Shortest Weighted Path

BFS gives the shortest path in terms of the number of edges in the path

What if you have edge weights, and want the shortest path in terms of the sum of the edge weights along the path?

Shortest Weighted Path

Fix.

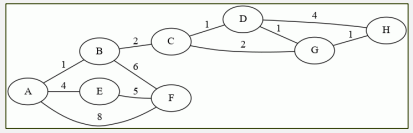
- allow dist[u] to be updated each time an edge (v,u) is encountered
- ensure that vertices along the shortest path s → u are processed before u
  - so that dist[u] won't need to be updated after u is processed

Implementation.

- observation: as long as all edge weights > 0, dist[v] < dist[u] for all vertices v on shortest path s → u
  - thus handling 'discovered' vertices in order of increasing dist[v] would satisfy the second part of the fix

Shortest Weighted Path – Attempt

```
bfs(G,s)
  for each vertex u in V-{s} do
    state[u] = "undiscovered"
    prev[u] = null
    dist[u] = ∞
  state[s] = "discovered"
  prev[s] = null
  dist[s] = 0
  Q.enqueue(s)
  while Q is not empty do
    u = Q.dequeue()
    for each edge (u,v) in G.incidentEdges(u) do
      if state[v] = "undiscovered" then
        state[v] = "discovered"
        prev[v] = u
        dist[v] = dist[u] + w(u,v)  // +1 for bfs
        Q.enqueue(v)
    state[u] = "processed"
```



A	0			
B	∞	1		
C	∞	∞	3	
D	∞	∞		
E	∞	4		
F	∞	8	*	
G	∞	∞		
H	∞	∞		

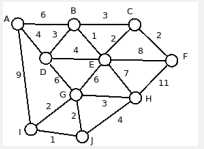
- BFS sets dist[v] when a vertex is first discovered
- this label will be wrong if there is a longer path (more edges) with a lower cost

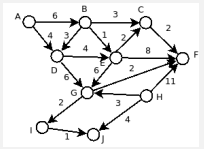
Dijkstra's Algorithm

finds the shortest path from *s* to every other vertex in the graph

to just find the shortest path from *s* to *t*, exit loop after *t* is removed from the PQ

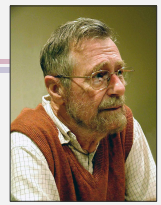
```
algorithm dijkstra(G,s):
  for all v in V do
    dist[v] ← ∞
    prev[v] = null

  dist[s] ← 0

  PQ ← makeQueue(V)

  while PQ is not empty do
    v ← PQ.removeMin()
    for each incident edge e=(v,u)
      if dist[u] > dist[v]+w(v,u) then
        dist[u] = dist[v]+w(v,u)
        PQ.decreaseKey(u)
        prev[u] = v
```





- can augment algorithm to store shortest path in addition to distance (add `prev` labels)

Edsger W Dijkstra



- Dutch computer scientist
- 1930 – 2002

- received the 1972 Turing Award for fundamental contributions to the development of structured programming languages
  - characteristics
    - three control structures: ordered sequence of statements, conditionals, loops
    - subroutines
    - blocks
  - in contrast to languages like BASIC which utilized GOTO statements

https://en.wikipedia.org/wiki/Edsger_W._Dijkstra

Dijkstra's Algorithm

```
algorithm dijkstra(G,s):
❶ for all v in V do
    dist[v] ← ∞
  dist[s] ← 0
❷ PQ ← makeQueue(V)
❸ while PQ is not empty do
    v ← PQ.removeMin()
❹ for each incident edge e=(v,u)
❺   if dist[u] > dist[v]+w(v,u) then
      dist[u] = dist[v]+w(v,u)
      PQ.decreaseKey(u)
```

Running time.

❶ initialize dist[v] labels
- O(n) with O(1) record-keeping and O(1) per element iteration (possible with any graph implementation)

❷ make PQ with n elements
- O(makePQ)

❸ n is-empty checks + n vertices removed from PQ
- O(n × 1) + O(n × removeMin) [O(1) isEmpty() possible with any PQ implementation)

❹ 2m (undirected) or m (directed) edges visited over all vertices [2m for undirected because an edge is visited from each of its endpoints]
- O(m) assuming O(1) per element iteration (possible with adjacency list)

❺ up to O(1) + O(decreaseKey) per edge (if label is updated)
- assuming O(1) record-keeping

