Weighted Graph Algorithms

- Beyond DFS/BFS exists an alternate universe of algorithms for edge-weighted graphs.

Weighted Graph Algorithms

- Beyond DFS/BFS exists an alternate universe of algorithms for edge-weighted graphs.
- Our adjacency list representation quietly supported these graphs. (just add a weight field to each node).

Minimum Spanning Trees: Definitions

- A tree is a connected graph with no cycles.

Minimum Spanning Trees: Definitions

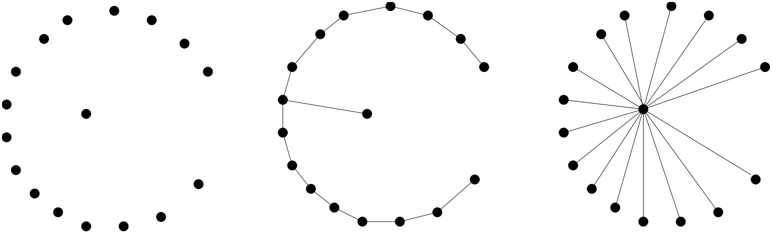
- A tree is a connected graph with no cycles.
- A spanning tree is a subgraph of G which has the same set of vertices of G and is a tree.

Minimum Spanning Trees: Definitions

- A tree is a connected graph with no cycles.
- A spanning tree is a subgraph of G which has the same set of vertices of G and is a tree.
- A minimum spanning tree of a weighted graph G is the spanning tree of G whose edges sum to minimum weight.

Minimum Spanning Trees

There can be more than one minimum spanning tree in a graph $\rightarrow$ consider a graph with identical weight edges.



(a) (b) (c)

Why Minimum Spanning Trees?

The MST problem has a long history – the first algorithm dates back at least to 1926!

Why Minimum Spanning Trees?

The MST problem has a long history – the first algorithm dates back at least to 1926!
MST is taught in algorithm courses because

1. it arises in many applications,
2. it is a problem for which greedy algorithms give the optimal answer
3. Clever data structures are necessary to make it work well.

Applications of MSTs

- MSTs are useful in constructing networks, by describing the way to connect a set of sites using the smallest total amount of wire.

Applications of MSTs

- MSTs are useful in constructing networks, by describing the way to connect a set of sites using the smallest total amount of wire.
- MSTs provide a reasonable way for *clustering* points in space into natural groups.

Recall: Algorithmic Strategies

1. Divide and Conquer.
2. Greedy.
3. Exhaustive Search.
4. Dynamic Programming.
5. Backtracking.

Greedy Algorithms

In greedy algorithms, we make the decision of what next to do by selecting the best local option from all available choices – without regard to the global structure.

Two Greedy MST Algorithms

1. Prim's Algorithm
2. Kruskal's Algorithm

Prim's Algorithm

- If G is connected, every vertex will appear in the minimum spanning tree.

Prim's Algorithm

- If G is connected, every vertex will appear in the minimum spanning tree.
- Prim's algorithm starts from one vertex and grows the rest of the tree an edge at a time.

Prim's Algorithm

- If G is connected, every vertex will appear in the minimum spanning tree.
- Prim's algorithm starts from one vertex and grows the rest of the tree an edge at a time.
- Q: As a greedy algorithm, which edge should we pick?

Prim's Algorithm

- If G is connected, every vertex will appear in the minimum spanning tree.
- Prim's algorithm starts from one vertex and grows the rest of the tree an edge at a time.
- Q: As a greedy algorithm, which edge should we pick?
- A: The cheapest edge with which can grow the tree by one vertex without creating a cycle.

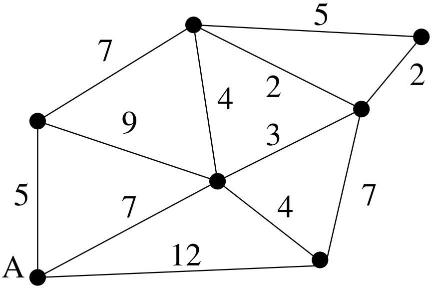
Prim's Algorithm (Pseudocode)

Prim-MST(G)

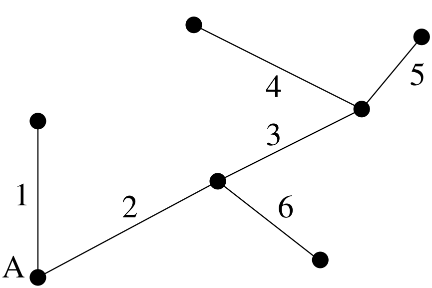
    Select an arbitrary vertex s to start the tree from.
    While (there are still non-tree vertices)
        Select min wt edge between tree & non-tree verte
        Add selected edge and vertex to the tree T_{prim}.

