Heuristic Search

- Uniform Cost Method
- Best First Search
- A*

Heuristic search

- In Hill Climbing Algorithm unexpanded node are never reconsidered , even if the current path appears to be worst
- This is because the algorithm maintains only the best current path, remaining nodes are abandoned
- Search in state space based on three parameter
  - Cost of moving from initial to current state ----g
  - Estimated cost of moving from current to goal state-------h’
  - Total cost f’= g+h’
- Next set of algorithm uses these values
  - Uniform cost method------g
  - Best first search------------h’
  - A* ------------------------------f’

Uniform Cost Method

- Two List are maintained
  - OPEN LIST
    - Nodes that have been generated and have had the cost function applied to them but which have not yet been examined.
  - CLOSED LIST
    - Nodes that have already been examined. We need to keep these nodes in memory if we want to search a graph rather than a tree
- State space in form of Graph or tree where
  - Node represents a state in the state space
  - Arc represent change in state when an operator is applied
- Cost of moving from initial to currents state (g) is calculated for each state
- Link to parent node is also maintained to trace back the path

Algorithm: UCM

Given : s-starting state, n-any node, g- goal state, o-set of operators, c()-cost function

1. Start with OPEN containing just the initial state

   initialization *open = {s} closed={ } c{s}=0*

2. Until a goal is found or there are no nodes left on OPEN do

   { **fail** if open ={ }; **terminate** if n € G then terminate with success }

   a. Pick the best node on OPEN

      select *: select n from open*

   b Generate its successors

      Expand *Generate the successor of n using set of operators- o*

b. For each successor do:

i. If it has not been generated before, evaluate it, add it to OPEN, and record its parent.

```
V succ m , if m ~€ {OPEN U CLOSE}
 set C(m)=C(n)+C(n,m)
 insert m in OPEN
```

ii If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

```
if m € {OPEN U CLOSE}
     set C(m) = min{ C(m),C(n)+C(n,m)}
     if C{m} has decreased
            and m € {CLOSE}  move it to OPEN
             if m € {OPEN}  change the cost
```

Best First Search

- Combines the advantages of both DFS and BFS into a single method.
- DFS is good because it allows a solution to be found without all competing branches having to be expanded.
- BFS is good because it does not get branches on dead end paths.
- One way of combining the two is to follow a single path at a time, but switch paths whenever some competing path looks more promising than the current one does.
- At each step of the BFS search process, we select the most promising of the nodes we have generated so far.
- This is done by applying an appropriate heuristic function to each of them.
- We then expand the chosen node by using the rules to generate its successors

Best First Search

- Similar to Steepest ascent hill climbing with two exceptions:
  - In hill climbing, one move is selected and all the others are rejected, never to be reconsidered. This produces the straight-line behavior that is characteristic of hill climbing.
  - In BFS, one move is selected, but the others are kept around so that they can be revisited later if the selected path becomes less promising.
  - The best available state is selected in the BFS, even if that state has a value that is lower than the value of the state that was just explored. This contrasts with hill climbing, which will stop if there are no successor states with better values than the current state.

OR-graph

- It is sometimes important to search graphs so that duplicate paths will not be pursued.
- An algorithm to do this will operate by searching a directed graph in which each node represents a point in problem space.
- Each node will contain:
  - Description of problem state it represents
  - Indication of how promising it is
  - Parent link that points back to the best node from which it came
  - List of nodes that were generated from it

