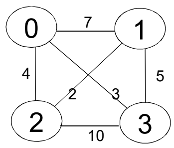
COMP20003
Algorithms and Data Structures
Graph: Shortest Paths

Nir Lipovetzky
Department of Computing and Information Systems
University of Melbourne
Semester 2

Example weighted graph

Adjacency List
0→1→2
1→0→2→3
2→0→1→3
3→1→2

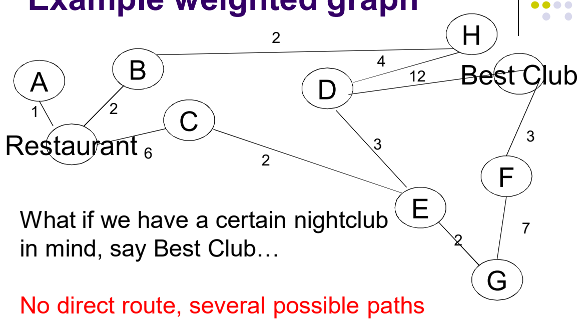


Previous visit order from node 0:

- But if these are restaurants and bars, and we want to go to a nearby bar From restaurant 0…

… in this case the answer is easy. But if you scale it…

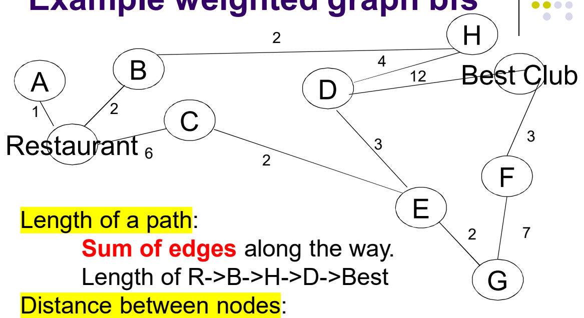
Example weighted graph



What if we have a certain nightclub in mind, say Best Club…

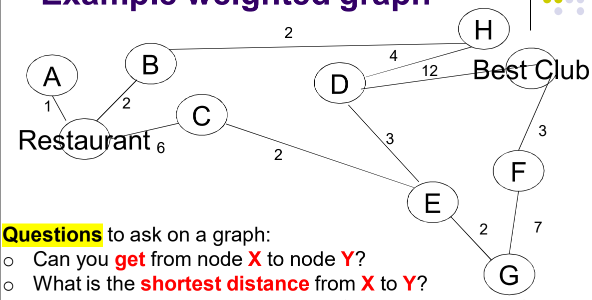
No direct route, several possible paths

Example weighted graph bfs



Length of a path:
  Sum of edges along the way.
  Length of R->B->H->D->Best
Distance between nodes:
  Length of **shortest path**.

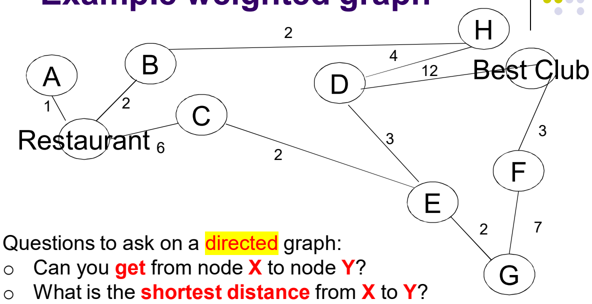
Example weighted graph



Questions to ask on a graph:

- Can you **get** from node **X** to node **Y**?
- What is the **shortest distance** from **X** to **Y**?
  What are the **shortest distances** from **X** to **any node**?
- What are the **shortest distances** from **any** node to **any** other node?

Example weighted graph



Questions to ask on a directed graph:

- Can you **get** from node **X** to node **Y**?
- What is the **shortest distance** from **X** to **Y**?
  What are the **shortest distances** from **X** to **any node**?
- What are the **shortest distances** from **any** node to **any** other node?

Single Source Shortest Path Problem

- Given:
  - Directed graph G(V,E)
  - Source vertex s in V
- Determine:
  - ***Shortest distance*** path

  from s to every other vertex in V

Brute force approach

- For each vertex v_i:
  - Enumerate all paths from *s* to v_i
  - Calculate cost of each path $s \rightarrow\rightarrow v_i$
  - Pick minimum cost.
- How many possible paths from *s* to v_i?
  - For a dense graph O(V!)
  - V=20: 2432902008176640000 paths
  - Not feasible!

