scanning vertices to find minimum d_v takes $O(|V|)$, which results in $O(|V|^2)$ overall
  - at most one update per edge, for a total of $O(|E| + |V|^2) = O(|V|^2)$
    - if graph is dense, with $|E| = \Theta(|V|^2)$, algorithm is close to optimal
    - if graph is sparse, with $|E| = \Theta(|V|)$, algorithm is too slow
- distances could be kept in a priority queue that reduces running time to $O(|E| + |V| \lg |V|)$

48

Dijkstra's Algorithm

- implementation
  - information for each vertex

```
/**
 * PSEUDOCODE sketch of the Vertex structure.
 * In real C++, path would be of type Vertex *,
 * and many of the code fragments that we describe
 * require either a dereferencing * or use the
 * -> operator instead of the . operator.
 * Needless to say, this obscures the basic algorithmic ideas.
 */
struct Vertex
{
    List      adj;    // Adjacency list
    bool      known;
    DistType  dist;   // DistType is probably int
    Vertex    path;   // Probably Vertex *, as mentioned above
        // Other data and member functions as needed
};
```

49

49

Dijkstra's Algorithm

- implementation (cont.)
  - path can be printed recursively

```
/**
 * Print shortest path to v after dijkstra has run.
 * Assume that the path exists.
 */
void Graph::printPath( Vertex v )
{
    if( v.path != NOT_A_VERTEX )
    {
        printPath( v.path );
        cout << " to ";
    }
    cout << v;
}
```

50

50

Dijkstra's Algorithm

- implementation (cont.)

```
void Graph::dijkstra( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    while( there is an unknown distance vertex )
    {
        Vertex v = smallest unknown distance vertex;

        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
            {
                DistType cvw = cost of edge from v to w;

                if( v.dist + cvw < w.dist )
                {
                    // Update w
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
            }
    }
}
```

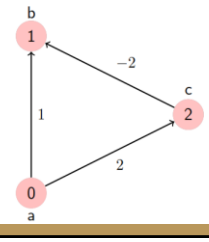
51

51

Graphs with Negative Edges

- try to apply Dijkstra's algorithm to graph with negative edges

Label each node with best known distance from origin a. Relax the edges adjacent to a. Select b, the closest node to S, and add it to S. Relax the outgoing edges adjacent to b. Select c, the closest node to S, and add it to S. We've now assimilated all nodes into S, so we're done.



52

52

Graphs with Negative Edges

- possible solution: add a delta value to all weights such that none are negative
  - calculate shortest path on new graph
  - apply path to original graph
  - does not work: longer paths become weightier
- combination of algorithms for weighted graphs and unweighted graphs can work
  - drastic increase in running time: $O(|E| \cdot |V|)$

53

53

All-Pairs Shortest Paths

- given a weighted digraph, find the shortest paths between all vertices in the graph
- one approach: apply Dijkstra's algorithm repeatedly
  - results in $O(|V|^3)$
- another approach: apply Floyd-Warshall algorithm
  - uses dynamic programming
  - also results in $O(|V|^3)$

54

54

Minimum Spanning Tree

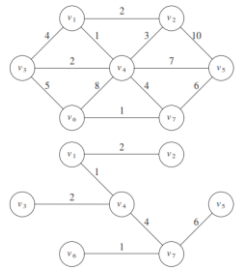
- assumptions
  - graph is connected
  - edge weights are not necessarily Euclidean distances
  - edge weights need not be all the same
  - edge weights may be zero or negative
- minimum spanning tree (MST)
  - also called minimum-weight spanning tree of a weighted graph
  - spanning tree whose weight (the sum of the weights of the edges in the tree) is the smallest among all spanning trees

55

55

Minimum Spanning Tree

example



56

56

Minimum Spanning Tree

- two algorithms to find the minimum spanning tree
  - Prim's Algorithm
    - R. C. Prim, *Shortest Connection Networks and Some Generalizations*, *Bell System Technical Journal* (1957)
  - Kruskal's Algorithm
    - J. B. Kruskal, Jr., *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem*, *Proc. of the American Mathematical Society* (1956)

57

57

Minimum Spanning Tree

- Prim's algorithm
  - grows tree in successive stages

Initialization: $S \leftarrow \{s\}$ and $D \leftarrow V - \{s\}$.

