COMPSCI 220S1T, 2008

Mark C. Wilson

April 1, 2008

(Di)graphs

- A digraph is a finite nonempty set V of nodes along with a set $E \subset V \times V$ of arcs. We often write $n = |V|, e = |E|$.
- Think of arc (v, w) as an arrow from v to w.
- A graph is similar, but edges between vertices are undirected. Can interpret a graph as a special type of digraph (2 arcs represent an undirected edge).
- A tree is a very special type of (di)graph.

Organizational matters

- Lecturer: Dr Mark Wilson.
- Office: City 303.588, office hours by appointment and whenever my door is open (most of the time 10am-2pm). Many questions can be answered by email.
- Textbook: must be read before lecture and brought to lecture; prizes for finding errors.
- "Handouts": available from my website; will not be numerous.
- Lectures: will stick mostly to textbook, but there may be some extra material. Please ask questions but we need only one person talking at a time.
- Other resources: lecturer, tutor, course webpages, forum, library (check books on reserve and textbook references).

(Di)graphs

- A digraph is a finite nonempty set V of nodes along with a set $E \subset V \times V$ of arcs. We often write $n = |V|, e = |E|$.

(Di)graphs

- A digraph is a finite nonempty set V of nodes along with a set $E \subset V \times V$ of arcs. We often write $n = |V|, e = |E|$.
- Think of arc (v, w) as an arrow from v to w.

(Di)graphs

- A digraph is a finite nonempty set V of nodes along with a set $E \subset V \times V$ of arcs. We often write $n = |V|, e = |E|$.
- Think of arc (v, w) as an arrow from v to w.
- A graph is similar, but edges between vertices are undirected. Can interpret a graph as a special type of digraph (2 arcs represent an undirected edge).

(Di)graphs

- A digraph is a finite nonempty set V of nodes along with a set $E \subset V \times V$ of arcs. We often write $n = |V|, e = |E|$.
- Think of arc (v, w) as an arrow from v to w.
- A graph is similar, but edges between vertices are undirected. Can interpret a graph as a special type of digraph (2 arcs represent an undirected edge).
- A tree is a very special type of (di)graph.
- Key concepts: node/vertex, arc/edge, walk, path, cycle, connected/strongly connected, (strong) component, degree/indegree/outdegree (sequence), distance, diameter, etc.

(Di)graphs

- A digraph is a finite nonempty set V of nodes along with a set $E \subset V \times V$ of arcs. We often write $n = |V|, e = |E|$.
- Think of arc (v, w) as an arrow from v to w.
- A graph is similar, but edges between vertices are undirected. Can interpret a graph as a special type of digraph (2 arcs represent an undirected edge).
- A tree is a very special type of (di)graph.
- Key concepts: node/vertex, arc/edge, walk, path, cycle, connected/strongly connected, (strong) component, degree/indegree/outdegree (sequence), distance, diameter, etc.
- Applications: many concerned with networks (communication, transport, electrical, computer, social); other interpretations such as job precedence, tournaments, molecule structure.

Computer representation of (di)graphs

- Two main ways: adjacency matrix or adjacency lists. Neither is better in every case; adjacency lists are usually better for sparse (di)graphs.

Computer representation of (di)graphs

- Two main ways: adjacency matrix or adjacency lists. Neither is better in every case; adjacency lists are usually better for sparse (di)graphs.
- Adjacency matrix needs $\Theta(n^2)$ storage, adjacency list $\Theta(n+e)$. With matrix, arc query takes $\Theta(1)$ but it takes $\Theta(d)$ with list, where d is the maximum outdegree.

Computer representation of (di)graphs

- Two main ways: adjacency matrix or adjacency lists. Neither is better in every case; adjacency lists are usually better for sparse (di)graphs.
- Adjacency matrix needs $\Theta(n^2)$ storage, adjacency list $\Theta(n+e)$. With matrix, arc query takes $\Theta(1)$ but it takes $\Theta(d)$ with list, where d is the maximum outdegree.
- Basic graph methods: add/delete node, add/delete arc, find neighbours, check adjacency, compute (in/out-)degree of node, etc; time complexity of these operations depends on particular data structures used.

