Computer Science 331

Computation of Minimum-Cost Paths — Dijkstra's Algorithm

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #34

Outline

Computation of Minimum Cost Paths

Presented Here:

- *Dijkstra's Algorithm:* a generalization of **breadth-first search** to weighed graphs
- Rather than looking for paths with minimum *length* we will look for paths with minimum *cost*, that is, minimum *total weight*
- Application: finding the best *route* from one place to another on a map, when multiple routes are available (single-source shortest path problem)
- This is also an interesting application of **priority queues**

Definitions: Paths and Their Costs

Suppose now that $G = (V, E)$ is a *weighted* graph.

- Consider a *path*, that is, a sequence of edges

$$(u_0, u_1), (u_1, u_2), \ldots, (u_{k-2}, u_{k-1}), (u_{k-1}, u_k)$$

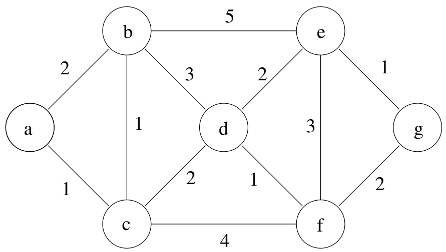
  in E where $k \geq 0$. Recall that this is a path *from* u to v if $u_0 = u$ and $u_k = v$.
- The **cost** of this path is defined to be

$$\sum_{i=0}^{k-1} w((u_i, u_{i+1})).$$

  Note that if $k = 0$ then the path has *length* 0 and it also has *cost* 0 (because the above sum has no terms).

Example

Consider the following graph G and the weights shown near the edges.



The following are paths from a to g with cost 6 :

- a, c, d, e, g (consists of edges $(a, c), (c, d), (d, e), (e, g)$)
- a, c, d, f, g (consists of edges $(a, c), (c, d), (d, f), (f, g)$)

Minimum Cost Paths

The path $(u_0, u_1), (u_1, u_2), \ldots, (u_{k-1}, u_k)$ is a *minimum-cost path from u to v* if

- this is a path from u to v (as defined above), and
- the cost of this path is *less than or equal to* the cost of any *other* path from u to v (in this graph).

Note:

- If some weights of edges are *negative* then minimum cost paths might not exist (because there may be paths from u to v that include negative-cost cycles, whose costs are smaller than any bound you could choose)!
- In this lecture we will consider a version of the problem where edges weights are all *nonnegative*, in order to avoid this problem.

Specification of Requirements

Inputs and Outputs

- Inputs and outputs have the same names and types as for "Breadth First Search" but somewhat different meanings.

Pre-Condition

- $G = (V, E)$ is a weighted graph such that

$$w((u, v)) \geq 0$$

  for every edge $(u, v) \in E$
- $s \in V$

Specification of Requirements (cont.)

Post-Condition:

- The predecessor graph $G_p = (V_p, E_p)$ corresponding to the function π and vertex s is a spanning tree for the connected component of G that contains s.
- For every vertex $v \in V$, $d[v]$ is the cost of a minimum-cost path from s to v in G. In particular, $d[v] = +\infty$ if and only if v is not reachable from s in G at all.
- For every vertex $v \in V$ that is reachable from s, the path from s to v in the predecessor graph G_p is a *minimum-cost* path from s to v in G.

Data Structures

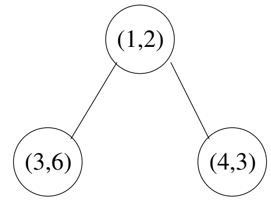
The algorithm (to be presented next) will use a **priority queue** to store information about costs of paths that have been found.

- The priority queue will be a *MinHeap*: the entry with the *smallest* priority will be at the top of the heap.
- Each node in the priority queue will store a *vertex* in G and the *cost* of a path to this vertex.
- The *cost* will be used as the node's priority.
- An array-based representation of the priority queue will be used.

A second array will be used to locate each entry of the priority queue for a given vertex in constant time.

Data Structures

Example:



heap-size$(A) = 3$

	0	1	2	3	4
A:	$(1,2)$	$(3,6)$	$(4,3)$	?	?

