Week 03a: Graph Algorithms
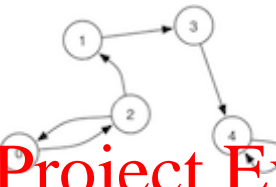
Directed Graphs

Directed Graphs (Digraphs)

In our previous discussion of graphs:

- an edge indicates a relationship between two vertices
- an edge indicates nothing more than a relationship

In many real-world applications of graphs:

- edges are directional   ($v \rightarrow w \neq w \rightarrow v$)

- edges have a *weight*   (cost to go from $v \rightarrow w$)

... Directed Graphs (Digraphs)
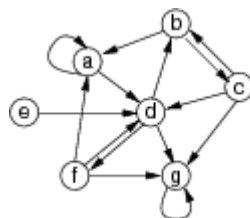
Example digraph and adjacency matrix representation:

Undirectional ⇒ symmetric matrix
Directional ⇒ non-symmetric matrix

Maximum #edges in a digraph with V vertices: V^2

... Directed Graphs (Digraphs)

Terminology for digraphs ...

Directed path: sequence of $n \geq 2$ vertices $v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_n$

- where $(v_i, v_{i+1}) \in edges(G)$ for all v_i, v_{i+1} in sequence
- if $v_1 = v_n$, we have a *directed cycle*

Reachability: *w* is reachable from *v* if ∃ directed path *v,...,w*

Digraph Applications

Potential application areas:

Domain	Vertex	Edge
Web	web page	hyperlink
scheduling	task	precedence
chess	board position	legal move
science	journal article	citation
dynamic data	malloc'd object	pointer
programs	function	function call
make	file	dependency

Assignment Project Exam Help

... Digraph Applications

https://tutorcs.com

Problems to solve on digraphs:

WeChat: cstutorcs

- is there a directed path from *s* to *t*? (transitive closure)
- what is the shortest path from *s* to *t*? (shortest path)
- are all vertices mutually reachable? (strong connectivity)
- how to organise a set of tasks? (topological sort)
- which web pages are "important"? (PageRank)
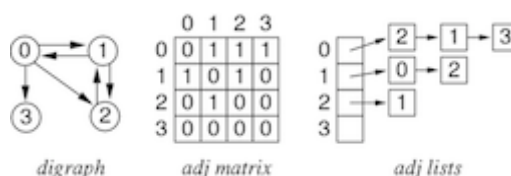- how to build a web crawler? (graph traversal)

Digraph Representation

Similar set of choices as for undirectional graphs:

- array of edges   (directed)
- vertex-indexed adjacency matrix   (non-symmetric)
- vertex-indexed adjacency lists

V vertices identified by *0 .. V-1*

digraph   *adj matrix*   *adj lists*

Reachability

Transitive Closure

Given a digraph G it is potentially useful to know

- is vertex t reachable from vertex s?

Example applications:

- can I complete a schedule from the current state?
- is a malloc'd object being referenced by any pointer?

How to compute transitive closure?

... Transitive Closure

One possibility:

- implement it via $\text{hasPath}(G, s, t)$ (itself implemented by DFS or BFS algorithm)
- feasible if $reachable(G,s,t)$ is infrequent operation

What if we have an algorithm that frequently needs to check reachability?
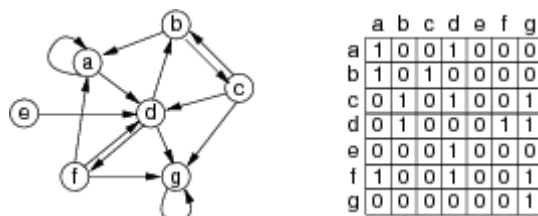
Would be very convenient/efficient to have:

```
reachable(G, s, t):
|  return G.tc[s][t]  // transitive closure matrix
```

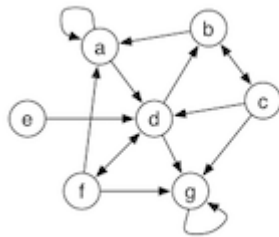Of course, if V is *very* large, then this is not feasible.

