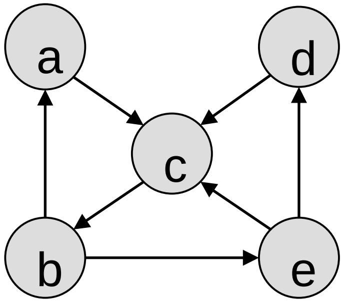
CSCI 104
Graph Algorithms

Mark Redekopp
David Kempe
Sandra Batista

PAGERANK ALGORITHM

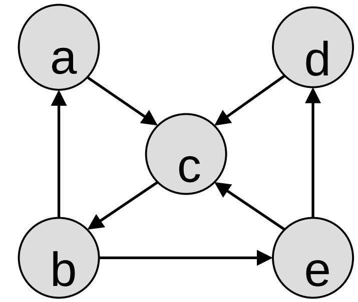
PageRank

- Consider the graph at the right
  - These could be webpages with links shown in the corresponding direction
  - These could be neighboring cities
- PageRank generally tries to answer the question:
  - **If we let a bunch of people randomly "walk" the graph, what is the probability that they end up at a certain location (page, city, etc.) in the "steady-state"**
- We could solve this problem through Monte-Carlo simulation (essentially the CS 103 PA5 or PA1 Coin-flipping or Zombie assignment…depending on semester)
  - Simulate a large number of random walkers and record where each one ends to build up an answer of the probabilities for each vertex
- But there are more efficient ways of doing it



PageRank

- Let us write out the adjacency matrix for this graph
- Now let us make a weighted version by normalizing based on the out-degree of each node
  – Ex. If you're at node B we have a 50-50 chance of going to A or E
- From this you could write a system of linear equations (i.e. what are the chances you end up at vertex I at the next time step, given you are at some vertex J now
  – pA = 0.5*pB
  – pB = pC
  – pC = pA + pD + 0.5*pE
  – pD = 0.5*pE
  – pE = 0.5*pB
  – We also know: pA + pB + pC + pD + pE = 1



Source

Target

	a	b	c	d	e
a	0	1	0	0	0
b	0	0	1	0	0
c	1	0	0	1	1
d	0	0	0	0	1
e	0	1	0	0	0

Adjacency Matrix

Source=j

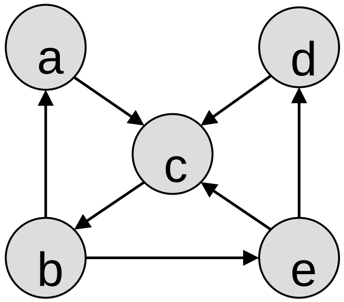
Target=i

	a	b	c	d	e
a	0	0.5	0	0	0
b	0	0	1	0	0
c	1	0	0	1	0.5
d	0	0	0	0	0.5
e	0	0.5	0	0	0

Weighted Adjacency Matrix
[Divide by $(a_{i,j})$/degree(j)]

PageRank

- System of Linear Equations
  - pA = 0.5*pB
  - pB = pC
  - pC = pA + pD + 0.5*pE
  - pD = 0.5*pE
  - pE = 0.5*pB
  - We also know: pA + pB + pC + pD + pE = 1
- If you know something about linear algebra, you know we can write these equations in matrix form as a linear system
  - Ax = y



Source=j

Target=i

	a	b	c	d	e
a	0	0.5	0	0	0
b	0	0	1	0	0
c	1	0	0	1	0.5
d	0	0	0	0	0.5
e	0	0.5	0	0	0

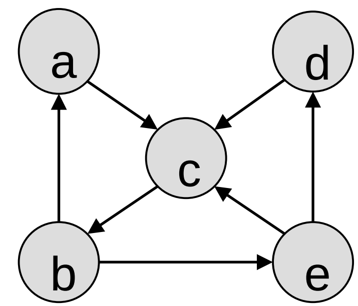
Weighted Adjacency Matrix
[Divide by $(a_{i,j})$/degree(j)]

$$\begin{vmatrix} 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0 & 0 \end{vmatrix} * \begin{vmatrix} pA \\ pB \\ pC \\ pD \\ pE \end{vmatrix}$$

$$\begin{vmatrix} 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0 & 0 \end{vmatrix} * \begin{vmatrix} pA \\ pB \\ pC \\ pD \\ pE \end{vmatrix} = \begin{vmatrix} pA = 0.5PB \\ pB = pC \\ pC = pA+pD+0.5*pE \\ pD = 0.5*pE \\ pE = 0.5*pB \end{vmatrix}$$

PageRank

- But remember we want the steady state solution
  - The solution where the probabilities don't change from one step to the next
- So we want a solution to: **A**p = p
- We can:
  - Use a linear system solver (Gaussian elimination)
  - Or we can just seed the problem with some probabilities and then just iterate until the solution settles down



$$\begin{vmatrix} 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0 & 0 \end{vmatrix} * \begin{vmatrix} pA \\ pB \\ pC \\ pD \\ pE \end{vmatrix} = \begin{vmatrix} pA \\ pB \\ pC \\ pD \\ pE \end{vmatrix}$$

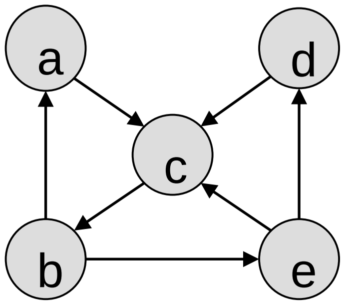
Source=j

Target=i

	a	b	c	d	e
a	0	0.5	0	0	0
b	0	0	1	0	0
c	1	0	0	1	0.5
d	0	0	0	0	0.5
e	0	0.5	0	0	0

Weighted Adjacency Matrix
[Divide by $(a_{i,j})$/degree(j)]

Iterative PageRank



- But remember we want the steady state solution
  - The solution where the probabilities don't change from one step to the next
- So we want a solution to: **A**p = p
- We can:
  - Use a linear system solver (Gaussian elimination)
  - Or we can just seed the problem with some probabilities and then just iterate until the solution settles down

Step 0 Sol. Step 1 Sol.

$$\begin{vmatrix} 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0 & 0 \end{vmatrix} * \begin{vmatrix} .2 \\ .2 \\ .2 \\ .2 \\ .2 \end{vmatrix} = \begin{vmatrix} .1 \\ .2 \\ .5 \\ .1 \\ .1 \end{vmatrix}$$

Step 29 Sol. Step 30 Sol.

$$\begin{vmatrix} 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0 & 0 \end{vmatrix} * \begin{vmatrix} ? \\ ? \\ ? \\ ? \\ ? \end{vmatrix} = \begin{vmatrix} .1507 \\ .3078 \\ .3126 \\ .0783 \\ .1507 \end{vmatrix}$$

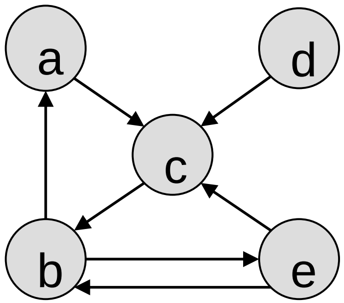
Step 1 Sol. Step 2 Sol.

$$\begin{vmatrix} 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0 & 0 \end{vmatrix} * \begin{vmatrix} .1 \\ .2 \\ .5 \\ .1 \\ .1 \end{vmatrix} = \begin{vmatrix} .1 \\ .5 \\ .25 \\ .05 \\ .1 \end{vmatrix}$$

Actual PageRank Solution from solving linear system:

$$\begin{vmatrix} .1538 \\ .3077 \\ .3077 \\ .0769 \\ .1538 \end{vmatrix}$$

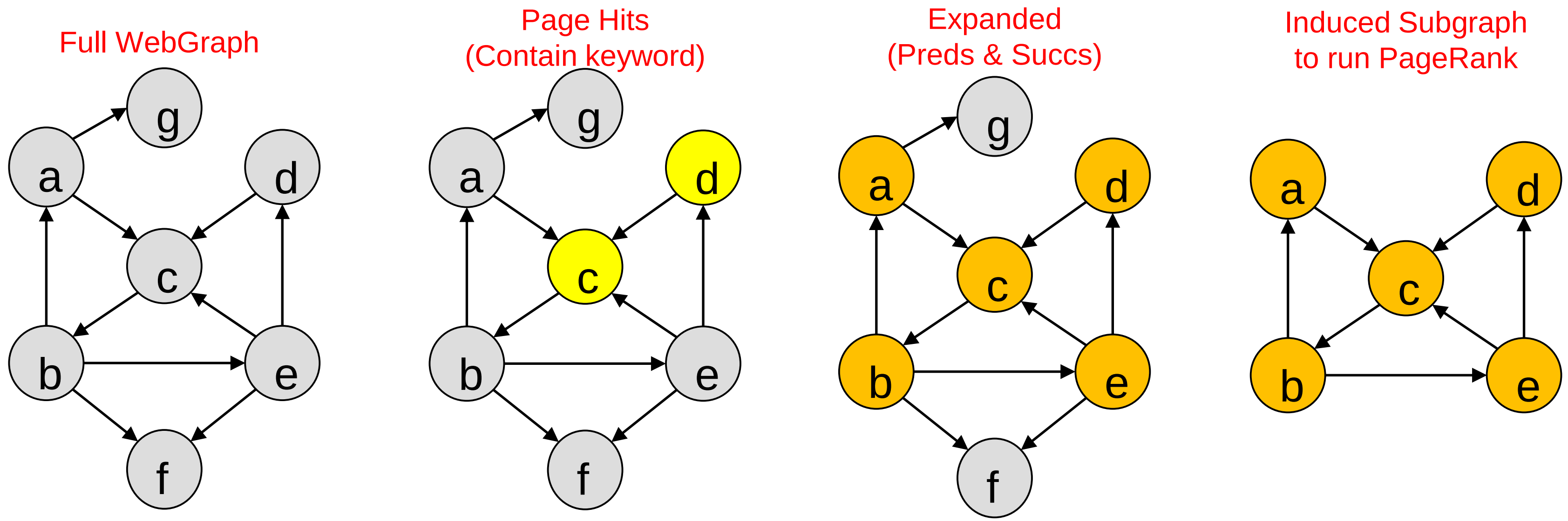
Additional Notes



- What if we change the graph and now D has no incoming links…what is its PageRank?
  - 0
- Most PR algorithms add a probability that someone just enters that URL (i.e. enters the graph at that node)
  - Usually define something called the damping factor, α (often chosen around 0.15)
  - Probability of randomly starting or jumping somewhere = 1-α
- So at each time step the next PR value for node i is given as:
  - $\Pr(i) = \frac{\alpha}{N} + (1-\alpha) * \sum_{j \in Pred(i)} \frac{\Pr(j)}{OutDeg(j)}$
  - N is the total number of vertices
  - Usually run 30 or so update steps
  - Start each Pr(i) = 1/N

In a Web Search Setting

- Given some search keywords we could find the pages that have that matching keywords
- We often expand that set of pages by including all successors and predecessors of those pages
  - Include all pages that are within a radius of 1 of the pages that actually have the keyword
- Now consider that set of pages and the subgraph that it induces
- Run PageRank on that subgraph

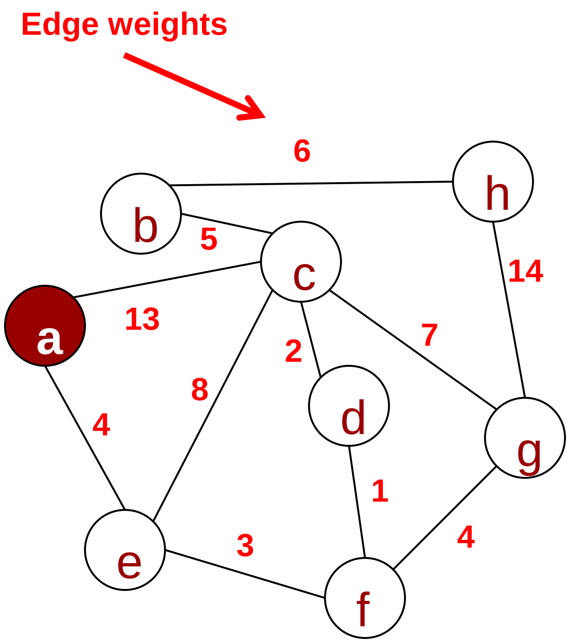


Dijkstra's Algorithm

SINGLE-SOURCE SHORTEST PATH (SSSP)

SSSP

- Let us associate a 'weight' with each edge
  - Could be physical distance, cost of using the link, etc.
- Find the shortest path from a source node, 'a' to all other nodes

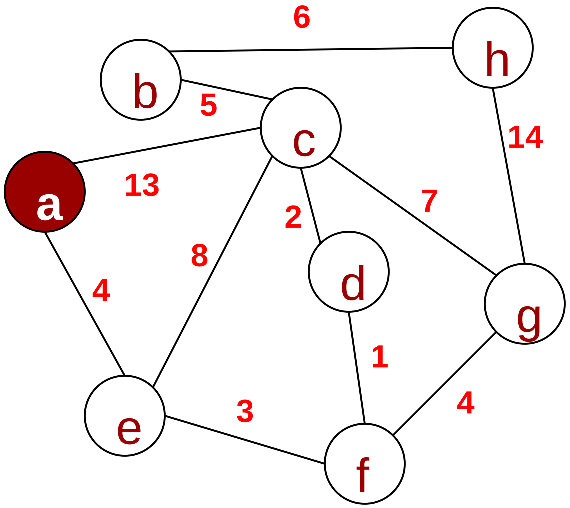


List of Vertices / Adjacency Lists

a	(c,13),(e,4)
b	(c,5),(h,6)
c	(a,13),(b,5),(d,2),(e,8),(g,7)
d	(c,2),(f,1)
e	(a,4),(c,8),(f,3)
f	(d,1),(e,3),(g,4)
g	(c,7),(f,4),(h,14)
h	(b,6),(g,14)

SSSP

- What is the shortest distance from 'a' to all other vertices?
- How would you go about computing those distances?



