Shortest Paths

- introduction
- Dijkstra's algorithm
- implementation
- priority-first search
- negative weights

References: *Algorithms in Java (Part 5)*, Chapter 21
Intro to Algs and Data Structures, Section 5.5

introduction
Dijkstra's algorithm
implementation
priority-first search
negative weights

Edsger W. Dijkstra: a few select quotes

The question of whether computers can think is like the question of whether submarines can swim.

Do only what only you can do.

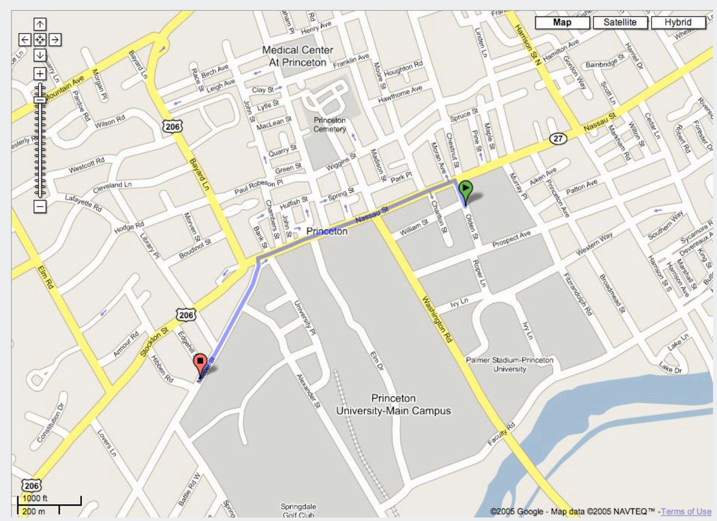
In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.

The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.

APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.

Edger Dijkstra
Turing award 1972

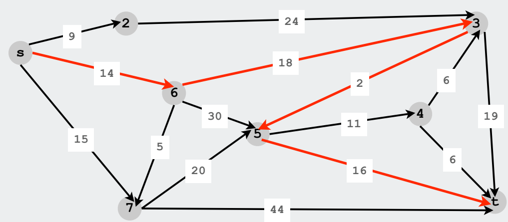
Shortest paths in a weighted digraph



Shortest paths in a weighted digraph

Given a weighted digraph, find the shortest directed path from `s` to `t`.

cost of path = sum of edge costs in path



Path: s→6→3→5→t
Cost: 14 + 18 + 2 + 16 = 50

Note: weights are arbitrary numbers (not necessarily distances) that need not satisfy the triangle inequality

- ex. airline fares
- [stay tuned for others]

Versions

- source-target (s-t)
- single source
- all pairs.
- nonnegative edge weights
- arbitrary weights
- Euclidean weights.

Early history of shortest paths algorithms

Shimbel (1955). Information networks.

Ford (1956). RAND, economics of transportation.

Leyzorek, Gray, Johnson, Ladew, Meaker, Petry, Seitz (1957).
Combat Development Dept. of the Army Electronic Proving Ground.

Dantzig (1958). Simplex method for linear programming.

Bellman (1958). Dynamic programming.

Moore (1959). Routing long-distance telephone calls for Bell Labs.

Dijkstra (1959). Simpler and faster version of Ford's algorithm.

Applications

Shortest-paths is a broadly useful problem-solving model

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

Reference: Network Flows: Theory, Algorithms, and Applications, R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Prentice Hall, 1993.

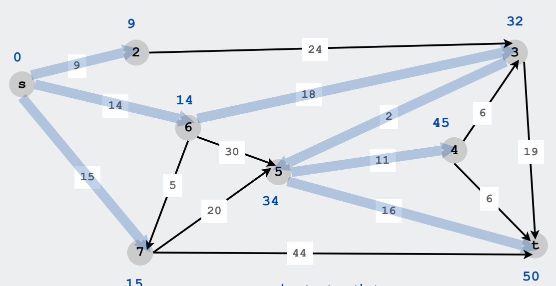
- introduction
- **Dijkstra's algorithm**
- indexed heaps
- priority-first search
- negative weights
- negative cycles

Single-source shortest-paths

Given. Weighted digraph, single source `s`.

Def. Distance from s to v: length of the shortest path from s to v .

Goal. Find distance (and shortest path) from `s` to every other vertex.