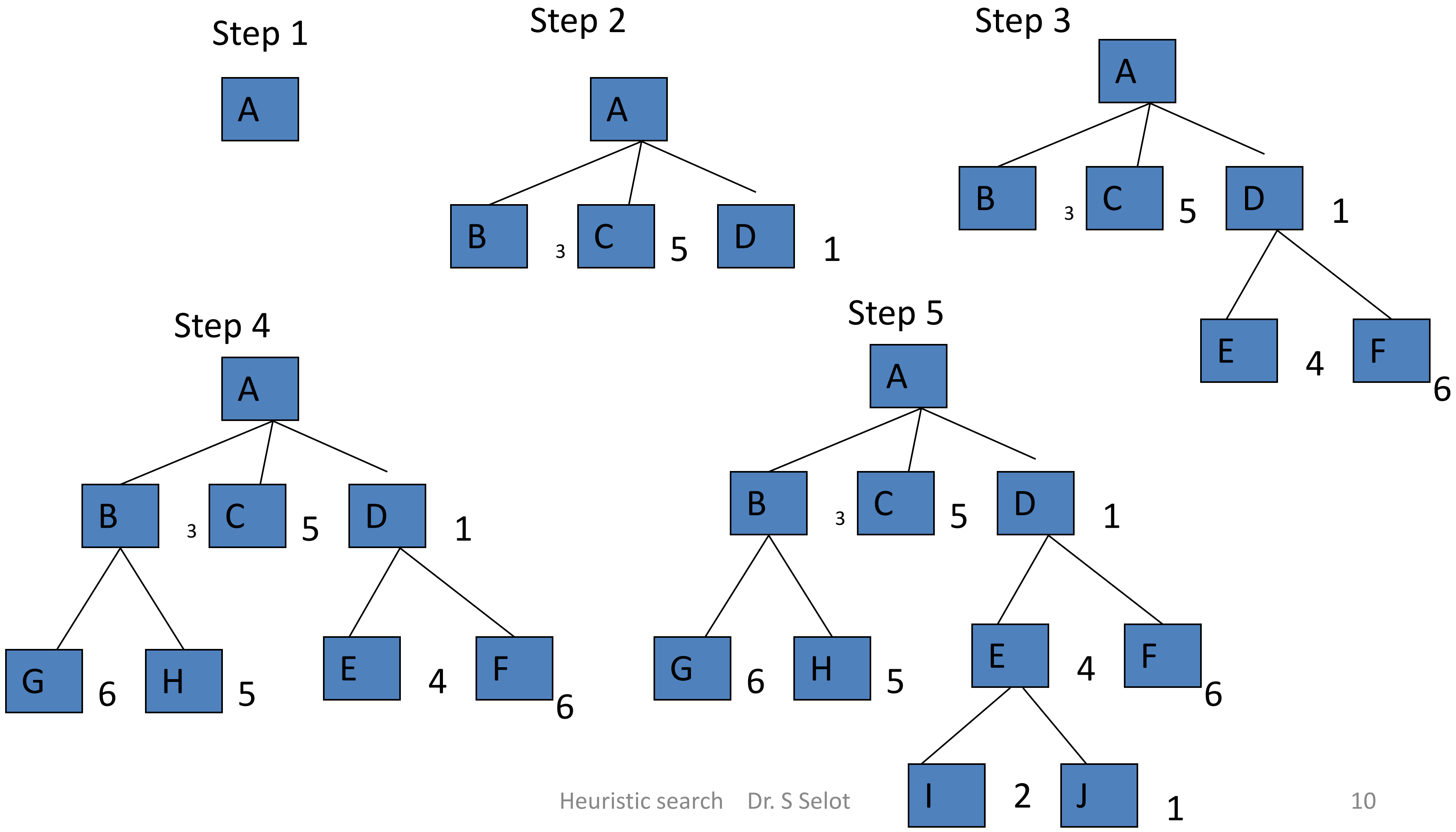
OR-Graph

- Parent link will make it possible to recover the path to the goal once the goal is found.

- The list of successors will make it possible, if a better path is found to an already existing node, to propagate the improvement down to its successors.

- This is called OR-graph, since each of its branches represents an alternative problem solving path

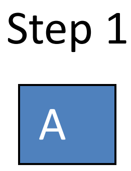
BFS

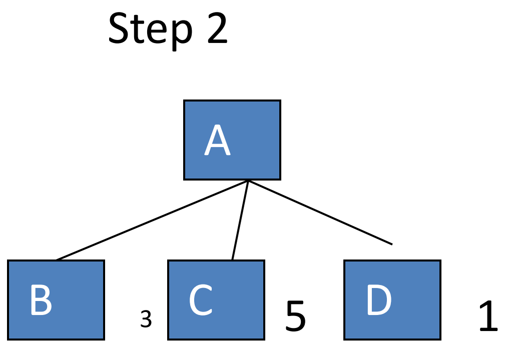

Step 1
A
Step 2
A
B 3 C 5 D 1
Step 3
A
B 3 C 5 D 1
E 4 F 6
Step 4
A
B 3 C 5 D 1
G 6 H 5 E 4 F 6
Step 5
A
B 3 C 5 D 1
G 6 H 5 E 4 F 6
I 2 J 1

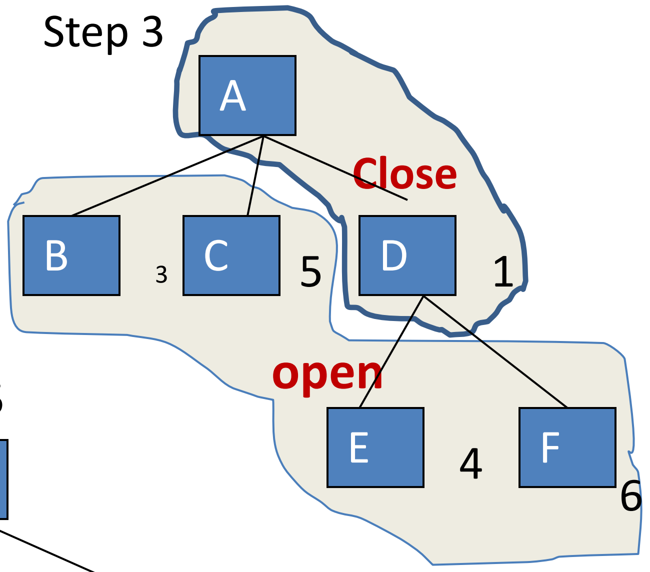
Implementation of OR graphs

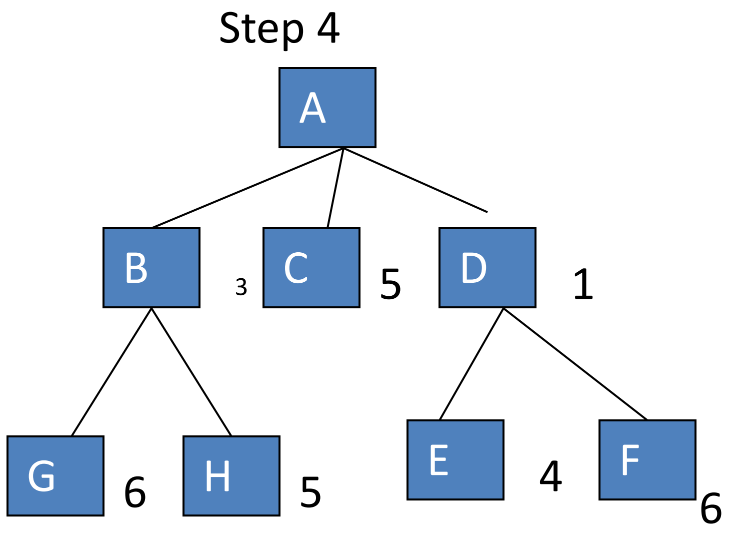
- We need two lists of nodes:
  - OPEN LIST
    - Nodes that have been generated and have had the heuristic function applied to them but which have not yet been examined.
    - OPEN is actually a priority queue in which the elements with the highest priority are those with the most promising value of the heuristic function.
  - CLOSED LIST
    - Nodes that have already been examined. We need to keep these nodes in memory if we want to search a graph rather than a tree
    - since whenever a new node is generated, we need to check whether it has been generated before.

