COMP2521 19T0
Week 5, Tuesday: Graphic Content (IV)!

Jashank Jeremy

jashank.jeremy@unsw.edu.au

weighted graphs
directed graphs

Administrivia

prac exam #1 **10 January**
at 10am, see WebCMS3 for details
(probably) no sample questions released

census date **13 January**
if you hate me and/or the course
prac exam marks back before then

assignment 2 part 1 is out now:
the Fury of Dracula: the View
make sure you have a group on WebCMS 3

Assignment 2, Part 1

the Fury of Dracula: the View

use a version control system
like Fossil, Git, SVN, etc.

use documentation tools like Doxygen

start sooner rather than later;
write some tests before you begin

Digraphs
Applications
Terminology
Representation
DAGs

Wgraphs

Directed Graphs

Directed Graphs

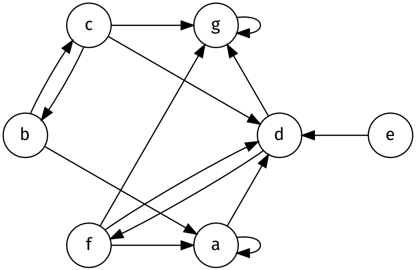
We've mostly considered *undirected* graphs:
an edge relates two vertices equivalently.

Some applications require us to consider
directional edges: $v \to w \neq w \to v$
e.g., 'follow' on Twitter, one-way streets, etc.

In an directed graph or digraph:
edges have direction;
self-loops are allowed;
'parallel' edges are allowed.

Directed Graphs

Example



Where can we get to from g?
Can we get to e from anywhere else?

Directed Graphs

Common Domains

domain	vertex is...	edge is...
WWW	web page	hyperlink
chess	board state	legal move
scheduling	task	precedence
program	function	function call
journals	article	citation

Directed Graphs

Problems and Applications

- Is there a directed path from s to t? (transitive closure)
- What is the shortest path from s to t? (shortest path search)
- Are all vertices mutually reachable? (strong connectivity)

Directed Graphs

Problems and Applications

- Is there a directed path from s to t? (transitive closure)
- What is the shortest path from s to t? (shortest path search)
- Are all vertices mutually reachable? (strong connectivity)

- How can I organise a set of tasks? (topological sort)
- How can I crawl the web? (graph traversal)
- Which web pages are important? (PageRank)

Digraph Terminology (I)

in-degree or $d^{-1}(v)$: the number of directed edges leading **into** a vertex

out-degree or $d(v)$: the number of directed edges leading **out of** a vertex

Digraph Terminology (I)

in-degree or $d^{-1}(v)$: the number of directed edges leading into a vertex

out-degree or $d(v)$: the number of directed edges leading out of a vertex

sink a vertex with out-degree 0;

source a vertex with in-degree 0

Digraph Terminology (II)

reachability indicates existence of directed path:
if a directed path $v, \ldots, w$ exists,
w is reachable from v

strongly connected indicates mutual reachability:
if both paths $v, \ldots, w$ and $w, \ldots, v$ exist,
v and w are strongly connected

Digraph Terminology (II)

reachability indicates existence of directed path:
if a directed path $v, \ldots, w$ exists,
w is reachable from v

strongly connected indicates mutual reachability:
if both paths $v, \ldots, w$ and $w, \ldots, v$ exist,
v and w are strongly connected

strong connectivity every vertex reachable from every other vertex;

strongly-connected component maximal strongly-connected subgraph

Digraph Representation

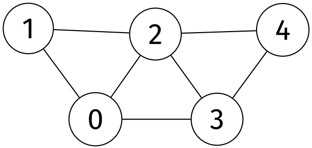
Similar choices as for undirected graphs:

- adjacency matrix … asymmetric, sparse; less space efficient
- adjacency lists … fairly common solution
- edge lists … order of edge components matters
- linked data structures … pointers inherently directional

Can we make our undirected graph implementations directed? Yes!

