Informed Search

INFORMED SEARCH

- All the previous searches have been blind searches .They make no use of any knowledge of the problem

- When more information than the initial state , the operator , and the goal test is available, the size of the search space can usually be constrained.

- HEURISTIC INFORMATION:

  Information about the problem (the nature of the states, the cost of transforming from one state to another , the promise of taking a certain path, and the characteristics of the goals).can sometimes be used to help guide the search more efficiently.

  Information in form of heuristic evaluation function=f(n,g),a function of the nodes n, and/or the goals g.

- They help to reduce the number of alternatives from an exponential number to a polynomial number and , thereby , obtain a solution in a tolerable amount of time.

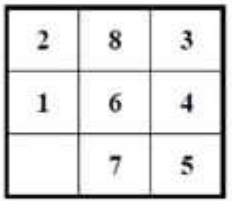
Heuristics

- A heuristic is a rule of thumb for deciding which choice might be best
- There is no general theory for finding heuristics, because every problem is different
- Choice of heuristics depends on knowledge of the problem space
- An informed guess of the next step to be taken in solving a problem would prune the search space
- A heuristic may find a sub-optimal solution or fail to find a solution since it uses limited information
- In search algorithms, heuristic refers to a function that provides an estimate of solution cost

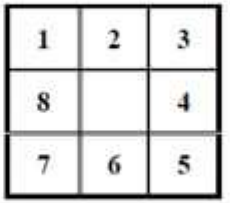
The notion of Heuristics

- Heuristics use domain specific knowledge to estimate the quality or potential of partial solutions.
- Example:
- Manhattan distance heuristic for 8 puzzle.
- Minimum Spanning Tree heuristic for TSP.

The 8-puzzle



Initial State



Goal state

- **Heuristic Fn-1 :** Misplaced Tiles Heuristics is the number of tiles out of place.
- The first picture shows the current state *n, and the second picture the goal state.*
- *h(n) = 5* because the tiles 2, 8, 1, 6 and 7 are out of place.
- **Heuristic Fn-2:** Manhattan Distance Heuristic: Another heuristic for 8-puzzle is the Manhattan distance heuristic. This heuristic sums the distance that the tiles are out of place. The distance of a tile is measured by the sum of the differences in the x-positions and the y-positions.
- For the above example, using the Manhattan distance heuristic,
- *h(n) = 1 + 1 + 0 + 0 + 0 + 1 + 1 + 2 = 6*
- This piece will have to be moved *at least* that many times to get it to where it belongs
- Suppose, from a given position, we try every possible single move (there can be up to four of them), and pick the move with the smallest sum
- This is a reasonable *heuristic* for solving the 8-puzzle

The Informed Search Problem

- Given [S,s,O,G,h] where
  - S is the (implicitly specified) set of states.
  - s is the start state.
  - O is the set of state transition operators each having some cost.
  - G is the set of goal states.
  - h() is a heuristic function estimating the distance to a goal.
- To find :
  - A min cost seq. of transition to a goal state .

Hill Climbing

- Hill climbing is a variant of Generate and test in which feedback from test procedure is used to help the generator decide which direction to move in search space.
- Feedback is provided in terms of heuristic function

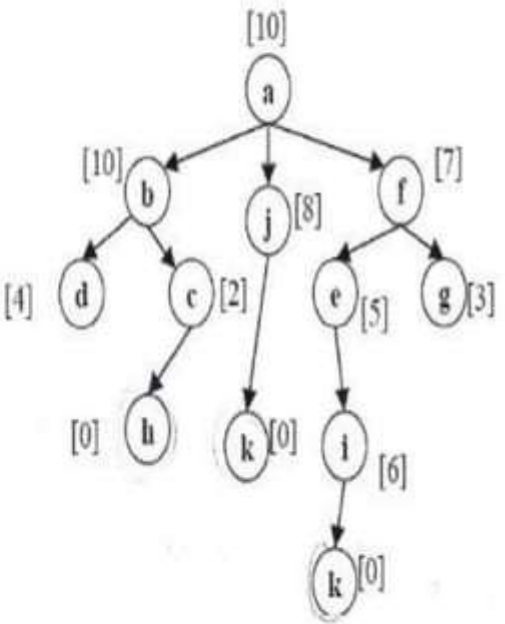
HILL CLIMBING SEARCH

1. Evaluate the initial state. If it is a goal state then return it and quit. Otherwise, continue with initial state as current state.
2. Loop until a solution is found or there are no new operators left to be applied in current state:
   - Select an operator that has not yet been applied to the current state and apply it to produce a new state
   - Evaluate the new state:
     * if it is a goal state, then return it and quit
     * if it is not a goal state but better than current state then make new state as current state
     * if it is not better than current state then continue in the loop

- It is a simply a loop that continually moves in the direction of increasing value.
- The algorithm does not maintain a search tree, so the node structures need only record the state and its elevation which denote by VALUE.
- It terminates when it reaches a "peak" where no neighbor has a higher value.
- When there is more than one best successor choose from, the algorithm can select them at random.

Hill Climbing Example

- Goal state : h and k
- Local minimum: A-> F -> G
- Solution:
  - A, J, K
  - A, F, E, I, K



Steepest-Ascent Hill Climbing (Gradient Search)

- Considers all the moves from the current state.
- Selects the best one as the next state.

Steepest-Ascent Hill Climbing (Gradient Search)

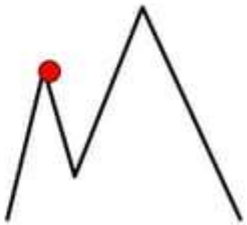
1. Evaluate the initial state. If it is a goal state then return it and quit. Otherwise, continue with initial state as current state.

2. Loop until a solution is found or a complete iteration produces no change to current state:

   - let SUCC be a state such that any possible successor of the current state will be better than SUCC (the worst state).
   - For each operator that applies to the current state do:
     * Apply any operator and generate new state
     * evaluate the new state:
       * if it is a goal state, then return it and quit
       * if it is not a goal state, compare it to SUCC.
         If it is better than set SUCC to this state
         If it is not better than leave SUCC alone
     * if SUCC is better than the current state then set the current state to SUCC.

DRAWBACK

- Local maxima: A local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum. Once a local maximum peak the algorithm is halt even though the solution may be far from satisfactory.



- Plateau: A plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum , from which no uphill exit exists .The search will conduct a random walk.

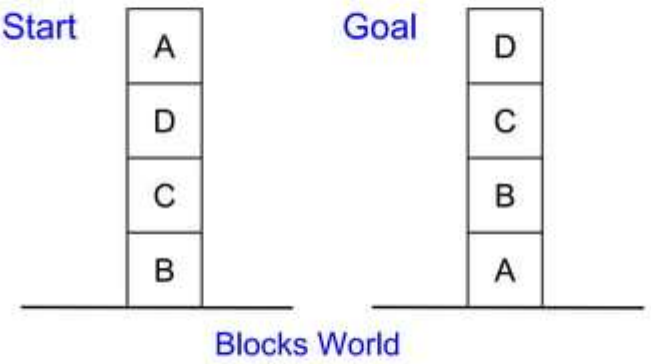
- Ridges: A ridge is a special kind of local maximum. It is an area of the search space that is higher than surrounding areas and that itself has a slope (which one would like to climb). But the orientation of the high region, compared to the set of available moves and the directions in which they move, makes it impossible to traverse a ridge by single moves.

Hill Climbing: Disadvantages

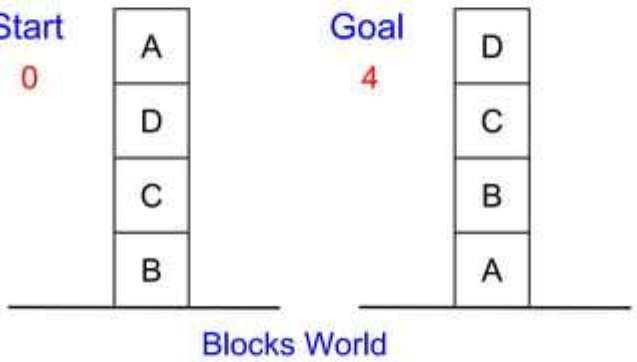
Ways Out

- **Local maximum :**Backtrack to some earlier node and try going in a different direction.
- **Plateau:** Here make a big jump to some direction and try to get to new section of the search space.
- **Ridge:** Here apply two or more rules before doing the test i.e., moving in several directions at once.
- Hill climbing is a local method:
  Decides what to do next by looking only at the "immediate" consequences of its choices.
- Global information might be encoded in heuristic functions.

