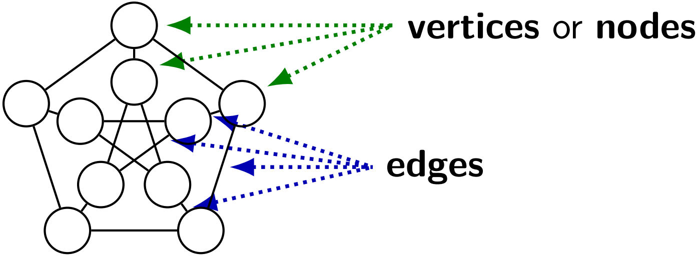
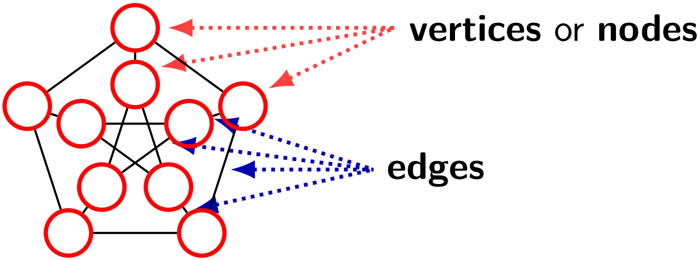
graphs

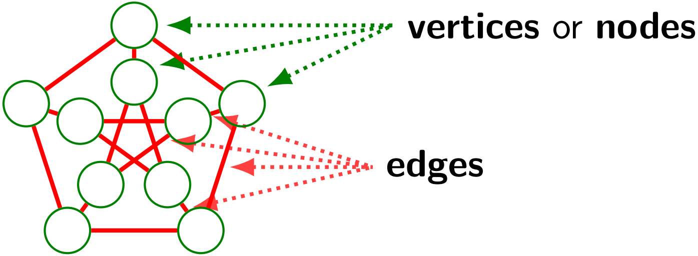
vertices and edges



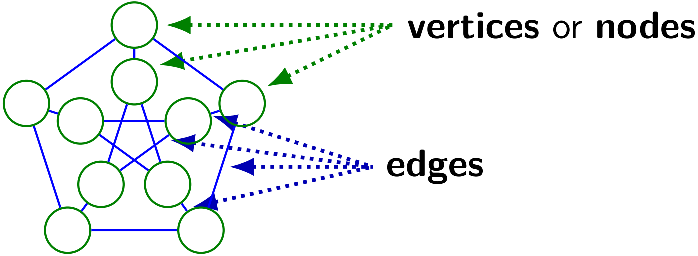
vertices and edges



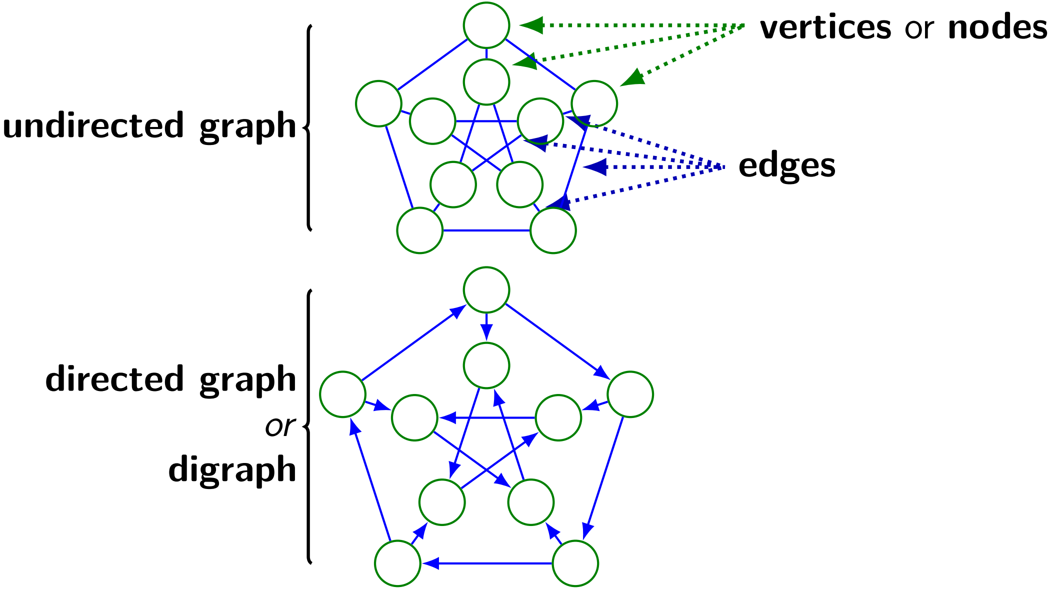
vertices and edges



vertices and edges



vertices and edges



example graphs

lots of things can be represented as graphs

maps

nodes: intersections?

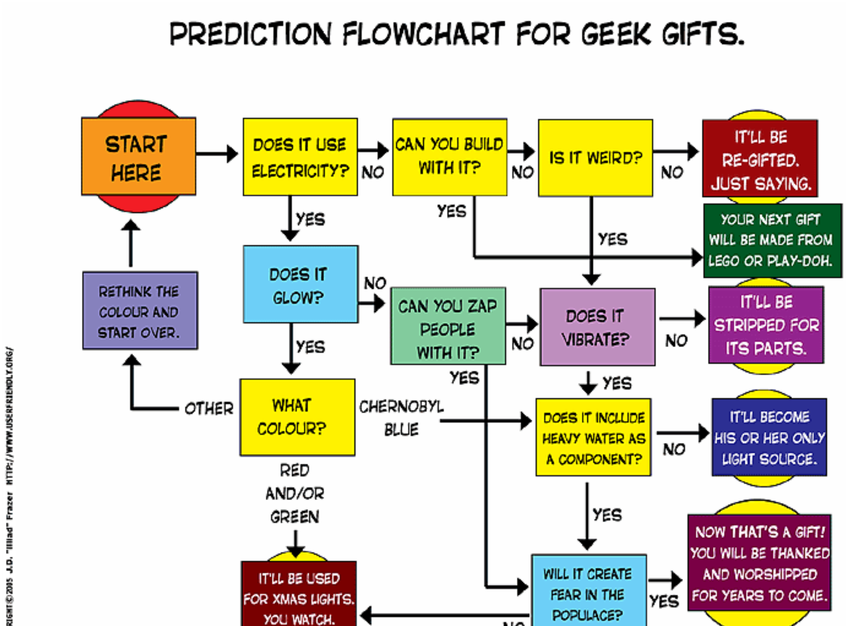
edges: roads?

image: open street map

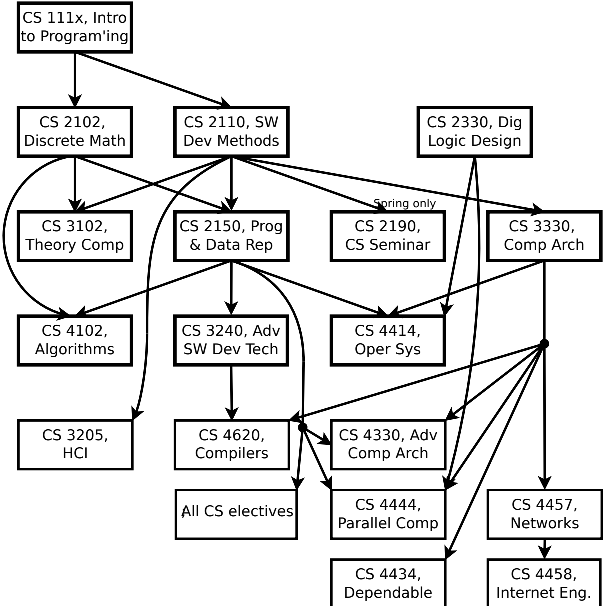
airline routes

image: openflights

flowcharts


PREDICTION FLOWCHART FOR GEEK GIFTS.
START HERE
DOES IT USE ELECTRICITY?
NO
CAN YOU BUILD WITH IT?
NO
IS IT WEIRD?
NO
IT'LL BE RE-GIFTED. JUST SAYING.
YES
YES
YES
YOUR NEXT GIFT WILL BE MADE FROM LEGO OR PLAY-DOH.
RETHINK THE COLOUR AND START OVER.
DOES IT GLOW?
NO
CAN YOU ZAP PEOPLE WITH IT?
NO
DOES IT VIBRATE?
NO
IT'LL BE STRIPPED FOR ITS PARTS.
YES
YES
YES
OTHER
WHAT COLOUR?
CHERNOBYL BLUE
DOES IT INCLUDE HEAVY WATER AS A COMPONENT?
NO
IT'LL BECOME HIS OR HER ONLY LIGHT SOURCE.
RED AND/OR GREEN
YES
NOW THAT'S A GIFT! YOU WILL BE THANKED AND WORSHIPPED FOR YEARS TO COME.
IT'LL BE USED FOR XMAS LIGHTS. YOU WATCH.
WILL IT CREATE FEAR IN THE POPULACE?
YES
HTTP://WWW.USERFRIENDLY.ORG/

