# Shortest Paths

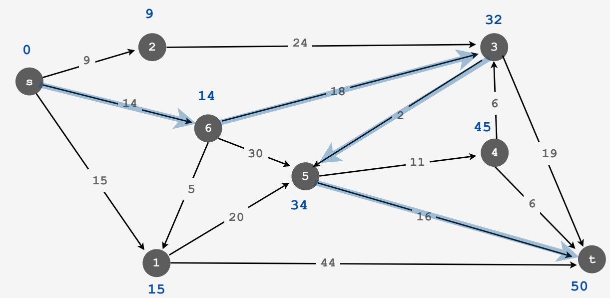
- Dijkstra's algorithm
- implementation
- negative weights

References:
Algorithms in Java, Chapter 21
`http://www.cs.princeton.edu/algs4/54sp`



## Shortest paths in a weighted digraph

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




shortest path: s→6→3→5→t
cost: 14 + 18 + 2 + 16 = 50

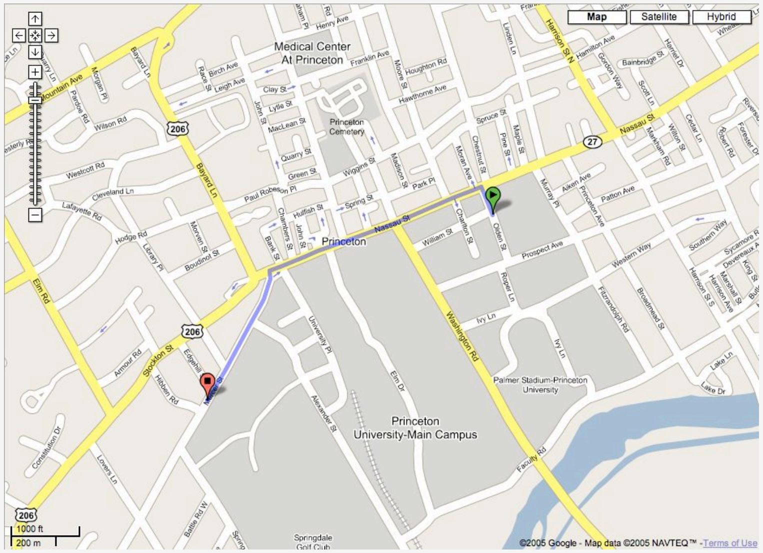


## Versions

- Source-target (s→t).
- Single source.
- All pairs.
- Nonnegative edge weights.
- Arbitrary weights.
- Euclidean weights.



## Google maps



## Applications

- 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.



## 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.



- **Dijkstra's algorithm**
- implementation
- negative weights



## Edsger W. Dijkstra: select quote

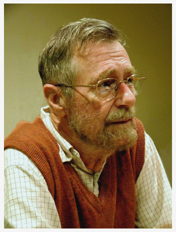
*" The question of whether computers can think is like the question of whether submarines can swim. "*

*" 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. "*



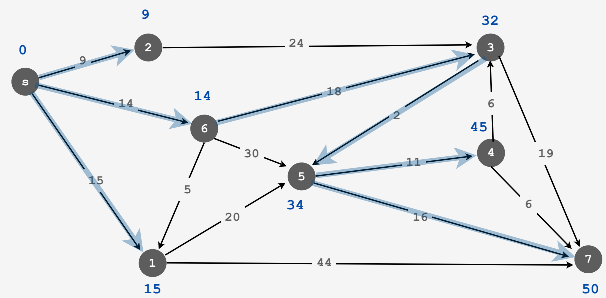
Edger Dijkstra
Turing award 1972

## Single-source shortest-paths

Given. Weighted digraph G, source vertex s.

Goal. Find shortest path from s to every other vertex.

Observation. Use parent-link representation to store shortest path tree.