Exercise #1: Transitive Closure Matrix

Which reachable $s .. t$ exist in the following graph?

	a	b	c	d	e	f	g
a	1	0	0	1	0	0	0
b	1	0	1	0	0	0	0
c	0	1	0	1	0	0	1
d	0	1	0	0	0	1	1
e	0	0	0	1	0	0	0
f	1	0	0	1	0	0	1
g	0	0	0	0	0	0	1

Transitive closure of example graph:

adjacency matrix   reachability matrix

... Transitive Closure

Goal: produce a matrix of reachability values

- if *tc[s][t]* is 1, then *t* is reachable from *s*
- if *tc[s][t]* is 0, then *t* is not reachable from *s*

So, how to create this matrix?

Observation:

$\forall i,s,t \in$ vertices(G):
    $(s,i) \in$ edges(G) and $(i,t) \in$ edges(G) $\Rightarrow$ *tc[s][t]* = 1

$\texttt{tc[s][t]=1}$ if there is a path from *s* to *t* of length 2   ($s{\to}i{\to}t$)

... Transitive Closure

If we implement the above as:

```
make tc[][] a copy of edges[][]
for all i∈vertices(G) do
   for all s∈vertices(G) do
      for all t∈vertices(G) do
         if tc[s][i]=1 and tc[i][t]=1 then
            tc[s][t]=1
         end if
      end for
   end for
end for
```

then we get an algorithm to convert `edges` into a *tc*

This is known as *Warshall's algorithm*

... Transitive Closure

How it works ...

After iteration 1, $\texttt{tc[s][t]}$ is 1 if

- either $s{\to}t$ exists or $s{\to}0{\to}t$ exists

After iteration 2, `tc[s][t]` is 1 if any of the following exist

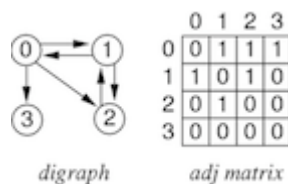- *s→t* or *s→0→t* or *s→1→t* or *s→0→1→t* or *s→1→0→t*

Etc. ... so after the Vth iteration, `tc[s][t]` is 1 if

- there is any directed path in the graph from *s* to *t*

Exercise #2: Transitive Closure

Trace Warshall's algorithm on the following graph:

1st iteration `i=0`:

Assignment Project Exam Help

https://tutorcs.com

`tc`	[0]	[1]	[2]	[3]
[0]	0	1	1	1
[1]	1	1	1	1
[2]	0	1	0	0
[3]	0	0	0	0

WeChat: cstutorcs

2nd iteration `i=1`:

`tc`	[0]	[1]	[2]	[3]
[0]	1	1	1	1
[1]	1	1	1	1
[2]	1	1	1	1
[3]	0	0	0	0

3rd iteration `i=2`: unchanged

4th iteration `i=3`: unchanged

... Transitive Closure

Cost analysis:

- storage: additional V^2 items (each item may be 1 bit)
- computation of transitive closure: $O(V^3)$
- computation of `reachable()`: $O(1)$ after having generated `tc[][]`

Amortisation: would need many calls to `reachable()` to justify other costs

Alternative: use DFS in each call to `reachable()`
Cost analysis:

- storage: cost of queue and set during reachable
- computation of `reachable()`: cost of DFS = $O(V^2)$ (for adjacency matrix)

Digraph Traversal

Same algorithms as for undirected graphs:

`depthFirst(v):`

1. mark `v` as visited
2. for each $(v, w) \in$ *edges*(G) do
   if `w` has not been visited then
      `depthFirst(w)`

`breadth-first(v):`

1. enqueue `v`
2. while queue not empty do
      dequeue `v`
      if `v` not already visited then
         mark `v` as visited
         enqueue each vertex `w` adjacent to `v`