pre-requisite tree



formal definition

graph G: $G = (V, E)$

V: set of vertices (possibly empty)

E: set of edges — pairs of vertices (possibly empty)

directed graph/digraph — ordered pairs
undirected graph — unordered pairs

paths, etc.

vertices v and w **adjacent** iff $(v, w) \in E$ or $(w, v) \in E$

path: $v_1, v_2, \ldots v_n$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq n$

length of path: number of **edges** in path

simple path: path of distinct vertices

weighted graphs

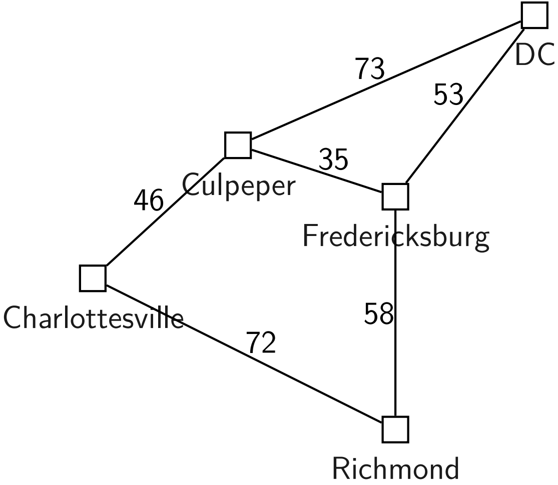
some graphs have **weights** or **costs** associated with edges

example motivation:

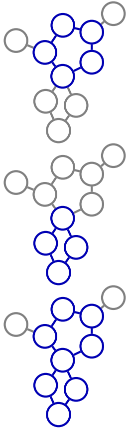
graph representing roads: weight $=$ travel time

weight or cost **of a path** $=$ sum of weights of edges in path

weighted graph example

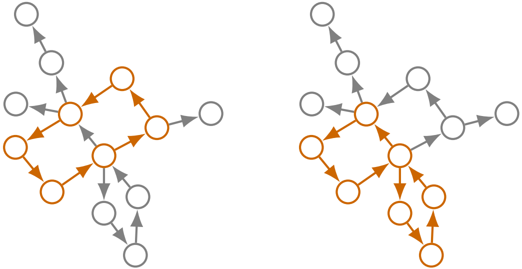


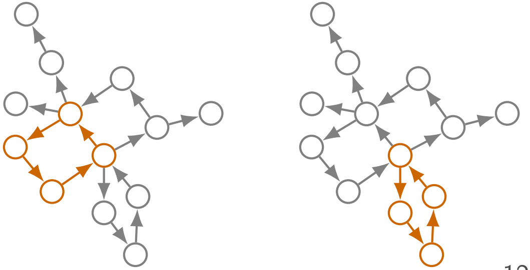
cycles, etc.



cycle: path where length ≥ 1, $v_1 = v_n$

undirected graph: ...and no repeated edges





loops

$(v, v) \in E$

graph terminology is not universal

some sources will use slightly different definitions:

walk instead of **path**

path instead of **simple path**

closed walk instead of **cycle**

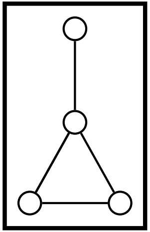
cycle instead of **cycle that is also a simple path**

connectivity

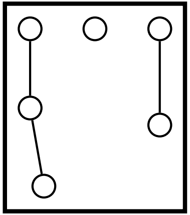
connected graph: for all $x, y \in V$, there exists a path from x to y

N.B: includes 0-length paths

a connected graph



a non-connected graph



in a directed graph...

DAG — directed acyclic graph

no cycles

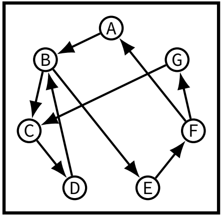
strongly connected — path from every vertex to every other

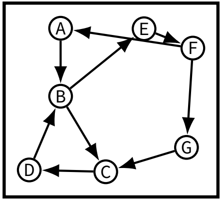
implies cycles (or digraph of 0 or 1 nodes)

weakly connected — would be connected as undirected graph

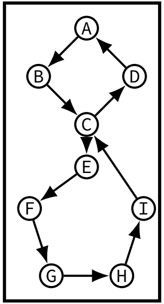
strong/weak connected examples

a strongly connected graph
drawn in two ways

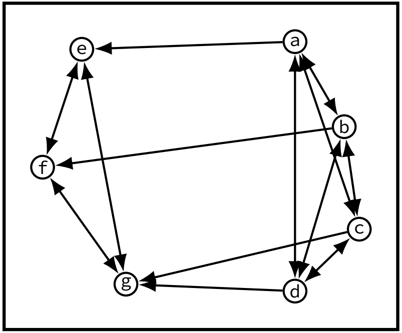




another strongly
connected graph

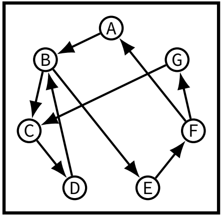


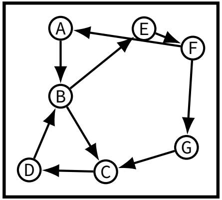
a weakly connected graph



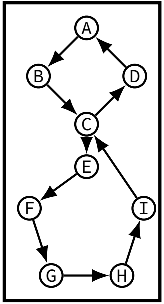
strong/weak connected examples

a strongly connected graph
drawn in two ways

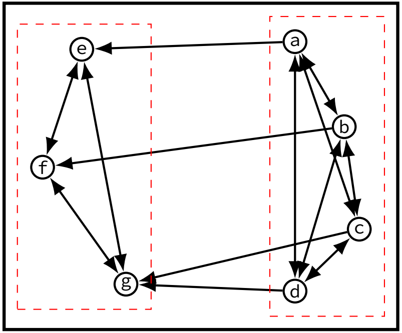




another strongly
connected graph



a weakly connected graph



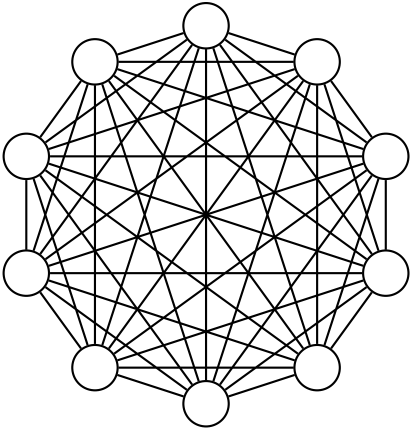
two *strongly connected components*

trees as graphs

trees are connected, acyclic graphs
(with a root chosen)

complete graph

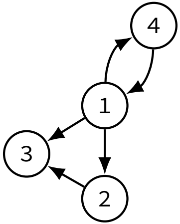
complete graph: graph with edges between every pair of distinct vertices



adjacency matrix

$$A[u][v] = \begin{cases} weight & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

