Module 2

Informed search, A* algorithm, Heuristic functions – Inventing Heuristic functions - Heuristic for constraint satisfaction problem – Iterative deepening – Hill climbing – Simulated Annealing.

Heuristic search

In order to solve many hard problems efficiently, it is necessary to construct a control structure that is no longer guaranteed to find the best answer but that will almost always find a very good answer. Here comes the idea of a heuristic. A heuristic is a technique that improves the efficiency of a search process.

Using good heuristics, we can hope to get good solutions to hard problems in less exponential time. A heuristic is a strategy for selectively searching a problem space. It guides our search along lines that have a high probability of success while avoiding wasted or apparently stupid efforts. Human beings use a large number of heuristics in problem solving. If you ask a doctor what could cause nausea and stomach pains, he might say it is "probably either stomach flu or food poisoning". Heuristics are not fool proof. Even the best game strategy can be defeated, diagnostic tools developed by expert physicians sometimes fail; experienced mathematicians sometimes fail to prove a difficult theorem.

State space search gives us a means of formalizing the problem solving process, and heuristics allow us to infuse that formalism with intelligence.

There are two major ways in which domain specific, heuristic knowledge can be incorporated in to a rule based search procedure.

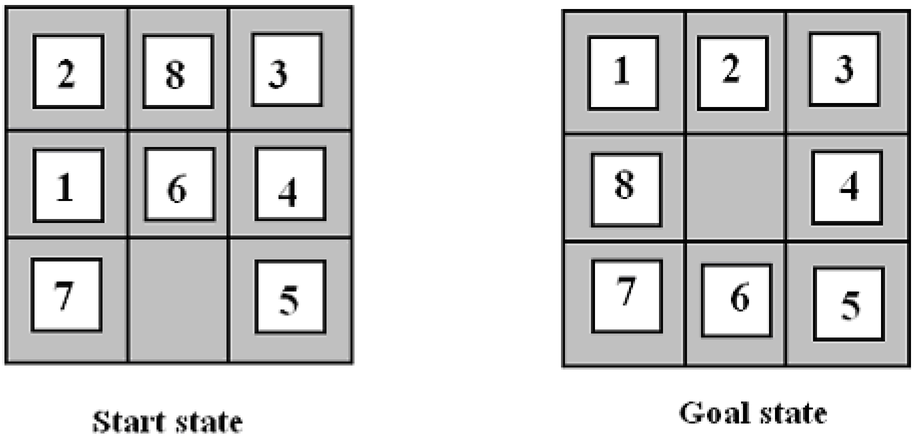
1. In the rules themselves. For example the rules for a chess playing system might describe not simply the set of legal moves but rather a set of sensible moves.
2. As a heuristic function that evaluates individual problem states and determines how desirable they are.

A heuristic function is a function that maps from problem state descriptions to measures of desirability, usually represented as numbers.

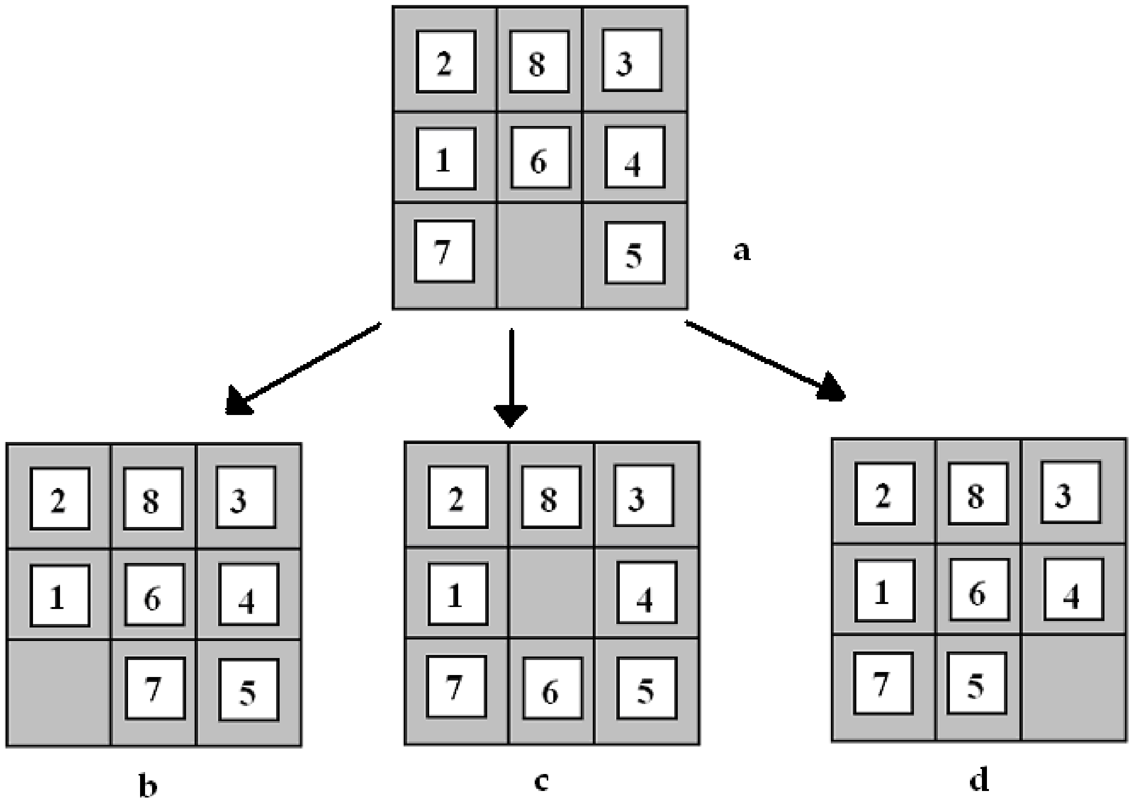
The following gives an example for evaluating a state with a heuristic function.

Eg. Consider the 8-puzzle problem.

The object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration.



The above figure shows the start state and goal state for the 8-puzzle.



In the above diagram, the start state and the first set of moves are shown.

Consider a heuristic function for evaluating each of the states. The number of tiles that are out of place in each state when it is compared with the goal state.

Consider the states b, c and d in the above diagram.

Compare the state b with the goal state. We will get the heuristic function value for b as 5.

In the same way, compare c and d with the goal state.

The heuristic function value for c is 3. The heuristic function value for d is 5.

