Shortest Paths

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

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

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.

Edsger 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 $t$.

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

Ford (1956). RAND, economics of transportation.

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.

**Single-source shortest-paths**

*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

**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, \( \text{dist}[w] \) is the length of shortest known path from s to w.
- for v in S, \( \text{dist}[v] \) is the length of the shortest path from s to v.

Initialize S to s, \( \text{dist}[s] \) to 0, \( \text{dist}[v] \) to \( \infty \) 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 \( \text{dist}[v] + \text{weight}[v-w] \)
  - 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, \( \text{dist}[v] \) is the length of shortest known path from s to v.
- for v in S, \( \text{dist}[v] \) 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 \( \text{P}^* \) be the s-w path through v.
- Consider any other s-w path \( \text{P} \), and let \( x \) be first node on path outside S.
- \( \text{P} \) is already longer than \( \text{P}^* \) as soon as it reaches \( x \) by greedy choice.
Shortest Path Tree

Dijkstra's algorithm implementation approach

Initialize $S$ to $s$, $\text{dist}[s]$ to 0, $\text{dist}[v]$ to $\infty$ 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 $\text{dist}[v] + \text{weight}[v-w]$
  * relax along that edge
  * add $w$ to $S$

Idea 1 (easy): Try all edges

Total running time proportional to $VE$

Dijkstra's algorithm implementation

Initialize $S$ to $s$, $\text{dist}[s]$ to 0, $\text{dist}[v]$ to $\infty$ 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 $\text{dist}[v] + \text{weight}[v-w]$
  * relax along that edge
  * add $w$ to $S$

Idea 2 (Dijkstra):
  * for all $v$ in $S$, $\text{dist}[v]$ is the length of the shortest path from $s$ to $w$.
  * for all $w$, $\text{dist}[w]$ 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 $\text{dist}[]$)
  * 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 approach

Dijkstra's algorithm implementation

Initialize $S$ to $s$, $\text{dist}[s]$ to 0, $\text{dist}[v]$ to $\infty$ 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 $\text{dist}[v] + \text{weight}[v-w]$
  * relax along that edge
  * add $w$ to $S$

Idea 3 (this lecture):
  * for all $v$ in $S$, $\text{dist}[v]$ 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 $E \lg V$
Q. What goes onto the priority queue?
A. Fringe vertices connected by a single edge to a vertex in S

Dijkstra's algorithm implementation

Weighted digraphs in Java: weighted edge data type

```java
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

```java
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

```java
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.
    }
}
```

Designing a data type for the fringe

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

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

```java
public class Fringe {
    public void insert(int i, Value val) {
        // add item i with given value
    }
    public void decrease(int i, Value val) {
        // decrease value of item i
    }
    public int delMin() {
        // delete and return smallest item
    }
    public boolean isEmpty() {
        // is the fringe empty?
    }
    public 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

**Challenge:**
- How can we design a data type for the fringe that supports all operations in logarithmic time?
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

<table>
<thead>
<tr>
<th></th>
<th>cost</th>
<th>frequency</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>delete-min</td>
<td>V</td>
<td>V</td>
<td>V^2</td>
</tr>
<tr>
<td>contains</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>decrease val</td>
<td>1</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>is empty</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>TOTAL</td>
<td>V</td>
<td>V</td>
<td>V^2</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>pq</th>
<th>qp</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>47</td>
<td>78</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>10</td>
<td>47</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>91</td>
<td>34</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>42</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>83</td>
<td>78</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>4</td>
<td>78</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>11</td>
<td>77</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>10</td>
<td>81</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td></td>
<td>45</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>cost</th>
<th>frequency</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>lg V</td>
<td>V</td>
<td>V lg V</td>
</tr>
<tr>
<td>delete-min</td>
<td>lg V</td>
<td>V</td>
<td>V lg V</td>
</tr>
<tr>
<td>contains</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>decrease val</td>
<td>lg V</td>
<td>E</td>
<td>E lg V</td>
</tr>
<tr>
<td>is empty</td>
<td>1</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>TOTAL</td>
<td>E</td>
<td>lg V</td>
<td>E lg V</td>
</tr>
</tbody>
</table>

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;
   }
}
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

Indirect heap for fringe: Java code for ST operations

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

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

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

Indirect heap for fringe: Java code for PQ operations

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

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

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

Indirect heap for fringe: Java code for ST operations

```java
public class Fringe
{
   private int N; 
   ... j)
   {
      int swap = pq[i]; pq[i] = pq[j]; pq[j] = swap;
      qp[pq[i]] = i; qp[qp[i]] = j;
   }
```

