CSCI 104
Dijkstra's algorithm and A*

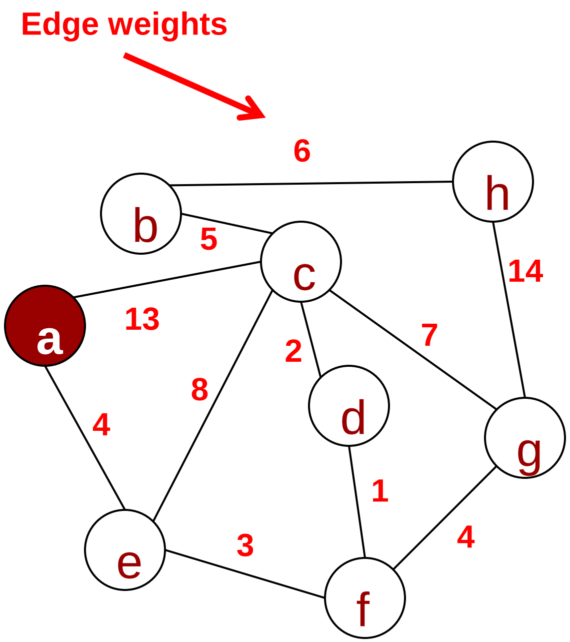
Mark Redekopp
David Kempe
Sandra Batista

Dijkstra's Algorithm

SINGLE-SOURCE SHORTEST PATH (SSSP)

SSSP

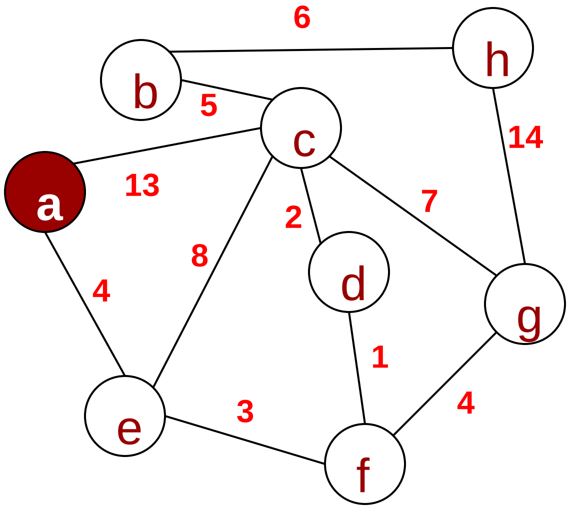
- Assign to each edge a positive weight
  - Could be physical distance, cost of using the link, etc.
- Find the shortest path from a source node, 'a' to all other nodes



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?
- **Distance** is defined to be **sum of weights on path between source and node.**
- How would you compute these 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)

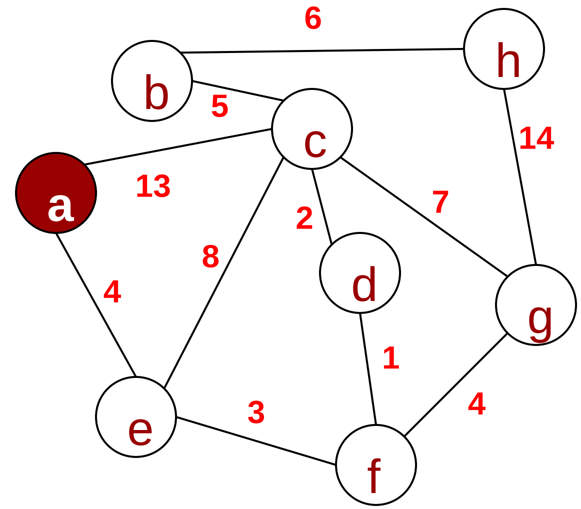
Adjacency Lists

List of Vertices

Vert	Dist
a	0
b	
c	
d	
e	
f	
g	
h	

Dijkstra's Algorithm

- On each iteration, Dijkstra's algorithm selects vertex with minimum distance to source vertex
- BFS uses **queue** to maintain vertices in order discovered
- Dijkstra’s uses a **priority queue** to maintain vertices in shortest distance to source
  - To demonstrate, we’ll use table of all vertices with their current known distances to source

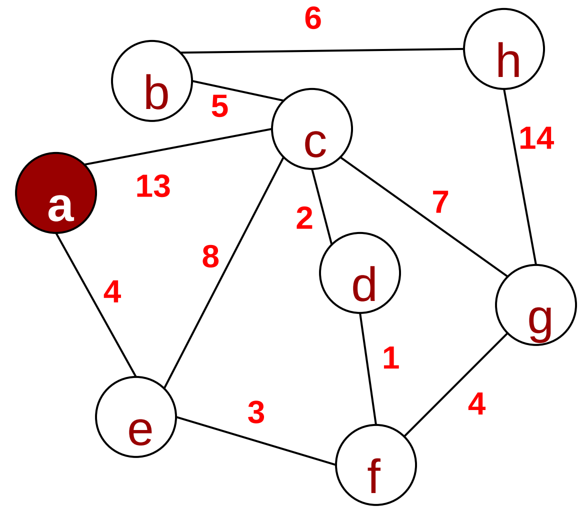


List of Vertices

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)
```

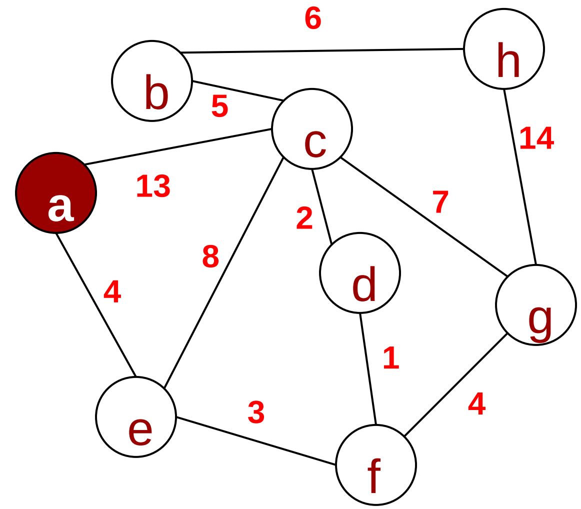


List of Vertices

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)
```



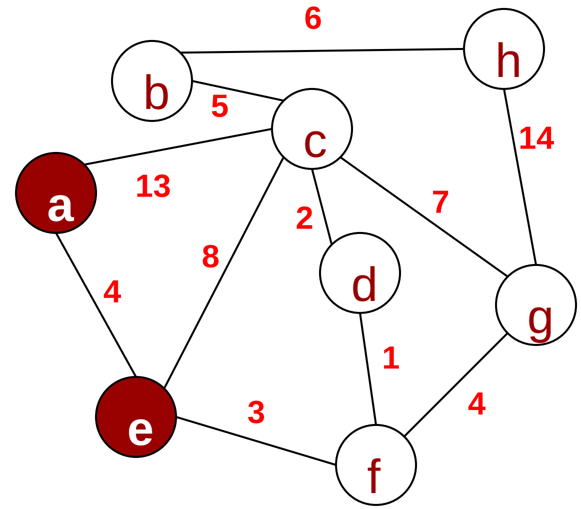
List of Vertices

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)
```



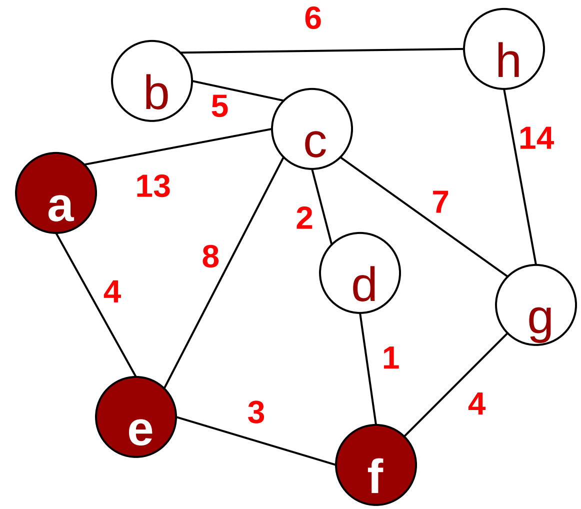
List of Vertices

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)
```



