CSE 326: Data Structures

Graphs, APSP, A*

James Fogarty

Spring 2009
Dijkstra’s Algorithm

Adapt BFS to handle weighted graphs

Two kinds of vertices:
- Finished or known vertices
  - Shortest distance has been computed
- Unknown vertices
  - Have tentative distance
void Graph::dijkstra(Vertex s) {
    Vertex v, w;

    Initialize s.dist = 0 and set dist of all other vertices to infinity

    while (there exist unknown vertices, find the one b with the smallest distance)
        b.known = true;

        for each a adjacent to b
            if (!a.known)
                if (b.dist + weight(b, a) < a.dist) {
                    a.dist = (b.dist + weight(b, a));
                    a.path = b;
                }
    }
}

Running time: O(|E| log |V|) – there are |E| edges to examine, and each one causes a heap operation of time O(log |V|)
Follow-On Question

• What if I had multiple potential start points, and need to know the minimum cost of reaching each node from any start point?

• Can do this by changing the algorithm
  – Add each start point to initial queue with cost 0

• If the algorithm is encapsulated (and highly tuned for efficiency), this seems bad
  – You need to re-implement the whole thing
  – Your implementation probably isn’t as good
Thinking About Graph Structure

• Working with graphs is often a problem of setting up the right graph so that you can apply the unmodified standard algorithm.

• Change the graph, apply the encapsulated and optimized SSSP implementation:
  – Add a meta-start node
  – Include 0 cost edges from it to the start nodes
All Pairs Shortest Paths (APSP)

• Given a graph G and edge costs $c_{i,j}$, find the shortest paths between all pairs of vertices in G.

  – Is this harder or easier than SSSP?

  – Could we use SSSP as a subroutine?
All Pairs Shortest Paths

• Run Dijkstra’s algorithm $|V|$ times
  – With PQ, each run has cost $|E| \log |V|$
  – Total cost is $|V| |E| \log |V|$

• Today we’ll see Floyd-Warshall, which computes APSP in $|V|^3$

• When might this be preferable?
  – If $|E| \log |V| > |V|^2$ (dense graphs)
  – Might also be preferable based on the constant factors ignored by an asymptotic analysis
Dynamic Programming

Algorithmic technique that systematically records the answers to sub-problems in a table and re-uses those recorded results (rather than re-computing them).

Simple Example: Calculating the Nth Fibonacci number.

\[ \text{Fib}(N) = \text{Fib}(N-1) + \text{Fib}(N-2) \]
Floyd-Warshall

for (int k = 1; k <= V; k++)
    for (int i = 1; i <= V; i++)
        for (int j = 1; j <= V; j++)
            if ( ( M[i][k] + M[k][j] ) < M[i][j] )
                M[i][j] = M[i][k] + M[k][j]
Floyd-Warshall APSP

\[ M[i][j] = \min(M[i][j], M[i][k] + M[k][j]) \]
Floyd-Warshall APSP

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>2</td>
<td>-</td>
<td>-4</td>
<td>-</td>
</tr>
<tr>
<td>b</td>
<td>-2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>e</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ M_{i,j} = \min(M_{i,j}, M_{i,k} + M_{k,j}) \]

K = 1

b:a:c  d:a:b
b:a:d  d:a:c
b:a:e  d:a:e
c:a:b  e:a:b
c:a:d  e:a:c
c:a:e  e:a:d
Floyd-Warshall APSP

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>2</td>
<td>-</td>
<td>-4</td>
<td>-</td>
</tr>
<tr>
<td>b</td>
<td>-</td>
<td>0</td>
<td>-2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>e</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ M[i][j] = \min(M[i][j], M[i][k] + M[k][j]) \]
Floyd-Warshall
APSP

M[i][j] = min(M[i][j], M[i][k] + M[k][j])
Floyd-Warshall APSP

\[ M[i][j] = \min(M[i][j], M[i][k] + M[k][j]) \]
Floyd-Warshall APSP

\[ M[i][j] = \min(M[i][j], M[i][k] + M[k][j]) \]
Floyd-Warshall APSP

\[
M[i][j] = \min(M[i][j], M[i][k]+ M[k][j])
\]
Floyd-Warshall
APSP

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>a</strong></td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>-4</td>
<td>1</td>
</tr>
<tr>
<td><strong>b</strong></td>
<td>-</td>
<td>0</td>
<td>-2</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td><strong>c</strong></td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td><strong>d</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td><strong>e</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

M[i][j] = min(M[i][j], M[i][k] + M[k][j])
Floyd-Warshall APSP

\[ M[i][j] = \min(M[i][j], M[i][k] + M[k][j]) \]
**Floyd-Warshall APSP**

The Floyd-Warshall algorithm is used to compute the all-pairs shortest paths in a graph. The algorithm proceeds by considering each vertex in turn as an intermediate point in the path between two vertices. The algorithm iterates over all vertices, updating the distances between all pairs of vertices based on the distances through the current vertex and the direct distance between the two vertices.

