SHAMBHUNATH INSTITUTE OF ENGINEERING AND TECHNOLOGY

Department of Computer Science & Engineering

LAB MANUAL

Course: **B.Tech Computer Science & Engineering**
Subject: **Artificial Intelligence**
Branch/Semester: **VII Semester**

Index

1. Write a program to implementation of DFS
2. Write a program to implement BFS
3. Write a program to implement Traveling Salesman Problem
4. Write a program to implement Simulated Annealing Algorithm
5. Write a program to implement 8 puzzle problem
6. Write a program to implement Tower of Hanoi problem
7. Write a program to implement A* Algorithm
8. Write a program to implement Hill Climbing Algorithm
9. To Study JESS expert system
10. To Study RVD expert system

Experiment No: 1

Aim: Write a program to implement Depth First Search

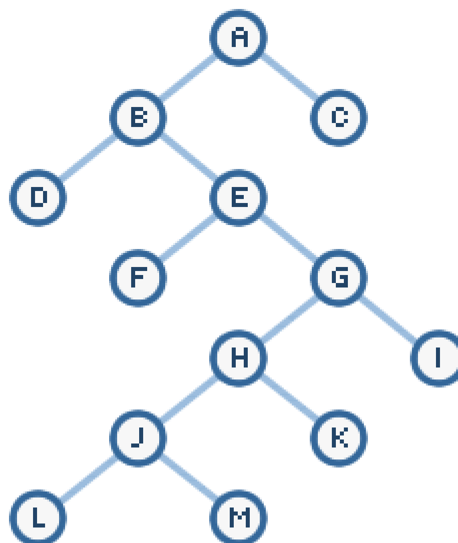
Theory:

If we want to go from Point A to Point B, you are employing some kind of search. For a direction finder, going from Point A to Point B literally means finding a path between where you are now and your intended destination. For a chess game, Point A to Point B might be two points between its current position and its position 5 moves from now. For a genome sequencer, Points A and B could be a link between two DNA sequences.

As you can tell, going from Point A to Point B is different for every situation. If there is a vast amount of interconnected data, and you are trying to find a relation between few such pieces of data, you would use search. In this tutorial, you will learn about two forms of searching, depth first and breadth first.

Searching falls under Artificial Intelligence (AI). A major goal of AI is to give computers the ability to think, or in other words, mimic human behavior. The problem is, unfortunately, computers don't function in the same way our minds do. They require a series of *well-reasoned out* steps before finding a solution. Your goal, then, is to take a complicated task and convert it into simpler steps that your computer can handle. That conversion from something complex to something simple is what this tutorial is primarily about. Learning how to use two search algorithms is just a welcome side-effect.

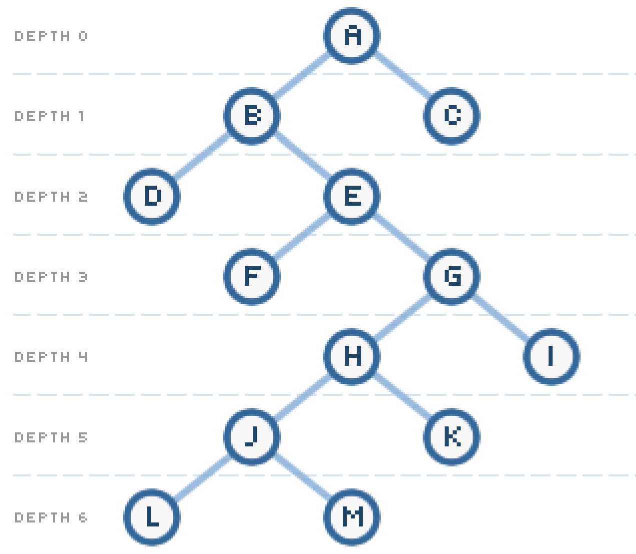
Our Search Representation Let's first learn how we humans would solve a search problem. First, we need a representation of how our search problem will exist. The following is an example of our search tree. It is a series of interconnected nodes that we will be searching through:



In our above graph, the path connections are not two-way. All paths go only from top to bottom. In other words, A has a path to B and C, but B and C do not have a path to A. It is basically like a one-way street.