List of Vertices

a	(c,13),(e,4)
b	(c,5),(h,6)
c	(a,13),(b,5),(d,2),(e,8),(g,7)
d	(c,2),(f,1)
e	(a,4),(c,8),(f,3)
f	(d,1),(e,3),(g,4)
g	(c,7),(f,4),(h,14)
h	(b,6),(g,14)

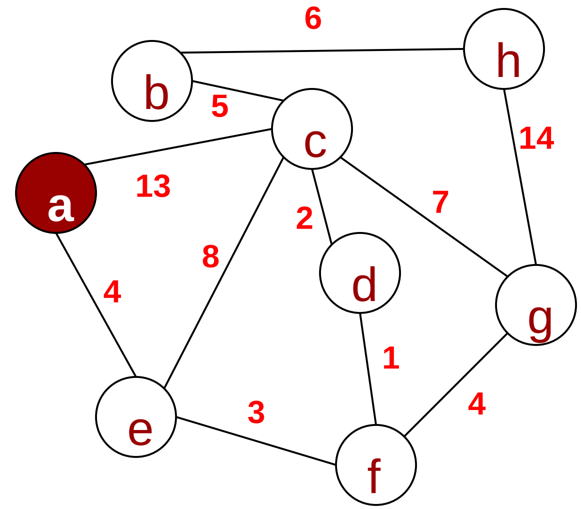
Adjacency Lists

List of Vertices

Vert	Dist
a	0
b	
c	
d	
e	
f	
g	
h	

Dijkstra's Algorithm

- Dijkstra's algorithm is similar to a BFS but pulls out the smallest distance vertex (from the source) rather than pulling vertices out in FIFO order (as in BFS)
- Maintain a data structure that you can identify shortly
  - We'll show it as a table of all vertices with their currently 'known' distance from the source
    - Initially, a has dist=0
    - All others = infinite distance

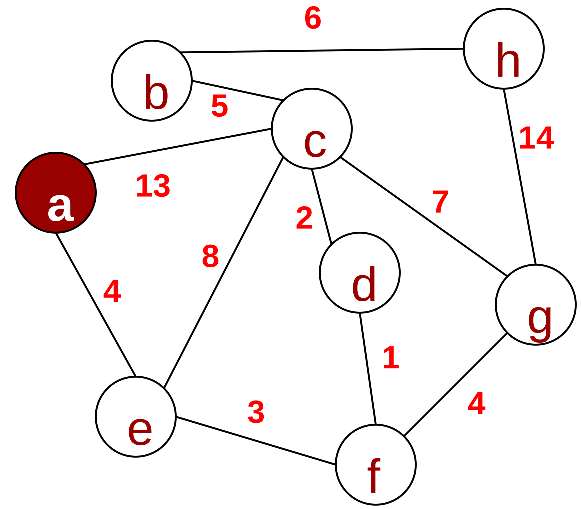


List of Vertices

Vert	Dist
a	0
b	inf
c	inf
d	inf
e	inf
f	inf
g	inf
h	inf

Dijkstra's Algorithm

```
SSSP(G, s)
  PQ = empty PQ
  s.dist = 0;  s.pred = NULL
  PQ.insert(s)
  For all v in vertices
    if v != s then v.dist = inf;  PQ.insert(v)
  while PQ is not empty
    v = min(); PQ.remove_min()
    for u in neighbors(v)
      w = weight(v,u)
      if(v.dist + w < u.dist)
        u.pred = v
        u.dist = v.dist + w;
        PQ.decreaseKey(u, u.dist)
```

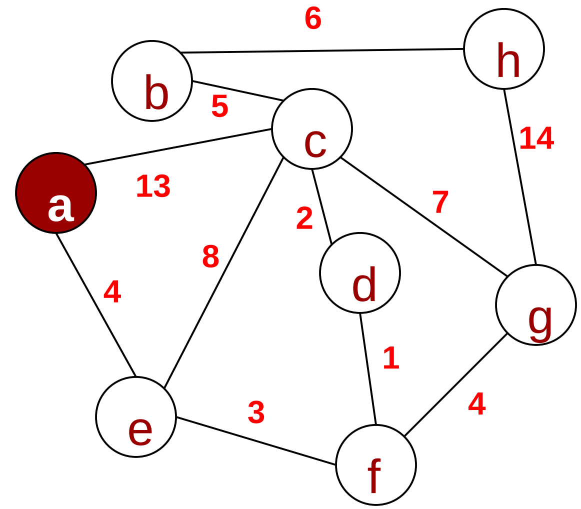


List of Vertices

Vert	Dist
a	0
b	inf
c	inf
d	inf
e	inf
f	inf
g	inf
h	inf

Dijkstra's Algorithm

```
SSSP(G, s)
  PQ = empty PQ
  s.dist = 0;  s.pred = NULL
  PQ.insert(s)
  For all v in vertices
     if v != s then v.dist = inf;  PQ.insert(v)
  while PQ is not empty
     v = min(); PQ.remove_min()
     for u in neighbors(v)
        w = weight(v,u)
        if(v.dist + w < u.dist)
           u.pred = v
           u.dist = v.dist + w;
           PQ.decreaseKey(u, u.dist)
```



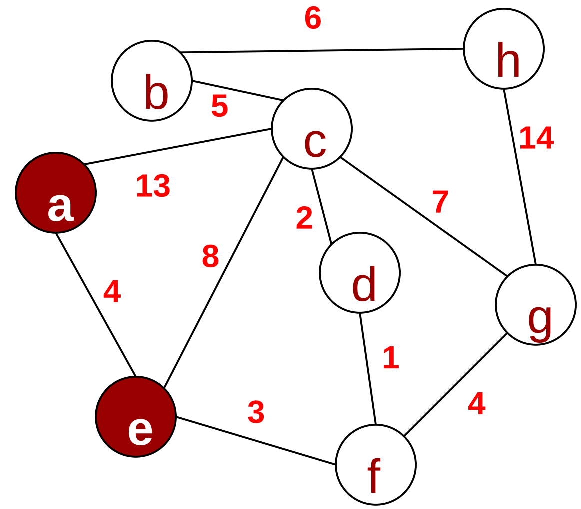
List of Vertices

Vert	Dist	
a	0	
b	inf	
c	inf	13
d	inf	
e	inf	4
f	inf	
g	inf	
h	inf	

v=a

Dijkstra's Algorithm

```
SSSP(G, s)
  PQ = empty PQ
  s.dist = 0;  s.pred = NULL
  PQ.insert(s)
  For all v in vertices
     if v != s then v.dist = inf;  PQ.insert(v)
  while PQ is not empty
     v = min(); PQ.remove_min()
     for u in neighbors(v)
         w = weight(v,u)
         if(v.dist + w < u.dist)
             u.pred = v
             u.dist = v.dist + w;
             PQ.decreaseKey(u, u.dist)
```



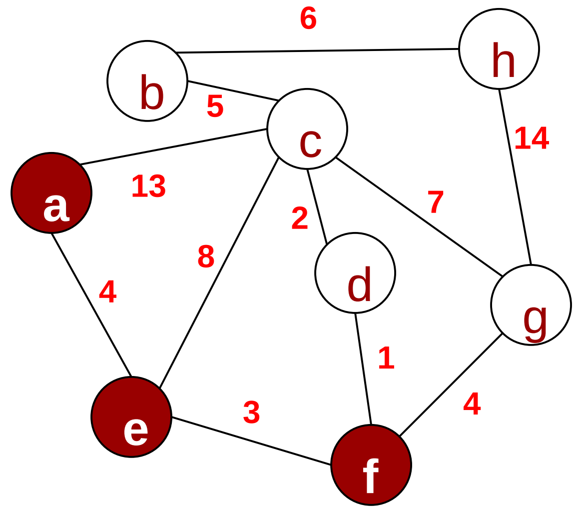
List of Vertices

Vert	Dist	
a	0	
b	inf	
c	13	12
d	inf	
e	4	
f	inf	7
g	inf	
h	inf	

v=e

Dijkstra's Algorithm

```
SSSP(G, s)
  PQ = empty PQ
  s.dist = 0;  s.pred = NULL
  PQ.insert(s)
  For all v in vertices
     if v != s then v.dist = inf;  PQ.insert(v)
  while PQ is not empty
     v = min(); PQ.remove_min()
     for u in neighbors(v)
        w = weight(v,u)
        if(v.dist + w < u.dist)
           u.pred = v
           u.dist = v.dist + w;
           PQ.decreaseKey(u, u.dist)
```



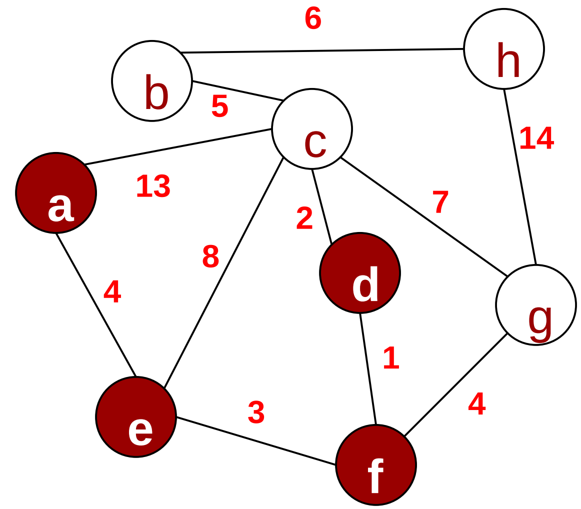
List of Vertices

Vert	Dist	
a	0	
b	inf	
c	12	
d	inf	8
e	4	
f	7	
g	inf	11
h	inf	

v=f

Dijkstra's Algorithm

```
SSSP(G, s)
  PQ = empty PQ
  s.dist = 0;  s.pred = NULL
  PQ.insert(s)
  For all v in vertices
     if v != s then v.dist = inf;  PQ.insert(v)
  while PQ is not empty
     v = min(); PQ.remove_min()
     for u in neighbors(v)
         w = weight(v,u)
         if(v.dist + w < u.dist)
             u.pred = v
             u.dist = v.dist + w;
             PQ.decreaseKey(u, u.dist)
```



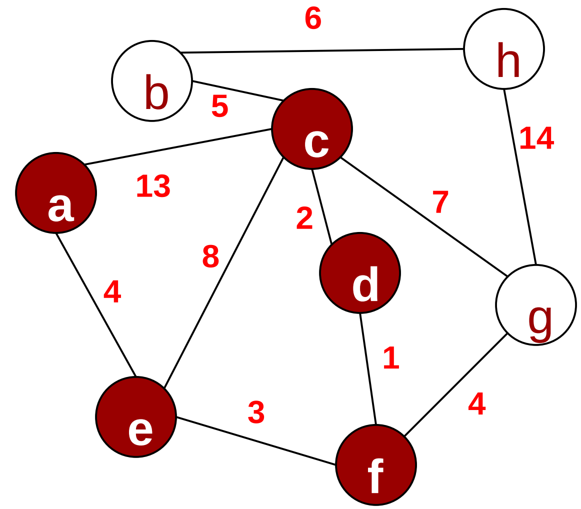
List of Vertices

Vert	Dist	
a	0	
b	inf	
c	12	10
d	8	
e	4	
f	7	
g	11	
h	inf	

v=d

Dijkstra's Algorithm

```
SSSP(G, s)
  PQ = empty PQ
  s.dist = 0;  s.pred = NULL
  PQ.insert(s)
  For all v in vertices
    if v != s then v.dist = inf;  PQ.insert(v)
  while PQ is not empty
    v = min(); PQ.remove_min()
    for u in neighbors(v)
      w = weight(v,u)
      if(v.dist + w < u.dist)
        u.pred = v
        u.dist = v.dist + w;
        PQ.decreaseKey(u, u.dist)
```



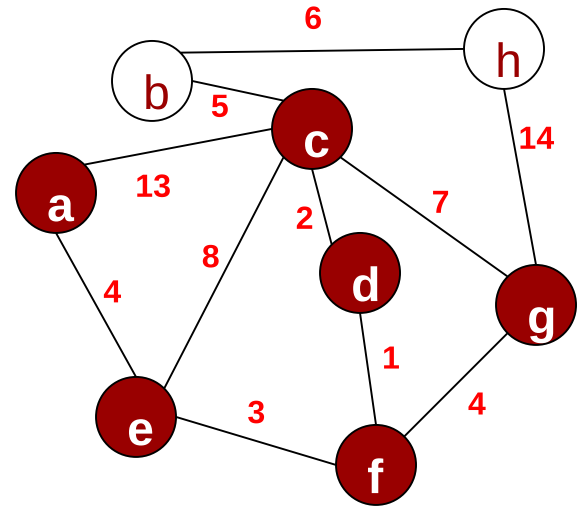
List of Vertices

Vert	Dist	
a	0	
b	inf	15
c	10	
d	8	
e	4	
f	7	
g	11	
h	inf	

v=c

Dijkstra's Algorithm

```
SSSP(G, s)
  PQ = empty PQ
  s.dist = 0;  s.pred = NULL
  PQ.insert(s)
  For all v in vertices
     if v != s then v.dist = inf;  PQ.insert(v)
  while PQ is not empty
     v = min(); PQ.remove_min()
     for u in neighbors(v)
        w = weight(v,u)
        if(v.dist + w < u.dist)
           u.pred = v
           u.dist = v.dist + w;
           PQ.decreaseKey(u, u.dist)
```



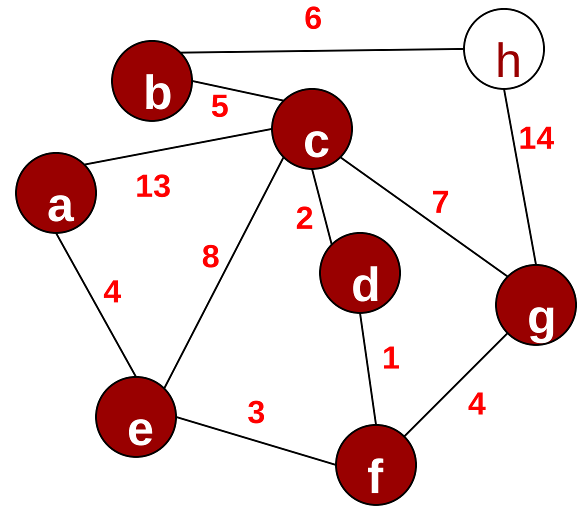
List of Vertices

Vert	Dist	
a	0	
b	15	
c	10	
d	8	
e	4	
f	7	
g	11	
h	inf	25

v=g

Dijkstra's Algorithm

```
SSSP(G, s)
  PQ = empty PQ
  s.dist = 0;  s.pred = NULL
  PQ.insert(s)
  For all v in vertices
    if v != s then v.dist = inf;  PQ.insert(v)
  while PQ is not empty
    v = min(); PQ.remove_min()
    for u in neighbors(v)
      w = weight(v,u)
      if(v.dist + w < u.dist)
        u.pred = v
        u.dist = v.dist + w;
        PQ.decreaseKey(u, u.dist)
```