→ O(makePQ + n × removeMin + m × decreaseKey) total

Dijkstra's Algorithm

→ O(makePQ + n × removeMin + m × decreaseKey) total

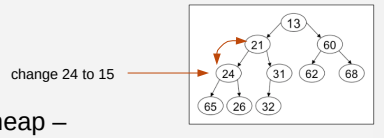
operation	heap
makeQueue	O(n log n) – repeated insert O(n) – heapify
removeMin	O(log n)
decreaseKey	O(log n) (**)
total running time for Dijkstra's algorithm	O(n + n log n + m log n) = O((n+m) log n) = O(m log n) for connected graphs

This is –
- O(n log n) for sparse graphs [m = O(n)]
- O(n^2 log n) for dense graphs [m = O(n^2)]

(**) assuming O(1) to locate element within PQ (which can be done with a *locator*), otherwise O(n) to search PQ to find entry

Locators

- decrease key can be done in O(log n) time if the location (index) of the key in the heap is known
  - update key, then bubble up (min heap) to fix ordering property



- searching for a key in a heap –
  - O(n) – may need to check all elements
- how to avoid search?
  - look up instead!
  - what if we add a map key → array index?
    - O(1) to locate key with hashtable
    - map must be updated for each bubble up or bubble down swap
      - but that only involves two elements, so two updates – +O(1) per swap doesn't change the big-Oh for bubbling

https://cs.stackexchange.com/questions/6990/deletion-in-min-max-heaps

Dijkstra's Algorithm

→ O(makePQ + n × removeMin + m × decreaseKey) total

operation	array – unsorted	heap
makeQueue	O(n) – take elements in whatever order	O(n log n) – repeated insert O(n) – heapify
removeMin	O(n) – search, then swap with last	O(log n)
decreaseKey	O(1) – update key value (**)	O(log n) (**)
total running time for Dijkstra's algorithm	$O(n + n^2 + m) = O(n^2)$	O(n + n log n + m log n) = O((n+m) log n) = O(m log n) for connected graphs

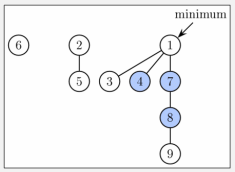
Observation:

- for sparse graphs [m = O(n)], the heap is more efficient
- for dense graphs [m = $O(n^2)$], the unsorted array is more efficient

(**) assuming O(1) to locate element within PQ (which can be done with a *locator*), otherwise O(n) to search PQ to find entry

Other Options

(Binary) heaps are not the only choice.



- Fibonacci heaps
  - maintain a forest of heaps
  - insert/remove involves splitting and merging heaps in order to keep the degree of each node low and the size of each subtree sufficiently high
  - achieves O(log n) removeMin and O(1) decreaseKey
    - *amortized time* – on average

  → total time for Dijkstra's algorithm: O(m + n log n)

  ★
    - for sparse graphs [m = O(n)], get O(n log n) – as good as a binary heap
    - for dense graphs [m = $O(n^2)$], get $O(n^2)$ – as good as an unsorted array

Other Options

(Binary) heaps are not the only choice for implementing priority queues.

- *d*-ary heaps
  - each node has *d* children instead of two, reducing the height of the tree by a factor of log d
  - insert – O(log n / log d)
  - removeMin – O(d log n / log d)
    - have to check all *d* children at each level to determine smallest element
  - *d* should be chosen to be m/n (the average degree of the graph)

  → total time for Dijkstra's algorithm:
  O((nd+m) log n / log d) = O(m log n / log (m/n))

  ★
    - for sparse graphs [m = O(n)], get O(n log n) – as good as a binary heap
    - for dense graphs [m = $O(n^2)$], get $O(n^2)$ – as good as an unsorted array
    - in between [m = $n^{1+\delta}$], get O(m) - δ is a constant

Shortest Paths with Negative Edges

Dijkstra's algorithm requires w(u,v) > 0.
But what if there are edges with negative weights?

Approach this like you are dealing with a special case:
see where the algorithm you have breaks.

- Dijkstra's algorithm relies on d(s,w) < d(s,v) for all vertices w on the shortest path s → v
- negative or zero weights mean that d(s,w) ≥ d(s,v) is possible
  - thus v may be removed from the PQ before w, and dist[v] is not correct at the time v is marked 'processed' because w has not yet been processed

Bellman-Ford

Idea.