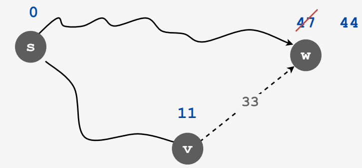

| v | s | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| dist[v] | 0 | 15 | 9 | 32 | 45 | 34 | 14 | 50 |
| pred[v] | 0 | 0 | 0 | 6 | 5 | 3 | 0 | 5 |



## Edge relaxation

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].

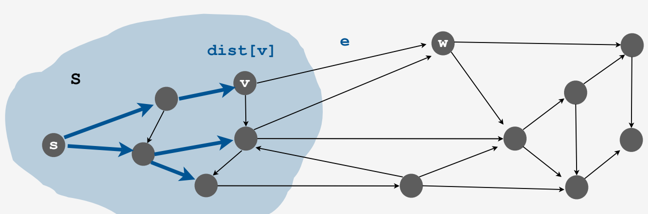



```
int v = e.from(), w = e.to();
if (dist[w] > dist[v] + e.weight())
{
    dist[w] = dist[v] + e.weight());
    pred[w] = e;
}
```



## Dijkstra's algorithm

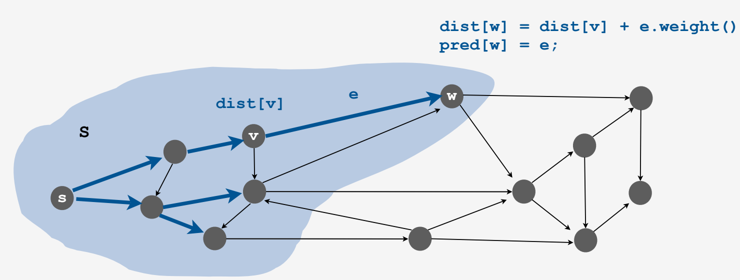
- Initialize S to s, dist[s] to 0, dist[v] to ∞ for all other v.
- Repeat until S contains all vertices connected to s:
  - find edge e with v in S and w not in S that minimizes dist[v] + e.weight().
  - relax along edge e
  - add w to S



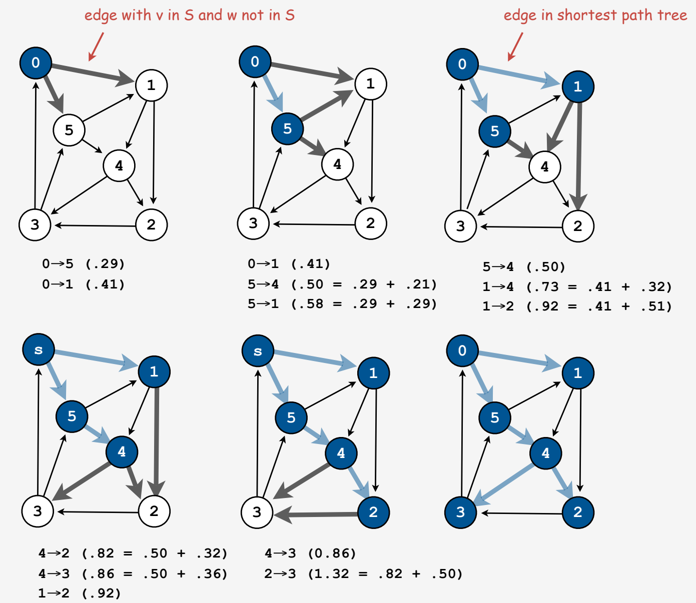



## Dijkstra's algorithm

- 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 edge e with v in S and w not in S that minimizes dist[v] + e.weight().
  - relax along edge e
  - add w to S



## Dijkstra's algorithm example




| | |
|---|---|
| 0→1 | .41 |
| 0→5 | .29 |
| 1→2 | .51 |
| 1→4 | .32 |
| 2→3 | .50 |
| 3→0 | .45 |
| 3→5 | .38 |
| 4→2 | .32 |
| 4→3 | .36 |
| 5→1 | .29 |
| 5→4 | .21 |

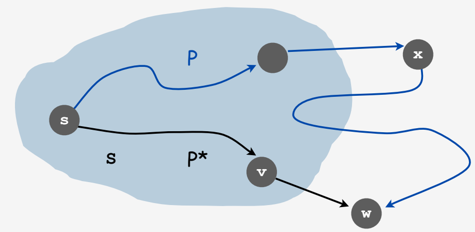


## Dijkstra's algorithm: correctness proof

Invariant. For v in S, `dist[v]` is the length of the shortest path from s to v.