Hill Climbing: Disadvantages



Hill Climbing: Disadvantages

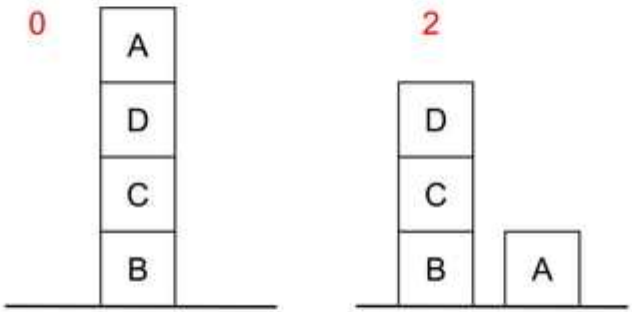


Local heuristic:

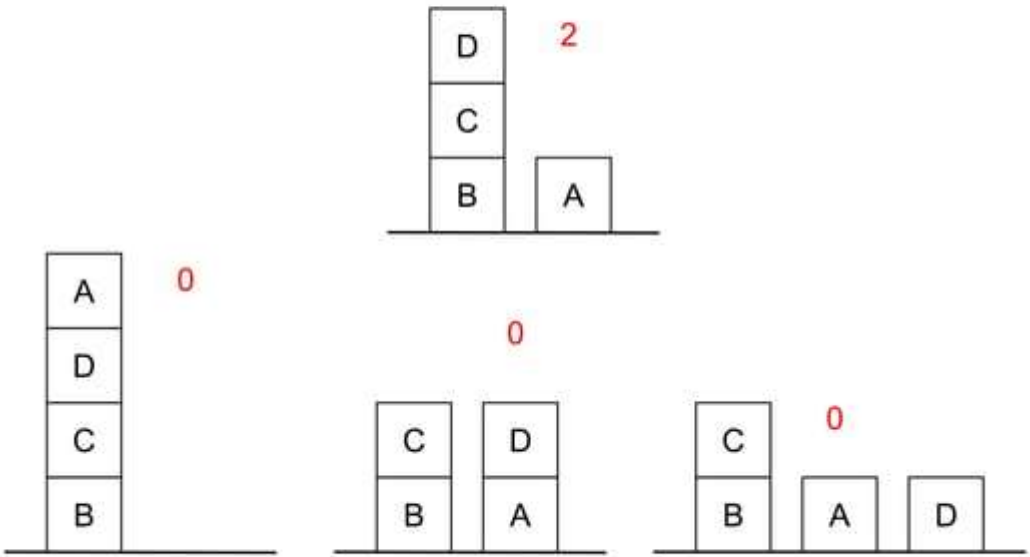
+1 for each block that is resting on the thing it is supposed to be resting on.

–1 for each block that is resting on a wrong thing.

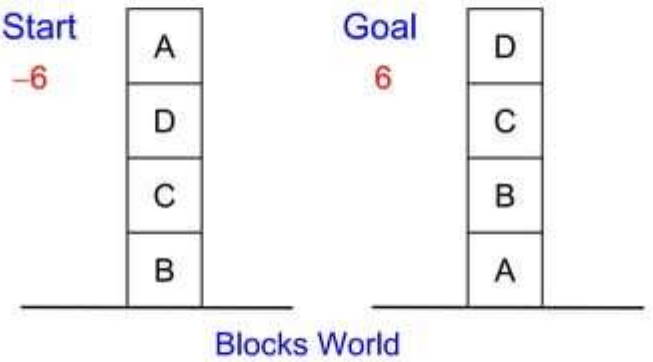
Hill Climbing: Disadvantages



Hill Climbing: Disadvantages



Hill Climbing: Disadvantages

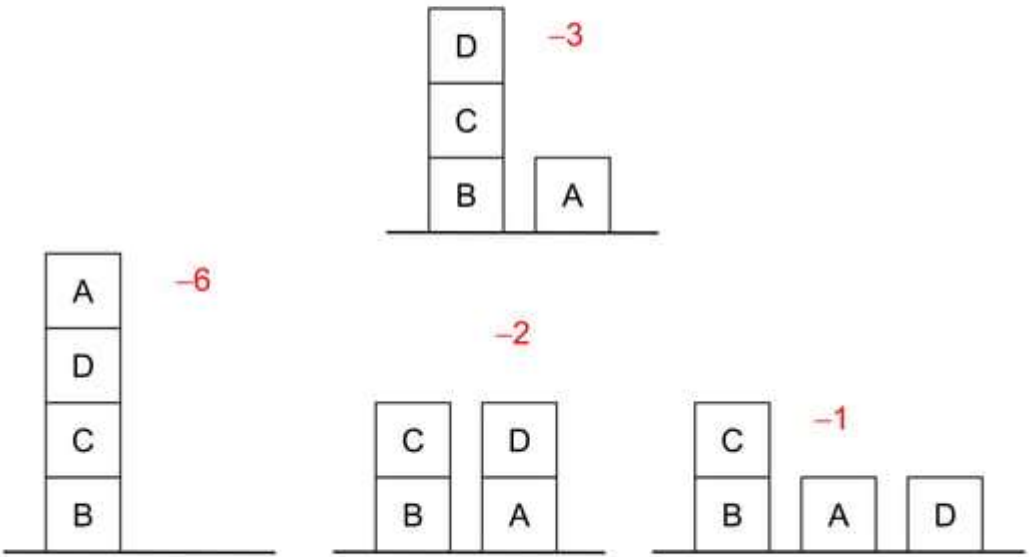


Global heuristic:

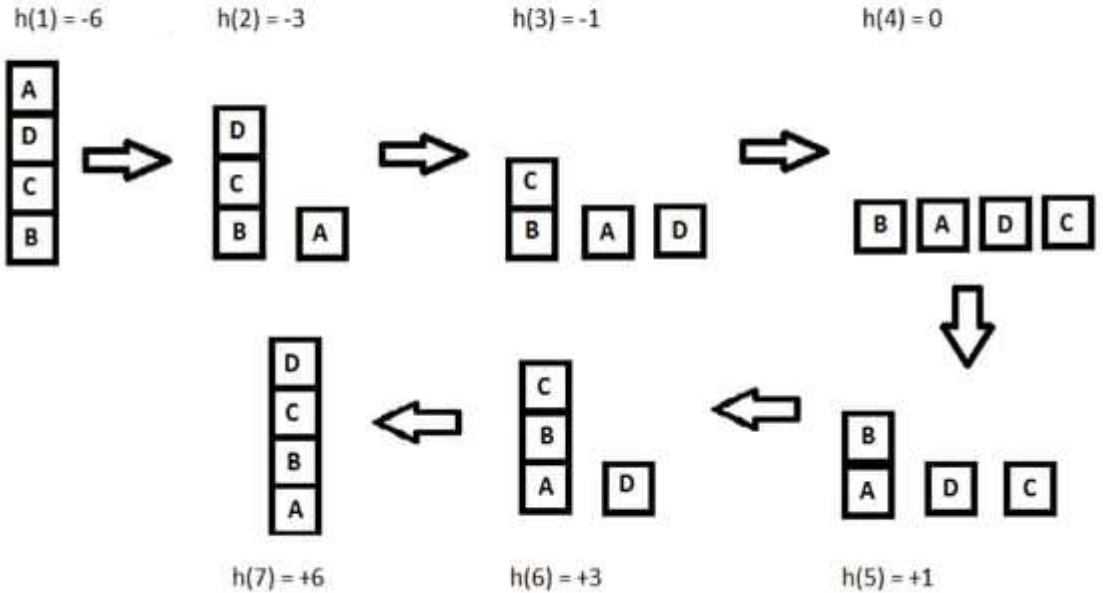
For each block that has the correct support structure: +1 to every block in the support structure.

For each block that has a wrong support structure: –1 to every block in the support structure.

Hill Climbing: Disadvantages



One solution with Global heuristic



Hill Climbing: Conclusion

- Can be very inefficient in a large, rough problem space.
- Global heuristic may have to pay for computational complexity.
- Often useful when combined with other methods, getting it started right in the right general neighbourhood.

Simulated annealing search

- A hill-climbing algorithm that never makes "downhill" moves towards states with lower value (or high cost) is guaranteed to be incomplete , because it can get stuck on a local maxima.
- Instead of starting again randomly when stuck on a local maximum , we could allow the search to take some downhill steps to escape the local maximum. This is the idea of simulated annealing.
- Innermost loop of simulated annealing is quite similar to hill climbing . Instead of picking the best move, however , it picks a random move.
- Lowering the chances of getting caught at a local maximum, or plateau, or a ridge.

Simulated Annealing

- A variation of hill climbing in which, at the beginning of the process, some downhill moves may be made.

- To do enough exploration of the whole space early on, so that the final solution is relatively insensitive to the starting state.

Simulated Annealing

Physical Annealing

- Physical substances are melted and then gradually cooled until some solid state is reached.
- The goal is to produce a minimal-energy state.
- Annealing schedule: if the temperature is lowered sufficiently slowly, then the goal will be attained.
- Nevertheless, there is some probability for a transition to a higher energy state: $e^{-\Delta E/T}$.

Simulated Annealing

1. Evaluate the initial state. If it is a goal state then return it and quit. Otherwise, continue with initial state as current state.
2. Initialize best-so-far to the current state.
3. Initialize T according to the annealing schedule.
4. Loop until a solution is found or there are no new operators left to be applied in current state:
   1. Select an operator that has not yet been applied to the current state and apply it to produce a new state
   2. Evaluate the new state, compute:

      ΔE = Val(current state) - Val(new state)

      * if new state is a goal state, then return it and quit
      * if it is not a goal state but better than current state then make new state as current state
      * if it is not better than current state then make new state as current state with probability p` = e ^ -$\Delta$E/T. This step is usually implemented by invoking a random number generator to produce a number in the range of [0,1]. If the number is less then p`, then the move is accepted ,otherwise do nothing.
3. Revise T as necessary according to the annealing schedule.

Simulated Annealing Example

Place: where each tile I should go. Place(i)=i.
Position: where it is at any moment.
Energy: sum(distance(i, position(i))), for i=1,8.
Energy(solution) = 0

Initial Sta

4	1	7
6		2
3	5	8

Random neighbor: from each state there are at most 4 possible moves. Choose one randomly.
T = temperature. For example, we start with T=40, and at every iteration we decrease it by 1.
If T=1 then we stop decreasing it.