Traversing a (di)graph

- How to visit all the nodes of G in an efficient and systematic way? Main idea: start somewhere, and colour in nodes as we visit them. At each step choose a coloured node, and visit one of its unvisited neighbours.

Traversing a (di)graph

- How to visit all the nodes of G in an efficient and systematic way? Main idea: start somewhere, and colour in nodes as we visit them. At each step choose a coloured node, and visit one of its unvisited neighbours.
- There are three main ways in common use:

Traversing a (di)graph

- How to visit all the nodes of G in an efficient and systematic way? Main idea: start somewhere, and colour in nodes as we visit them. At each step choose a coloured node, and visit one of its unvisited neighbours.
- There are three main ways in common use:
  - Breadth-first search (BFS): choose coloured node using first-in, first-out;

Traversing a (di)graph

- How to visit all the nodes of G in an efficient and systematic way? Main idea: start somewhere, and colour in nodes as we visit them. At each step choose a coloured node, and visit one of its unvisited neighbours.
- There are three main ways in common use:
  - Breadth-first search (BFS): choose coloured node using first-in, first-out;
  - Depth-first search (DFS): choose coloured node using last-in, first-out;

Traversing a (di)graph

- How to visit all the nodes of G in an efficient and systematic way? Main idea: start somewhere, and colour in nodes as we visit them. At each step choose a coloured node, and visit one of its unvisited neighbours.
- There are three main ways in common use:
  - Breadth-first search (BFS): choose coloured node using first-in, first-out;
  - Depth-first search (DFS): choose coloured node using last-in, first-out;
  - Priority-first search (PFS): choose coloured node using a priority function, which may be updated at each step.

Traversing a (di)graph

- How to visit all the nodes of G in an efficient and systematic way? Main idea: start somewhere, and colour in nodes as we visit them. At each step choose a coloured node, and visit one of its unvisited neighbours.
- There are three main ways in common use:
  - Breadth-first search (BFS): choose coloured node using first-in, first-out;
  - Depth-first search (DFS): choose coloured node using last-in, first-out;
  - Priority-first search (PFS): choose coloured node using a priority function, which may be updated at each step.
- Each builds a tree rooted at v, containing all nodes reachable from v. Repeating from different roots yields a collection of disjoint trees containing all nodes (a spanning forest). Given a search forest, we can classify each arc of G as a tree arc, a forward arc, a back arc or a cross arc.

Basic properties of breadth-first search

- Implemented using a queue containing nodes visited but not finished with; takes $\Theta(n+e)$ time using adjacency list, $\Theta(n^2)$ using adjacency matrix.
- The level (distance from root in BFS tree) of each node can be stored. Level of a node equals distance from root in original digraph.
- There are no forward arcs; in a graph every edge is a tree edge or cross edge.
- In a graph, every edge connects two vertices at the same level (hence is a cross edge) or at levels differing by 1 (may be tree or cross edge).
- In the following pseudocode, a FIFO queue Q and arrays $colour, d, pred$ are used.

Breadth-first search pseudocode

algorithm BFSv(node s)
$colour[s] \leftarrow GREY$; $d[s] \leftarrow 0$; $pred[s] \leftarrow NULL$
$level \leftarrow 0$; `insert`(Q, s)
while not `empty`(Q) **do**
    $u \leftarrow$ `next`(Q)
    $level \leftarrow level + 1$
    for each v adjacent to u **do**
        if $colour[v] = WHITE$ **then**
            $colour[v] \leftarrow GREY$; $d[v] \leftarrow level$; $pred[v] \leftarrow u$
            `insert`(Q, v)
    `delete`(Q)
    $colour[u] \leftarrow BLACK$
end

Depth-first search

- Grows search tree by getting as far from the root as possible.
- Implemented recursively (or with stack); takes $\Theta(n+e)$ time using adjacency list, $\Theta(n^2)$ using adjacency matrix.
- Can store the time a node is first seen, and the time its recursive call finishes; these values are related to pre- and post-order traversal of a tree.
- In following pseudocode, arrays `colour`, `seen`, `done` are used. They are initialized so all entries are $WHITE, 0, 0$ respectively.