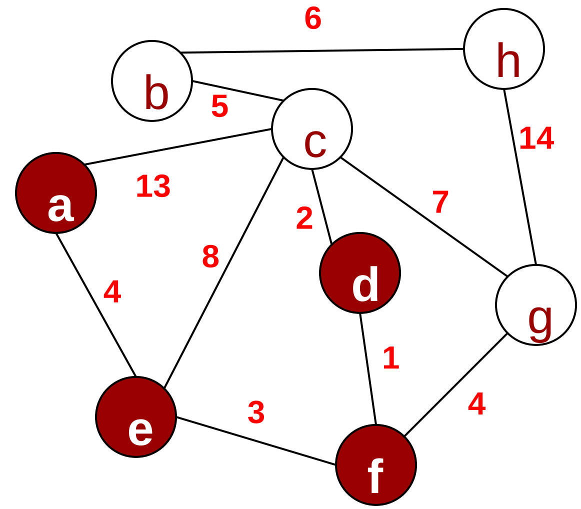
List of Vertices

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)
```



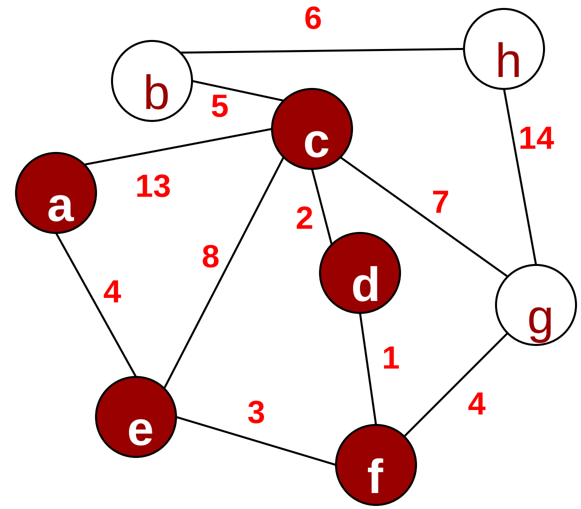
List of Vertices

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)
```



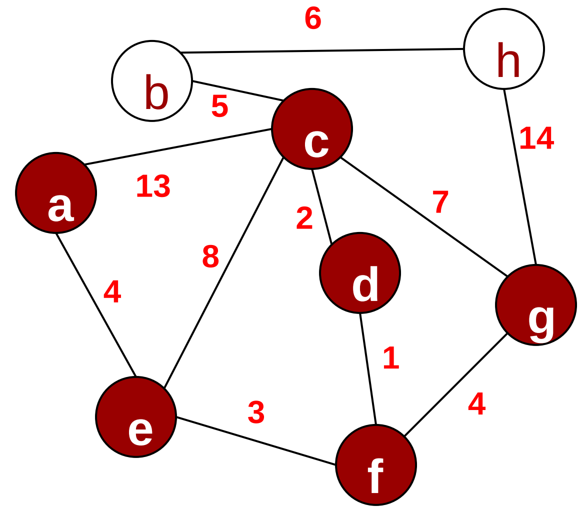
List of Vertices

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)
```



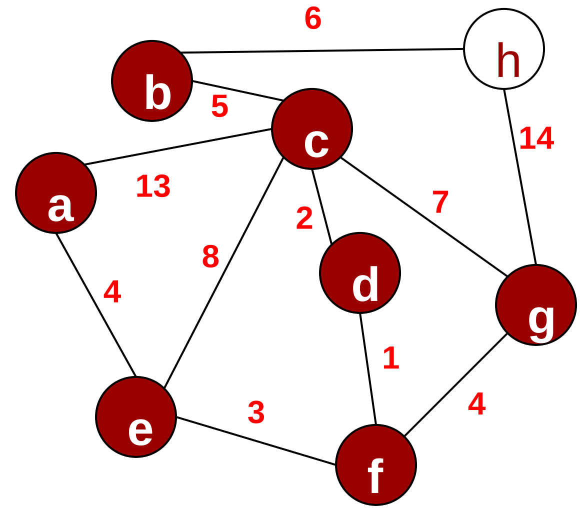
List of Vertices

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)
```



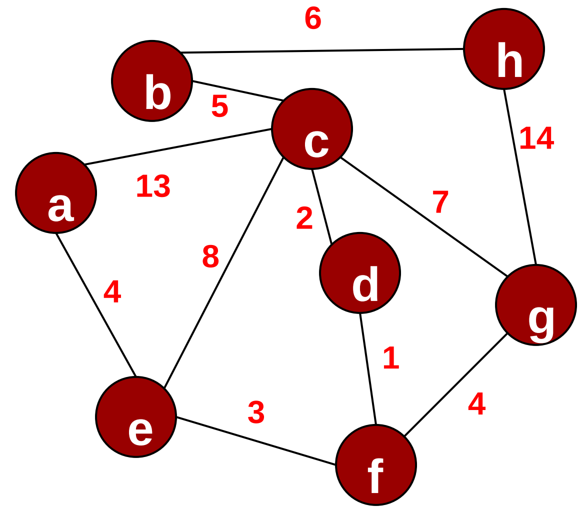
List of Vertices

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

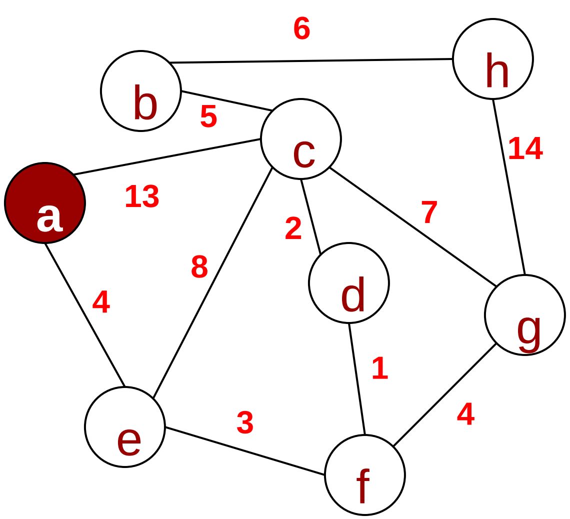
Vert	Dist
a	0
b	15
c	10
d	8
e	4
f	7
g	11
h	21

v=h

Analysis

- What is the loop invariant?
  - The vertex v removed from PQ is guaranteed to be the vertex with shortest path to source of all vertices in PQ and its distance is set to its shortest distance to the source.
  - All vertices whose distances and predecessors are set have shortest known distance thus far to source.

- Proof sketch by induction
  - First node from PQ is source itself with distance 0 to itself.
  - Decrease the distance to its neighbors; its neighbor with the shortest distance will be at front of PQ
  - No shorter path from source to vertex at front of PQ; any other path would use some edge from the start having greater distance



A* Search Algorithm

ALGORITHM HIGHLIGHT

Search Methods

- Many systems require searching for goal states
  - Path Planning
    - 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	8
7	6	4
5	3	2

Original: No order

1	2	3
4	5	6
7	8	

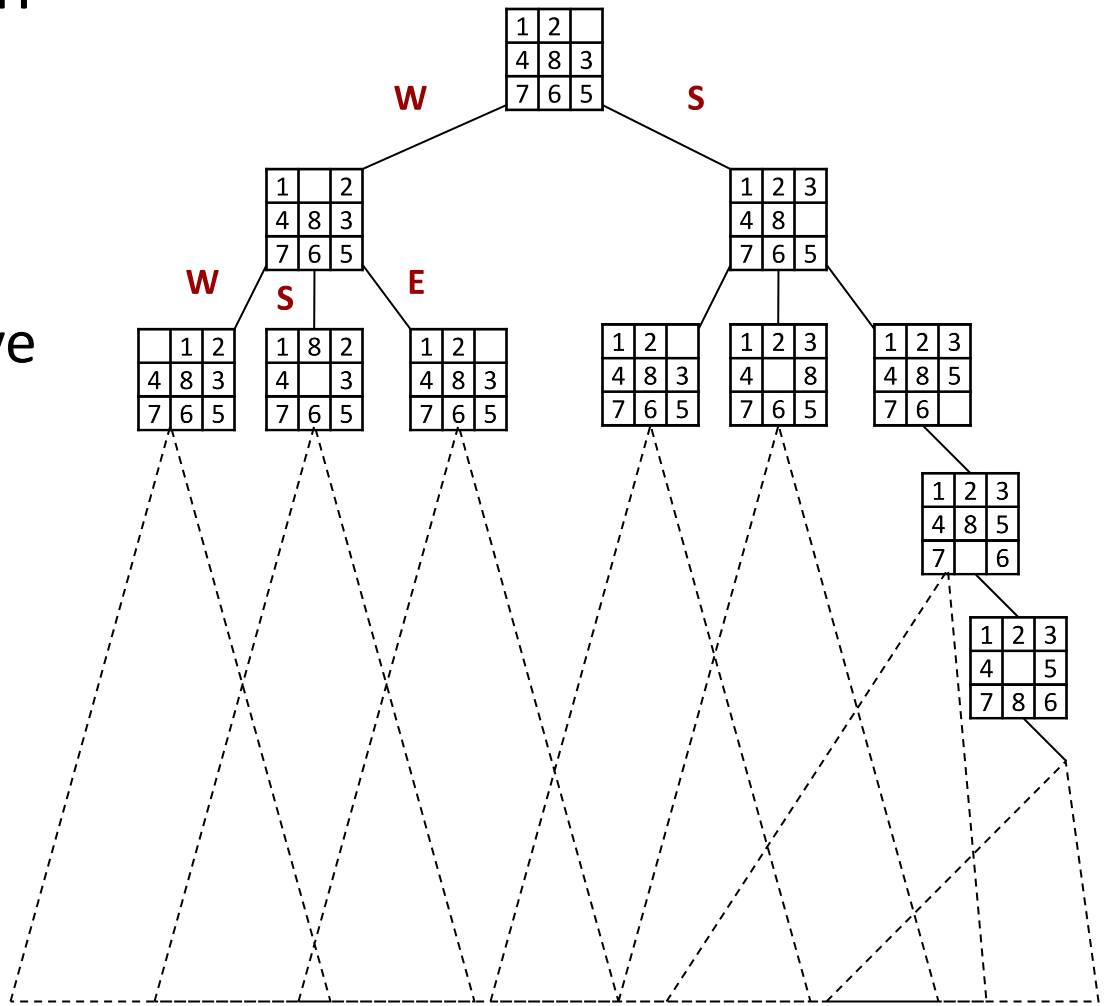
Goal State

Search Methods

- **Brute-Force Search**: Search all possibilities until you find it! ☹
- **Heuristic Search**: A heuristic is a “rule of thumb”.
  - Heuristics are not perfect; they are quick computations to give an approximate measure.