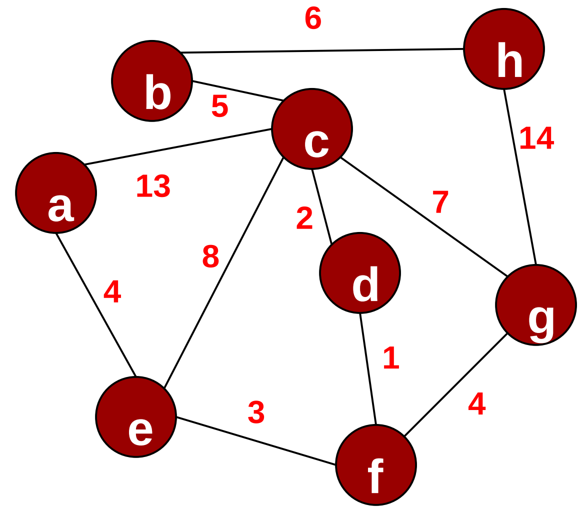
List of Vertices

Vert	Dist	
a	0	
b	15	
c	10	
d	8	
e	4	
f	7	
g	11	
h	25	21

v=b

Dijkstra's Algorithm

```
SSSP(G, s)
  PQ = empty PQ
  s.dist = 0;  s.pred = NULL
  PQ.insert(s)
  For all v in vertices
    if v != s then v.dist = inf;  PQ.insert(v)
  while PQ is not empty
    v = min(); PQ.remove_min()
    for u in neighbors(v)
      w = weight(v,u)
      if(v.dist + w < u.dist)
        u.pred = v
        u.dist = v.dist + w;
        PQ.decreaseKey(u, u.dist)
```



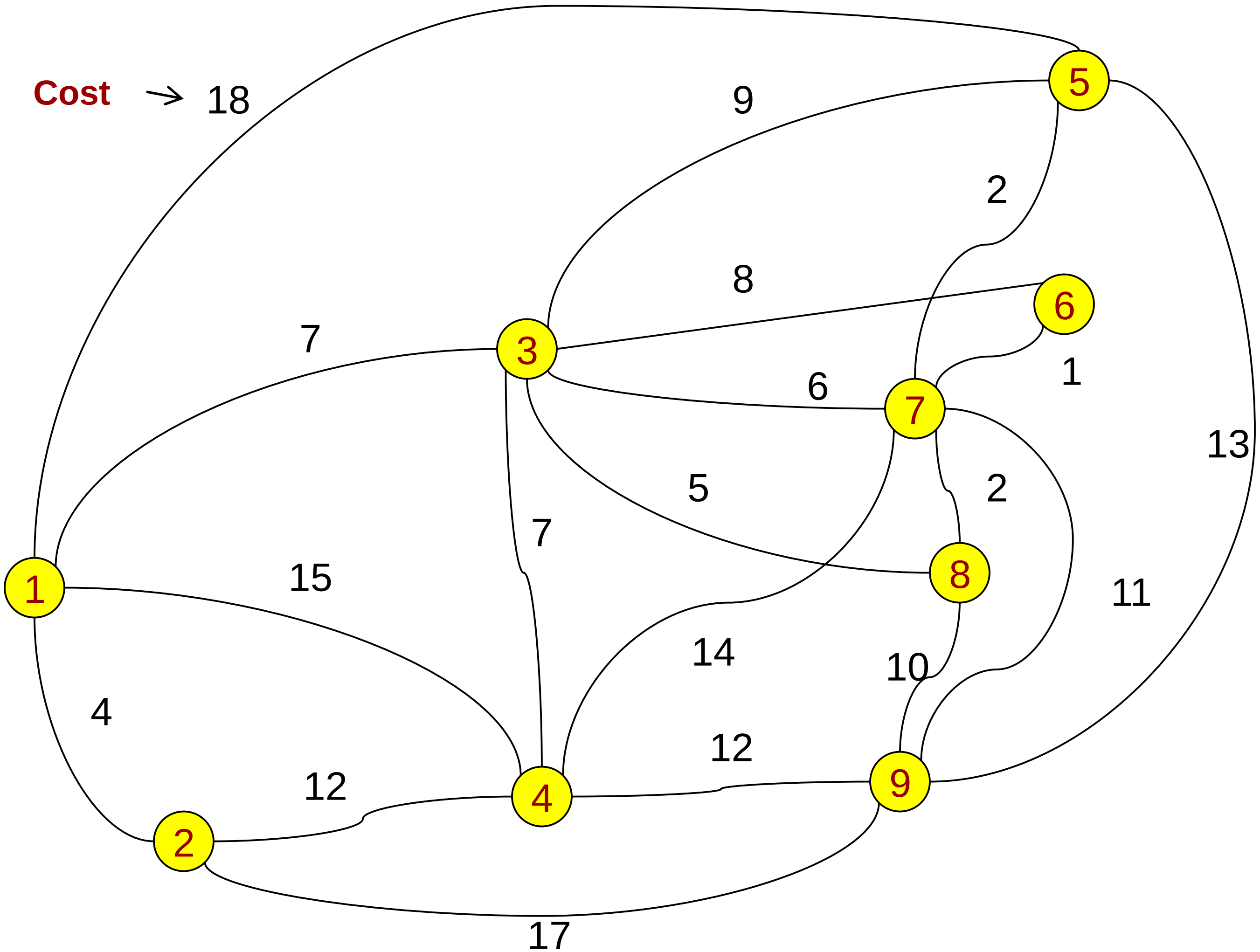
List of Vertices

Vert	Dist
a	0
b	15
c	10
d	8
e	4
f	7
g	11
h	21

v=h

Another Example

- Try another example of Dijkstra's

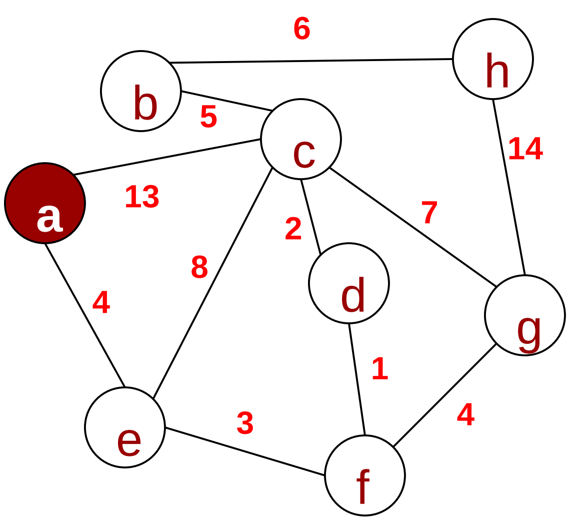


List of Vertices

Vert	Dist
1	0
2	-
3	-
4	-
5	-
6	-
7	-
8	-
9	-

Analysis

- What is the loop invariant? What can I say about the vertex I pull out from the PQ?
  - It is guaranteed that there is no shorter path to that vertex
  - UNLESS: negative edge weights
- Could use induction to prove
  - When I pull the first node out (it is the start node) it's weight has to be 0 and that is definitely the shortest path to itself
  - I then "relax" (i.e. decrease) the distance to neighbors it connects to and the next node I pull out would be the neighbor with the shortest distance from the start
    - Could there be shorter path to that node?
  - No, because any other path would use some other edge from the start which would have to have a larger weight



Dijkstra's Run-time Analysis

- What is the run-time of Dijkstra's algorithm?
- How many times do you execute the while loop on 8?
- How many total times do you execute the for loop on 10?

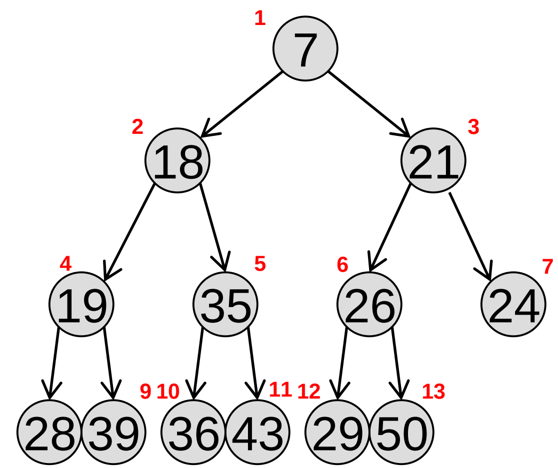
```
1.    SSSP(G, s)
2.      PQ = empty PQ
3.      s.dist = 0;  s.pred = NULL
4.      PQ.insert(s)
5.      For all v in vertices
6.          if v != s then v.dist = inf;
7.          PQ.insert(v)
8.      while PQ is not empty
9.          v = min(); PQ.remove_min()
10.         for u in neighbors(v)
11.             w = weight(v,u)
12.             if(v.dist + w < u.dist)
13.                 u.pred = v
14.                 u.dist = v.dist + w;
15.                 PQ.decreaseKey(u, u.dist)
```

Dijkstra's Run-time Analysis

- What is the run-time of Dijkstra's algorithm?
- How many times do you execute the while loop on 8?
  - V total times because once you pull a node out each iteration that node's distance is guaranteed to be the shortest distance and will never be put back in the PQ
  - What does each call to remove_min() cost…
  - …log(V) [at most V items in PQ]
- How many total times do you execute the for loop on 10?
  - E total times: Visit each vertex's neighbors
  - Each iteration may call decreaseKey() which is log(V)
- Total runtime = V*log(V) + E*log(V) = (V+E)*log(V)
  - This is usually dominated by E*log(V)

```
1.    SSSP(G, s)
2.      PQ = empty PQ
3.      s.dist = 0;  s.pred = NULL
4.      PQ.insert(s)
5.      For all v in vertices
6.          if v != s then v.dist = inf;
7.          PQ.insert(v)
8.      while PQ is not empty
9.          v = min(); PQ.remove_min()
10.         for u in neighbors(v)
11.             w = weight(v,u)
12.             if(v.dist + w < u.dist)
13.                 u.pred = v
14.                 u.dist = v.dist + w;
15.                 PQ.decreaseKey(u, u.dist)
```

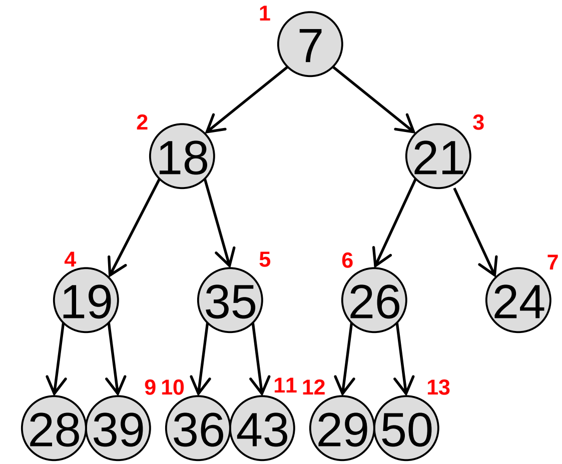
Tangent on Heaps/PQs



- Suppose min-heaps
  - Though everything we're about to say is true for max heaps but for increasing values
- We know insert/remove is log(n) for a heap
- What if we want to decrease a value already in the heap…
  - Example: Decrease 26 to 9
  - Could we find 26 easily?
    - No requires a linear search through the array/heap => O(n)
  - Once we find it could we adjust it easily?
    - Yes, just promote it until it is in the right location => O(log n)
- So currently decrease-key() would cost O(n) + O(log n) = O(n)
- Can we do better?