Energy = (2-1)+(6-2)+(7-3)+(4-1)+(8-5) +(6-4)+(7-3)+(9-8) = 1 + 4 + 4 + 3 + 3 + 2 + 4 + 1 = 22

Following are the **neighbors**

4		7
6	1	2
3	5	8

Energy = 25

4	1	7
	6	2
3	5	8

Energy = 21

4	1	7
6	2	
3	5	8

Energy = 21

4	1	7
6	5	2
3		8

Energy = 19

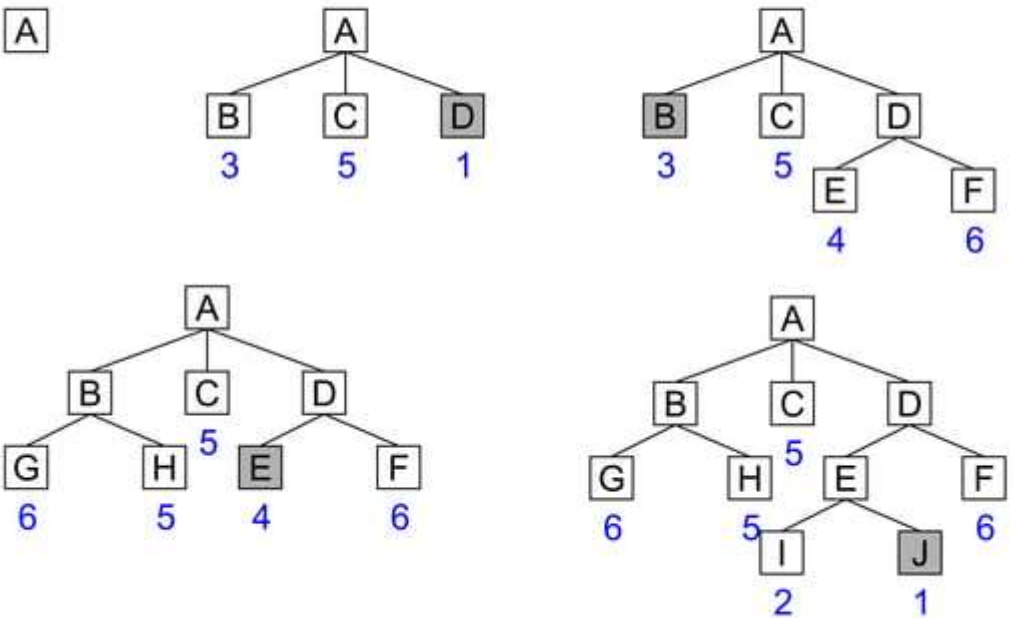
Suppose T = 40. Then the probability of the first neighbor is $e^{-(25-22)/40} = 0.9277 = 92.77\%$

Best-First Search

- Depth-first search: not all competing branches having to be expanded.
- Breadth-first search: not getting trapped on dead-end paths.

  $\Rightarrow$ Combining the two is to follow a single path at a time, but switch paths whenever some competing path look more promising than the current one.

Best-First Search



Best-first Vs Hill Climbing

hill climbing	best-first search
• In hill climbing, one move is selected and all the others are rejected and are never reconsidered.	• one move is selected, but the others are kept around so that they can be revisited later if the selected path becomes less promising
• It stop if there are no successor states with better values than the current state.	• the best available state is selected in best-first search, even if that state has a value that is lower than the value of the state that was just explored.

Best-First Search

- To get the path information, Each node will contain, in addition to a description of the problem state it represents,
  - an Indication of how promising it is,
  - a parent link that points back to the best node from which it came,
  - a list of the nodes that were generated from it.
- Once the goal is found the parent link will make it possible to recover the path to the goal.
- 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.
- We will call a graph of this sort an **OR *graph,*** since each of its branches represents an alternative problem-solving path.
- To implement such a graph-search procedure, we will need to use two lists of nodes:

- OPEN: nodes that have been generated, but have not examined. This is organized as a priority queue in which the elements with the highest priority are those with the most promising value of the heuristic function
- CLOSED: nodes that have already been examined. It used, whenever a new node is generated, check whether it has been generated before.

Best-First Search

Algorithm

1. Start with *OPEN* containing just the initial state.
2. Until a goal is found or there are no nodes left on *OPEN do:*
   I. Pick the best node on *OPEN.*
   II. Generate its successors.
   III. For each successor do:
      a. If it has not been generated before, evaluate it, add it to *OPEN, and record its* parent.
      b. 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.

Best-First Search

- Best-First Search

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

✓ $f(n)$ = measures the value of the current state (its "goodness").

✓ $h(n)$ = the estimated cost of the cheapest path from node n to a goal state.

✓ $g(n)$ = the exact cost of the cheapest path from the initial state to node n.

Traditional informed search strategies

- Greedy search:
  f(n)=h(n) ; i.e. g(n) = 0. it means the estimated cost of the cheapest path from node n to a goal state.

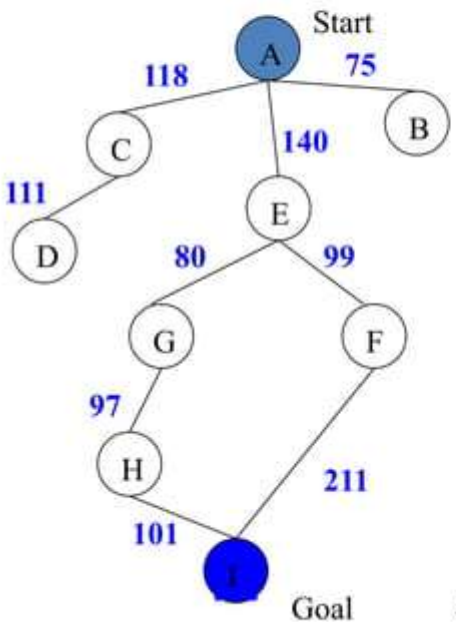
  Neither optimal nor complete

- Uniform-cost search:
  f(n)=g(n) ; i.e. h(n) = 0. it means the cost of the cheapest path from the initial state to node n.

  Optimal and complete, but very inefficient

- A* Search:
  - f(n)=g(n)+h(n), search uses an "admissible" heuristic function *f* that takes into account the current cost *g*
  - A heuristic function *f(n) = g(n) + h(n)* is admissible if *h(n)* **never** overestimates the cost to reach the goal.
  - *f(n)* never over-estimate the true cost to reach the goal state through node *n*.
  - Always returns the optimal solution path

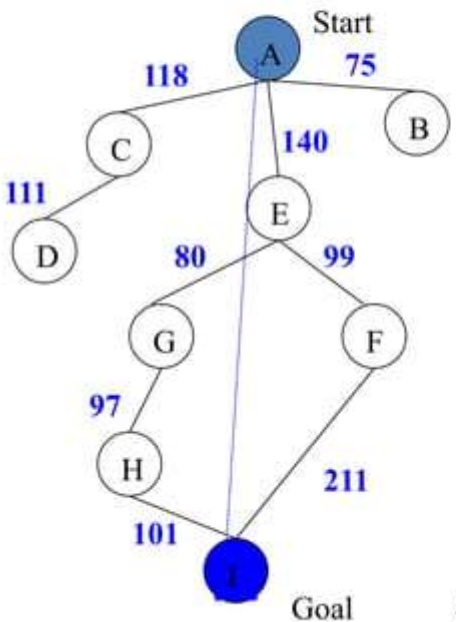
Greedy Best First Search



State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

***f(n) = h (n)* = straight-line distance heuristic**

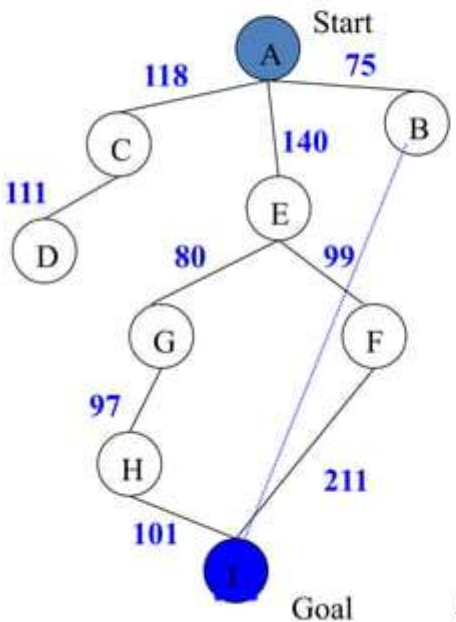
Greedy Best First Search



State	Heuristic: h(n)
A	**366**
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