Dijkstra's algorithm for single source shortest path

- Greedy algorithm:
  - Based on idea that **any subpath** along a shortest path **is also a shortest path**
  - NodeA→→→→→NodeX→NodeY
    - If shortest path from A to Y is through X,
    - then this path from A to X is also a shortest path
- Dijkstra, E. W., *Numerische Mathematik* **1**: 269–271,1959

Dijkstra's algorithm for single source shortest path

- Greedy algorithm:
  - Based on idea that **any subpath** along a shortest path **is also a shortest path**
  - NodeA→→→→→NodeX→NodeY
    - If shortest path from A to Y is through X,
    - then this path from A to X is also a shortest path
- Assumes **no negative edges**, so:
  - Distance(A→X) $\leq$ Distance(A→Y)

Dijkstra's algorithm for single source shortest path

- Algorithm will give us a shortest path *tree*
- Root = source node
  - **Every node** is **connected** to the **root** through its *shortest* path

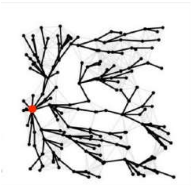


Image from R. Sedgewick, Lecture Notes http://www.cs.princeton.edu/courses/archive/ fall05/cos226/lectures/shortest-path.pdf

Dijkstra's Algorithm: Overview

For every vertex v and source s, maintain *estimate* `dist[v]` of minimum distance *δ(s,v)*

`dist[v]`: length of *a known path s→v*, but not necessarily the shortest path

- `dist[v]`$\geq \delta(s,v)$ Always
- When `dist[v]`$=\infty$, there is no estimate (yet)

Initially `dist[s]`=0, all other `dist[v]`$=\infty$

Dijkstra's Algorithm: Overview

Process vertices one-by-one, updating `dist[v]` until `dist[v]` = $\delta(s,v)$, for every vertex *v*

- Along the way, keep track of best path information in array `pred[v]`

When algorithm finishes:

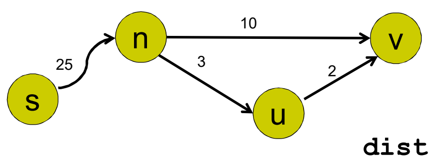
- Have shortest distances in `dist[]`
- Can reconstruct shortest path from `pred[]`

Relaxation: Updating estimated distances

- Relaxation:
  - **Estimate** the solution by answering an **easier problem** (relax the conditions)
  - `dist[]` **Keeps updating** the relaxed estimate **until it is the solution** to the original problem
- For shortest paths:
  - **Estimate**: known distance of ***best*** path **so far**
  - **Solution**: shortest possible distance

Relaxation: Updating estimated distances

- Example:

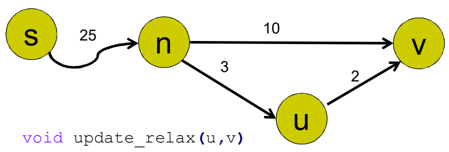



```
dist[v]: 35
pred[v]: n
```

```
dist[u]: 28
dist[v]: 30
pred[v]: u
```


Relaxation: Updating estimated distances

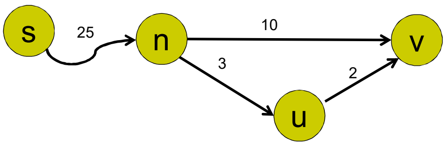
- Example:




```
void update_relax(u,v)
{
	if( dist[u] + edgeweight(u,v) < dist[v] )
	{
		dist[v] = dist[u] + edgeweight(u,v);
		pred[v] = u;
	}
}
```

Relaxation: Updating estimated distances

- Example:



Note:

```
pred[v] = u;
pred[u] = n;
 …
pred[j] = s;
```

Reconstruct path $s \rightsquigarrow v$ going backwards through `pred[]`

Dijkstra's algorithm: successive relaxations

How do we **pick the next node** to look at?

- Process vertices in order of **estimated closeness** to source, value of `dist[v]`
- Priority queue to store vertex *v* and `dist[v]` value

Dijkstra's algorithm (C-ish pseudocode)

```
/* Find shortest paths in graph G from source s*/
/* vertices identified by number for convenience */

void dijkstra(int** G, int s)
{
     int  dist[Vsize], pred[Vsize];
     initialize(G, s, pred, dist);
     run(G, s, pred, dist);

     reconstruct(s, pred, dist);
}
```


Dijkstra's algorithm (C-ish pseudocode)

```
void initialize(int** G, int Vsize, int s, int* pred, int* dist)
{
     int i;
     for( i = 0; i < Vsize; i++)
          dist[i] = MAX_INT;
     dist[s] = 0;
     for( i = 0; i < V; i++)
          pred[i] = NULL;
}
```

Dijkstra's algorithm(C-ish pseudocode)

```
void run(int** G, int Vsize, int s, int* pred, int* dist)
{
  pq_node_t* pq;
  int u, v;
  pq = makePQ(G); /* vertices into min PQ, dist as priority */

  while( !emptyPQ(pq) )
  {
    u = deletemin(pq);
    for(/*each v conneted to u */)
       if(dist[u] + edgeweight(u,v) < dist[v])
          update(v, pred, dist, pq);
   }
   /* At this point vertex u has been processed,
    * i.e. dist[u] = delta(s,u) = shortest path to u found */
}
```

1-21

Dijkstra's algorithm (C-ish pseudocode)

```
void update(int v, int* pred, int* dist, pq_node_t* pq)
{
     dist[v] = dist[u] + edgeweight(u,v);
     pred[v] = u;
     decreaseweight(pq, v, dist[v]);
}
```

1-22

Dijkstra's algorithm (Example)

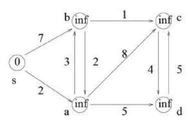
D =
Pred =
PQ =

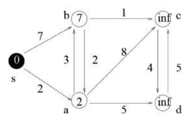
Graph

1-23

Dijkstra's algorithm: example

Example:





Step 0: Initialization.

v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞
$pred[v]$	nil	nil	nil	nil	nil
$color[v]$	W	W	W	W	W

Priority Queue:

v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞

Step 1: As $Adj[s] = \{a, b\}$, work on a and b and update information.