This creates a spanning tree, since no cycle can be introduced, but is it minimum?

Prim's Algorithm in Action



G



Prim(G,A)

Why is Prim Correct?

- We use a proof by contradiction: suppose Prim's algorithm does not always give the minimum cost spanning tree on some graph.

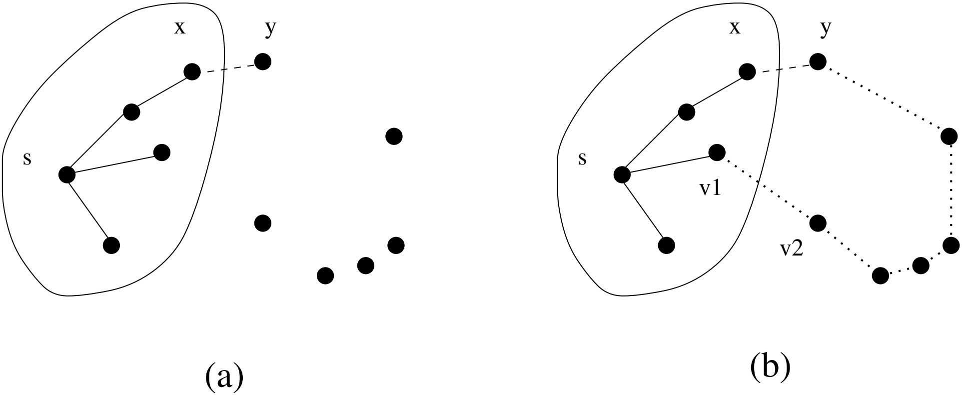
Why is Prim Correct?

- We use a proof by contradiction: suppose Prim's algorithm does not always give the minimum cost spanning tree on some graph.
- If so, there is a graph on which it fails.

Why is Prim Correct?

- We use a proof by contradiction: suppose Prim's algorithm does not always give the minimum cost spanning tree on some graph.
- If so, there is a graph on which it fails.
- And if so, there must be a first edge (x, y) Prim adds such that the partial tree V' cannot be extended into a minimum spanning tree.

Diagram: redraw on board



Proof (need to look at diagram)

- But if (x, y) is not in $MST(G)$, then there must be a path in $MST(G)$ from x to y since the tree is connected.

Proof (need to look at diagram)

- But if (x, y) is not in $MST(G)$, then there must be a path in $MST(G)$ from x to y since the tree is connected.
- Let (v_1, v_2) be the first edge on this path with one vertex in V'

Proof (need to look at diagram)

- But if (x, y) is not in $MST(G)$, then there must be a path in $MST(G)$ from x to y since the tree is connected.
- Let (v_1, v_2) be the first edge on this path with one vertex in V'
- Replacing it with (x, y) we get a spanning tree. with smaller weight, since $W(v_1, v_2) > W(x, y)$.

Proof (need to look at diagram)

- But if (x, y) is not in $MST(G)$, then there must be a path in $MST(G)$ from x to y since the tree is connected.
- Let (v_1, v_2) be the first edge on this path with one vertex in V'
- Replacing it with (x, y) we get a spanning tree. with smaller weight, since $W(v_1, v_2) > W(x, y)$.
- Thus $MST(G)$ is not a minimum spanning tree!!

Prim's Algorithm is correct!

We cannot go wrong with the greedy strategy the way we could with the traveling salesman problem.

How Fast is Prim's Algorithm?

Depends on data structs used to determine smallest weighted edge from tree to non-tree vertex.

- Simple: use BFS/DFS to look at all edges that connect tree vertex to non-tree vertex in $O(m)$. But do this n times, so total $O(nm)$.

How Fast is Prim's Algorithm?

Depends on data structs used to determine smallest weighted edge from tree to non-tree vertex.

- Simple: use BFS/DFS to look at all edges that connect tree vertex to non-tree vertex in $O(m)$. But do this n times, so total $O(nm)$.
- Smart: takes $O(n)$ time to figure out smallest weighted edge, $O(n^2)$ time total.

How Fast is Prim's Algorithm?

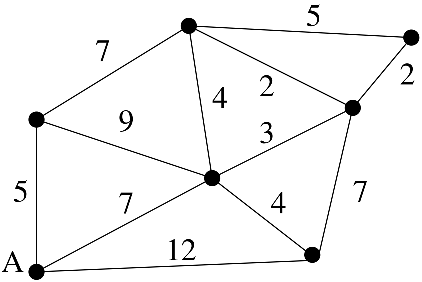
Depends on data structs used to determine smallest weighted edge from tree to non-tree vertex.