Each lettered circle in our graph is a node. A node can be connected to other via our edge/path, and those nodes that its connects to are called neighbors. B and C are neighbors of A. E and D are neighbors of B, and B is not a neighbors of D or E because B cannot be reached using either D or E.

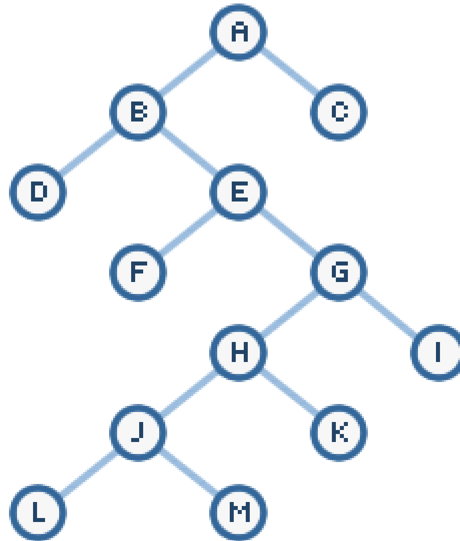
Our search graph also contains depth:



We now have a way of describing location in our graph. We know how the various nodes (the lettered circles) are related to each other (neighbors), and we have a way of characterizing the depth each belongs in. Knowing this information isn't directly relevant in creating our search algorithm, but they do help us to better understand the problem.

Depth First Search Depth first search works by taking a node, checking its neighbors, expanding the first node it finds among the neighbors, checking if that expanded node is our destination, and if not, continue exploring more nodes.

The above explanation is probably confusing if this is your first exposure to depth first search. I hope the following demonstration will help more. Using our same search tree, let's find a path between nodes A and F:

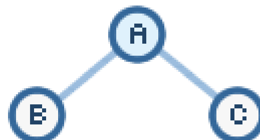


Step 0
let's start with our root/goal node:

I will be using two lists to keep track of what we are doing - an Open list and a Closed List. An Open list keeps track of what you need to do, and the Closed List keeps track of what you have already done. Right now, we only have our starting point, node A. We haven't done anything to it yet, so let's add it to our Open list.

- Open List: A
- Closed List: <empty>

Step 1
Now, let's explore the neighbors of our A node. To put another way, let's take the first item from our Open list and explore its neighbors:



Node A's neighbors are the B and C nodes. Because we are now done with our A node, we can remove it from our Open list and add it to our Closed List. You aren't done with

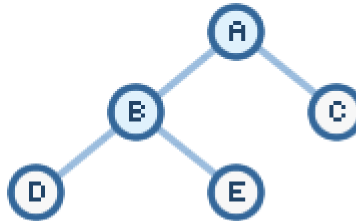
this step though. You now have two new nodes B and C that need exploring. Add those two nodes to our Open list.

Our current Open and Closed Lists contain the following data:

- Open List: B, C
- Closed List: A

Step 2

Our Open list contains two items. For depth first search and breadth first search, you always explore the first item from our Open list. The first item in our Open list is the B node. B is not our destination, so let's explore its neighbors:



Because I have now expanded B, I am going to remove it from the Open list and add it to the Closed List. Our new nodes are D and E, and we add these nodes to the *beginning* of our Open list:

- Open List: D, E, C
- Closed List: A, B

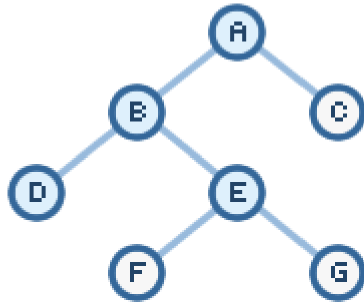
Step 3

We should start to see a pattern forming. Because D is at the beginning of our Open List, we expand it. D isn't our destination, and it does not contain any neighbors. All you do in this step is remove D from our Open List and add it to our Closed List:

- Open List: E, C
- Closed List: A, B, D

Step 4

We now expand the E node from our Open list. E is not our destination, so we explore its neighbors and find out that it contains the neighbors F and G. Remember, F is our target, but we don't stop here though. Despite F being on our path, we only end when we are about to *expand* our target Node - F in this case:

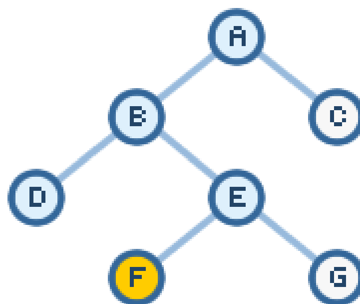


Our Open list will have the E node removed and the F and G nodes added. The removed E node will be added to our Closed List:

- Open List: F, G, C
- Closed List: A, B, D, E

Step 5

We now expand the F node. Since it is our intended destination, we stop:



We remove F from our Open list and add it to our Closed List. Since we are at our destination, there is no need to expand F in order to find its neighbors. Our final Open and Closed Lists contain the following data:

- Open List: G, C
- Closed List: A, B, D, E, F

The final path taken by our depth first search method is what the final value of our Closed List is: A, B, D, E, F. Towards the end of this tutorial, I will analyze these results in greater detail so that you have a better understanding of this search method.

Pseudocode / Informal Algorithms

Now that we have a good idea on how to solve these various search problems, we need to translate them so that a computer can solve it. Before we get into the code, let's solve it generally using pseudocode, or if you prefer a better name, an informal algorithm.

The pseudocode for depth first search is:

i. Declare two empty lists: Open and Closed.
ii. Add Start node to our Open list.
iii. While our Open list is not empty, loop the following:

   a. Remove the first node from our Open List.
   b. Check to see if the removed node is our destination.

      i. If the removed node is our destination, break out of the loop, add the node to our closed list, and return the value of our Closed list.
      ii. If the removed node is not our destination, continue the loop (go to Step c).

   c. Extract the neighbors of our above removed node.
   d. Add the neighbors to the *beginning* of our Open list, and add the removed node to our Closed list. Continue looping.

The following two subsections explain what the grayed out code does in both our search methods:

The gray code in a depth first search is replaced with the following:

```
openList.unshift(neighbors[nLength - i - 1]);
```

Unshift inserts values at the beginning of our array. Therefore, I add the values to the beginning of our array in reverse; hence the awkward syntax: nLength - i - 1. Let me give you an example.

Let's say our neighbors array contains the following three items: **a, b, c**. If we were to unshift each item using neighbors[i], in the first iteration of the for loop, the value of **a** will be added to the first position in our openList. So far so good.

In the second iteration, the value of b will be added to the first position in our openList. So now, openList contains b first, then a: [b, a]. In the third iteration, our openList will look like [c, b, a]. The end result is that the data is reversed!

To avoid the reversal, I add the items to our openList in reverse. First c gets inserted, then b, then a. That works in keeping with our algorithm and examples from the previous pages.

Result: The depth first search algorithm is implemented.

Experiment No: 2

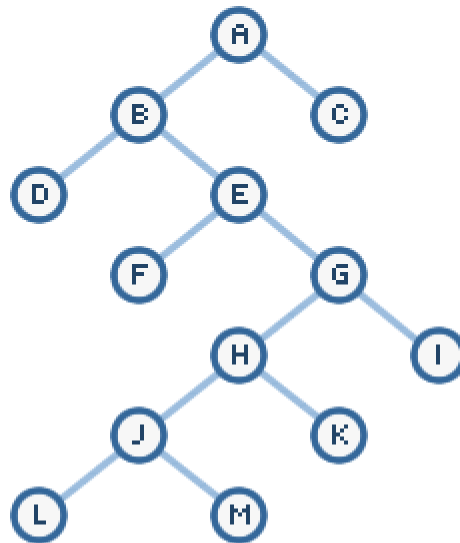
Aim: To implement Breadth First Search Algorithm

Theory:

Breadth First Search

The reason we cover both depth and breadth first search methods in the same tutorial is because they are both similar. In depth first search, newly explored nodes were added to the beginning of your Open list. In breadth first search, newly explored nodes are added to the end of your Open list.