Directed Graphs

Implementation: Adjacency Matrix

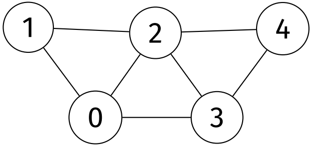


$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

unweighted, undirected

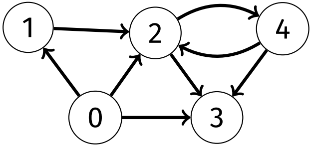
Directed Graphs

Implementation: Adjacency Matrix



$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

unweighted, undirected



$$\begin{bmatrix} - & 1 & 1 & 1 & - \\ - & - & 1 & - & - \\ - & - & - & 1 & 1 \\ - & - & - & - & - \\ - & - & 1 & 1 & - \end{bmatrix}$$

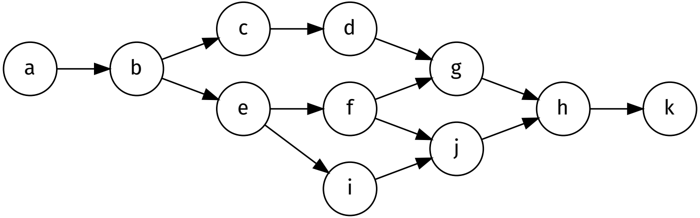
unweighted, directed

Digraph Complexity

	storage	edge add	has edge	outdegree
adj.matrix	$O\left(V + V^2\right)$	$O(1)$	$O(1)$	$O(V)$
adj.list	$O(V + E)$	$O(d(v))$	$O(d(v))$	$O(d(v))$

Overall, adjacency lists tend to be ideal:
real digraphs tend to be sparse
(large V, small average $d(v)$);
algorithms often iterate over v's edges

Directed Acyclic Graphs



Is it a tree? Is it a graph?
No: it's a DAG, a directed acyclic graph.

Tree-like: each vertex has 'children'.
Graph-like: a child vertex may have multiple parents.

Directed Acyclic Graphs

Application: the Topological Sort

NOT EXAMINABLE (and not taught until '4128)

The most common application of a DAG is *topological sorting*:
ordering vertices such that, for any vertices u and v,
if u has a directed edge to v, then v comes after u in the ordering.

Directed Acyclic Graphs

Application: the Topological Sort

NOT EXAMINABLE (and not taught until '4128)

The most common application of a DAG is *topological sorting*:
ordering vertices such that, for any vertices u and v,
if u has a directed edge to v, then v comes after u in the ordering.

Computable with a DFS, tracking *post-order sequence*:
vertices only added after their children have been visited
$\Rightarrow$ a valid topological ordering

Directed Acyclic Graphs

Application: the Topological Sort

NOT EXAMINABLE (and not taught until ’4128)

The most common application of a DAG is *topological sorting*:
ordering vertices such that, for any vertices u and v,
if u has a directed edge to v, then v comes after u in the ordering.

Computable with a DFS, tracking *post-order sequence*:
vertices only added after their children have been visited
$\Rightarrow$ a valid topological ordering

dependency problems: *make(1)*, spreadsheets
version-control systems: Git, Fossil, etc.

Digraph Traversal

Mostly the same algorithms as for undirected graphs:
DFS and BFS should all Just Work

e.g., Web crawling: visit every page on the web.
BFS with implicit graph;
on visit, scans page for content, keywords, links
... assumption: WWW is fully connected.

Digraphs

Wgraphs

Shortest Paths

Single-Source, Dijkstra

Single-Source, Others

All-Pairs

MSTs

Kruskal

Prim

Others

Weighted Graphs

Weighted Graphs

Some applications require us to consider a cost or weight assigned to a relation between two nodes.

Often, we use a geometric interpretation:
low weight $\Rightarrow$ short edge;
high weight $\Rightarrow$ long edge;

Weighted Graphs

Some applications require us to consider a cost or weight assigned to a relation between two nodes.

Often, we use a geometric interpretation:
low weight $\Rightarrow$ short edge;
high weight $\Rightarrow$ long edge;

Weights aren't always geometric:
some weights are negative.
(We assume we have non-negative weights,
as graphs with negative weights tend to cause problems...)