Tangent on Heaps/PQs

- Can we provide a decrease-key() that runs in O(log n) and not O(n)
  - Remember we'd have to first find then promote
- We need to know where items sit in the heap
  - Essentially we want to quickly know the location given the key (i.e. Map key => location)
  - Unfortunately storing the heap as an array does just the opposite (maps location => key)
- What if we maintained an alternative map that did provide the reverse indexing
  - Then I could find where the key sits and then promote it
- If I keep that map as a balanced BST can I achieve O(log n) decreaseKey() time?
  - No! each promotion swap requires update your location and your parents
  - O(log n) swaps each requiring lookup(s) in the location map [O(log n)] yielding O($\log^2$(n))



Heap Array

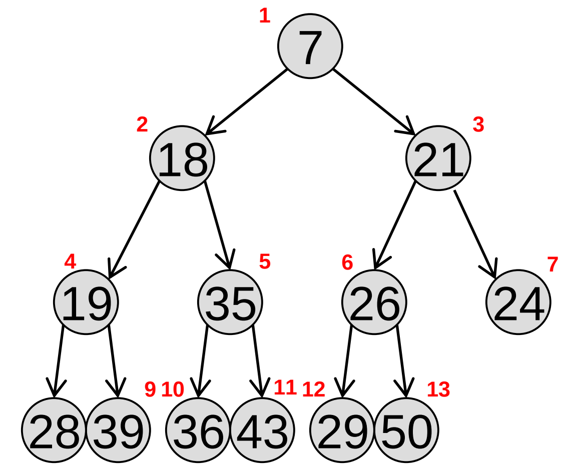
0	1	2	3	4	5	6	7	8	9	10	11	12	13
em	7	18	21	19	35	26	24	28	39	36	43	29	50

Map of key to loc.

em	7	18	21	19	35	26	24	28	39	36	43	29	50
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Tangent on Heaps/PQs

- Am I out of luck then?
- No, try a hash map
  - O(1) lookup
- Now each swap/promotion up the heap only costs O(1) and thus I have:
  - Find => O(1)
    - Using the hashmap
  - Promote => O(log n)
    - Bubble up at most log(n) levels with each level incurring O(1) updates of locations in the hashmap
- Decrease-key() is an important operation in the next algorithm we'll look at



	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Heap Array	em	7	18	21	19	35	26	24	28	39	36	43	29	50

	em	7	18	21	19	35	26	24	28	39	36	43	29	50
Map of key to loc.	0	1	2	3	4	5	6	7	8	9	10	11	12	13

A* Search Algorithm

ALGORITHM HIGHLIGHT

Search Methods

- Many systems require searching for goal states
  - Path Planning
    - Roomba Vacuum
    - Mapquest/Google Maps
    - Games!!
  - Optimization Problems
    - Find the optimal solution to a problem with many constraints

Search Applied to 8-Tile Game

- 8-Tile Puzzle
  - 3x3 grid with one blank space
  - With a series of moves, get the tiles in sequential order
  - Goal state:

	1	2
3	4	5
6	7	8

HW6 Goal State

1	2	3
4	5	6
7	8	

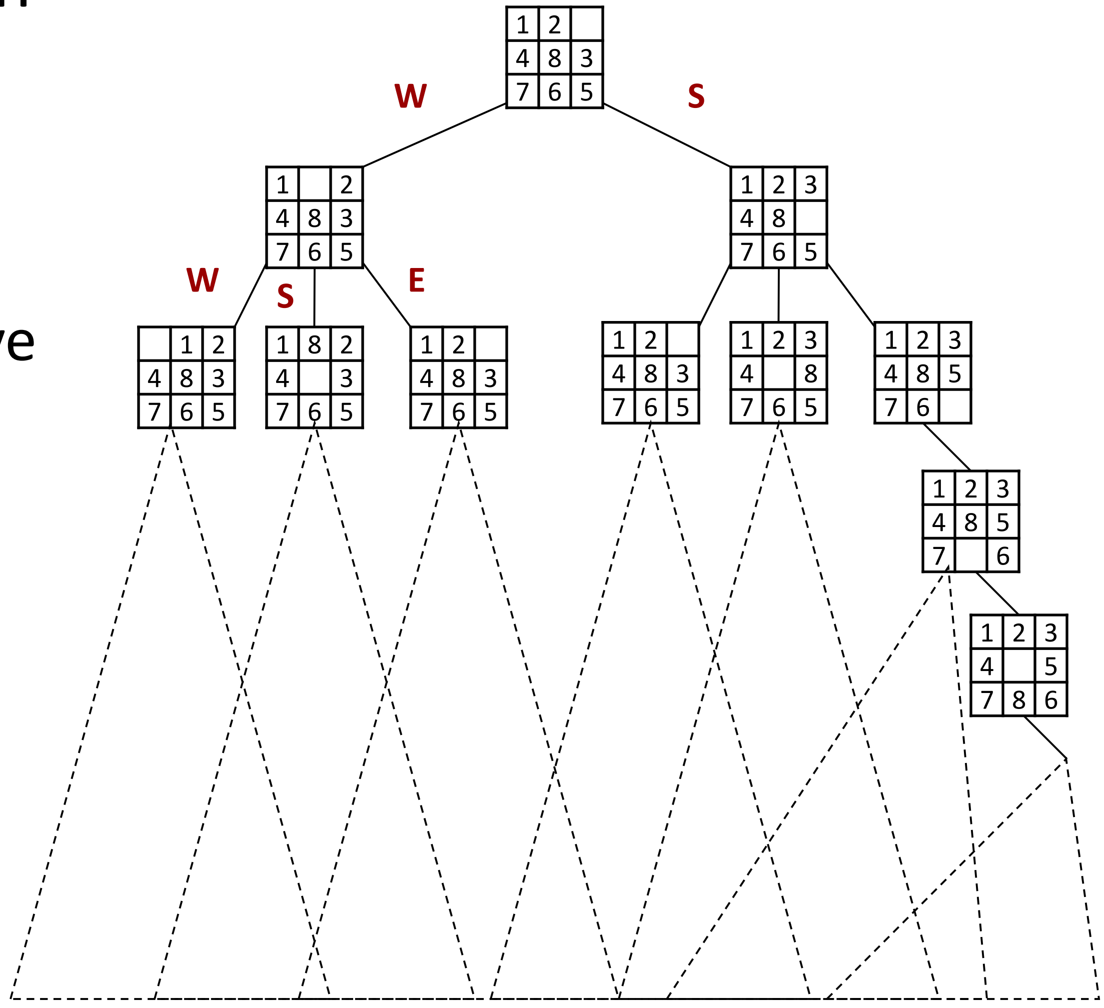
Goal State for these slides

Search Methods

- **Brute-Force Search**: When you don't know where the answer is, just search all possibilities until you find it.
- **Heuristic Search**: A heuristic is a "rule of thumb". An example is in a chess game, to decide which move to make, count the values of the pieces left for your opponent. Use that value to "score" the possible moves you can make.
  - Heuristics are not perfect measures, they are quick computations to give an approximation (e.g. may not take into account "delayed gratification" or "setting up an opponent")

Brute Force Search

- Brute Force Search Tree
  - Generate all possible moves
  - Explore each move despite its proximity to the goal node



Heuristics

- Heuristics are "scores" of how close a state is to the goal (usually, lower = better)
- These scores must be easy to compute (i.e. simpler than solving the problem)
- Heuristics can usually be developed by simplifying the constraints on a problem
- Heuristics for 8-tile puzzle
  - # of tiles out of place
    - Simplified problem: If we could just pick a tile up and put it in its correct place
  - Total x-, y- distance of each tile from its correct location (Manhattan distance)
    - Simplified problem if tiles could stack on top of each other / hop over each other

1	8	3
4	5	6
2	7	

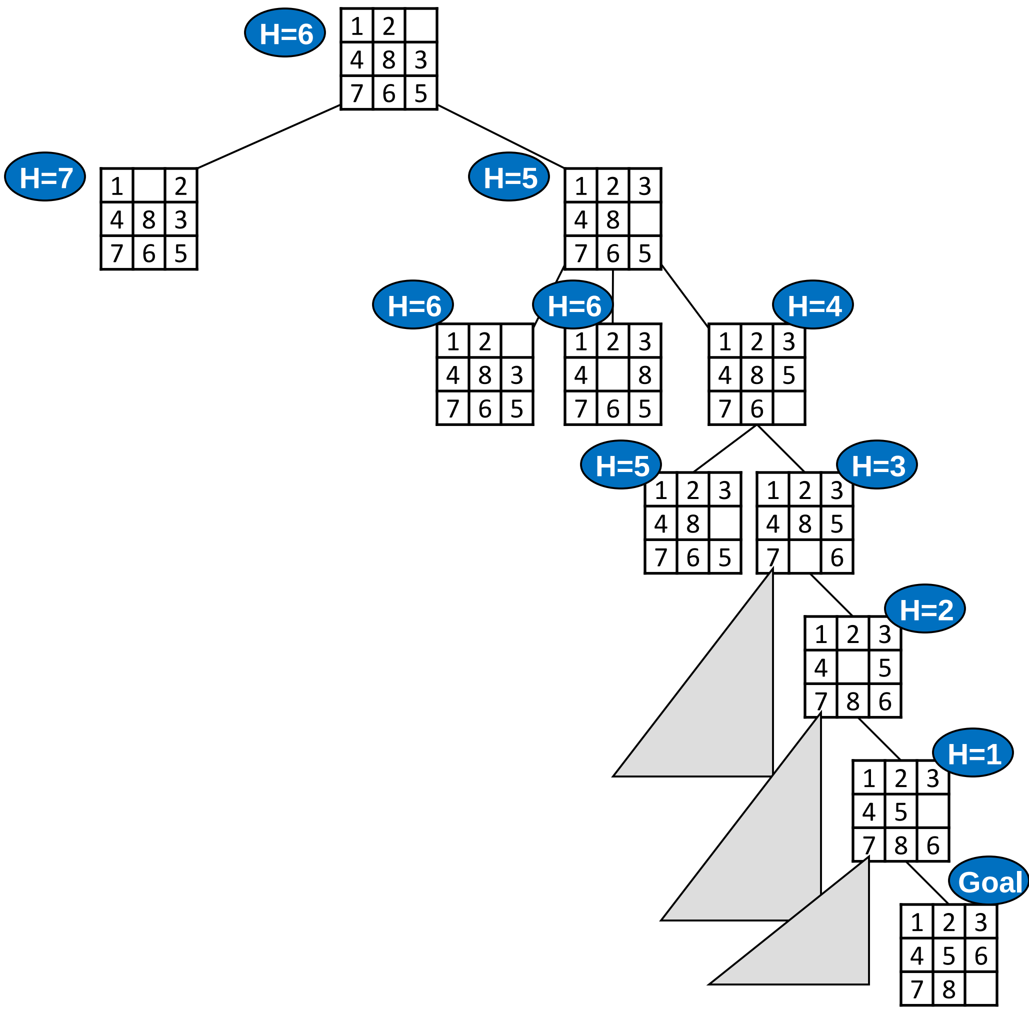
# of Tiles out of Place = 3

1	8	3
4	5	6
2	7	

Total x-/y- distance = 6

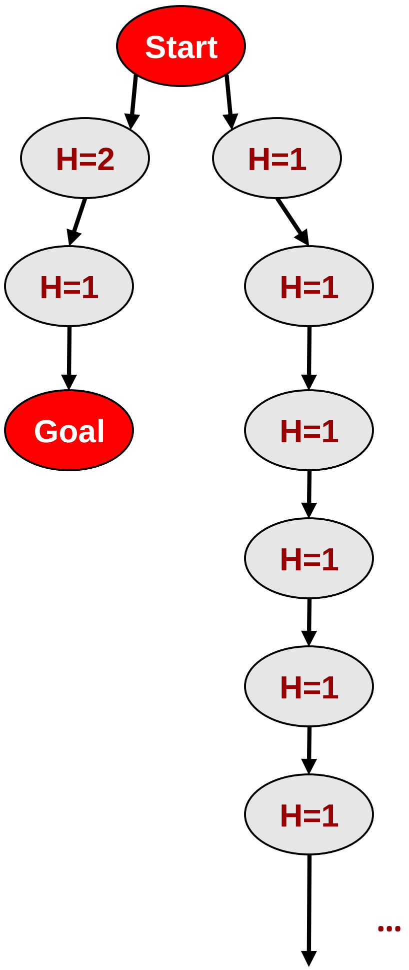
Heuristic Search

- Heuristic Search Tree
  - Use total x-/y-distance (Manhattan distance) heuristic
  - Explore the lowest scored states



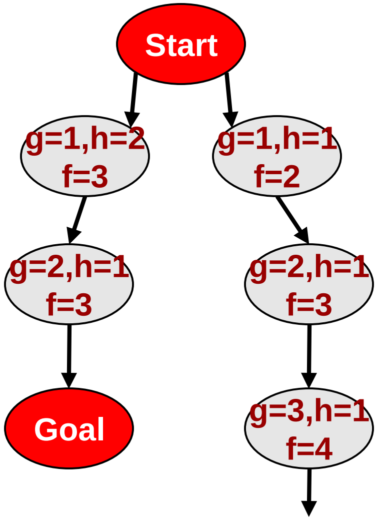
Caution About Heuristics

- Heuristics are just estimates and thus could be wrong
- Sometimes pursuing lowest heuristic score leads to a less-than optimal solution or even no solution
- Solution
  - Take # of moves from start (depth) into account



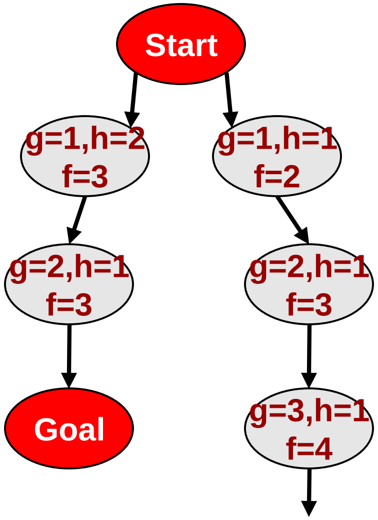
A-Star Algorithm

- Use a new metric to decide which state to explore/expand
- Define
  - h = heuristic score (same as always)
  - g = number of moves from start it took to get to current state
  - f = g + h
- As we explore states and their successors, assign each state its f-score and always explore the state with lowest f-score
- Heuristics should always underestimate the distance to the goal
  - If they do, A* guarantees optimal solutions



A-Star Algorithm

- Maintain 2 lists
  - Open list = Nodes to be explored (chosen from)
  - Closed list = Nodes already explored (already chosen)
- General A* Pseudocode



open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
    (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start; STOP!
  3b. Generate successors/neighbors of s, compute their f values, and add them to open_list if they are not in the closed_list (so we don't re-explore), or if they are already in the open list, update them if they have a smaller f value

Path-Planning w/ A* Algorithm

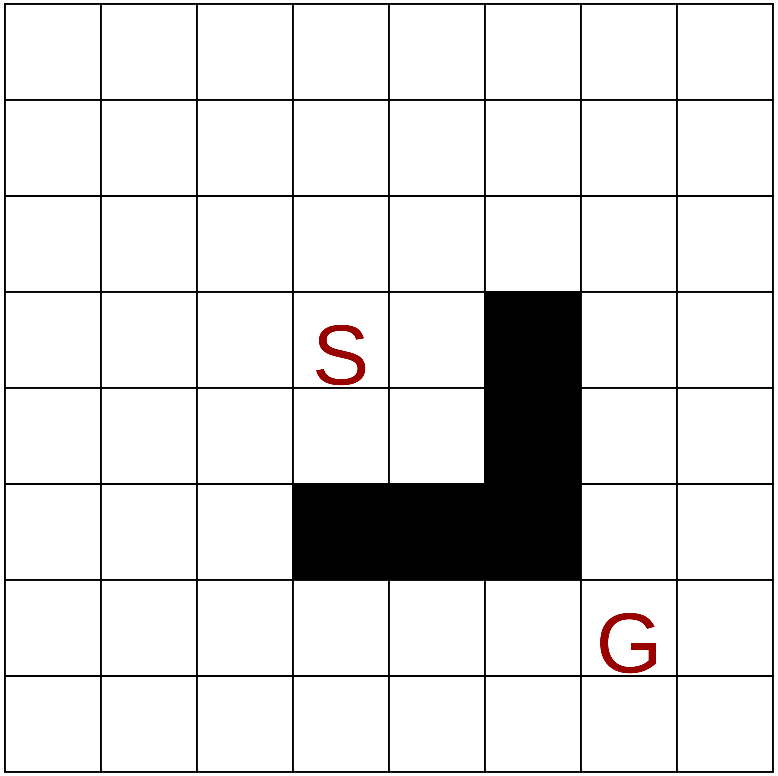
- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

**If implementing this for a programming assignment, please see the slide at the end about alternate closed-list implementation

open_list.push(Start State)
while(open_list is not empty)
1. **s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)**
2. **Add s to closed list**

3a. if s = goal node then
trace path back to start; STOP!

