Advanced Algorithms/Analysis

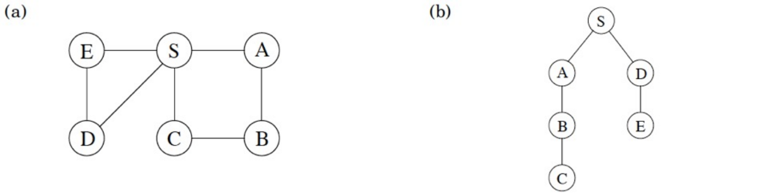
BFS, Dijkstra's algorithm and Bellman Ford algorithm
(Read Ch. 4 from reference book)
CS 721
Fal 2019

September 12, 2019

DFS and shortest path

- DFS readily identifies all the vertices of a graph that can be reached from a starting vertex. It also finds path to these vertices which may not be most economical (shortest path)
- To get a feeling for this, consider the following graph , where each edge weight is 1 and, DFS starts from node S

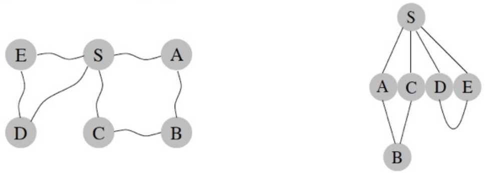
Figure 4.1 (a) A simple graph and (b) its depth-first search tree.



Breadth-first search (BFS): Basic idea

- Consider a graph where each edgee is of same unit length (weight=1)
- Consider each edge to be a string and we want to traverse the graph from a start node S
- One possibility is let the graph hang from node S and traverse the graph level by level as shown in the following figure

Figure 4.2 A physical model of a graph.



- BFS graph search tries to implement this idea

BFS Algorithm

- Here is the algorithm for BFS

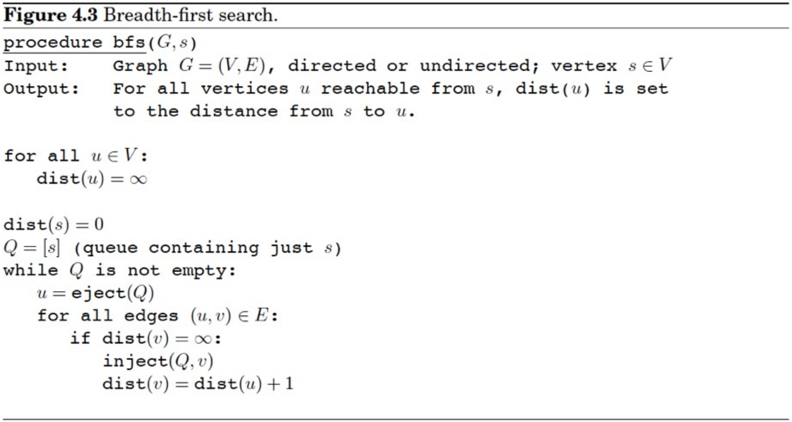


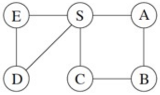
Figure 4.3 Breadth-first search.

```
procedure bfs($G, s$)
Input:   Graph $G = (V, E)$, directed or undirected; vertex $s \in V$
Output:  For all vertices $u$ reachable from $s$, dist($u$) is set
         to the distance from $s$ to $u$.

for all $u \in V$:
    dist($u$) $= \infty$

dist($s$) $= 0$
$Q = [s]$ (queue containing just $s$)
while $Q$ is not empty:
    $u =$ eject($Q$)
    for all edges $(u, v) \in E$:
        if dist($v$) $= \infty$:
            inject($Q, v$)
            dist($v$) $=$ dist($u$) $+ 1$
```

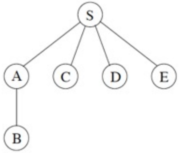
BFS Algorithm output

- Here is the result of running BFS on the same graph

Figure 4.4 The result of breadth-first search on the graph of Figure 4.1.