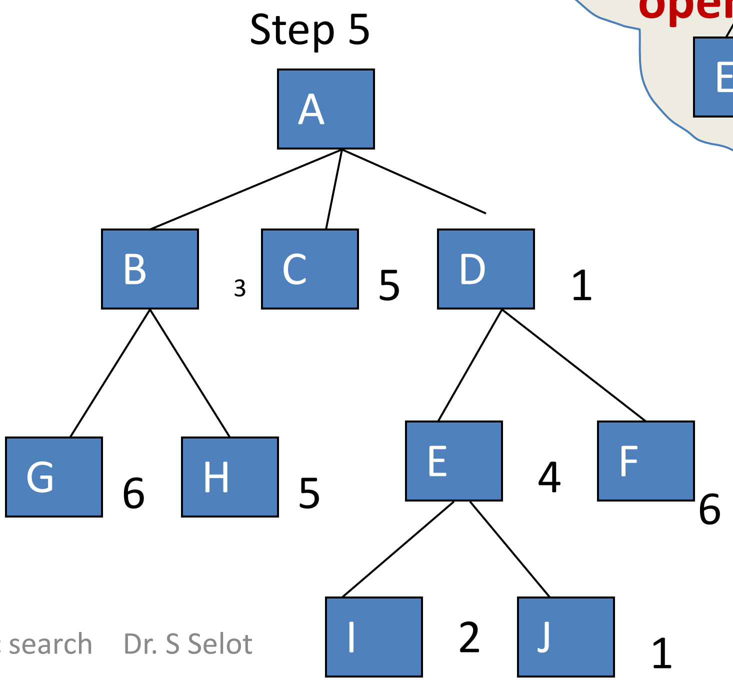
BFS


Step 1
A


Step 2
A
B
3
C
5
D
1


Step 3
A
Close
B
3
C
5
D
1
open
E
4
F
6


Step 4
A
B
3
C
5
D
1
G
6
H
5
E
4
F
6


Step 5
A
B
3
C
5
D
1
G
6
H
5
E
4
F
6
I
2
J
1

Algorithm: BFS

Given : s-starting state, n-any node, g- goal state, o-set of operators, h-estimated cost of moving from current to goal state

1. Start with OPEN containing just the initial state
   initialization *open = {s} closed={ }*
2. Until a goal is found or there are no nodes left on OPEN do
   { **fail** if open ={ }; **terminate** if n € G then terminate with success }
   a. Pick the best node on OPEN
      select *: select n from open*
   b Generate its successors
      Expand *Generate the successor of n using set of operators- o*

b. For each successor do:

i. If it has not been generated before, evaluate it, add it to OPEN, and record its parent.

```
V succ m , if m ~€ {OPEN U CLOSE}
 set C(m)=h'(m)
 insert m in OPEN
```

ii If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

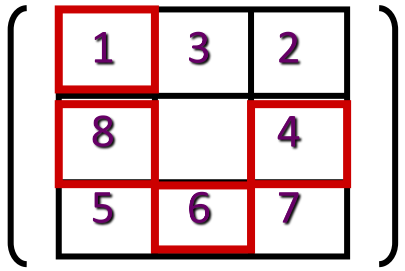
```
if m € {OPEN U CLOSE}
    set C(m) = min{ C(m),h'(m)}
    if C{m} has decreased
        and m € {CLOSE}  move it to OPEN
        if m € {OPEN}  update the cost
```

BFS : simple explanation

- It proceeds in steps, expanding one node at each step, until it generates a node that corresponds to a goal state.
- At each step, it picks the most promising of the nodes that have so far been generated but not expanded.
- It generates the successors of the chosen node, applies the heuristic function to them, and adds them to the list of open nodes, after checking to see if any of them have been generated before.
- By doing this check, we can guarantee that each node only appears once in the graph, although many nodes may point to it as a successor.

Examples (2): 8-puzzle

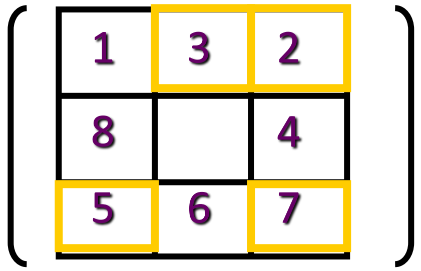
- f1(T) = the number correctly placed tiles on the board:

f1 

= 4

◎ **f2(T) = number or incorrectly placed tiles on board:**

➔ **gives (rough!) estimate of how far we are from goal**

f2 

= 4

Most often, 'distance to goal' heuristics are more useful !

Examples (3): Manhattan distance

- f3(T) = the sum of (the horizontal + vertical distance that each tile is away from its final destination):
  - gives a better estimate of distance from the goal node

$$f3\begin{pmatrix}1 & \boxed{5} & \boxed{2}\\ 8 & & 4\\ \boxed{3} & 6 & \boxed{7}\end{pmatrix} = 1 + 4 + 2 + 3 = 10$$