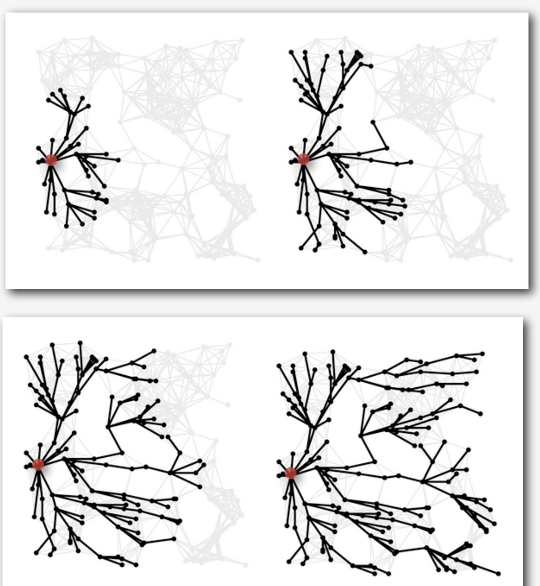
Pf. (by induction on |S|)

- 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.
- Thus, `dist[w]` is the length of the shortest path from s to w.






## Shortest path trees



25% 50%

75% 100%



- Dijkstra's algorithm
- **implementation**
- negative weights

## Weighted directed graph API

| public class DirectedEdge implements Comparable<Edge> | | |
|---|---|---|
| | DirectedEdge(int v, int w, double weight) | *create a weighted edge v→w* |
| int | from() | *vertex v* |
| int | to() | *vertex w* |
| double | weight() | *the weight* |
| String | toString() | *string representation* |

| public class WeightedDigraph | | *weighted digraph data type* |
|---|---|---|
| | WeightedDigraph(int V) | *create an empty digraph with V vertices* |
| | WeightedDigraph(In in) | *create a digraph from input stream* |
| void | insert(DirectedEdge e) | *add an edge from v to w* |
| Iterable<DirectedEdge> | adj(int v) | *return an iterator over edges leaving v* |
| int | V() | *return number of vertices* |
| String | toString() | *return a string representation* |



## Weighted digraph: adjacency-set implementation

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

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

   public void addEdge(DirectedEdge e)
   {
      int v = e.from();
      adj[v].add(e);
   }

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

}
```

same as `Edge`, but only add edge to v's adjacency set



## Weighted directed edge: Java implementation

```
public class DirectedEdge implements Comparable<DirectedEdge>
{
   private final int v, w;
   private final double weight;

   public DirectedEdge(int v, int w, double weight)
   {
      this.v = v;
      this.w = w;
      this.weight = weight;
   }