Order of visitation	Queue contents after processing node
	$[S]$
S	$[A\ C\ D\ E]$
A	$[C\ D\ E\ B]$
C	$[D\ E\ B]$
D	$[E\ B]$
E	$[B]$
B	$[\,]$



- Unlike DFS tree, it has the property that all paths from S are shortest possible

Running time of BFS

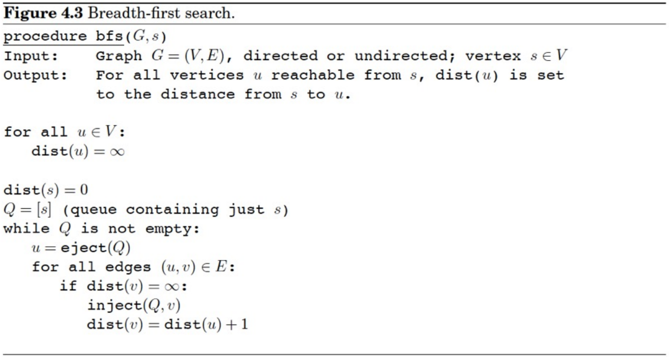


Figure 4.3 Breadth-first search.

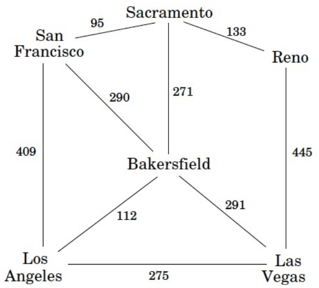
```
procedure bfs($G, s$)
Input:    Graph $G = (V, E)$, directed or undirected; vertex $s \in V$
Output:   For all vertices $u$ reachable from $s$, dist($u$) is set
          to the distance from $s$ to $u$.

for all $u \in V$:
    dist($u$) $= \infty$

dist($s$) $= 0$
$Q = [s]$ (queue containing just $s$)
while $Q$ is not empty:
    $u =$ eject($Q$)
    for all edges $(u, v) \in E$:
        if dist($v$) $= \infty$:
            inject($Q, v$)
            dist($v$) $=$ dist($u$) $+ 1$
```

-
  - Each vertex is put into the queue exactly once, when it is first encountered and is dequeued exactly once, so there are $2|V|$ queue operations.
  - Rest of the action is in the inner loop. Each edge, over the entire run of BFS is checked once (directed graph) or twice (undirected graph) and takes $O(|E|)$ time
  - So total running time is $O(|V| + |E|)$

BFS : unequal edge weight?

- How do we use DFS if edge weights are positive but unequal?

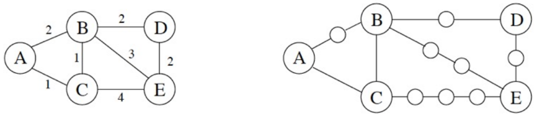
Figure 4.5 Edge lengths often matter.



Dijkstra's algorithm: adaptation from BFS

- Break G's long edges into unit length pieces by introducing dummy nodes.
- For each edge $e = (u, v)$ of E with edge weight l_e, replace it by l_e edges of length 1 by adding $l_e - 1$ dummy nodes between u and v

Figure 4.6 Breaking edges into unit-length pieces.

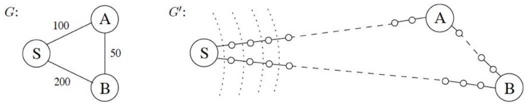


- This new graph G' contains all the vertices of G and distance between them is exactly same as in G
- We can perform BFS on G' and get our answer but that would be very inefficient

Alarm clock algorithm

- BFS on G' is mostly uneventful but we can set an alarm and update alarm time as we discover new nodes

Figure 4.7 BFS on G' is mostly uneventful. The dotted lines show some early "wavefronts."



- This gives rise to alarm clock algorithm

- Set an alarm clock for node s at time 0.
- Repeat until there are no more alarms:

  Say the next alarm goes off at time T, for node u. Then:

  - The distance from s to u is T.
  - For each neighbor v of u in G:
    - If there is no alarm yet for v, set one for time $T + l(u, v)$.
    - If v's alarm is set for later than $T + l(u, v)$, then reset it to this earlier time.