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)
- Heuristics must be easy to compute from current state
- Heuristics for 8-tile puzzle
  - # of tiles out of place
  - Total x-, y- distance of each tile from its correct location (Manhattan distance)

1	8	3
4	5	6
2	7	

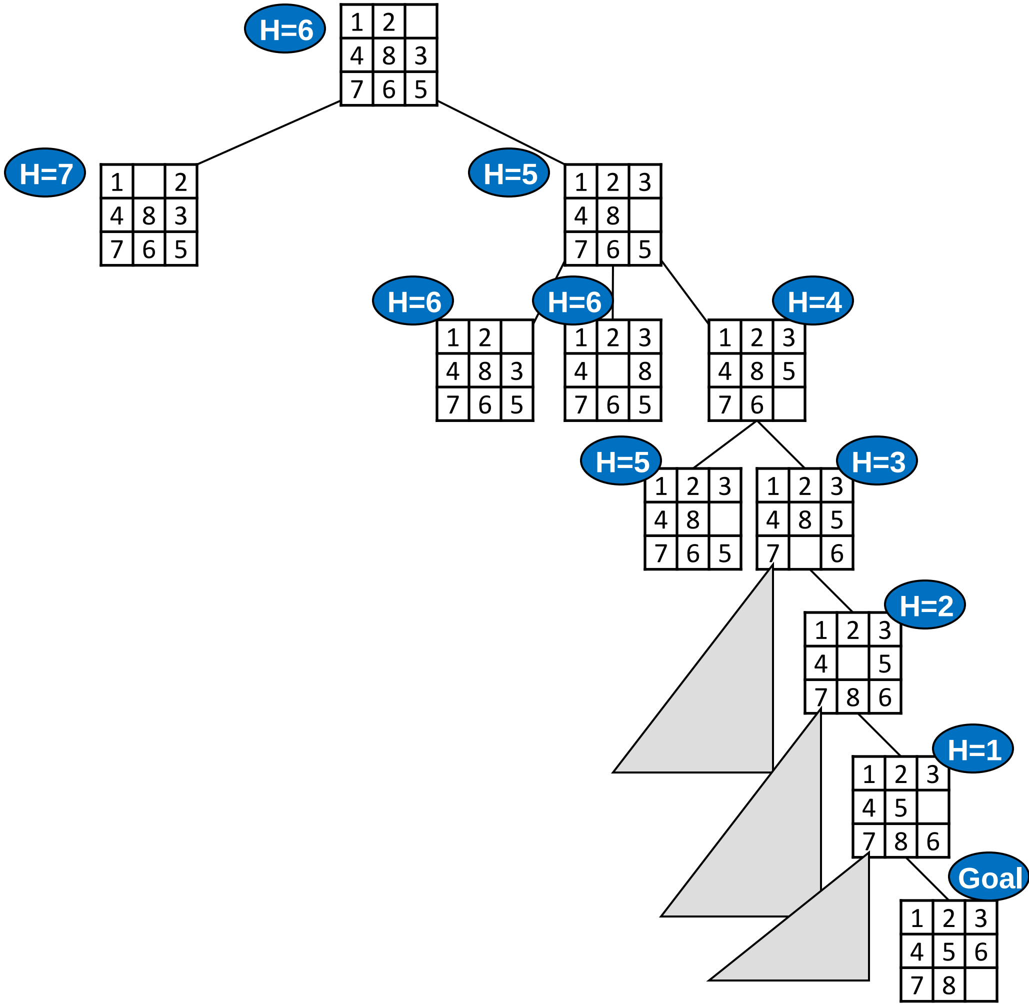
# of Tiles out of Place = 3

1	8	3
4	5	6
2	7	

Total x-/y- distance = 6

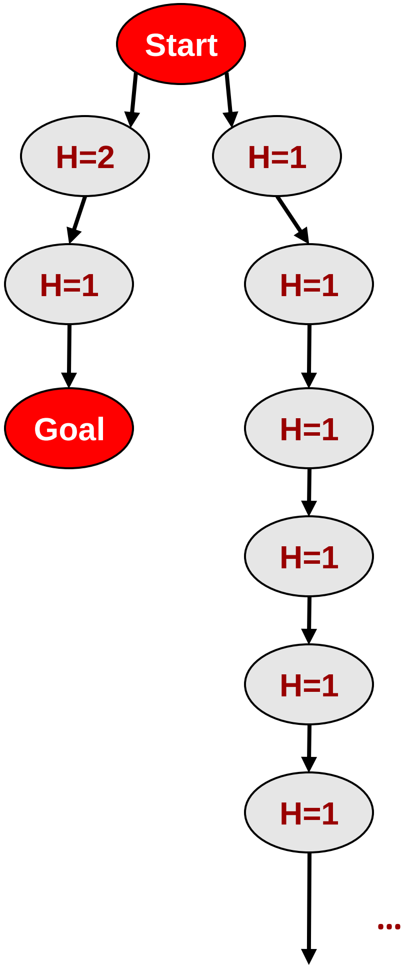
Heuristic Search

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



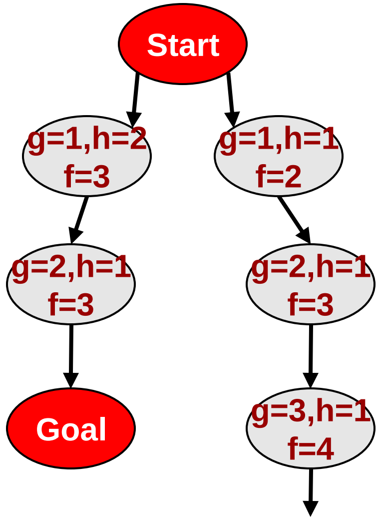
Caution About Heuristics

- Heuristics may 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

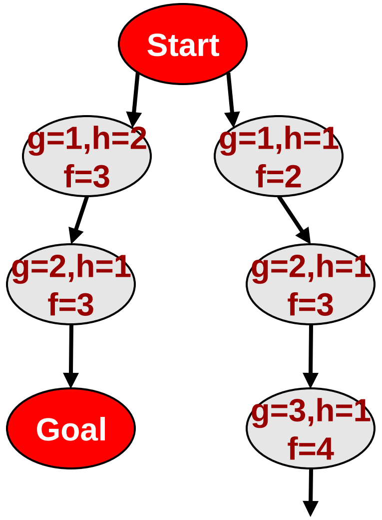
- Use a new metric to decide which state to explore/expand
- Define
  - h = heuristic score (Manhattan distance)
  - g = number of moves from start to current state
  - f = g + h
- As we explore states and their successors, assign each state its f-score and always explore the state with lowest f-score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



USC Viterbi School of Engineering

A-Star Algorithm

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

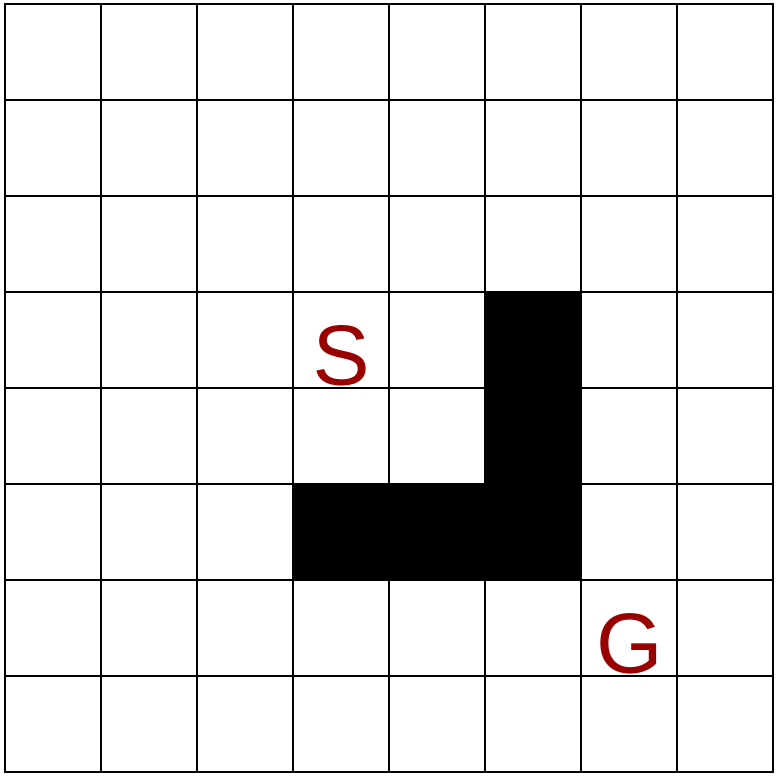



```
 open_list.push(Start State)
while(open_list is not empty)
   1. s ← remove min. f-value state from open_list
 (if tie in f-values, select one w/ larger g-value)
   2. Add s to closed list
   3a. if s = goal node then trace path back to start; STOP!
   3b. For each neighbor, v, of s, compute  f-value
        if v on closed_list, skip v.
        if v not on open_list, add v to open list
        if v on open_list, update f-value if current value is smaller
```