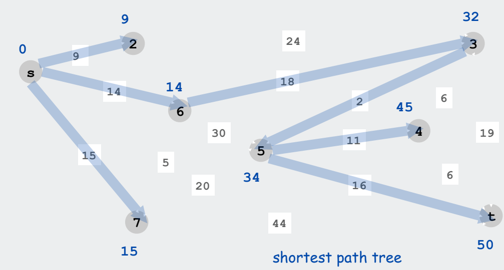
shortest path tree
(parent-link representation)

Shortest paths form a tree

Single-source shortest-paths: basic plan

Given. Weighted digraph, single source `s`.
Goal. Find distance (and shortest path) from `s` to every other vertex.

Design pattern:
- `ShortestPaths` class (`WeightedDigraph` client)
- instance variables: vertex-indexed arrays `dist[]` and `pred[]`
- client query methods return distance and path iterator



shortest path tree
(parent-link representation)

Note: Same as DFS, BFS; BFS works when weights are all 1.

Edge relaxation

For all `v`, `dist[v]` is the length of some path from `s` to `v`.

Relaxation along edge `e` from `v` to `w`.
- `dist[v]` is length of some path from `s` to `v`
- `dist[w]` is length of some path from `s` to `w`
- if `v-w` gives a shorter path to `w` through `v`, update `dist[w]` and `pred[w]`

```
if (dist[w] > dist[v] + e.weight)
{   dist[w] = dist[v] + e.weight);   pred[w] = v;   }
```

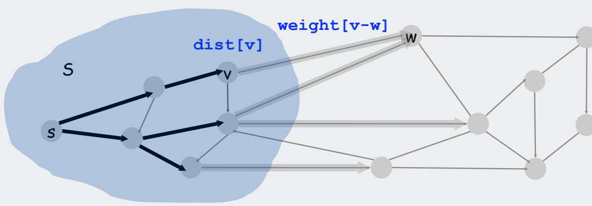
Relaxation sets `dist[w]` to the length of a shorter path from `s` to `w` (if `v-w` gives one)

Dijkstra's algorithm

S: set of vertices for which the shortest path length from s is known.

Invariants
- for all w, dist[w] is the length of shortest known path from s to w.
- for v in S, dist[v] is the length of the shortest path from s to v.

Initialize S to s, dist[s] to 0, dist[v] to ∞ for all other v
Repeat until S contains all vertices connected to s
- find v-w with v in S and w in S' that minimizes dist[v] + weight[v-w]
- relax along that edge
- add w to S

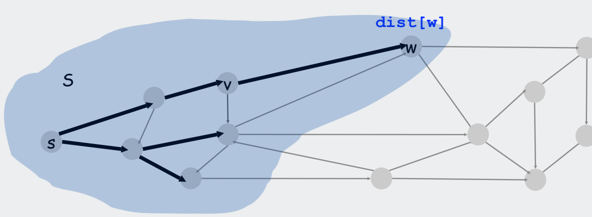


Dijkstra's algorithm

S: set of vertices for which the shortest path length from s is known.

Invariants
- for all v, dist[v] is the length of shortest known path from s to v.
- for v in S, dist[v] is the length of the shortest path from s to v.

Initialize S to s, dist[s] to 0, dist[v] to ∞ for all other v
Repeat until S contains all vertices connected to s
- find v-w with v in S and w in S' that minimizes dist[v] + weight[v-w]
- relax along that edge
- add w to S

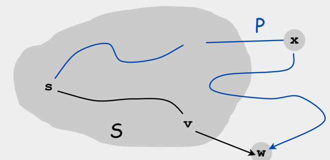


Dijkstra's algorithm proof of correctness

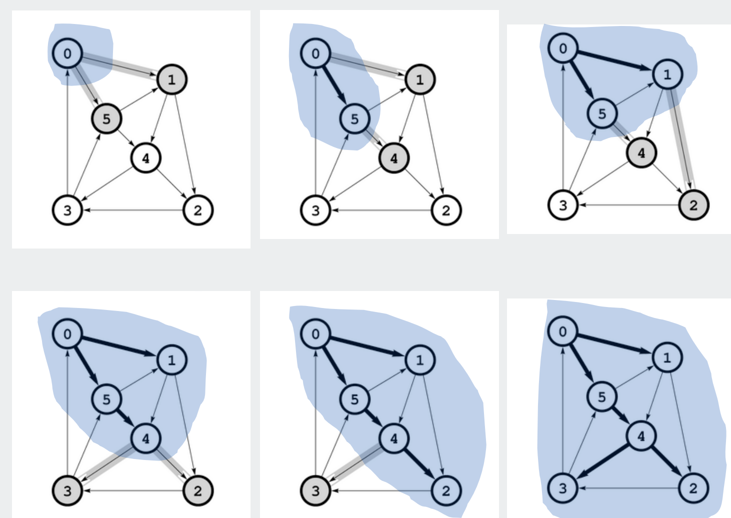
S: set of vertices for which the shortest path length from s is known.