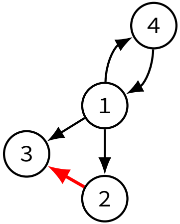
	1	2	3	4
1	0	1	1	1
2	0	0	1	0
3	0	0	0	0
4	1	0	0	0



adjacency matrix

$$A[u][v] = \begin{cases} weight & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

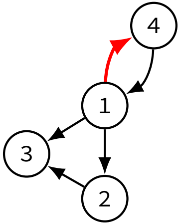
	1	2	3	4
1	0	1	1	1
2	0	0	1	0
3	0	0	0	0
4	1	0	0	0



adjacency matrix

$$A[u][v] = \begin{cases} weight & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

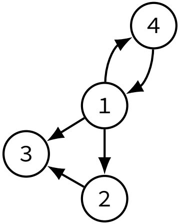
	1	2	3	4
1	0	1	1	1
2	0	0	1	0
3	0	0	0	0
4	1	0	0	0



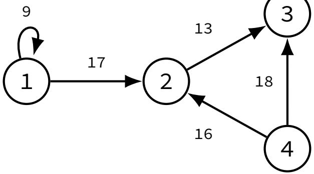
adjacency matrix

$$A[u][v] = \begin{cases} weight & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

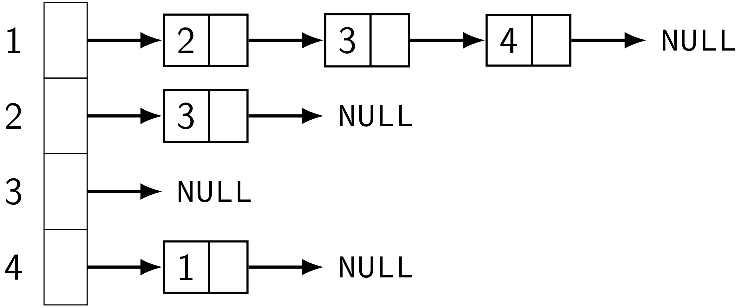
	1	2	3	4
1	0	1	1	1
2	0	0	1	0
3	0	0	0	0
4	1	0	0	0

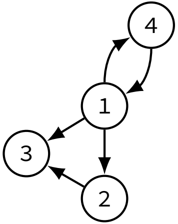


	1	2	3	4
1	9	17	0	0
2	0	0	13	0
3	0	0	0	10
4	0	16	18	0

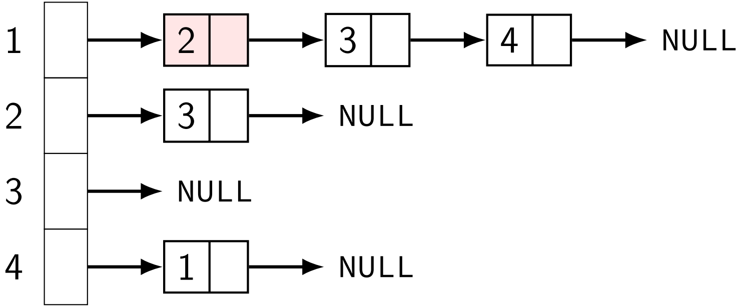


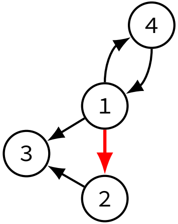
adjacency lists





adjacency lists





choosing representations

choice:

- adjacency matrix
- adjacency list
- more?

issues to consider:

- size
- ease of listing edges from node
- ease of determining if node X has an edge
- ...

variations and alternate representations

adjacency lists might not use linked lists

adjacency matrix can be stored as hashtable (keys=pair of nodes)

...

additional information with nodes

often want to store additional information with vertices, edges...

street names, speed limits, ...

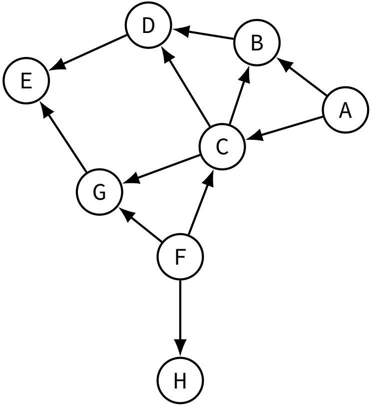
IP addresses, link speeds, ...

...

topological sort

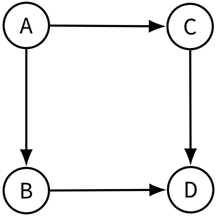
only defined for *directed acyclic graph*

order vertices such that if there is a path from v_i to v_j, then v_j is after v_i

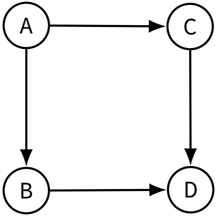


topological sorts:
A, C, B, D, F, G, E, H *or*
F, A, H, C, G, B, D, E *or*
...

exercise: topological sort

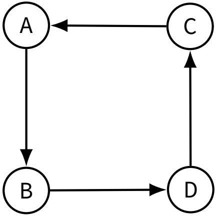


exercise: topological sort

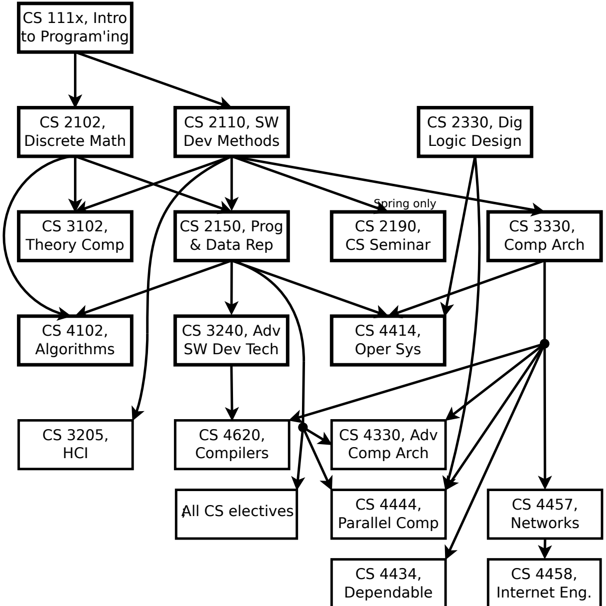


possible answers: A, B, C, D *or* A, C, B, D

no topological sort

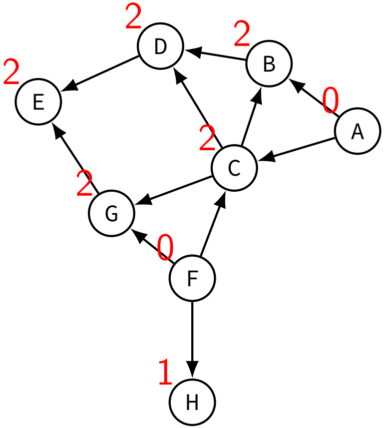


pre-requisite tree



definition: in-degree

indegree of vertex: number of *incoming* edges

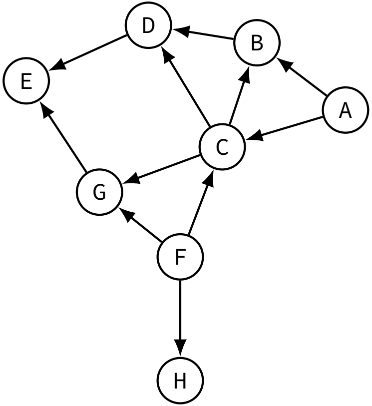


algorithm (simple)