Let's see how that change will affect our results. For reference, here is our original search tree:



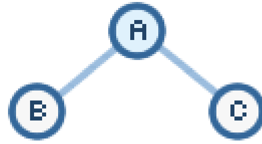
Let's try to find a path between nodes A and E.

Step 0
Let's start with our root/goal node:

Like before, I will continue to employ the Open and Closed Lists to keep track of what needs to be done:

- Open List: A
- Closed List: <empty>

Step 1
Now, let's explore the neighbors of our A node. So far, we are following in depth first's foot steps:



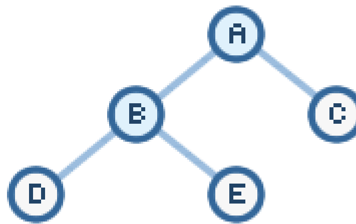
We remove A from our Open list and add A to our Closed List. A's neighbors, the B and C nodes, are added to our Open list. They are added to the end of our Open list, but since our Open list was empty (after removing A), it's hard to show that in this step.

Our current Open and Closed Lists contain the following data:

- Open List: B, C
- Closed List: A

Step 2

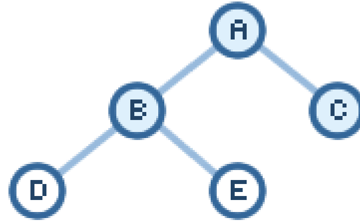
Here is where things start to diverge from our depth first search method. We take a look the B node because it appears first in our Open List. Because B isn't our intended destination, we explore its neighbors:



B is now moved to our Closed List, but the neighbors of B, nodes D and E are added to the *end* of our Open list:

- Open List: C, D, E
- Closed List: A, B

Step 3
we now expand our C node:



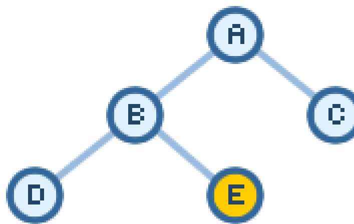
Since C has no neighbors, all we do is remove C from our Closed List and move on:

- Open List: D, E
- Closed List: A, B, C

Step 4
Similar to Step 3, we expand node D. Since it isn't our destination, and it too does not have any neighbors, we simply remove D from our to Open list, add D to our Closed List, and continue on:

- Open List: E
- Closed List: A, B, C, D

Step 5
because our Open list only has one item, we have no choice but to take a look at node E. Since node E is our destination, we can stop here:



Our final versions of the Open and Closed Lists contain the following data:

- Open List: <empty>
- Closed List: A, B, C, D, E

The pseudocode for breadth first first search is:

i. Declare two empty lists: Open and Closed.
ii. Add Start node to our Open list.
iii. While our Open list is not empty, loop the following:

    a. Remove the first node from our Open List.
    b. Check to see if the removed node is our destination.

    i. If the removed node is our destination, break out of the loop, add the node to our Closed list, and return the value of our Closed list.
    ii. If the removed node is not our destination, continue the loop (go to Step c).
c. Extract the neighbors of our above removed node.
d. Add the neighbors to the *end* of our Open list, and add the removed node to our Closed list.

Because both depth first and breadth first search methods contain the same data except for one line, I will not describe each method's code separately.

```
var origin:Node = nodeOrigin;
var destination:Node = nodeDestination;
```

In these two lines, you are defining and initializing your origin and destination variables. In order to keep the code example applicable to a wider set of Flash implementations, we are assuming your origin and target nodes are already somehow given to you and stored in the nodeOrigin and nodeDestination variables.

Your implementation of it, if you are using the Graph ADT will be different. In the next page, you will see in my implementation, nodeOrigin and nodeDestination are replaced with something else.

```
// Creating our Open and Closed Lists
var closedList:Array = new Array();
var openList:Array = new Array();
// Adding our starting point to Open List
openList.push(origin);
```

We declare our closedList and openList arrays. These arrays store the progress of what we have done so far and a list of what needs to be done.