   public int from() {  return v;  }
   public int to()   {  return w;  }

   public int weight()
   {  return weight;  }

   public int compareTo(Edge that)
   {
      if (this.v < that.v) return -1;
      if (this.v > that.v) return +1;
      if (this.w < that.w) return -1;
      if (this.w > that.w) return +1;
      return 0;
    }
}
```

same as `Edge`, except `from()` and `to()` replace `either()` and `other()`



## Shortest path data type

Design pattern.

- `Dijkstra` class is a `WeightedDigraph` client.
- Client query methods return distance and path iterator.

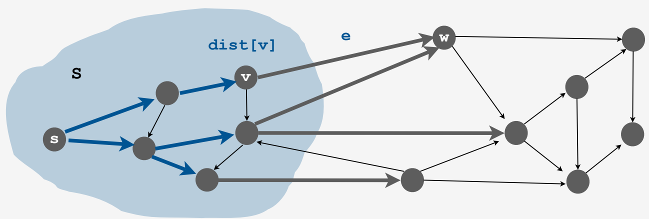
| public class Dijkstra | | |
|---|---|---|
| | Dijkstra(WeightedDigraph G, int s) | *shortest path from s in graph G* |
| double | distance(int v) | *length of shortest path from s to v* |
| Iterable <DirectedEdge> | path(int v) | *shortest path from s to v* |

## Dijkstra implementation challenge

Find edge `e` with `v` in S and `w` not in S that minimizes `dist[v] + e.weight()`.

### How difficult?

- Intractable.
- O(E) time. ← try all edges
- O(V) time. ← Dijkstra with an array priority queue
- O(log V) time. ← Dijkstra with a binary heap
- O(log* V) time.
- Constant time.

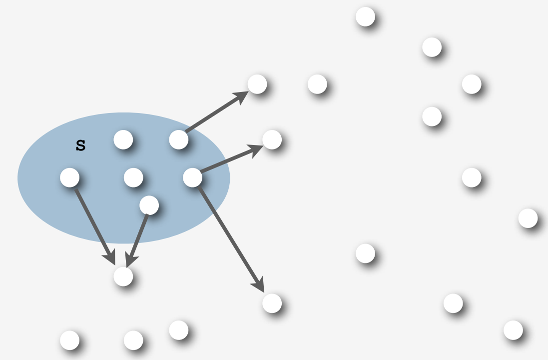





## Dijkstra's algorithm implementation

Q. What goes onto the priority queue?

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




Starting to look familiar?



## Dijkstra's algorithm: implementation approach

### Maintain these invariants.

- For `v` in S, `dist[v]` is the length of the shortest path from `s` to `v`.
- For `w` not in S, `dist[w]` minimizes `dist[v] + e.weight()` among all edges `e` with `w` in S.
- PQ contains vertices not in S, with `dist[w]` as priority.

### Implications.

- To find next vertex `v` to add to S, delete the vertex with min `dist[]`.
- To maintain invariants, update `dist[]` by relaxing all edges leaving `v`.

**Total running time.** Depends on PQ implementation.

- Exactly V `delMin()` operations.
- At most E `insert()` operations.



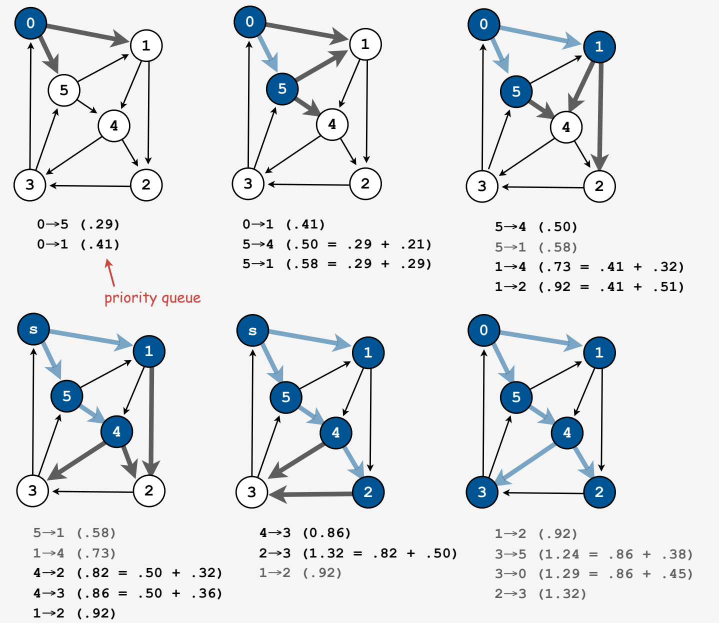
## Lazy implementation of Dijkstra's SPT algorithm