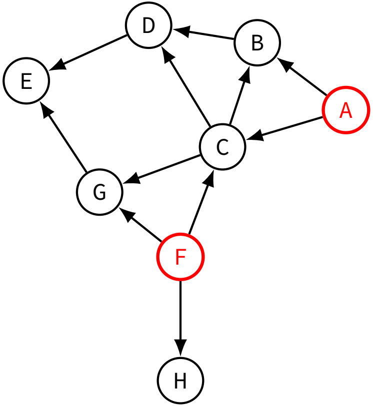
psuedocode:

```
vector<Vertex> topologicalSort(Graph g) {
    vector<Vertex> result;
    for (int i = 0; i < g.numVertices(); ++i) {
        Vertex v = g.findVertexOfInDegreeZero();
        if (did not find v) throw CycleFound();
        result.push_back(v);
        for (Vertex w : v.adjacentVertices()) {
            g.deleteEdge(v, w);
        }
    }
    return result;
}
```

example

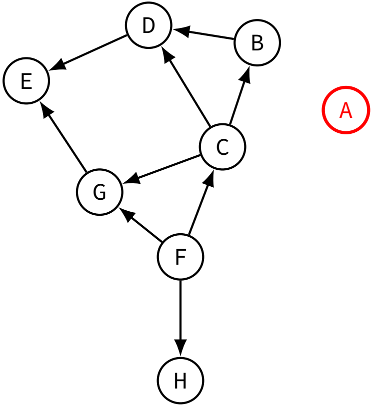


example



initial in-degree 0 vertices — two choices

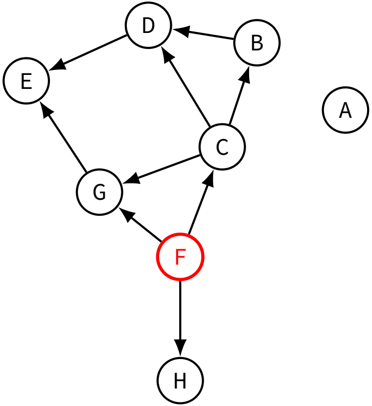
example



choose one (A — arbitrary),
add to result, remove edges

result: A,

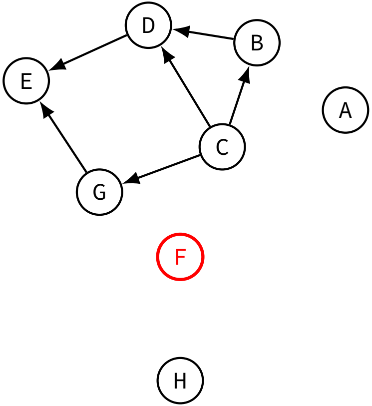
example



one in-degree 0 vertex: F

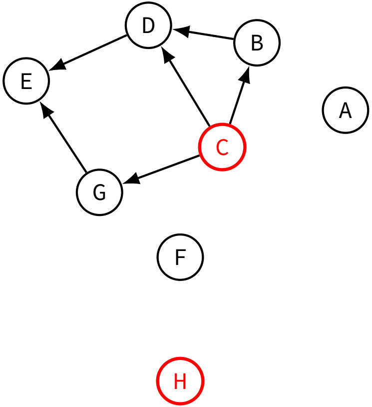
result: A,

example



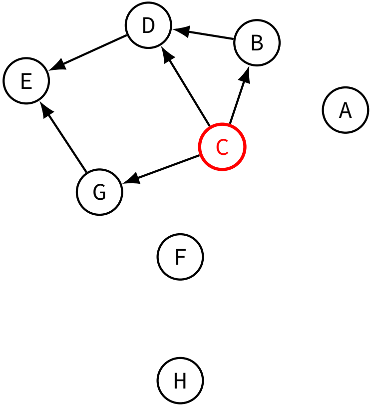
result: A, F,

example



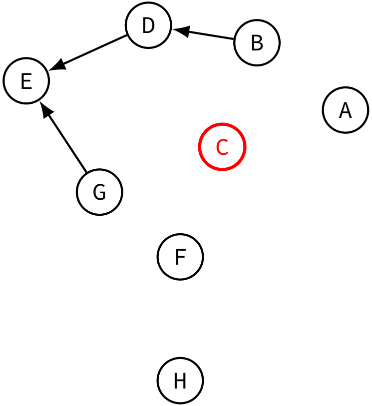
result: A, F, H,

example



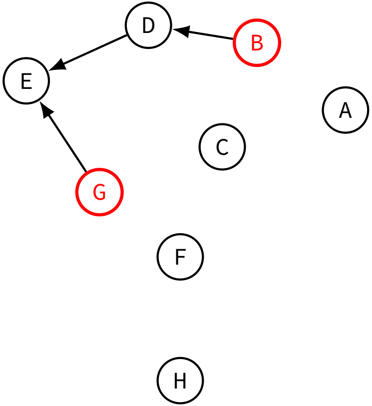
result: A, F, H,

example



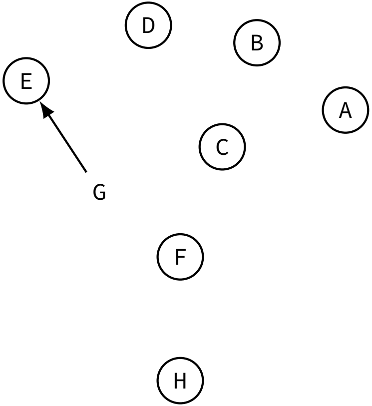
result: A, F, H, C,

example



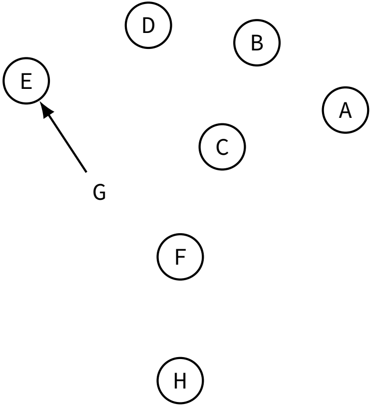
result: A, F, H, C, B, G,

example



result: A, F, H, C, B, G, D,

example



result: A, F, H, C, B, G, D, E,

simple topological sort problems

problem: copying the graph?

problem: finding in-degree 0 vertex?

- scan all vertices and all edges???

better pseudocode

```
vector<Vertex> topologicalSort(Graph g) {
    vector<Vertex> result;
    map<Vertex, int> remainingInDegree = g.getInDegrees();

    Queue<Vertex> pending;
    for (Vertex v : g.vertices())
        if (remainingInDegree[v] == 0)
            pending.enqueue(v);

    while (!pending.empty()) {
        Vertex v = pending.dequeue();
        result.push_back(v);
        for (Edge e: g.edgesFrom(v)) {
            int newDegree = --remainingInDegree[e.toVertex()];
            if (newDegree == 0) pending.enqueue(e.toVertex());
        }
    }
    return result;
}
```

psuedocode idea

track in-degree changes instead of full list of edges

- all we care about is in-degree becoming 0

queue: vertices which have in-degree 0 to process

detect cycles? see if result size $==$ number of vertices

runtime analysis

assuming $|E|$ edges, $|V|$ vertices, and adjacency lists

and in-degree map is constant time (e.g. vertices are 0, 1, 2, …, so it's an array)

step 1: get all in-degrees

$\Theta(|E|)$ (iterate over edges)

step 2: find $+$ enqueue in-degree 0 vertices

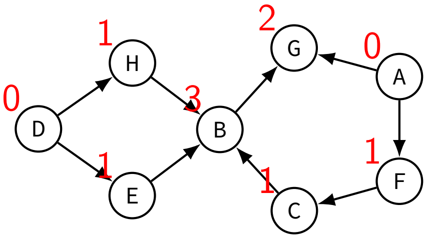
$\Theta(|V|)$ (iterate over vertices)

step 3: for each vertex, check outgoing edges

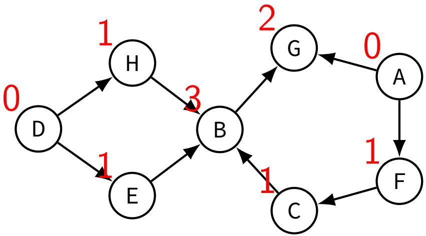
$\Theta(|V| + |E|)$ (each vertex checked exactly once, each edge checked exactly once)

overall: $\Theta(|V| + |E|)$

example



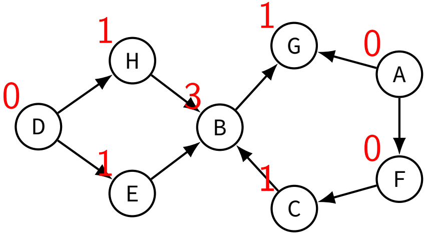
example



queue: A, D,

result:

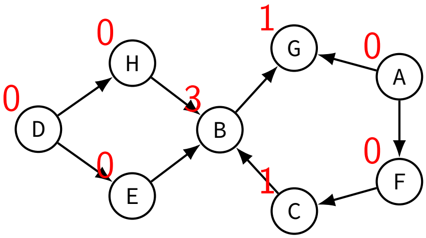
example



queue: ~~A,~~ D, F,

result: A,

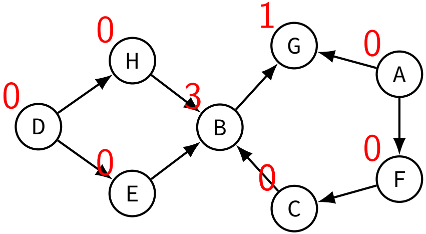
example



queue: ~~A,~~ ~~D,~~ F, H, E,

result: A, D,

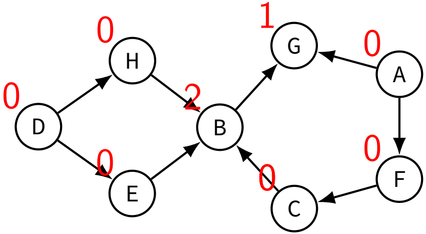
example



queue: ~~A,~~ ~~D,~~ ~~F~~, H, E, C,

result: A, D, F,

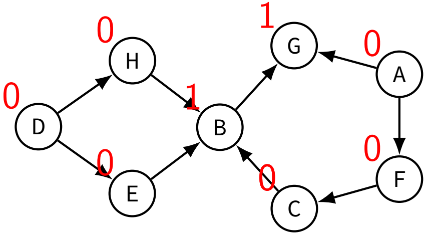
example



queue: ~~A,~~ ~~D,~~ ~~F,~~ ~~H,~~ E, C,

result: A, D, F, H,

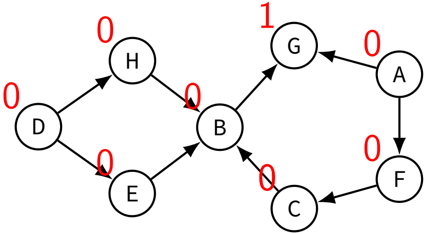
example



queue: ~~A~~, ~~D~~, ~~F~~, ~~H~~, ~~E~~, C,

result: A, D, F, H, E,

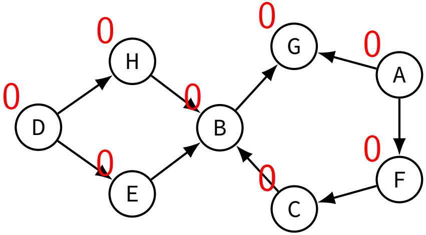
example



queue: ~~A~~, ~~D~~, ~~F~~, ~~H~~, ~~E~~, ~~C~~, B,

result: A, D, F, H, E, C,

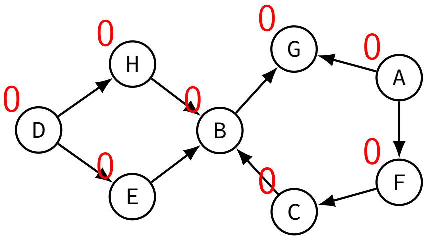
example



queue: ~~A,~~ ~~D,~~ ~~F~~, ~~H~~, ~~E~~, ~~C~~, ~~B~~, G,

result: A, D, F, H, E, C, B,

example



queue: ~~A,~~ ~~D,~~ ~~F~~, ~~H~~, ~~E~~, ~~C~~, B, ~~G~~,

result: A, D, F, H, E, C, B, G

shortest path

shortest path

lowest $\{$weight,number of edges$\}$ path from vertex i to j

shortest path applications

map routing

N degrees of separation'

Internet routing

puzzle/game solving (e.g. rubrik's cube, tic-tac-toe, …)

shortest path algorithm kinds

single pair: path from V to W

single source: for each vertex W, path from V to W

all pairs: for each pair of vertices V, W, path from V to W

shortest path algorithm kinds

single pair: path from V to W

single source: for each vertex W, path from V to W

all pairs: for each pair of vertices V, W, path from V to W

more formally

given graph $G = (V, E)$ and a vertex s (the *source*)...

where an edges (v, w) has weight $w_{v,w}$

for each vertex x find a path $v_1 = s, v_2, \ldots, v_n = x$ such that the $\sum w_{v_i,v_{i+1}}$ is minimum

breadth-first search

shortest path special case: weights $= 1$

algorithm is breadth-first search

special case: breadth-first search on trees

can look at breadth-first search as generalization of pre-order traversal

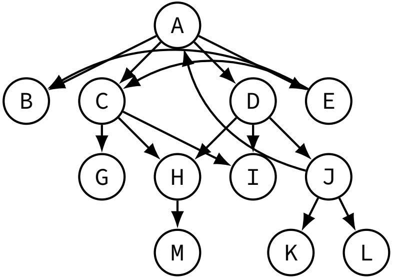
difference: now we can have other paths from roots

breadth first search intuition

start with just source

follow edges to first find vertices at distance 1

then use those to find vertices at distance 2, then distance 3, …

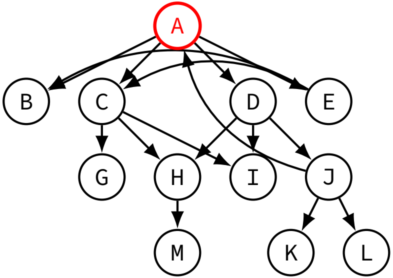


breadth first search intuition

start with just source

follow edges to first find vertices at distance 1

then use those to find vertices at distance 2, then distance 3, …

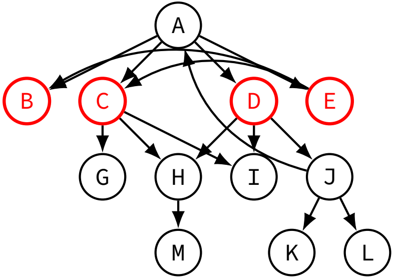


breadth first search intuition

start with just source

follow edges to first find vertices at distance 1

then use those to find vertices at distance 2, then distance 3, …

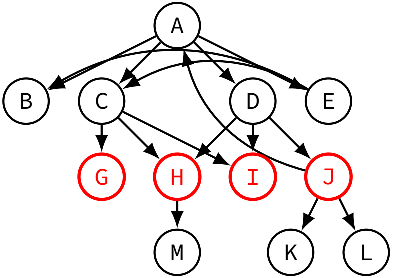


breadth first search intuition

start with just source

follow edges to first find vertices at distance 1

then use those to find vertices at distance 2, then distance 3, …

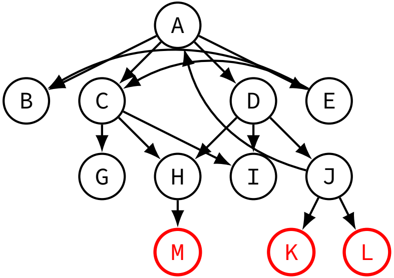


breadth first search intuition

start with just source

follow edges to first find vertices at distance 1

then use those to find vertices at distance 2, then distance 3, …

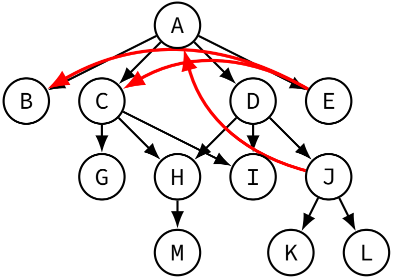


breadth first search intuition

start with just source

follow edges to first find vertices at distance 1

then use those to find vertices at distance 2, then distance 3, …



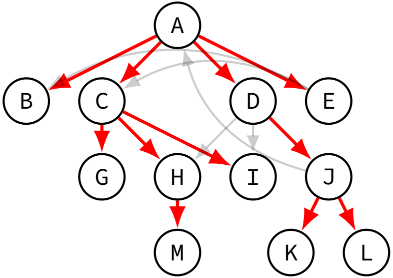
key idea: track visited nodes
so we don't check them again
(already found the shortest path)

breadth first search intuition

start with just source

follow edges to first find vertices at distance 1

then use those to find vertices at distance 2, then distance 3, ...