Depth-first search pseudocode

algorithm DFS(digraph G)
{initialize arrays $colour, pred, seen, done$ of size $|V(G)|$}
$time \leftarrow 0$
for $s \in V(G)$ **do**
    if $colour[s] = WHITE$ **then**
        DFSv(s)
end
algorithm DFSv(node s)
$colour[s] \leftarrow GREY$; $seen[s] \leftarrow time$; $time \leftarrow time + 1$
for each v adjacent to s **do**
    if $colour[v] = WHITE$ **then**
        $pred[v] \leftarrow s$; DFSv(v)
$colour(s) \leftarrow BLACK$; $done[s] \leftarrow time$; $time \leftarrow time + 1$;
end

Basic properties of depth-first search

- Each call to DFSv(v) terminates only when all nodes reachable from v via a path of white nodes have been seen.
- If $seen[v] < seen[w]$ then either
  - w is a descendant of v, $seen[v] < seen[w] < done[w] < done[v]$, or
  - w is not a descendant of v, $seen[v] < done[v] < seen[w] < done[w]$.
- Suppose that (v, w) is an arc. Cases:
  - tree or forward arc: $seen[v] < seen[w] < done[w] < done[v]$;
  - back arc: $seen[w] < seen[v] < done[v] < done[w]$;
  - cross arc: $seen[w] < done[w] < seen[v] < done[v]$.

  Hence on a graph, there are no cross edges.

Nice DFS application: (Strong) components

- Nodes v and w are mutually reachable if there is a path from v to w and a path from w to v. The nodes of a digraph divide up into disjoint subsets of mutually reachable nodes, called strong components. For a graph, we just call it a component.
- (Strong) components are precisely the equivalence classes under the mutual reachability relation.
- (Di)graph is (strongly) connected iff it has only one (strong) component.
- Components of a graph are found easily by BFS or DFS (each tree spans a component). However, this doesn't work well for digraphs (a digraph may have a connected underlying graph yet not be strongly connected). A new idea is needed.

Strong components algorithm

- Run DFS on G, to get depth-first forest F. Create reverse digraph G_r by reversing all arcs. Run DFS on G_r; choose root from unseen nodes finishing latest in F. This gives a forest F_r.
- Suppose v in tree of F_r with root w. Consider the 4 possibilities in F:
  - $seen[w] < seen[v] < done[v] < done[w]$
  - $seen[w] < done[w] < seen[v] < done[v]$
  - $seen[v] < seen[w] < done[w] < done[v]$
  - $seen[v] < done[v] < seen[w] < done[w]$

  By root choice, 2nd and 3rd impossible. By root choice and since w reachable from v in G, 4th impossible. So v is descendant of w in F, and v, w are in the same strong component. The converse is easy.

Cycles

- Suppose that there is a cycle in G and let v be the node in the cycle visited first by DFS. If (u, v) is an arc in the cycle then it must be a back arc (check timestamps).
- Conversely if there is a back arc, we must have a cycle. So a digraph is acyclic iff there are no back arcs from DFS.
- An acyclic digraph is called a DAG (directed acyclic graph). An acyclic graph is a forest.
- Cycles can also be easily detected in a graph using BFS. Finding a cycle of minimum length in a graph is also not difficult using BFS.

Topological sorting

- Try to draw digraph in a line so all arcs go in one direction. Possible if and only if digraph is a DAG.
- Main application: scheduling events (putting on clothes, university prerequisites, etc).
- List of finishing times for depth-first search, in reverse order, solves the problem (since there are no back arcs, each node finishes before anything pointing to it).
- Another solution: zero in-degree sorting. Find node of indegree zero, delete it and repeat until all nodes listed. Less efficient(?)

Weighted (di)graphs

- Also called "networks". Very common in applications. Optimization problems on networks are important in operations research.
- Each arc carries a real number "weight", usually positive, can be $+\infty$. Weight typically represents cost, distance, time.
- Representation: weighted adjacency matrix or double adjacency list.
- Standard problems concern finding a minimum or maximum weight path between given nodes (covered here), spanning tree (here and CS 225), cycle or tour (e.g TSP), matching, flow, etc.