Weighted Graphs

Implementation

Adjacency matrix:

- store *weight* in each cell, not just true/false.
- need some “no edge exists” value: zero might be a valid weight.

Adjacency list

- add weight to each list node

Edge list:

- add weight to each edge

Linked data structure:

- links become link/weight pairs

Weighted Graphs

Implementation

Adjacency matrix:

- store *weight* in each cell, not just true/false.
- need some "no edge exists" value: zero might be a valid weight.

Adjacency list

- add weight to each list node

Edge list:

- add weight to each edge

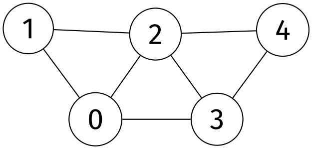
Linked data structure:

- links become link/weight pairs

Works for directed and undirected graphs!

Weighted Graphs

Implementation: Adjacency Matrix

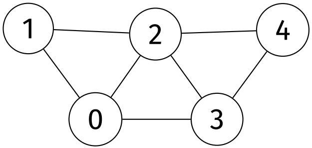


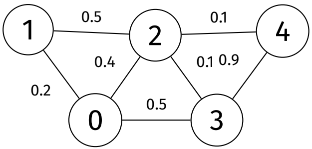
$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

unweighted, undirected

Weighted Graphs

Implementation: Adjacency Matrix





$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

unweighted, undirected

$$\begin{bmatrix} - & 0.2 & 0.4 & 0.5 & - \\ 0.2 & - & 0.5 & - & - \\ 0.4 & 0.5 & - & 0.1 & 0.1 \\ 0.5 & - & 0.1 & - & 0.9 \\ - & - & 0.1 & 0.9 & - \end{bmatrix}$$

weighted, undirected

Digraphs
Wgraphs
Shortest Paths
Single-Source, Dijkstra
Single-Source, Others
All-Pairs
MSTs
Kruskal
Prim
Others

Weighted Graph Problems

The shortest path problem:

- find the minimum cost path between two vertices
- edges may be directed or undirected
- assuming non-negative weights!

Weighted Graph Problems

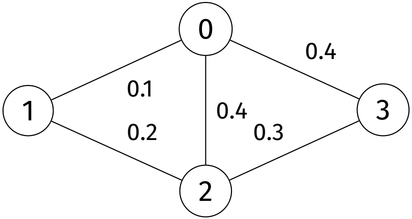
The shortest path problem:

- find the minimum cost path between two vertices
- edges may be directed or undirected
- assuming non-negative weights!

minimum spanning trees (MST):

- find the *weight-minimal* set of edges
  that connect all vertices in a weighted graph
- multiple solutions may exist!
- assuming undirected, non-negatively-weighted graphs

Weighted Graph Problems



What’s the shortest path from 0 to 3?
What’s the least-hops (shortest unweighted path) from 0 to 2?
What is the minimum spanning tree?

Shortest-Path Search

Shortest-path is useful in navigation and route-finding on physical maps, in computer networks, etc.

Several flavours of shortest-path searches exist:

source-target the shortest path from v to w;
single-source the shortest path from v to all other vertices;
all-pairs the shortest paths for all pairs of v, w

Digraphs

Wgraphs

Shortest Paths

Single-Source, Dijkstra

Single-Source, Others

All-Pairs

MSTs

Kruskal

Prim

Others

Shortest-Path Search

Formally

On graph G, the weight of p (as $\mathrm{weight}(p)$)
is the sum of weights of p's edges.

Shortest-Path Search

Formally

On graph G, the weight of p (as $\mathrm{weight}(p)$)
is the sum of weights of p's edges.

The shortest path between v and w
is a simple path $p = [v, \ldots, w]$,
where no other simple path $q = [v, \ldots, w]$, with $q \neq p$,
has a lesser weight (i.e., $\forall q, \mathrm{weight}(p) < \mathrm{weight}(q)$).