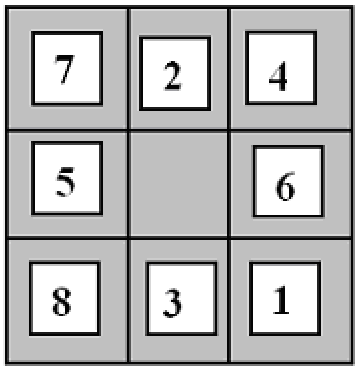
This example demonstrates the use of a heuristic function to evaluate the states. Which aspects of the problem state are considered, how those aspects are evaluated, and the weights given to individual aspects are chosen in such a way that the value of the heuristic function at a given node in the search process gives as good an estimate as possible of whether that node is on the desired path to a solution.

Well designed heuristic functions can play an important part efficiently guiding a search process towards a solution. Sometimes very simple heuristic functions can provide a fairly good estimate of whether a path is any good or not. In other situations, more complex heuristic functions should be employed.

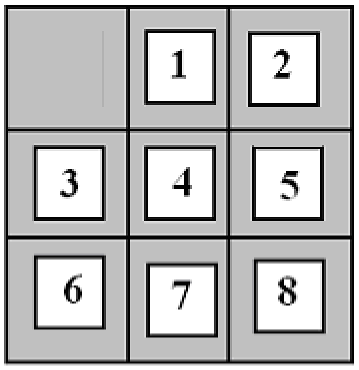
The purpose of a heuristic function is to guide the search process in the most profitable direction by suggesting which path to follow first when more than one is available. The more accurately the heuristic function estimates the true merits of each node in the search graph, the more direct the solution process.

Let us consider heuristic functions in detail

The 8-puzzle was one of the earliest heuristic search problems.



Start state



Goal state

The average solution cost for a randomly generated 8-puzzle instance is about 22 steps. The branching factor is about 3. (Branching factor means average number of successors of a state or the number of branches from a state).

Consider two heuristics for the 8 puzzle problem.

h1 = the number of misplaced tiles

For the above figure, for the start state value of h1 is 8.

h2 = the sum of the distances of the tiles from their goal positions

For the above figure, for the start state the value of

h2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18.

Branching factor

One way to characterize the quality of a heuristic is the effective branching factor, b*.

If the total number of nodes generated by A* algorithm for a particular problem is N, and the solution depth is d, then b* is the branching factor that a uniform tree of depth d would have to have in order to contain N + 1 nodes. Thus

$N + 1 = 1 + b^* + (b^*)^2 + …….. + (b^*)^d$

For example if A* finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92.

A well designed heuristic would have a value of b* close to 1.

To see the effect of heuristic functions on the 8 puzzle problem, see the table below. Different instances of 8-puzzle problem are solved using iterative deepening search and A* search using h1 and h2.

	Search cost			Branching factor		
d	IDS	A* (h1)	A* (h2)	IDS	A* (h1)	A* (h2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24

The results show that h2 is better than h1, and is far better than iterative deepening search.

Inventing heuristic functions

We have seen two heuristic functions for the 8 puzzle problem, h1 and h2.

h1 = the number of misplaced tiles

h2 = the sum of the distances of the tiles from their goal positions

Also we found that h2 is better. Is it possible for a computer to invent such heuristic functions mechanically? Yes, it is possible.

If the rules of the puzzle were changed so that a tile could move anywhere, then h1 would give the exact number of steps in the shortest solution.

Similarly if a tile could move one square in any direction, then h2 would give the exact number of steps in the shortest solution.

Suppose we write the definition of 8-puzzle problem as

A tile can move from square A to square B if

A is horizontally or vertically adjacent to B and B is blank.

From this statement we can generate three statements.

a. A tile can move from square A to Square B if A is adjacent to B.
b. A tile can move from square A to square B if B is blank.
c. A tile can move from square A to square B.

From a, we can derive h2. This is because h2 would be a proper score if we move each tile to its destination.

From c we can derive h1. This is because h1 would be a proper score, if tiles could move to their intended destinations in one step.

A program called ABSOLVER can generate heuristics automatically from problem definitions.

If a collection of heuristics h1, h2, h3… hm is available for a problem, then which one we should choose? We would choose

h (n) = max { h1, h2….hm}

Heuristic search techniques

Many of the problems that come in artificial intelligence are too complex to be solved by direct techniques. They must be attacked by appropriate search methods in association with whatever direct techniques are available to guide the search. In this topic, we will learn some general purpose search techniques. These methods are all varieties of heuristic search. These techniques form the core of most AI systems.

The following are some of the search strategies.

Depth first search,

Breadth first search,

Hill climbing,

Best first search (A* algorithm),

Constraint satisfaction search

The first two techniques, breadth first search and depth first search we have already learned.

Hill climbing **(AI by Rich & Knight)**

Hill climbing strategies expand the current state in the search and evaluate its children. The best child is selected for further expansion; neither its siblings nor its parent is retained. Search halts when it reaches a state that is better than any of its children. Hill climbing is named for the strategy that might be used by an eager, but blind mountain climber: go uphill along the steepest possible path until you can go no farther. Because it keeps no history, the algorithm cannot recover from failures of its strategy.

There are three various strategies for hill climbing. They are

Simple hill climbing,

Steepest ascent hill climbing and

Simulated annealing.

Simple hill climbing

The simplest way to implement hill climbing is as follows.