Invariants
- for all v, dist[v] is the length of shortest known path from s to v.
- for v in S, dist[v] is the length of the shortest path from s to v.

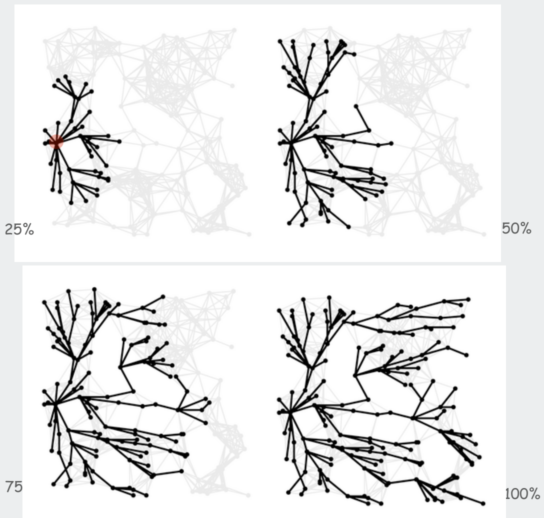
Pf. (by induction on |S|)
- Let w be next vertex added to S.
- Let P* be the s-w path through v.
- Consider any other s-w path P, and let x be first node on path outside S.
- P is already longer than P* as soon as it reaches x by greedy choice.



Dijkstra's Algorithm



Shortest Path Tree



Dijkstra's algorithm implementation approach

Initialize S to s, dist[s] to 0, dist[v] to ∞ for all other v
Repeat until S contains all vertices connected to s

- find `v-w` with `v` in S and `w` in S' that minimizes `dist[v] + weight[v-w]`
- relax along that edge
- add w to S

Idea 1 (easy): Try all edges

Total running time proportional to VE

Dijkstra's algorithm implementation approach

Initialize S to s, dist[s] to 0, dist[v] to ∞ for all other v
Repeat until S contains all vertices connected to s

- find `v-w` with `v` in S and `w` in S' that minimizes `dist[v] + weight[v-w]`
- relax along that edge
- add w to S

Idea 2 (Dijkstra):

- for all v in S, `dist[v]` is the length of the shortest path from `s` to `w`.
- for all w, `dist[w]` is the length of the shortest path to w ending in an edge v-w from a vertex v in S (all other vertices in S).

Two implications

- can find next vertex to add to S in V steps (smallest in dist[])
- can update dist in at most V steps (check neighbors of vertex just added)

Total running time proportional to V^2

Dijkstra's algorithm implementation

Initialize S to s, dist[s] to 0, dist[v] to ∞ for all other v
Repeat until S contains all vertices connected to s

- find `v-w` with `v` in S and `w` in S' that minimizes `dist[v] + weight[v-w]`
- relax along that edge
- add w to S

Idea 3 (this lecture):

- for all v in S, `dist[v]` is the length of the shortest path from `s` to `v`.
- use a priority queue to find the edge to relax

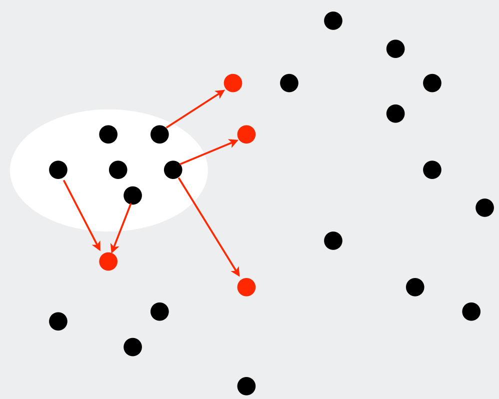
Total running time proportional to E lg V

	sparse	dense
easy	V^3	EV
Dijkstra	V^2	V^2
this lecture	E lg V	E lg V

Dijkstra's algorithm implementation

Q. What goes onto the priority queue?

A. Fringe vertices connected by a single edge to a vertex in S



introduction

Dijkstra's algorithm

implementation

priority-first search

negative weights

Weighted digraphs in Java: weighted edge data type

```
public class Edge
{
   public final int source;
   public final int target;
   public final double weight;

   public Edge(int v, int w, double weight)
   {
      this.source = v;
      this.target = w;
      this.weight = weight;
   }

   public int source()
   {  return source;  }

   public int target()
   {  return target;  }

}
```