```java
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[qp[i]] = j;
   }
```
Dijkstra’s algorithm: performance

Fringe implementation directly impacts performance.

<table>
<thead>
<tr>
<th>Frequency of operation</th>
<th>Array implementation</th>
<th>Indirect heap implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>each op</td>
<td>total</td>
</tr>
<tr>
<td>insert</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>delete-min</td>
<td>V</td>
<td>V^2</td>
</tr>
<tr>
<td>contains</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>decrease val</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>is empty</td>
<td>V</td>
<td>V</td>
</tr>
</tbody>
</table>

**Total**

- Best for dense graphs
- Best for sparse graphs

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.

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

---

**Currency conversion.** Given currencies and exchange rates, what is best way to convert one ounce of gold to US dollars?

- 1 oz. gold \( \Rightarrow \) $327.25.
- 1 oz. gold \( \Rightarrow \) £208.10 \( \Rightarrow \) $327.00.
- 1 oz. gold \( \Rightarrow \) 455.2 Francs \( \Rightarrow \) 304.39 Euros \( \Rightarrow \) $327.28.

**Currency conversion table:**

<table>
<thead>
<tr>
<th>Currency</th>
<th>£</th>
<th>Euro</th>
<th>¥</th>
<th>Franc</th>
<th>$</th>
<th>Gold (oz.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UK Pound</td>
<td>1.0000</td>
<td>0.4853</td>
<td>0.005290</td>
<td>0.4569</td>
<td>0.6368</td>
<td>208.100</td>
</tr>
<tr>
<td>Euro</td>
<td>1.4599</td>
<td>1.0000</td>
<td>0.007721</td>
<td>0.6677</td>
<td>0.9303</td>
<td>304.028</td>
</tr>
<tr>
<td>Japanese Yen</td>
<td>189.050</td>
<td>129.520</td>
<td>1.0000</td>
<td>85.4694</td>
<td>120.400</td>
<td>39346.7</td>
</tr>
<tr>
<td>Swiss Franc</td>
<td>2.1904</td>
<td>1.4978</td>
<td>0.011574</td>
<td>1.0000</td>
<td>1.3929</td>
<td>455.200</td>
</tr>
<tr>
<td>US Dollar</td>
<td>1.5714</td>
<td>1.0752</td>
<td>0.008309</td>
<td>0.7182</td>
<td>1.0000</td>
<td>327.250</td>
</tr>
<tr>
<td>Gold (oz.)</td>
<td>0.004816</td>
<td>0.003295</td>
<td>0.000255</td>
<td>0.002201</td>
<td>0.003065</td>
<td>1.0000</td>
</tr>
</tbody>
</table>
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.

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 \( v \), \( \text{dist}[v] \) is the length of some path from \( s \) to \( v \).

Relaxation along edge \( e \) from \( v \) to \( w \).

- \( \text{dist}[v] \) is length of some path from \( s \) to \( v \)
- \( \text{dist}[w] \) is length of some path from \( s \) to \( w \)
- if \( v-w \) gives a shorter path to \( w \) through \( v \), update \( \text{dist}[w] \) and \( \text{pred}[w] \)

Relaxation sets \( \text{dist}[w] \) 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 \( \text{dist}[v] = \infty \), \( \text{dist}[s] = 0 \).
- Repeat \( V \) times: relax each edge \( e \).

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

Running time proportional to \( E \ V \)

Invariant. At end of phase \( i \), \( \text{dist}[v] \leq \) length of any path from \( s \) to \( v \) using at most \( i \) edges.

Theorem. If there are no negative cycles, upon termination \( \text{dist}[v] \) is the length of the shortest path from \( s \) to \( v \).

and \( \text{pred}[\_] \) gives the shortest paths

Observation. If \( \text{dist}[v] \) 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 \( E \ V \) in worst case
- much faster than that in practice
Shortest paths with negative weights: Bellman-Ford-Moore algorithm

Initialize dist[v] = \infty and marked[v] = false for all vertices v.

```java
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: $1 \rightarrow 1.3941$ Francs $\rightarrow 0.9308$ Euros $\rightarrow 1.0008$
- Is there a negative cost cycle?
- Fastest algorithm very valuable!

![Currency graph](image)

-0.4793 + 0.5827 - 0.1046 < 0

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.

Single Source Shortest Paths Implementation: Cost Summary

<table>
<thead>
<tr>
<th>algorithm</th>
<th>worst case</th>
<th>typical case</th>
</tr>
</thead>
<tbody>
<tr>
<td>nonnegative costs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dijkstra (classic)</td>
<td>(V^2)</td>
<td>(V^2)</td>
</tr>
<tr>
<td>Dijkstra (heap)</td>
<td>(E \lg V)</td>
<td>(E)</td>
</tr>
<tr>
<td>Dynamic programming</td>
<td>(EV)</td>
<td>(EV)</td>
</tr>
<tr>
<td>Bellman-Ford-Moore</td>
<td>(EV)</td>
<td>(E)</td>
</tr>
</tbody>
</table>

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