Assignment Project Exam Help

https://tutorcs.com

WeChat: cstutorcs

Example: Web Crawling

Goal: visit every page on the web

Solution: breadth-first search with "implicit" graph

```
webCrawl(startingURL):
|   mark startingURL as alreadySeen
|   enqueue(Q,startingURL)
|   while Q is not empty do
|   |   nextPage=dequeue(Q)
|   |   visit nextPage
|   |   for each hyperLink on nextPage do
|   |   |   if hyperLink not alreadySeen then
|   |   |      mark hyperLink as alreadySeen
|   |   |      enqueue(Q,hyperLink)
|   |   |   end if
|   |   end for
|   end while
```

`visit` scans page and collects e.g. keywords and links

Weighted Graphs

Weighted Graphs

Graphs so far have considered

- edge = an association between two vertices/nodes
- may be a precedence in the association (directed)

Some applications require us to consider

- a *cost* or *weight* of an association
- modelled by assigning values to edges (e.g. positive reals)

Weights can be used in both directed and undirected graphs.

... Weighted Graphs

Example: major airline flight routes in Australia

Representation: edge = direct flight;   weight = approx flying time (hours)

... Weighted Graphs

Weights lead to minimisation-type questions, e.g.

1. Cheapest way to connect all vertices?

- a.k.a. *minimum spanning tree* problem
- assumes: edges are weighted and undirected

2. Cheapest way to get from *A* to *B*?

- a.k.a *shortest path* problem
- assumes: edge weights positive, directed or undirected

Exercise #3: Implementing a Route Finder

If we represent a street map as a graph

- what are the vertices?
- what are the edges?
- are edges directional?
- what are the weights?
- are the weights fixed?

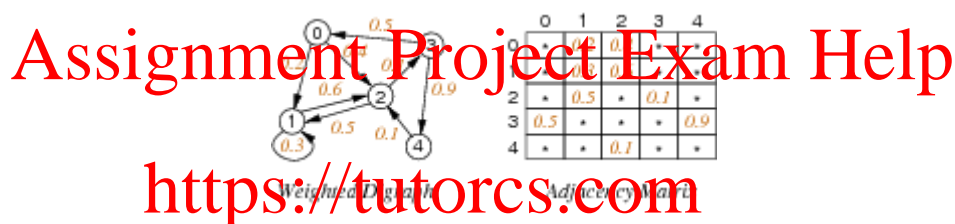# Weighted Graph Representation

Weights can easily be added to:

- adjacency matrix representation   (0/1 → int or float)
- adjacency lists representation   (add int/float to list node)

Both representations work whether edges are directed or not.

... Weighted Graph Representation
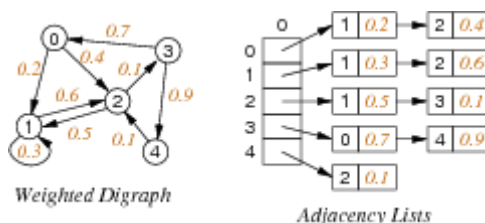
Adjacency matrix representation with weights:

Note: need distinguished value to indicate "no edge".

... Weighted Graph Representation

Adjacency lists representation with weights:

Weighted Digraph   *Adjacency Lists*

Note: if undirected, each edge appears twice with same weight

... Weighted Graph Representation

Sample adjacency matrix implementation in C requires minimal changes to previous Graph ADT:

WGraph.h

```
// edges are pairs of vertices (end-points) plus positive weight
typedef struct Edge {
    Vertex v;
```

```
    Vertex w;
    int    weight;
} Edge;

// returns weight, or 0 if vertices not adjacent
int adjacent(Graph, Vertex, Vertex);
```

