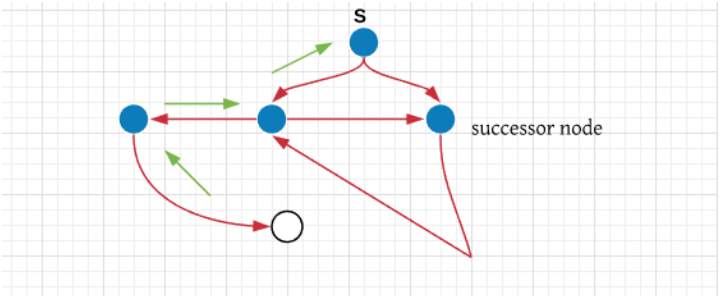
Graph Search

- It is a general abstraction for how to structure instances of specific algorithm that could search or build up and explicit portion of entire implicit search space as search graph and capture what would constitute solution from star node to goal node with some criteria of optimization and heuristic merit.

Procedure GRAPHSEARCH (Nils Nilsson)

1. Create a search graph, G, which consists only of the start node, s. s is placed on a list called OPEN.
2. Create a list called CLOSED. This list is initially empty.
3. LOOP: if Empty(OPEN), then exit with failure.
4. n <-- First(OPEN)
   OPEN = OPEN - {n}
   CLOSED = CLOSED + {n}
5. If Goal(n), exit successfully with the solution obtained by tracing a path along the pointers from n to s in G (pointers are established in step 7).
6. Expand node n, generating the set, M, of its successors and install them as successors of n in G
7. Establish a pointer to n from those members of M that were not already in G, i.e., on either CLOSED or OPEN. Add these members of M to OPEN.
   For each member of M that was already on OPEN or CLOSED, decide whether or not to redirect its pointer to n.
   For each member of M already on CLOSED, decide for each of its descendants in G whether or not to redirect its pointer.
8. Reorder the list OPEN, either according to an arbitrary scheme or according to heu[illegible]
9. End LOOP



state expansion or generat children of parent node

keep track of node visited and expanded.
Also, keep trak of node that basically are open and condidate for futher exploration

Green Arrows represent the way of recording the path from where are i am right now and go back to start node

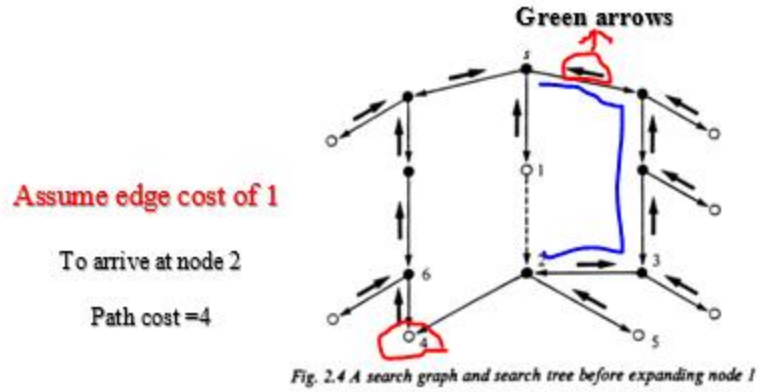


Fig. 2.4 A search graph and search tree before expanding node 1

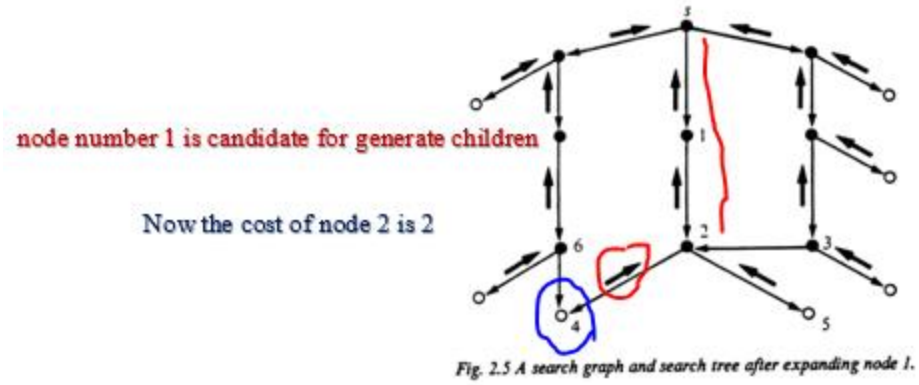
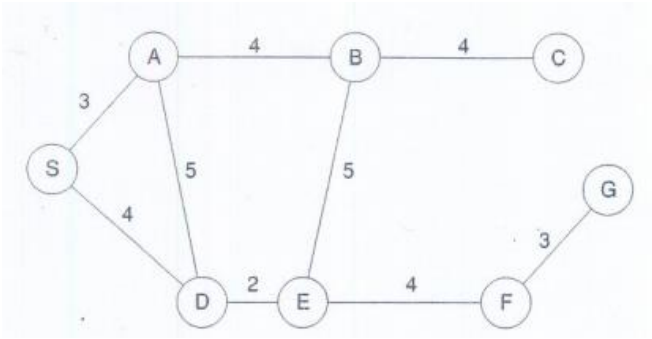


Fig. 2.5 A search graph and search tree after expanding node 1.

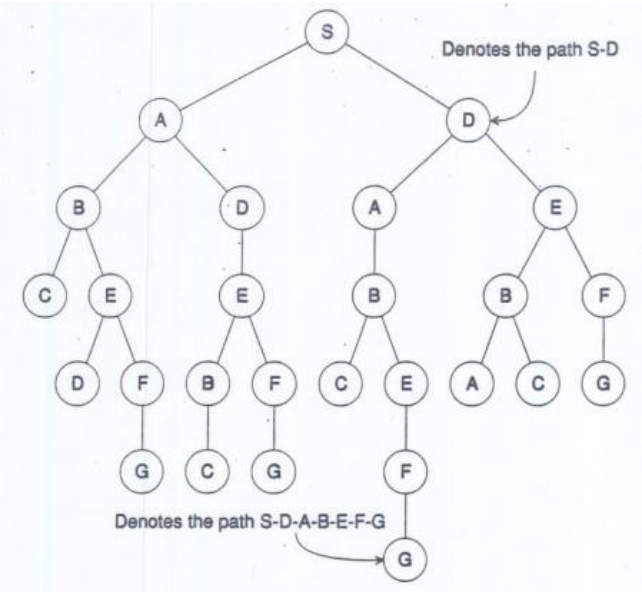
Nets and Basic Search

- Treat as network search

- Find any path between S and G



- Complete explicit search tree generated from that graph connected between the cities