Dijkstra’s algorithm

- The alarm clock algorithm computes distances in any graph with positive integral edge weight
- It is almost ready to be used, except that we need to somehow implement the system of alarm
- The right data structure for this priority queue (implemented using heap) and supports the following operations:
    - **Insert:** Add a new element to the set
    - **Decrease-key:** Decrease the key value of a particular element
    - **Delete-min:** Return the element with smallest key and remove it from the queue
    - **Make-queue:** Build a priority queue with given elements and their key values

Dijkstra's algorithm

- Dijkstra's shortest path algorithm

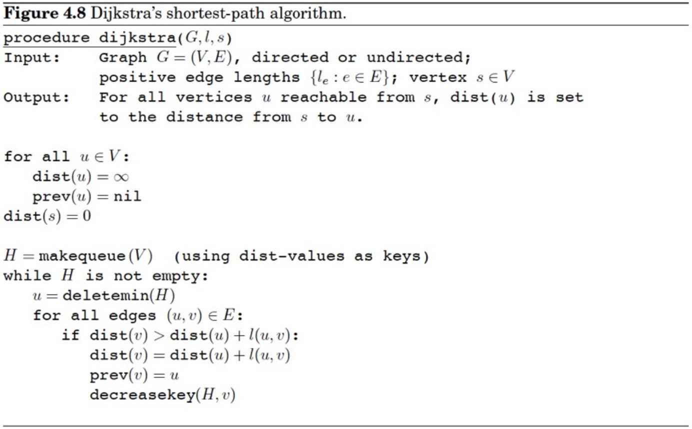


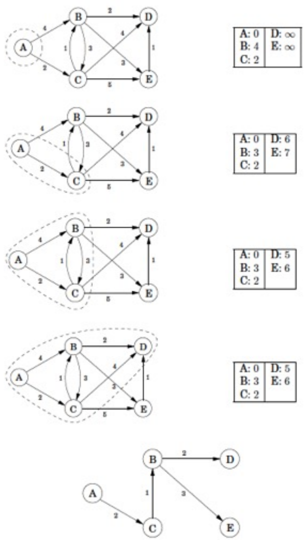
Figure 4.8 Dijkstra's shortest-path algorithm.

```
procedure dijkstra($G, l, s$)
Input:     Graph $G = (V, E)$, directed or undirected;
           positive edge lengths $\{l_e : e \in E\}$; vertex $s \in V$
Output:    For all vertices $u$ reachable from $s$, dist($u$) is set
           to the distance from $s$ to $u$.

for all $u \in V$:
    dist($u$) $= \infty$
    prev($u$) $=$ nil
dist($s$) $= 0$

$H =$ makequeue($V$)  (using dist-values as keys)
while $H$ is not empty:
    $u =$ deletemin($H$)
    for all edges $(u, v) \in E$:
        if dist($v$) $>$ dist($u$) $+ l(u, v)$:
            dist($v$) $=$ dist($u$) $+ l(u, v)$
            prev($v$) $= u$
            decreasekey($H, v$)
```

Dijkstra's algorithm result

- A complete run of Dijkstra's shortest path algorithm

Figure 4.9 A complete run of Dijkstra's algorithm, with node A as the starting point. Also shown are the associated `dist` values and the final shortest-path tree.



A: 0	D: ∞
B: 4	E: ∞
C: 2	

A: 0	D: 6
B: 3	E: 7
C: 2	

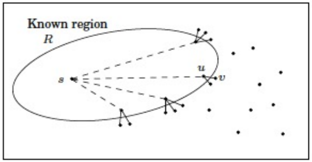
A: 0	D: 5
B: 3	E: 6
C: 2	

A: 0	D: 5
B: 3	E: 6
C: 2	

Alternative derivation of Dijkstra's algorithm

- An alternative way of computing shortest path is as follows:
  - Expand outward from the starting point s, steadily growing the region of the graph to which distances and shortest paths are known.
    - This growth should be orderly, first incorporating the closest nodes and then moving on to those further away.
    - More precisely, when the "known region" is some subset of vertices R that includes s, the next addition to it should be the node outside R that is closest to s. Let us call this node v; now the question is: how do we identify it?