- Simple: use BFS/DFS to look at all edges that connect tree vertex to non-tree vertex in $O(m)$. But do this n times, so total $O(nm)$.
- Smart: takes $O(n)$ time to figure out smallest weighted edge, $O(n^2)$ time total.
- Smartest: using Fibonacci Heaps (sophisticated priority queue), take $O(m + n \log n)$.

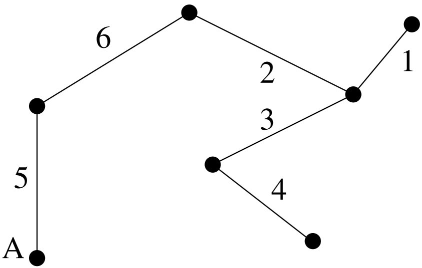
Kruskal's Algorithm

Kruskal's algorithm is also greedy. It repeatedly adds the smallest edge to the spanning tree that does not create a cycle.

Kruskal's Algorithm in Action



G



Kruskal(G)

Proof of Correctness

Again, use proof by contradiction. Similar to proof of Prims, so omit!

How fast is Kruskal's algorithm?

What is the simplest implementation?

- Sort the m edges in $O(m \lg m)$ time.

How fast is Kruskal's algorithm?

What is the simplest implementation?

- Sort the m edges in $O(m \lg m)$ time.
- For each edge in order, test whether it creates a cycle in the forest we have thus far built – if "yes" discard, else add to forest.

How fast is Kruskal's algorithm?

What is the simplest implementation?

- Sort the m edges in $O(m \lg m)$ time.
- For each edge in order, test whether it creates a cycle in the forest we have thus far built – if "yes" discard, else add to forest.
- Testing for a cycle can be done by BFS/DFS in $O(n)$ time (since the tree has at most n edges).

How fast is Kruskal's algorithm?

What is the simplest implementation?

- Sort the m edges in $O(m \lg m)$ time.
- For each edge in order, test whether it creates a cycle in the forest we have thus far built – if "yes" discard, else add to forest.
- Testing for a cycle can be done by BFS/DFS in $O(n)$ time (since the tree has at most n edges).

How fast is Kruskal's algorithm?

What is the simplest implementation?

- Sort the m edges in $O(m \lg m)$ time.
- For each edge in order, test whether it creates a cycle in the forest we have thus far built – if "yes" discard, else add to forest.
- Testing for a cycle can be done by BFS/DFS in $O(n)$ time (since the tree has at most n edges).

But can we do better?

Another Strategy for Testing for Cycles

- Kruskal's algorithm builds up connected components.

Another Strategy for Testing for Cycles

- Kruskal's algorithm builds up connected components.
- If the two end-vertices of an edge that is being considered are in the same connected component, then it creates a cycle.

Another Strategy for Testing for Cycles

- Kruskal's algorithm builds up connected components.
- If the two end-vertices of an edge that is being considered are in the same connected component, then it creates a cycle.
- Thus if we can maintain which vertices are in which component fast, we do not have to test for cycles!

Component Testing

- *Same component*(v, w) – Do vertices v and w lie in the same connected component of the current graph?

Component Testing

- *Same component*(v, w) – Do vertices v and w lie in the same connected component of the current graph?
- *Merge components*(C_1, C_2) – Merge the given pair of connected components into one component.

Fast Kruskal Implementation

```
Put the edges in a min-heap
count = 0
while (count < n − 1) do
        get min-weight edge (v, w) from heap
        if (component (v) ≠ component(w))
                add to T
                count++
                Merge(component (v), component(w))
```

If comp ops take $O(\log n)$, Kruskal is $O(m \log m)$!

Fast Kruskal Implementation

Put the edges in a min-heap
$count = 0$
while $(count < n - 1)$ do
    get min-weight edge (v, w) from heap
    if (component (v) $\neq$ component(w))
        add to T
        count++
        Merge(component (v), component(w))

If comp ops take $O(\log n)$, Kruskal is $O(m \log m)$!
Q: Is $O(m \log n)$ better than $O(m \log m)$?

Specialized Set Data Structure

- Need to (i) test if two elements are in the same set and (ii) merge two sets together.

Specialized Set Data Structure

- Need to (i) test if two elements are in the same set and (ii) merge two sets together.
- Represent each set as a tree, with pointers from a node to its parent.