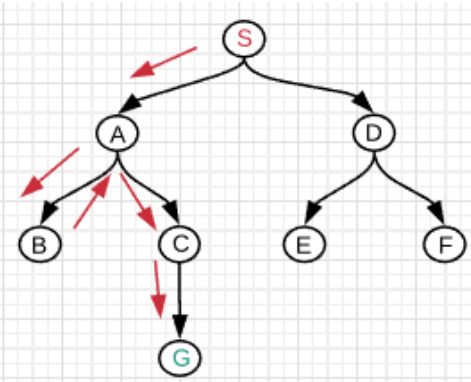
- Classical method called any path. not imposing optimization criteria

- Two classical algorithms
  - Depth first search (DFS)
  - Breath first search (BFS)

Pseudocode of depth first search (DFS)

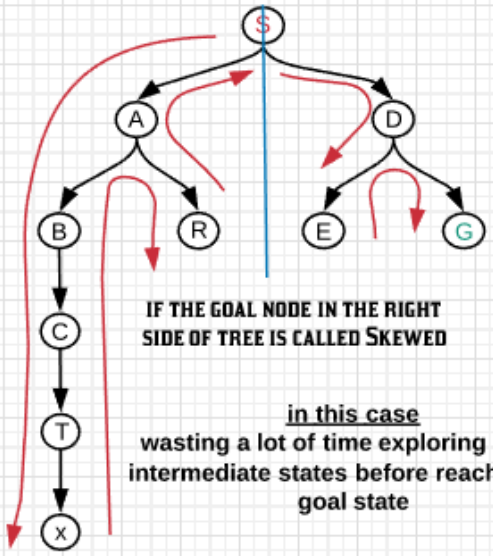
To conduct a depth-first search,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - Reject all new paths with loops.
  - Add the new paths, if any, to the *front* of the queue.
- If the goal node is found, announce success; otherwise, announce failure.



Do guarantee to find the solution?

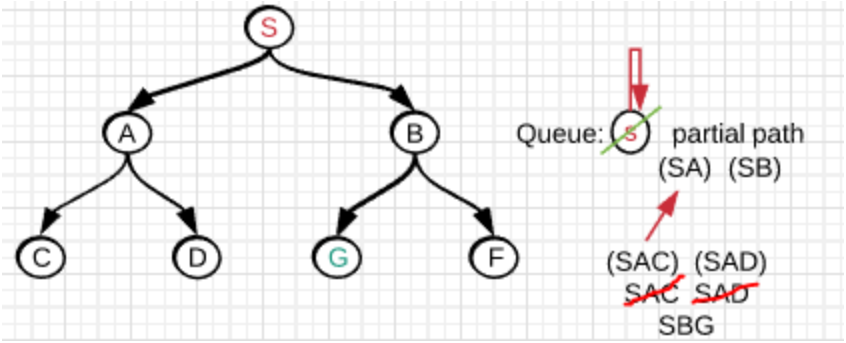
Yes



IF THE GOAL NODE IN THE RIGHT SIDE OF TREE IS CALLED SKEWED

in this case

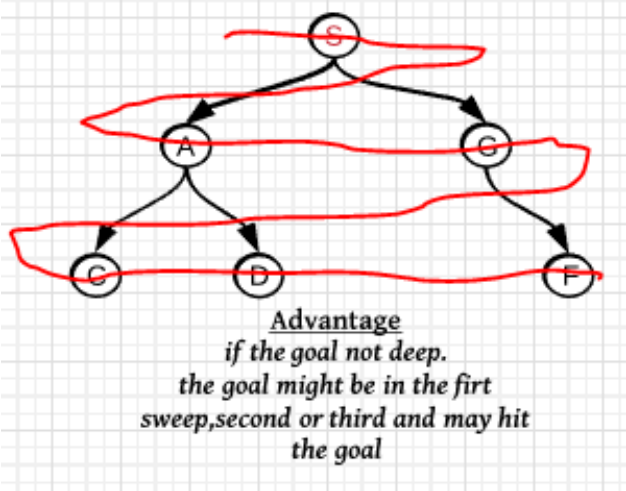
wasting a lot of time exploring a lot of intermediate states before reaching the goal state



Pseudocode of breath first search (DFS)

To conduct a breadth-first search,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - Reject all new paths with loops.
  - Add the new paths, if any, to the *back* of the queue.
- If the goal node is found, announce success; otherwise, announce failure.

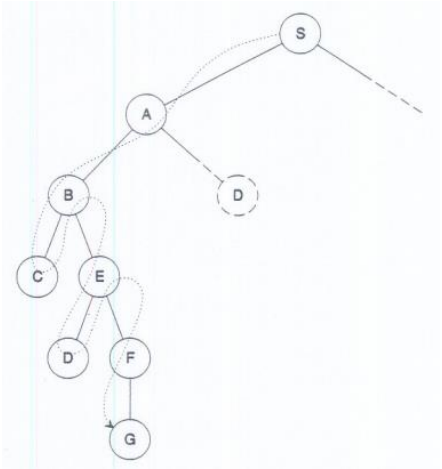


The different between DFS and BFS

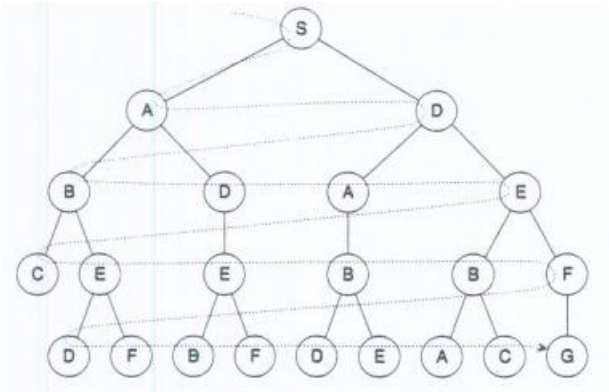
- *In the BFS, changing order of how attach partial path to the queue.*

Examples

- Depth first search (DFS)



- Breath first search (BFS)