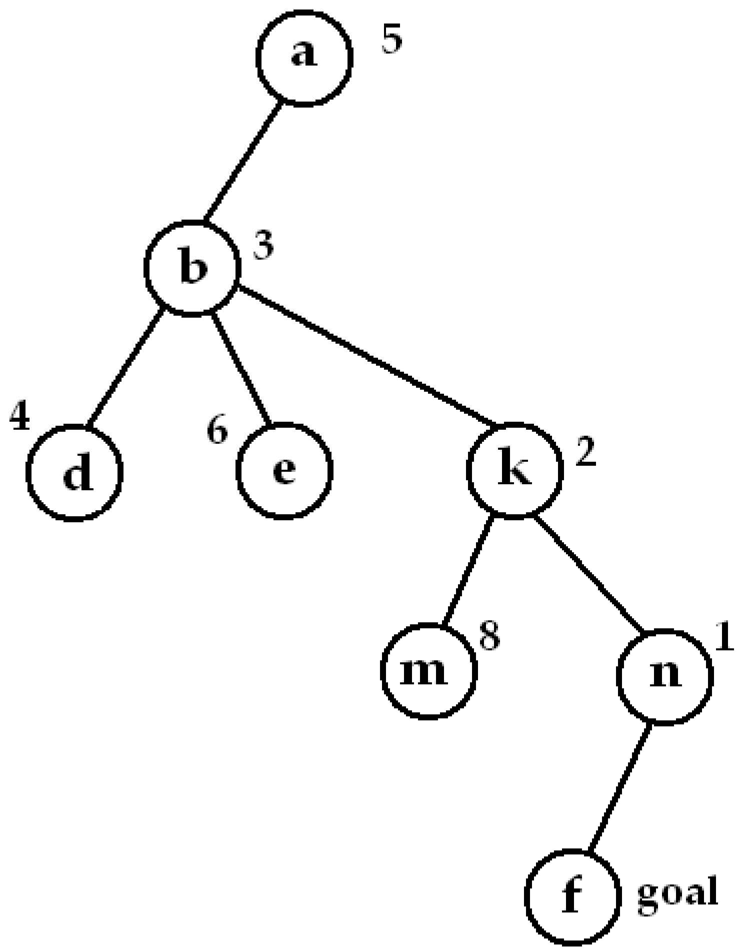
Algorithm

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

Example:

A problem is given. Given the start state as ‘a’ and the goal state as ‘f’.

Suppose we have a heuristic function h (n) for evaluating the states. Assume that a lower value of heuristic function indicates a better state.



Here a has an evaluation value of 5. a is set as the current state.

Generate a successor of a. Here it is b. The value of b is 3. It is less than that of a. that means b is better than the current state a. So set b as the new current state.

Generate a successor of b. it is d. D has a value of 4. It is not better than the current state b (3). So generate another successor of b. it is e. It has a value of 6. It is not better than the current state b (3). Then generate another successor of b. We get k. It has an evaluation value of 2. k is better than the current state b (3). So set k as the new current state.

Now start hill climbing from k. Proceed with this, we may get the goal state f.

Steepest ascent hill climbing

A useful variation on simple hill climbing considers all the moves from the current state and selects the best one as the next state.

Algorithm

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Loop until a solution is found or until a complete iteration produces no change to current state.

a. Let SUCC be a state such that any possible successor of the current will be better than SUCC.

b. For each operator that applies to the current state, do:

   i. Apply the operator and generate a new state.

   ii. Evaluate the new state. If it is a goal state, then return it and quit. If not, compare it to SUCC. If it is better, then set SUCC to this state. If it is not better, leave SUCC alone.

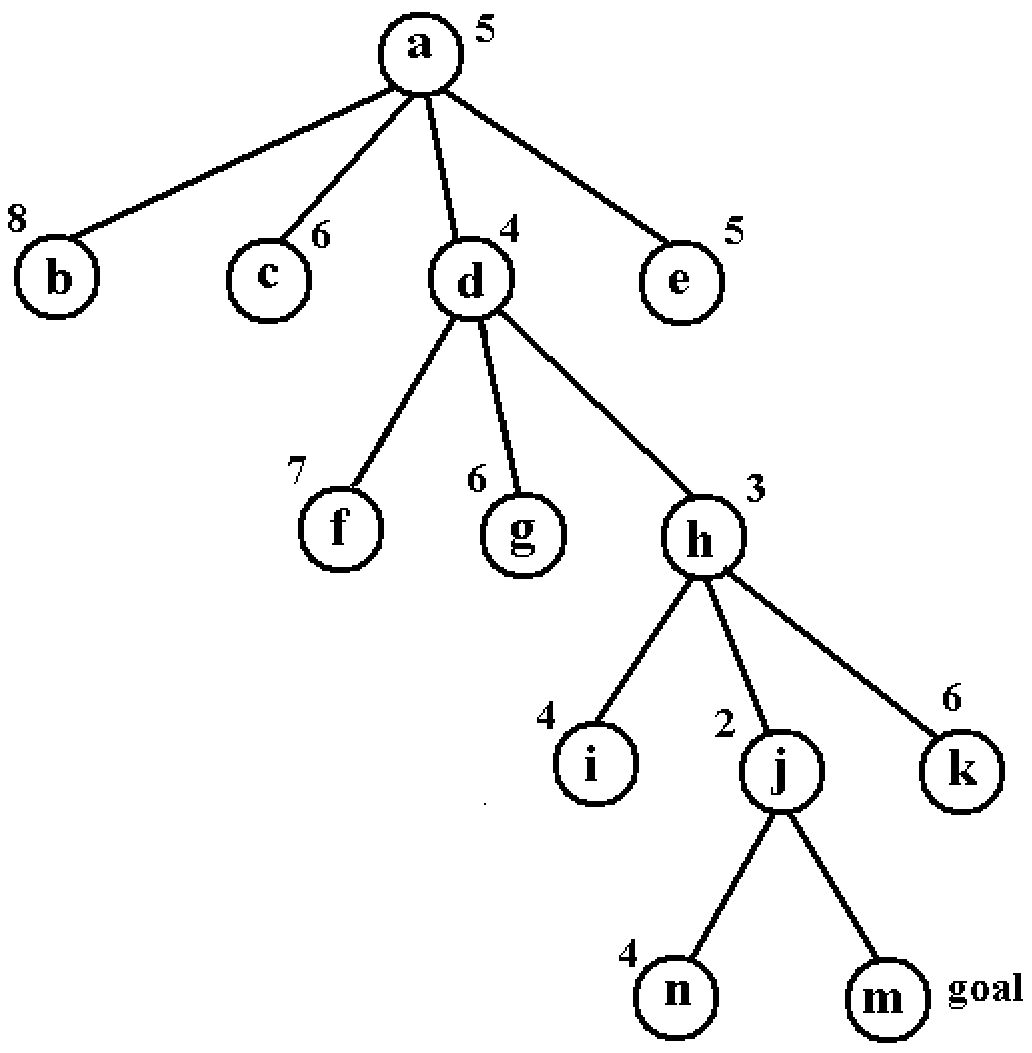
c. if the SUCC is better than current state, then set current state to SUCC.

An example is shown below.

Consider a problem. Initial state is given as a. the final state is m.