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. For each neighbor, v, of s, compute f-value
        if v on closed_list, skip v.
        if v not on open_list, add v to open list
        if v on open_list, update f-value if current value is smaller



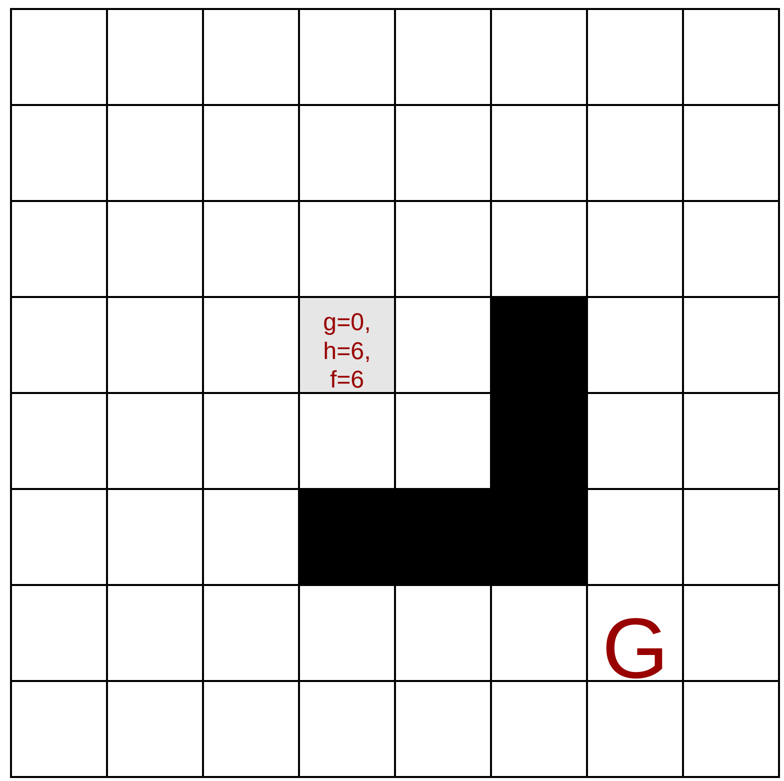
Closed List

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. For each neighbor, v, of s, compute f-value
    if v on closed_list, skip v.
    if v not on open_list, add v to open list
    if v on open_list, update f-value if current value is smaller



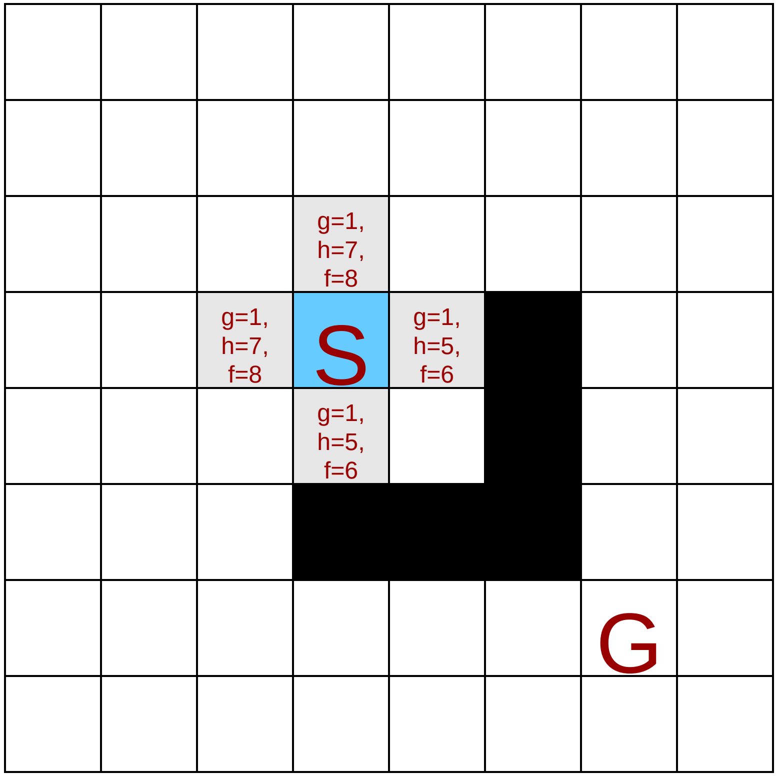
Closed List

Open List

Path-Planning w/ A* Algorithm

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

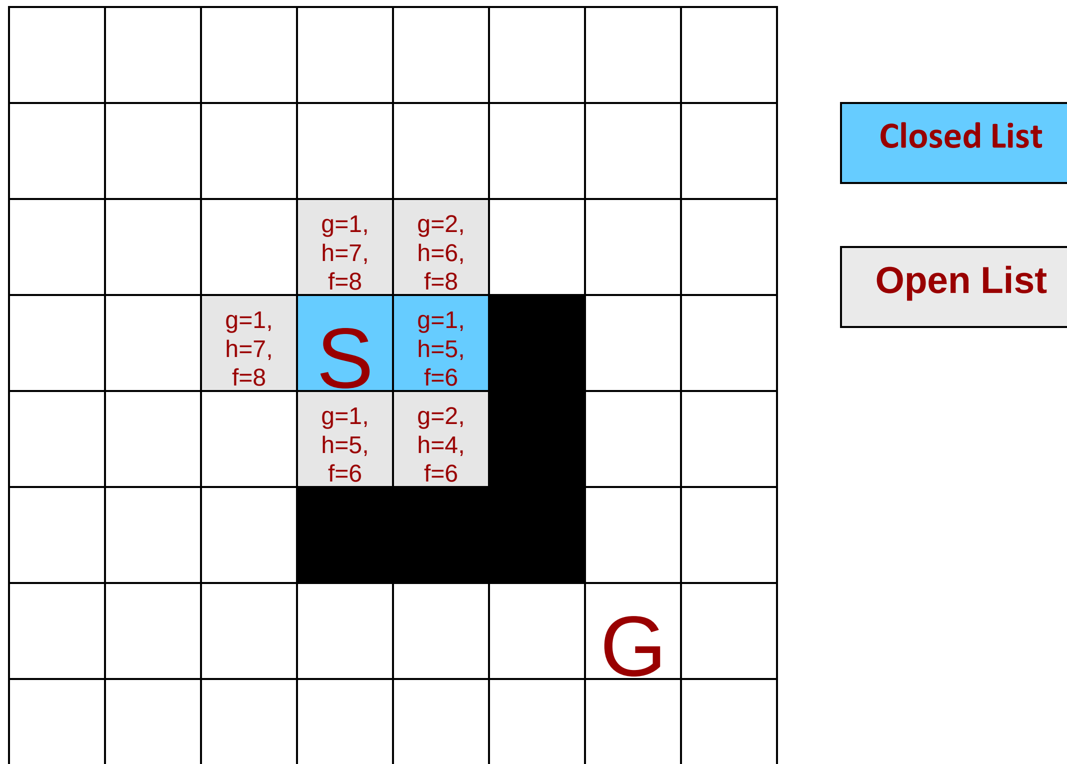
open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
 (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start; STOP!
  3b. For each neighbor, v, of s, compute f-value
    if v on closed_list, skip v.
    if v not on open_list, add v to open list
    if v on open_list, update f-value if current value is smaller



Path-Planning w/ A* Algorithm

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

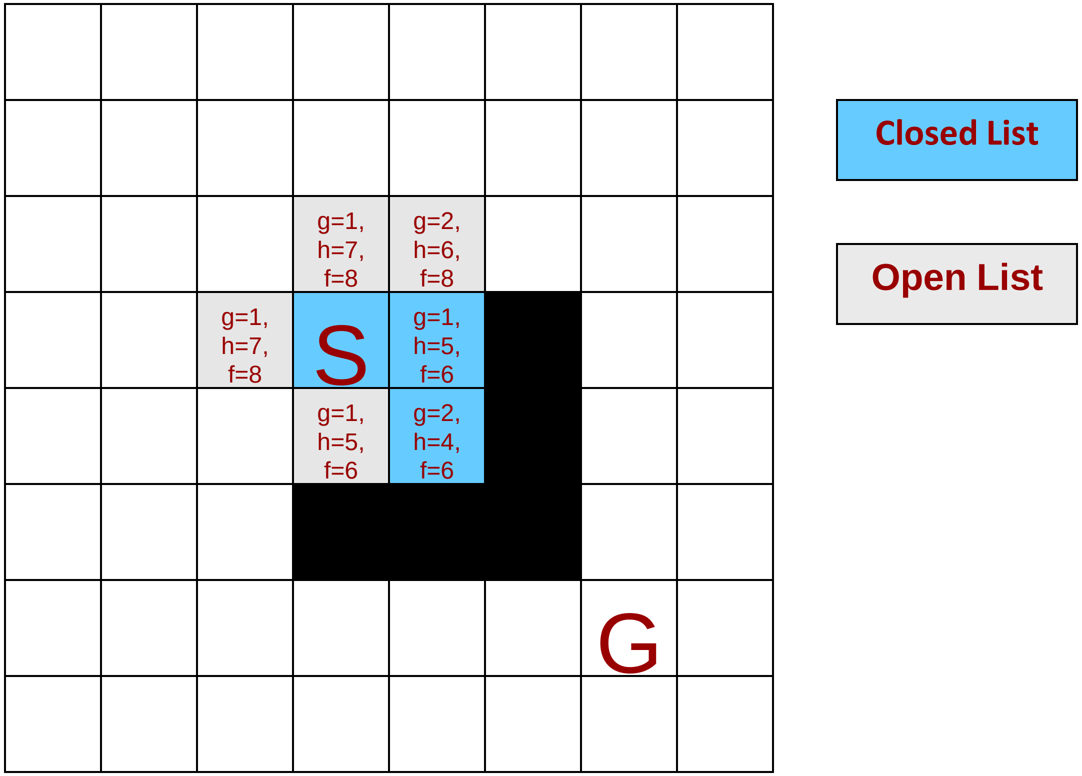
open_list.push(Start State)
while(open_list is not empty)
    1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
    2. Add s to closed list
    3a. if s = goal node then trace path back to start; STOP!
    3b. For each neighbor, v, of s, compute f-value
        if v on closed_list, skip v.
        if v not on open_list, add v to open list
        if v on open_list, update f-value if current value is smaller