Heuristics: Example

- 8-puzzle: h(n) = tiles out of place

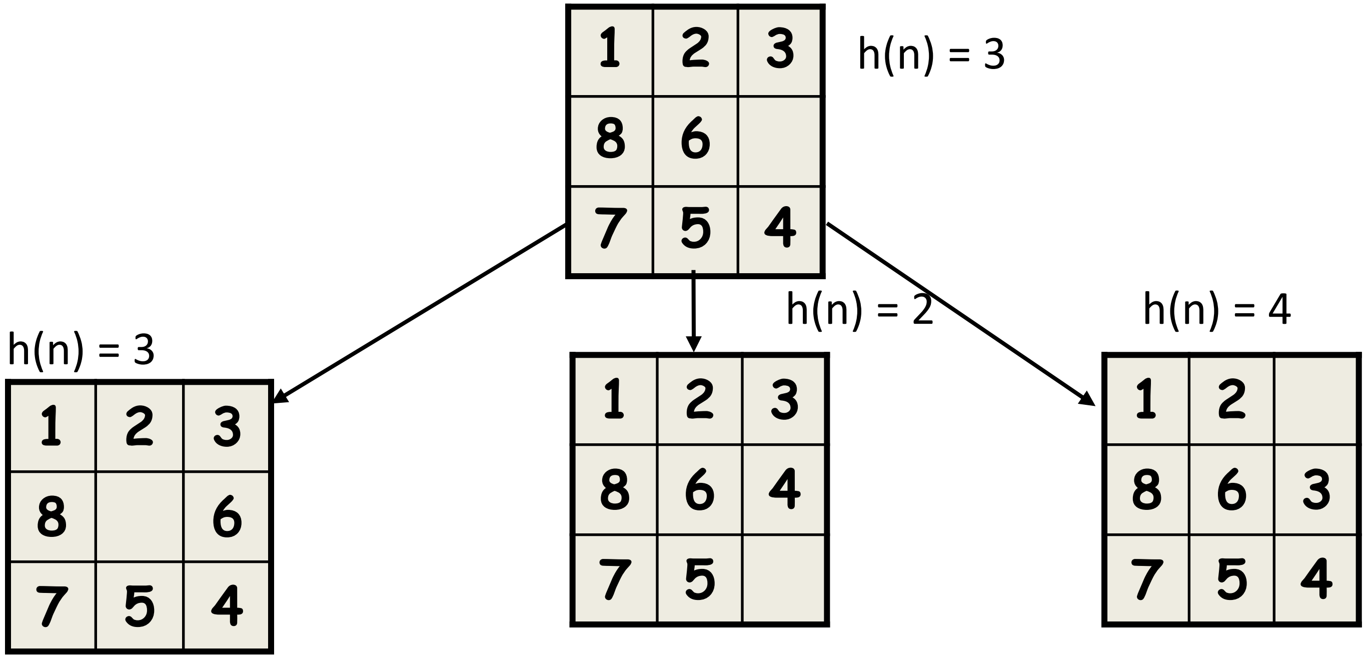
1	2	3
8	6	
7	5	4

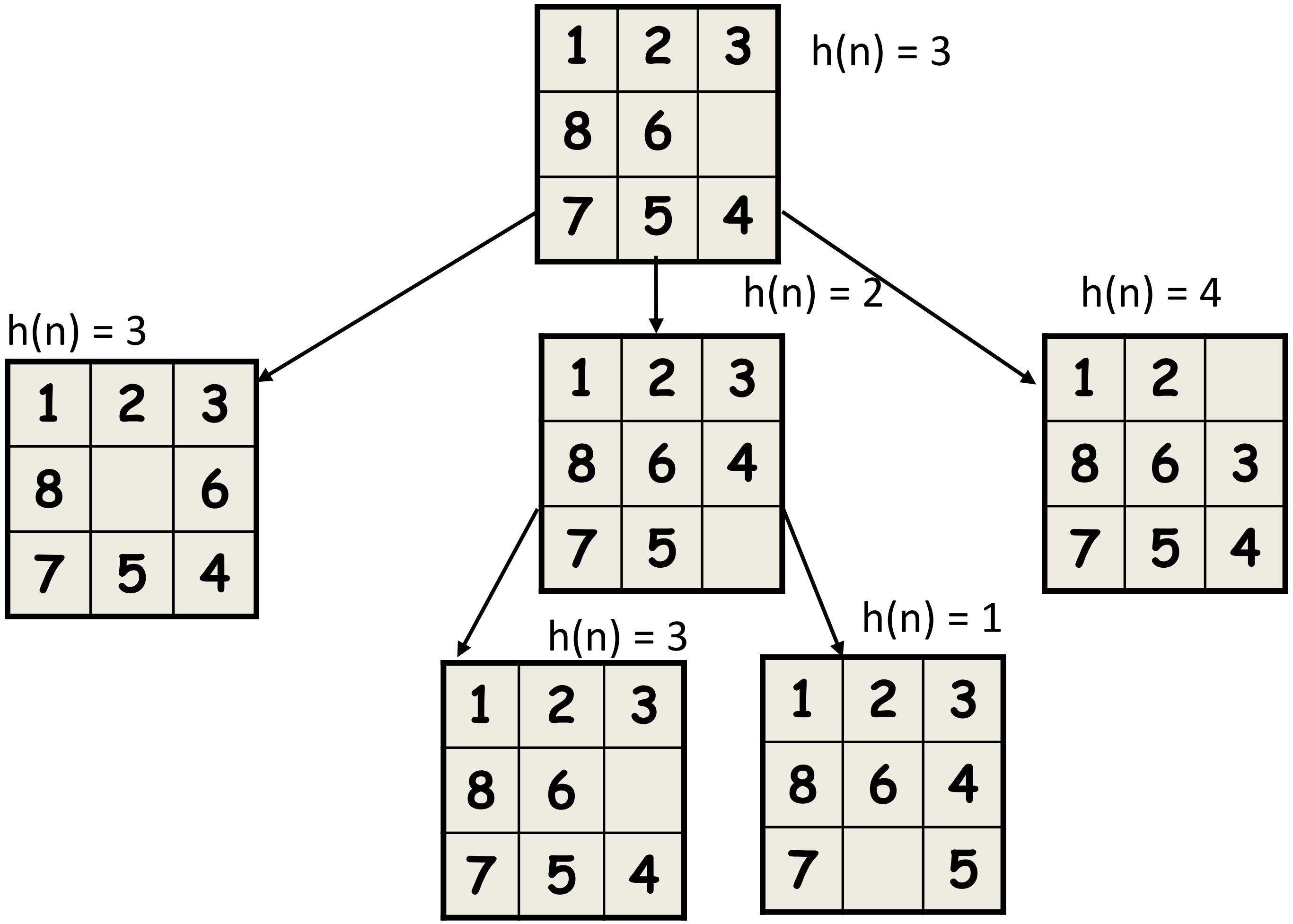
h(n) = 3

Goal state

1	2	3
8		4
7	6	5

Example - cont




h(n) = 3
1 2 3 / 8 6 _ / 7 5 4
h(n) = 3
1 2 3 / 8 _ 6 / 7 5 4
h(n) = 2
1 2 3 / 8 6 4 / 7 5 _
h(n) = 4
1 2 _ / 8 6 3 / 7 5 4
h(n) = 3
1 2 3 / 8 6 _ / 7 5 4
h(n) = 1
1 2 3 / 8 6 4 / 7 _ 5

A* Algorithm

- BFS is a simplification of A* Algorithm
- Presented by Hart et al
- Algorithm uses:
  - **f’:** Heuristic function that estimates the merits of each node we generate. This is sum of two components, g and h’ and f’ represents an estimate of the cost of getting from the initial state to a goal state along with the path that generated the current node.
  - **g** : The function g is a measure of the cost of getting from initial state to the current node.
  - **h’** : The function h’ is an estimate of the additional cost of getting from the current node to a goal state.
  - OPEN
  - CLOSED

A* Algorithm

1. Start with OPEN containing only initial node. Set that node's g value to 0, its h' value to whatever it is, and its f' value to h'+0 or h'. Set CLOSED to empty list.

   Initialize *open = {s}   closed = { }   g(s)=0   f'(s)=h'(s)*

2. Until a goal node is found, repeat the following procedure:

   (a) If there are no nodes on OPEN, report failure.