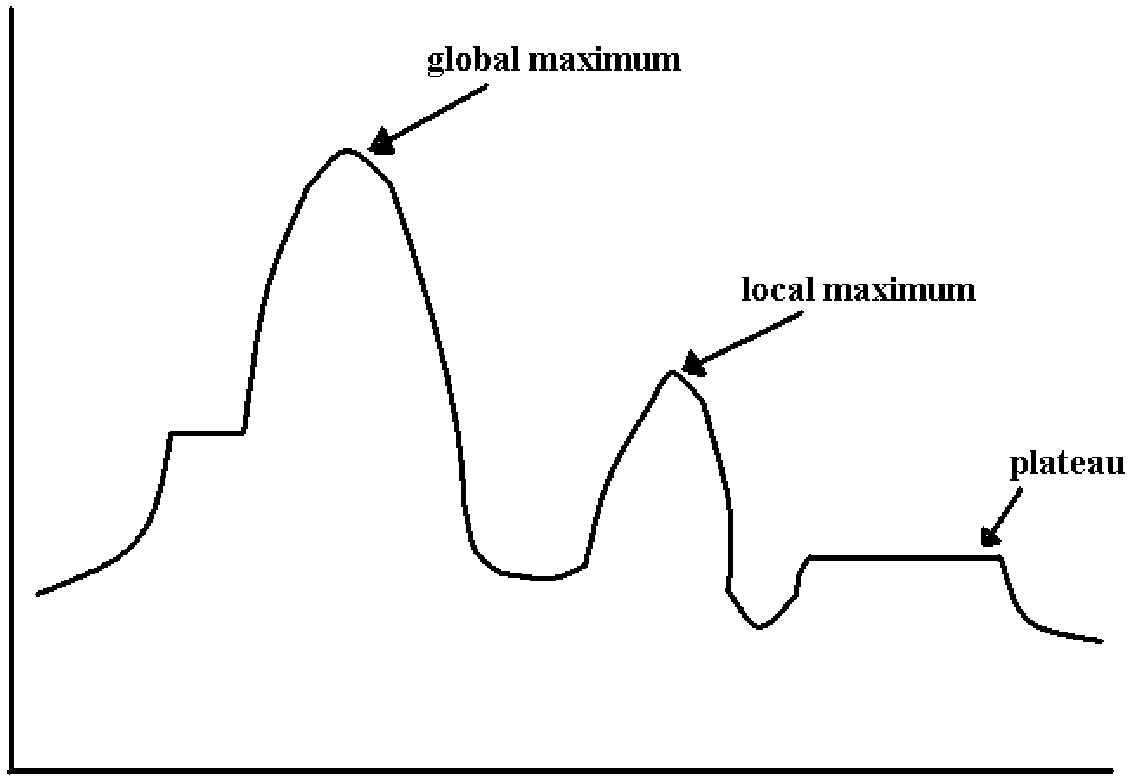
Let h(n) be a heuristic function for evaluating the states

.



In this, all the child states (successors) of a are generated first. The state d has the least value of heuristic function. So d is the best among the successors of a. then d is better than the current state a. so make d as the new current state. Then start hill climbing from d.

Both basic and steepest ascent hill climbing may fail to find a solution. Either algorithm may stop not by finding a goal state but by getting to a state from which no better states can be generated. This will happen if the program has reached either a local maximum, a plateau or a ridge.



Here in problem solving, our aim is to find the global maximum.

A local maximum is a state that is better than all its neighbors but is not better than some other states farther away.

A plateau is a flat area of the search space in which a whole set of neighboring states have the same value. On a plateau, it is not possible to determine the best direction in which to move by making local comparisons.

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.

There are some ways of dealing with these problems.

Backtrack to some earlier node and try going in a different direction. This is a fairly good way of dealing with local maxima.

Make a big jump in some direction to try to get to a new section of the search space. This is a good way of dealing with plateaus.

Apply 2 or more rules before doing the test. This corresponds to moving in several directions at once. This is a good way for dealing with ridges.

Simulated annealing

We have seen that hill climbing never makes a down hill move. As a result, it can stuck on a local maximum. In contrast, a purely random wall- that is, moving to a successor chosen uniformly at random from the set of successors- is complete, but extremely inefficient.

Simulated annealing combines hill climbing with a random walk. As a result, it yields efficiency and completeness.

Simulated annealing is a variation of hill climbing in which, at the beginning of the process, some downhill moves may be made. The idea is to do enough exploration of the whole space early on so that the final solution is relatively insensitive to the starting state. This should lower the chances of getting caught at a local maximum, a plateau or a ridge.

In metallurgy, annealing is the process in which metals are melted and then gradually cooled until some solid state is reached. It is used to tamper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to coalesce in to a low energy crystalline state.

Physical substances usually move from higher energy configurations to lower ones. But there is some probability that a transition to a higher energy state will occur. This probability is given by the function

$$P = e^{-\Delta E / kT}$$

Where ΔE is positive change in the energy level, T is the temperature and k is Boltzmann's constant.

Physical annealing has some properties. The probability of a large uphill move is lower than the probability of a small one. Also, the probability that an uphill move will be made decreases as the temperature decreases. Large uphill moves may occur early on, but as the process progresses, only relatively small upward moves are allowed until finally the process converges to a local maximum configuration.

If cooling occurs so rapidly, stable regions of high energy will form. If however, a slower schedule is used, a uniform crystalline structure which corresponds to a global minimum is more likely to develop. If the schedule is too slow, time is wasted.

These properties of physical annealing can be used to define the process of simulated annealing. In this process, ΔE represents change in the value of heuristic evaluation function instead of change in energy level. k can be integrated in to T. hence we use the revised probability formula

$$P' = e^{-\Delta E / T}$$

The algorithm for simulated annealing is as follows.

Algorithm

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

        ΔE = (value of current) – (value of new state)

        - If the new state is a goal state, then return it and quit.
        - If it is not a goal state, but is better than the current state, then make it the current state. Also set BEST-SO-FAR to this new state.
        - If it is not better than the current state, then make it the current state with probability p' as defined above. This step is usually implemented by invoking a random number generator to produce a number in the range [0, 1]. If that number is less than p', then the move is accepted. Otherwise, do nothing.

    c. Revise T as necessary according to the annealing schedule.

5. Return BEST- SO- FAR as the answer.

To implement this algorithm, it is necessary to maintain an annealing schedule. That is first we must decide the initial value to be used for temperature. The second criterion is to decide when the temperature of the system should be reduced. The third is the amount by which the temperature will be reduced each time it is changed.

In this algorithm, instead of picking the best move, it picks a random move. If the move improves the situation, it is always accepted. Otherwise the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the amount ΔE by which the evaluation is worsened. The probability also decreases as the temperature T goes down. Thus bad moves are more likely to be allowed at the start when the temperature is high, and they become more unlikely as T decreases.