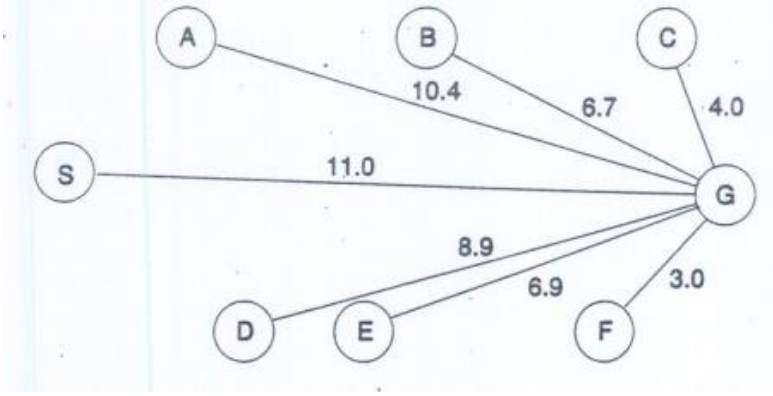
- It is possible to improve this by introducing heuristics
  - Hill climbing search

To conduct a hill-climbing search,

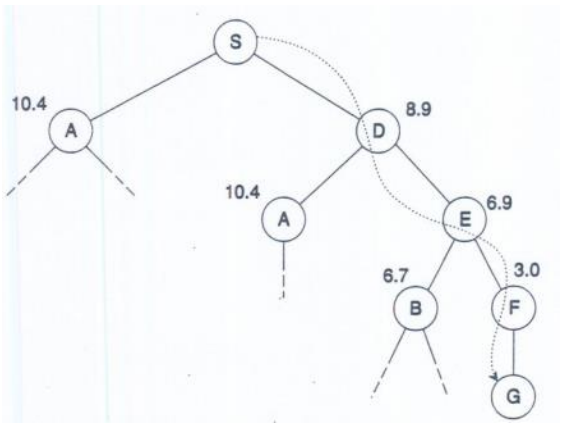
- ▷ Form a one-element queue consisting of a zero-length path that contains only the root node.
- ▷ Until the first path in the queue terminates at the goal node or the queue is empty,
  - ▷ Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - ▷ Reject all new paths with loops.
  - ▷ *Sort the new paths, if any, by the estimated distances between their terminal nodes and the goal.*
  - ▷ Add the new paths, if any, to the *front* of the queue.
- ▷ If the goal node is found, announce success; otherwise, announce failure.

Example

Having estimated distances between each city and the goal



- *Augment the search by modifying (DFS)*
- *One possible estimate is Straight line distance*



What is the advantage of taking straight line distance between two points?

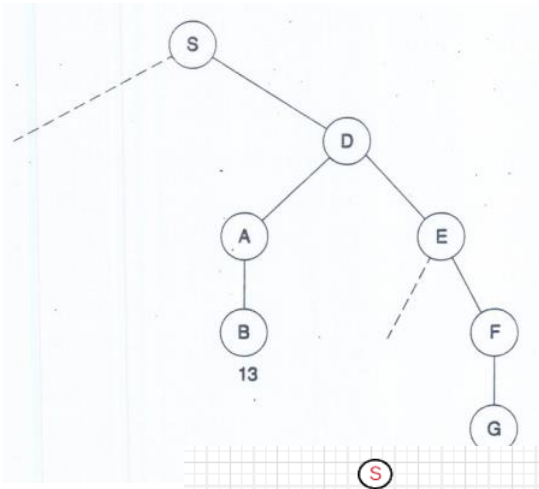
These gives brief introduction into any path procedures and there is systematic possibly that it could introduce some heuristic but do not make any claim about optimality of this path.

- *Done Any path*
- *optimal path (like to do)*
- *Ability to do game trace (it comes later)*

Optimal path

The description

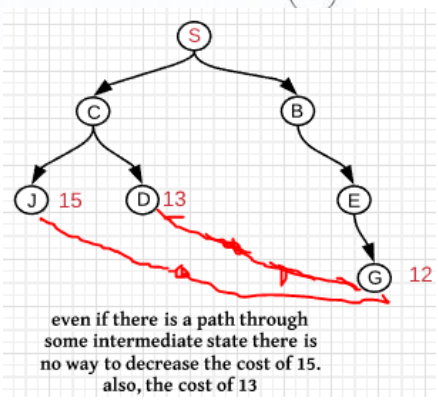
The length of the complete path from S to G, S-D-E-F-G is 13. Similarly, the length of the partial path S-D-A-B also is 13 and any additional movement along a branch will make it longer than 13. Accordingly, there is no need to purse S-D-A-B any further because any complete path starting with S-D-A-B has to be longer than a complete path already known. Only the other paths emerging from S and from S-D-E have to be considered, as they may provide a shorter path



Branch and bound search

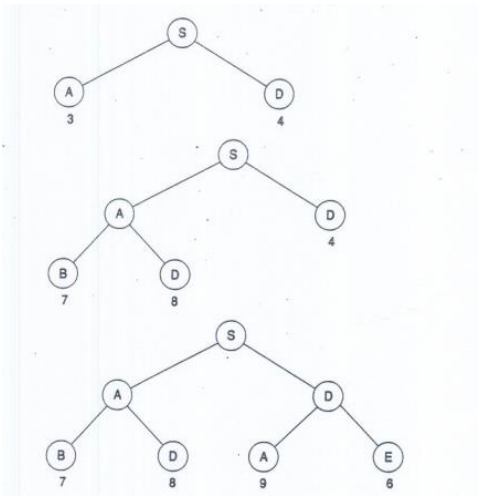
To conduct a branch-and-bound search,

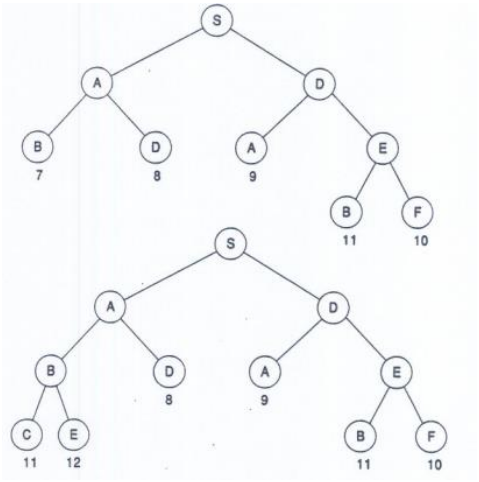
- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - Reject all new paths with loops.
  - *Add the remaining new paths, if any, to the queue.*
  - *Sort the entire queue by path length with least-cost paths in front.*
- If the goal node is found, announce success; otherwise, announce failure.