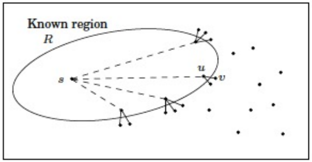
Figure 4.10 Single-edge extensions of known shortest paths.



Alternative derivation of Dijkstra's algorithm contd.

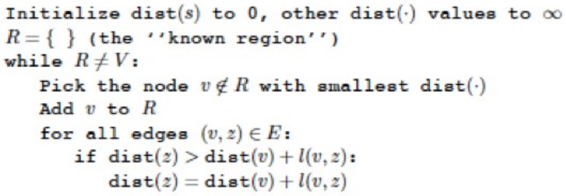
- To answer, consider u, the node just before v in the shortest path from s to v
  - Since all edge lengths are positive, u must be closer to s than v is. This means that u is in R, otherwise it would contradict v' s status as the closest node to s outside R.
  - So, the shortest path from s to v is simply a known shortest path extended by a single edge. But how to identify v?
    - The answer is, the shortest of all extended paths: it is the node outside R for which the smallest value of $dist(s,u) + l(u,v)$ is attained, as u ranges over R.
    - In other words, try all single-edge extensions of the currently known shortest paths, find the shortest such extended path, and proclaim its endpoint to be the next node of R.

Figure 4.10 Single-edge extensions of known shortest paths.



Alternative derivation of Dijkstra's algorithm contd.

- We now have an algorithm for growing R by looking at extensions of the current set of shortest paths.
  - Some extra efficiency comes from noticing that on any given iteration, the only new extensions are those involving the node most recently added to region R.
  - All other extensions will have been assessed previously and do not need to be recomputed.
  - In the following pseudocode, $dist(v)$ is the length of the currently shortest single-edge-extended path leading to v; it is ∞ for nodes not adjacent to R.



```
Initialize dist($s$) to 0, other dist($\cdot$) values to $\infty$
$R = \{ \}$ (the ''known region'')
while $R \neq V$:
    Pick the node $v \notin R$ with smallest dist($\cdot$)
    Add $v$ to $R$
    for all edges $(v, z) \in E$:
        if dist($z$) $>$ dist($v$) $+ l(v, z)$:
            dist($z$) $=$ dist($v$) $+ l(v, z)$
```

Running time of Dijkstra's algorithm

- At the level of abstraction Dijkstra's algorithm is structurally identical to BFS
- However, it is more slower because priority queue operations are slower compared to standard enqueue and dequeue operations of a simple queue
- Main components contributing to running time:
  - Make-queue takes at least as long as $|V|$ insert operations
  - Each node is deleted once, thus total $|V|$ deletemin operations
  - $|E|$ decrease-key operations over entire run of Dijkstra's algorithm
- For priority queue implementation all three operations (insert, deletemin,decrease-key) take $O(\log(|V|))$ time.
- Thus total running time of Dijkstra's algorithm is $O\left((|V| + |E|)\log(|V|)\right)$.

Binary heap

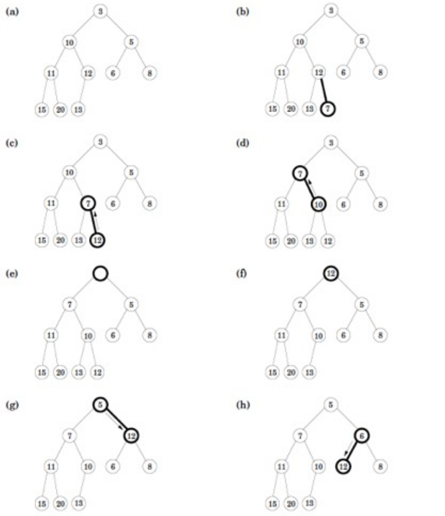
- In a binary (min) heap elements are stored in a complete binary tree, namely, a binary tree in which each level is filled in from left to right, and must be full before the next level is started.
- The key value of any node of the tree is less than or equal to that of its children, thus root contains the smallest element.
    - To **insert**, place the new element at the bottom of the tree (in the first available position), and let it "bubble up." That is, if it is smaller than its parent, swap the two and repeat
    - The number of swaps is at most the height of the tree, which is $\log n$ when there are n elements.
    - A **decreasekey** is similar, except that the element is already in the tree, so we let it bubble up from its current position.
    - To **deletemin**, return the root value, remove this element from the heap, then take the last node in the tree (in the rightmost position in the bottom row) and place it at the root and let it "sift down": if it is bigger than either child, swap it with the smaller child and repeat. Again this takes O(log n) time.