While there remains a $v \in D$:

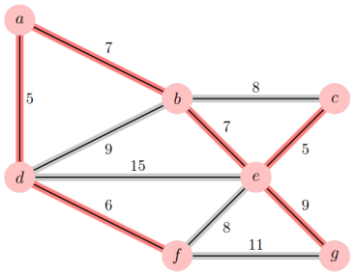
1. Find an edge with minimum weight (u, v) such that $u \in S$ and $v \in D$.
2. $S \leftarrow S \cup \{v\}$ and $D \leftarrow D - \{v\}$.

58

58

Minimum Spanning Tree

- Prim's algorithm: example

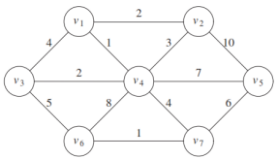


59

59

Minimum Spanning Tree

- Prim's algorithm: example 2

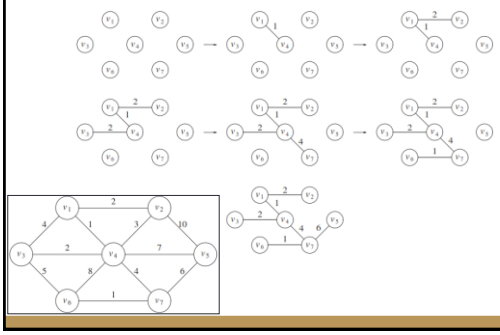


60

60

Minimum Spanning Tree

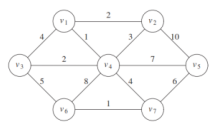
- Prim's algorithm: example 2 (cont.)



61

Minimum Spanning Tree

- Prim's algorithm: example 2 (cont.)
  - v_1, v_4, v_2 & v_3, v_7, v_6 & v_5



v	known	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

v	known	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	4	v_1
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

v	known	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	2	v_4
v_4	T	1	v_1
v_5	F	7	v_4
v_6	F	8	v_4
v_7	F	4	v_4

v	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	F	7	v_4
v_6	F	5	v_3
v_7	F	4	v_4

v	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	F	6	v_7
v_6	F	1	v_7
v_7	T	4	v_4

v	known	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	T	6	v_7
v_6	T	1	v_7
v_7	T	4	v_4

62

Minimum Spanning Tree

- Prim's algorithm
  - runs on undirected graphs
  - running time: $O(|V|^2)$ without heaps, which is optimal for dense graphs
  - running time: $O(|E| \lg |V|)$ using binary heaps, which is good for sparse graphs

63

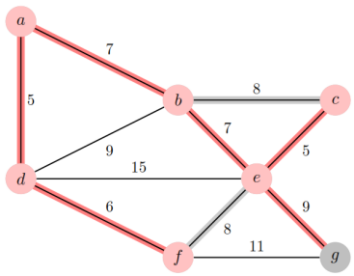
Minimum Spanning Tree

- Kruskal's algorithm
  - continually select the edges in order of smallest weight
  - accept the edge if it does not cause a cycle with already accepted edges

64

Minimum Spanning Tree

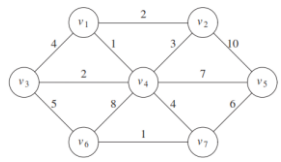
- Kruskal's algorithm: example



65

Minimum Spanning Tree

- Kruskal's algorithm: example 2

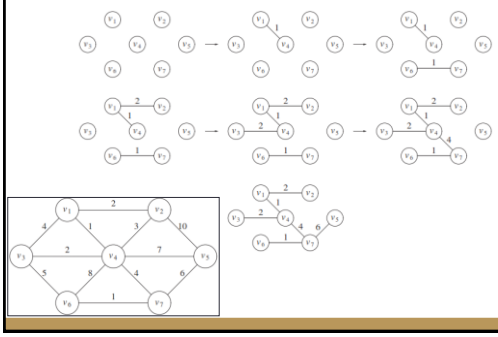


Edge	Weight	Action
(v_1, v_4)	1	Accepted
(v_6, v_7)	1	Accepted
(v_1, v_2)	2	Accepted
(v_3, v_4)	2	Accepted
(v_2, v_4)	3	Rejected
(v_1, v_3)	4	Rejected
(v_4, v_7)	4	Accepted
(v_3, v_6)	5	Rejected
(v_5, v_7)	6	Accepted

66

Minimum Spanning Tree

- Kruskal's algorithm: example 2 (cont.)