Single-source shortest path problem

- Given an originating node s, find shortest (minimum weight) path to each other node. Write $dist(s, v)$ for this minimum weight.

Single-source shortest path problem

- Given an originating node s, find shortest (minimum weight) path to each other node. Write $dist(s, v)$ for this minimum weight.
- If all weights are equal then BFS works, but it fails in general.

Single-source shortest path problem

- Given an originating node s, find shortest (minimum weight) path to each other node. Write $dist(s, v)$ for this minimum weight.
- If all weights are equal then BFS works, but it fails in general.
- We present two algorithms: the first is faster but fails when weights can be negative; the second is slower but always works.

Single-source shortest path problem

- Given an originating node s, find shortest (minimum weight) path to each other node. Write $dist(s, v)$ for this minimum weight.
- If all weights are equal then BFS works, but it fails in general.
- We present two algorithms: the first is faster but fails when weights can be negative; the second is slower but always works.
- Of course no algorithm can work if there exists a cycle of negative total weight, since there is no minimum value in that case. A robust algorithm will detect such a cycle if it exists, and give the correct answer when it doesn't.

Dijkstra's algorithm

- E. W. Dijkstra (1930–2002) discovered this in the late 1950's. He was a very famous computer scientist with many strong opinions and interesting quotations — look him up.
- An example of a greedy algorithm; sequence of locally best choices gives globally best solution.
- Maintain a list S of visited nodes (say using a priority queue) and an array of best distances found so far; choose node $u \notin S$ with best distances; update distances in case adding u has created shorter paths.
- With negative weights, doesn't detect or find correct solution.
- Complexity depends on data structures used, especially for priority queue; $O(e + n \log n)$ is possible. For simple matrix implementation we have $\Theta(n^2)$, as good as can be expected.

Dijkstra's algorithm pseudocode

algorithm Dijkstra(weighted digraph (G, c), node v)
for $u \in V(G)$ **do**
    $d[u] \leftarrow \infty$
$d[v] \leftarrow 0$; $S \leftarrow \emptyset$
while $S \neq V(G)$ **do**
    find $u \in V(G) \setminus S$ so that $d[u]$ is minimum; $S \leftarrow S \cup \{u\}$
    for $x \in V(G) \setminus S$ **do**
        $d[x] \leftarrow \min\{d[x], d[u] + c[u, x]\}$

Claim: at the top of the `while` loop, (P1) if $w \in S$, $d[w]$ equals the optimal path length, whereas (P2) if w is adjacent to S, $d[w]$ holds the best value achievable using only nodes seen so far.

Correctness of Dijkstra's algorithm I

- By induction on the `while` loop iteration number m. When $m = 0$, $S_0 = \{v\}$ and clearly claim holds.

Correctness of Dijkstra's algorithm I

- By induction on the `while` loop iteration number m. When $m = 0$, $S_0 = \{v\}$ and clearly claim holds.
- Suppose claim holds for m and let u be the next special node (so $S_{m+1} = S_m \cup \{u\}$).

Correctness of Dijkstra's algorithm I

- By induction on the `while` loop iteration number m. When $m = 0$, $S_0 = \{v\}$ and clearly claim holds.
- Suppose claim holds for m and let u be the next special node (so $S_{m+1} = S_m \cup \{u\}$).
- Let $w \in S_{m+1}$. Note $d_{m+1}[w] = d_m[w]$. If $w \neq u$ then claim holds for $m + 1$, clearly.

Correctness of Dijkstra's algorithm I

- By induction on the `while` loop iteration number m. When $m = 0$, $S_0 = \{v\}$ and clearly claim holds.
- Suppose claim holds for m and let u be the next special node (so $S_{m+1} = S_m \cup \{u\}$).
- Let $w \in S_{m+1}$. Note $d_{m+1}[w] = d_m[w]$. If $w \neq u$ then claim holds for $m + 1$, clearly.
- Now suppose that $w = u$ and there is a path to u shorter than $d_m[u]$. Let y be the first node in this path not in S_m. Then $d_m[y] = dist(s, y)$ by inductive hypothesis. Thus $d_{m+1}[u] = d_m[u] > dist(s, y) + dist(y, u) \geq d_m[y]$. This contradicts the choice of u.