Binary heap

- The regularity of a complete binary tree makes it easy to represent using an array.
  - If there are n nodes, this ordering specifies their positions $1, 2, ..., n$ within the array.
  - Moving up and down the tree is easily simulated on the array, using the fact that node number j has parent $\lfloor j/2 \rfloor$ and children $2j$ and $2j + 1$.

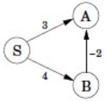
Binary heap

Figure 4.11 (a) A binary heap with 10 elements. Only the key values are shown. (b)–(d) The intermediate "bubble-up" steps in inserting an element with key 7. (e)–(g) The "sift-down" steps in a delete-min operation.



Shortest path in presence of negative edge

- Dijkstra's algorithm works in part because the shortest path from the starting point s to any node v must pass exclusively through nodes that are closer than v.
- This no longer holds when edge lengths can be negative.



- What needs to be changed in order to accommodate this new complication?
  - To answer this, observe that dist values it maintains are always either overestimates or exactly correct.
  - They start off at ∞, and the only way they ever change is by updating along an edge:

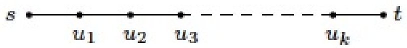
```
procedure update($(u,v) \in E$)
dist($v$) = min{dist($v$), dist($u$) + $l(u,v)$}
```

Shortest path in presence of negative edge contd.

- The update operation says that the distance to v cannot possibly be more than the distance to u, plus $l(u, v)$. It has the following properties.
  1. It gives the correct distance to v in the particular case where u is the second-last node in the shortest path to v, and $dist(u)$ is correctly set.
  2. It will never make $dist(v)$ too small, and in this sense it is safe.
- This operation is extremely useful: it is harmless, and if used carefully, will correctly set distances.
- In fact, Dijkstra's algorithm can be thought of simply as a sequence of updates.
- We know this particular sequence doesn't work with negative edges, but is there some other sequence that does?

Shortest path in presence of negative edge contd.

- To get a sense of the properties this sequence must possess, let's pick a node t and look at the shortest path to it from s.



- This path can have at most $|V| - 1$ edges (why?).
- If the sequence of updates per- formed includes $(s, u_1), (u_1, u_2), (u_2, u_3), ..., (u_k, t)$, in that order (though not necessarily consecutively), then by the first property the distance to t will be correctly computed. It doesn't matter what other updates occur on these edges, or what happens in the rest of the graph, because updates are safe.
- However, if we don't know all the shortest paths beforehand, how can we be sure to update the right edges in the right order?

Shortest path in presence of negative edge: Bellman Ford algorithm

- Here is an easy solution:
  - simply update all the edges, $|V|-1$ times! The resulting $O(|V||E|)$ procedure is called the Bellman-Ford algorithm.

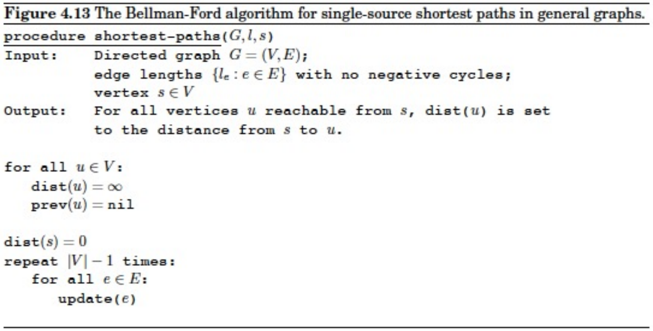
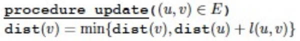


Figure 4.13 The Bellman-Ford algorithm for single-source shortest paths in general graphs.