67

Minimum Spanning Tree

- Kruskal's algorithm
  - time complexity: $O(|E| \lg |E|)$ with proper choice and use of data structures
  - in the worst case, $|E| = \Theta(|V|^2)$, so the worst-case time complexity is $O(|E| \lg |V|)$

68

NP-Complete Problems

- What problems can we solve algorithmically? which problems are easy? which problems are hard?
  - Eulerian circuit: Given a vertex *s*, start at *s* and find a cycle that visits every edge exactly once
    - easy: solvable in $O(|E| + |V|)$ using depth-first search
  - Hamiltonian circuit: Given a vertex *s*, start at *s* and find a cycle that visits each remaining vertex exactly once
    - really, really hard!

69

NP-Complete Problems

- halting problem
  - in 1936, A. Church and A. Turing independently proved the non-solvability of the halting problem:
    - is there an algorithm *terminates(p,x)* that takes an arbitrary program *p* and input *x* and returns True if *p* terminates when given input *x* and False otherwise?
    - difficult: try to run it on itself

70

NP-Complete Problems

- halting problem
  - suppose we had such an algorithm *terminates(p,x)*
  - create a new program:

```
program evil (z) {
  1: if terminates(z,z) goto 1
}
```

  - program *evil()* terminates if and only if the program *z* does not terminate when given its own code as input
    - no such algorithm can exist

71

NP-Complete Problems

- decision problem
  - has a yes or no answer
  - undecidable if it is impossible to construct a single algorithm that will solve all instances of the problem
  - the halting problem is undecidable

72

NP-Complete Problems

- the class *P*
  - set of problems for which there exists a polynomial time algorithm for their solution
  - the runtime is bounded by a polynomial function of the size of the problem
- the class *NP*
  - set of decision problems for which the certification of a candidate solution as being correct can be performed in polynomial time
  - non-deterministic polynomial time

73

73

NP-Complete Problems

- the class *NP*
  - for problems in *NP*, certifying a solution may not be difficult, but finding a solution may be very difficult
  - example: Hamiltonian circuit
    - given a graph *G*, is there a simple cycle in *G* that includes every vertex?
    - given a candidate solution, we can check whether it is a simple cycle in time $\propto |V|$, simply by traversing the path
    - however, finding a Hamiltonian circuit is hard!

74

74

NP-Complete Problems

- reductions
  - problem *A* reduces to problem *B* if the solvability of *B* implies the solvability of *A*
    - if *A* is reducible to *B*, then *B* is at least as hard to solve as *A*
  - in the context of algorithms, reducibility means an algorithm that solves *B* can be converted into an algorithm to solve *A*
    - example: if we can sort a set of numbers, we can find the median, so finding the median reduces to sorting

75

75

NP-Complete Problems

- reductions
  - problem *A* can be polynomially reduced to *B* if we can solve problem *A* using an algorithm for problem *B* such that the cost of solving *A* is

    cost of solving *B* + a polynomial function of the problem size
  - example: once we have sorted an array $a[]$ of N numbers, we can find the median in constant time by computing $N/2$ and accessing $a[N/2]$

76

76

NP-Complete Problems

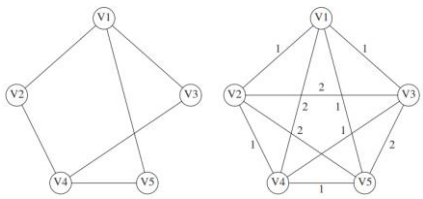
- reductions
  - decision version of traveling salesperson problem (TSP):
    - given a complete weighted graph and an integer K, does there exist a simple cycle that visits all vertices (tour) with total weight $\leq K$?
      - clearly, this is in *NP*
  - Hamiltonian circuit: given a graph $G = (V, E)$, find a simple cycle that visits all the vertices
    - construct a new graph G' with the same vertices as G but which is complete; if an edge in G' is in G, give it weight 1; otherwise, give it weight 2
      - construction requires $O(|E| + |V|)$ work
      - apply TSP to see if there exists a tour with total weight $|V|$

