Classic algorithms for natural network problems

**SHORTEST PATH**
- shortest way to get from u to v

**SINGLE-SOURCE SHORTEST PATHS (SPT)**
- PFS implementation
- Dijkstra’s algorithm

**ALL SHORTEST PATHS**
- Floyd’s algorithm

Negative weights?

**REDUCTION**

Problem-solving models
Single-source shortest paths

Defines SHORTEST PATHS TREE (SPT) rooted at source

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

\[
\begin{array}{ccccccc}
\text{st} & 0 & 0 & 4 & 4 & 5 & 0 \\
\text{wt} & 0 & .41 & .32 & .36 & .21 & .29 \\
\end{array}
\]
Another generalized graph-search implementation

RELAXATION

- if \( wt[w] < wt[v] + wt(v-w) \) then set \( wt[w] \) to that value

(v-w gives a shorter path to w than the best known)

SPT ALGORITHM

1. put \( s \) on fringe
2. while fringe nonempty
3. choose node from fringe that is closest to \( s \)
4. relax along all its edges

v on TREE \( wt[v] \) is shortest distance from \( s \) to \( v \)
v on FRINGE: \( wt[v] \) is shortest KNOWN distance from \( s \) to \( v \)

- won't find a shorter path to node with smallest value
larger SPT example
Dijkstra's algorithm

Classical implementation of generic SPT algorithm

SAME CODE as Prim's MST algorithm with

```c
#define P wt[v] + t->wt
```

DENSE graphs
- classical Dijkstra's algorithm
- time cost: $O(V^3)$

SPARSE graphs
- use PQ (heap) implementation
- time cost: $O(E \log V)$

Better PQs give faster algorithms for sparse graphs
- d-way heap: $O(E \log_d V)$
- F-heap: $O(E + V \log V)$
Shortest paths in Euclidean graphs

Problem: find shortest path from s to d

Algorithm:
- start shortest-path PFS at s
- stop when reaching d

SUBLINEAR algorithm
- need not touch all nodes

better yet: use geometry to limit search

wt[v]:
- TREE: shortest distance from s to v
- FRINGE: shortest POSSIBLE distance from s to d through v
  tree path from s to v PLUS distance from v to d

#define P wt[v] + t->wt + dist(t->v, d) - dist(k, d)
All shortest paths

Table of shortest paths for each vertex pair

**Ex:** map of New England

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>W</th>
<th>L</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Providence</td>
<td>0</td>
<td>53</td>
<td>54</td>
<td>48</td>
</tr>
<tr>
<td>Westerley</td>
<td>53</td>
<td>0</td>
<td>18</td>
<td>101</td>
</tr>
<tr>
<td>New London</td>
<td>54</td>
<td>18</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>Norwich</td>
<td>48</td>
<td>101</td>
<td>12</td>
<td>0</td>
</tr>
</tbody>
</table>

Norwich-Westerly: 101 miles??
- 12 miles Norwich-New London
- 18 miles New London-Westerly
- 30 miles total

Need correct algorithm to get correct table
Floyd's algorithm

Another ancient algorithm (1962)
[same as Warshall, in a different context]

Want shorter path from s to d?
  • take s to i, then i to d, if shorter (vertex relaxation)

```plaintext
for (i = 0; i < G->V; i++)
    for (s = 0; s < G->V; s++)
        if (G->adj[s][i] != maxWT)
            for (t = 0; t < G->V; t++)
                if (G->adj[i][t] != maxWT)
                    if (d[s][t] > d[s][i]+d[i][t])
                        d[s][t] = d[s][i]+d[i][t];
```

Correctness proof:
  • induction on i (same as Warshall)
Shortest paths ADT

Same issues as reachability in digraphs

Classical Floyd-Warshall algorithm gives
- query: $O(1)$
- preprocessing: $O(V^3)$
- space: $O(V^2)$

Easy to reduce preprocessing to $O(VE)$
- use Dijkstra for each vertex

End of story?