For the initial step from our algorithm, I add the origin node to our openList. With that done, it's time to get into our loop!

```
// Loop while openList contains some data.
while (openList.length != 0) {
```

As long as our openList contains some data, we continue looping. Ideally, openList should always contain some data such as nodes from recently explored neighbors. If

openList is empty, besides the loop ending, it also means that you have explored your entire search tree without finding a solution.

```
// Loop while openList contains some data.
var n:Node = Node(openList.shift());
```

An array's shift method removes the first item from the array and returns that value. Because I specify that the variable n requires an object of type Node, we typecast our openList.shift() operation with the Node(data) operation. If we did not do that, Flash would give me an error.

The main thing to remember for the rest of this code explanation is that the variable n refers to the current, active node removed from openList.

```
// Check if node is Destination
if (n == destination) {
    closedList.push(destination);
    trace("Done!");
    break;
}
```

we first check if n is our destination. If the current node you are examining is the destination, you are done. we add n to our closedList array and trace the string Done!.

If you are going to be parsing your data or making any modifications to your final output stored in closedList, you would do so right here before the break command.

```
// Store n's neighbors in array
var neighbors:Array = n.getNeighbors();
var nLength:Number = neighbors.length;
```

If your program makes it here, it means that n is not our intended destination. We now have to find out who its neighbors are. I use the getNeighbors method on n to return an array of its neighbors.

In order to simplify how future lines of code look, I store the length of our recent neighbors data in the nLength variable.

```
// Add each neighbor to the beginning of our openList
for (i=0; i< nLength; i++) {
    < depends on search method >
}
```

This for loop cycles through each of our neighbors and adds the values to openList. The location at which the neighbors get added to our openList (grayed out line) is where the difference between depth first and breadth first searches occur.

The gray code in a breadth first search is replaced with the following:

```
openList.push(neighbors[i]);
```

This is very straightforward. Push adds a value to the end of your array, and since that is what we need to do for breadth first search anyway, we are done...well, almost.

```
// Add current node to closedList
closedList.push(n);
```

Finally we are at the end. Since we have already done everything we can to this node (check if it is our destination and get its neighbors), let's retire it by adding it to our closedList array.

Result: The breadth first search algorithm is implemented.

Experiment No: 3

Aim: Write a Program to implement Traveling Salesman Problem

Introduction:

Problem

A traveling salesman has to travel through a bunch of cities, in such a way that the expenses on traveling are minimized. This is the infamous Traveling Salesman Problem (aka **TSP**) problem it belongs to a family of problems, called NP-complete problem. It is conjectured that all those problems requires exponential time to solve them. In our case, this means that to find the optimal solution you have to go through all possible routes, and the numbers of routes increases exponential with the numbers of cities.

Formal Definition

We are given a complete undirected graph **G** that has a nonnegative integer cost (weight) associated with each edge, and we must find a Hamiltonian cycle (a tour that passes through all the vertices) of G with minimum cost.

Theory:

If you want to get a notion of what numbers we are talking about look at this: the number of routes with 50 cities is (50-2)!, which is

12,413,915,592,536,072,670,862,289,047,373,375,038,521,486,354,677,760,000,000,000

An alternative approach would be to compute a solution which is not optimal, but is guaranteed to be close the optimal solution. We present here an applet that implements such an approximation algorithm for the Euclidean TSP problem.

In our case we have points in the plane (i.e. cities) and the cost of the traveling between two points is the distance between them. In other words, we have a map with cities, any two of which are connected by a direct straight road and we want to find a shortest tour for our poor traveling salesman, who "wants" to visit every city.

Description of Algorithms

x2 Algorithm:

- Find Minimum Spanning Tree (MST).
- Perform preoder tree walk on MST, and construct a list of the vertices as we encounter them. (i.e. each vertex will appear only one - corresponding to its first encounter)
- The output tour is hamiltonian cycle that visits the vertices in order of this list.

The running time of this algorithm is O (n^2log (n)) time; since the input is a complete graph (n is the number of inserted points).

The length of the resulting tour is at most twice of the optimal tour, since its length is at most twice that of the MST, and the optimal tour is longer than the MST.