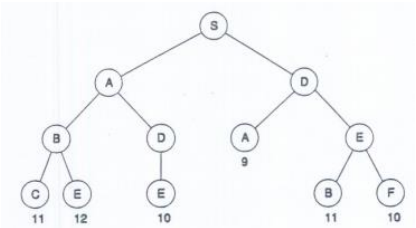


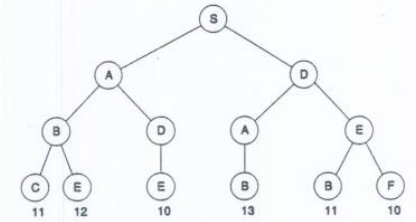
Point to this line. It is the Control strategy

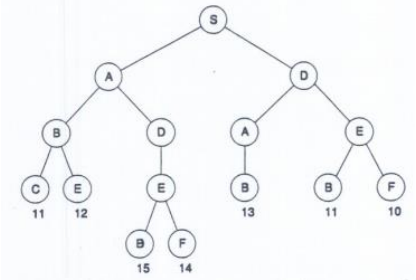
Entire solution

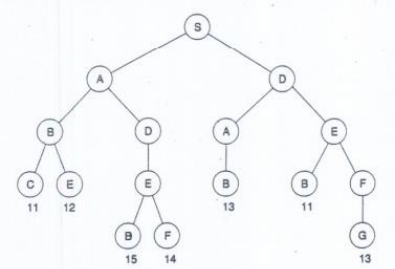


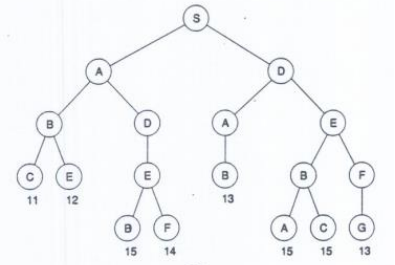


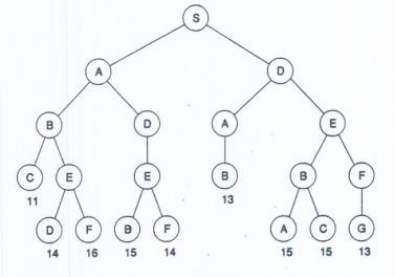






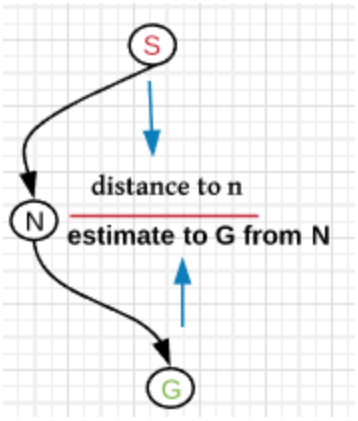






Two components

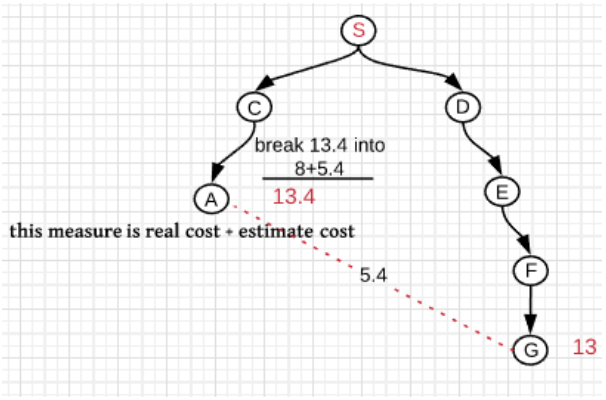
1. Distance to (n)
2. Estimate to (G) from (N)

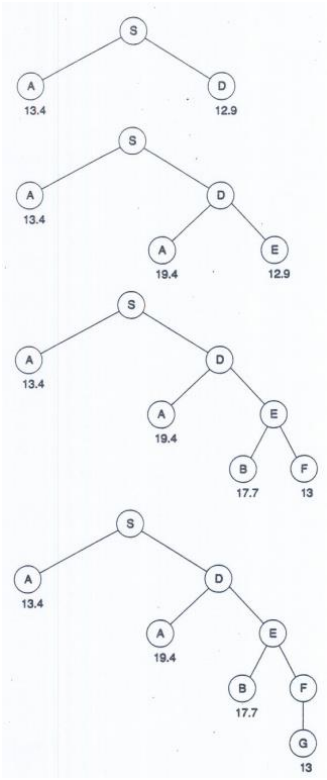


The description

Branch and bound search augmented by underestimates determines that the path S-D-E-F-G is optimal. The numbers beside the nodes are accumulated distances plus underestimates of distances remaining.

Underestimates quickly push up the lengths associated with bad paths. In this example expanded with branch and bound each operating without underestimates



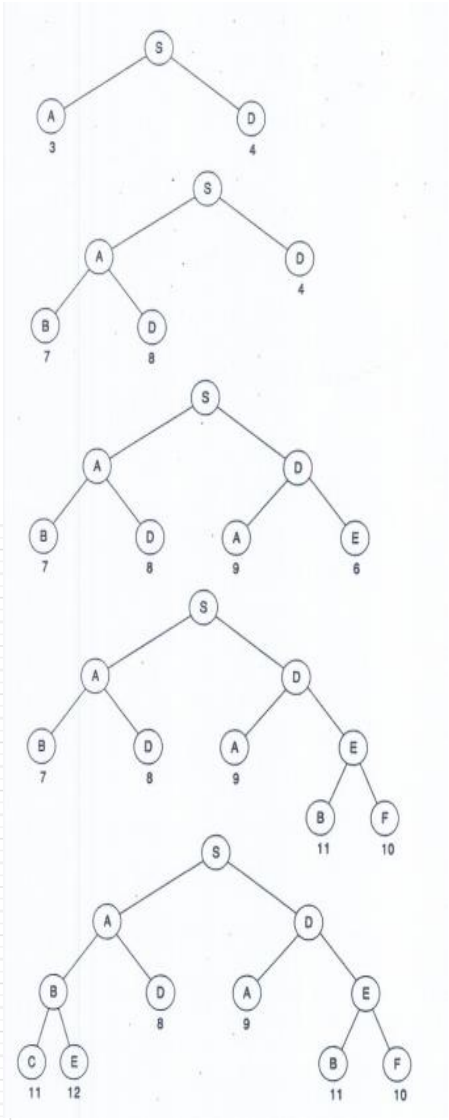


Nets and Basic Search

Branch and bound Search

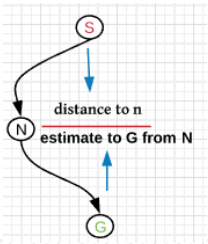
The description

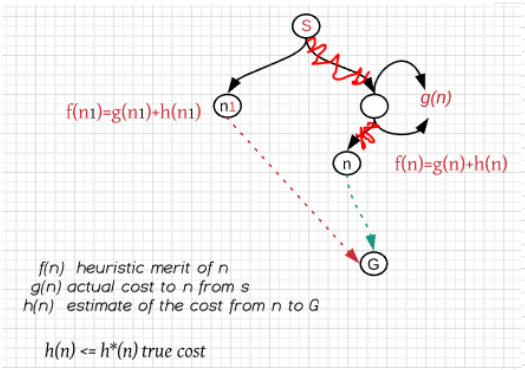
Branch and bound search determine that path S-D-E-F-G is optimal. The numbers beside the nodes are accumulated distances. Search steps when all partial paths to open nodes are as long as or longer than the complete path S-D-E-F-G