Shortest-Path Search

Formally

On graph G, the weight of p (as $\mathrm{weight}(p)$)
is the sum of weights of p's edges.

The shortest path between v and w
is a simple path $p = [v, \ldots, w]$,
where no other simple path $q = [v, \ldots, w]$, with $q \neq p$,
has a lesser weight (i.e., $\forall q, \mathrm{weight}(p) < \mathrm{weight}(q)$).

Assuming a weighted graph, with no negative weights.
(On an unweighted graph, devolves to least-hops.)

Digraphs

Wgraphs

Shortest Paths

Single-Source, Dijkstra

Single-Source, Others

All-Pairs

MSTs

Kruskal

Prim

Others

Single-Source Shortest-Path Search

Given a weighted graph G, and a start vertex v,
we want shortest paths from v to all other vertices.

Single-Source Shortest-Path Search

Given a weighted graph G, and a start vertex v,
we want shortest paths from v to all other vertices.

ASIDE how do we represent it?
we get a vertex-indexed array of distances from v,
and a vertex-indexed array of shortest-path predecessors

Single-Source Shortest-Path Search

Given a weighted graph G, and a start vertex v,
we want shortest paths from v to all other vertices.

ASIDE how do we represent it?
we get a vertex-indexed array of distances from v,
and a vertex-indexed array of shortest-path predecessors
... it's a spanning tree rooted at v.
(Spanning trees can have weighted and/or directed edges, too!)

Single-Source Shortest-Path Search

A Sketch of the Algorithm

`sssp` (Graph g, vertex v):
    `dists[]` $:= [\infty, \cdots]$
    `dists[v]` $= 0$

Single-Source Shortest-Path Search

A Sketch of the Algorithm

`sssp` (Graph g, vertex v):

  `dists[]` $:= [\infty, \cdots]$

  `dists[v]` $= 0$

  pq $:= \textsc{newPQueue}$

Single-Source Shortest-Path Search

A Sketch of the Algorithm

`sssp` (Graph g, vertex v):
    `dists[]` $:= [\infty, \cdots]$
    `dists[v]` $= 0$
    pq $:=$ $\textsc{newPQueue}$
    for each $e := (s, t, \omega)$ **in** $\textsc{adjacent}(v)$,
        $\textsc{enPQueue}(\text{pq}, (s, t), \omega)$

Single-Source Shortest-Path Search

A Sketch of the Algorithm

`sssp` (Graph g, vertex v):
  `dists[]` $:= [\infty, \cdots]$
  `dists[v]` $= 0$
  pq $:=$ $\textsc{newPQueue}$
  for each $e := (s, t, \omega)$ **in** $\textsc{adjacent}(v)$,
    $\textsc{enPQueue}($pq$, (s, t), \omega)$

  while $\textsc{length}(pq) > 0$:
    $(\mathtt{s}, \mathtt{t}), \omega := \textsc{dePQueue}(pq)$
    get edges that connect s and t

Single-Source Shortest-Path Search

A Sketch of the Algorithm

`sssp` (Graph g, vertex v):
    `dists[]` $:= [\infty, \cdots]$
    `dists[v]` $= 0$
    pq $:=$ $\textsc{newPQueue}$
    for each $e := (s, t, \omega)$ **in** $\textsc{adjacent}(v)$,
        $\textsc{enPQueue}$(pq, $(s, t), \omega)$

    while $\textsc{length}(pq) > 0$:
        $(\mathtt{s}, \mathtt{t}), \omega :=$ $\textsc{dePQueue}$(pq)
        get edges that connect s and t
        relax along edge if new distance is better
        add edges with total path weights

Single-Source Shortest-Path Search

Edge Relaxation

"Edge relaxation" along edge e from s to t:

`dist[s]` is length of some path from v to s;
`dist[t]` is length of some path from v to t
if e gives shorter path v to t via s,
update `dist[t]` and `st[t]`.