... Weighted Graph Representation

WGraph.c

```
typedef struct GraphRep {
    int **edges;  // adjacency matrix storing positive weights
                  // 0 if nodes not adjacent
    int   nV;     // #vertices
    int   nE;     // #edges
} GraphRep;


void insertEdge(Graph g, Edge e) {
    assert(g != NULL && validV(g,e.v) && validV(g,e.w));
    if (g->edges[e.v][e.w] == 0) {  // edge e not in graph
        g->edges[e.v][e.w] = e.weight;
        g->nE++;
    }
}

int adjacent(Graph g, Vertex v, Vertex w) {
    assert(g != NULL && validV(g,v) && validV(g,w));
    return g->edges[v][w];
}
```
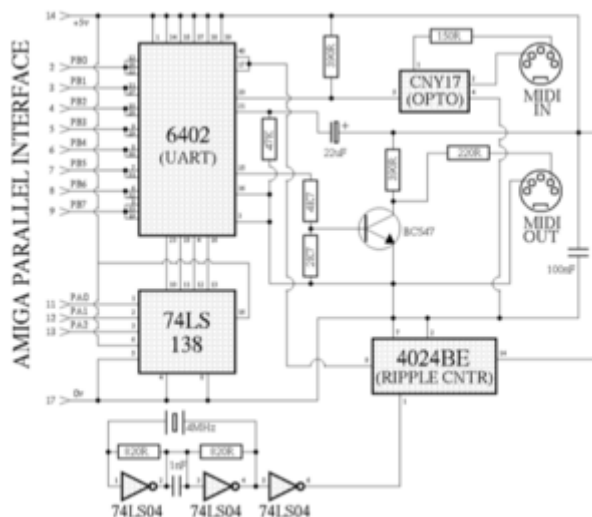
Minimum Spanning Trees

Exercise #4: Minimising Wires in Circuits

Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together.

To interconnect a set of *n* pins we can use an arrangement of *n-1* wires each connecting two pins.

What kind of algorithm would ...

- help us find the arrangement with the least amount of wire?

Minimum Spanning Trees

Reminder: *Spanning tree ST of graph G=(V,E)*

- *spanning* = all vertices, *tree* = no cycles
  - *ST* is a subgraph of *G* (*G'=(V,E')* where *E' ⊆ E*)
  - *ST* is *connected* and *acyclic*

Minimum spanning tree MST of graph *G*

- *MST* is a spanning tree of *G*
- sum of edge weights is no larger than any other ST

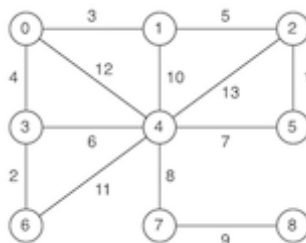Applications: Computer networks, Electrical grids, Transportation networks ...

Problem: how to (efficiently) find MST for graph *G*?

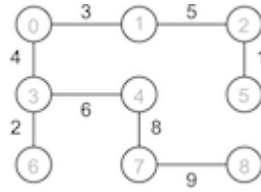NB: MST may not be unique   (e.g. all edges have same weight ⇒ every ST is MST)

... Minimum Spanning Trees

Example:

An MST ...

... Minimum Spanning Trees

Brute force solution:

```
findMST(G):
|  Input   graph G
|  Output  a minimum spanning tree of G
|
|  bestCost=∞
|  for all spanning trees t of G do
|  |  if cost(t)<bestCost then
|  |     bestTree=t
|  |     bestCost=cost(t)
|  |  end if
|  end for
|  return bestTree
```

Example of *generate-and-test* algorithm.

Not useful because #spanning trees is potentially large (e.g. n^{n-2} for a complete graph with n vertices)

... Minimum Spanning Trees

Simplifying assumption:

- edges in *G* are not directed   (MST for digraphs is harder)

Kruskal's Algorithm

One approach to computing MST for graph *G* with *V* nodes:

1. start with empty MST
2. consider edges in increasing weight order
   - add edge if it does not form a cycle in MST
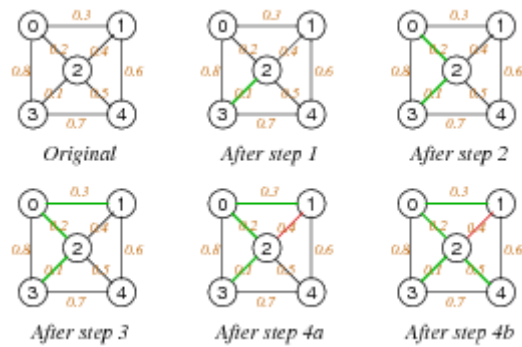3. repeat until *V-1* edges are added