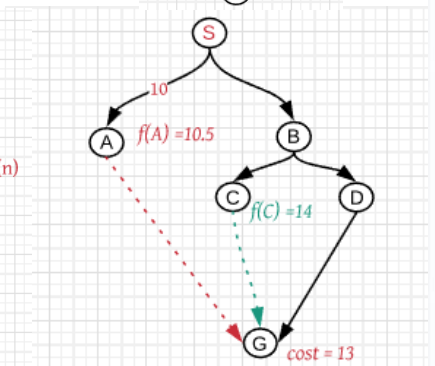


Think about the following

- *In branch and bound search, touch the idea of assigning a heuristic merit to a node that consists of two measures.*
  1. *Distance to n*
  2. *Estimate to G from N*







There are two open paths

node (A) ----> f(A)=10.5
node(C) ----> f(C)=14

it cannot expand node C in order to get node G becuse it cannot do better than 13

f(c) 8 + 6
6 is underestimate here

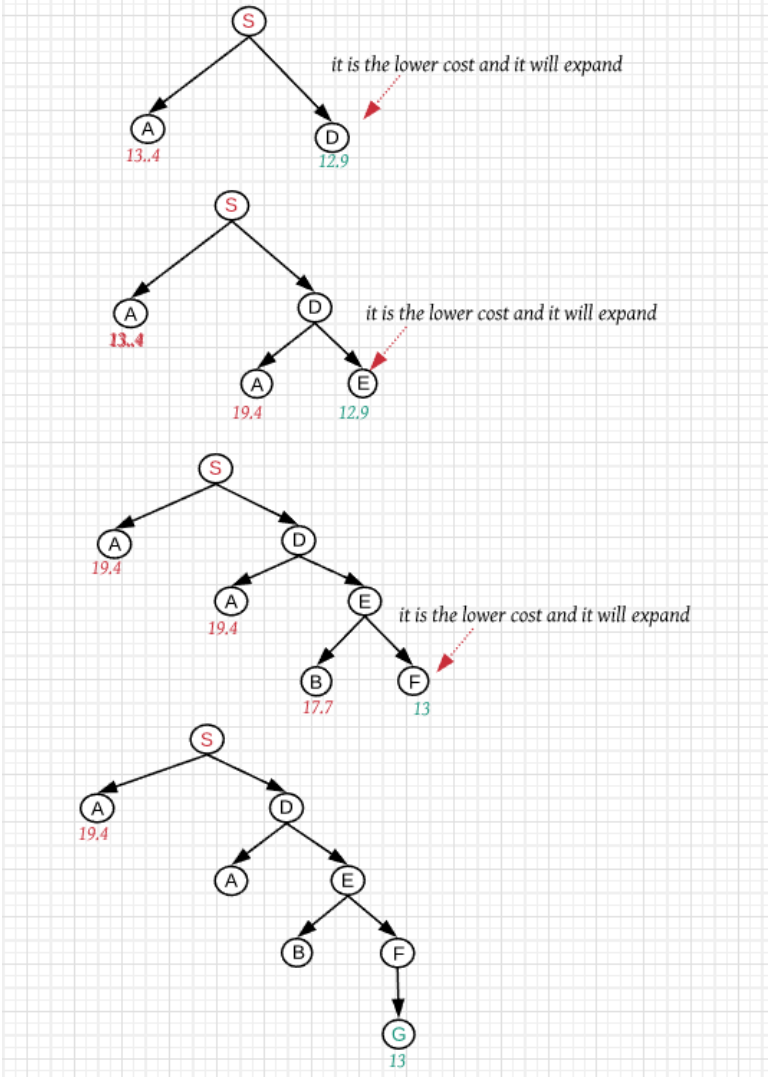
it can expand node A inoder to get node G because the cost of node A is (10.5) which is less than the cost of node G which is (13)

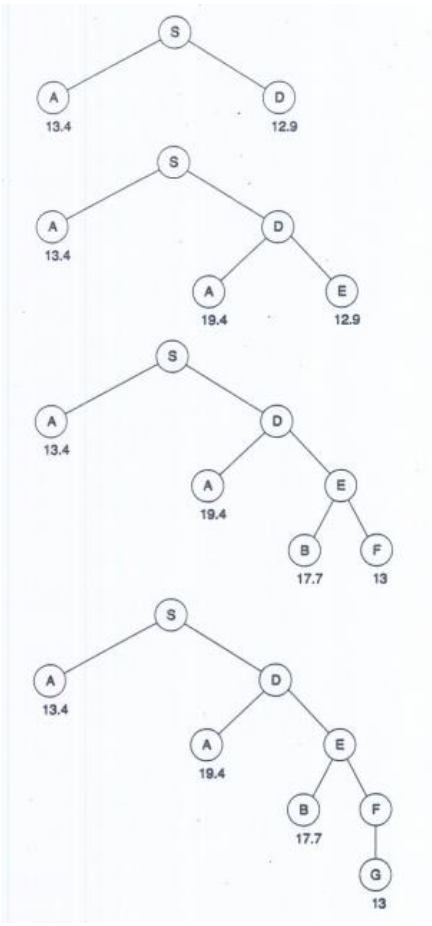
f(A) 10 + 0.5
0.5 is underestimate here and it can do better than 13 . can not stop becuse still have chance to beat 13

Branch and bound search algorithm with lower bound

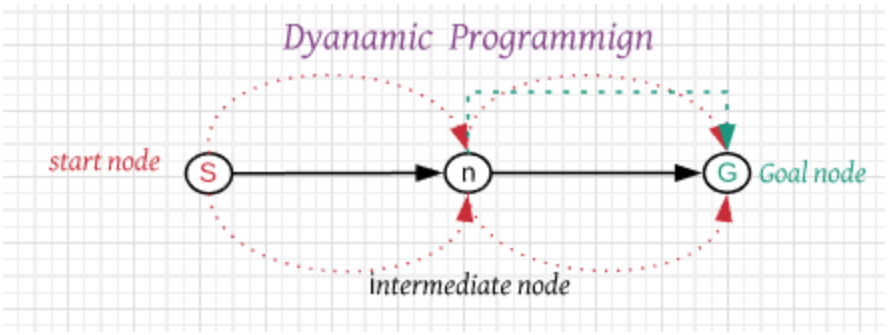
To conduct a branch-and-bound search with a lower-bound estimate,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - Reject all new paths with loops.
  - Add the remaining new paths, if any, to the queue.
  - Sort the entire queue by *the sum of the path length and a lower-bound estimate of the cost remaining*, with least-cost paths in front.
- If the goal node is found, announce success; otherwise, announce failure.