Specialized Set Data Structure

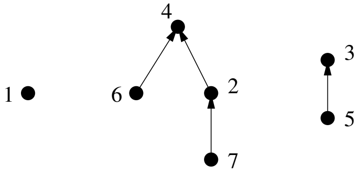
- Need to (i) test if two elements are in the same set and (ii) merge two sets together.
- Represent each set as a tree, with pointers from a node to its parent.
- Each element is contained in a node, and the name of the set is the key at the root.

Specialized Set Data Structure

- Need to (i) test if two elements are in the same set and (ii) merge two sets together.
- Represent each set as a tree, with pointers from a node to its parent.
- Each element is contained in a node, and the name of the set is the key at the root.

Specialized Set Data Structure

- Need to (i) test if two elements are in the same set and (ii) merge two sets together.
- Represent each set as a tree, with pointers from a node to its parent.
- Each element is contained in a node, and the name of the set is the key at the root.



(l)

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

(r)

Union & Find operations

- *Find(i)* – Return the label of the root of tree containing element i, by walking up the parent pointers until you can't go up any more.

Union & Find operations

- *Find(i)* – Return the label of the root of tree containing element i, by walking up the parent pointers until you can't go up any more.
- *Union(i,j)* – Make the root of one of the trees the parent of the root of the other tree so $Find(i)$ now equals $Find(j)$.

Component Testing

Are i and j in the same component?

$t = \text{Find}(i)$

$u = \text{Find}(j)$

Return (Is $t = u$?)

Component Testing

<u>Are i and j in the same component?</u>

$t = \text{Find}(i)$
$u = \text{Find}(j)$
Return (Is $t = u$?)

Components merged by using Union operation

Union-Find Trees

- We are interested in minimizing the time it takes to execute *any* sequence of unions and finds.

Union-Find Trees

- We are interested in minimizing the time it takes to execute *any* sequence of unions and finds.
- In the worst case, heights of union-find trees can be $O(n)$.

Who's The Daddy?

When we union, simply make the tree with the smaller height the child of the other one.

What does this buy us?

- The height of the final tree will increase only if both subtrees are of equal height!

What does this buy us?

- The height of the final tree will increase only if both subtrees are of equal height!
- Analysis shows that Unions and Finds take $O(\log n)$, so a sequence of n union-finds takes $O(n \log n)$.

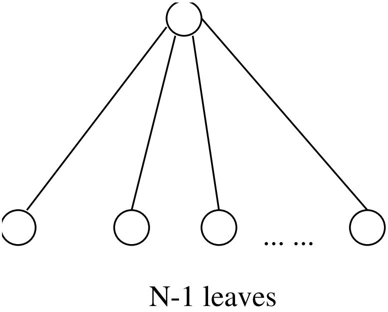
Can we do better?

$O(n \log n)$ is good enough for Kruskal's algorithm, but can we do better?

Can we do better?

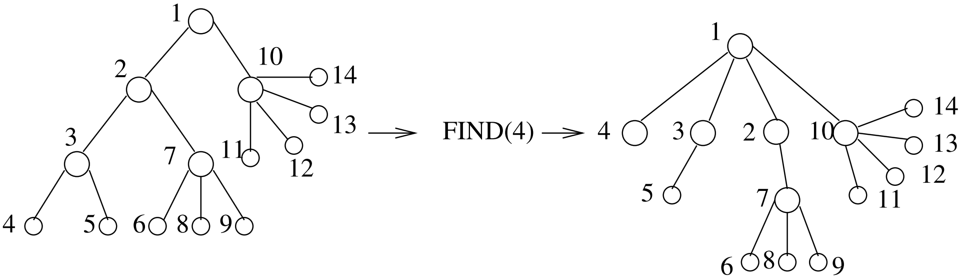
$O(n \log n)$ is good enough for Kruskal's algorithm, but can we do better?

The ideal *Union-Find* tree has depth 1:



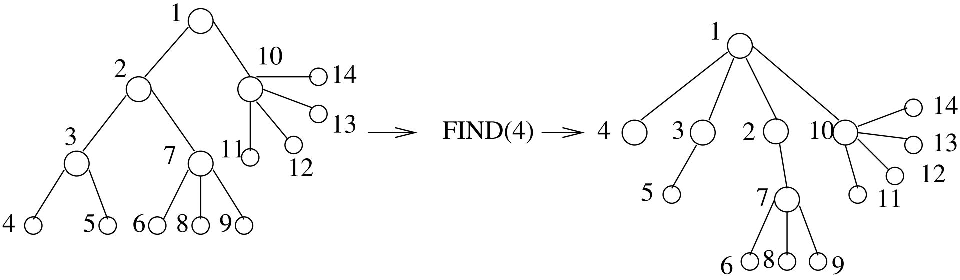
Path Compression

On a find, if we are going to walk along a path anyway, why not change ptrs to point to the root?