Single-Source Shortest-Path Search

Edge Relaxation

"Edge relaxation" along edge e from s to t:

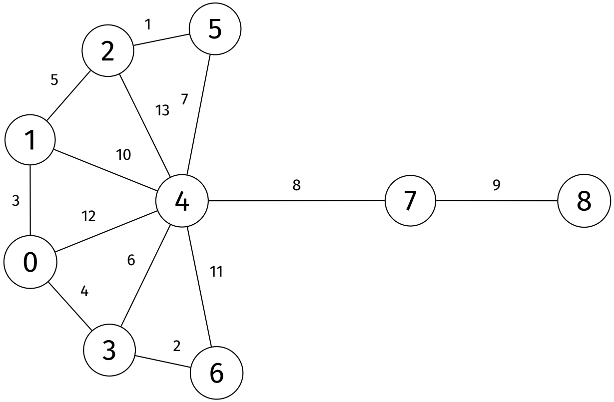
`dist[s]` is length of some path from v to s;
`dist[t]` is length of some path from v to t
if e gives shorter path v to t via s,
update `dist[t]` and `st[t]`.

Relaxation updates data on t, *if* we find a shorter path from v.

```
if (dist[s] + e.weight < dist[t]) {
    dist[t] = dist[s] + e.weight;
    pqueue_en (pq, t, dist[t]);
    st[t] = s;
}
```

Single-Source Shortest-Path Search

Demonstration



	0	1	2	3	4	5	6	7	8
dist									
st									

Single-Source Shortest-Path Search

Results; Complexity

Once this algorithm has run:
shortest path distances are in `dist`;
predecessors in `st` array; trace for a path

Single-Source Shortest-Path Search

Results; Complexity

Once this algorithm has run:
shortest path distances are in `dist`;
predecessors in `st` array; trace for a path

COMPLEXITY:
using an adjacency list and a heap: $O(E \log V)$;
using an adjacency matrix: $O(V^2)$.

Single-Source Shortest-Path Search

Results; Complexity

Once this algorithm has run:
shortest path distances are in `dist`;
predecessors in `st` array; trace for a path

COMPLEXITY:
using an adjacency list and a heap: $O(E \log V)$;
using an adjacency matrix: $O(V^2)$.

Just a graph traversal (*a la* BFS, DFS),
but using a PQueue, instead of a Stack/Queue.

Single-Source Shortest-Path Search

Results; Complexity

Once this algorithm has run:
shortest path distances are in `dist`;
predecessors in `st` array; trace for a path

COMPLEXITY:
using an adjacency list and a heap: $O(E \log V)$;
using an adjacency matrix: $O(V^2)$.

Just a graph traversal (*a la* BFS, DFS),
but using a PQueue, instead of a Stack/Queue.

This algorithm is usually known as
Dijkstra's algorithm.
Sedgewick calls this a PRIORITY-FIRST SEARCH.

Single-Source Shortest-Path Search

Situation Overview

constraint	algorithm	cost	remark
single-source shortest:			
non-negative weights	Dijkstra	V^2	optimal (dense)
non-negative weights	Dijkstra	$E \log V$	conservatively
acyclic	source-queue	E	optimal
no negative cycles	Bellman-Ford	VE	improvements?
(none)	?	?	NP-hard

All-Pairs Shortest-Path Search

Do Dijkstra's SSSP at every vertex.
(This sucks as much as it sounds like it does.)

Floyd-Warshall.
(Out of scope, see '4121/'4128).

All-Pairs Shortest-Path Search

Situation Overview

constraint	algorithm	cost	remark
all-pairs shortest:			
non-negative weights	Floyd	V^3	same for all
non-negative weights	Dijkstra (PFS)	$VE \log V$	conservatively
acyclic	DFS	VE	same for all
no negative cycles	Floyd	V^3	same for all
no negative cycles	Johnson	$VE \log V$	conservatively
(none)	?	?	NP-hard

Minimum Spanning Trees
History

originally, Otakar Borůvka in 1926:
most economical construction of electric power network
(*O jistém problému minimálním*, 'On a certain minimal problem')

routing and network layout:
electricity, telecommunications, electronic, road, ...
widely applicable $\Rightarrow$ intensely studied problem