Path-Planning w/ A* Algorithm

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

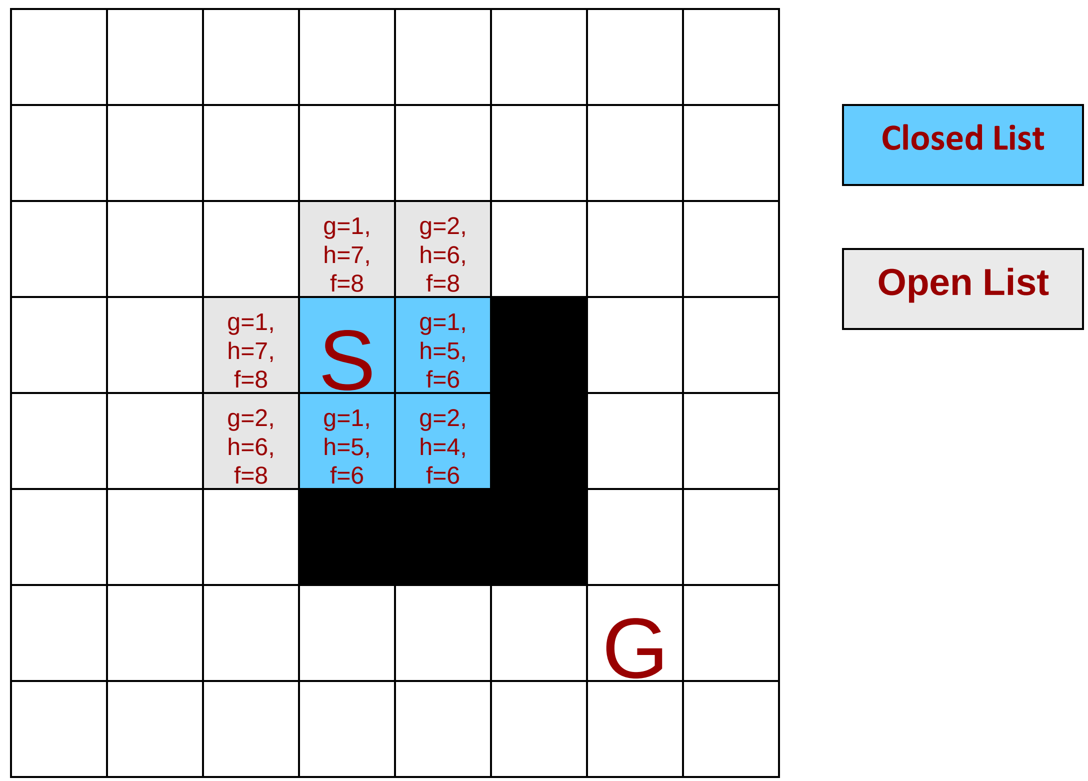
open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start; STOP!
  3b. For each neighbor, v, of s, compute f-value
    if v on closed_list, skip v.
    if v not on open_list, add v to open list
    if v on open_list, update f-value if current value is smaller



Path-Planning w/ A* Algorithm

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

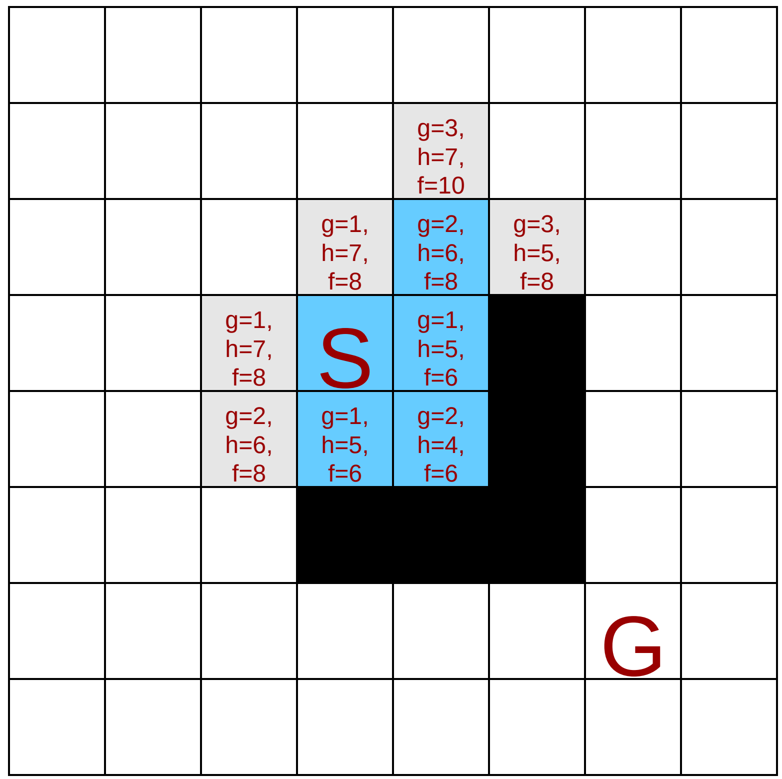
open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
 (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start; STOP!
  3b. For each neighbor, v, of s, compute f-value
    if v on closed_list, skip v.
    if v not on open_list, add v to open list
    if v on open_list, update f-value if current value is smaller



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. For each neighbor, v, of s, compute f-value
      if v on closed_list, skip v.
      if v not on open_list, add v to open list
      if v on open_list, update f-value if current value is smaller