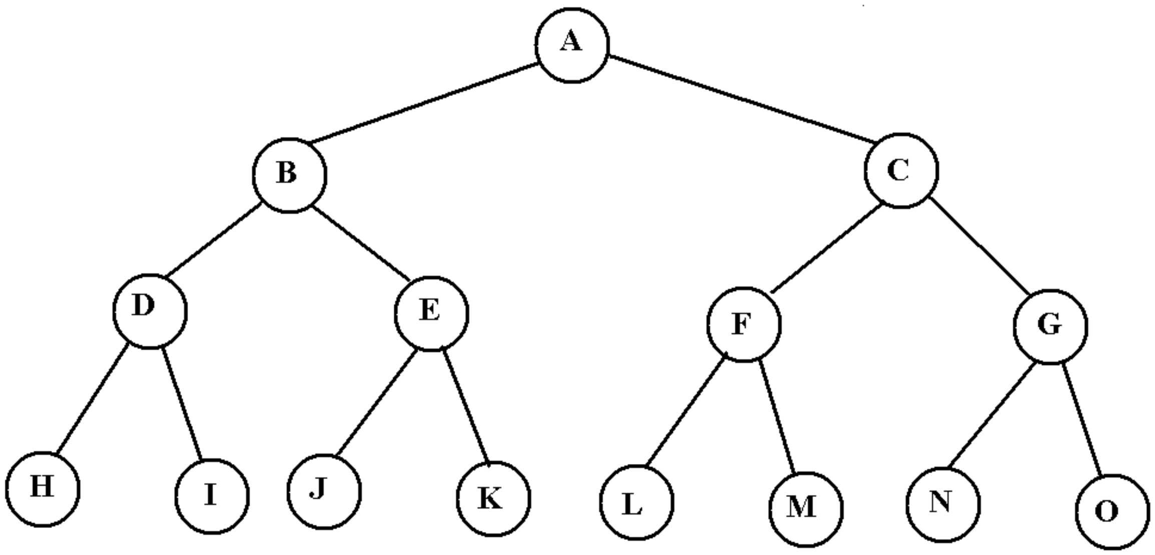
Simulated annealing was first used extensively to solve VLSI layout problems.

Iterative deepening search

Iterative deepening search (or iterative deepening depth first search) is a general strategy used in combination with depth first search. It uses a depth bound on depth first search. It does by gradually increasing the limit – first 0, then 1, then 2 and so on – until a goal is found.

Figure shows four iterations of iterative deepening search on a binary search tree. Here the solution is found on the 4th iteration.

Consider a state space graph shown below.



The iterative deepening search on the above graph generates states as given below.

	States generated in the order
Limit = 0	A
Limit = 1	A B C
Limit = 2	A B D E C F G
Limit = 3	A B D H I E J K C F L M G N O

Iterative deepening search performs a depth first search of the space with a depth bound of 1. If it fails to find a goal, it performs another depth first search with a depth bound of 2. This continues, increasing the depth bound by 1 at each iteration. At each iteration, the algorithm performs a complete depth first search to the current depth bound.

Algorithm

1. Set depth limit = 0.
2. Conduct a depth first search to a depth of depth limit. If a solution path is found, then return it.
3. Otherwise, increment depth limit by 1 and go to step 2.

Iterative deepening search continues the benefits of depth first and breadth first search. It is the preferred search method when there is a large search space and the depth of the solution is not known.

For example, a chess program may be required to complete all its moves within 2 hours. Since it is impossible to know in advance how long a fixed depth tree search will take, a program may find itself running out of time. With iterative deepening, the current search can be aborted at any time and the best move found by the previous iteration can be played. Previous iterations can provide invaluable move ordering constraints.

Informed search

These search strategies use problem specific knowledge for finding solutions.

Best first search (A* algorithm)

Best first search is a general informed search strategy. Here a node is selected for an expansion based on an evaluation function, f(n). Traditionally, the node with the lowest evaluation is selected for expansion. It is implemented using a priority queue.

Best first search combines breadth first and depth first search. That is to follow a single path at a time, but switch paths whenever some competing path looks more promising than the current one does.

Best first search uses 2 lists, open and closed. Open to keep track of the current fringe of search and closed to record states already visited.

Algorithm

```
function best_first_search ( )
{
open = [start];
closed = [ ];
while (open not empty)
{
        remove the left most state from open, call it X;
        If X = goal then return the path from start to X;
        Else
        {
                generate children of X;
                for each child of X do
                {
                        case
                        the child is not in open or closed :
                        {
                                assign the child a heuristic value;
                                add the child to open;
                        }
```

```
                    case
                    the child is already on open:
                    {
                        if the child was reached by a shorter path
                        then give the state on open the shorter path;
                    }
                    case
                    the child is already on closed :
                    {
                        if the child was reached by a shorter path
                        then
                        {
                        remove the state from closed;
                        add the child to open;
                        }
                    }
                }  /*end of for */
    put X on closed;
    reorder states on open by heuristic merit;
    }  /* end of else */
return FAIL;
}
```