```
public class LazyDijkstra {
   private double[] dist = new double[G.V()];
   private Edge[] pred = new Edge[G.V()];
   public LazyDijkstra(WeightedDigraph G, int s) {
      boolean[] marked = new boolean[G.V()];
      for (int v = 0; v < G.V(); v++)
         dist[v] = Double.POSITIVE_INFINITY;
      MinPQplus<Double, Integer> pq;
      pq = new MinPQplus<Double, Integer>();
      dist[s] = 0.0;
      pq.put(dist[s], s);
      while (!pq.isEmpty())
      {
         int v = pq.delMin();
         if (marked[v]) continue;
         marked(v) = true;
         for (Edge e : G.adj(v)) {
            int w = e.to();
            if (dist[w] > dist[v] + e.weight()) {
                dist[w] = dist[v] + e.weight();
                pred[w] = e;
                pq.insert(dist[w], w);
            }
         }
      }
   }
}
```

code is the same as Prim's (!!) except weight is distance to s, not to tree

← relax edge e

## Lazy Dijkstra's algorithm example




| | |
|---|---|
| 0→1 | .41 |
| 0→5 | .29 |
| 1→2 | .51 |
| 1→4 | .32 |
| 2→3 | .50 |
| 3→0 | .45 |
| 3→5 | .38 |
| 4→2 | .32 |
| 4→3 | .36 |
| 5→1 | .29 |
| 5→4 | .21 |



## Dijkstra's algorithm: which priority queue?

Running time of Dijkstra depends on priority queue implementation.

| PQ implementation | insert | delmin | total |
|---|---|---|---|
| array | 1 | V | $V^2$ |
| binary heap | log V | log V | E log V |
| d-way heap (Johnson) | $\log_d V$ | $\log_d V$ | $E \log_d V$ |
| Fibonacci heap (Sleator-Tarjan) | 1 | V | E + V log V |

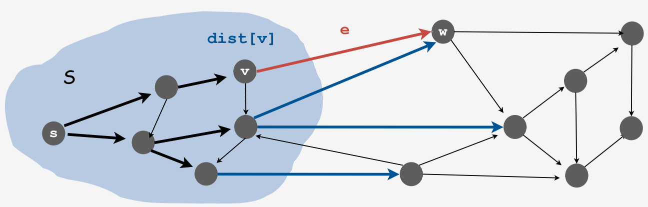
Bottom line.
- Array implementation optimal for dense graphs.
- Binary heap far better for sparse graphs.
- d-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.



## 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.




Challenge. Express this insight in (re)usable Java code.



- Dijkstra's algorithm
- implementation
- negative weights

## 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 ]

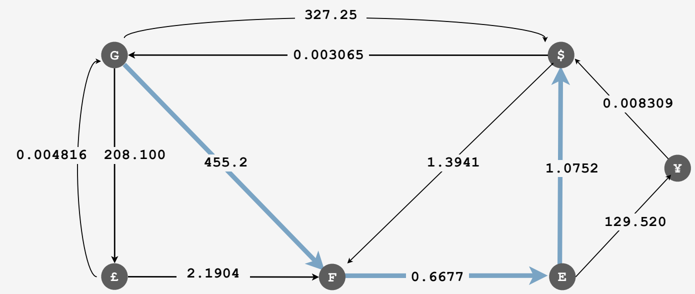
| currency | £ | Euro | ¥ | Franc | $ | Gold |
|---|---|---|---|---|---|---|