Correctness of Dijkstra's algorithm I

- By induction on the `while` loop iteration number m. When $m = 0$, $S_0 = \{v\}$ and clearly claim holds.
- Suppose claim holds for m and let u be the next special node (so $S_{m+1} = S_m \cup \{u\}$).
- Let $w \in S_{m+1}$. Note $d_{m+1}[w] = d_m[w]$. If $w \neq u$ then claim holds for $m + 1$, clearly.
- Now suppose that $w = u$ and there is a path to u shorter than $d_m[u]$. Let y be the first node in this path not in S_m. Then $d_m[y] = dist(s, y)$ by inductive hypothesis. Thus $d_{m+1}[u] = d_m[u] > dist(s, y) + dist(y, u) \geq d_m[y]$. This contradicts the choice of u.
- This completes the induction step for P1.

Correctness of Dijkstra's algorithm II

- Now let $w \in V(G) \setminus S_{m+1}$ and suppose that there is a path γ to w, using only nodes in S_{m+1}, whose length $|\gamma|$ is less than $d_{m+1}[w]$. By the inductive hypothesis, γ must include u.

Correctness of Dijkstra's algorithm II

- Now let $w \in V(G) \setminus S_{m+1}$ and suppose that there is a path γ to w, using only nodes in S_{m+1}, whose length $|\gamma|$ is less than $d_{m+1}[w]$. By the inductive hypothesis, γ must include u.
- If γ goes straight from S_m to u and then w, then $|\gamma| \geq d_{m+1}[w]$ by update formula in algorithm. So this can't happen.

Correctness of Dijkstra's algorithm II

- Now let $w \in V(G) \setminus S_{m+1}$ and suppose that there is a path γ to w, using only nodes in S_{m+1}, whose length $|\gamma|$ is less than $d_{m+1}[w]$. By the inductive hypothesis, γ must include u.
- If γ goes straight from S_m to u and then w, then $|\gamma| \geq d_{m+1}[w]$ by update formula in algorithm. So this can't happen.
- Otherwise γ goes from S_m to u, back inside S_m and emerges for the last time at some node $x \in S_m$, before going straight to w. By inductive hypothesis (P1), there is some optimal path to x of length $d_m[x]$. Replacing part of γ by this, we obtain a path to w, using only nodes in S_m, of length less than $d_{m+1}[w]$ and hence less than $d_m[w]$, which contradicts the inductive hypothesis.

Correctness of Dijkstra's algorithm II

- Now let $w \in V(G) \setminus S_{m+1}$ and suppose that there is a path γ to w, using only nodes in S_{m+1}, whose length $|\gamma|$ is less than $d_{m+1}[w]$. By the inductive hypothesis, γ must include u.
- If γ goes straight from S_m to u and then w, then $|\gamma| \geq d_{m+1}[w]$ by update formula in algorithm. So this can't happen.
- Otherwise γ goes from S_m to u, back inside S_m and emerges for the last time at some node $x \in S_m$, before going straight to w. By inductive hypothesis (P1), there is some optimal path to x of length $d_m[x]$. Replacing part of γ by this, we obtain a path to w, using only nodes in S_m, of length less than $d_{m+1}[w]$ and hence less than $d_m[w]$, which contradicts the inductive hypothesis.
- Thus no such path exists; this completes inductive step for P2.

Correctness of Dijkstra's algorithm II

- Now let $w \in V(G) \setminus S_{m+1}$ and suppose that there is a path γ to w, using only nodes in S_{m+1}, whose length $|\gamma|$ is less than $d_{m+1}[w]$. By the inductive hypothesis, γ must include u.
- If γ goes straight from S_m to u and then w, then $|\gamma| \geq d_{m+1}[w]$ by update formula in algorithm. So this can't happen.
- Otherwise γ goes from S_m to u, back inside S_m and emerges for the last time at some node $x \in S_m$, before going straight to w. By inductive hypothesis (P1), there is some optimal path to x of length $d_m[x]$. Replacing part of γ by this, we obtain a path to w, using only nodes in S_m, of length less than $d_{m+1}[w]$ and hence less than $d_m[w]$, which contradicts the inductive hypothesis.
- Thus no such path exists; this completes inductive step for P2.
- This completes the proof of correctness.