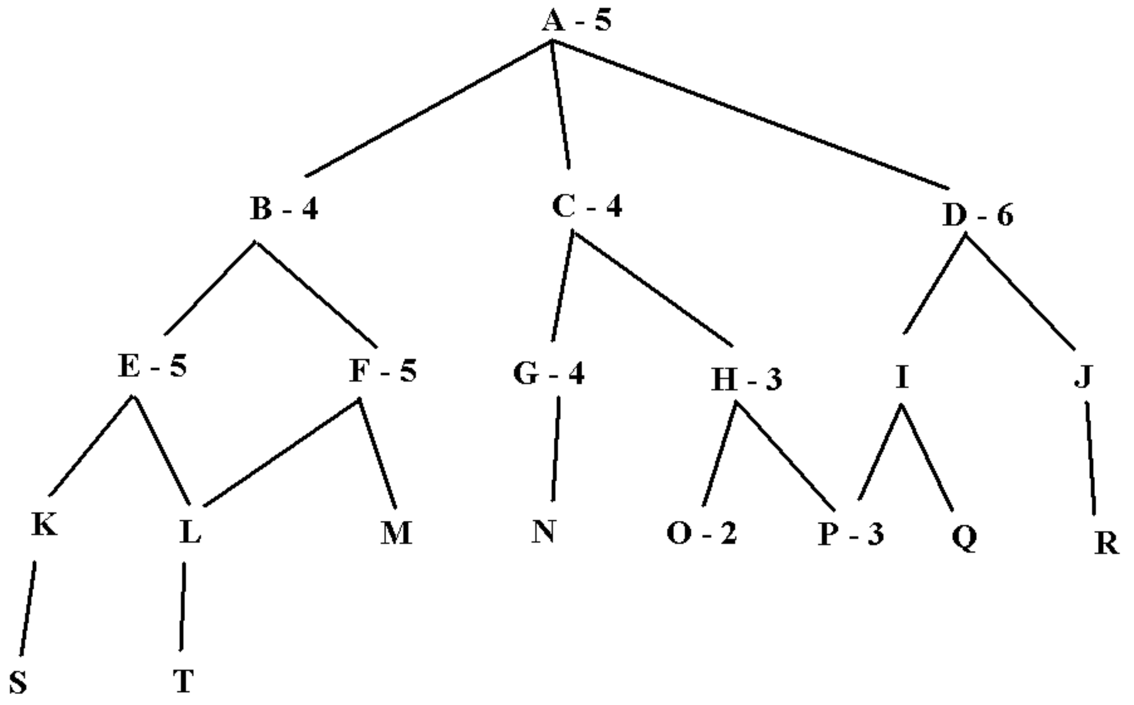
Here open acts as a priority queue. Algorithm orders the states on open according to some heuristic estimate of their "closeness" to a goal. Each iteration of the loop considers the most promising state on the open list. At each iteration, best first search removes the first element from the open list. If it meets the goal conditions, the algorithm returns the solution path that led to the goal. Each state retains ancestor information to determine, if it had previously been reached by a shorter path and to allow the algorithm to return the final solution path.

If the first element on open is not a goal, the algorithm applies all matching production rules or operators to generate its descendents. If a child state is already on open or closed, the algorithm checks to make sure that the state records the shorter of the 2 partial solution paths. Duplicate states are not retained. By updating the ancestor history of nodes on open and closed when they are rediscovered, the algorithm is more likely to find a shorter path to a goal.

Best first search applies a heuristic evaluation to the states on open, and the list is sorted according to the heuristic values of these states. This brings the best states to the front of open.

Eg.

Figure shows a state space with heuristic evaluations. Evaluations are attached to some of its states.



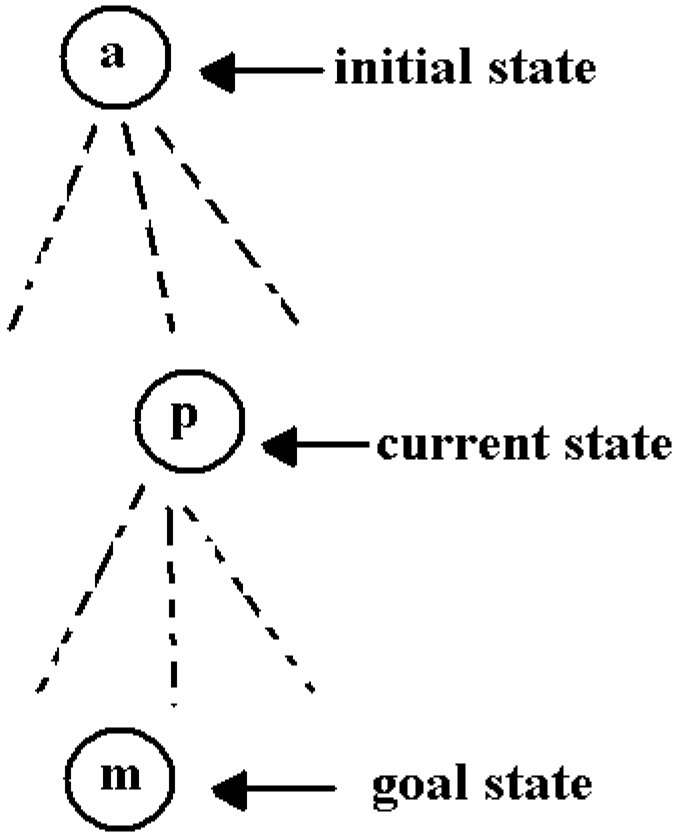
A trace of the execution of best first search on this graph appears below. Suppose P is the goal state.

		Open	closed
1.		A5	empty
2.	Evaluate A5	B4, C4, D6	A5
3.	Evaluate B4	C4, E5, F5, D6	B4, A5
4.	Evaluate C4	H3, G4, E5, F5, D6	C4, B4, A5
5.	Evaluate H3	O2, P3, G4, E5, F5, D6	H3, C4, B4, A5
6.	Evaluate O2	P3, G4, E5, F5, D6	O2, H3, C4, B4, A5
7.	Evaluate P3	the solution is found.	

The best first search algorithm always selects the most promising state on open for further expansion. It does not abandon all other states but maintains them on open. In the event a heuristic leads the search down a path that proves incorrect, the algorithm will eventually retrieve some previously generated, next best state from open and shifts its focus to another part of the space. In the figure, after the children of state B were found to have poor heuristic evaluations the search shifted its focus to state c. the children of B were kept on open in case the algorithm needed to return them later.

Implementing heuristic evaluation functions

We need a heuristic function that estimates a state. We call this function f’. it is convenient to define this function as the sum of 2 components, g and h’.