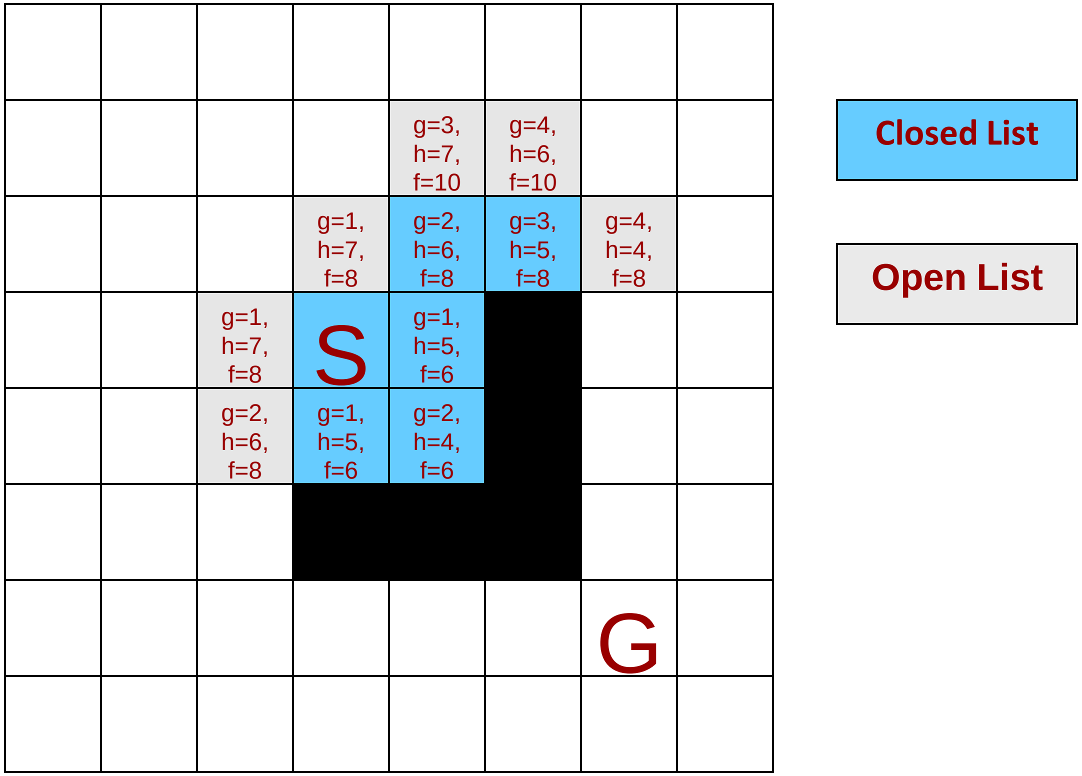
Closed List

Open List

Path-Planning w/ A* Algorithm

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

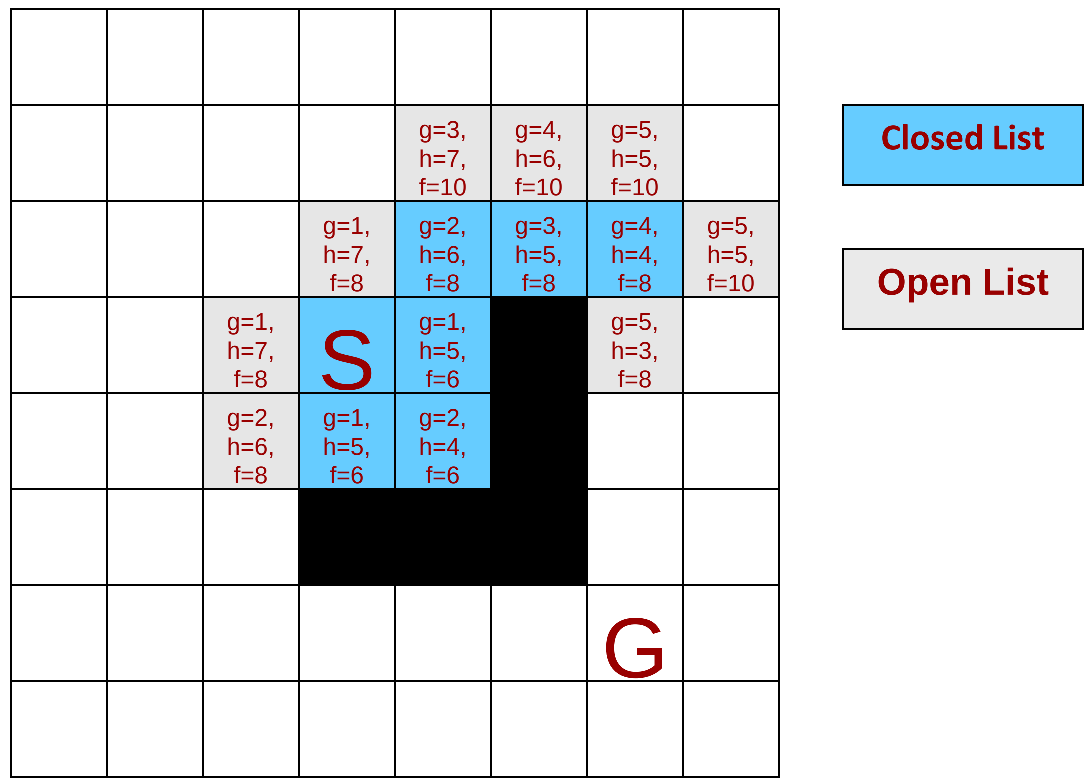
open_list.push(Start State)
while(open_list is not empty)
    1. s ← remove min. f-value state from open_list
 (if tie in f-values, select one w/ larger g-value)
    2. Add s to closed list
    3a. if s = goal node then trace path back to start; STOP!
    3b. For each neighbor, v, of s, compute f-value
        if v on closed_list, skip v.
        if v not on open_list, add v to open list
        if v on open_list, update f-value if current value is smaller



Path-Planning w/ A* Algorithm

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

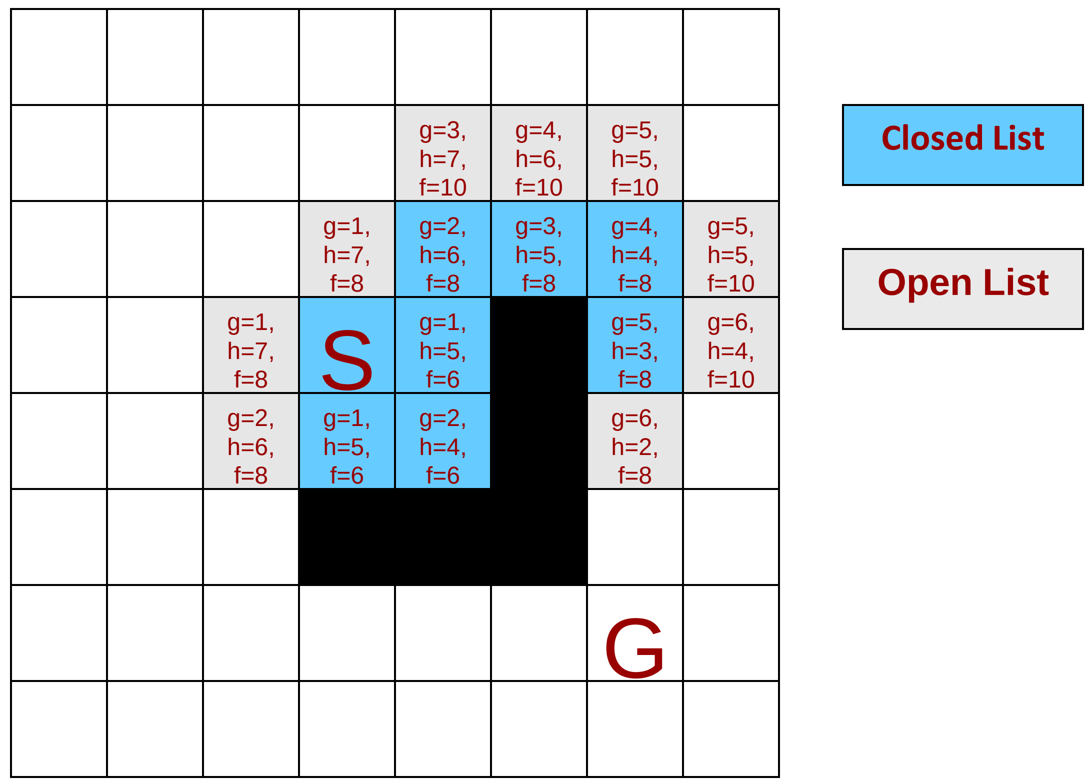
open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
 (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start; STOP!
  3b. For each neighbor, v, of s, compute f-value
    if v on closed_list, skip v.
    if v not on open_list, add v to open list
    if v on open_list, update f-value if current value is smaller



Path-Planning w/ A* Algorithm

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

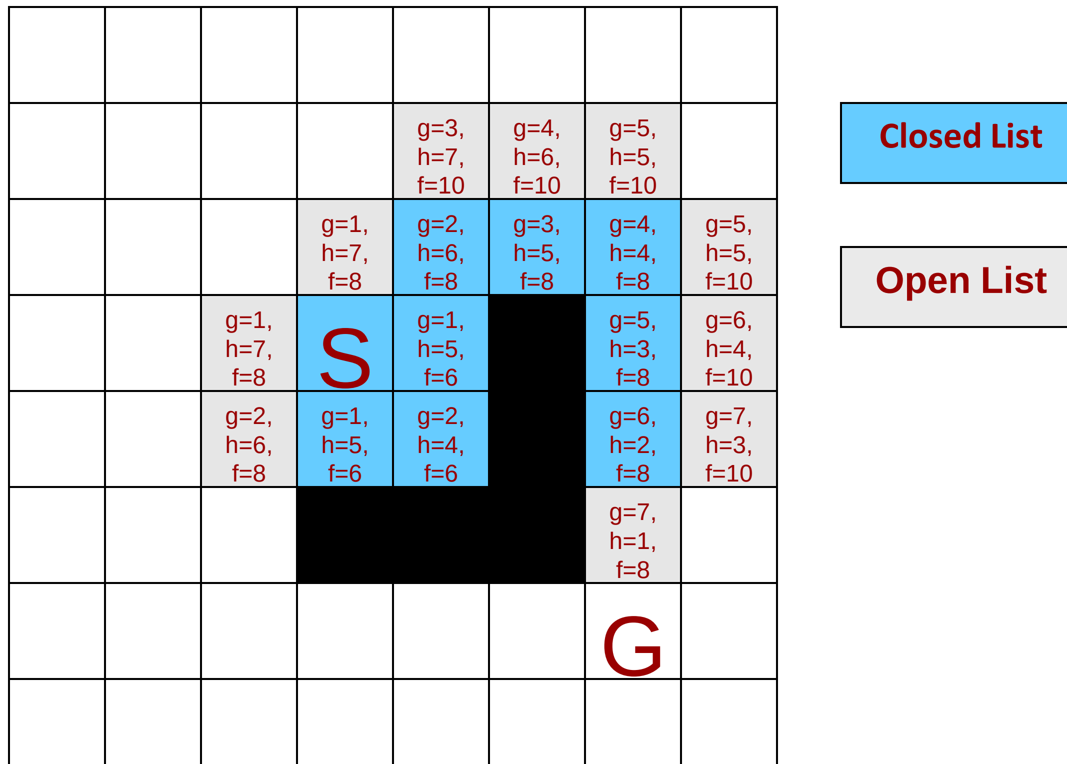
open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
  (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start; STOP!
  3b. For each neighbor, v, of s, compute f-value
    if v on closed_list, skip v.
    if v not on open_list, add v to open list
    if v on open_list, update f-value if current value is smaller