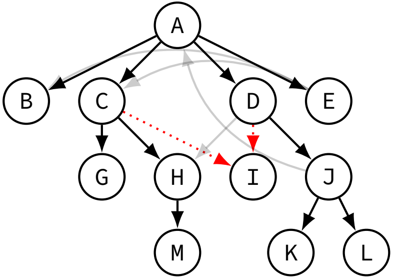
could have list of paths, one per node
but more compact idea:
store one source edge per node
also called *shortest path tree*

breadth first search intuition

start with just source

follow edges to first find vertices at distance 1

then use those to find vertices at distance 2, then distance 3, …



multiple possible answers!

breadth first search pseudocode

```
void Graph::bfs(Vertex start) {
    for (Vertex v: vertices) {
        v.distance = INFINITY;
        v.previous = NULL;
    }
    Queue frontier;
    start.distance = 0;
    frontier.enqueue(start);
    while (!frontier.isEmpty()) {
        Vertex v = q.dequeue();
        for (Vertex w : verticesWithEdgeFrom(v)) {
            // w.distance == INFINITY --> we haven't visited t
            if (w.distance == INFINITY) {
                w.distance = v.distance + 1;
                w.previous = v;
                frontier.enqueue(w);
            }
```

BFS runtime?

need to initialize distances to infinity: $\Theta(|V|)$ operations

need to check every edge: $\Theta(|E|)$ operations

runtime $\Theta(|V| + |E|)$

breadth-first search is greedy

greedy algorithms: make the locally optimal choice, never take it back

BFS: once one finds a node, one enqueues it once

find the node later — skip it

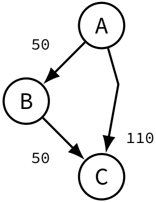
why this is okay: find nodes in order of distance

second time 'visiting' a node — won't be a shorter path!

add weights: a broken idea

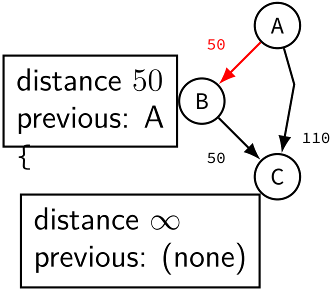
```
void Graph::BROKEN_bestFirstSarch(Vertex start) {
  ...
  while (!frontier.isEmpty()) {
    Vertex v = q.dequeue();
    for (Vertex w : verticesWithEdgeFrom(v)) {
       // BROKEN!
       if (w.distance == INFINITY) {
          w.distance = v.distance + weightOfEdge(v, w);
          w.previous = v;
          frontier.enqueue(w);
       }
     }
   }
  }
}
```

add weights: a broken idea



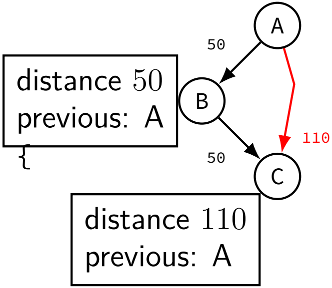

```
void Graph::BROKEN_bestFirstSarch(Vertex start) {
  ...
  while (!frontier.isEmpty()) {
    Vertex v = q.dequeue();
    for (Vertex w : verticesWithEdgeFrom(v)) {
      // BROKEN!
      if (w.distance == INFINITY) {
        w.distance = v.distance + weightOfEdge(v, w);
        w.previous = v;
        frontier.enqueue(w);
      }
    }
  }
  }
}
```

add weights: a broken idea



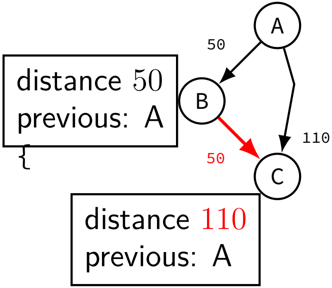

```
void Graph::BROKEN_bestFirstSarch(Vertex start) {
  ...
  while (!frontier.isEmpty()) {
    Vertex v = q.dequeue();
    for (Vertex w : verticesWithEdgeFrom(v)) {
      // BROKEN!
      if (w.distance == INFINITY) {
        w.distance = v.distance + weightOfEdge(v, w);
        w.previous = v;
        frontier.enqueue(w);
      }
    }
  }
}
```

add weights: a broken idea



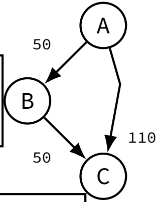

```
void Graph::BROKEN_bestFirstSarch(Vertex start) {
  ...
  while (!frontier.isEmpty()) {
    Vertex v = q.dequeue();
    for (Vertex w : verticesWithEdgeFrom(v)) {
      // BROKEN!
      if (w.distance == INFINITY) {
        w.distance = v.distance + weightOfEdge(v, w);
        w.previous = v;
        frontier.enqueue(w);
      }
    }
  }
}
```

add weights: a broken idea




```
void Graph::BROKEN_bestFirstSarch(Vertex start) {
  ...
  while (!frontier.isEmpty()) {
    Vertex v = q.dequeue();
    for (Vertex w : verticesWithEdgeFrom(v)) {
      // BROKEN!
      if (w.distance == INFINITY) {
        w.distance = v.distance + weightOfEdge(v, w);
        w.previous = v;
        frontier.enqueue(w);
      }
    }
  }
}
```

add weights: a broken idea

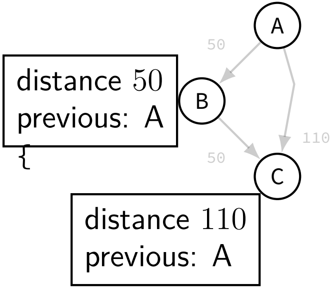


distance 50
previous: A

distance 110
previous: A

```
void Graph::BROKEN_bestFirstSarch(Vertex start) {
  ...
  while (!frontier.isEmpty()) {
    Vertex v = q.dequeue();
    for (Vertex w : verticesWithEdgeFrom(v)) {
      // BROKEN!
      if (w.distance == INFINITY) {
        w.distance = v.distance + weightOfEdge(v, w);
        w.previous = v;
        frontier.enqueue(w);
      }
    }
  }
}
```

add weights: a broken idea




```
void Graph::BROKEN_bestFirstSarch(Vertex start) {
  ...
  while (!frontier.isEmpty()) {
    Vertex v = q.dequeue();
    for (Vertex w : verticesWithEdgeFrom(v)) {
      // BROKEN!
      if (w.distance == INFINITY) {
        w.distance = v.distance + weightOfEdge(v, w);
        w.previous = v;
        frontier.enqueue(w);
      }
    }
  }
}
}
```

fix part 1: update to smaller distance

```
void Graph::BROKEN_bestFirstSarch(Vertex start) {
  ...
  while (!frontier.isEmpty()) {
    Vertex v = q.dequeue();
    for (Vertex w : verticesWithEdgeFrom(v)) {
       int newDistance = v.distance + weightOfEdge(v, w);
       if (newDistance < w.distance) {
          w.distance = newDistance;
          w.previous = v;
          frontier.enqueue(w);
       }
    }
  }
}
```