Critical operations:

- iterating over edges in weight order
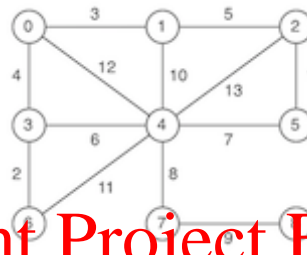- checking for cycles in a graph

... Kruskal's Algorithm

Execution trace of Kruskal's algorithm:

Original   *After step 1*   *After step 2*

After step 3   *After step 4a*   *After step 4b*

Exercise #5: Kruskal's Algorithm
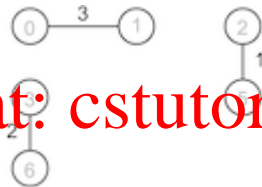
39/67

Show how Kruskal's algorithm produces an MST on:

Assignment Project Exam Help

After 3rd iteration:

https://tutorcs.com
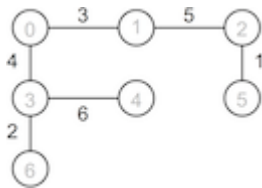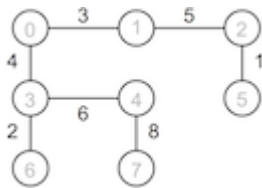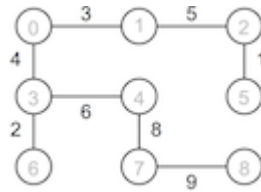

WeChat: cstutorcs

After 6th iteration:

After 7th iteration:

After 8th iteration (V-1=8 edges added):

... Kruskal's Algorithm

Pseudocode:

```
KruskalMST(G):
|  Input   graph G with n nodes
|  Output  a minimum spanning tree of G
|
|  MST=empty graph
|  sort edges(G) by weight
|  for each e∈sortedEdgeList do
|  |  MST = MST ∪ {e}
|  |  if MST has a cyle then
|  |     MST = MST \ {e}
|  |  end if
|  |  if MST has n-1 edges then
|  |     return MST
|  |  end if
|  end for
```

Assignment Project Exam Help

https://tutorcs.com

... Kruskal's Algorithm

WeChat: cstutorcs

Rough time complexity analysis ...

- sorting edge list is *O(E·log E)*
- at least *V* iterations over sorted edges
- on each iteration ...
  - getting next lowest cost edge is *O(1)*
  - checking whether adding it forms a cycle: cost = ??

Possibilities for cycle checking:

- use DFS ... too expensive?
- could use *Union-Find data structure* (see Sedgewick Ch.1)

Prim's Algorithm

Another approach to computing MST for graph *G=(V,E)*:

1. start from any vertex *v* and empty MST
2. choose edge not already in MST to add to MST
   - must be incident on a vertex *s* already connected to *v* in MST
   - must be incident on a vertex *t* not already connected to *v* in MST
   - must have minimal weight of all such edges
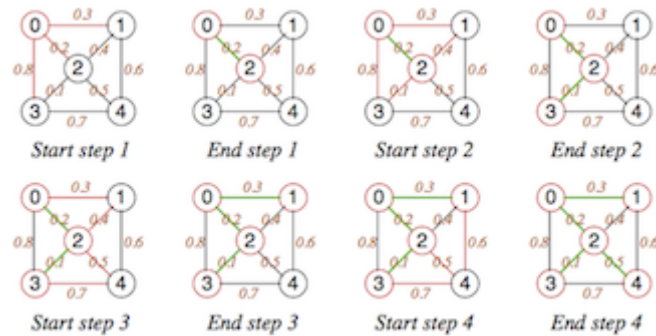3. repeat until MST covers all vertices