Path-Planning w/ A* Algorithm

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

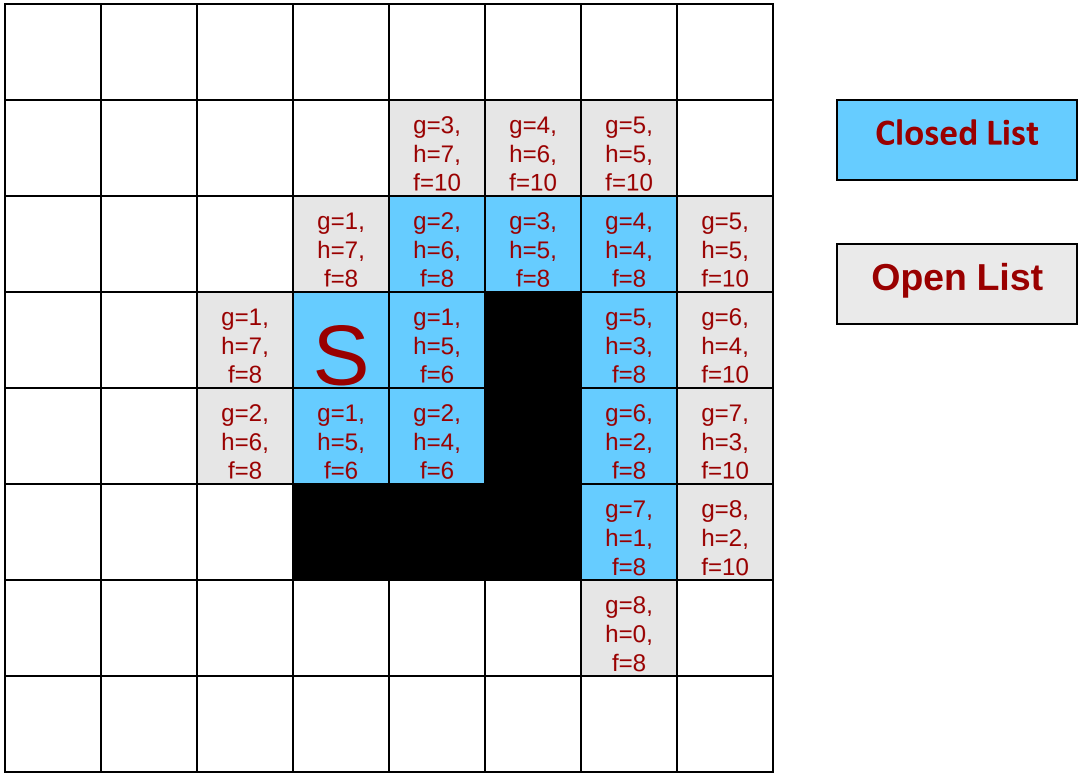
open_list.push(Start State)
while(open_list is not empty)
    1. s ← remove min. f-value state from open_list
  (if tie in f-values, select one w/ larger g-value)
    2. Add s to closed list
    3a. if s = goal node then trace path back to start; STOP!
    3b. For each neighbor, v, of s, compute f-value
        if v on closed_list, skip v.
        if v not on open_list, add v to open list
        if v on open_list, update f-value if current value is smaller



Path-Planning w/ A* Algorithm

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

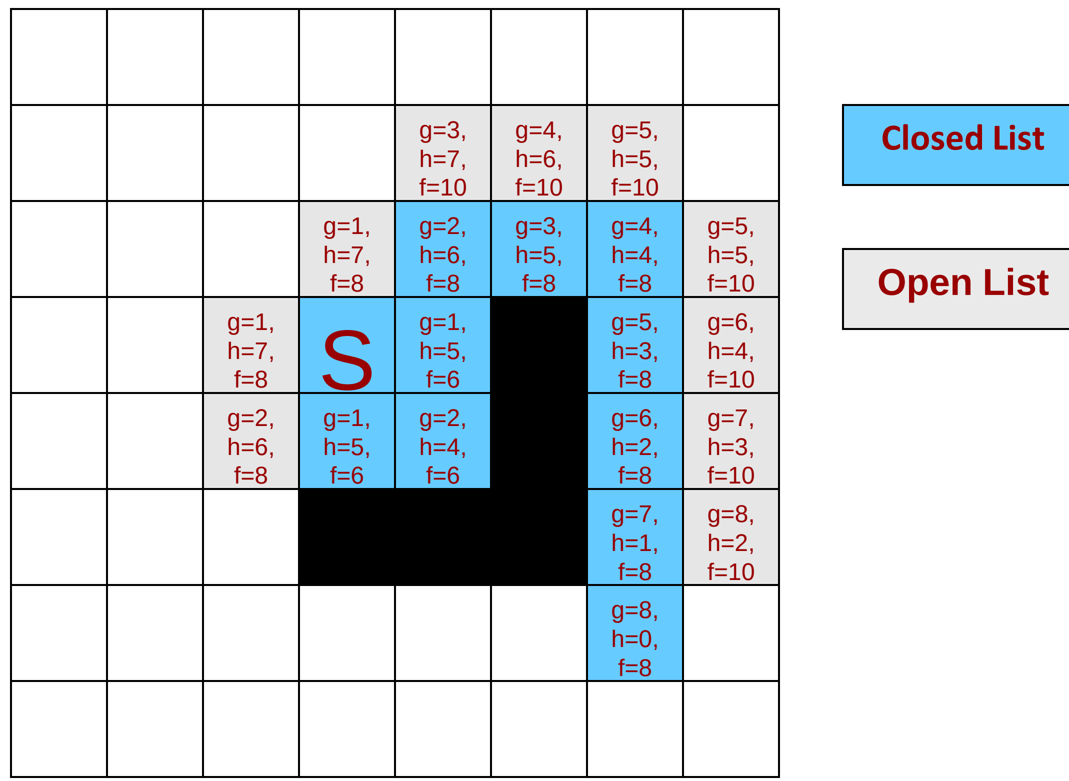
```
open_list.push(Start State)
while(open_list is not empty)
   1. s ← remove min. f-value state from open_list
 (if tie in f-values, select one w/ larger g-value)
   2. Add s to closed list
   3a. if s = goal node then trace path back to start;
STOP!
   3b. For each neighbor, v, of s, compute  f-value
        if v on closed_list, skip v.
        if v not on open_list, add v to open list
        if v on open_list, update f-value if current
value is smaller
```



Path-Planning w/ A* Algorithm

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

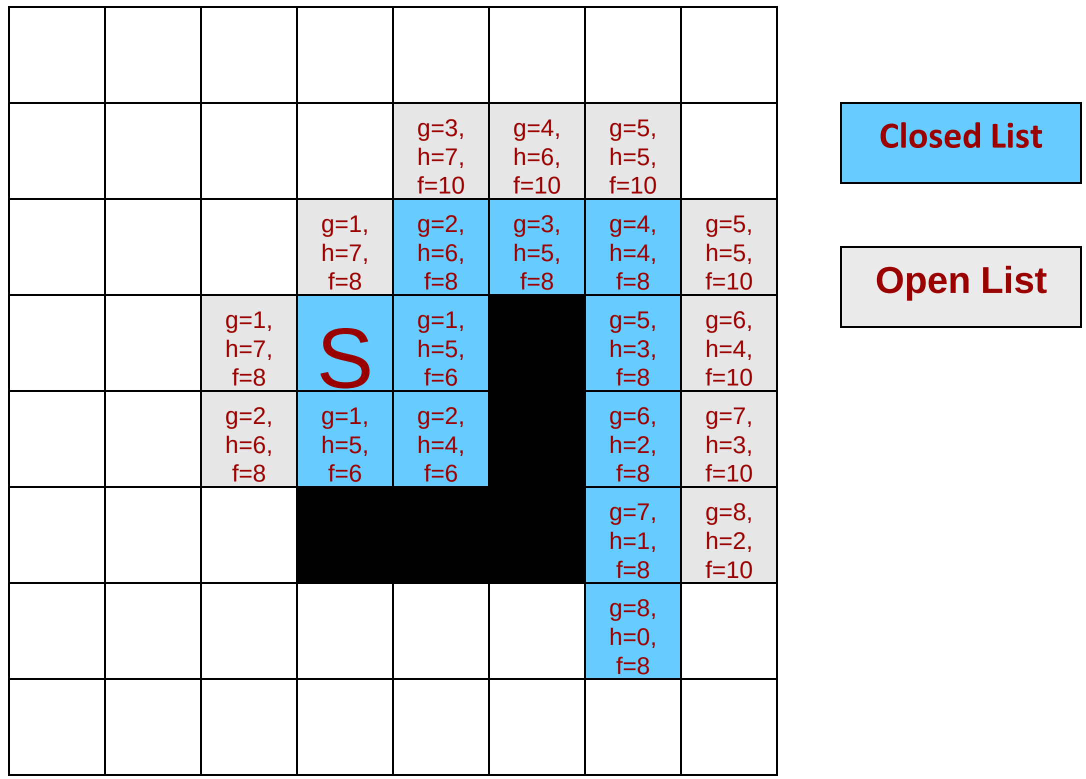
open_list.push(Start State)
while(open_list is not empty)
   1. s ← remove min. f-value state from open_list
 (if tie in f-values, select one w/ larger g-value)
   2. Add s to closed list
   3a. if s = goal node then trace path back to start; STOP!
   3b. For each neighbor, v, of s, compute f-value
      if v on closed_list, skip v.
      if v not on open_list, add v to open list
      if v on open_list, update f-value if current value is smaller