77

77

NP-Complete Problems

- reductions



78

78

NP-Complete Problems

- *NP*-complete
  - a problem *A* is *NP*-complete if it is in *NP* and all other problems in *NP* can be reduced to *A* in polynomial time
  - Boolean satisfiablity (SAT): given a set of *N* boolean variables and *M* logical statements built from the variables using *and* and *not*, can you choose values for the variables so that all the statements are true?
    $(x_1\ AND\ !\,x_2\ AND\ x_3), (!\,x_1\ AND\ x_7), (x_1\ AND\ x_{42}), \ldots$
  - SAT is *NP*-complete

79

79

NP-Complete Problems

- *NP*-complete
  - if we restrict attention to sets of boolean statements involving 3 variables, the problem is known as 3-SAT
    - 3-SAT is *NP*-complete
    - so, if you can solve 3-SAT in polynomial time, you can solve all problems in *NP* in polynomial time
    - meanwhile, 2-SAT is solvable in linear time!

80

80

NP-Complete Problems

- *NP*-complete problems
  - traveling salesperson
  - bin packing
  - knapsack
  - graph coloring
  - longest-path

81

81

NP-Complete Problems

- *NP*-hard problems
  - a problem *A* is *NP*-hard if there exists a polynomial-time reduction from an *NP*-complete problem to *A*
  - an *NP*-hard problem is at least as hard as an *NP*-complete problem
  - optimization versions of *NP*-complete problems are typically *NP*-hard
    - optimization version of TSP: given a weighted graph, find a minimum cost Hamiltonian circuit
    - if we can solve TSP, we can solve Hamiltonian circuit

82

82

Bin Packing

We are given n items of lengths $\ell_1, \ldots, \ell_n$, where $0 < \ell_i \leq 1$ for all i.

The items must be packed in bins of length 1, and they must be placed end-to-end. Once placed in a bin, items cannot be moved.

How do we pack them in a way that uses the fewest number of bins?
This is an NP-hard problem!

Input: 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8.

Optimal packing:
B_1: 0.2, 0.8
B_2: 0.7, 0.3
B_3: 0.4, 0.1, 0.5

Online: item must be put in a bin before the next item is considered.

Offline: all the items are available for consideration at one time.

83

83

Bin Packing

Theorem. No algorithm for online bin packing can always give an optimal solution.

Proof. Let $\varepsilon > 0$ be small (say, $\varepsilon = 1/64$), and consider an input sequence of m items of length $\frac{1}{2} - \varepsilon$ followed by m items of length $\frac{1}{2} + \varepsilon$.

Clearly, the optimal packing requires m bins.

Suppose online algorithm A yields this optimal solution. Since it is online, A must place each of the first m items in a separate bin.

Now give A an input sequence of just m items of length $\frac{1}{2} - \varepsilon$. A will behave as before, placing each in a separate bin, thus using m bins.

However, the optimal packing in this case requires only $\lceil m/2 \rceil$ bins.

84

84

Bin Packing

Since an online algorithm never knows when the input might end, any performance guarantee for the algorithm must hold whenever an item is binned.

Theorem. There are inputs that cause any online bin packing algorithm to use at least $\frac{4}{3}$ the optimal number of bins.

Proof. Suppose not. Then there is an online algorithm A that always uses less than $\frac{4}{3}$ the optimal number of bins.

Let m be even. Apply A to an input sequence of m items of length $\frac{1}{2} - \varepsilon$ followed by m items of length $\frac{1}{2} + \varepsilon$.

Consider the situation after A has processed item m (the last of the smaller items). Suppose A has used b bins at this point.

85

Bin Packing

We know that the optimal number of bins for the first m items is $m/2$, so our performance assumption means $b < \frac{4}{3}\frac{m}{2}$, or $2b/m < \frac{4}{3}$.

Now consider the situation when A is finished packing all $2m$ items. All the bins used after bin b can only contain one item since the inputs are the longer items.

The first b bins can have at most 2 items each, and the remaining bins have one item each, so packing all $2m$ items requires at least $2m - b$ bins.

Since the optimal packing requires m bins, the performance assumption means $2m - b < \frac{4}{3}m$, or $(2m - b)/m < \frac{4}{3}$.