| UK pound | 1.0000 | 0.6853 | 0.005290 | 0.4569 | 0.6368 | 208.100 |
| Euro | 1.45999 | 1.0000 | 0.007721 | 0.6677 | 0.9303 | 304.028 |
| Japanese Yen | 189.50 | 129.520 | 1.0000 | 85.4694 | 120.400 | 39346.7 |
| Swiss Franc | 2.1904 | 1.4978 | 0.01574 | 1.0000 | 1.3941 | 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 |



## Currency conversion

Graph formulation.

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

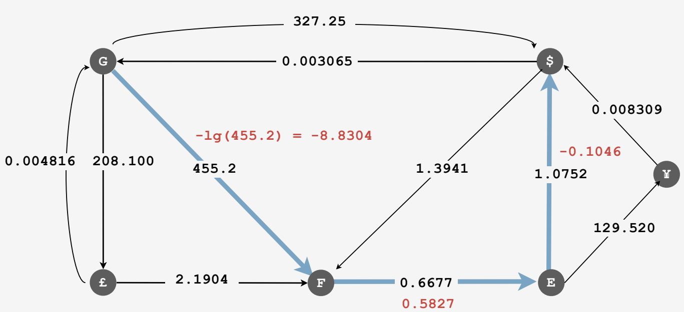





## Currency conversion

Reduce to shortest path problem by taking logs.

- Let weight of edge v→w be - lg (exchange rate from currency v to w).
- Multiplication turns to addition.
- Shortest path with given weights corresponds to best exchange sequence.

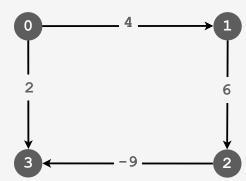



Challenge. Solve shortest path problem with negative weights.



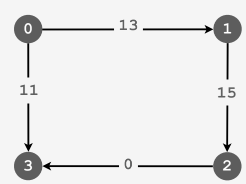
## Shortest paths with negative weights: failed attempts

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. Add a constant to every edge weight also doesn't work.

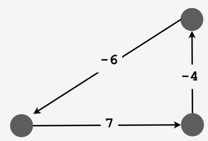



Adding 9 to each edge changes the shortest path because it adds 9 to each edge; wrong thing to do for paths with many edges.

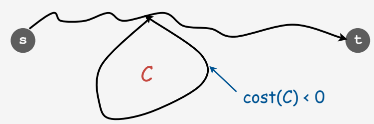
Bad news. Need a different algorithm.

## Negative cycles

**Def.** A negative cycle is a directed cycle whose sum of edge weights is negative.




**Observations.** If negative cycle $C$ is on a path from `s` to `t`, then shortest path can be made arbitrarily negative by spinning around cycle.



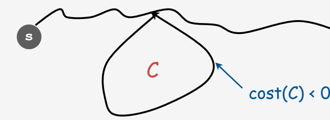

**Worse news.** Need a different problem.



## Shortest paths with negative weights

**Problem 1.** Does a given digraph contain a negative cycle?

**Problem 2.** Find the shortest simple path from `s` to `t`.




**Bad news.** Problem 2 is intractable.

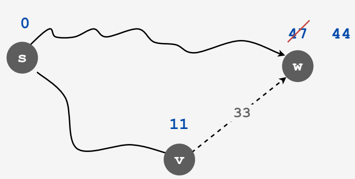
**Good news.** Can solve problem 1 in O(VE) steps; if no negative cycles, can solve problem 2 with same algorithm!



## Edge relaxation (review)

**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] = e;
}
```