fix part 1: update to smaller distance

```
void Graph::BROKEN_bestFirstSarch(Vertex start) {
  ...
  while (!frontier.isEmpty()) {
    Vertex v = q.dequeue();
    for (Vertex w : verticesWithEdgeFrom(v)) {
       int newDistance = v.distance + weightOfEdge(v, w);
       if (newDistance < w.distance) {
          w.distance = newDistance;
          w.previous = v;
          frontier.enqueue(w);
        }
      }
    }
  }
}
```

problem: now enqueuing nodes multiple timeswant to only visit node once

fix part 2: visit nodes once, order by distance

```
void Graph::SLOW_bestFirstSarch(Vertex start) {
    for (Vertex v: vertices) {
        v.distance = INFINITY;
        v.previous = NULL;
        v.visited = false;
    }
    start.distance = 0;
    while (!haveUnvisitedNode()) {
        Vertex v = findUnvisitedNodeWithSmallestDistance();
        v.visited = true;
        for (Vertex w : verticesWithEdgeFrom(v)) {
            // w.distance == INFINITY --> we haven't visited this ve
           int newDistance = v.distance + weightOfEdge(v, w);
            if (newDistance < w.distance) {
                w.distance = newDistance;
                w.previous = v;
            }
        }
    }
```

visiting by distance?

assumption: no negative weights

given this: distance only decreases

and can't find shorter path from further node!

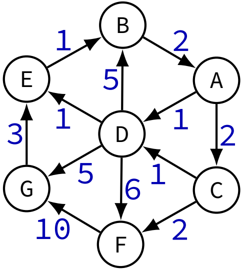
fix part 3: a faster search

```
void Graph::bestFirstSarch(Vertex start) {
    PriorityQueue pq;
    for (Vertex v: vertices) {
        v.distance = INFINITY; v.previous = NULL;
    }
    start.distance = 0;
    pq.insert(0, start);
    while (!pq.empty()) {
        Vertex v = pq.deleteMin();
        for (Vertex w : verticesWithEdgeFrom(v)) {
            // w.distance == INFINITY --> we haven't visited this ve
            int oldDistance = w.distance;
            int newDistance = v.distance + weightOfEdge(v, w);
            if (newDistance < oldDistance) {
                w.distance = newDistance;
                w.previous = v;
                if (oldDistance == INFINITY)
                    pq.insert(newDistance, w);
                else
```

a note on names

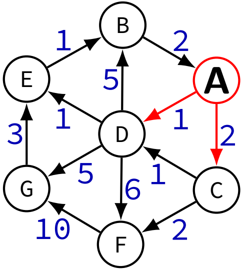
often also called *Dijkstra's algorithm*

Dijkstra's algorithm example 1



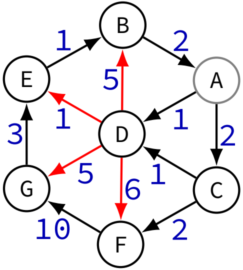
	dist	prev	path
A	0	—	A
B	∞	—	—
C	∞	—	—
D	∞	—	—
E	∞	—	—
F	∞	—	—
G	∞	—	—

Dijkstra's algorithm example 1



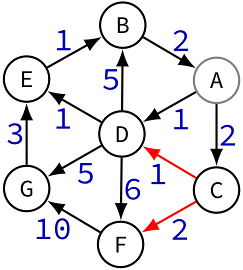
	dist	prev	path
A	0	—	A
B	∞	—	—
C	2	A	A$\to$C
D	1	A	A$\to$D
E	∞	—	—
F	∞	—	—
G	∞	—	—

Dijkstra's algorithm example 1



	dist	prev	path
A	0	—	A
B	6	D	A→D→B
C	2	A	A→C
D	1	A	A→D
E	2	D	A→D→E
F	7	D	A→D→F
G	6	D	A→D→G

Dijkstra's algorithm example 1

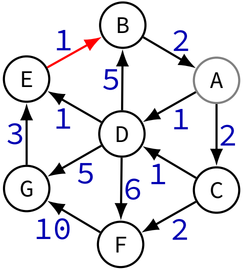


D is adjacent —
but not a shorter path

	dist	prev	path
A	0	—	A
B	6	D	A→D→B
C	2	A	A→C
D	1	A	A→D
E	2	D	A→D→E
F	4	C	A→C→F
G	6	D	A→D→G

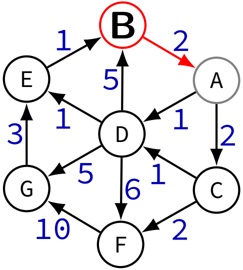
F updated from distance 7 (via D)
to distance 4 (via C)

Dijkstra's algorithm example 1



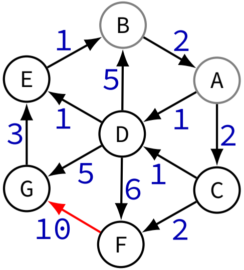
	dist	prev	path
A	0	—	A
B	3	E	A→D→E→B
C	2	A	A→C
D	1	A	A→D
E	2	D	A→D→E
F	4	C	A→C→F
G	6	D	A→D→G

Dijkstra's algorithm example 1



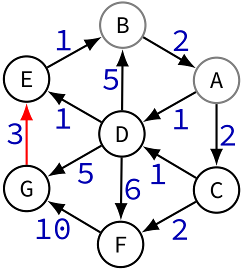
	dist	prev	path
A	0	—	A
B	3	E	A→D→E→B
C	2	A	A→C
D	1	A	A→D
E	2	D	A→D→E
F	4	C	A→C→F
G	6	D	A→D→G

Dijkstra's algorithm example 1



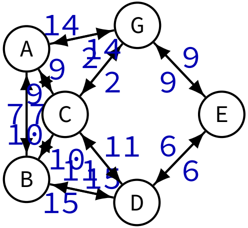
	dist	prev	path
A	0	—	A
B	3	E	A→D→E→B
C	2	A	A→C
D	1	A	A→D
E	2	D	A→D→E
F	4	C	A→C→F
G	6	D	A→D→G

Dijkstra's algorithm example 1



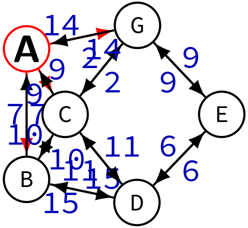
	dist	prev	path
A	0	—	A
B	3	E	A→D→E→B
C	2	A	A→C
D	1	A	A→D
E	2	D	A→D→E
F	4	C	A→C→F
G	6	D	A→D→G

Dijkstra's algorithm example 2



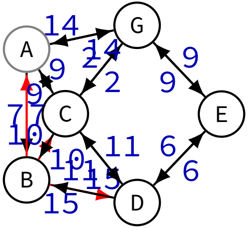
	dist	prev	path
A	0	—	A
B	∞	—	—
C	∞	—	—
D	∞	—	—
E	∞	—	—
G	∞	—	—

Dijkstra's algorithm example 2



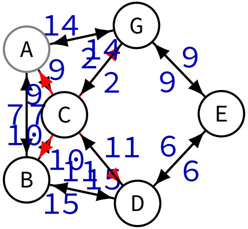
	dist	prev	path
A	0	—	A
B	7	A	A→B
C	9	A	A→C
D	∞	—	—
E	∞	—	—
G	14	A	A→G

Dijkstra's algorithm example 2



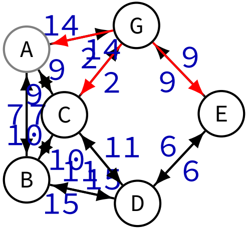
	dist	prev	path
A	0	—	A
B	7	A	A→B
C	9	A	A→C
D	22	B	A→B→D
E	∞	—	—
G	14	A	A→G