	0	1	2	3	4
B:	NIL	0	NIL	1	2

Explanation:

- element (v,c) in the priority queue consists of vertex v and cost c of a path from s to v
- A contains an array representation of the min-heap
- B gives the index of a vertex in the array representation of the priority queue. Examples:
  - vertex 3 is in the priority queue (at index $B[3] = 1$)
  - vertex 0 is not in the priority queue ($B[0] = NIL$)

A New Problem for Priority Queues

The "Decrease-Priority" Problem has inputs `A`, `i` and `p` and is defined as follows.

Precondition 1:

a) `A` is a Min-Heap (representing a min-priority queue Q)
b) `i` is an integer such that $0 \leq i <$ heap-size(A)
c) `p` is a value of the same type as the priorities in A
d) The priority q of the value that is currently stored at location i of A is greater than or equal to p

Postcondition 1:

a) A is now a Min-Heap storing a set in which the priority of the value originally at location i has been *decreased* from q to p (and such the set is otherwise unchanged)

A New Problem for Priority Queues

Precondition 2:

a) (a), (b) and (c) are the same as for Precondition #1
b) The priority q of the value currently stored at location i is already less than p

Postcondition 2:

a) A is not changed
b) A `LargePriorityException` is thrown

Precondition 3:

a) (a) is the same as for Precondition #1
b) i is an integer such that either $i < 0$ or $i \geq$ heap-size(A)

Postcondition 3:

a) A is not changed
b) A `RangeException` is thrown

Idea and Pseudocode

Idea: Move the modified value up in the heap until it is place.

Notation: $P(y)$ will denote the priority of a value y.

`void Decrease-Priority (A,i,p)`

if $i < 0$ or $i \geq heapsize(A)$ **then**
  throw `RangeException`
else if $p > P(A[i])$ **then**
  throw `LargePriorityException`
else
  Change $P(A[i])$ to p
  $j = i$
  while $j > 0$ and $P(A[parent(j)]) > P(A[j])$ **do**
    $tmp = A[j]$; $A[j] = A[parent(j)]$; $A[parent(j)] = tmp$
    $j = parent(j)$
  end while
end if

Correctness and Efficiency

Properties of This Algorithm:

- The given algorithm is correct.
- If A stores a set with size n then the number of steps used by the algorithm is in $\Theta(\text{logn})$ in the worst case.

Details of the proof of correctness and the analysis of this algorithm will be included in the tutorial exercise on this topic.

Dijkstra's Algorithm: Pseudocode

MCP(G, s)
  for $v \in V$ **do**
    $colour[v] = \text{white}$
    $d[v] = +\infty$
    $\pi[v] = \text{NIL}$
  end for
  Initialize an empty priority queue Q
  $colour[s] = \text{grey}$
  $d[s] = 0$
  add vertex s with priority 0 to Q

Pseudocode, Continued

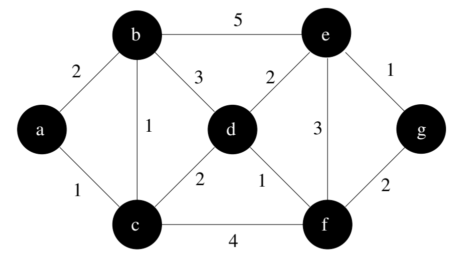
while (Q is not empty) **do**
  $(u, c) = \text{extract-min}(Q)$ {Note: $c = d[u]$}
  for each $v \in Adj[u]$ **do**
    if ($colour[v] == \text{white}$) **then**
      $d[v] = c + w((u, v))$
      $colour[v] = \text{grey}$; $\pi[v] = u$
      add vertex s with priority $d[v]$ to Q
    else if ($colour[v] == \text{grey}$) **then**
      Update information about v (shown on next slide)
    end if
  end for
  $colour[u] = \text{black}$
end while
return π, d

Pseudocode, Concluded

Updating Information About v

if $(c + w((u, v)) < d[v])$ **then**
  $old = d[v]$
  $d[v] = c + w((u, v))$
  $\pi[v] = u$
  Use `Decrease-Priority` to replace (v, old) on Q with $(v, d[v])$
end if

Example



	a	b	c	d	e	f	g
d	0	2	1	3	5	4	6
π	-	a	a	c	d	d	f

Q: (empty)

Step 7:

- Extract-Min (returns $(g, 6)$)
- color g black — done!