The table below shows the initial distances between vertices:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>-4</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>-</td>
<td>0</td>
<td>-2</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>c</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>e</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

The formula for updating the distance matrix is:

\[ M[i][j] = \min(M[i][j], M[i][k] + M[k][j]) \]

The graph below represents the vertices and edges of the network, with the distances marked on the edges.

- **K = 4**
  - a:d:b
  - c:d:a
  - a:d:c
  - c:d:b
  - a:d:e
  - c:d:e
  - b:d:a
  - e:d:a
  - b:d:c
  - e:d:b
  - b:d:e
  - e:d:d
Floyd-Warshall
APSP

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td></td>
<td></td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>-4</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>-</td>
<td>0</td>
<td>-2</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>c</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>e</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

to

K = 5

M[i][j] = min(M[i][j], M[i][k] + M[k][j])
Floyd-Warshall
APSP

from
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>-4</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>-0</td>
<td>-2</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td>0</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

to

Final Matrix Contents
Floyd-Warshall

for (int $k = 1; k <= V; k++)
for (int $i = 1; i <= V; i++)
for (int $j = 1; j <= V; j++)
  if ( ($M[i][k] + M[k][j]) < M[i][j] )
    M[i][j] = $M[i][k] + M[k][j]

**Invariant:** After the $k$th iteration, the matrix includes the shortest paths for all pairs of vertices ($i,j$) containing only vertices $1..k$ as intermediate vertices

Simple for loop implementation intended to be fast (especially with the help of a modern compiler). Does not bother with if statements to filter out comparisons that will never result in a change.
Problem: Large Graphs

- It is expensive to find optimal paths in large graphs, using BFS or Dijkstra’s algorithm (for weighted graphs)

- How can we search large graphs efficiently by using “commonsense” about which direction looks most promising?
Plan a route from 9th & 50th to 3rd & 51st
Plan a route from 9th & 50th to 3rd & 51st
Best-First Search

• The *Manhattan distance* ($\Delta x + \Delta y$) is an estimate of the distance to the goal
  – It is a search heuristic

- Best-First Search
  – Order nodes in priority to minimize estimated distance to the goal

- Compare: BFS / Dijkstra
  – Order nodes in priority to minimize distance from the start
Best-First Search

Open – Heap (priority queue)
Criteria – Smallest key (highest priority)
h(n) – heuristic estimate of distance from n to closest goal

```
Best_First_Search( Start, Goal_test)
    insert(Start, h(Start), heap);

    repeat
        if (empty(heap)) then return fail;
        Node := deleteMin(heap);
        if (Goal_test(Node)) then return Node;
        for each Child of node do
            if (Child not already visited) then
                insert(Child, h(Child),heap);
        end
        Mark Node as visited;
    end
```
Obstacles

- Best-FS eventually will expand vertex to get back on the right track
Best-First

Path found by Best-first

Shortest Path
Improving Best-First

- Best-first is often tremendously faster than BFS/Dijkstra, but might stop with a non-optimal solution
- How can it be modified to be (almost) as fast, but guaranteed to find optimal solutions?
  - One of the first significant algorithms developed in AI
  - Widely used in many applications
A*

Exactly like Best-first search, but using a different criteria for the priority queue:

minimize  (distance from start) + 
(estimated distance to goal)

priority $f(n) = g(n) + h(n)$
$f(n) =$ priority of a node
$g(n) =$ true distance from start
$h(n) =$ heuristic distance to goal
Optimality of A*

• Suppose the estimated distance is *always* less than or equal to the true distance to the goal
  – heuristic is a lower bound

• Then: when the goal is removed from the priority queue, we are guaranteed to have found a shortest path!
  – Everything in the queue has true distance greater than or equal to estimated distance
A* in Action

\[ h = 6 + 2 \]

\[ h = 7 + 3 \]

\[ H = 1 + 7 \]
Application of A*:
Speech Recognition

• (Simplified) Problem:
  – System hears a sequence of 3 words
  – It is unsure about what it heard
    • For each word, it has a set of possible “guesses”
      • E.g.: Word 1 is one of { “hi”, “high”, “I” }
  – What is the most likely sentence it heard?
Speech Recognition as Shortest Path

• Convert to a shortest-path problem:
  – Utterance is a “layered” DAG
  – Begins with a special dummy “start” node
  – Next: A layer of nodes for each word position, one node for each word choice
  – Edges between every node in layer i to every node in layer i+1
    • Cost of an edge is smaller if the pair of words frequently occur together in real speech
      – Technically: - log probability of co-occurrence
  – Finally: a dummy “end” node
  – Find shortest path from start to end node
Summary: Graph Search

- **Depth First**
  - Little memory required
  - Might find non-optimal path
- **Breadth First**
  - Much memory required
  - Always finds optimal path
- **Iterative Depth-First Search**
  - Repeated depth-first searches, little memory required
- **Dijkstra’s Short Path Algorithm**
  - Like BFS for weighted graphs
- **Floyd-Warshall**
  - All Pairs Shortest Paths
- **Best First**
  - Can visit fewer nodes
  - Might find non-optimal path
- **A***
  - Can visit fewer nodes than BFS or Dijkstra
  - Optimal if heuristic estimate is a lower-bound