Path Compression

On a find, if we are going to walk along a path anyway, why not change ptrs to point to the root?



Better than $O(n \log n)$ for n union-finds.

Inverse Ackermann function

Do we get $O(n)$ for a sequence of n union-finds?

Inverse Ackermann function

Do we get $O(n)$ for a sequence of n union-finds?

Not quite . . . difficult analysis shows that it takes $O(n\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackerman function and α(number of atoms in the universe)$=5$.

Ackermann Worksheet

Shortest Paths

Finding the shortest path between two nodes in a graph arises in many different applications:

- Transportation problems – finding the cheapest way to travel between two locations.
- Motion planning – what is the most natural way for a cartoon character to move about a simulated environment.
- Communications problems – how long will it take for a message to get from one place to another?

Shortest Paths: Unweighted Graphs

In an unweighted graph, the cost of a path is just the number of edges on the shortest path, which can be found in $O(n + m)$ time via breadth-first search.

Shortest Paths: Weighted Graphs

- The length of a path between two vertices is the sum of the weights of the edges on a path.

Shortest Paths: Weighted Graphs

- The length of a path between two vertices is the sum of the weights of the edges on a path.
- BFS will not work on weighted graphs because sometimes visiting more edges can lead to shorter distance, ie. $1 + 1 + 1 + 1 + 1 + 1 + 1 < 10$.

Shortest Paths: Weighted Graphs

- The length of a path between two vertices is the sum of the weights of the edges on a path.
- BFS will not work on weighted graphs because sometimes visiting more edges can lead to shorter distance, ie. $1 + 1 + 1 + 1 + 1 + 1 + 1 < 10$.
- There can be an exponential number of shortest paths between two nodes – so we cannot report all shortest paths efficiently.

Negative Edge Weights

- The problem with negative cost cycles.

Negative Edge Weights

- The problem with negative cost cycles.
- We will assume that all edge weights are positive. Other algorithms deal correctly with negative cost edges.

Negative Edge Weights

- The problem with negative cost cycles.
- We will assume that all edge weights are positive. Other algorithms deal correctly with negative cost edges.
- MST algorithms are unaffected by negative cost edges.

Edsger Wybe Dijkstra: 1930-2002

- "Computer Science is no more about computers than astronomy is about telescopes."

Edsger Wybe Dijkstra: 1930-2002

- “Computer Science is no more about computers than astronomy is about telescopes.”
- “The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague” (from 1972 Turing Award Lecture)

Dijkstra's Algorithm

- The principle behind Dijkstra's algorithm is that if $s, \ldots, x, \ldots, t$ is the shortest path from s to t, then $s, \ldots, x$ had better be the shortest path from s to x.

Dijkstra's Algorithm

- The principle behind Dijkstra's algorithm is that if $s, \ldots, x, \ldots, t$ is the shortest path from s to t, then $s, \ldots, x$ had better be the shortest path from s to x.
- This suggests a dynamic programming-like strategy, where we store the distance from s to all nearby nodes, and use them to find the shortest path to more distant nodes.

Dijkstra Basics

Output

- Shortest Path Tree rooted at s (similar to BF tree)
- parent$[v]$
- distance (from s) $d[v]$

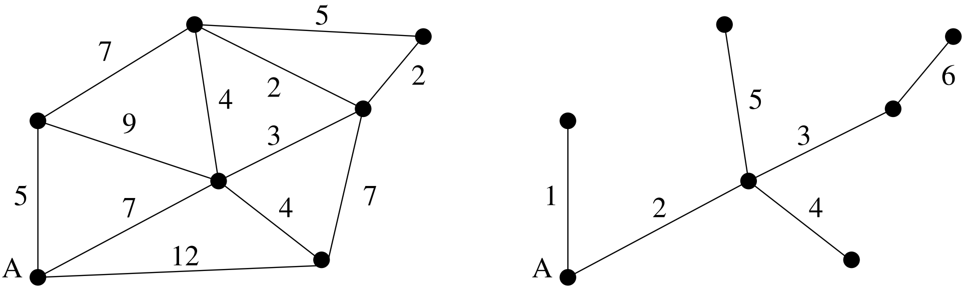
Initialization

- set each $d[v]$ to ∞ (except $d[s] = 0$)
- set each parent$[v]$ to nil.