- abandon traversal – instead repeatedly update every edge in the graph

```
algorithm bellmanford(G,s):
  for all v in V do
    dist[v] ← ∞

  dist[s] ← 0

  repeat |V|-1 times
    updated ← false
    for all edges e=(v,u) in E do
      if dist[u] > dist[v]+w(v,u) then
        dist[u] = dist[v]+w(v,u)
        updated ← true
    if !updated then
      break
```

(the longest path from s to any reachable vertex has length n-1, so that many repetitions will serve to propagate dist[s] to every vertex)

(can stop repetitions early if there are no more updates to propagate)

Bellman-Ford(-Moore) algorithm

- L. R. Ford, 1927-2017
- American mathematician also known for
  - network flow problems
  - Ford-Fulkerson algorithm (max flow)
  - Ford-Johnson algorithm (sorting)
- Richard Bellman, 1920-1984
- American applied mathematician also known for
  - introducing dynamic programming (1953)
  - Bellman equation, Hamilton-Jacobi-Bellman equation (optimal control theory)
  - *curse of dimensionality* – exponential increase in volume when adding dimensions

- Edward Moore, 1925-2003
- American professor of mathematics and computer science also known for
  - Moore finite state machine
  - an early pioneer of artificial life
  - Moore graphs
  - Shortest Path Faster Algorithm (variation of Bellman-Ford)

https://en.wikipedia.org/wiki/Richard_E._Bellman

Bellman-Ford

```
algorithm bellmanford(G,s):
  for all v in V do                    ❶
    dist[v] ← ∞

  dist[s] ← 0

  repeat |V|-1 times                   ❸
    updated ← false
    for all edges e=(v,u) in E do      ❷
      if dist[u] > dist[v]+w(v,u) then
        dist[u] = dist[v]+w(v,u)
        updated ← true
    if !updated then
      break
```

Running time.

❶ initialize dist[v] for all vertices
- O(n) assuming O(1) record-keeping and O(1) per element iteration (possible with any graph implementation)

❷ for all edges in E, repeated up to |V|-1 times ❸
- O(m) × O(n) = O(mn) assuming O(1) record-keeping and O(1) per element iteration (possible with any graph implementation)

→ O(nm) total

Negative Weight Cycles

A possibility with negative weight edges is that there is a negative weight cycle.

- important to detect, because “shortest path” isn't meaningful in that case

Observation: a negative-weight cycle means that there is always at least one edge being updated in a given round of Bellman-Ford.

Solution: repeat *n* times. (one extra repetition)

- there is a negative-weight cycle if the loop terminates because the *n*th iteration has been completed instead of the no-more-updates break being reached

Other Related Algorithms

- Dijkstra's and Bellman-Ford find the shortest path from a vertex *s* to all other vertices
- all pairs shortest path – find shortest path from every vertex to every other vertex
  - for each vertex *v* in V, run Dijkstra's algorithm with *v* as the starting vertex
    → $O(n^2 \log n)$ to $O(n^3)$ depending on the density of the graph and the PQ implementation
  - Floyd-Warshall algorithm
    → $O(n^3)$
    - lower constant factors than the $O(n^3)$ version of Dijkstra's
    - simpler to implement than Dijkstra's, though finding the shortest path and not just the distance takes more effort

Floyd-Warshall

- Robert Floyd, 1936-2001
- computer scientist also known for
  - Floyd's cycle-finding algorithm
  - Floyd-Steinberg dithering
  - contributions to Hoare Logic
- received the Turing Award in 1978 for contributions relating to the creation of efficient and reliable software

- Stephen Warshall, 1935-2006
- American computer scientist also known for
  - work in operating systems, compilers, language design, and operations research

https://en.wikipedia.org/wiki/Robert_W._Floyd
https://en.wikipedia.org/wiki/Stephen_Warshall

Other Algorithms

- transitive closure – for all pairs of vertices (u,v), determine whether *v* is reachable from *u*
  - for each vertex *u* in V, run bfs(u) to find the reachable vertices
    → $O(n^2 + nm)$
  - run Floyd-Warshall algorithm – *v* is reachable from *u* if the length of the shortest path u→v is not ∞
    → $O(n^3)$

Takeaways

- know the algorithm (can trace it), when it is applicable, its running time
  - Dijkstra's algorithm – heap, fancier implementations
  - Bellman-Ford – supports negative weight edges, detects negative weight cycles
- know what it is, suitable algorithms/approaches for solving and their running times
  - all pairs shortest path – repeated Dijkstra, Floyd-Warshall
  - transitive closure – repeated bfs, Floyd-Warshall