Weighted digraphs in Java: weighted digraph data type

```
public class WeightedDigraph
{
   private int V;
   private Sequence<Edge>[] adj;

   public WeightedDigraph(int V)
   {
      this.V = V;
      adj = (Sequence<Edge>[]) new Sequence[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Sequence<Edge>();
   }

   public int V()
   {  return V;  }

   public void addEdge(Edge e)
   {  adj[e.source].add(e);  }

   public Iterable<Edge> adj(int v)
   {  return adj[v];  }

}
```

Dijkstra's algorithm scaffolding

```
public class ShortestPaths
{
   private double[] dist;
   private Edge[] pred;

   public ShortestPaths(WeightedDigraph G, int s)
   {
      dist = new int[G.V()];
      for (int v = 0; v < G.V(); v++)
         dist[v] = INFINITY;
      dist[s] = 0;

      // See next slide.
   }

   public double distance(int v)
   {  return dist[v];  }

   public Iterable<Edge> path(int v)
   {  // As in DFS: eee Lecture 18.  }
}
```

distances from s

previous edge on shortest path from s

initialize distances

compute distances and paths

answer client query for distance from s

answer client query for shortest path from s

Dijkstra's algorithm (compute shortest-path distances)

```
Fringe<Double> fringe;
fringe = new Fringe<Double>(G.V());

dist[s] = 0.0;
fringe.insert(s, dist[s]);

while (!fringe.isEmpty())
{
   int v = fringe.delMin();
   for (Edge e : G.adj(v))
   {
      int w = e.target;
      if (dist[w] > dist[v] + e.weight)
      {
         dist[w] = dist[v] + e.weight;
         pred[w] = e;

         if (fringe.contains(w))
            fringe.decrease(w, dist[w]);
         else
            fringe.insert   (w, dist[w]);

      }
   }
}
```

distances from s

shortest path to v is known (vertex not in S that is closest to s)

relax on edge v-w

update fringe

Designing a data type for the fringe

Fringe operations for Dijkstra's algorithm
- insert
- delete minimum ← like a priority queue
- contains
- decrease value
- test if empty

like a symbol table

Can assume that element keys are integers between 0 and V-1

public class Fringe

	`Fringe(int V)`	create an empty fringe on V elements
`void`	`insert(int i, Value val)`	add item i with given value
`void`	`decrease(int i, Value val)`	decrease value of item i
`int`	`delMin()`	delete and return smallest item
`boolean`	`isEmpty()`	is the fringe empty?
`boolean`	`contains(int i)`	does the fringe contain item i?

Designing a data type for the fringe

Fringe operations for Dijkstra's algorithm frequency counts
- insert V
- delete minimum V
- contains V
- decrease value E
- test if empty V

Challenge: fast implementations of all operations

Implementing a data type for the fringe: array representation

Maintain vertex-indexed arrays `vals[]` and `marked[]`.

- insert key i with value v: `vals[i] = v` and `marked[i] = true`
- delete-min: find smallest `vals[]` entry
- decrease key i to value v: `vals[i] = v`.
- contains: `marked[i] == true`
- is empty: also need count of items on fringe

	cost	frequency	total
insert	1	V	V
delete-min	V	V	V^2
contains	1	V	V
decrease val	1	E	E
is empty	1	V	V
		TOTAL	V^2

Implementing a data type for the fringe: heap representation

Maintain vertex-indexed array `vals[]` and a heap

- insert key i with value v: `vals[i] = v` and update heap
- delete-min: find smallest `vals[]` entry using heap
- decrease key i to value v: `vals[i] = v` and update heap
- contains: [stay tuned]
- is empty: also need count of items on fringe

	cost	frequency	total
insert	lg V	V	V lg V
delete-min	lg V	V	V lg V
contains	1	V	V
decrease val	lg V	E	E lg V
is empty	1	V	V
		TOTAL	E lg V