DIJKSTRA(G, s)

```
S = { }
Q = V[G] // Q is a priority queue on d
while (Q not empty)
    u = Extract-Min(Q)
    S = S ∪ {u}
    for each vertex v adjacent to u
        // "Relaxation" step
        if d[v] > d[u] + w(u, v)
            d[v] = d[u] + w(u, v) //DecreaseKey op
            parent[v] = u
```

Dijkstra Example



G

Dijkstra(G,A)

Proof that Dijkstra is correct

Proof by contradiction. Omit.

Analysis

- Initializing Q takes $O(n)$ (remember Build_Heap).
- $O(n)$ iterations of while loop.
- So, $O(n)$ Extract-Min ops on Q
- Total $O(m)$ iterations of inner for loop
- At most $O(m)$ relaxations
- Need to implement Decrease-Key on Q.

Effect of Data Structures

1. Min-heap
   - EXTRACT-MIN is $O(\log n)$
   - DECREASE-KEY is $O(\log n)$
   - $\Rightarrow (n+m)\log n \Rightarrow m \log n$

Effect of Data Structures

1. Min-heap
   - EXTRACT-MIN is $O(\log n)$
   - DECREASE-KEY is $O(\log n)$
   - $\Rightarrow (n+m)\log n \Rightarrow m\log n$
2. Fibonacci-Heap
   - EXTRACT-MIN is $O(\log n)$
   - DECREASE-KEY is $O(1)$ amortized
   - $\Rightarrow n\log n + m$

Closing Thoughts

Similar to Prim's MST algorithm.
Dijkstra will not work if edges have negative weights.

Dijkstra Quotes (2)

- "Program testing can best show the presence of errors but never their absence"

Dijkstra Quotes (2)

- “Program testing can best show the presence of errors but never their absence”
- “If you don’t know what your program is supposed to do, you’d better not start writing it.”

Dijkstra Quotes (2)

- “Program testing can best show the presence of errors but never their absence”
- “If you don’t know what your program is supposed to do, you’d better not start writing it.”
- “The price of reliability is the pursuit of the utmost simplicity.”

Review

- Notice that finding the shortest path between a pair of vertices (s, t) in worst case requires first finding the shortest path from s to all other vertices in the graph.

Review

- Notice that finding the shortest path between a pair of vertices (s, t) in worst case requires first finding the shortest path from s to all other vertices in the graph.
- Dijkstra animation.

All-Pairs Shortest Paths

- Many applications, such as finding the diameter of a graph, require finding the shortest path between all pairs of vertices.

All-Pairs Shortest Paths

- Many applications, such as finding the diameter of a graph, require finding the shortest path between all pairs of vertices.
- We can run Dijkstra's algorithm n times (once from each possible start vertex) to solve all-pairs shortest path problem in $O(n(n \log n + m))$.

All-Pairs Shortest Paths

- Many applications, such as finding the diameter of a graph, require finding the shortest path between all pairs of vertices.
- We can run Dijkstra's algorithm n times (once from each possible start vertex) to solve all-pairs shortest path problem in $O(n(n \log n + m))$.
- Can we do better?

All-Pairs Shortest Paths

- Many applications, such as finding the diameter of a graph, require finding the shortest path between all pairs of vertices.
- We can run Dijkstra's algorithm n times (once from each possible start vertex) to solve all-pairs shortest path problem in $O(n(n \log n + m))$.
- Can we do better?
- Not today :-).

The Floyd-Warshall Algorithm

- Develop recurrence that yields a dyn prog formulation. Number the vertices from 1 to n.

The Floyd-Warshall Algorithm

- Develop recurrence that yields a dyn prog formulation. Number the vertices from 1 to n.
- *Let $d[i,j]^k$ be the shortest path from i to j using only vertices from $1, 2, \ldots, k$ as possible intermediate vertices.*

The Floyd-Warshall Algorithm

- Develop recurrence that yields a dyn prog formulation. Number the vertices from 1 to n.
- *Let* $d[i,j]^k$ *be the shortest path from* i *to* j *using only vertices from* $1, 2, ..., k$ *as possible intermediate vertices.*
- Q: What is $d[i,j]^0$? $d[i,j]^n$?

The Floyd-Warshall Algorithm

- Develop recurrence that yields a dyn prog formulation. Number the vertices from 1 to n.
- *Let* $d[i,j]^k$ *be the shortest path from* i *to* j *using only vertices from* $1, 2, ..., k$ *as possible intermediate vertices.*
- Q: What is $d[i,j]^0$? $d[i,j]^n$?
- A: With no intermediate vertices, any path consists of one edge, so $d[i,j]^0 = w[i,j]$.