Critical operations:

- checking for vertex being connected in a graph
- finding min weight edge in a set of edges

... Prim's Algorithm

Execution trace of Prim's algorithm (starting at $s=0$):

Start step 1   End step 1   Start step 2   End step 2

Start step 3   End step 3   Start step 4   End step 4

Exercise #6: Prim's Algorithm

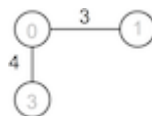Show how Prim's algorithm produces an MST on:

Start from vertex 0

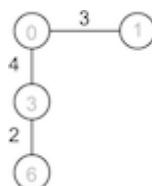After 1st iteration:
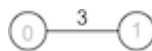
After 2nd iteration:

After 3rd iteration:

After 4th iteration:
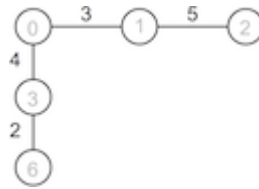
After 8th iteration (all vertices covered):
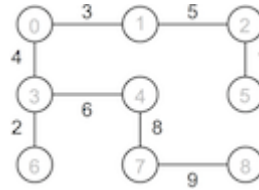
... Prim's Algorithm

Pseudocode:

```
PrimMST(G):
│  Input   graph G with n nodes
│  Output  a minimum spanning tree of G
│
│  MST=empty graph
│  usedV={0}
│  unusedE=edges(g)
│  while |usedV|<n do
│  │  find e=(s,t,w)∈unusedE such that {
│  │      s∈usedV, t∉usedV and w is min weight of all such edges
│  │  }
│  │  MST = MST ∪ {e}
│  │  usedV = usedV ∪ {t}
│  │  unusedE = unusedE \ {e}
│  end while
│  return MST
```

Critical operation: finding best edge

... Prim's Algorithm

Rough time complexity analysis ...

- V iterations of outer loop
- in each iteration ...
  - find min edge with set of edges is $O(E) \Rightarrow O(V{\cdot}E)$ overall
  - find min edge with *priority queue* is $O(\log E) \Rightarrow O(V{\cdot}\log E)$ overall

---

Sidetrack: Priority Queues

Some applications of queues require

- items processed in order of "priority"

- rather than in order of entry (FIFO — first in, first out)

Priority Queues (*PQueues*) provide this via:

- `join`: insert item into PQueue with an associated priority (replacing `enqueue`)
- `leave`: remove item with highest priority (replacing `dequeue`)

Time complexity for naive implementation of a PQueue containing *N* items ...

- *O(1)* for `join`   *O(N)* for `leave`

Most efficient implementation ("heap") ...

- *O(log N)* for `join`, `leave`

Other MST Algorithms

Boruvka's algorithm ... complexity *O(E·log V)*

- the oldest MST algorithm
- start with *V* separate components
- join components using min cost links
- continue until only a single component

Karger, Klein, and Tarjan ... complexity *O(E)*

- based on Boruvka, but non-deterministic
- randomly selects subset of edges to consider
- for the keen, here's the paper describing the algorithm

Shortest Path

Shortest Path

Path = sequence of edges in graph G   *p = (v$_0$,v$_1$), (v$_1$,v$_2$), ..., (v$_{m-1}$,v$_m$)*

cost(path) = sum of edge weights along path

Shortest path between vertices *s* and *t*

- a simple path *p(s,t)* where *s = first(p), t = last(p)*
- no other simple path *q(s,t)* has *cost(q) < cost(p)*

Assumptions: weighted digraph, no negative weights.

Finding shortest path between two given nodes known as *source-target* SP problem

Variations: *single-source* SP, *all-pairs* SP

Applications: navigation, routing in data networks, ...