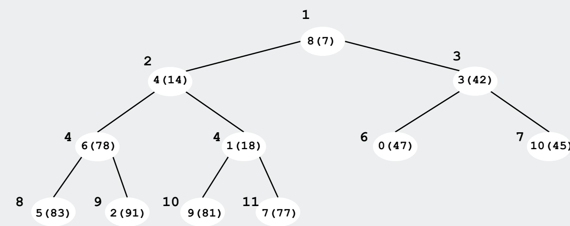
Implementing a data type for the fringe: heap representation

Maintain vertex-indexed arrays `pq[]` and `qp[]`

- `pq[]` is for priority-queue operations (heap code).
- `qp[]` is for symbol-table operations (use vertex index)

`pq[]` implements an indirect heap with values in `val[pq[]]`

- smallest value is at `val[pq[]]`
- compare children at root with `val[pq[2]] < val[pq[3]]`, etc.
- can reuse heap code for priority queue operations



i	pq	qp	val
0		6	47
1	8	5	18
2	4	9	91
3	3	3	42
4	6	2	14
5	1	8	83
6	0	4	78
7	10	11	77
8	5	1	7
9	2	10	81
10	9	7	45
11	7		

Indirect heap for fringe: scaffolding for PQ operations

```
public class Fringe
{
   private int N;
   private int[] pq;
   private double[] val;

   public Fringe(int MAXN)
   {
      val = new double[MAXN + 1];
      pq = new int[MAXN + 1];
   }

   private boolean greater(int i, int j)
   {
      return val[pq[i]] > val[pq[j]];
   }

   private void exch(int i, int j)
   {
      int swap = pq[i]; pq[i] = pq[j]; pq[j] = swap;
   }
```

Indirect heap for fringe: Java code for PQ operations

```
public void insert(int i, Key key)
{
   pq[++N] = i;
   vals[i] = key;
   swim(N);
}

public int delMin()
{
   int min = pq[1];
   exch(1, N--);
   sink(1);
   return min;
}

public boolean isEmpty()
{  return N == 0;  }
```

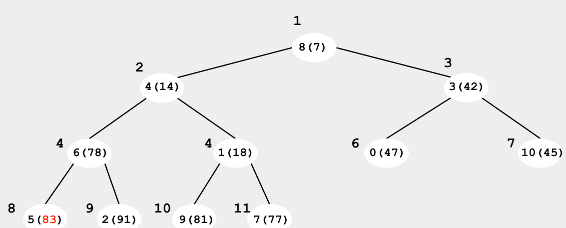

Implementing a data type for the fringe: ST representation

Maintain vertex-indexed arrays `pq[]` and `qp[]`

- `pq[]` is for priority-queue operations (heap code).
- `qp[]` is for symbol-table operations (use vertex index)

`qp[]` implements a vertex-indexed ST giving access to heap positions

- `qp[i]` is heap index of i (`qp[pq[i]] = pq[qp[i]] = i`)
- `qp[i] = -1` iff vertex not in fringe
- decrease key by directly accessing `val[i]`
- then reuse heap code to bubble `qp[i]` up in the heap



i	pq	qp	val
0		6	47
1	8	5	18
2	4	9	91
3	3	3	42
4	6	2	14
5	1	8	83
6	0	4	78
7	10	11	77
8	5	1	7
9	2	10	81
10	9	7	45
11	7		

Indirect heap for fringe: Java code for ST operations

```
public void insert(int i, Key key)
{
   qp[i] = ++N;
   pq[N] = i;
   vals[i] = key;
   swim(N);
}

public int delMin()
{
   int min = pq[1];
   qp[min] = -1;
   exch(1, N--);
   sink(1);
   return min;
}

public void decrease(int i, Key key)
{
   keys[i] = key;
   swim(qp[i]);
}

public boolean contains(int i)
{  return qp[i] != -1;  }

public boolean isEmpty()
{  return N == 0;  }
```


Indirect heap for fringe: additions to scaffolding for ST operations

```
public class Fringe
{
   private int N;
   private int[] pq, qp;
   private double[] val;

   public Fringe(int MAXN)
   {
      val = new double[MAXN + 1];
      pq = new int[MAXN + 1];
      qp = new int[MAXN + 1];
      for (int i = 0; i <= MAXN; i++) qp[i] = -1;
   }

   private boolean greater(int i, int j)
   {
      return val[pq[i]] > val[pq[j]];
   }

   private void exch(int i, int j)
   {
      int swap = pq[i]; pq[i] = pq[j]; pq[j] = swap;
      qp[pq[i]] = i; qp[pq[j]] = j;
   }
```