   fail *if open={ } terminate with failure*.

   (b) Otherwise pick the node on OPEN with the lowest f' value.

   Call it BESTNODE. Remove it from OPEN. Place it in CLOSED.

   Select *Select minimum cost state n from OPEN Save n in CLOSE*

   (c) See if the BESTNODE is a goal state. If so exit and report a solution.

   terminate *if n € G terminate with success and f'(n)*

A* algorithm

d)Otherwise, generate the successors of BESTNODE
For each of the SUCCESSOR, do the following:

Expand:

```
    ∨SUCC m of n
      if m ~€ {OPEN U CLOSE }
                          set g(m) = g(n)+C(n,m)
                        set f’(m)=g(m)+h’(m)
                      Insert m in open
                   set the link to parent node

      if m € {OPEN U CLOSE }
                          set g(m)=min[g(m),g(n)+C(n,m)]
                          set f’(m)=g(m)+h’(m)
      if f’(m) has decreased and m € {CLOSE }   move it to OPEN
       if f’(m) has decreased and m € {OPEN}   update the cost and change the link to parent node
```

Observations about A*

- Role of g function: This lets us choose which node to expand next on the basis of not only of how good the node itself looks, but also on the basis of how good the path to the node was.

- h’, the distance of a node to the goal. If h’ is a perfect estimator of h, then A* will converge immediately to the goal with no search.