***f(n) = h (n)* = straight-line distance heuristic**

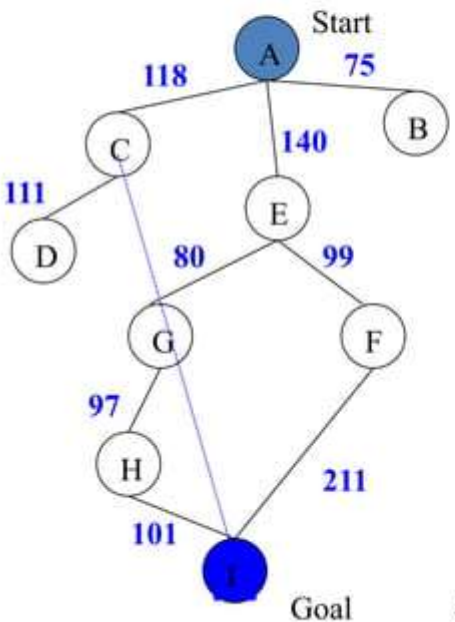
Greedy Best First Search



State	Heuristic: h(n)
A	366
B	**374**
C	329
D	244
E	253
F	178
G	193
H	98
I	0

***f(n) = h (n)* = straight-line distance heuristic**

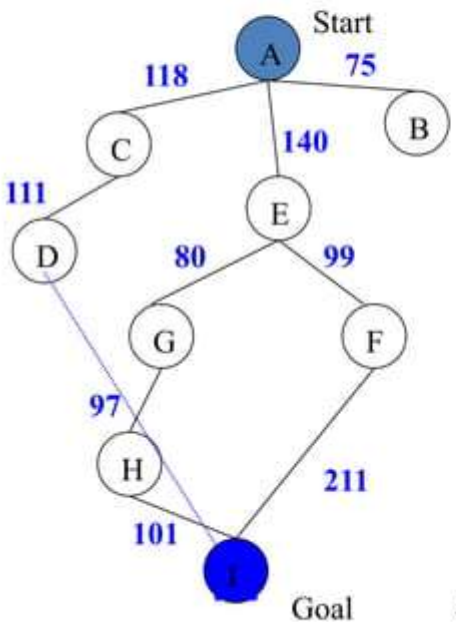
Greedy Best First Search



State	Heuristic: h(n)
A	366
B	374
C	**329**
D	244
E	253
F	178
G	193
H	98
I	0

***f(n)* = *h* (*n*) = straight-line distance heuristic**

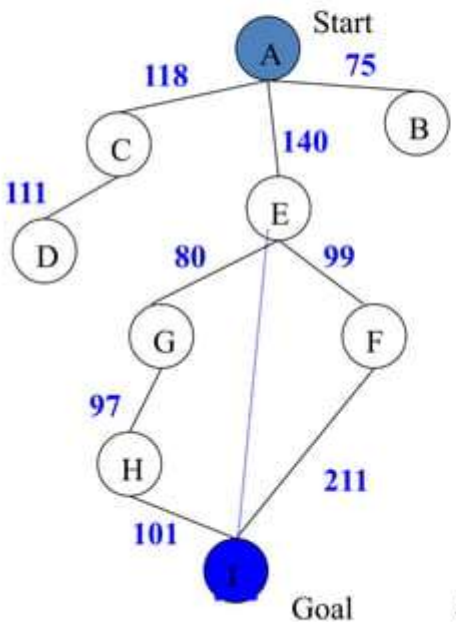
Greedy Best First Search



State	Heuristic: h(n)
A	366
B	374
C	329
D	**244**
E	253
F	178
G	193
H	98
I	0

***f(n) = h (n)* = straight-line distance heuristic**

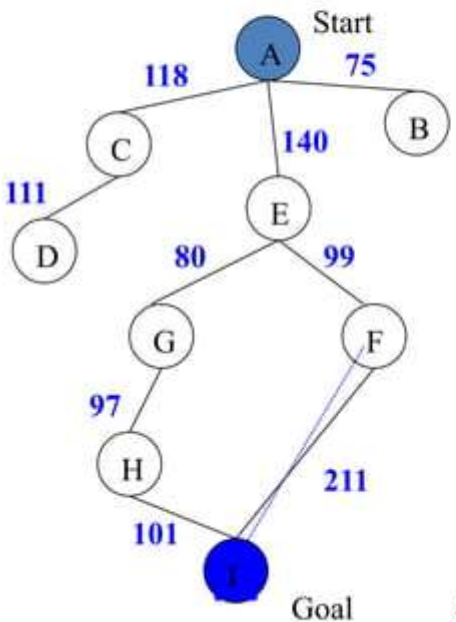
Greedy Best First Search



State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	**253**
F	178
G	193
H	98
I	0

***f(n) = h (n)* = straight-line distance heuristic**

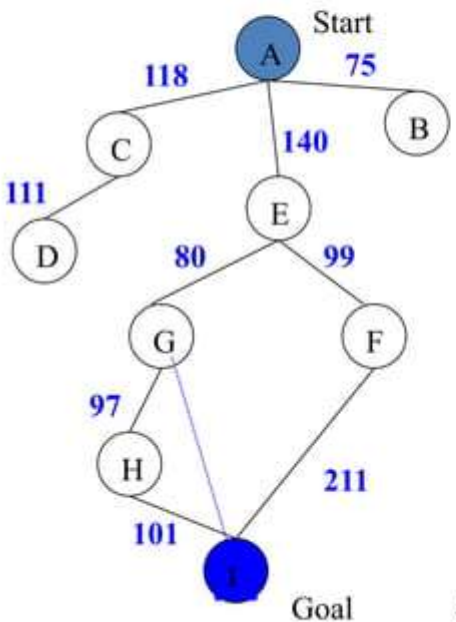
Greedy Best First Search



State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	**178**
G	193
H	98
I	0

***f(n)* = *h* (*n*) = straight-line distance heuristic**

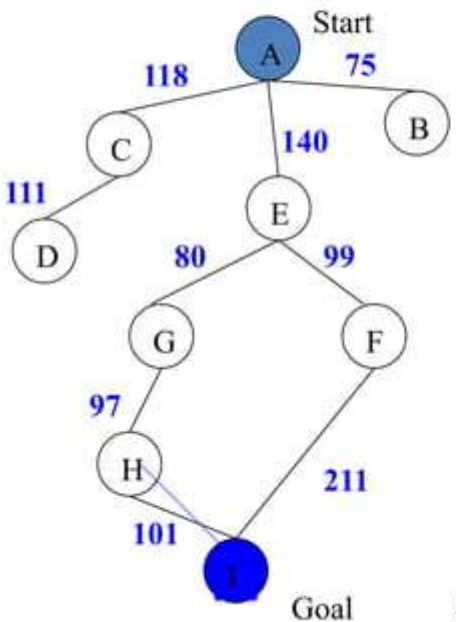
Greedy Best First Search



State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	**193**
H	98
I	0

***f(n) = h (n)* = straight-line distance heuristic**

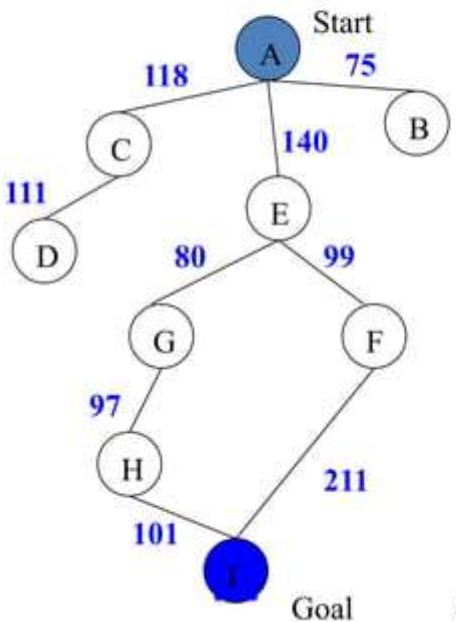
Greedy Best First Search



State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	**98**
I	0

***f(n)* = *h* (*n*) = straight-line distance heuristic**

Greedy Best First Search



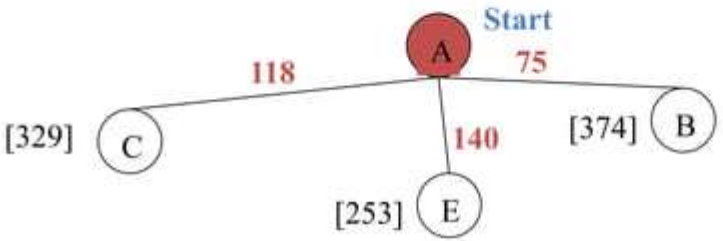
State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	**0**

***f(n)* = *h* (*n*) = straight-line distance heuristic**

Greedy Best First Search

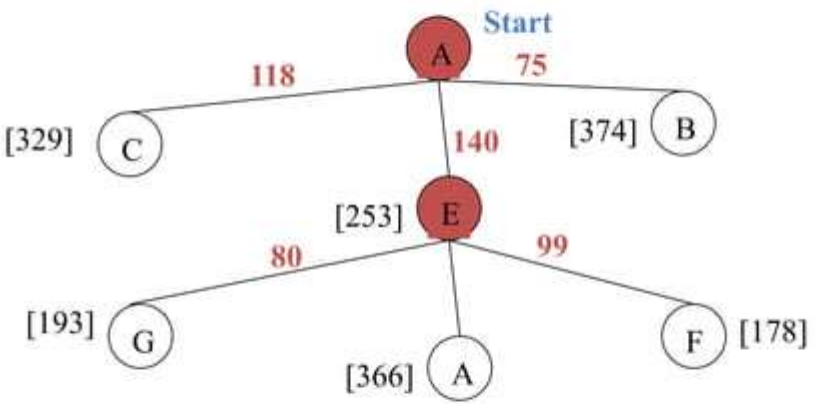
State	h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	**0**