Dijkstra's algorithm: performance

Fringe implementation directly impacts performance

		array implementation		indirect heap implementation	
	frequency	each op	total	each op	total
insert	V	1	V	lg V	V lg V
delete-min	V	V	V^2	lg V	V lg V
contains	V	1	V	1	V
decrease val	E	1	E	lg V	E lg V
is empty	V	1	V	1	V
		TOTAL	V^2	TOTAL	E lg V

best for dense graphs

best for sparse graphs

Dijkstra's algorithm: advanced implementations

Johnson (1970s): use d-way heap (easy)
Sleator-Tarjan (1980s): use Fibonacci heap (very complicated)

		d-way		Fibonacci	
	frequency	each op	total	each op	total
insert	V	d $\lg_d$ V	V d $\lg_d$ V	lg V	V lg V
delete-min	V	d $\lg_d$ V	V d $\lg_d$ V	lg V	V lg V
contains	V	1	V	V	V
decrease val	E	$\lg_d$ V	E $\lg_d$ V	lg V	E lg V
is empty	V	1	V	V	V
		TOTAL	E $\lg_d$ V	TOTAL	E + V lg V

indistinguishable from linear in practice

amortized bound

Linear worst-case guarantee? Open.

Dijkstra's Algorithm: performance summary

Fringe implementation directly impacts performance

Best choice depends on sparsity of graph.

- 2,000 vertices, 1 million edges. heap 2-3x slower than array
- 100,000 vertices, 1 million edges. heap gives 500x speedup.
- 1 million vertices, 2 million edges. heap gives 10,000x speedup.

Bottom line.

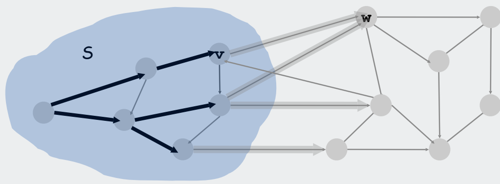
- array implementation optimal for dense graphs
- binary heap far better for sparse graphs
- d-way heap worth the trouble in performance-critical situations
- Fibonacci heap best in theory, but not worth implementing

Priority-first search

Insight: All of our graph-search methods are the same algorithm!

Maintain a set of explored vertices S
Grow S by exploring edges with exactly one endpoint leaving S.

DFS. Take edge from vertex which was discovered most recently.
BFS. Take from vertex which was discovered least recently.
Prim. Take edge of minimum weight.
Dijkstra. Take edge to vertex that is closest to `s`.
... Gives simple algorithm for many graph-processing problems



Challenge: express this insight in usable Java code

Priority-first search: application example

Shortest `s-t` paths in Euclidean graphs (maps)
- Vertices are points in the plane.
- Edge weights are Euclidean distances.

Sublinear algorithm.
- Assume graph is already in memory.
- Start Dijkstra at `s`.
- Stop when you reach `t`.

Even better: exploit geometry (A* algorithm)
- For edge `v-w`, use weight `d(v, w) + d(w, t) – d(v, t)`.
- Proof of correctness for Dijkstra still applies.
- In practice only O($V^{1/2}$) vertices examined.

Euclidean distance

[Practical map-processing programs precompute many of the paths.]

Shortest paths application: Currency conversion

Currency conversion. Given currencies and exchange rates, what is best way to convert one ounce of gold to US dollars?
- 1 oz. gold ⇒ $327.25.
- 1 oz. gold ⇒ £208.10 ⇒ ⇒ $327.00. [208.10 × 1.5714]
- 1 oz. gold ⇒ 455.2 Francs ⇒ 304.39 Euros ⇒ $327.28.

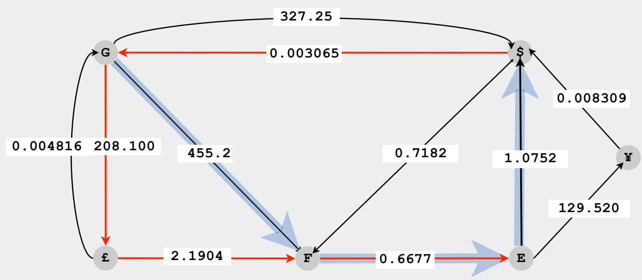
[455.2 × .6677 × 1.0752]

Currency	£	Euro	¥	Franc	$	Gold
UK Pound	1.0000	0.6853	0.005290	0.4569	0.6368	208.100
Euro	1.4599	1.0000	0.007721	0.6677	0.9303	304.028
Japanese Yen	189.050	129.520	1.0000	85.4694	120.400	39346.7
Swiss Franc	2.1904	1.4978	0.011574	1.0000	1.3929	455.200
US Dollar	1.5714	1.0752	0.008309	0.7182	1.0000	327.250
Gold (oz.)	0.004816	0.003295	0.0000255	0.002201	0.003065	1.0000

Shortest paths application: Currency conversion

Graph formulation.