Bellman-Ford algorithm

algorithm Bellman-Ford(weighted digraph (G, c), node s)
for $u \in V(G)$ **do**
    $dist[u] \leftarrow \infty$
$dist[s] \leftarrow 0$
for i from 0 to $n-1$ **do**
    for $x \in V(G)$ **do**
        for $v \in V(G)$ **do**
            $d[v] \leftarrow \min\{d[v], d[x] + c[x, v]\}$
end

Claim: After m times round the outer **for** loop, $d[v]$ holds the optimal value for all nodes v such that v has a minimum weight path with at most m arcs.

Correctness of Bellman-Ford algorithm

- We prove claim holds for all m with $0 \leq m \leq n-1$. Given this, then the algorithm is correct as long as there are no negative weight cycles, because in that case every minimum weight path has at most $n-1$ arcs.

Correctness of Bellman-Ford algorithm

- We prove claim holds for all m with $0 \leq m \leq n-1$. Given this, then the algorithm is correct as long as there are no negative weight cycles, because in that case every minimum weight path has at most $n-1$ arcs.
- When $m = 0$, claim is true by initialization. Suppose that $0 < m$ and claim is true for values less than m. Let γ be a minimum weight path to v with $m+1$ arcs. Let y be the last node before v and γ_1 the subpath to y. Then γ_1 is an optimal path to y and so $dist[y] = |\gamma_1|$.

Correctness of Bellman-Ford algorithm

- We prove claim holds for all m with $0 \leq m \leq n-1$. Given this, then the algorithm is correct as long as there are no negative weight cycles, because in that case every minimum weight path has at most $n-1$ arcs.
- When $m = 0$, claim is true by initialization. Suppose that $0 < m$ and claim is true for values less than m. Let γ be a minimum weight path to v with $m+1$ arcs. Let y be the last node before v and γ_1 the subpath to y. Then γ_1 is an optimal path to y and so $dist[y] = |\gamma_1|$.
- Thus by the update formula we have $dist[v] \leq dist[y] + c[y,v] = |\gamma_1| + c[y,v] = |\gamma|$. This completes the induction step and hence the proof.

All pairs shortest path problem

- Several algorithms are known; we present one, Floyd's algorithm. Alternative to running Dijkstra from each node.
- Number nodes (say from 0 to $n-1$) and at each step k, maintain matrix of shortest distances from node i to node j not passing through nodes higher than k. Update at each step to see whether node k shortens current best distance.
- Basically a triply nested `for` loop, runs in $\Theta(n^3)$ time. Better than Dijkstra for dense graphs, probably not for sparse ones.
- Based on Warshall's algorithm (just tells whether there is a path from node i to node j, not concerned with length).

Floyd's algorithm

algorithm Floyd(weighted digraph (G, c))
for $x \in V(G)$ **do**
  for $u \in V(G)$ **do**
    for $v \in V(G)$ **do**
      $c[u, v] \leftarrow \min\{c[u, v], c[u, x] + c[x, v]\}$

Claim: At the bottom of the outer `for` loop, the current value of $c[u, v]$ is the minimum length of a path from u to v involving only other nodes that have been seen in the outer `for` loop.

Correctness of Floyd's algorithm

- Let S_m be the set of nodes seen so far; call a path with all intermediate nodes in S an S-path. Claim is true for $m = 0$.

Correctness of Floyd's algorithm

- Let S_m be the set of nodes seen so far; call a path with all intermediate nodes in S an S-path. Claim is true for $m = 0$.
- Suppose claim is true for m and let x be the newest node seen at iteration $m + 1$. Fix u, v and let L be the minimum length of an S_{m+1}-path from u to v. Certainly $L \leq c_{m+1}[u, v]$ by construction. We show that $c_{m+1}[u, v] \leq L$.