3b. else
Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don't re-explore), or if they are already in the open list, update them if they have a smaller f value



Closed List

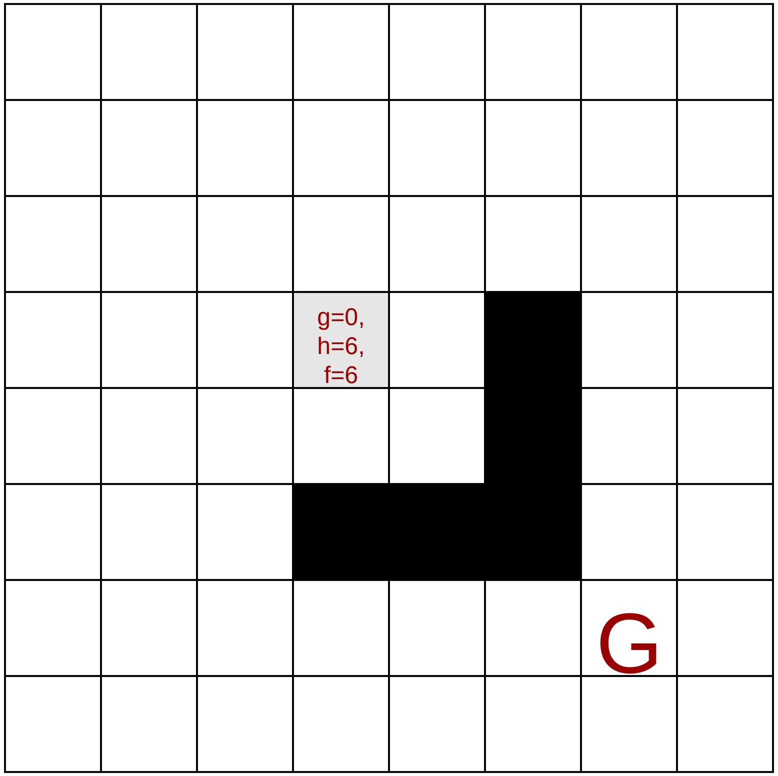
Open List

Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

**If implementing this for a programming assignment, please see the slide at the end about alternate closed-list implementation

open_list.push(Start State)
while(open_list is not empty)
1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
2. Add s to closed list
3a. if s = goal node then
trace path back to start; STOP!
3b. else
Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don't re-explore), or if they are already in the open list, update them if they have a smaller f value



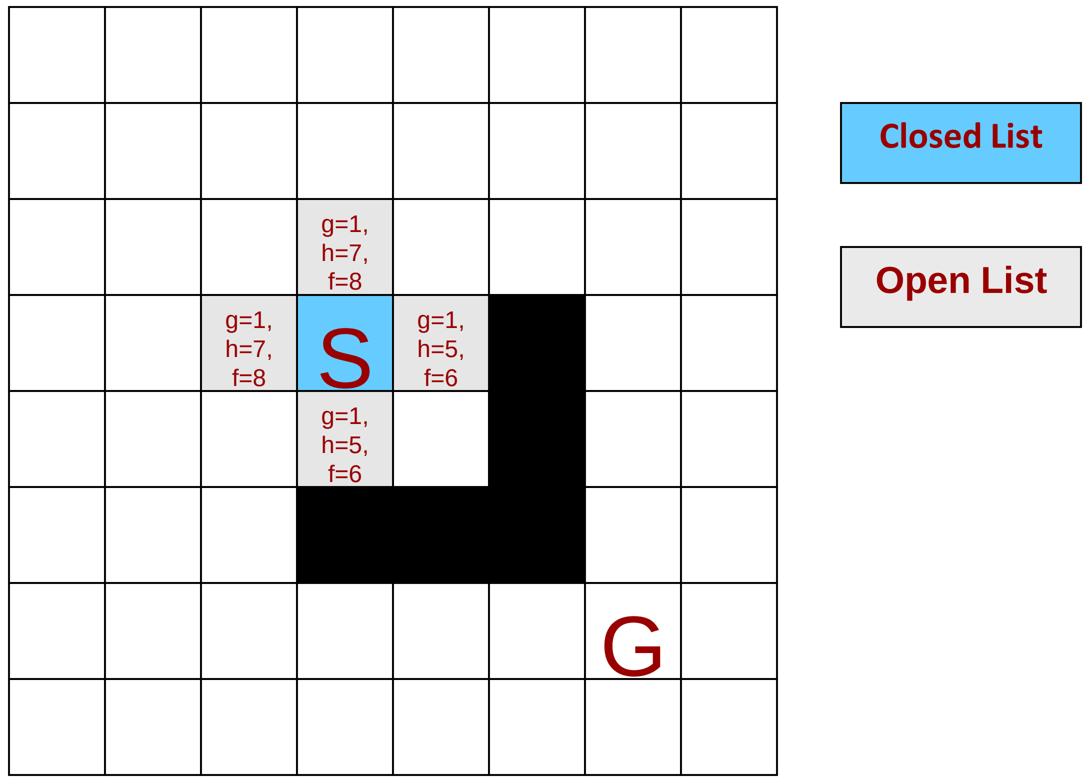
Closed List

Open List

Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
2. Add s to closed list
3a. if s = goal node then
    trace path back to start; STOP!
3b. else
    Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don't re-explore), or if they are already in the open list, update them if they have a smaller f value



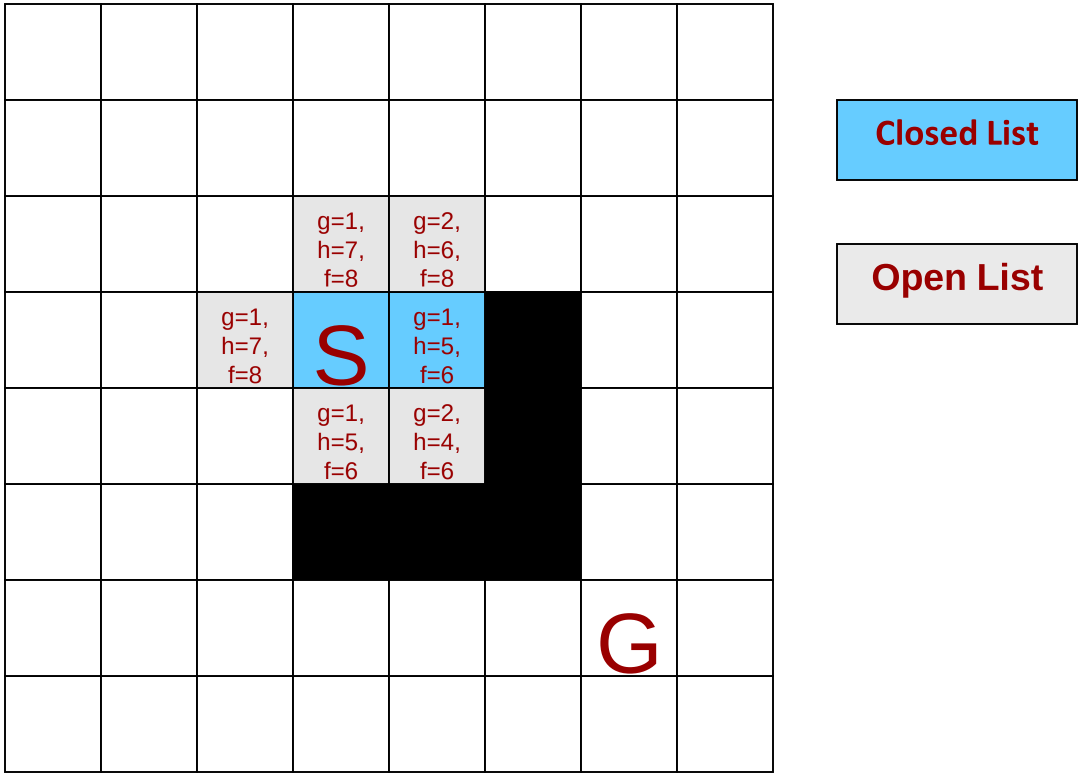
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
1. **s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)**
2. **Add s to closed list**

3a. if s = goal node then
trace path back to start; STOP!

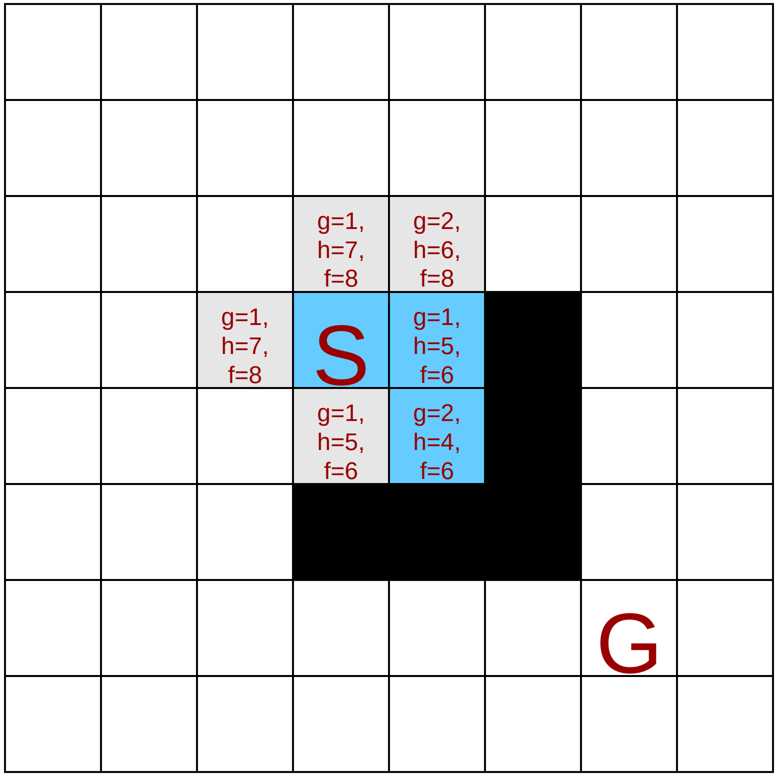
3b. else
Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don't re-explore), or if they are already in the open list, update them if they have a smaller f value



Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. **s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)**
  2. **Add s to closed list**
  3a. **if s = goal node then**
    trace path back to start; STOP!
  3b. **else**
    Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don't re-explore), or if they are already in the open list, update them if they have a smaller f value



Closed List

Open List

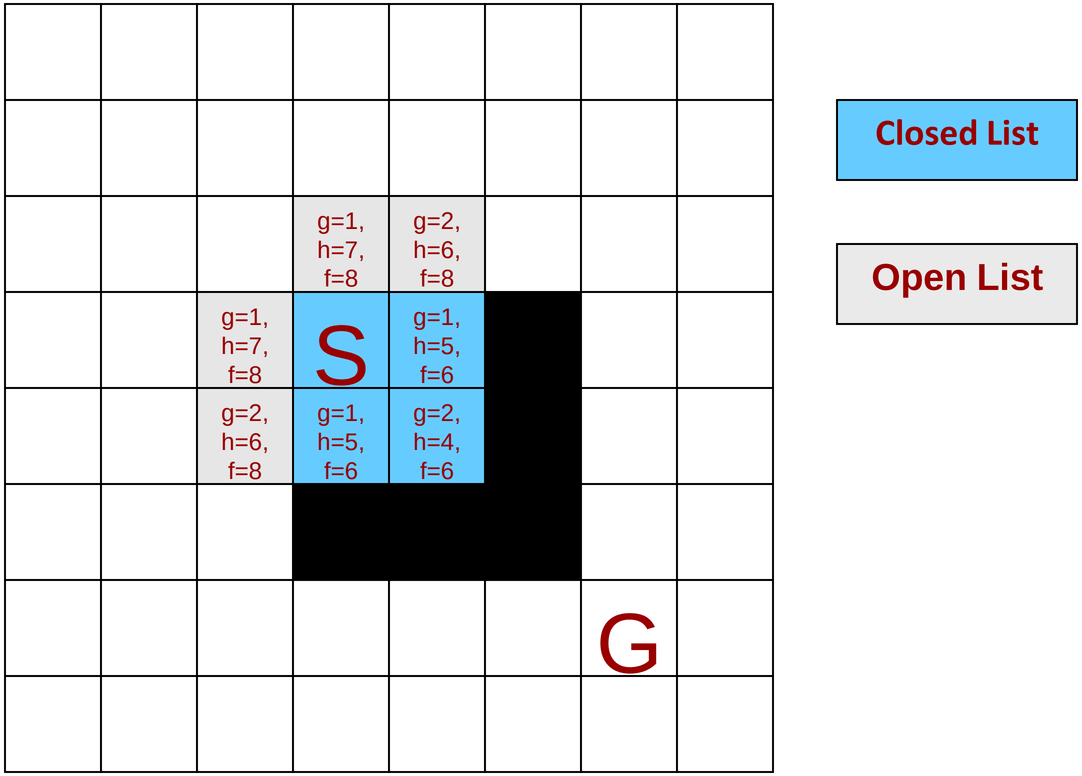
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
1. **s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)**
2. **Add s to closed list**

3a. if s = goal node then
trace path back to start; STOP!