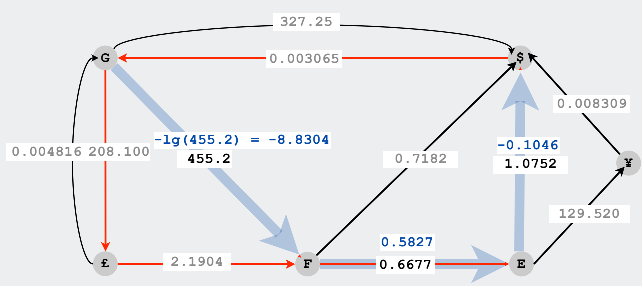
- Vertex = currency.
- Edge = transaction, with weight equal to exchange rate.
- Find path that maximizes product of weights.



Shortest paths application: Currency conversion

Reduce to shortest path problem by taking logs

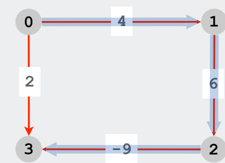
- Let weight(v-w) = – lg (exchange rate from currency v to w)
- multiplication turns to addition
- Shortest path with costs c corresponds to best exchange sequence.



Challenge. Solve shortest path problem with negative weights.

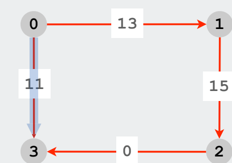
Shortest paths with negative weights: failed attempts

Dijkstra. Doesn't work with negative edge weights.



Dijkstra selects vertex 3 immediately after 0.
But shortest path from 0 to 3 is 0→1→2→3.

Re-weighting. Adding a constant to every edge weight also doesn't work.

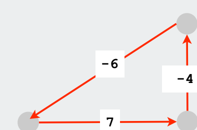


Adding 9 to each edge changes the shortest path because it adds 9 to each segment, wrong thing to do for paths with many segments.

Bad news: need a different algorithm.

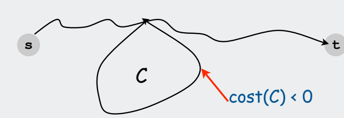
Shortest paths with negative weights: negative cycles

Negative cycle. Directed cycle whose sum of edge weights is negative.



Observations.

- If negative cycle C on path from s to t, then shortest path can be made arbitrarily negative by spinning around cycle
- There exists a shortest s-t path that is simple.



Worse news: need a different problem

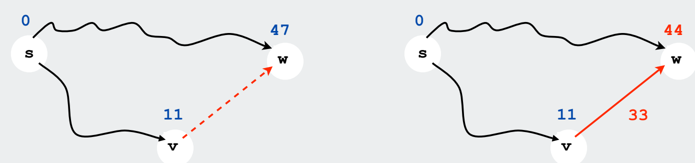
Edge relaxation

For all `v`, `dist[v]` is the length of some path from `s` to `v`.

Relaxation along edge `e` from `v` to `w`.

- `dist[v]` is length of some path from `s` to `v`
- `dist[w]` is length of some path from `s` to `w`
- if `v-w` gives a shorter path to `w` through `v`, update `dist[w]` and `pred[w]`

```
if (dist[w] > dist[v] + e.weight)
{   dist[w] = dist[v] + e.weight);   pred[w] = v;   }
```



Relaxation sets `dist[w]` to the length of a shorter path from `s` to `w` (if `v-w` gives one)

Shortest paths with negative weights: dynamic programming algorithm

A simple solution that works!

- Initialize `dist[v] = ∞`, `dist[s]= 0`.
- Repeat v times: relax each edge `e`.

phase i

```
for (int i = 1; i <= G.V(); i++)
   for (int v = 0; v < G.V(); v++)
      for (Edge e : G.adj(v))
      {
         int w = e.target;
         if (dist[w] > dist[v] + e.weight)
         {
              dist[w] = dist[v] + e.weight)
              pred[w] = v;
         }
      }
```

relax v-w

Shortest paths with negative weights: dynamic programming algorithm

Running time proportional to E V

Invariant. At end of phase `i`, `dist[v]` ≤ length of any path from `s` to `v` using at most `i` edges.