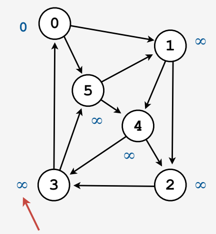
## Shortest paths with negative weights: dynamic programming algorithm

**A simple solution that works!**

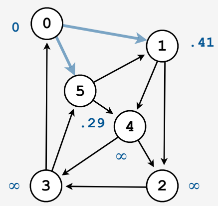
- Initialize `dist[v] = ∞`, `dist[s]= 0`.
- Repeat `v` times: relax each edge `e`.

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

## Dynamic programming algorithm trace

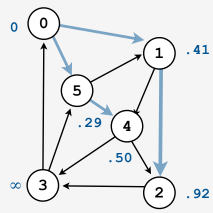






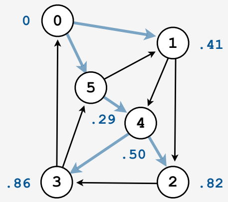
```
0→1 (.41 = 0 + .41)
0→5 (.50 = 0 + .50)
```

relaxed edges that update dist[]

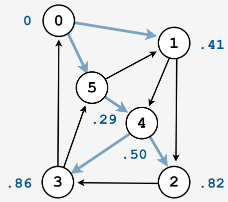


```
1→2 (.92 = .41 + .51)
1→4 (.73 = .41 + .32)
5→4 (.50 = .29 + .21)
```

| | |
|---|---|
| 0→1 | .41 |
| 0→5 | .29 |
| 1→2 | .51 |
| 1→4 | .32 |
| 2→3 | .50 |
| 3→0 | .45 |
| 3→5 | .38 |
| 4→2 | .32 |
| 4→3 | .36 |
| 5→1 | .29 |
| 5→4 | .21 |



```
2→3 (1.33 = .83 + .50)
4→3 (.86 = .50 + .36)
4→2 (.82 = .50 + .32)
```



can stop early since no entries in dist[] updated



## 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.

Proposition. 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



## 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.

- Proportional to EV in worst case.
- Much faster than that in practice.



## Single source shortest paths implementation: cost summary

| | algorithm | worst case | typical case |
|---|---|---|---|
| nonnegative costs | Dijkstra (array heap) | $V^2$ | $V^2$ |
| | Dijkstra (binary heap) | $E \lg V$ | $E$ |
| no negative cycles | dynamic programming | $E\,V$ | $E\,V$ |
| | Bellman-Ford | $E\,V$ | $E$ |

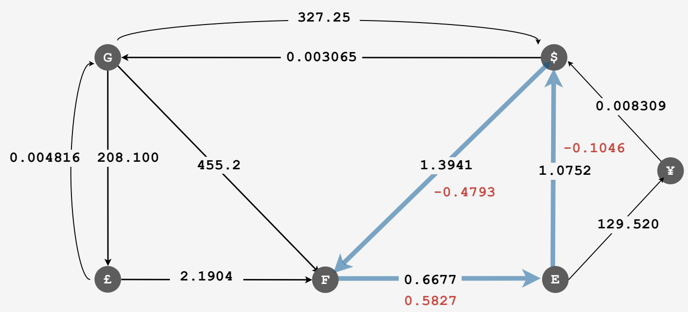
Remark 1. Negative weights makes the problem harder.

Remark 2. Negative cycles makes the problem intractable.

## Shortest paths application: arbitrage

Is there an arbitrage opportunity in currency graph?

- Ex: \$1 $\Rightarrow$ 1.3941 Francs $\Rightarrow$ 0.9308 Euros $\Rightarrow$ \$1.00084.
- Is there a negative cost cycle?




0.5827 - 0.1046 - 0.4793 < 0

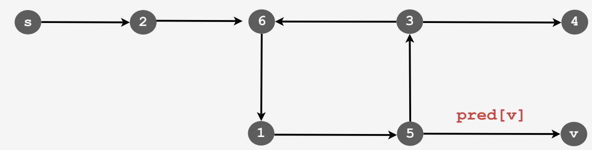
Remark. Fastest algorithm is valuable!



## Negative cycle detection

If there is a negative cycle reachable from s.

Bellman-Ford-Moore gets stuck in loop, updating vertices in cycle.

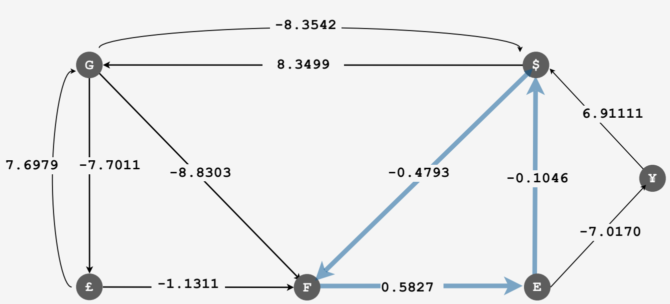



Finding a negative cycle. If any vertex `v` is updated in phase `v`, there exists a negative cycle, and we can trace back `pred[v]` to find it.



## Negative cycle detection

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




Solution. Initialize Bellman-Ford by setting `dist[v] = 0` for all vertices `v`.



## Shortest paths summary

Dijkstra's algorithm.

- Easy and optimal for dense digraphs.
- PQ yields 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.