Minimum Spanning Trees

The Rules of the Game

A spanning tree ST of a graph $G(V, E)$
is a subgraph $G'(V, E')$, such that $E' \subseteq E$.
ST is connected (spanning) and acyclic (tree)

Minimum Spanning Trees

The Rules of the Game

A spanning tree ST of a graph $G(V, E)$
is a subgraph $G'(V, E')$, such that $E' \subseteq E$.
ST is connected (spanning) and acyclic (tree)

A minimum spanning tree MST of a graph G
is a spanning tree of G,
where the sum of edge weight
is no larger than any other spanning tree.

Minimum Spanning Trees

The Rules of the Game

A spanning tree ST of a graph $G(V, E)$
is a subgraph $G'(V, E')$, such that $E' \subseteq E$.
ST is connected (spanning) and acyclic (tree)

A minimum spanning tree MST of a graph G
is a spanning tree of G,
where the sum of edge weight
is no larger than any other spanning tree.

So: how do we (efficiently) find a MST for G?

Digraphs

Wgraphs
Shortest Paths
Single-Source, Dijkstra
Single-Source, Others
All-Pairs
MSTs
Kruskal
Prim
Others

Kruskal’s Algorithm

take all edges, sorted according to their weight;
then, for each edge:
add it to the proto-MST;
unless it would introduce a cycle,

Kruskal’s Algorithm

take all edges, sorted according to their weight;
then, for each edge:
add it to the proto-MST;
unless it would introduce a cycle,

Cycle-checking is really expensive
(DFS everything!)
Sedgewick has a ‘union-find’ that works fine here

Digraphs
Wgraphs
Shortest Paths
Single-Source, Dijkstra
Single-Source, Others
All-Pairs
MSTs
Kruskal
Prim
Others

Kruskal’s Algorithm

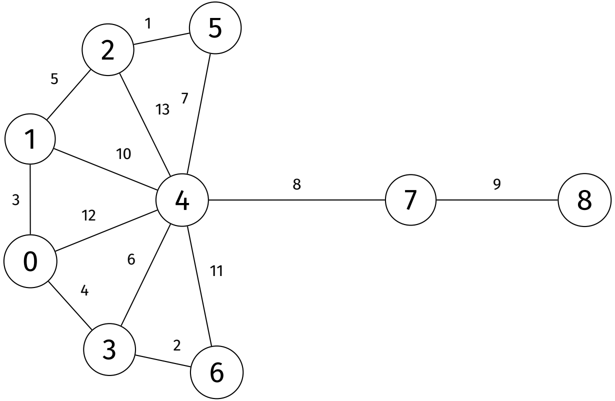
take all edges, sorted according to their weight;
then, for each edge:
add it to the proto-MST;
unless it would introduce a cycle,

Cycle-checking is really expensive
(DFS everything!)
Sedgewick has a ‘union-find’ that works fine here

Sorting dominates overall:
$O\left(E \log E\right)$.

Kruskal's Algorithm

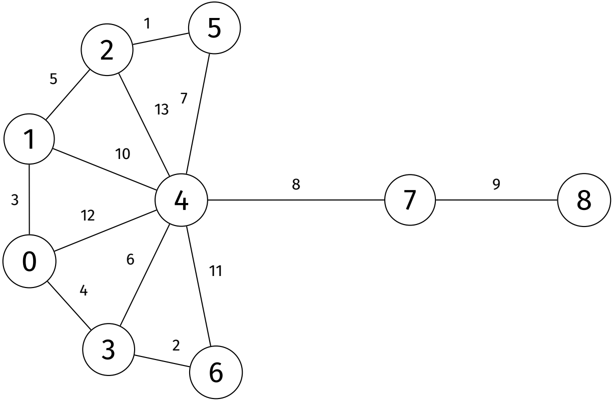
Demonstration (I)



v	0	0	0	1	1	2	2	3	3	4	4	4	7
w	1	3	4	2	4	4	5	4	6	5	6	7	8
ω	3	4	12	5	10	13	1	6	2	7	11	8	9