Theorem. If there are no negative cycles, upon termination `dist[v]` is the length of the shortest path from from `s` to `v`.

and `pred[]` gives the shortest paths

Shortest paths with negative weights: Bellman-Ford-Moore algorithm

Observation. If `dist[v]` doesn't change during phase `i`, no need to relax any edge leaving `v` in phase `i+1`.

FIFO implementation. Maintain queue of vertices whose distance changed.

be careful to keep at most one copy of each vertex on queue

Running time.

- still could be proportional to EV in worst case
- much faster than that in practice

Shortest paths with negative weights: Bellman-Ford-Moore algorithm

Initialize `dist[v] = ∞` and `marked[v]= false` for all vertices `v`.

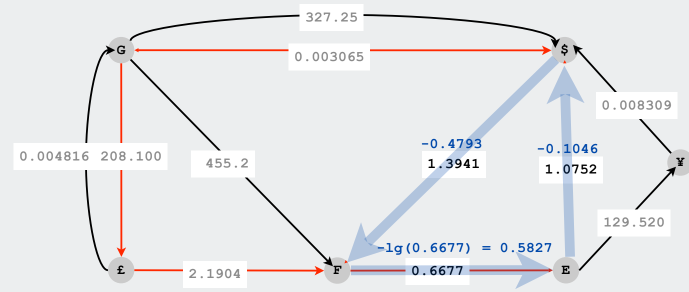
```
Queue<Integer> q = new Queue<Integer>();
marked[s] = true;
dist[s] = 0;
q.enqueue(s);


while (!q.isEmpty())
{
   int v = q.dequeue();
   marked[v] = false;
   for (Edge e : G.adj(v))
   {
      int w = e.target();
      if (dist[w] > dist[v] + e.weight)
      {
          dist[w] = dist[v] + e.weight;
          pred[w] = v;
          if (!marked[w])
          {
             marked[w] = true;
             q.enqueue(w);
          }
      }
   }
}
```


Shortest paths application: arbitrage

Is there an arbitrage opportunity in currency graph?

- Ex: \$1 ⇒ 1.3941 Francs ⇒ 0.9308 Euros ⇒ \$1.00084.
- Is there a negative cost cycle?
- Fastest algorithm very valuable!



`-0.4793 + 0.5827 - 0.1046 < 0`

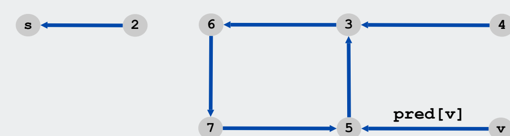
Single Source Shortest Paths Implementation: Cost Summary

	algorithm	worst case	typical case
nonnegative costs	Dijkstra (classic)	V^2	V^2
	Dijkstra (heap)	E lg V	E
no negative cycles	Dynamic programming	EV	EV
	Bellman-Ford-Moore	EV	E

Remark 1. Negative weights makes the problem harder.
Remark 2. Negative cycles makes the problem intractable.

Negative cycle detection

If there is a negative cycle reachable from s.
Bellman-Ford-Moore gets stuck in loop, updating vertices in cycle.

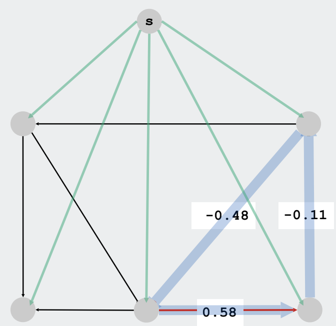


Finding a negative cycle. If any vertex `v` is updated in phase v, there exists a negative cycle, and we can trace back `pred[v]` to find it.

Negative cycle detection

Goal. Identify a negative cycle (reachable from any vertex).

Solution. Add 0-weight edge from artificial source `s` to each vertex `v`.
Run Bellman-Ford from vertex `s`.



Shortest paths summary

Dijkstra's algorithm
- easy and optimal for dense digraphs
- PQ/ST data type gives near optimal for sparse graphs

Priority-first search
- generalization of Dijkstra's algorithm
- encompasses DFS, BFS, and Prim
- enables easy solution to many graph-processing problems

Negative weights
- arise in applications
- make problem intractable in presence of negative cycles (!)
- easy solution using old algorithms otherwise

Shortest-paths is a broadly useful problem-solving model