NOT QUITE
- ADT is useful for a variety of disparate problems
- negative weights complicate matters
**DEF:** Problem A REDUCES TO Problem B

if we can use an algorithm that solves B
to develop an algorithm that solves A

Typical reduction:
- given an instance of A
- transform it to an instance of B
- solve that instance of B
- transform the solution to be a solution of A

Uses of reduction
- algorithm for A (programmer using ADT)
- lower bound on B

**PROBLEM-SOLVING MODELS**
- problems that many other problems reduce to

**NP-HARD PROBLEMS**
- problems that ANY NP-hard problem reduces to
Reduction example: longest paths

**THM:** Longest-paths reduces to shortest-paths

**Proof:**
- given an instance of longest-paths
- transform it to shortest-paths by negating weights
- solve shortest-paths
- negate weights on path to get longest path

**CATCH**
- SP algs don’t work in the presence of negative weights!

**Lessons:**
- reductions have to be constructed with care
- they may not always give useful information
**Reduction example: arbitrage**

### Currency conversion

<table>
<thead>
<tr>
<th></th>
<th>dollars</th>
<th>pounds</th>
<th>1K yen</th>
</tr>
</thead>
<tbody>
<tr>
<td>dollars</td>
<td>1.000</td>
<td>1.631</td>
<td>0.669</td>
</tr>
<tr>
<td>pounds</td>
<td>0.613</td>
<td>1.000</td>
<td>0.411</td>
</tr>
<tr>
<td>1K yen</td>
<td>1.495</td>
<td>2.436</td>
<td>1.000</td>
</tr>
</tbody>
</table>

- $1000\ \text{dollars-pounds-dollars}$
  \[1000 \times (1.631) \times (0.613) = 999\]  
- $1000\ \text{dollars-pounds-yen-dollars}$
  \[1000 \times (1.631) \times (0.411) \times (1.495) = 1002\]

**SHORTEST PATH** is best arbitrage opportunity.

- replace table entry $x$ by $-\log x$
- BUT, weights may be negative!

Need SP algs that work with negative weights
Shortest paths with negative weights

Negative weights
- completely change SPT
- can introduce negative cycles

$$
\begin{array}{c|c}
0-1 & 0.41 \\
1-2 & 0.51 \\
2-3 & 0.50 \\
4-3 & 0.36 \\
3-5 & -0.38 \\
3-0 & 0.45 \\
0-5 & 0.29 \\
5-4 & 0.21 \\
1-4 & 0.32 \\
4-2 & 0.32 \\
5-1 & -0.29 \\
\end{array}
$$

shortest path from 4 to 2: 4-3-5-1-2
**Reduction example: SP with negative weights**

**THM:** SP with negative weights is NP-hard

**A:** Hamilton path
**B:** SP with negative weights

Hamilton path reduces to SP with negative weights

- given an undirected graph
- transform to network with -1 wt on each edge
- find shortest simple path
- YES to Hamilton path if SP length is \(-V\)
Negative weights in SP problems

NP-complete: don’t try to solve general problem
- restrict problem to solve it

Versions that we can solve
- no negative weights
- no cycles
- negative-cycle detection
- no negative cycles

Dijkstra’s algorithm: doesn’t work at all with negative weights
Floyd’s algorithm
- detects negative cycles
- solves all-pairs shortest paths if no neg cycles present

Ex: use Floyd’s to find SOME arbitrage opportunity
- (much harder to find the BEST one)
Bellman-Ford shortest-paths algorithm

Generic algorithm for single-source problem

- initialize wt[s] to 0, other wts to max
- repeat V times: relax on each edge

Order of processing edges not specified

Running time O(VE)

If no negative cycles present
- can use as preprocessing step for Dijkstra
- VE lg V for all-pairs problem
- improves on V^3 for Floyd

Not much harder to solve all-pairs than single-source (?!)

OPEN: Better alg for single-source?