Correctness of Floyd's algorithm

- Let S_m be the set of nodes seen so far; call a path with all intermediate nodes in S an S-path. Claim is true for $m = 0$.
- Suppose claim is true for m and let x be the newest node seen at iteration $m + 1$. Fix u, v and let L be the minimum length of an S_{m+1}-path from u to v. Certainly $L \leq c_{m+1}[u, v]$ by construction. We show that $c_{m+1}[u, v] \leq L$.
- Choose an S_{m+1}-path P from u to v of length L. If x is not involved then P is an S_m-path, so by inductive hypothesis $L = |P| \geq c_m[u, v] \geq c_{m+1}[u, v]$.

Correctness of Floyd's algorithm

- Let S_m be the set of nodes seen so far; call a path with all intermediate nodes in S an S-path. Claim is true for $m = 0$.
- Suppose claim is true for m and let x be the newest node seen at iteration $m + 1$. Fix u, v and let L be the minimum length of an S_{m+1}-path from u to v. Certainly $L \leq c_{m+1}[u, v]$ by construction. We show that $c_{m+1}[u, v] \leq L$.
- Choose an S_{m+1}-path P from u to v of length L. If x is not involved then P is an S_m-path, so by inductive hypothesis $L = |P| \geq c_m[u, v] \geq c_{m+1}[u, v]$.
- If x is involved, let P_1, P_2 be the subpaths from u to x and x to v. Then P_1 and P_2 are S_m-paths, so by the inductive hypothesis
  $L = |P| = |P_1| + |P_2| \geq c_m[u, x] + c_m[x, v] \geq c_{m+1}[u, v]$.

Minimum spanning tree problem

- Given a connected weighted graph, find a spanning tree (subgraph containing all vertices that is a tree) of minimum total weight. Many obvious applications.

Minimum spanning tree problem

- Given a connected weighted graph, find a spanning tree (subgraph containing all vertices that is a tree) of minimum total weight. Many obvious applications.
- Two efficient greedy algorithms presented here: Prim's and Kruskal's.

Minimum spanning tree problem

- Given a connected weighted graph, find a spanning tree (subgraph containing all vertices that is a tree) of minimum total weight. Many obvious applications.
- Two efficient greedy algorithms presented here: Prim's and Kruskal's.
- Each selects edges in order of increasing weight, subject to not obviously creating a cycle.

Minimum spanning tree problem

- Given a connected weighted graph, find a spanning tree (subgraph containing all vertices that is a tree) of minimum total weight. Many obvious applications.
- Two efficient greedy algorithms presented here: Prim's and Kruskal's.
- Each selects edges in order of increasing weight, subject to not obviously creating a cycle.
- Prim maintains a tree at each stage that grows to span; Kruskal maintains a forest whose trees coalesce into one spanning tree.

Minimum spanning tree problem

- Given a connected weighted graph, find a spanning tree (subgraph containing all vertices that is a tree) of minimum total weight. Many obvious applications.
- Two efficient greedy algorithms presented here: Prim's and Kruskal's.
- Each selects edges in order of increasing weight, subject to not obviously creating a cycle.
- Prim maintains a tree at each stage that grows to span; Kruskal maintains a forest whose trees coalesce into one spanning tree.
- Prim implementation very similar to Dijkstra, get $O(e + n \log n)$; Kruskal uses disjoint sets ADT and can be implemented to run in time $O(e \log n)$.

Prim's algorithm

algorithm Prim(weighted digraph (G, c), node v)
for $u \in V(G)$ **do**
    $d[u] \leftarrow \infty$
$d(v) \leftarrow 0$
$S \leftarrow \emptyset$
while $S \neq V(G)$ **do**
    find $u \in V(G) \setminus S$ so that $d[u]$ is minimum
    $S \leftarrow S \cup \{u\}$
    for $x \in V(G) \setminus S$ **do**
        $d[x] \leftarrow \min\{d[x], c[u, x]\}$

Very similar to Dijkstra - uses a priority queue to hold elements of d. EXTRACT-MIN, CHANGE-PRIORITY dominate runtime.

Priority-first search

- Build a priority queue containing all nodes, each with an initial priority value.