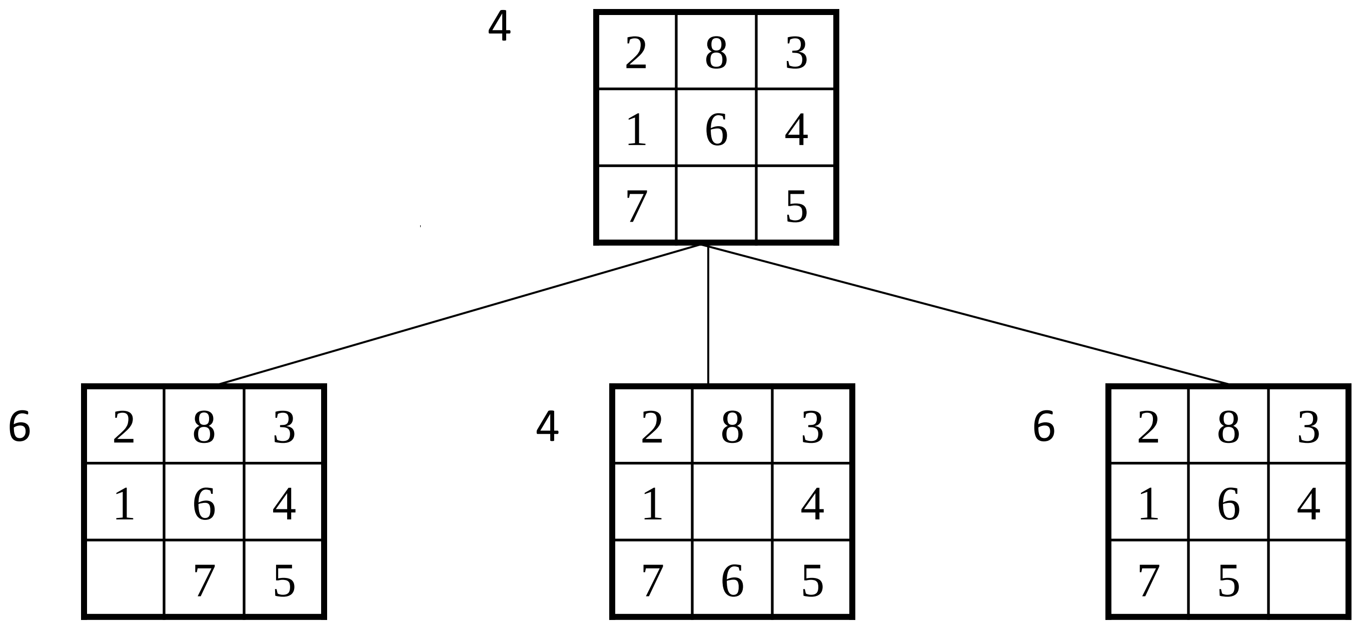
8-Puzzle Evaluation Function

$$f(n) = g(n) + h(n)$$

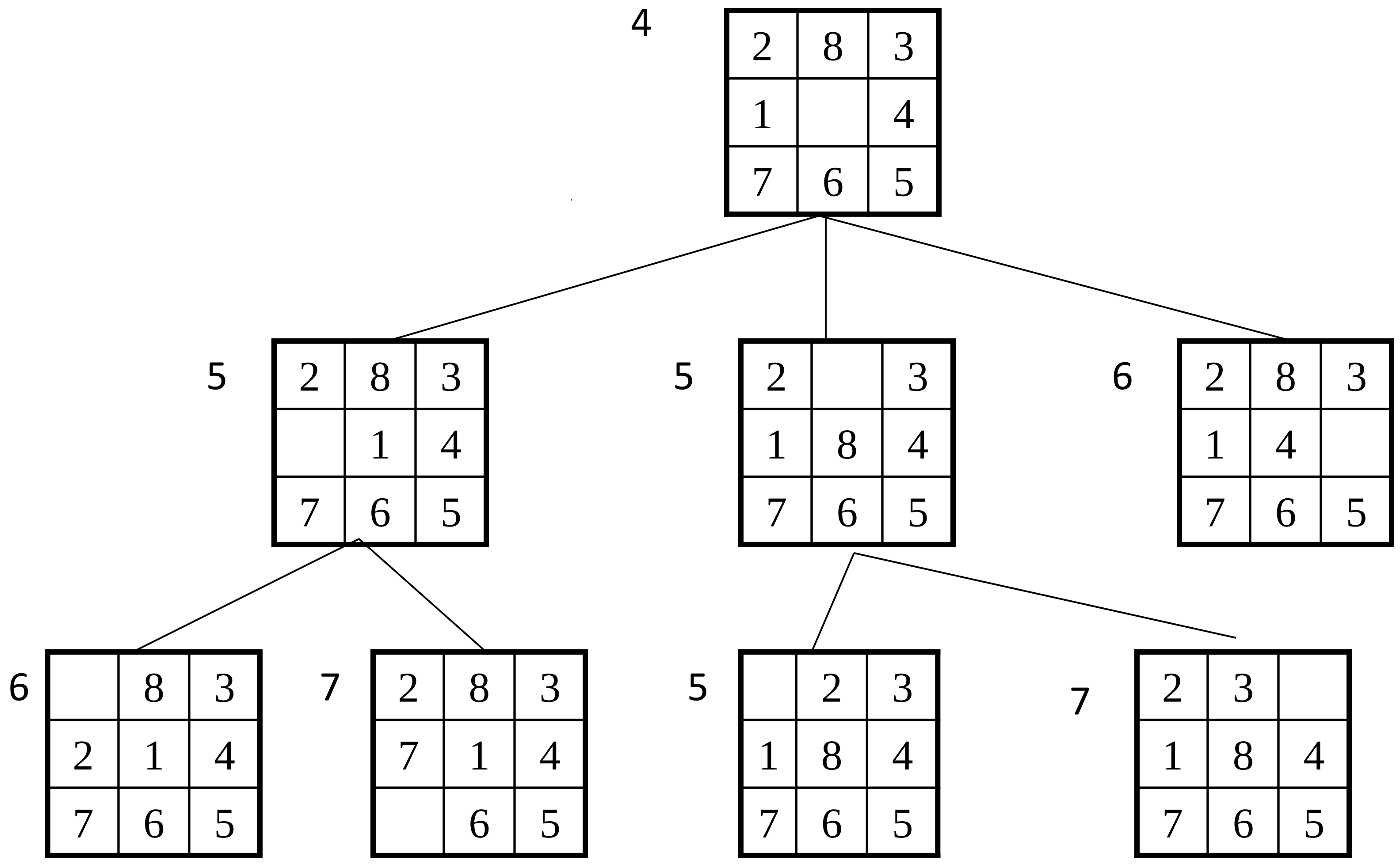
h(n) counts the number of displaced tiles in that database associated with the node *n*