Greedy Search: Tree Search



State	h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	**0**

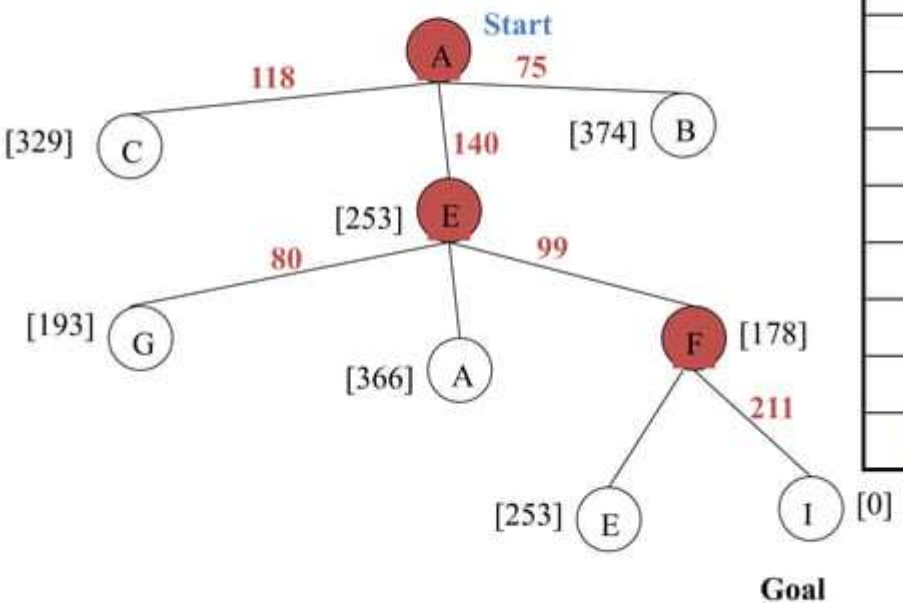
Greedy Search: Tree Search



State	h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	**0**

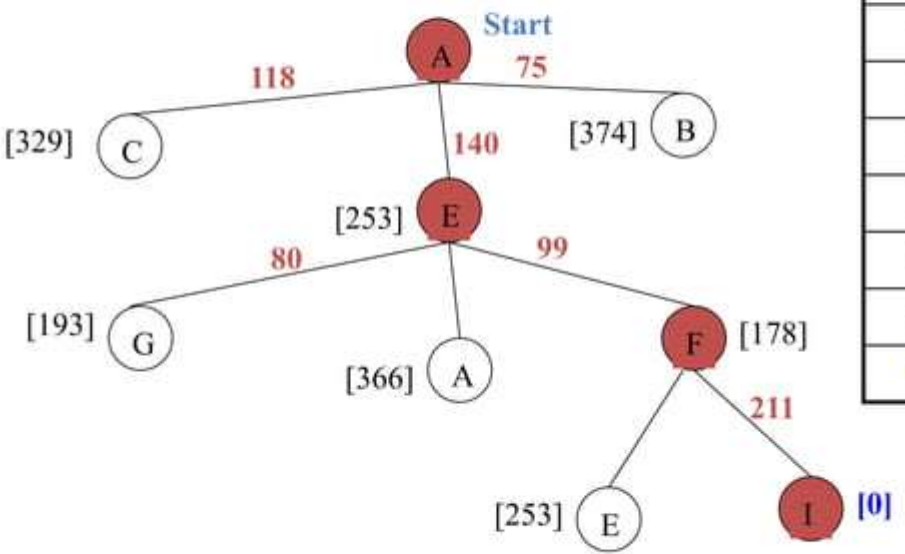
Greedy Search: Tree Search

State	h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	**0**



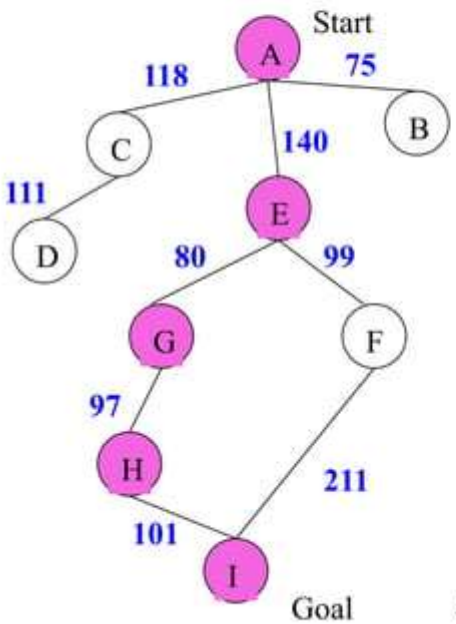
Greedy Search: Tree Search

State	h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	**0**



dist(A-E-F-I) = 140 + 99 + 211 = 450

Greedy Best First Search : Optimal ? NO

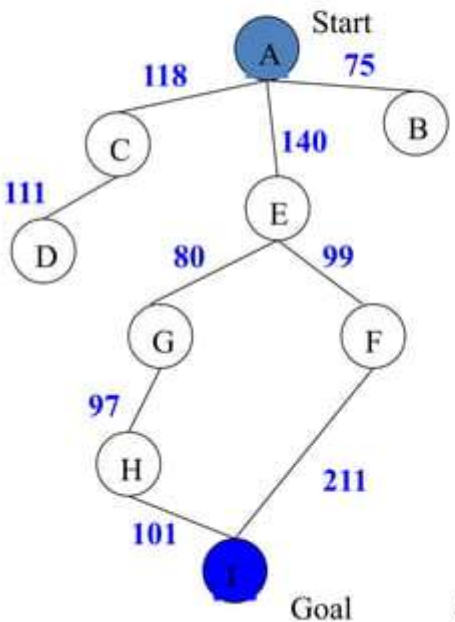


State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

***f(n)* = *h* (*n*) = straight-line distance heuristic**

dist(A-E-G-H-I)=140+80+97+101=418

Greedy Best First Search : Optimal ?



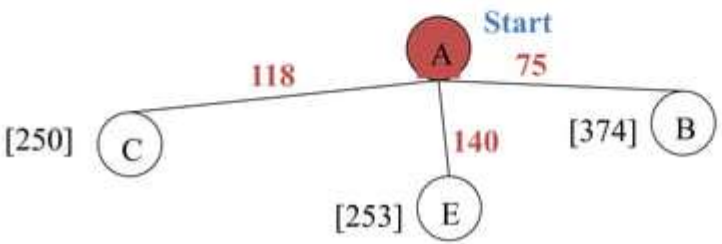
State	Heuristic: h(n)
A	366
B	374
** C	**250**
D	244
E	253
F	178
G	193
H	98
I	0

f(n) = h (n) = straight-line distance heuristic

Greedy Search: Tree Search

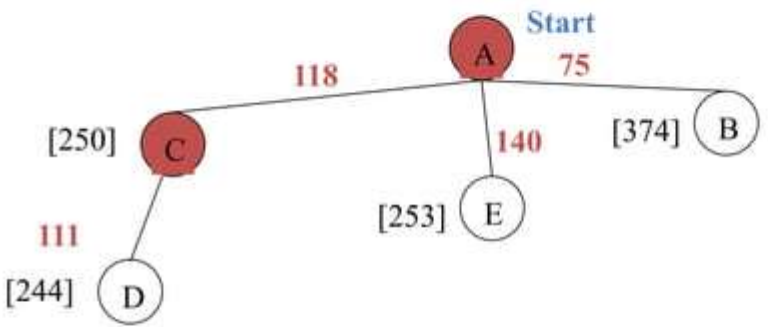
State	h(n)
A	366
B	374
C	250
D	244
E	253
F	178
G	193
H	98
I	0

Greedy Search: Tree Search



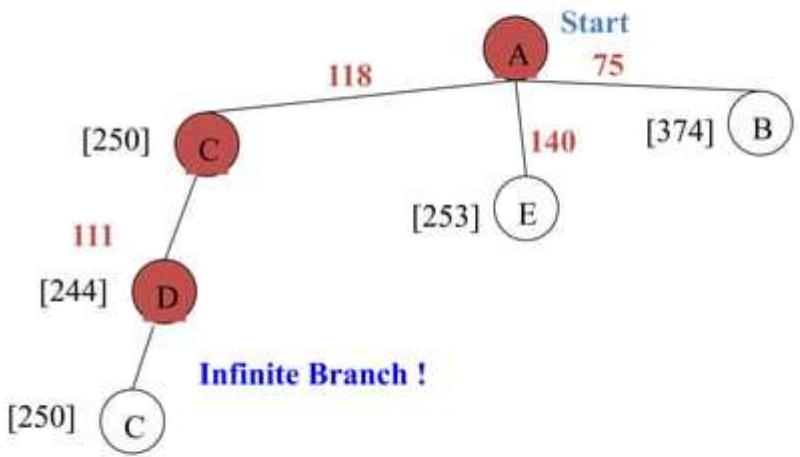
State	h(n)
A	366
B	374
C	250
D	244
E	253
F	178
G	193
H	98
I	**0**