3b. else
Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don't re-explore), or if they are already in the open list, update them if they have a smaller f value



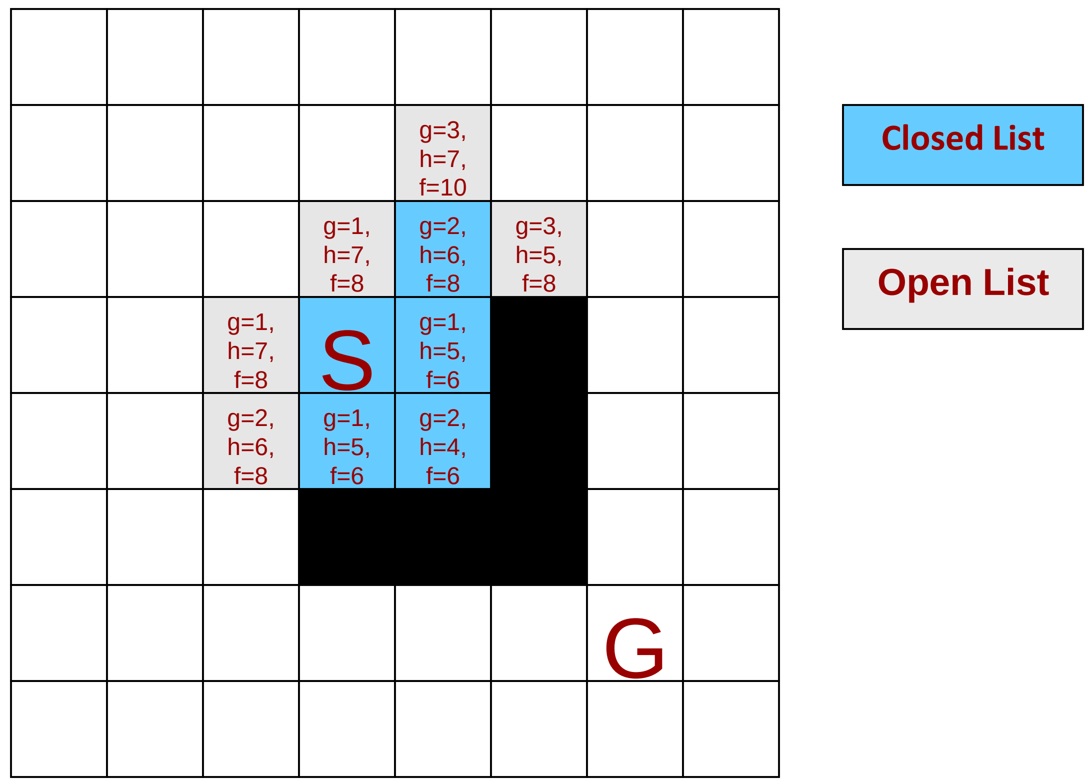
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
1. **s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)**
2. **Add s to closed list**

3a. if s = goal node then
trace path back to start; STOP!

3b. else
Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don't re-explore), or if they are already in the open list, update them if they have a smaller f value



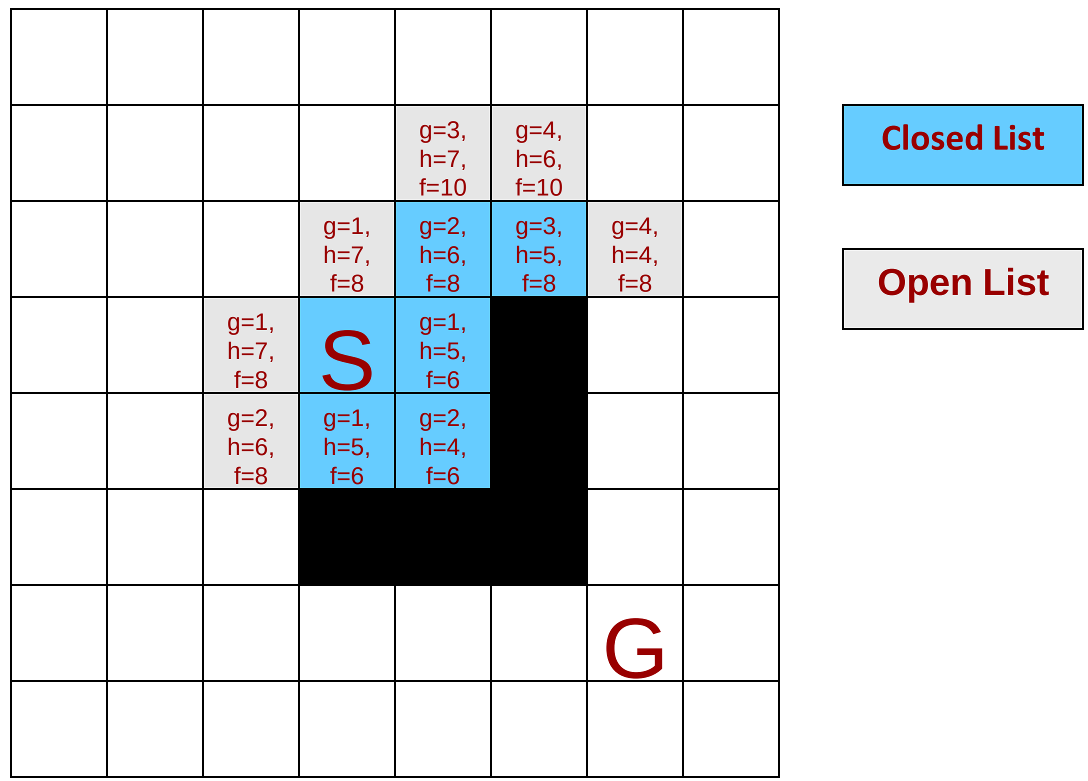
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
1. **s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)**
2. **Add s to closed list**

3a. if s = goal node then
trace path back to start; STOP!

3b. else
Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don't re-explore), or if they are already in the open list, update them if they have a smaller f value



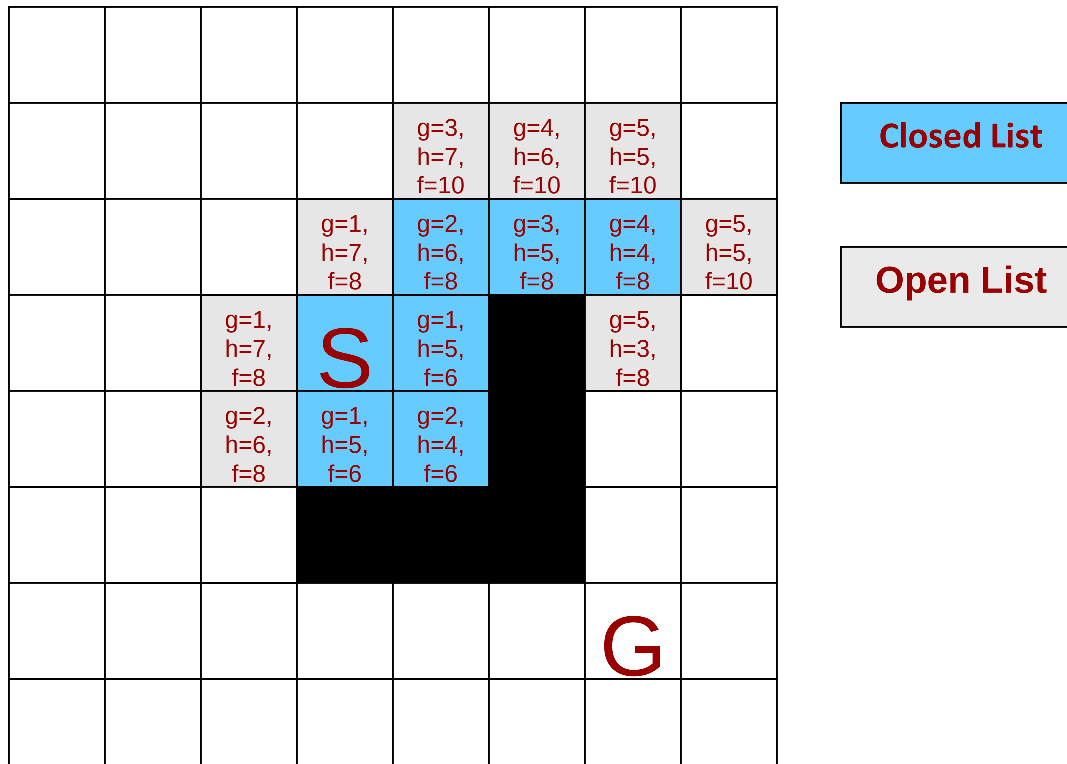
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
1. **s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)**
2. **Add s to closed list**

3a. if s = goal node then
trace path back to start; STOP!

3b. else
Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don't re-explore), or if they are already in the open list, update them if they have a smaller f value



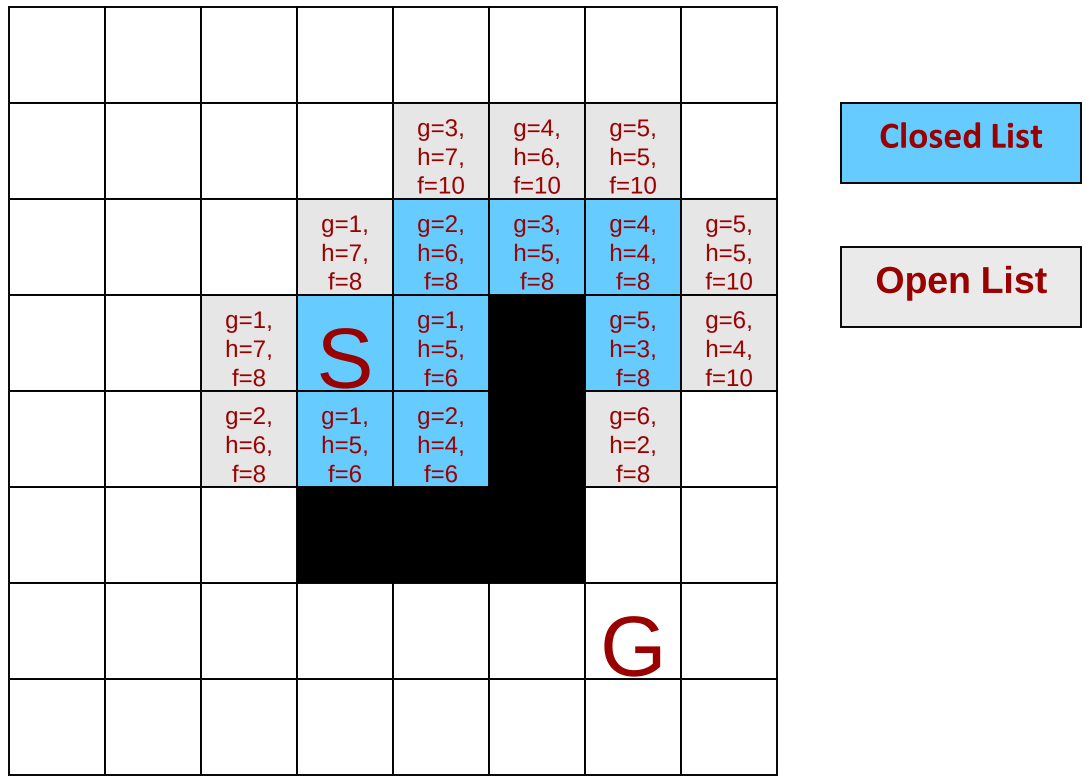
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
1. **s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)**
2. **Add s to closed list**

3a. if s = goal node then
trace path back to start; STOP!

3b. else
Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don't re-explore), or if they are already in the open list, update them if they have a smaller f value



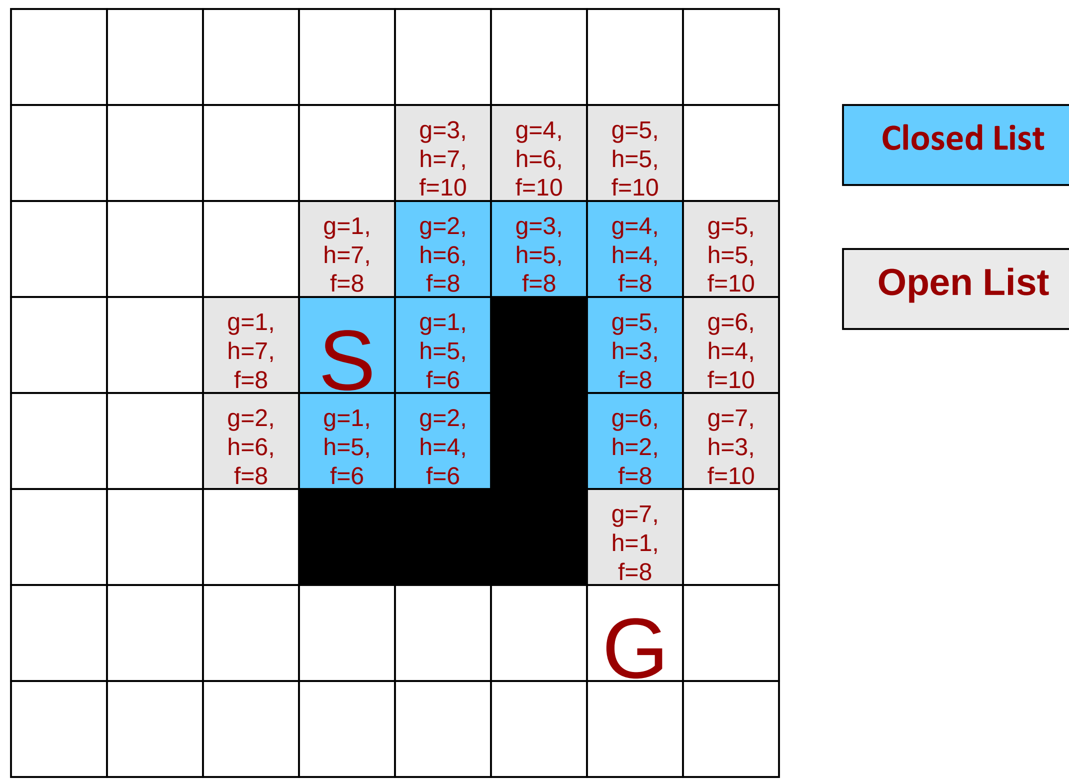
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
1. **s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)**
2. **Add s to closed list**

3a. **if s = goal node then**
   trace path back to start; STOP!