g(n) is the depth of the node in the search tree

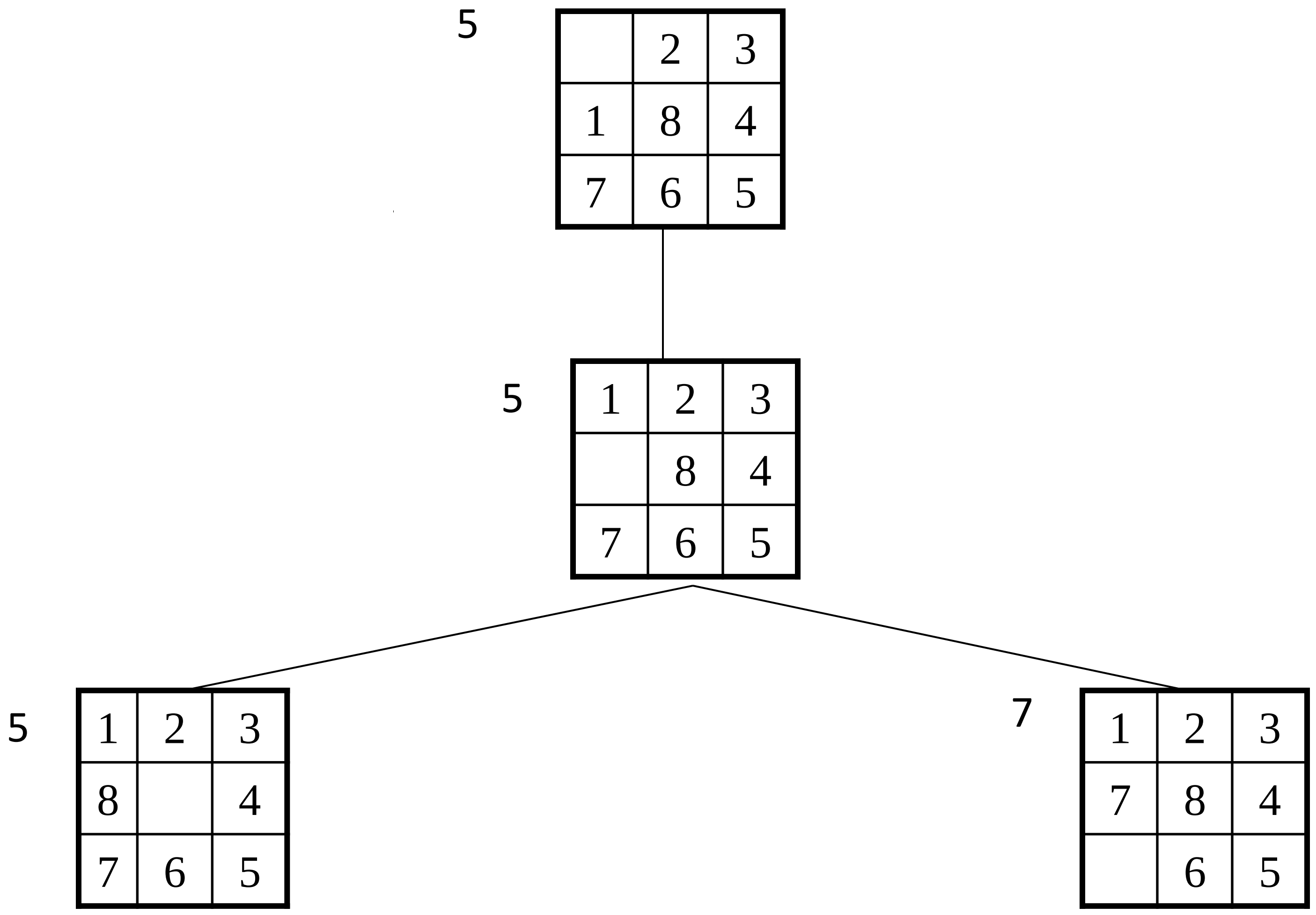
Playing 8-Puzzle



Playing 8-Puzzle



Playing 8-Puzzle

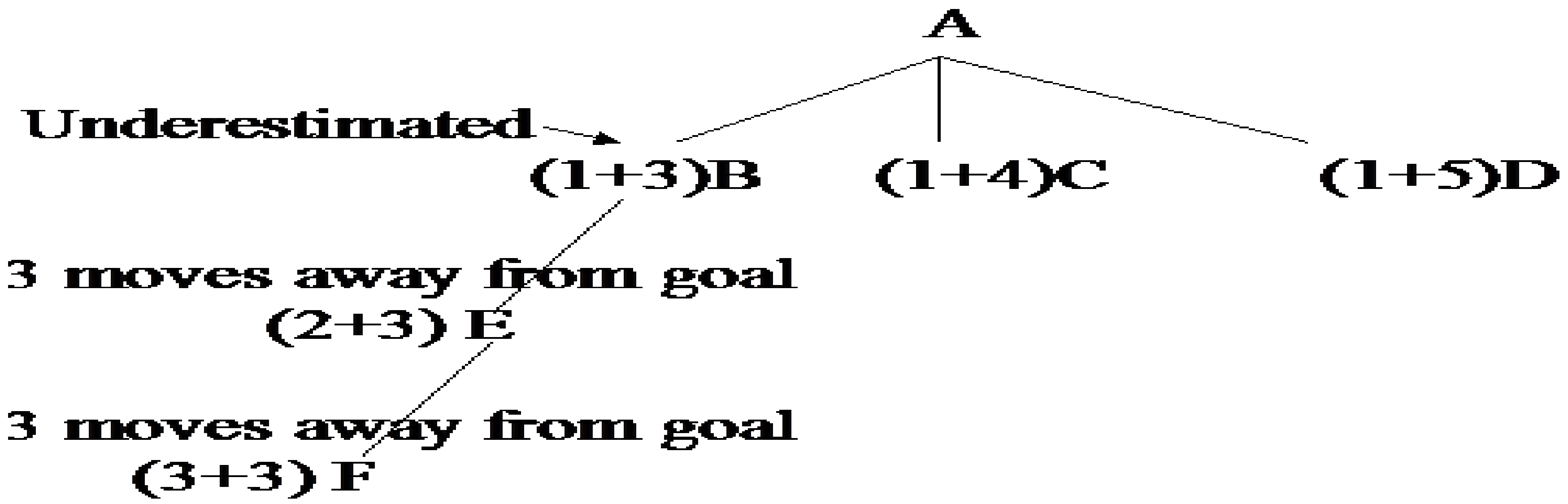


One code, many algorithms

- When h(n) = 0. => Uniform Cost Method
  - Search only depends on g
- When g(n)=0 => Best First Search
  - Search depends on h
- When f'=g+h' => A* algorithm
  - Search depends on both g and h
- When g(n) = 1, h(n) = 0
  - A* behaves like breadth first search
- If non-admissible heuristic
  - g(n) = 0, h(n) = 1/depth => depth first

Behavior of A* search

- **Underestimation and overestimation**
  - **Underestimation:**
    - If we can guarantee that h never over estimates actual value from current to goal, then A* algorithm is guaranteed to find an optimal path to a goal, if one exists.
    - We say that h value of node X is **underestimated** when heuristic value is less than actual value from X to goal
    - Consider the following example.