Greedy Search: Tree Search



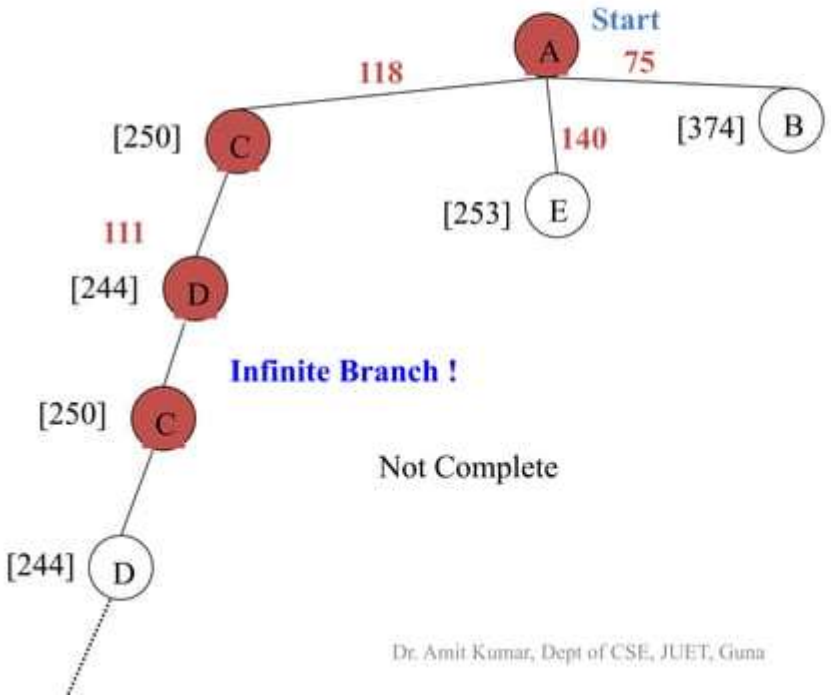
State	h(n)
A	366
B	374
C	250
D	244
E	253
F	178
G	193
H	98
I	0

Greedy Search: Tree Search



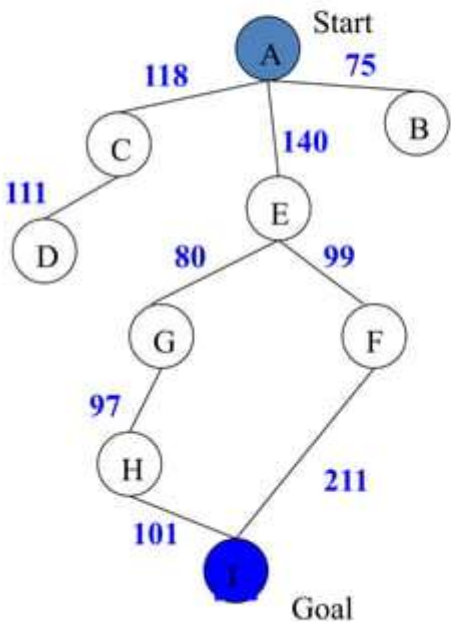
State	h(n)
A	366
B	374
C	250
D	244
E	253
F	178
G	193
H	98
I	**0**

Greedy Search: Tree Search



State	h(n)
A	366
B	374
C	250
D	244
E	253
F	178
G	193
H	98
I	**0**

Greedy Search: Time and Space Complexity ?



- Greedy search is not optimal.
- Greedy search is incomplete without systematic checking of repeated states.
- In the worst case, the Time and Space Complexity of Greedy Search are both $O(b^m)$

Where b is the branching factor and m the maximum path length

Informed Search Strategies

A* Search

eval-fn: f(n)=g(n)+h(n)

A* (A Star)

- Greedy Search minimizes a heuristic $h(n)$ which is an estimated cost from a node n to the goal state. However, although greedy search can considerably cut the search time (**efficient**), it is **neither optimal nor complete**.

- Uniform Cost Search minimizes the cost $g(n)$ from the initial state to n. UCS is **optimal and complete** but **not efficient**.

- **New Strategy**: Combine Greedy Search and UCS to get an efficient algorithm which is complete and optimal.

A* (A Star)

- A* uses a heuristic function which combines $g(n)$ and $h(n)$
  $f(n) = g(n) + h(n)$

- $\boldsymbol{g(n)}$ is the exact cost to reach node n from the initial state. Cost so far up to node n.

- $\boldsymbol{h(n)}$ is an estimation of the remaining cost to reach the goal.

Algorithm

1. Start with OPEN containing only the 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 the empty list.
2. Until a goal node is found, repeat the following procedure:
   - If there are no nodes on OPEN, report failure.
   - else, pick the node on OPEN with the lowest f' value. Call it BEST NODE. Remove it from OPEN. Place it on CLOSED.
     - See if BESTNODE is a goal node. If so, exit and report a solution (either BESTNODE ; if all we want is the node or the path that has been created between the initial state and BESTNODE if we are interested in the path).
     - else, generate the successors of BEST NODE but do not set BESTNODE to point to them yet. (First we need to see if any of them have already been generated.) For each such SUCCESSOR, do the following:

a) Set SUCCESSOR to point back to BEST NODE. These backwards links will make it possible to recover the path once a solution is found.

b) Compute g(SUCCESSOR) = g(BESTNODE) + the cost of getting from BEST NODE to SUCCESSOR.

c) See if SUCCESSOR is the same as any node on OPEN (i.e., it has already been generated but not processed).

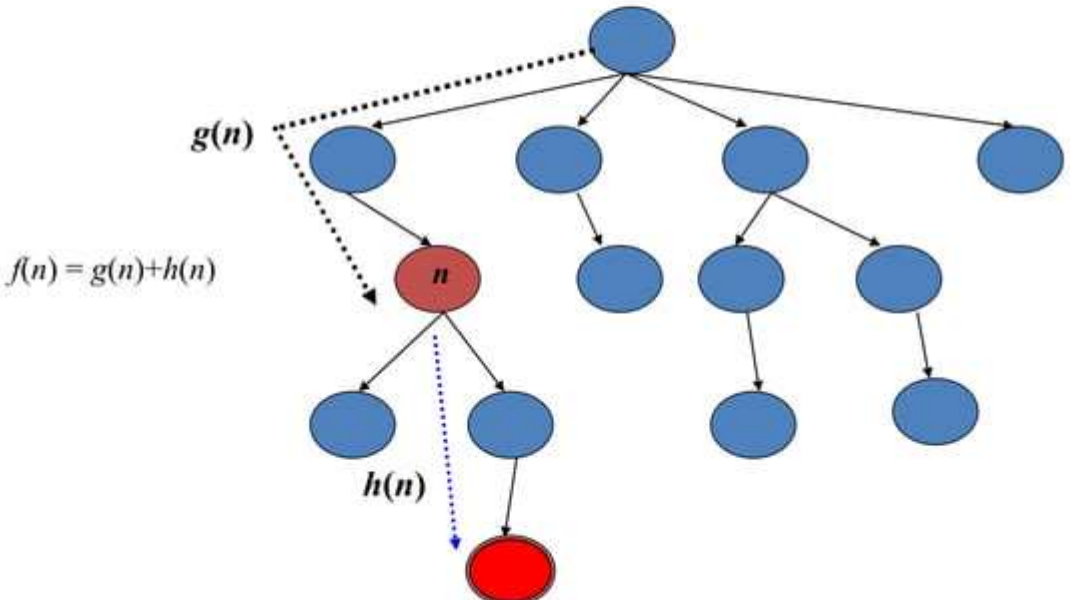
- If so, call that node OLD. Since this I node already exists in the graph, we can throw SUCCESSOR away and add OLD to the list of BESTNODE's successors. Now we must decide whether OLD's parent link should be reset to point to BESTNODE. It should be if the path we have just found to SUCCESSOR is cheaper than the current best path to OLD (since SUCCESSOR and OLD are really the same node). So see whether it is cheaper to get to OLD via its current parent or to SUCCESSOR via BESTNODE by comparing their g values. If OLD is cheaper (or just as cheap), then we need do nothing. If SUCCESSOR is cheaper, then reset OLD's parent link to point to BESTNODE, record the new cheaper path in g(OLD), and update f' (OLD).

d) If SUCCESSOR was not on OPEN, see if it is on CLOSED. If so, call the node on CLOSED OLD and add OLD to the list of BESTNODE's successors.

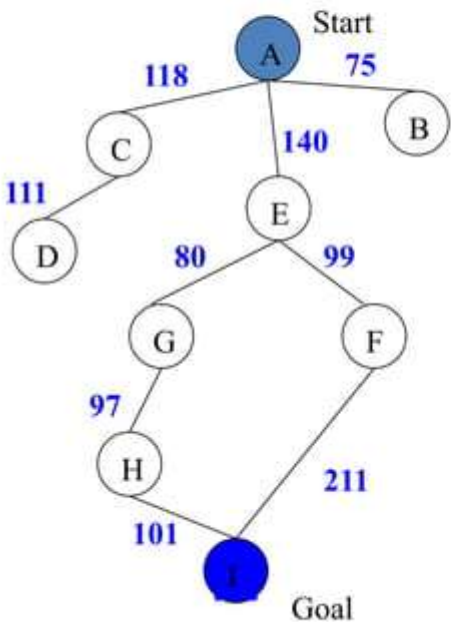
- Check to see if the new path or the old path is better just as in step 2(c), and set the parent link and g and f' values appropriately. If we have just found a better path to OLD, we must propagate the improvement to OLD's successors. This is a bit tricky. OLD points to its successors. Each successor in turn points to its successors, and so forth, until each branch terminates with a node that either is still on OPEN or has no successors. So to propagate the new cost downward, do a depth-first traversal of the tree starting at OLD, changing each node's g value (and thus also its f' value), terminating each branch when you reach either a node with no successors or a node to which an equivalent or better path has already been found.

e) If SUCCESSOR was not already on either OPEN or CLOSED, then put it on OPEN, and add it to the list of BESTNODE's successors. Compute f'(SUCCESSOR) = g(SUCCESSOR) + h'(SUCCESSOR)

A* (A Star)



A* Search



State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

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

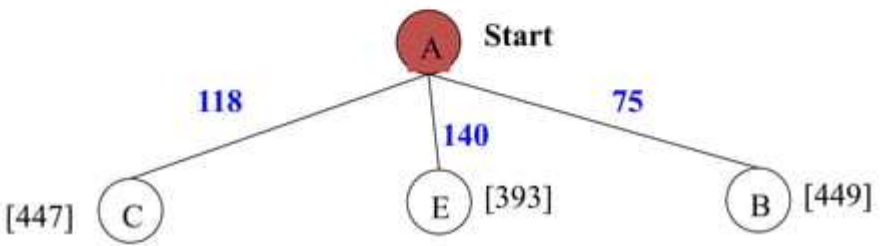
g(n): is the exact cost to reach node n from the initial state.