Single-source Shortest Path (SSSP)

Given: weighted digraph *G*, source vertex *s*

Result: shortest paths from *s* to all other vertices

- `dist[]` *V*-indexed array of cost of shortest path from *s*
- `pred[]` *V*-indexed array of predecessor in shortest path from *s*
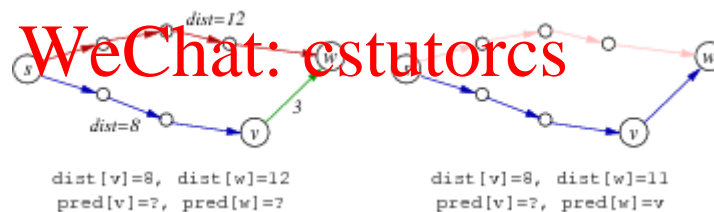
Example:

Edge Relaxation

Assume: `dist[]` and `pred[]` as above (but containing data for shortest paths *discovered so far*)

`dist[v]` is length of shortest known path from *s* to *v*
`dist[w]` is length of shortest known path from *s* to *w*

Relaxation updates data for *w* if we find a shorter path from *s* to *w*.

```
dist[v]=8, dist[w]=12          dist[v]=8, dist[w]=11
pred[v]=?, pred[w]=?           pred[v]=?, pred[w]=v
```

Relaxation along edge *e*=(`v`, `w`, `weight`):

- if `dist[v]+weight` < `dist[w]` then
  update `dist[w]:=dist[v]+weight` and `pred[w]:=v`

Dijkstra's Algorithm

One approach to solving single-source shortest path problem ...

Data: *G*, *s*, `dist[]`, `pred[]` and

- *vSet*: set of vertices whose shortest path from *s* is unknown

Algorithm:

```
dist[]   // array of cost of shortest path from s
pred[]   // array of predecessor in shortest path from s
```

```
dijkstraSSSP(G, source):
│  Input graph G, source node
│
│  initialise dist[] to all ∞, except dist[source]=0
│  initialise pred[] to all −1
│  vSet=all vertices of G
│  while vSet≠∅ do
│  │  find s∈vSet with minimum dist[s]
│  │  for each (s, t, w)∈edges(G) do
│  │     relax along (s, t, w)
│  │  end for
│  │  vSet=vSet\{s}
│  end while
```

Exercise #7: Dijkstra's Algorithm

56/67

Show how Dijkstra's algorithm runs on (source node = 0):

Assignment Project Exam Help

https://tutorcs.com
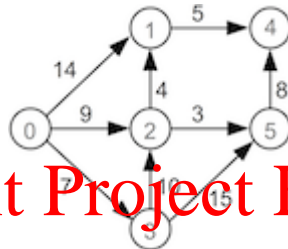
WeChat: cstutorcs

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	∞	∞	∞	∞	∞
pred	–	–	–	–	–	–

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	∞	∞	∞	∞	∞
pred	–	–	–	–	–	–

dist	0	14	9	7	∞	∞
pred	–	0	0	0	–	–

dist	0	14	9	7	∞	22
pred	–	0	0	0	–	3

dist	0	13	9	7	∞	12
pred	–	2	0	0	–	2

dist	0	13	9	7	20	12
pred						

	–	2	0	0	5	2

dist	0	13	9	7	18	12
pred	–	2	0	0	1	2

... Dijkstra's Algorithm

Why Dijkstra's algorithm is correct:

Hypothesis.
(a) For visited *s* ... *dist[s]* is shortest distance from source
(b) For unvisited *t* ... *dist[t]* is shortest distance from source *via visited nodes*

Proof.
Base case: no visited nodes, *dist[source]=0*, *dist[s]=∞* for all other nodes

Induction step:

1. If *s* is unvisited node with minimum *dist[s]*, then *dist[s]* is shortest distance from source to *s*:
   - if ∃ shorter path via only visited nodes, then *dist[s]* would have been updated when processing the predecessor of *s* on this path
   - if ∃ shorter path via an unvisited node *u*, then *dist[u]<dist[s]*, which is impossible if *s* has min distance of all unvisited nodes
2. This implies that (a) holds for *s* after processing *s*
3. (b) still holds for all unvisited nodes *t* after processing *s*:
   - if ∃ shorter path via *s* we would have just updated *dist[t]*
   - if ∃ shorter path without *s* we would have found it previously

... Dijkstra's Algorithm

Time complexity analysis ...