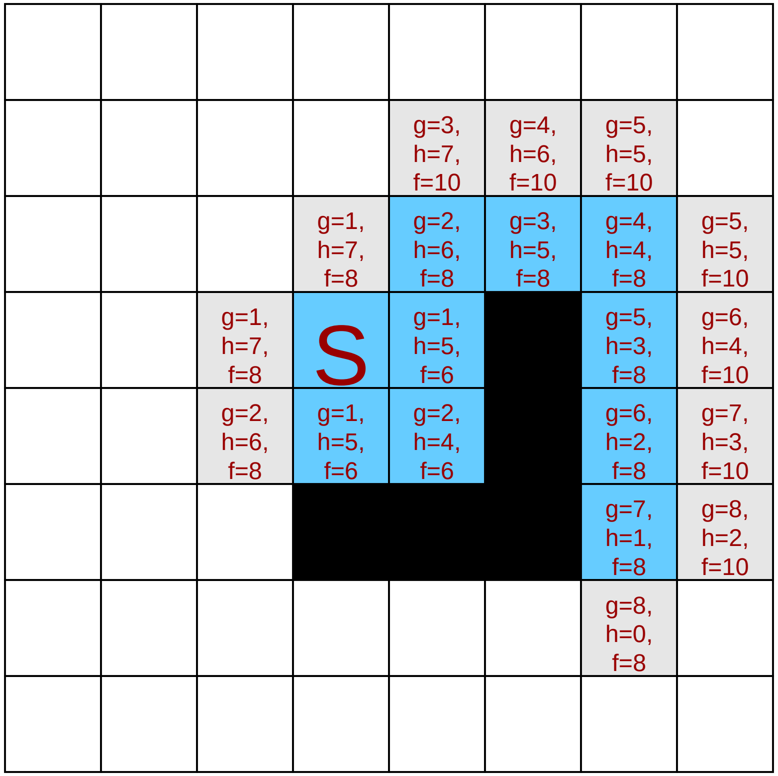
3b. **else**
   Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don't re-explore), or if they are already in the open list, update them if they have a smaller f value



Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then
    trace path back to start; STOP!
  3b. else
    Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don't re-explore), or if they are already in the open list, update them if they have a smaller f value



Closed List

Open List

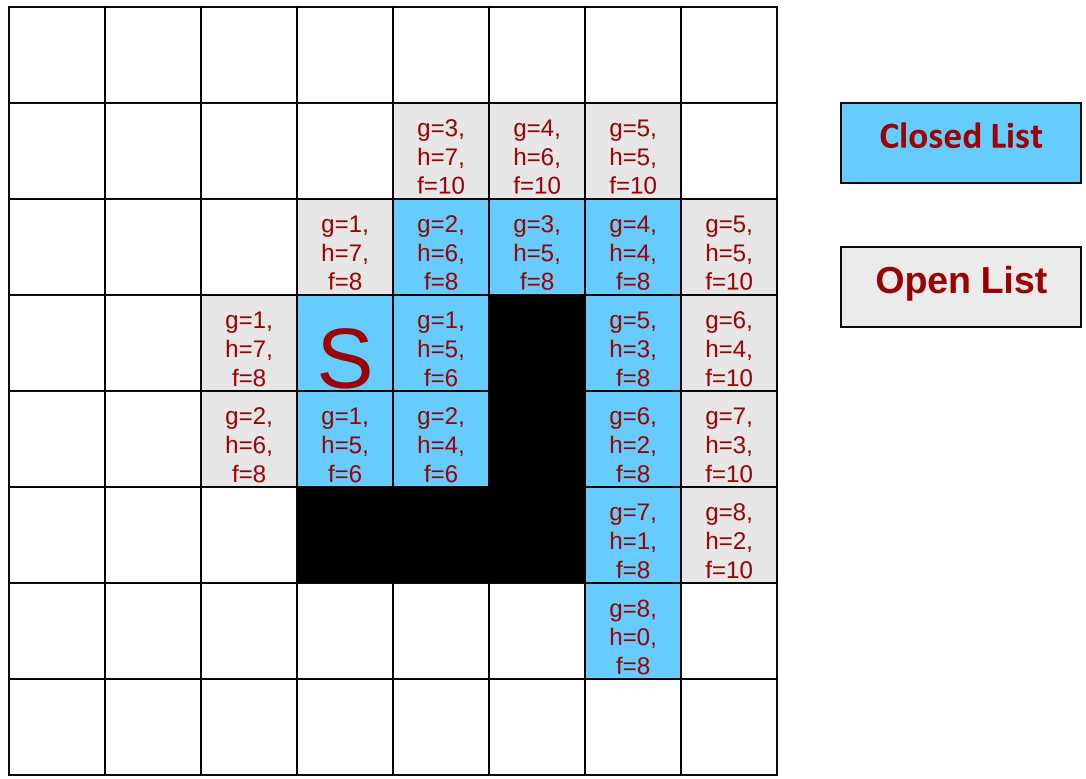
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
1. **s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)**
2. **Add s to closed list**

3a. if s = goal node then
trace path back to start; STOP!

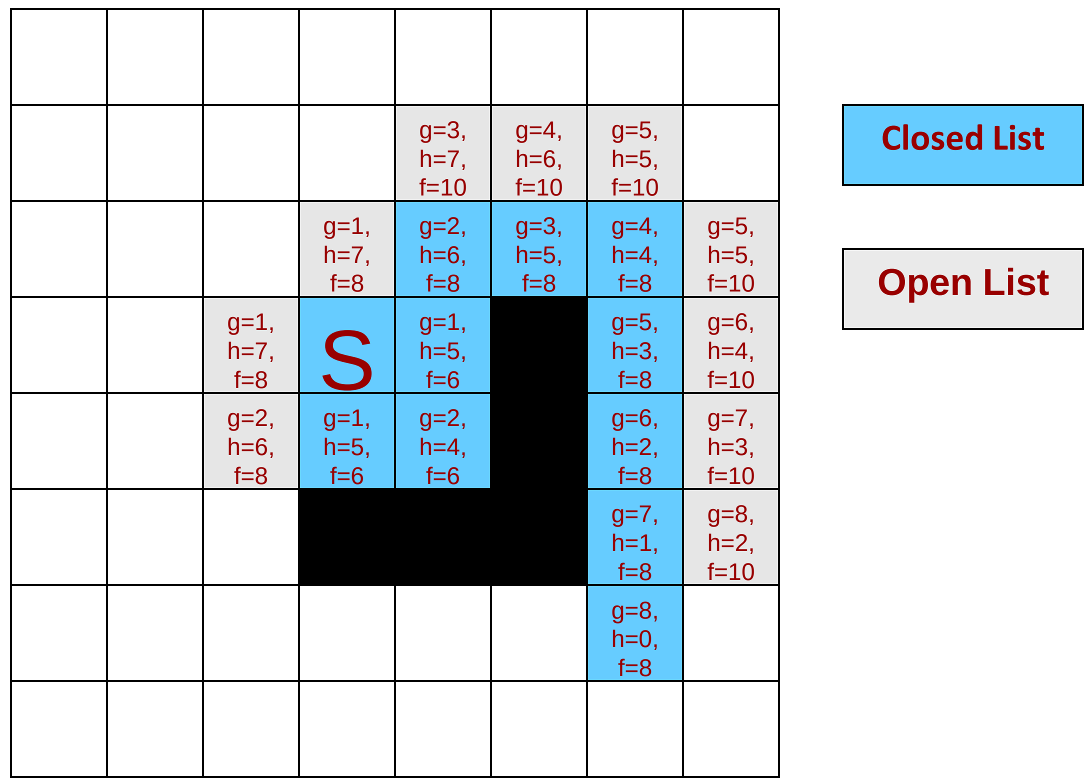
3b. else
Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don't re-explore), or if they are already in the open list, update them if they have a smaller f value



Path-Planning w/ A* Algorithm

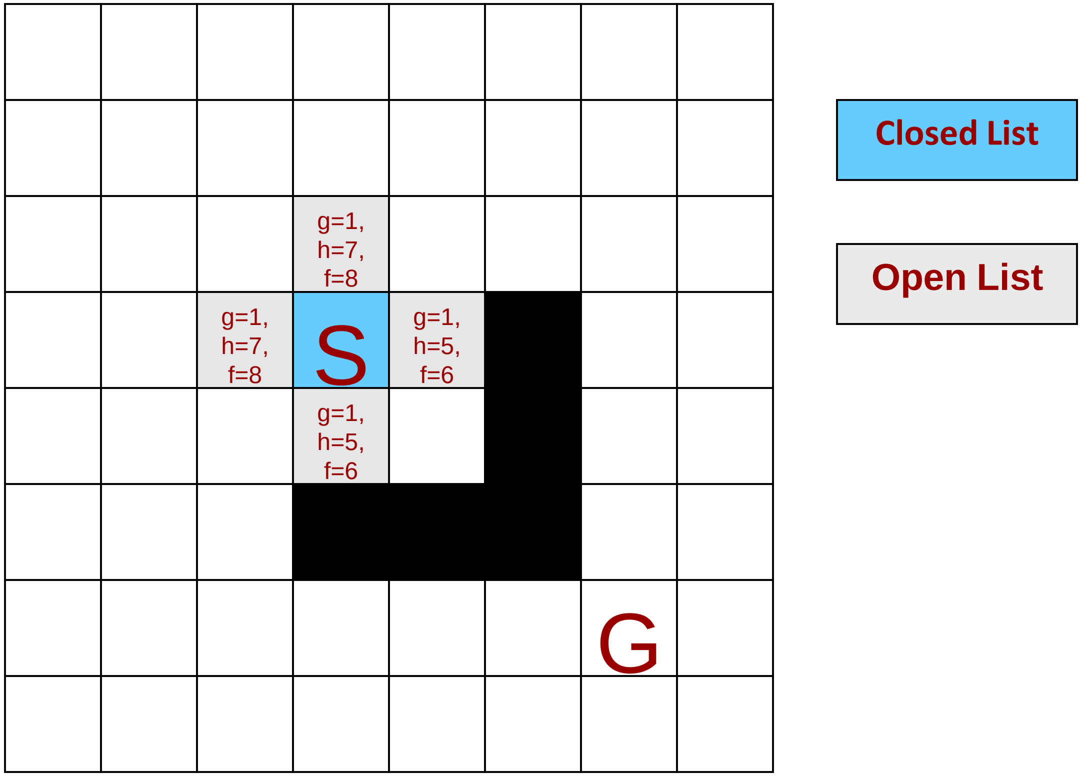
- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
2. Add s to closed list
3a. if s = goal node then
trace path back to start; STOP!
3b. else
Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don't re-explore), or if they are already in the open list, update them if they have a smaller f value



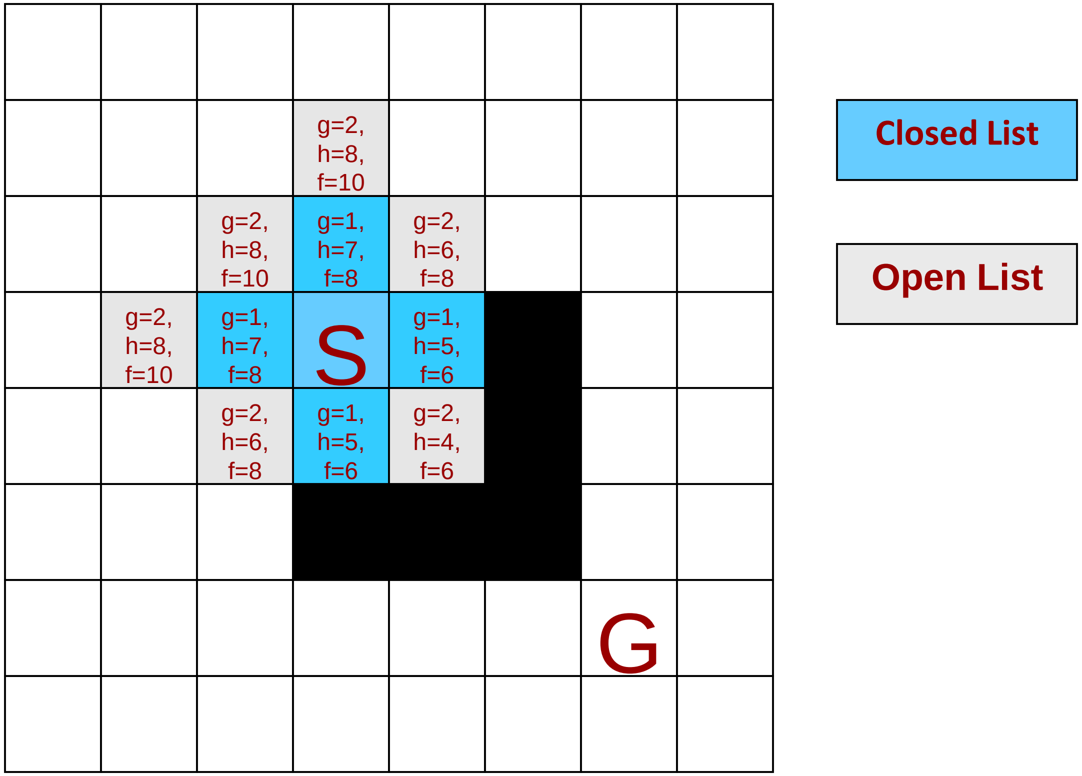
A* and BFS

- BFS explores all nodes at a shorter distance from the start (i.e. g value)