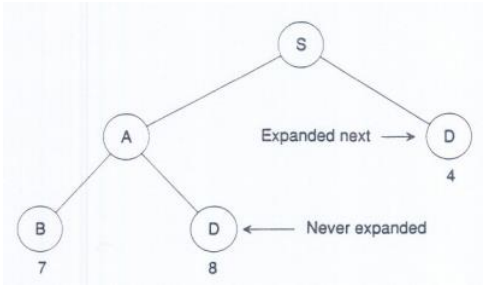
Dynamic Programming



- The solid line is the optimal solution
- Optimal path from S to G
- Instance if there is intermediate node in that path would you agree with me the segment S to N and the segment from N to G are also optimal segment
- Anything other than these two segments or dash lines not optimal solution
- There would be two paths going from start node to goal node that go through intermediate node and select one that gives the best cost and eliminate anything redundant.

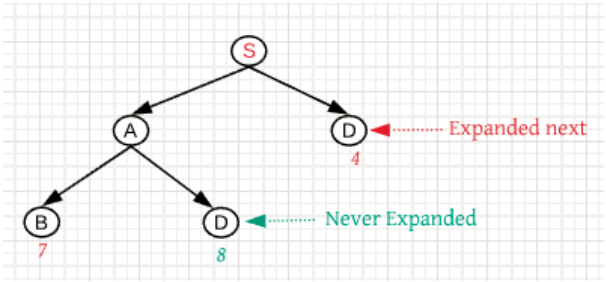
To illustrate this principle

The description



an illustration of the dynamic programming principle. The numbers beside the nodes are accumulated distances. There is no point in expanding the instance of node D at the end of S-A-D because getting to the goal via the instance of D at the end of S-D is obviously more efficient

- *There are two paths to arrive to D*
  - arrive at D directly with cost 4
  - arrived at D by through intermediate node A with cost 8



why would want to go through intermediate node A to node D with cost 8 if I have the guarantee cost of 4 correct

the key here is eliminating from search process any path that redundant node then decrease the size of search space that traverse in order to find the solution.

- Eliminate anything redundant but having some mechanisms to check for that redundancy. there is cost associate with checking logically for the benefit of not exploring so many things that is kind of tradeoff

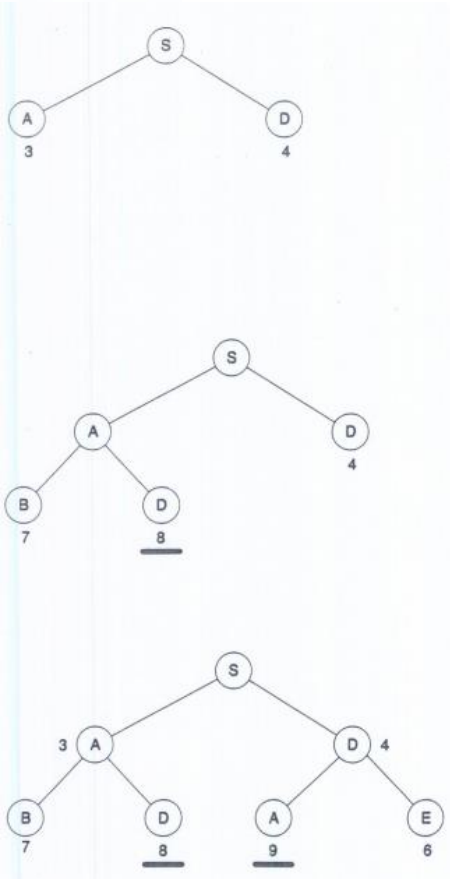
The **dynamic-programming principle:**

- The best way *through* a particular, intermediate place is the best way *to it* from the starting place, followed by the best way *from it* to the goal. There is no need to look at any other paths to or from the intermediate place.

The branch-and-bound procedure, with dynamic programming included, is as follows:

To conduct a branch-and-bound search with dynamic programming,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - Reject all new paths with loops.
  - Add the remaining new paths, if any, to the queue.
  - *If two or more paths reach a common node, delete all those paths except the one that reaches the common node with the minimum cost.*
  - Sort the entire queue by path length with least-cost paths in front.
- If the goal node is found, announce success; otherwise, announce failure.