Recurrence Relation

In general, adding a new vertex k helps if and only if a path goes through it, so

$$\begin{aligned} d[i,j]^k &= w[i,j] \text{ if } k = 0 \\ &= \min(d[i,j]^{k-1}, d[i,k]^{k-1} + d[k,j]^{k-1}) \text{ if } k \geq 1 \end{aligned}$$

Implementation

$d^o = w$
for $k = 1$ to n
 for $i = 1$ to n
  for $j = 1$ to n
   $d[i,j]^k = \min(d[i,j]^{k-1}, d[i,k]^{k-1} + d[k,j]^{k-1})$

This runs in $\Theta(n^3)$ time, which is worse than n calls to Dijkstra. However, the loops are so tight and it is so short and simple that it is practical.

Network Flows

Used to model

- Flow of current in an electrical circuit
- liquid flowing through pipes
- traffic flow thru a network of roads
- packets in a computer network

Flow Network: definition

- Directed graph G with two special vertices: source s and sink t

Flow Network: definition

- Directed graph G with two special vertices: source s and sink t
- Each edge (u, v) has a capacity $c(u, v) > 0$

Flow Network: definition

- Directed graph G with two special vertices: source s and sink t
- Each edge (u, v) has a capacity $c(u, v) > 0$
- If (u, v) is not an edge, $c(u, v) = 0$ by convention

Flow Network: definition

- Directed graph G with two special vertices: source s and sink t
- Each edge (u, v) has a capacity $c(u, v) > 0$
- If (u, v) is not an edge, $c(u, v) = 0$ by convention
- Draw figure on board

Flow function: definitions, rules

Flow function $f : V \times V \longrightarrow R$

Flow function: definitions, rules

Flow function $f : V \times V \longrightarrow R$

f satisfies three (intuitive) properties

1. Capacity constraint: $f(u, v) \leq c(u, v)$

Flow function: definitions, rules

Flow function $f : V \times V \longrightarrow R$

f satisfies three (intuitive) properties

1. Capacity constraint: $f(u, v) \leq c(u, v)$
2. Skew Symmetry: $f(u, v) = -f(v, u)$

Flow function: definitions, rules

Flow function $f : V \times V \longrightarrow R$

f satisfies three (intuitive) properties

1. Capacity constraint: $f(u, v) \leq c(u, v)$
2. Skew Symmetry: $f(u, v) = -f(v, u)$
3. Flow conservation: for any vertex u (other than s or t) $\sum_v f(u, v) = 0$

The Max Flow Problem

- Value of a flow f (denoted by $|f|$) is defined as $\sum_v f(s, v)$.

The Max Flow Problem

- Value of a flow f (denoted by $|f|$) is defined as $\sum_v f(s, v)$.
- i.e., the total flow leaving the source.

The Max Flow Problem

- Value of a flow f (denoted by $|f|$) is defined as $\sum_v f(s,v)$.
- i.e., the total flow leaving the source.
- Max Flow Problem: assign legal flow values to each edge so that the value of the flow is maximized.

The Max Flow Problem

- Value of a flow f (denoted by $|f|$) is defined as $\sum_v f(s,v)$.
- i.e., the total flow leaving the source.
- Max Flow Problem: assign legal flow values to each edge so that the value of the flow is maximized.
- Update figure.

Ford-Fulkerson Algorithm

Initialize f to 0 for each edge.
while there exists an augmenting path p
    augment flow along p

Augmenting Paths

- Augmenting path = sequence of edges from s to t along which flow can be increased.

Augmenting Paths

- Augmenting path = sequence of edges from s to t along which flow can be increased.
- Example on board.

Augmenting Paths

- Augmenting path = sequence of edges from s to t along which flow can be increased.
- Example on board.
- Sometimes an augmenting path will traverse edges in reverse direction!

Augmenting Paths

- Augmenting path = sequence of edges from s to t along which flow can be increased.
- Example on board.
- Sometimes an augmenting path will traverse edges in reverse direction!
- BFS/DFS on flow graph will not find an augmenting path.

Residual Graph/Network

Define residual capacity of edge (u, v) as
$r(u, v) = c(u, v) - f(u, v)$

Residual Graph/Network

Define residual capacity of edge (u, v) as
$r(u, v) = c(u, v) - f(u, v)$
Set up residual graph as follows:

- Same vertices as in original graph
- Include edge (u, v) if $r(u, v) > 0$

Residual Graph/Network

Define residual capacity of edge (u, v) as
$r(u, v) = c(u, v) - f(u, v)$
Set up residual graph as follows:

- Same vertices as in original graph
- Include edge (u, v) if $r(u, v) > 0$

NOW use BFS/DFS on residual graph to find augmenting path!