A* Search: Tree Search

State	h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

A* Search: Tree Search

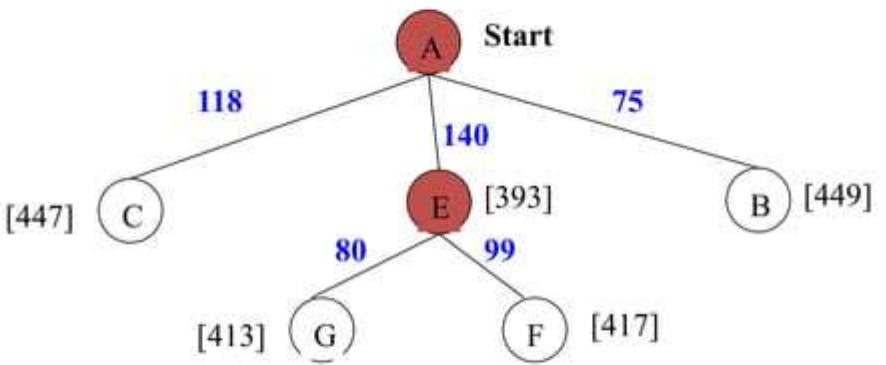
State	h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0



C[447] = 118 + 329
E[393] = 140 + 253
B[449] = 75 + 374

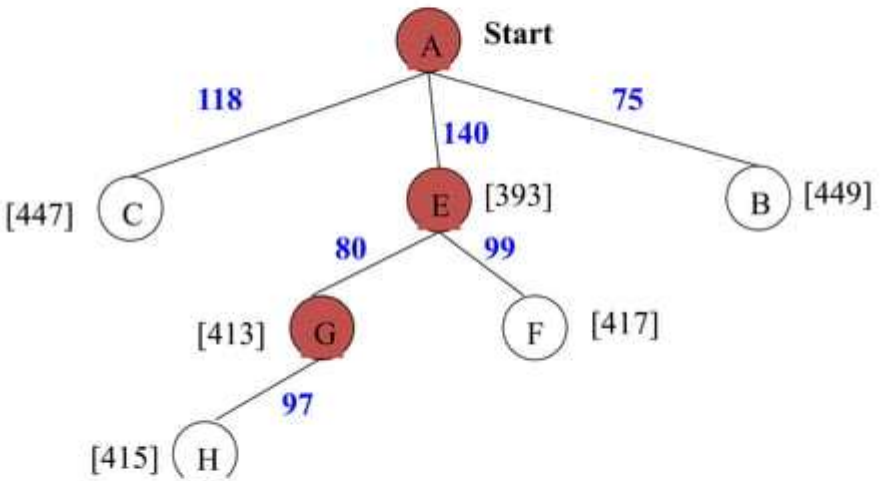
A* Search: Tree Search

State	h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0



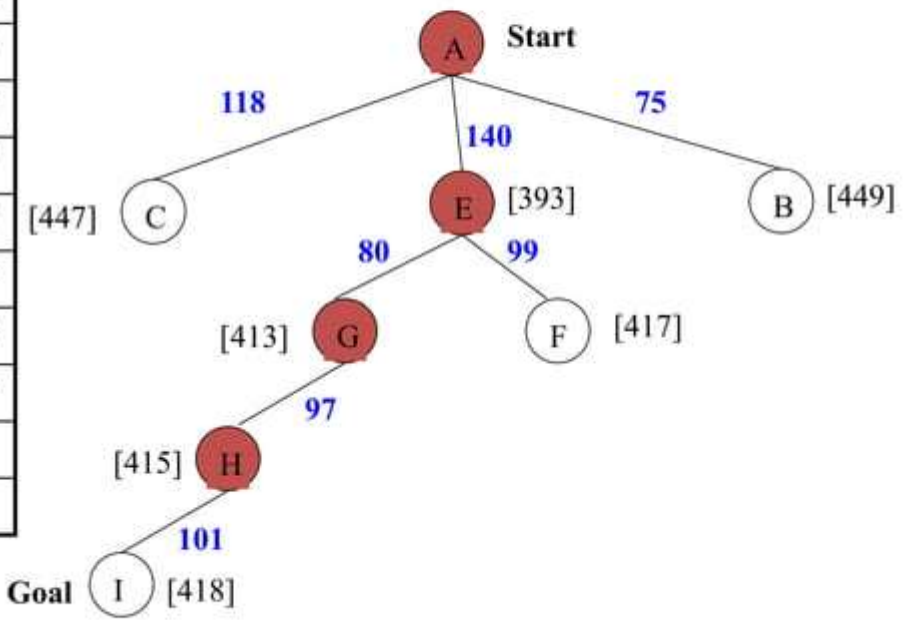
A* Search: Tree Search

State	h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0



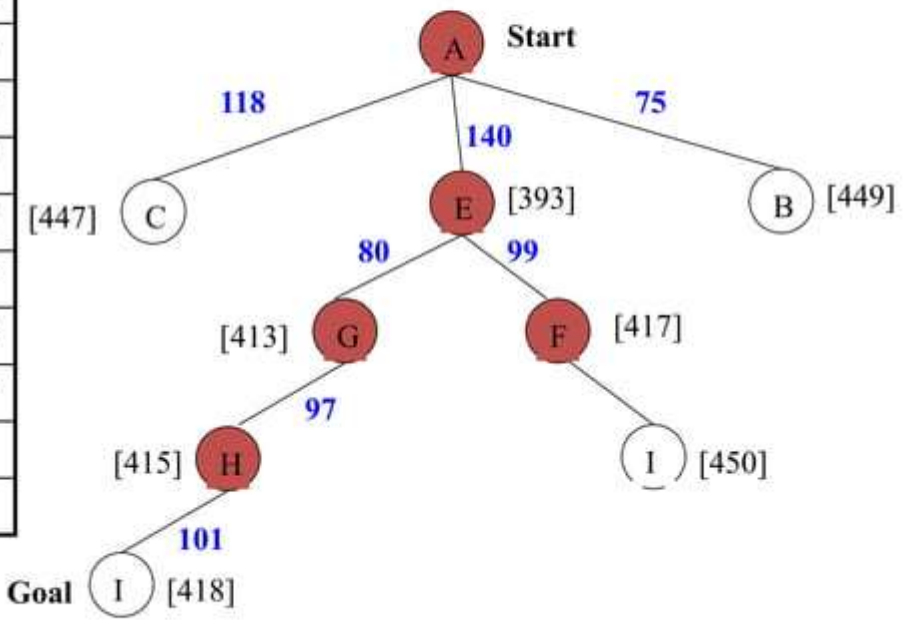
A* Search: Tree Search

State	h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0



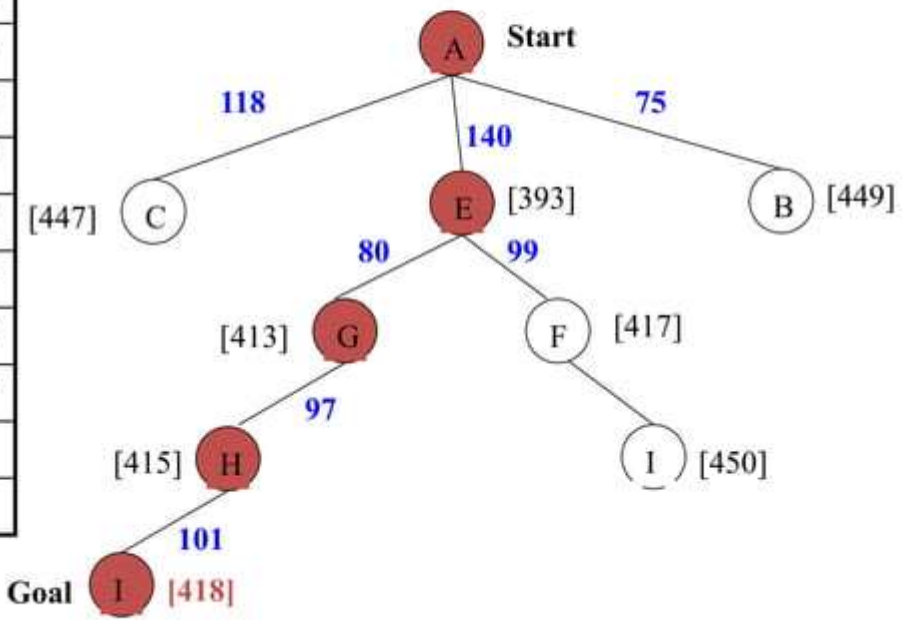
A* Search: Tree Search

State	h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0



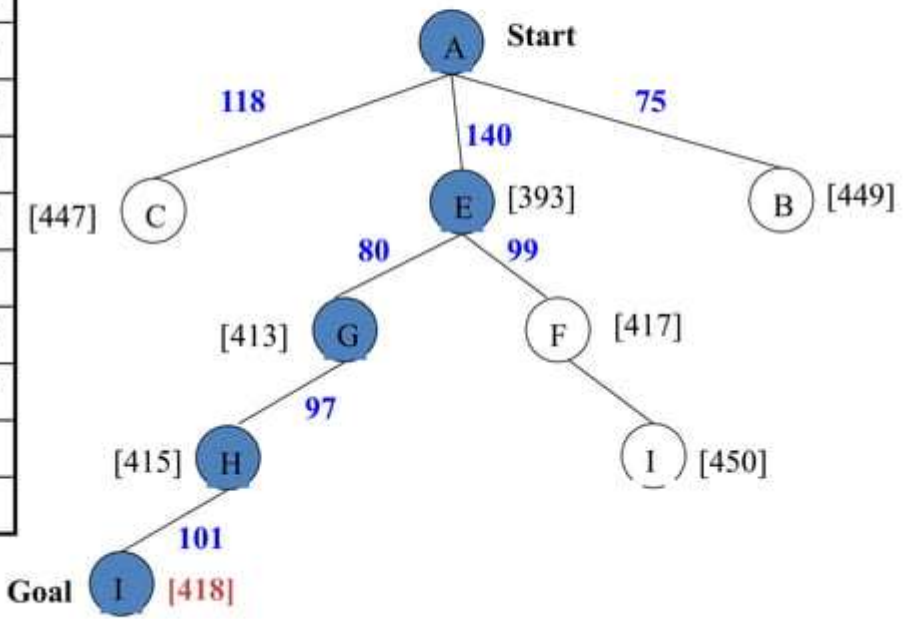
A* Search: Tree Search

State	h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0



A* Search: Tree Search

State	h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

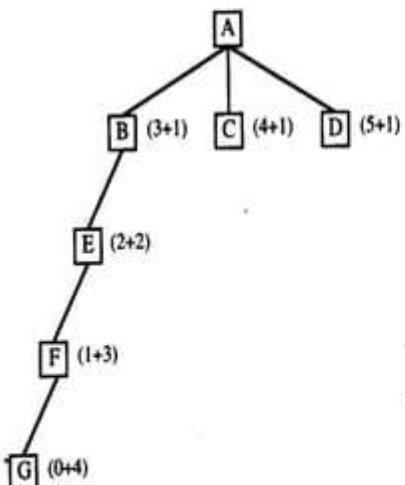


How good is A*?

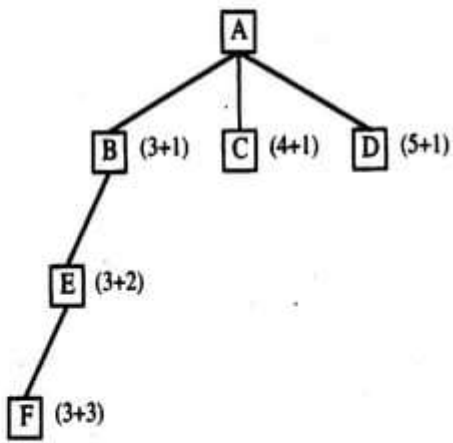
- Memory usage depends on the heuristic function
  - If g(N) = *constant*, H(n) = 0 then A* = breadth-first
  - If h(N) is a *perfect* estimator, A* goes straight to goal
  - ...but if h(N) were perfect, why search?
- Quality of solution also depends on h(N)
- It can be proved that, if h(N) is *optimistic* (never overestimates the distance to a goal), then A* will find a best solution (shortest path to a goal)

Estimation of h

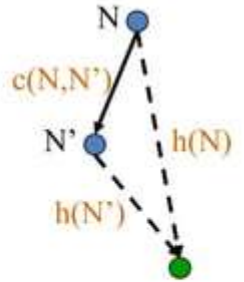
h' Overestimates h



h' Underestimates h



Conditions for optimality



- Admissibility
  - An admissible heuristic *never overestimates* the cost to reach the goal
  - Straight-line distance h_{SLD} obviously is an admissible heuristic

- A* is optimal if h is admissible or consistent

- Admissibility /Monotonicity