Eg. shortest path from a to g is a, c, d, f, g (cost $d[g] = 6$). Edges:

$$(\pi(g), g), (\pi(f), f), (\pi(d), d), (\pi(c), c) = (f, g), (d, f), (c, d), (a, c)$$

Easily Established Properties

Each of the following is easily established by inspecting the code:

1. *Colour Properties:*
   - The initial colour of every node $v \in V$ is **white**.
   - The colour of a vertex can change from **white** to **grey**.
   - The colour of a vertex can change from **grey** to **black**.
   - No other changes in colour are possible.
2. *Contents of Queue:* The following properties are part of the *loop invariant* for the **while** loop:
   - If (u, d) is an element of the queue then $u \in V$, $colour[u] =$ **grey**, and $d = d[u]$.
   - If a vertex v (and its cost) were included on the queue but have been removed, then $colour[v] =$ **black**.
   - Vertices that have never been on the queue are **white**.

Additional Properties (Proofs Not Too Hard)

The following are also part of the loop invariant for the **while** loop.

3. All vertices that belong to the predecessor subgraph (for π and s) are either **grey** or **black**.
4. All neighbours of any **black** vertex are either **black** or **grey**.
5. If the colour of a vertex v is **black** or **grey** then there exists a path

   $$(u_0, u_1), (u_1, u_2), \ldots, (u_{k-1}, u_k)$$

   from s to v *in the predecessor subgraph* with cost $d[v]$ such that $colour[u_i] =$ **black** for $1 \leq i \leq k - 1$ $(u_1 = s,\ u_k = v)$

   Furthermore, *all* paths from s to v in G with the above form (i.e., all but the final vertex is **black**) have cost *at least* $d[v]$.
6. If $colour[x] =$ **black** and $colour[y] =$ **grey** then $d[x] \leq d[y]$.
7. If $colour[x] =$ **white** then $d[x] = +\infty$.

One Final Property

The next property is part of the loop invariant, as well.

8. Suppose that the colour of v is either **grey** or **white**. Then *every* path from s to v in G *must begin* with a sequence of edges

$$(u_0, u_1), (u_1, u_2), \ldots, (u_{k-1}, u_k)$$

where $k \geq 2$, $colour[u_i] =$ **black** for $1 \leq i \leq k-1$, and where $colour[u_k] =$ **grey**.

Indeed, this is a consequence of Property #4 (listed above).

Undoubtedly, some of these properties do not seem very interesting. They are important because they help to establish the one that is given next.

Final Piece of the Loop Invariant

Here is the last piece of the loop invariant.

9. The following property is satisfied by every vertex v such that $colour[v] =$ **black**, and also by the vertex v such that $(v, d[v])$ is at the top of the priority queue, if Q is nonempty:
   - The unique path from s to v in the predecessor subgraph for π and s is a minimum-cost path from s to v in G, and the cost of this path is $d[v]$.

The **loop invariant** consists of the pieces of it that have now been identified.

One can establish that this *is* a loop invariant by induction on the number of executions of the loop body.

Application of the Loop Invariant

Notice that, if the loop terminates, then

- The priority queue is *empty*.
- Therefore there are no **grey** vertices left!
- Therefore the only neighbours of **black** vertices are also **black**.
- This can be used to show that no **white** vertex is reachable from s.
- This, and various pieces of the loop invariant, can be used to establish partial correctness of the algorithm.

Termination and Running Time

It follows by a modification of the analysis of the breadth-first search algorithm that

- The total number of operations *on* the priority queue, and the total number of operations that *do not involve* the priority queue, are each in $\Theta(|V| + |E|)$.

Since the size of the priority queue never exceeds $|V|$ each operation on the priority queue requires $O(\log |V|)$ steps.

Conclusion: This algorithm terminates (on inputs $G = (V, E)$ and $s \in V$) after using $O((|V| + |E|) \log |V|)$ steps.

- $O(|V| \log |V| + |E|)$ using a Fibonacci heap (amortized)

References

Further Reading and Java Code:

- **Introduction to Algorithms**, Chapter 24
- This also includes information about a slower algorithm (The "Bellman-Ford algorithm") that solves this problem when edge weights are allowed to be *negative*.
- **Data Structures: Abstraction and Design Using Java**, Chapter 10.6