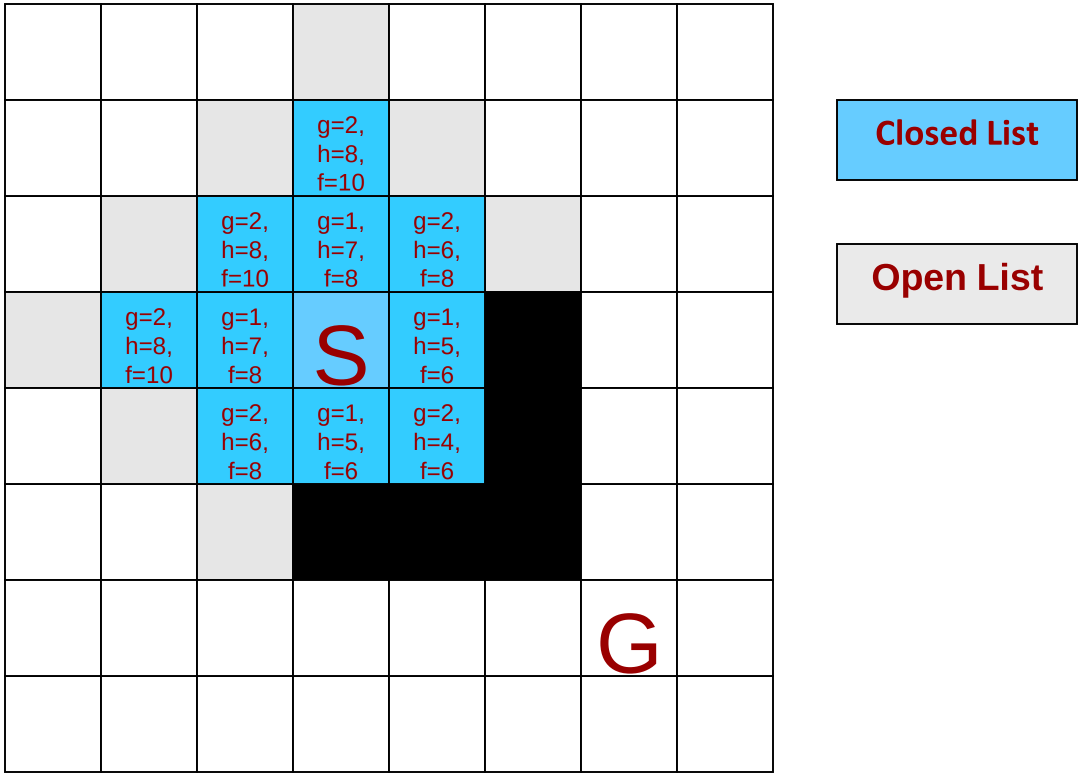
A* and BFS

- BFS explores all nodes at a shorter distance from the start (i.e. g value)



A* and BFS

- BFS is A* using just the g value to choose which item to select and expand



A* Analysis

- What data structure should we use for the open-list?
- What data structure should we use for the closed-list?
- What is the run time?
- Run time is similar to Dijkstra's algorithm…
  - We pull out each node/state once from the open-list so that incurs N*O(remove-cost)
  - We then visit each successor which is like O(E) and perform an insert or decrease operation which is like E*max(O(insert), O(decrease)
  - E = Number of potential successors and this depends on the problem and the possible solution space
  - For the tile puzzle game, how many potential boards are there?

open_list.push(Start State)
while(open_list is not empty)
1. s ← remove min. f-value state from open_list
(if tie in f-values, select one w/ larger g-value)
2. Add s to closed list
3a. if s = goal node then trace path back to start; STOP!
3b. Generate successors/neighbors of s, compute their f values, and add them to open_list if they are not in the closed_list (so we don't re-explore), or if they are already in the open list, update them if they have a smaller f value

Implementation Note

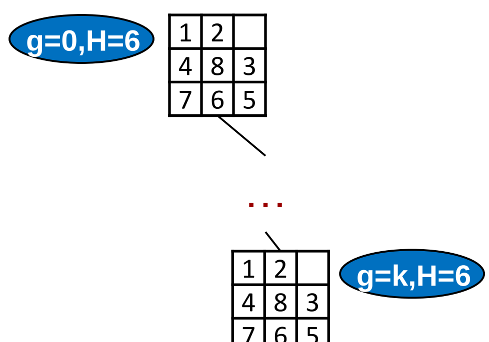
- When the distance to a node/state/successor (i.e. g value) is uniform, we can greedily add a state to the closed-list at the same time as we add it to the open-list

Non-uniform g-values

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
    (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start; STOP!
  3b. Generate successors/neighbors of s, compute their f values, and add them to open_list if they are not in the closed_list (so we don't re-explore), or if they are already in the open list, update them if they have a smaller f value

Uniform g-values

open_list.push(Start State)
Closed_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
    (if tie in f-values, select one w/ larger g-value)
  3a. if s = goal node then trace path back to start; STOP!
  3b. Generate successors/neighbors of s, compute their f values, and add them to open_list and closed_list if they are not in the closed_list



The first occurrence of a board has to be on the shortest path to the solution

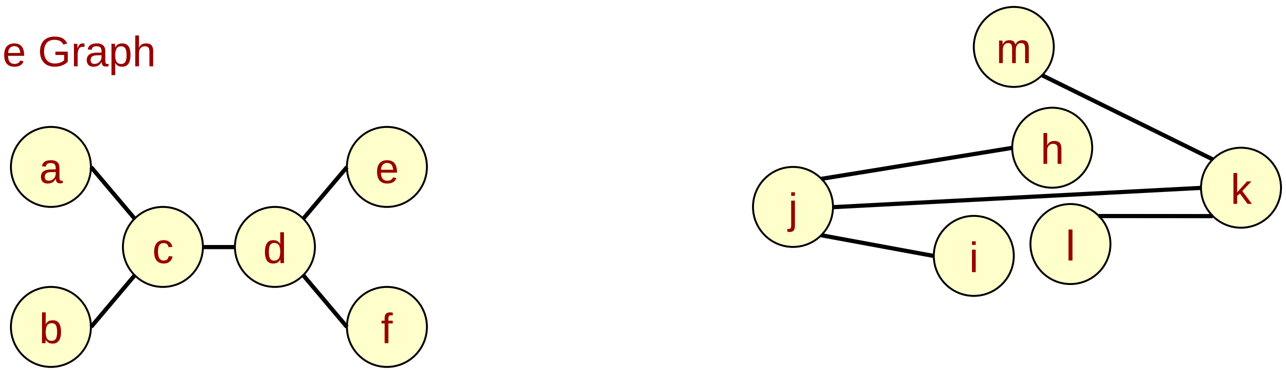
If time allows…

BETWEENESS CENTRALITY

BC Algorithm Overview

- What's the most central vertex(es) in the graph below?
- How do we define "centrality"?
- Betweeness centrality defines "centrality" as the nodes that are between the most other pairs

Sample Graph



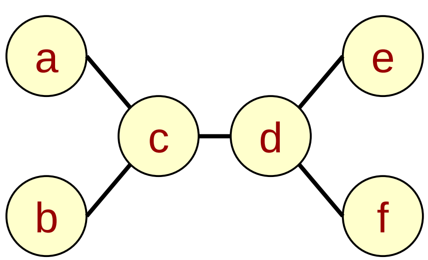
Graph 1

Graph 2

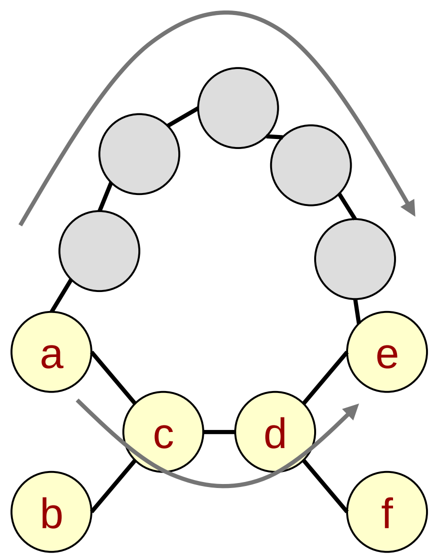
BC Algorithm Overview

- Betweeness centrality (BC) defines "centrality" as the nodes that are between (i.e. on the path between) the most other pairs of vertices
- BC considers betweeness on only "**<u>shortest</u>**" paths!
- To compute centrality score for each vertex we need to find shortest paths between all pairs…
  - Use the Breadth-First Search (BFS) algorithm to do this

Sample Graph



Original 1



Original w/ added path

Are these gray nodes 'between' a and e?

No, a-c-d-e is the shortest path?

BC Algorithm Overview

- Betweeness-Centrality determines "centrality" as the number of shortest paths from all-pairs upon which a vertex lies
- Consider the sample graph below
  - Each external vertex (a, b, e, f) lies is a member of only the shortest paths between itself and each other vertex
  - Vertices c and d lie on greater number of shortest paths and thus will be scored higher

Sample Graph

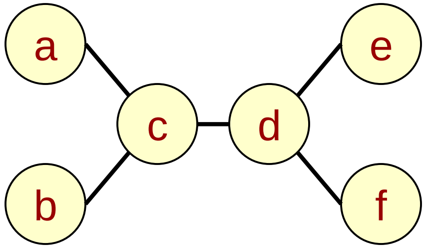
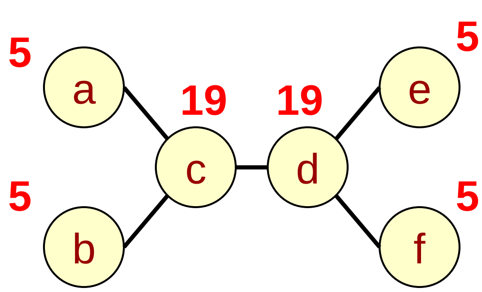


Image each vertex is a ball and each edge is a chain or string. What would this graph look like if you picked it up by vertex c? Vertex a?

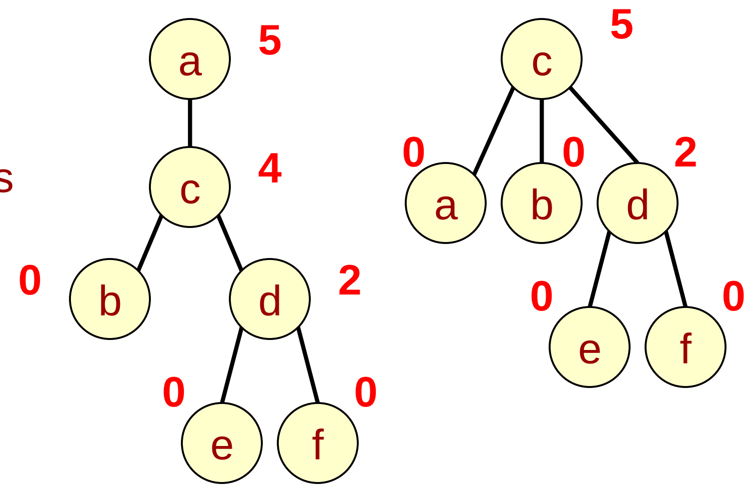
BC Implementation

- Based on Brandes' formulation for unweighted graphs
  - Perform |V| Breadth-first traversals
  - Traversals result in a subgraph consisting of shortest paths from root to all other vertices
  - Messages are then sent back up the subgraph from "leaf" vertices to the root summing the percentage of shortest-paths each vertex is a member of
  - Summing a vertex's score from each traversal yields overall BC result

Sample Graph with final BC scores

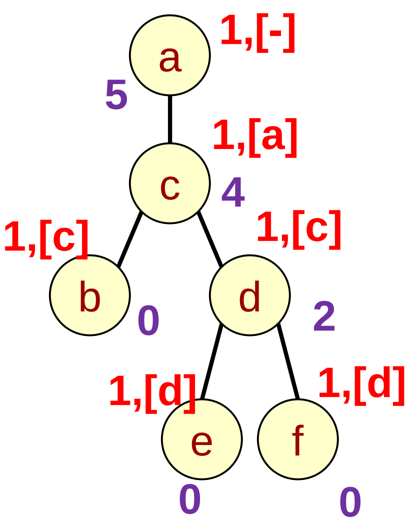


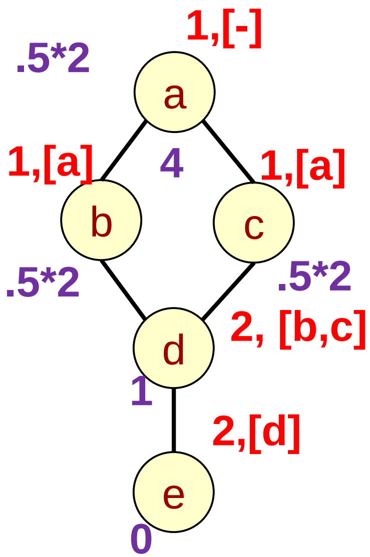
Traversals from selected roots and resulting partial BC scores (in this case, the number of descendants)



BC Implementation

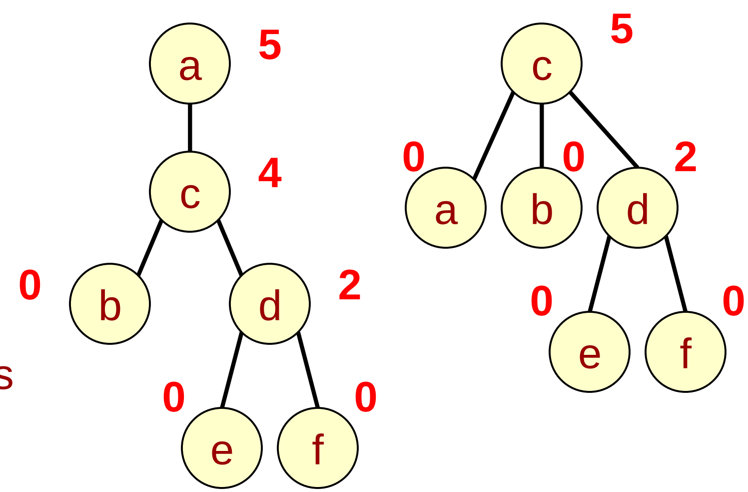
- As you work down, track # of shortest paths running through a vertex and its predecessor(s)
- On your way up, sum the nodes beneath





# of shortest paths thru the vertex, [List of predecessor]

Score on the way back up (if multiple shortest paths, split the score appropriately)



Traversals from selected roots and resulting partial BC scores (in this case, the number of descendants)