Kruskal's Algorithm

Demonstration (II)



v	2	3	0	0	1	3	4	4	7	1	4	0	2
w	5	6	1	3	2	4	5	7	8	4	6	4	4
ω	1	2	3	4	5	6	7	8	9	10	11	12	13

Digraphs

Wgraphs

Shortest Paths

Single-Source, Dijkstra

Single-Source, Others

All-Pairs

MSTs

Kruskal

Prim

Others

Prim-Jarník-Dijkstra Algorithm

Another approach to computing an MST for graph $G(V, E)$
discovered by Prim (1957), Jarník (1930), Dijkstra (1959)

Prim-Jarník-Dijkstra Algorithm

Another approach to computing an MST for graph $G(V, E)$
discovered by Prim (1957), Jarník (1930), Dijkstra (1959)

1. start from any vertex s and with an empty MST

Prim-Jarník-Dijkstra Algorithm

Another approach to computing an MST for graph $G(V, E)$
discovered by Prim (1957), Jarník (1930), Dijkstra (1959)

1. start from any vertex s and with an empty MST
2. choose edge not already in MST to add
   - must not contain a self-loop
   - must connect to a vertex already on MST (on the *fringe*)
   - must have minimal weight of all such edges
3. check to see whether adding the new edge brought any of the non-tree vertices closer to the MST

Prim-Jarník-Dijkstra Algorithm

Another approach to computing an MST for graph $G(V, E)$
discovered by Prim (1957), Jarník (1930), Dijkstra (1959)

1. start from any vertex s and with an empty MST
2. choose edge not already in MST to add
   - must not contain a self-loop
   - must connect to a vertex already on MST (on the *fringe*)
   - must have minimal weight of all such edges
3. check to see whether adding the new edge brought any of the non-tree vertices closer to the MST
4. repeat until MST covers all vertices

Prim-Jarník-Dijkstra Algorithm

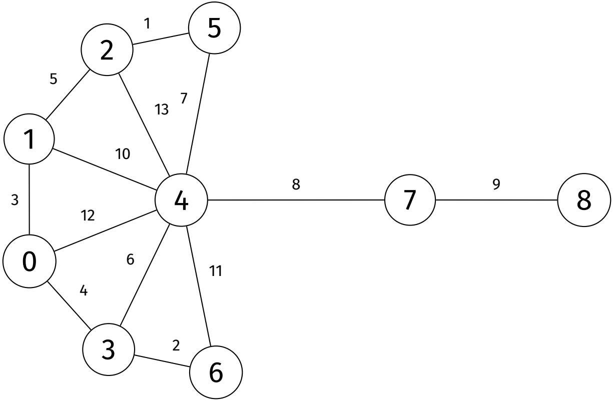
Another approach to computing an MST for graph $G(V, E)$
discovered by Prim (1957), Jarník (1930), Dijkstra (1959)

1. start from any vertex s and with an empty MST
2. choose edge not already in MST to add
   - must not contain a self-loop
   - must connect to a vertex already on MST (on the *fringe*)
   - must have minimal weight of all such edges
3. check to see whether adding the new edge brought any of the non-tree vertices closer to the MST
4. repeat until MST covers all vertices

basically just Dijkstra's SSSP algorithm,
just a graph search but using a PQueue;
$O(E \log V)$ (adjacency lists, heap)
or $O(V^2)$ (adjacency matrix).

Prim-Jarník-Dijkstra Algorithm

Demonstration



v	0	0	0	1	1	2	2	3	3	4	4	4	7
w	1	3	4	2	4	4	5	4	6	5	6	7	8
ω	3	4	12	5	10	13	1	6	2	7	11	8	9

Other MST Algorithms

- Kruskal: grow many forests
- Prim/Jarník/Dijkstra: maintain connectivity on frontier
- Borůvka/Sollin: component-wise
- Tarjan/Karger/Klein: randomised
- Chazelle: deterministic; best-performing