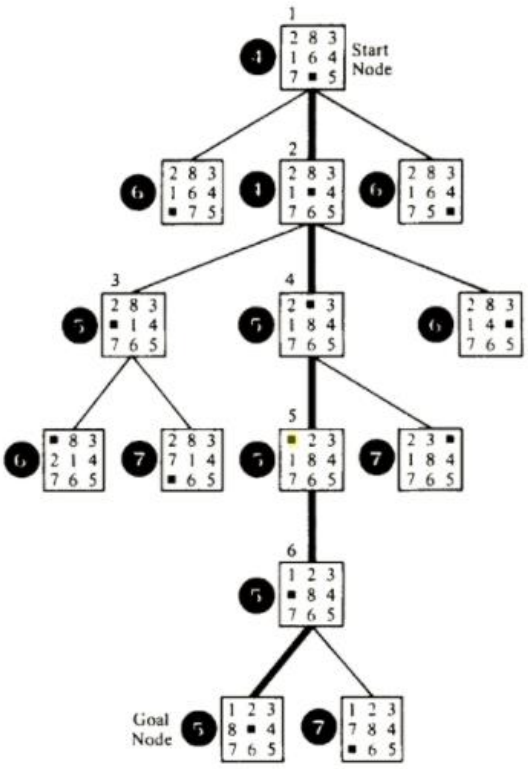


A* (A star) search algorithm

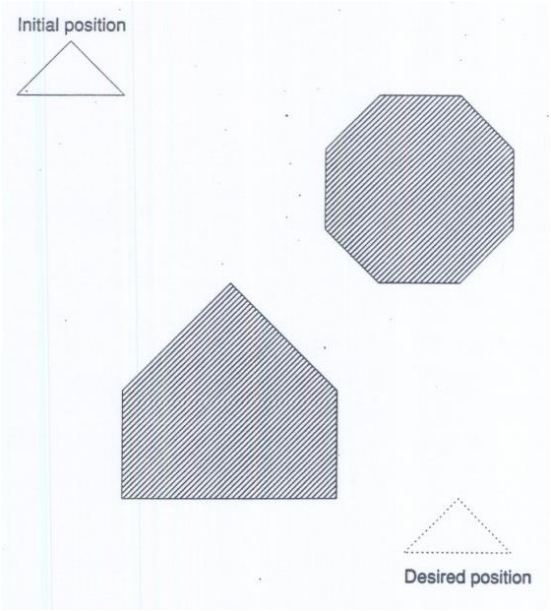
It is one of the most powerful mechanism of search process or graph search.

To conduct A* search,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - Reject all new paths with loops.
  - If two or more paths reach a common node, delete all those paths except the one that reaches the common node with the minimum cost.
  - Sort the entire queue by the sum of the path length and a lower-bound estimate of the cost remaining, with least-cost paths in front.
- If the goal node is found, announce success; otherwise, announce failure.



Classical robot problem or path



The description

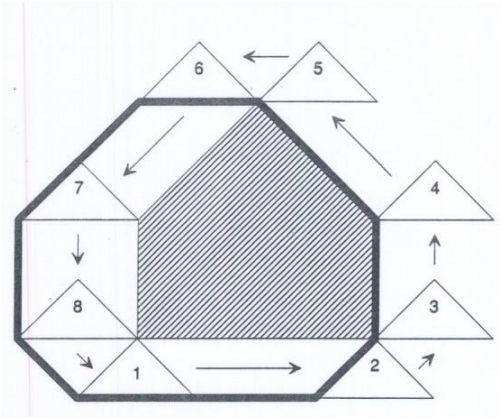
An obstacle avoidance problem. The problem is to move the small triangular robot to new position shown dotted without bumping into the pentagon or octagon

- *It happens in different level*
- *Robot hand*
- *Pick and place*
- *Two fundamental goals*

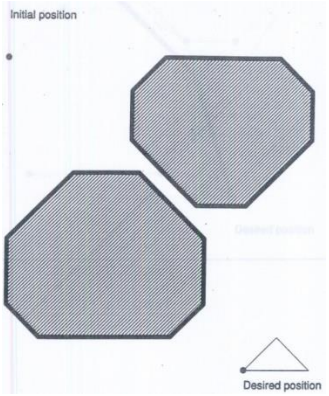
Classical approach

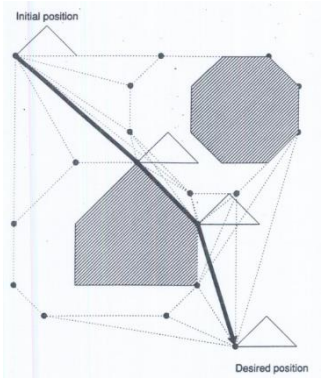
Configuration space transfer the geometry to create boundaries.

The description



The configuration space transformation. the heavy line shows the locus of the small tringle's lower left corner as the small triangle is moved around the big one. Numbered positions are the starting points for each straight-line run. Keeping the lower left corner away from the have line keeps the small tringle away from the pentagon

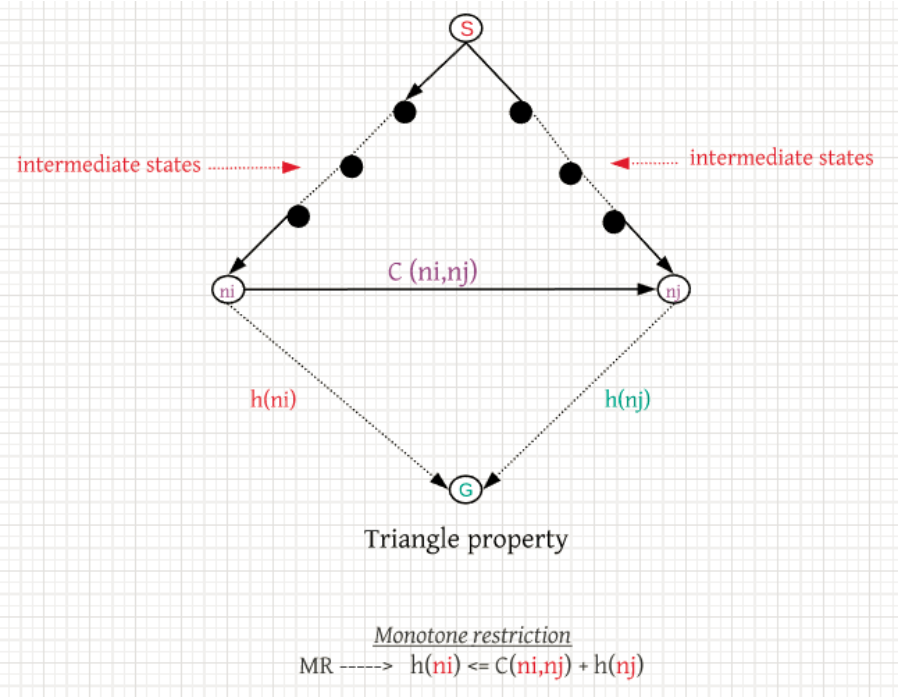




A (A star) properties of the " h" function*

1) *An underestimate of the real cost from n to goal state*
2) *Monotone restriction (MR)*

Imagine having this situation



"Next lecture"