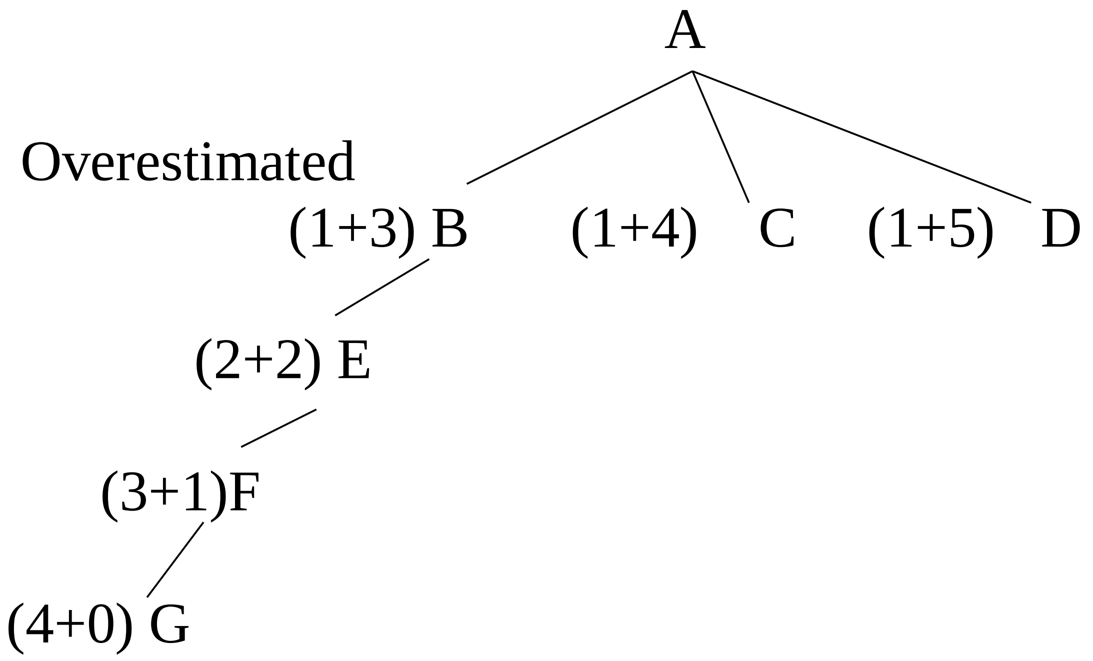
- Assume the cost of all arcs is 1.
- We see that f (E) = 5 = f (C)
- Suppose we resolve in favor of E, the path currently we are expanding.
- Clearly expansion of F(f = 6) is stopped and we will now expand C.
- Thus we see that by underestimating h(B), we have wasted some effort but eventually discovered that B was farther away than we thought.
- Then we go back and try another path, and will find optimal path.

Overestimation

- Now consider another situation. We expand B to E, E to F and F to G for a solution path of length 4. But suppose that there a direct path from D to a solution giving a path of length 2.

- We will never find it because of overestimating h(D). We may find some other worse solution without ever expanding D.

- So by overestimating h, we can not be guaranteed to find the cheaper path solution.

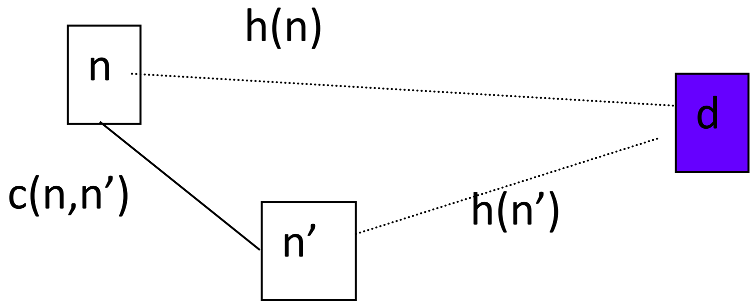

A
Overestimated
(1+3) B
(1+4) C
(1+5) D
(2+2) E
(3+1)F
(4+0) G

Properties of A*

- A* generates an optimal solution if h(n) is an admissible heuristic and the search space is a tree:
  - h(n) is **admissible** if it never overestimates the cost to reach the destination node

- A* generates an optimal solution if h(n) is a consistent heuristic and the search space is a graph:
  - h(n) is **consistent** if for every node n and for every successor node n' of n:

    $h(n) \leq c(n,n') + h(n')$



• If h(n) is consistent then h(n) is admissible
•Frequently when h(n) is admissible, it is also consistent

Admissibility of A*

- A search algorithm is **admissible,** if for any graph, it always terminates in an optimal path from initial state to goal state, if path exists.

- If heuristic function **h** is **underestimate** of actual value from current state to goal state, then the it is called **admissible function**.
- So, we can say that A* always terminates with the optimal path in case h(x) is an **admissible heuristic function**.

Monotonicity:

- A heuristic function **h** is monotone if
  1. ∀ states X_i and X_j such that X_j is successor of X_i
     $h(X_i) – h(X_j) \leq cost(X_i, X_j)$ i.e., actual cost of going from X_i to X_j
  2. h (goal) = 0

Gracefull Decay of Admissibility

- If h’ rarely overestimates h by more than δ, then A* algorithm will rarely find a solution whose cost is more than δ greater than the cost of the optimal solution.

- Under certain conditions, the A* algorithm can be shown to be optimal in that it generates the fewest nodes in the process of finding a solution to a problem.