Path-Planning w/ A* Algorithm

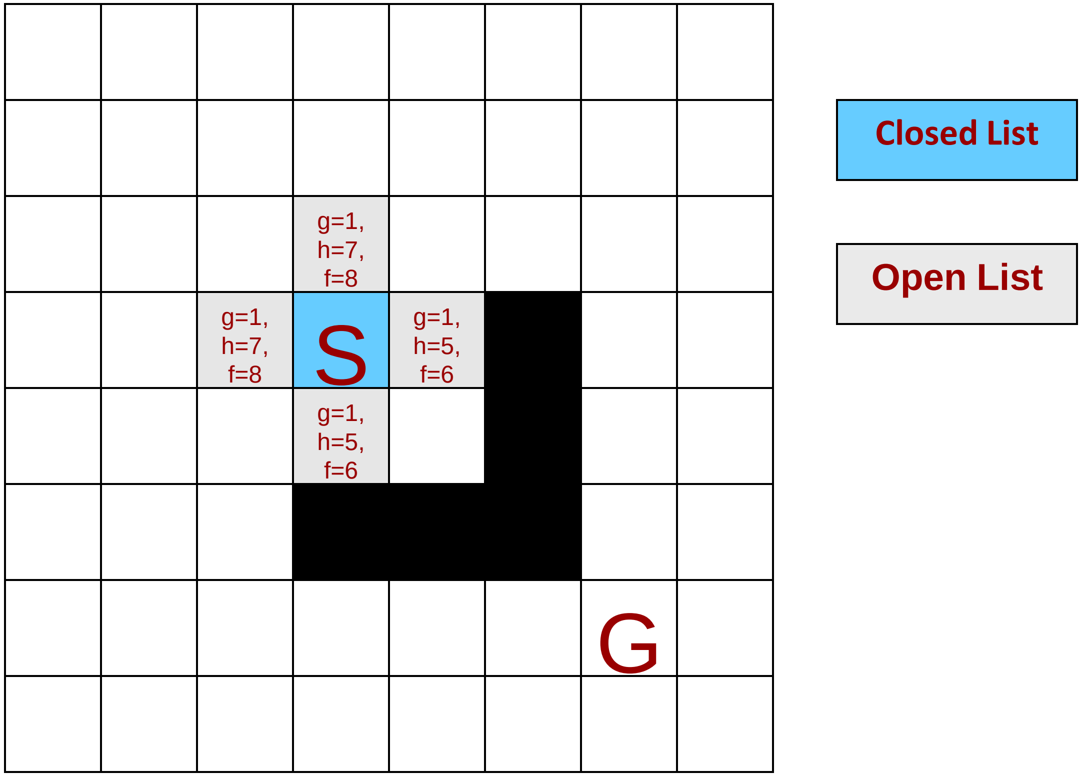
- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

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. For each neighbor, v, of s, compute f-value
        if v on closed_list, skip v.
        if v not on open_list, add v to open list
        if v on open_list, update f-value if current value is smaller



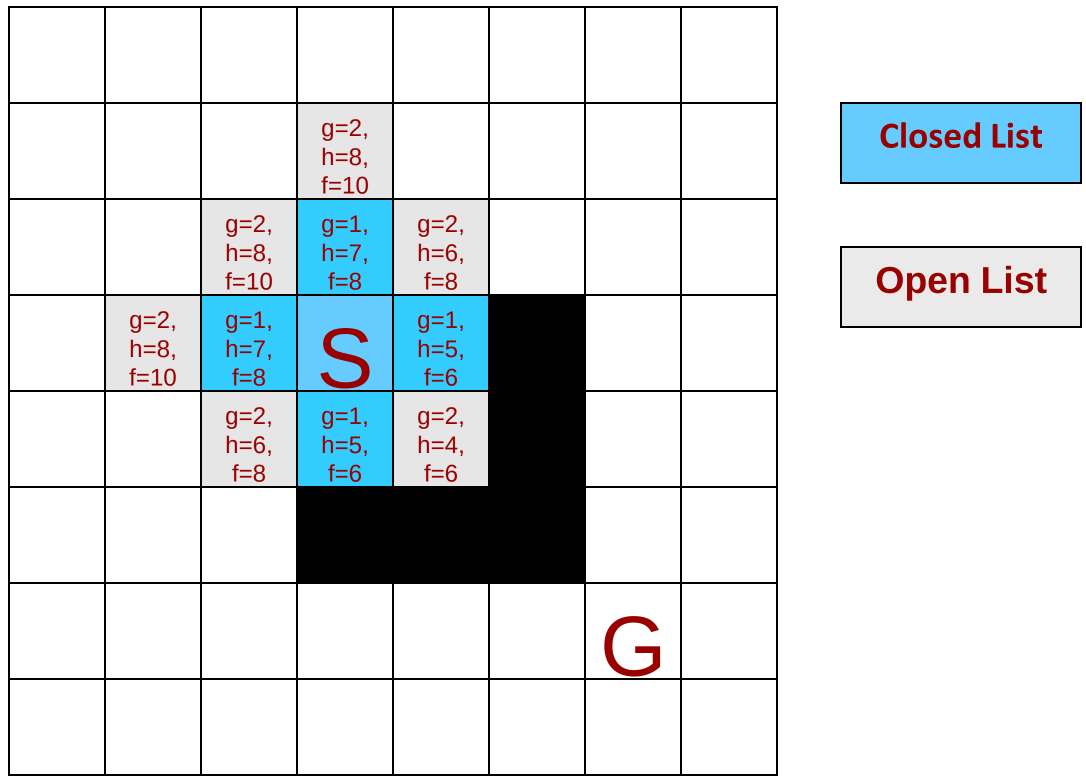
A* and BFS

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



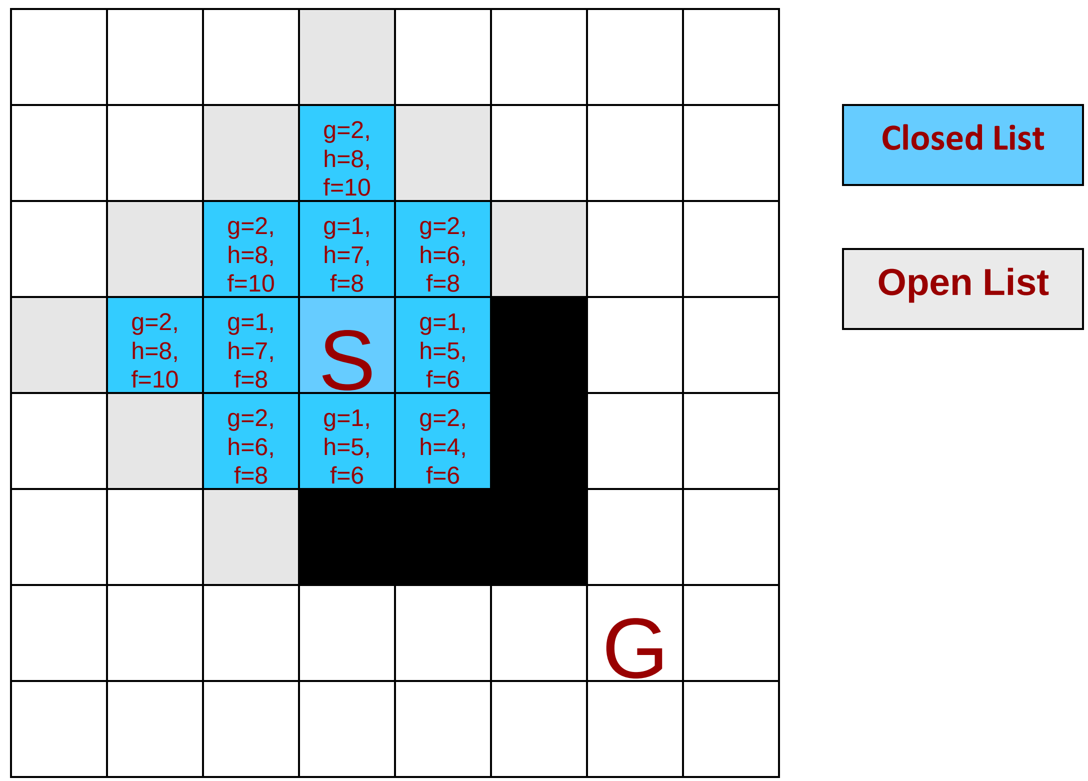
A* and BFS

- 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?
- A* is essentially modification of Dijkstra's with heuristic used for priorities in PQ
- Run time is similar to Dijkstra's algorithm…
  - Each node added/removed once from the open-list so that incurs N*O(remove-cost)
  - Visiting each neighbor requires O(E) operations and performing an insert or decrease operation is 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? (This is why there was a sad face next to the brute force approach…)