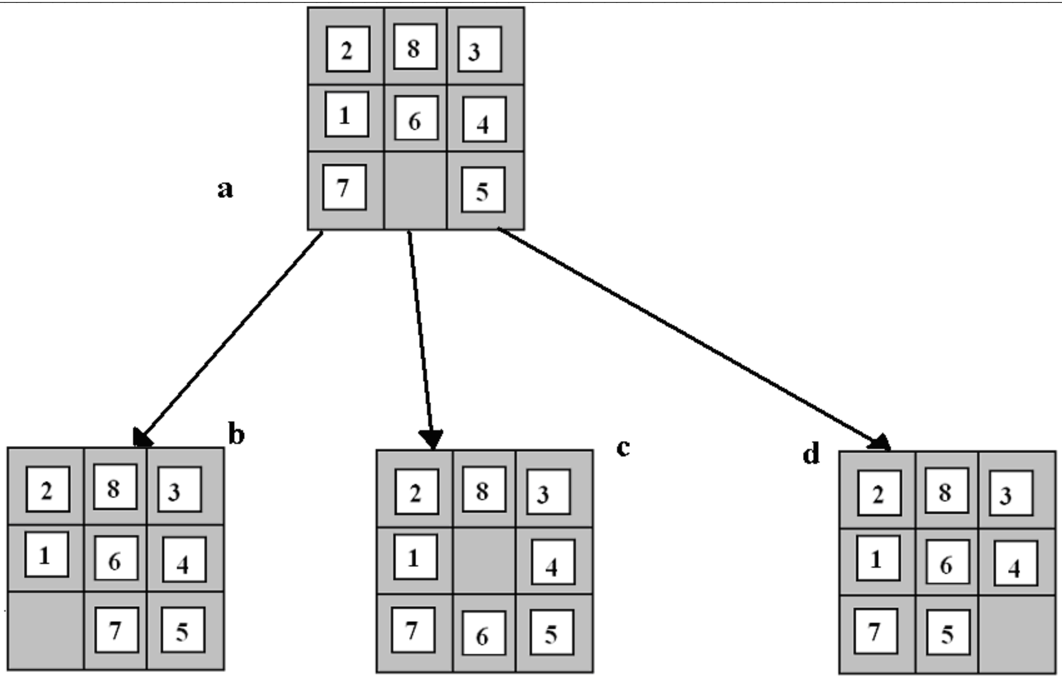
The function g is the actual cost of getting from the initial state (a) to the current state (p). The function h’ is an estimate of the cost of getting from the current state (p) to a goal state (m). Thus

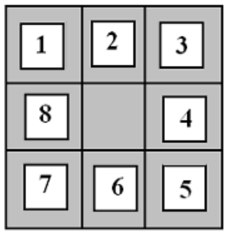
$$f' = g + h'$$

the function f’, then, represents an estimate of the cost of getting from the initial state (a) to a goal state (m) along the path that generated the current state (p).

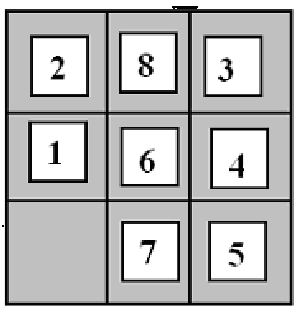
if this function f’ is used for evaluating a state in the best first search algorithm, the algorithm is called A* algorithm.

We now evaluate the performance of a heuristic for solving 8 puzzle problem. Figure shows the start and goal states, along with the first 3 states generated in the search.

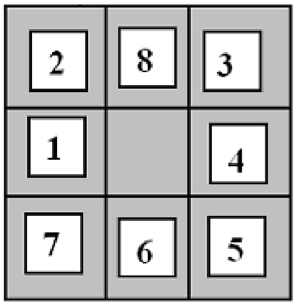



Goal state

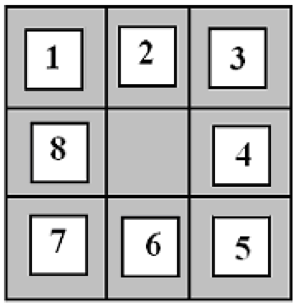
We consider a heuristic that counts the tiles out of place in each state when it is compared with the goal. The following shows the result of applying this heuristic to the 3 child states of the above figure.