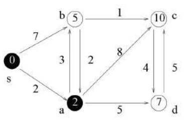
v	s	a	b	c	d
$d[v]$	0	2	7	∞	∞
$pred[v]$	nil	s	s	nil	nil
$color[v]$	B	W	W	W	W

Priority Queue:

v	a	b	c	d
$d[v]$	2	7	∞	∞

From lecture notes by Mordechai Golin, Univ Science and Technology, Hong Kong: http://www.cse.ust.hk/faculty/golin/COMP271Sp03/Notes/MyL09.pdf

1-24

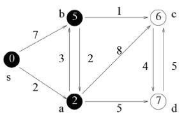


Step 2: After Step 1, a has the minimum key in th priority queue. As $Adj[a] = \{b, c, d\}$, work on b, c, and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	10	7
$pred[v]$	nil	s	a	a	a
$color[v]$	B	B	W	W	W

Priority Queue:

v	b	c	d
$d[v]$	5	10	7



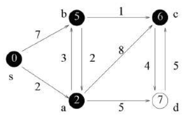
Step 3: After Step 2, b has the minimum key in the priority queue. As $Adj[b] = \{a, c\}$, work on a, c and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	W	W

Priority Queue:

v	c	d
$d[v]$	6	7

COMP 20003 Algorithms and Data Structures 1-25

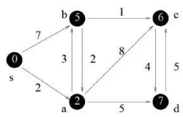


Step 4: After Step 3, c has the minimum key in the priority queue. As $Adj[c] = \{d\}$, work on d and up information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	B	W

Priority Queue:

v	d
$d[v]$	7



Step 5: After Step 4, d has the minimum key in the priority queue. As $Adj[d] = \{c\}$, work on c and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	B	B

Priority Queue: $Q = \emptyset$.

COMP 20003 Algorithms and Data Structures 1-26

Dijkstra's algorithm(ANALYSIS)

```
void run(int** G, int Vsize, int s, int* pred, int* dist)
{
  pq_node_t* pq; /********** Assuming PQ is a minheap */
  int u, v;
  pq = makePQ(G);  /************************* big-O()???? */

  while( !emptyPQ(pq) )
  {
    u = deletemin(pq); /****************** big-O()???? */
    for(/*each v conneted to u */)
       if(dist[u] + edgeweight(u,v) < dist[v])
          update(v, pred, dist, pq); /***** big-O()???? */
   }

}
```

1-27

Dijkstra's Algorithm: Analysis

Cost depends on implementation of PQ

- Using a heap:
  - `makePQ()` *O( )*
  - *V* * `deletemin()` `operations` @*O( )*
  - *O(E)* `decreaseweight()` ops @*O( )*
  - Total: *O( )*

1-28

Dijkstra's Algorithm: Limitations

Assumes no negative edges:
- Good for physical distances
- Distances are static

Negative edges:
- Use Bellman-Ford algorithm
- Cannot deal with negative cycles
- O(V*E)

1-29

Dijkstra's Algorithm: Limitations

Negative *cycles*:
- What is the shortest path?
- Problem is not well-formed, intractable
- Bellman-Ford detects negative cycles (algorithm does terminate, stops keeps shortening paths)

Tutorial:
https://www.dyclassroom.com/graph/detecting-negative-cycle-using-bellman-ford-algorithm

1-30

Applications

More applications
- Robot navigation
- Texture mapping
- Typesetting in TeX
- Urban traffic planning
- Optimal pipelining of VLSI chip
- Telemarketer operator scheduling
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP)
- Exploiting arbitrage opportunities in currency exchange
- Optimal truck routing through given traffic congestion pattern

1-31

Negative Cycle Detection: Arbitrage

Common example in CS materials is arbitrage:
- currency 1 → currency 2 → currency 3 → currency 1'
- If currency 1' > currency 1, you have made money

- Model problem as a graph:
  - Vertices = currency
  - Edges = - $\log_2$(exchange rate)
  - Detect negative cycle and change money → get rich!

Not realistic!
- D.J.Fenn *et al*., "The Mirage of Triangular Arbitrage in the Foreign Currency Exchange Market", *Int. J. Theoretical and Applied Finance* **12**(8), 1105-1123, 2009.

1-32

Edsger W. Dijkstra

- The question of whether computers can think is like the question of whether submarines can swim
- *Computer science is no more about computers than astronomy is about telescopes*
- How do we convince people that in programming simplicity and clarity —in short: what mathematicians call "elegance"— are not a dispensable luxury, but a crucial matter that decides between success and failure?

Elegance is not a dispensable luxury but a quality that decides between success and failure

Turing award 1972

A very nice explanation of Dijkstra's algorithm by Mordechai Golin can be found at

http://www.cse.ust.hk/faculty/golin/COMP271Sp03/Notes/MyL09.pdf