Priority-first search

- Build a priority queue containing all nodes, each with an initial priority value.
- At each step, choose the grey node with highest priority and choose the white neighbour arbitrarily. Delete the grey node you chose. Then update the priority of the nodes in the priority queue (usually just update the fringe nodes). End when the queue is empty.

Priority-first search

- Build a priority queue containing all nodes, each with an initial priority value.
- At each step, choose the grey node with highest priority and choose the white neighbour arbitrarily. Delete the grey node you chose. Then update the priority of the nodes in the priority queue (usually just update the fringe nodes). End when the queue is empty.
- Dijkstra's and Prim's algorithms use this approach to create a search tree.

Priority-first search

- Build a priority queue containing all nodes, each with an initial priority value.
- At each step, choose the grey node with highest priority and choose the white neighbour arbitrarily. Delete the grey node you chose. Then update the priority of the nodes in the priority queue (usually just update the fringe nodes). End when the queue is empty.
- Dijkstra's and Prim's algorithms use this approach to create a search tree.
- The main operations are extracting the minimum (n times) and changing the priority value (e times). Having a good data structure that supports these efficiently is very important.

Priority-first search

- Build a priority queue containing all nodes, each with an initial priority value.
- At each step, choose the grey node with highest priority and choose the white neighbour arbitrarily. Delete the grey node you chose. Then update the priority of the nodes in the priority queue (usually just update the fringe nodes). End when the queue is empty.
- Dijkstra's and Prim's algorithms use this approach to create a search tree.
- The main operations are extracting the minimum (n times) and changing the priority value (e times). Having a good data structure that supports these efficiently is very important.
- Binary heap: good for extract-min, bad for change-priority. Array: bad for extract-min, good for change-priority. More complicated data structures exist that are good for both.

Kruskal's algorithm

algorithm Kruskal(weighted digraph (G, c))
$T \leftarrow \emptyset$
sort $E(G)$ by increasing order of cost
for $e = \{u, v\} \in E(G)$ **do**
    if u and v are not in the same tree **then**
        $T \leftarrow T \cup \{e\}$
        merge the trees of u and v

Keep track of the trees using disjoint sets ADT, with standard operations FIND and UNION. They can be implemented efficiently so that the main time taken is the sorting step.

Proof that Prim, Kruskal work

Call a set of edges promising if it can extend to give a MST.
Claim: let $B \subset V$ and let $T \subseteq E$ be promising, and no edge in T leaves B. Let e be minimum weight edge leaving B. Then $T \cup \{e\}$ is promising.
Assuming claim, proof follows by taking $B =$ nodes of component including endpoint of next edge e (Kruskal) or $B =$ nodes of current tree (Prim).
Proof of claim: let U be MST containing T. If $e \in U$, done. Else there is another edge e' leaving B (to close the cycle). Then removing e' and adding e to U gives MST containing T.

Other graph problems

- Hamilton cycle: traverse graph, visiting each vertex exactly once. Example: knight's tour. Travelling sales rep problem is a generalization.
- Euler cycle: traverse graph, visiting each edge exactly once. Chinese postman problem is a generalization.
- Vertex colouring: colour each vertex one colour, so neighbours have different colours.
- Vertex cover: choose $S \subset V$ so each edge is incident to some $s \in S$.
- An Euler cycle can be found (if it exists) in linear time. but no fast algorithm is known for finding a Hamiltonian cycle in general, (given that it exists). However, a knight's tour can be found on an $n \times n$ chessboard in linear time ($O(n^2)$).

Hard problems

- "Easy" problems are solvable in polynomial time; let P denote the class of these problems. Includes all coursebook problems, plus planarity testing, Chinese postman and a few more.
- Let NP denote the class of problems for which a guess can be checked in polynomial time. Obviously $P \subseteq NP$ and NP "must" be much bigger. Main open problem of theoretical computer science: is $P = NP$? No problem has been proved to be in $NP \setminus P$, but there are many that are in NP and not known to be in P.
- A problem is NP-complete if it is in NP and is "as hard as" any other problem in NP. Example: Hamiltonian cycle. Usually we must solve NP-complete problems via exhaustive search of exponentially many possibilities. This leads to backtracking and branch-and-bound methods covered in CS 320.