Dijkstra's algorithm example 2



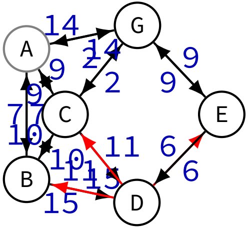
	dist	prev	path
A	0	—	A
B	7	A	A→B
C	9	A	A→C
D	20	C	A→C→D
E	∞	—	—
G	11	C	A→C→G

Dijkstra's algorithm example 2



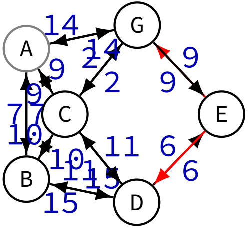
	dist	prev	path
A	0	—	A
B	7	A	A→B
C	9	A	A→C
D	20	C	A→C→D
E	20	G	A→C→G→E
G	11	C	A→C→G

Dijkstra's algorithm example 2



	dist	prev	path
A	0	—	A
B	7	A	A→B
C	9	A	A→C
D	20	C	A→C→D
E	20	G	A→C→G→E
G	11	C	A→C→G

Dijkstra's algorithm example 2



	dist	prev	path
A	0	—	A
B	7	A	A→B
C	9	A	A→C
D	20	C	A→C→D
E	20	G	A→C→G→E
G	11	C	A→C→G

Dijkstra's algorithm runtime

for every vertex (worst case):

find vertex smallest unknown distance

$\Theta(|V|^2)$ total — if checking every vertex

$\Theta(|V|\log|V|)$ total — if removing from heap

scan all edges of vertex, update distances

$\Theta(|E|)$ total — if not maintaining priority queue

$\Theta(|E|\log|V|)$ if updating binary heap

total with binary heap: $\Theta((|E|+|V|)\log|V|)$

Fibanocci heap instead: $\Theta(|E|+|V|\log|V|)$

negative weights

example: weight $=$ fuel used; negative weight $=$ refueling

Dijkstra's algorithm doesn't work

assumption: won't update a node's distance after visiting its edges

alternative algorithms do — e.g. Bellman-Ford ($\Theta(|E||V|)$ runtime)

negative cost cycles — infinitely small cost!

single-source to single-source+destination

what if want to get from A to Z

solution: Dijkstra's algorithm from A but stop early — when we proesss Z

gaurentee: won't update Z's distance again

heuristic shortest path

road map — still slow!

some ideas for speeding up:

- search highways instead of side-roads earlier
- search edges in correct direction earlier
- search from both directions, try to meet
- ...

if you take AI — major topic is *heuristic search*

- taking advantage of ideas like the above
- ...and still getting shortest path, if you want it

travelling salesperson problem

given cities, costs to travel between, least-cost trip that:

- visits each city exactly once, and
- returns to the starting city

as a graph:

- cities = nodes
- costs = edge weights

assume fully connected graph

- alternative: first add infinite weight edges between disconnected nodes

TSP difficulty

solving TSP exactly is NP-hard

worst case: essentially need to enumerate all possible tours

---

but, we can practically solve up to 10000s of cities on real (e.g. road) maps

obviously doing something smarter...

some definitions

Hamiltonian path — path that visits every vertex on a graph exactly once

Hamiltonian cycle — Hamiltonian path that where start node = end node

traveling sales man problem: find least weight Hamiltonian cycle

naive TSP algorithm

choose a starting city x_1

for each unused next city x_2: (n-1 possible)

    for each unused next city x_3: (n-2 possible)

        for each unused next city x_4: (n-3 possible)

        ...

    see if $x_1, x_2, x_3, x_4, \ldots, x_n$ is shorter than anything else

output shortest seen

$(N-1)!$ factorial runtime $= \Theta(N!)$

    worse than $\Theta(2^N)$

naive TSP implementation

psuedocode:

```
vector<Vertex> partial_tour;
void TestTours() {
    if (partial_tour.size() == vertices.size()) {
        partial_tour.push_back(partial_tour[0]);
        if (weightOf(partial_tour[0]) < best_tour_weight) {
            best_tour = partial_tour;
            best_tour_weight = weightOf(best_tour);
        }
        partial_tour.pop_back();
    } else {
        for (Vertex v : vertices − partial_tour) {
            partial_tour.push_back(v);
            FindTour();
            partial_tour.pop_back(v);
        }
    }
}
void TSP() {
```

(n-1)! is big

20 cities — $> 10^{16}$ tours to check

30 cities — $> 10^{30}$ tours to check

...

best gaurenteed TSP algorithm

TSP is NP-hard — no known subexponetial solution

best general algorithm: $\Theta(N^2 2^N)$

20 cities — $> 10^8$ operations

30 cities — $> 10^1 1$ operations

uses *dynamic programming* — covered in 4102

basic idea: if we know $1, 3, 2, 4$ is the best way to visit cities 1, 2, 3, 4 starting at city 1 and ending at 4, then don't figure that out multiple times

e.g. $1, 2, 3, 4, 5, 1$ cannot be shorter than $1, 3, 2, 4, 5, 1$

TSP heuristics

one idea: branch and bound

still: construct lots and lots of possible tours

- keep adding cities

but maintain track extra numbers:

- the best cost found so far
- lower bound on the tours we could find with chosen nodes

stop enumerating (return from `FindTour` early) if lower bound is too low

a lower bound

example lower bound:

if I've chosen cities 1, 2, 4, 3 in that order

minimum cost $=$
$w(1,2) + w(2,4) + w(4,3) + \sum_{i=3}^{n}$ minimum weight of edge from i

- if this is worse than best we've found so far — no sense continuing further

other TSP ideas

TSP on real maps — take advantage of geometry

try cities close to each other first

use map distances to compute minimum costs quickly

some *approximation algorithms*

- get within a certain factor of best solution
- good for pruning very bad solutiosn quickly

TSP records

2006: $85,900$ 'cities'

distances, etc. from real circuit production problem from the 1980s

lab 11

pre-lab: topological sort

in-lab: naive travelling salesperson (map = Tolkein's middle earth)

post-lab: some acceleration techniques

spanning tree definition

given a connected graph G, a spanning tree $G' = (V, E')$ is a *subgraph* such that:

- its edges are a subset of the original graph's (what *subgraph* means)
- it has the same vertices
- it is connected
- it has no cycles — i.e. it is a tree

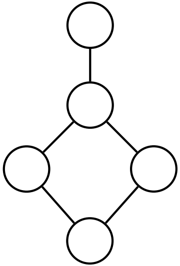
spanning tree construction

take a connected graph

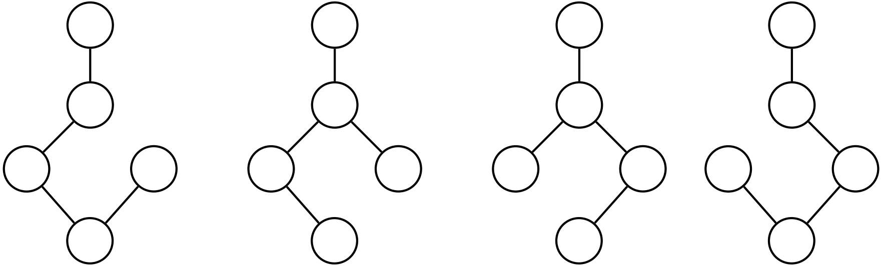
repeatedly: remove an edge that does not disconnect the graph

can't remove any more: a spanning tree — same vertices, but is a tree

spanning tree examples



original graph



spanning trees of graph

minimum spanning tree

A **minimum spanning tree** $T = (V, E')$ of a weighted graph G is a spanning tree such that $\sum e \in E'\text{weight}(e)$ is smallest.

NB: can be multiple minimum spanning trees

Prim's greedy MST algorithm

track: vertices in spanning tree, edges in spanning tree

add a vertex to the spanning tree (arbitrarily)

while not all vertices are in the spanning tree:

pick an edge (u, v) such that

- u is already in the spanning tree
- v is not already in the spanning tree
- (u, v) has the smallest weight of all possible edges

add the edge and v to the spanning tree