Cuts

- Cut(S, T) of a flow network is a partition of V into S and $T = V - S$ (where s is in S, t is in T).

Cuts

- Cut(S, T) of a flow network is a partition of V into S and $T = V - S$ (where s is in S, t is in T).
- Define net flow $f(S, T)$ across a cut

Cuts

- Cut(S, T) of a flow network is a partition of V into S and $T = V - S$ (where s is in S, t is in T).
- Define net flow $f(S, T)$ across a cut
- Define capacity of cut $c(S, T)$

Cuts

- Cut(S, T) of a flow network is a partition of V into S and $T = V - S$ (where s is in S, t is in T).
- Define net flow $f(S, T)$ across a cut
- Define capacity of cut $c(S, T)$
- Max flow cannot exceed capacity of the cut

Max-Flow Min-Cut Theorem

The following statements are equivalent

1. f is a maximum flow in G
2. The residual network (corrresp to f) contains no augmenting paths
3. $|f| = c(S, T)$ for some cut (S, T) of G

Complexity

Product of

- $\#$ of flow augmentations performed by FF.
  - Each augmentation increases flow by at least 1.
  - So, at most $|f|$ augmentations.
- Run time of one flow augmentation is $O(n+m) = O(m)$.

Edmonds-Karp Algorithm

- Identical to Ford-Fulkerson except it requires augmenting path to be computed with a BF traversal.

Edmonds-Karp Algorithm

- Identical to Ford-Fulkerson except it requires augmenting path to be computed with a BF traversal.
- So, augmenting path is a shortest path from s to t.

Edmonds-Karp Algorithm

- Identical to Ford-Fulkerson except it requires augmenting path to be computed with a BF traversal.
- So, augmenting path is a shortest path from s to t.
- Theorem: total number of flow augmentations performed by the E-K algorithm is $O(nm)$.

Edmonds-Karp Algorithm

- Identical to Ford-Fulkerson except it requires augmenting path to be computed with a BF traversal.
- So, augmenting path is a shortest path from s to t.
- Theorem: total number of flow augmentations performed by the E-K algorithm is $O(nm)$.
- Overall run time is $O(nm^2)$

Maximum Matching in Bipartite Graphs

Bipartite: Connected undirected graph G where

1. Vertices of G can be split into two sets X and Y so that ...
2. Every edge of G has one end point in X and the other in Y.

Maximum Matching in Bipartite Graphs

Bipartite: Connected undirected graph G where

1. Vertices of G can be split into two sets X and Y so that ...
2. Every edge of G has one end point in X and the other in Y.

Matching M in G is a set of edges that have no end points in common.

Maximum Matching in Bipartite Graphs

Bipartite: Connected undirected graph G where

1. Vertices of G can be split into two sets X and Y so that ...
2. Every edge of G has one end point in X and the other in Y.

Matching M in G is a set of edges that have no end points in common.

Maximum Bipartite Matching Problem: Find a matching with the greatest number of edges.

Application

- X is a set of college courses.
- Y is a set of classrooms.
- An edge joins x in X and y in Y if course x can be taught in classroom y (based on audio-visual needs, enrollment, etc.).

Reduction to the Maximum Flow Problem

1. Set up flow network H corresponding to bipartite graph G.
2. Run max-flow on H.
3. Use results of max-flow on H to construct matching in G.

Set up Flow Network H from G

- H contains all the vertices of G plus a new source vertex s and a new sink vertex t.
- Add every edge of G to H but direct it so that it goes from the vertex in X to the vertex in Y.
- Add a directed edge from s to each vertex in X.
- Add a directed edge from each vertex in Y to t.
- Assign a capacity of 1 to each edge.

Solve the maximum matching problem.

- Run the max-flow algorithm on H.
- The flow in each edge is either 0 or 1 (why?).
- For each vertex in X, there is at most one outgoing edge with a flow of 1 (why?).
- Similarly, for each vertex in Y, there is at most one incoming edge with a flow of 1.
- Matching = set of all (x, y)s with a flow of 1 from x to y.
- Max flow implies max matching!

Complexity Analysis

- Construct H from G: $O(n+m)$.
- Run FF algorithm: $(|f| \times m)$.
- But $|f| = |M| \leq n/2$.
- So $O(nm)$.

Yahoo Worksheet