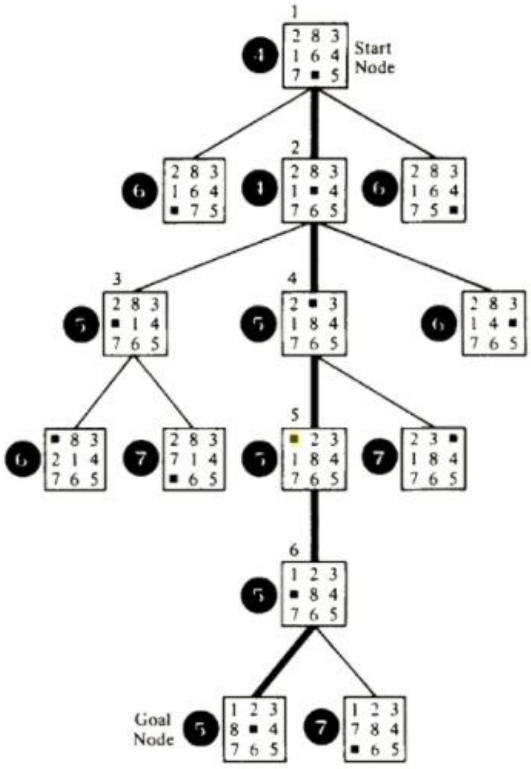
- Solving the 8-puzzle problem just an example
- Given three small thermoses to demonstrate if these properties met.
- A* is the excellent algorithm that allows to guarantee the optimality of solution and able to most efficiently of selecting that node for expansion that node end up with optimal path

Nets and Basic Search

A* (A star) search algorithm

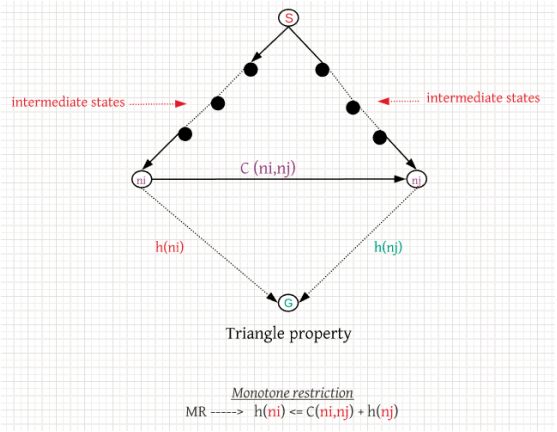
To conduct A* search,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - Reject all new paths with loops.
  - If two or more paths reach a common node, delete all those paths except the one that reaches the common node with the minimum cost.
  - Sort the entire queue by the sum of the path length and a lower-bound estimate of the cost remaining, with least-cost paths in front.
- If the goal node is found, announce success; otherwise, announce failure.



A (A star) properties of the" h" function*

3) *An underestimate of the real cost from n to goal state*
4) *Monotone restriction (MR)*



Continue with A*

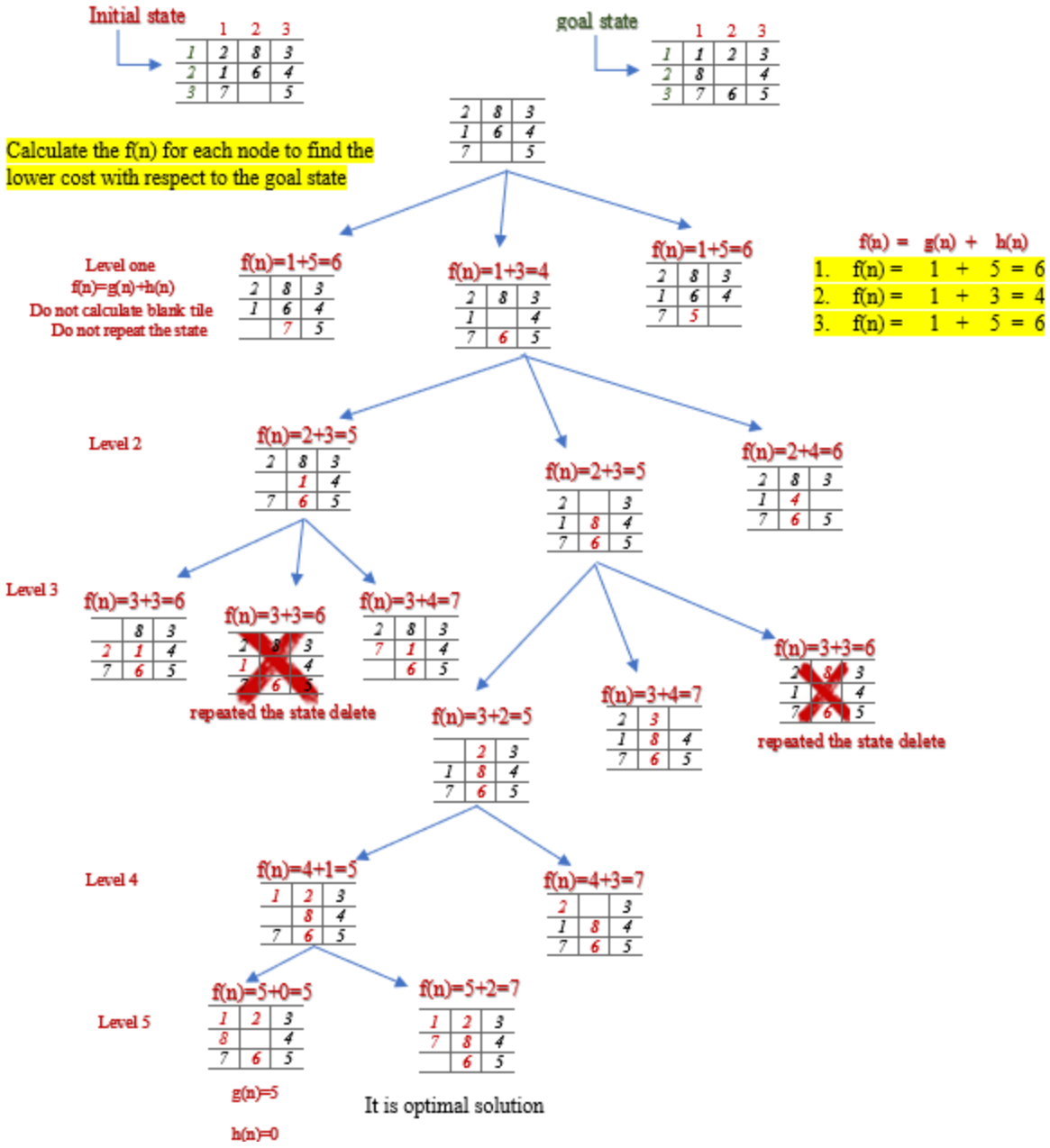
The 8-puzzle problem example

	1	2	3
1			
2			
3			

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

g(n) → level of the search tree

h(n) → number of misplace with respect to the goal state description



Using this heuristic

The 8-puzzle problem example

1. f(n)= g(n)+h(n)
   g(n) → level of the search tree
   h(n) → number of misplace with respect to the goal state description

2. Manhattan distance
   - Go along row and column to compute how far
   - Manhattan distance is more efficient