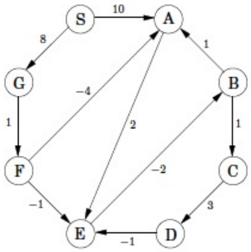
```
procedure shortest-paths($G,l,s$)
Input:     Directed graph $G=(V,E)$;
           edge lengths $\{l_e : e \in E\}$ with no negative cycles;
           vertex $s \in V$
Output:    For all vertices $u$ reachable from $s$, dist($u$) is set
           to the distance from $s$ to $u$.

for all $u \in V$:
   dist($u$) $= \infty$
   prev($u$) $=$ nil

dist($s$) $= 0$
repeat $|V|-1$ times:
   for all $e \in E$:
      update($e$)
```



```
procedure update($(u,v) \in E$)
dist($v$) $= \min\{$dist($v$), dist($u$) $+ l(u,v)\}$
```

Bellman Ford algorithm

Figure 4.14 The Bellman-Ford algorithm illustrated on a sample graph.



	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

- A note about implementation:
  - for many graphs, the maximum number of edges in any shortest path is substantially less than $|V| - 1$, with the result that fewer rounds of updates are needed.
  - Therefore, it makes sense to add an extra check to the shortest-path algorithm, to make it terminate immediately after any round in which no update occurred.

Negative cycle

- The shortest-path problem is ill-posed in graphs with negative cycles.
  - As might be expected, Bellman Ford algorithmworks only in the absence of such cycles.
  - But where did this assumption appear in the derivation of the algorithm?
    - Well, it slipped in when we asserted the existence of a shortest path from s to t.
- Fortunately, it is easy to automatically detect negative cycles and issue a warning.
  - Instead of stopping after $|V| - 1$ iterations, perform one extra round. There is a negative cycle if and only if some dist value is reduced during this final round.

Shortest paths in DAGs

- We will now see how the single-source shortest-path problem can be solved in just linear time on directed acyclic graphs.
- The key source of efficiency is the following
  - In any path of a DAG, the vertices appear in increasing linearized order.
    - Therefore, it is enough to linearize (that is, topologically sort) the DAG by depth-first search, and then visit the vertices in sorted order, updating the edges out of each vertex.
    - Notice that our scheme doesn't require edges to be positive.
    - In particular, we can find longest paths in a DAG by the same algorithm: just negate all edge lengths.

Single source shortest paths in DAGs

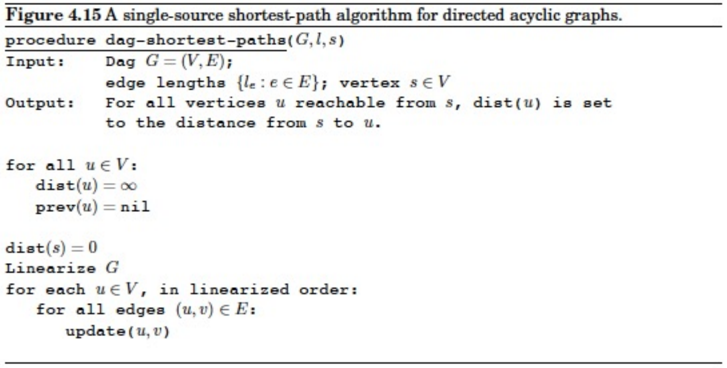


Figure 4.15 A single-source shortest-path algorithm for directed acyclic graphs.

```
procedure dag-shortest-paths($G, l, s$)
Input:    Dag $G = (V, E)$;
          edge lengths $\{l_e : e \in E\}$; vertex $s \in V$
Output:   For all vertices $u$ reachable from $s$, dist($u$) is set
          to the distance from $s$ to $u$.

for all $u \in V$:
   dist($u$) $= \infty$
   prev($u$) $=$ nil

dist($s$) $= 0$
Linearize $G$
for each $u \in V$, in linearized order:
   for all edges $(u, v) \in E$:
      update($u, v$)
```