86

Bin Packing

Thus we have two inequalities that hold:

$$\frac{2b}{m} < \frac{4}{3} \quad \text{after the first } m \text{ items,}$$
$$\frac{2m-b}{m} < \frac{4}{3} \quad \text{after the last } m \text{ items.}$$

From the first inequality we obtain $b/m < \frac{2}{3}$, while from the second we obtain $b/m > \frac{2}{3}$, which is a contradiction. ☝

87

Bin Packing

The next fit heuristic: when binning an item, check to see if it fits in the bin with the last item binned. If it does, place the new item there; otherwise, start a new bin.

Input: 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8.

B_1: 0.2, 0.5
B_2: 0.4
B_3: 0.7, 0.1
B_4: 0.3
B_5: 0.8

88

Bin Packing

Theorem

Let m be the optimal number of bins required to pack a list of items. Then next fit never uses more than $2m$ bins. There exist sequences for which next fit uses $2m - 2$ bins.

This is an example of an approximation result.

Next fit is an approximation algorithm; it is not guaranteed to produce optimal solutions, but there exists a bound on how poorly it can do, even though we do not know the optimal solution.

89

Bin Packing

Proof: Suppose next fit uses b bins. Let L_j be the total length of items in bin j; then

$$L_1 + L_2 + \cdots + L_b = \text{total length of items} \leq m.$$

Consider any adjacent bins B_j and B_{j+1}. The combined lengths of the items in these two bins must be larger than 1; otherwise all of these items would have been placed in B_j:

$$L_j + L_{j+1} > 1.$$

For simplicity, assume b is even. Then

$$m \geq \underbrace{(L_1 + L_2)}_{1} + \underbrace{(L_3 + L_4)}_{2} + \cdots + \underbrace{(L_{b-1} + L_{b-2})}_{b/2}$$
$$> \underbrace{1 + 1 + \cdots + 1}_{b/2 \text{ times}} = \frac{b}{2},$$

so $b < 2m$.

90

Bin Packing

For the second part of the theorem, suppose n, the number of items, is divisible by 4, and choose

$$\ell_i = \begin{cases} 1/2 & \text{if } i \text{ is odd,} \\ 2/n & \text{if } i \text{ is even.} \end{cases}$$

The optimal packing requires $n/4$ bins containing 2 items of length $1/2$ and one bin containing the $n/2$ items of length $2/n$.

Next fit, on the other hand, uses $n/2$ bins, with each bin containing one long and one short item.

91

Bin Packing

The first fit heuristic: when binning an item, scan over all the bins in order and place the new item in the first bin that has room for it. If no such bin exists, open a new one.

Input: 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8.

B_1: 0.2, 0.5, 0.1
B_2: 0.4, 0.3
B_3: 0.7
B_4: 0.8

92

Bin Packing

Theorem

Let m be the optimal number of bins required to pack a list of items. Then first fit never uses more than $\frac{17}{10}m + \frac{7}{10}$ bins. There exist sequences for which next fit uses $\frac{17}{10}(m-1)$ bins.

Consider a sequence of $6N$ items of size $\frac{1}{7} + \varepsilon$, followed by $6N$ items of size $\frac{1}{3} + \varepsilon$, followed by $6N$ items of size $\frac{1}{2} + \varepsilon$.

First fit will require $10N$ bins. However, if we apply next fit, we obtain a packing that requires only $6N$ bins.

93

Bin Packing

The best fit heuristic: when binning an item, scan over all the bins in order and place the new item in the tightest spot.

Input: 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8.

B_1: 0.2, 0.5, 0.1
B_2: 0.4
B_3: 0.7, 0.3
B_4: 0.8

In general, the worst-case behavior is the same as first fit.

94

Bin Packing

In offline bin packing, we can first sort the items, longest to shortest, and then pack them according to first fit or best fit.

The corresponding heuristics are called first fit decreasing and best fit decreasing (more properly they should be first fit nonincreasing and best fit nonincreasing).

First fit decreasing for: 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1.

B_1: 0.8, 0.2
B_2: 0.7, 0.3
B_3: 0.5, 0.4, 0.1

In this case we get the optimal solution.

95

Bin Packing

Theorem

Let m be the optimal number of bins required to pack a list of items. Then first fit decreasing never uses more than $\frac{11}{9}m + \frac{6}{9}$ bins. There exist sequences for which first fit decreasing uses $\frac{11}{9}m + \frac{6}{9}$ bins.

96