Each edge needs to be considered once ⇒ *O(E)*.

Outer loop has *O(V)* iterations.

Implementing "find s∈vSet with minimum dist[s]"

1. try all s∈vSet ⇒ cost = *O(V)* ⇒ overall cost = *O(E + V²)* = *O(V²)*
2. using a PQueue to implement extracting minimum
   - can improve overall cost to *O(E + V·log V)*   (for best-known implementation)
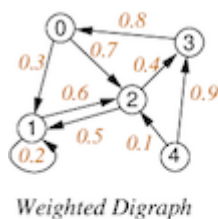
All-pair Shortest Path (APSP)

Given: weighted digraph *G*

Result: shortest paths between all pairs of vertices

- `dist[][]` $V \times V$-indexed matrix of cost of shortest path from v_{row} to v_{col}
- `path[][]` $V \times V$-indexed matrix of next node in shortest path from v_{row} to v_{col}

Example:

Weighted Digraph

Shortest paths between all vertices

Digraph Applications

PageRank

Goal: determine which Web pages are "important"

Approach: ignore page contents; focus on hyperlinks

- treat Web as graph: page = vertex, hyperlink = directed edge
- pages with many incoming hyperlinks are important
- need to compute "incoming degree" for vertices

Problem: the Web is a *very* large graph

- approx. 10^{14} pages, 10^{15} hyperlinks

Assume for the moment that we could build a graph ...

Most frequent operation in algorithm "Does edge (v,w) exist?"

... PageRank

Simple PageRank algorithm:

```
PageRank(myPage):
|  rank=0
|  for each page in the Web do
|  |  if linkExists(page,myPage) then
|  |     rank=rank+1
|  |  end if
|  end for
```

Note: requires *inbound* link check

... PageRank

V = # pages in Web,　　E = # hyperlinks in Web

Costs for computing PageRank for each representation:

Representation	linkExists(v,w)	Cost
Adjacency *matrix*	`edge[v][w]`	*1*
Adjacency *lists*	`inLL(list[v],w)`	$\cong E/V$

Not feasible ...

- adjacency matrix ... $V \cong 10^{14} \Rightarrow$ matrix has 10^{28} cells
- adjacency list ... V lists, each with $\cong$10 hyperlinks $\Rightarrow 10^{15}$ list nodes

So how to really do it?

... PageRank

Approach: the random web surfer

- if we randomly follow links in the web
- ... more likely to re-discover pages with many inbound links

```
curr=random page, prev=null
for a long time do
|   if curr not in array ranked[] then
|      rank[curr]=0
|   end if
|   rank[curr]=rank[curr]+1
|   if random(0, 100)<85 then              // with 85% chance ...
|      prev=curr
|      curr=choose hyperlink from curr  // ... crawl on
|   else
|      curr=random page                 // avoid getting stuck
|      prev=null
|   end if
end for
```

Could be accomplished while we crawl web to build search index

Exercise #8: Implementing Facebook

Facebook could be considered as a giant "social graph"

- what are the vertices?
- what are the edges?
- are edges directional?

What kind of algorithm would ...

- help us find people that you might like to "befriend"?

Summary

- Digraphs, weighted graphs: representations, applications
- Reachability
  - Warshall
- Minimum Spanning Tree (MST)
  - Kruskal, Prim
- Shortest path problems
  - Dijkstra (single source SPP)
  - Floyd (all-pair SSP)
- Flow networks
  - Edmonds-Karp (maximum flow)

- Suggested reading (Sedgewick):
  - digraphs ... Ch. 19.1-19.3
  - weighted graphs ... Ch. 20-20.1
  - MST ... Ch. 20.2-20.4
  - SSP ... Ch. 21-21.3
  - network flows ... Ch. 22.1-22.2

Produced: 6 Jan 2020