  - The admissible heuristic h is consistent (or satisfies the monotone restriction) if for every node N and every successor N' of N: $h(N) \leq c(N,N') + h(N')$ (triangular inequality)
  - A consistent heuristic is admissible.

Heuristic Evaluation

The Effect of Heuristic Accuracy on Performance

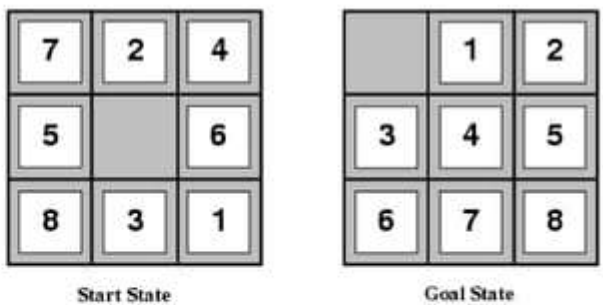
An Example: 8-puzzle

Admissible heuristics

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)



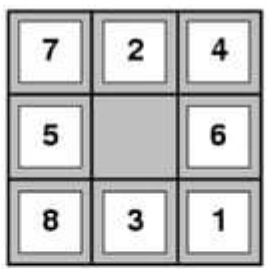
- $h_1(S) = ?$
- $h_2(S) = ?$

Admissible heuristics

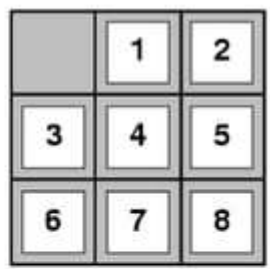
E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)



Start State



Goal State

- $h_1(S) = ?$ 8
- $h_2(S) = ?$ 3+1+2+2+2+3+3+2 = 18

Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible)
- then h_2 dominates h_1
- h_2 is better for search

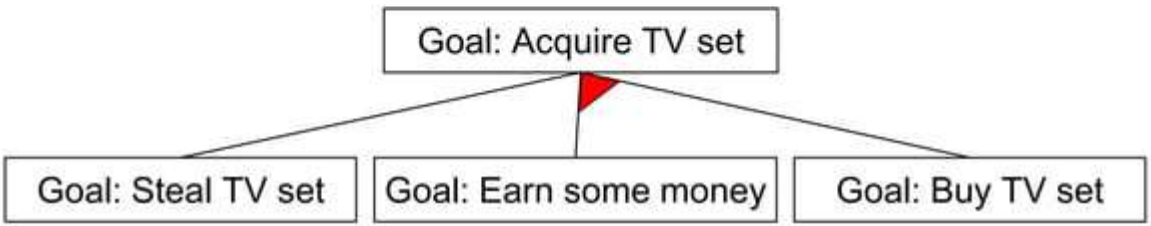
Why? Self Study

- Typical search costs (average number of nodes expanded):

- $A^*(h_1) = 227$ nodes
  $A^*(h_2) = 73$ nodes

- $A^*(h_1) = 39{,}135$ nodes
  $A^*(h_2) = 1{,}641$ nodes

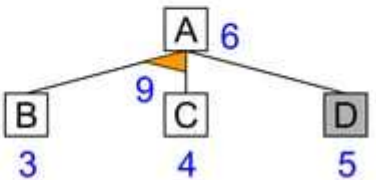
Problem Reduction

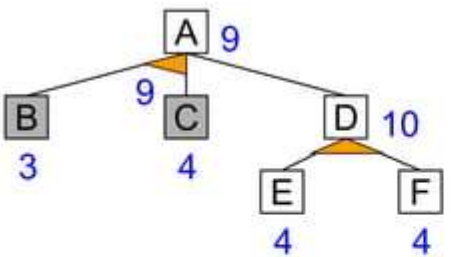


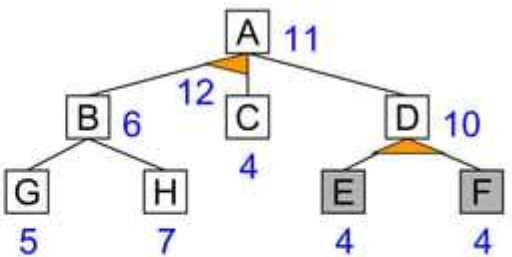
AND-OR Graphs

Algorithm AO* (Martelli & Montanari 1973, Nilsson 1980)

Problem Reduction: AO*







When to Use Search Techniques

- The search space is small, and
  - There are no other available techniques, or
  - It is not worth the effort to develop a more efficient technique

- The search space is large, and
  - There is no other available techniques, and
  - There exist "good" heuristics

Conclusions

- Frustration with *uninformed* search led to the idea of using domain specific knowledge in a search so that one can intelligently explore only the relevant part of the search space that has a good chance of containing the goal state. These new techniques are called informed (heuristic) search strategies.

- Even though heuristics improve the performance of informed search algorithms, they are still time consuming especially for large size instances.