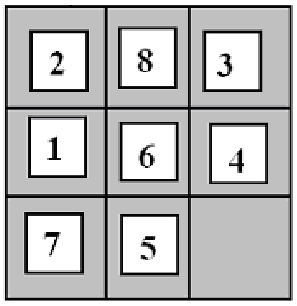
h' (n) = 5



h' (n) = 3

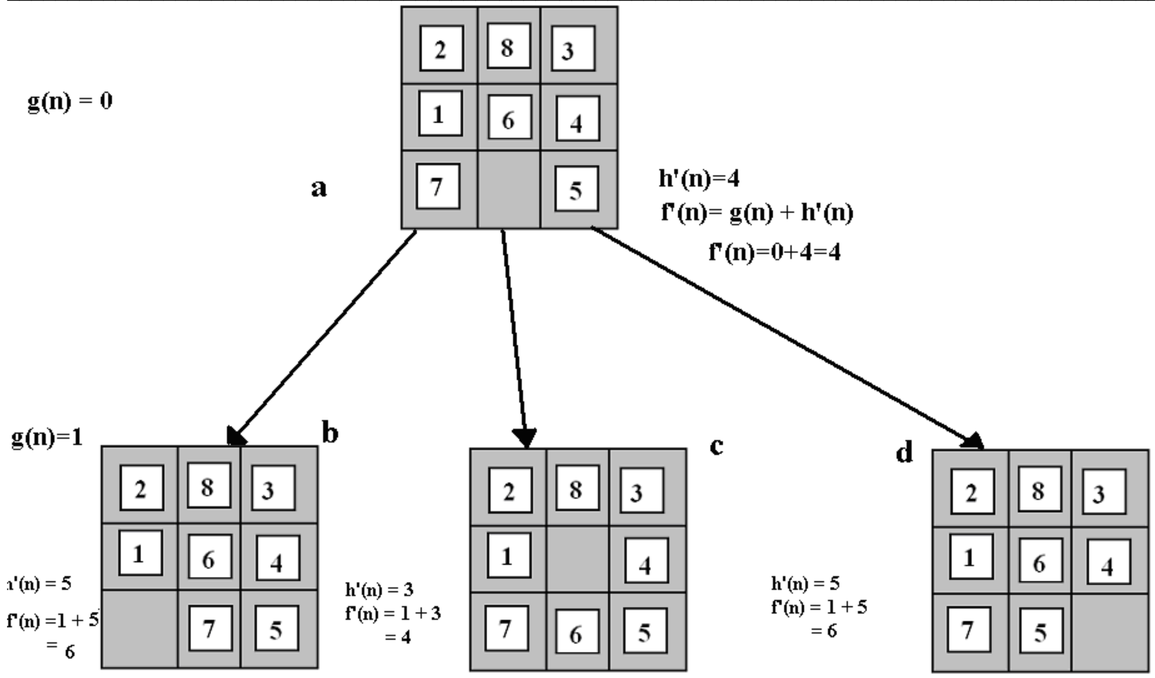


Goal state

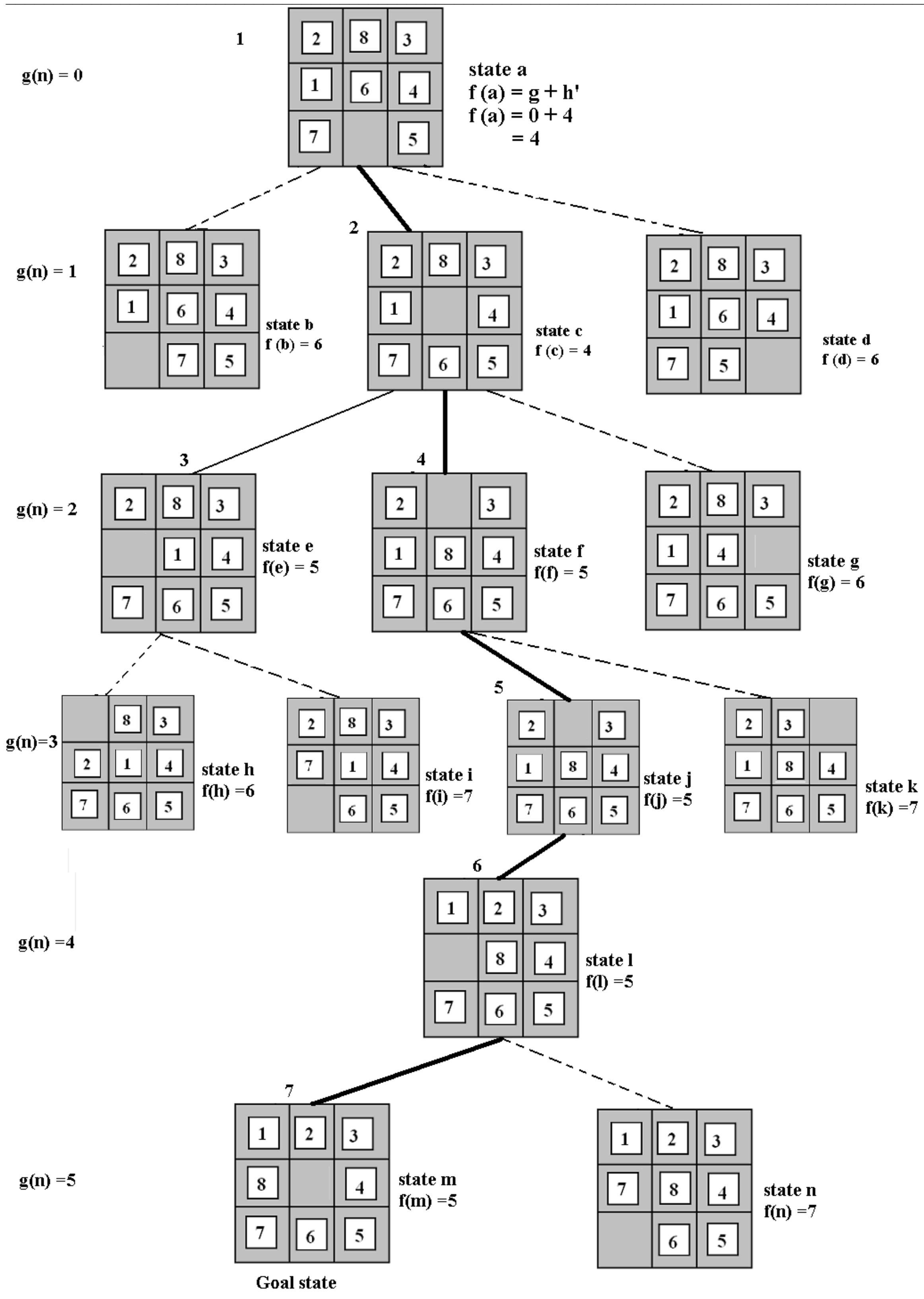


h' (n) = 5

The distance from staring state to its descendents (g) can be measured by maintaining a depth count for each state. This count is 0 for the beginning state and is incremented by 1 for each level of the search. It records the actual number of moves that have been used to go from the start state in a search to each descendent. The following shows the f values.



The best first search of 8 puzzle graph using f as defined above appears below.



Each state is labeled with a letter and its heuristic weight, f' (n) = g(n) + h'(n).

	Open	closed
1	a4	
2	c4, b6, d6	a4
3	c5, f5, b6, d6, g6	a4, c4
4	f5, h6, b6, d6, g6, i7	a4, c4, e5
5	j5, h6, b6, d6, g6, k7, i7	a4, c4, e5, f5
6	l5, h6, b6, d6, g6, k7, i7	a4, c4, e5,f5, j5
7	m5, h6, b6, d6, g6, n7, k7, i7	a4, c4, e5, f5, j5, l5
8	success, m= goal	

Notice the opportunistic nature of best first search.

The g(n) component of the evaluation function gives the search move of a breadth first flavor. This prevents it from being misled by an erroneous evaluation; if a heuristic continuously returns 'good' evaluations for states along a path that fails to reach a goal, the g value will grow to dominate h and force search back to a shorter solution path. This guarantees that the algorithm will not become permanently lost, descending an infinite branch.

A* algorithm

The following shows an expanded version of the above best first search algorithm called A* algorithm.

```
1. open = [start];
           Set the start node's  g = 0;
                     h' value to whatever it is.
                     f' = g + h' = 0 +h';
                     f' = h'
   closed = [  ];

2. Until a goal node is found, repeat the following procedure.
   {
     if (open = [ ] ) return failure;
     else
           {
           Pick the node on open with the lowest f' value. Call it BESTNODE;
           Remove it from open;          Place it on closed;
```

```
If (BESTNODE is a goal node) then return success;
Else
{
        Generate the successors of BESTNODE;
        for each SUCCESSOR
        {
        a. set SUCCESSOR to point back to BESTNODE;
                These backward 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 BESTNODE to SUCCESSOR.
        c. See if SUCCESSOR is same as any node on open. That is it has already been generated but not processed. If so, call that node OLD. Since the 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. To 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, 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 in open, see if it is in 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 2c, 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 downwards, do a depth first traversal of the tree starting at OLD, changing each node's g value and 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. This condition is easy to check for. Each node's parent link points back to its best known parent. As we propagate down to a node, see if its parent points to the node we are coming from. If so,
```

continue the propagation. If not, then its g value already reflects the better path of which it is part. So the propagation may stop here. But it is possible that with the new value of g being propagated downward, the path we are following may become better than the path through the current parent. So compare the two. If the path through the current parent is still better, stop the propagation. If the path we are propagating through is now better, reset the parent and continue propagation.

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).