X1.5 Algorithm:

- Find Minimum Spanning Tree (MST).
- Find minimum weighted matching between odd vertices of MST.
- Find an Euler tour in resulting graph and create list of vertices that represents it.
- Find Hamilton cycle (which is in fact TSP tour) by visiting vertices in order of created list when only first appearance of vertex in list is encountered and any other appearance is skipped.

The approximation ratio bound is 1.5, although the argument here is a bit more complicated.

Iterative Algorithm:

We generated an arbitrary initial solution, by visiting all points in order they were inserted.
Then in each iteration:

1. Select two random cities
2. Interchange the two cities predecessors
3. Calculate the weight of resulting tour.
4. If new weight is smaller then old one - if so, replace the old tour by the new tour, else undo (2).

Removing Intersections Scheme:

This algorithm is applied on already computed TSP tour. It performs a loop which stops only after all intersections have been removed (which must happen after at most n^2times). Each loop goes through all the pairs of edges in the tour and checks if they intersect.
- If so then two new edges will replace them. New edges created by interchanging the original edge's endpoints. To preserve TSP tour completeness the path between the original edges is reversed.

Result: The traveling salesman problem is implemented.

Experiment No: 4

Aim: To implement simulated Annealing Algorithm

Introduction:

Simulated annealing

Simulated annealing (SA) is a generic probabilistic metaheuristic for the global optimization problem, namely locating a good approximation to the global minimum of a given function in a large search space. It is often used when the search space is discrete (e.g., all tours that visit a given set of cities). For certain problems, simulated annealing may be more effective than exhaustive enumeration — provided that the goal is merely to find an acceptably good solution in a fixed amount of time, rather than the best possible solution.

The name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. The heat causes the atoms to become unstuck from their initial positions (a local minimum of the internal energy) and wander randomly through states of higher energy; the slow cooling gives them more chances of finding configurations with lower internal energy than the initial one.

By analogy with this physical process, each step of the SA algorithm replaces the current solution by a random "nearby" solution, chosen with a probability that depends on the difference between the corresponding function values and on a global parameter T (called the *temperature*), that is gradually decreased during the process. The dependency is such that the current solution changes almost randomly when T is large, but increasingly "downhill" as T goes to zero. The allowance for "uphill" moves saves the method from becoming stuck at local minima—which are the bane of greedier methods.

The method was independently described by S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi in 1983, and by V. Černý in 1985. The method is an adaptation of the Metropolis-Hastings algorithm, a Monte Carlo method to generate sample states of a thermodynamic system, invented by N. Metropolis et al in 1953.

Theory:

Overview

In the simulated annealing (SA) method, each point s of the search space is analogous to a state of some physical system, and the function $E(s)$ to be minimized is analogous to the internal energy of the system in that state. The goal is to bring the system, from an arbitrary *initial state*, to a state with the minimum possible energy.

The basic iteration

At each step, the SA heuristic considers some neighbour s' of the current state s, and probabilistically decides between moving the system to state s' or staying in state s. The probabilities are chosen so that the system ultimately tends to move to states of lower energy. Typically this step is repeated until the system reaches a state that is good enough for the application, or until a given computation budget has been exhausted.

The neighbours of a state

The neighbours of each state (the *candidate moves*) are specified by the user, usually in an application-specific way. For example, in the traveling salesman problem, each state is typically defined as a particular *tour* (a permutation of the cities to be visited); and one could define the neighbours of a tour as those tours that can be obtained from it by exchanging any pair of consecutive cities.

Acceptance probabilities

The probability of making the transition from the current state s to a candidate new state s' is specified by an *acceptance probability function* $P(e,e',T)$, that depends on the energies $e = E(s)$ and $e' = E(s')$ of the two states, and on a global time-varying parameter T called the *temperature*.

One essential requirement for the probability function P is that it must be nonzero when $e' > e$, meaning that the system may move to the new state even when it is *worse* (has a higher energy) than the current one. It is this feature that prevents the method from becoming stuck in a *local minimum*—a state that is worse than the global minimum, yet better than any of its neighbours.

On the other hand, when T goes to zero, the probability $P(e,e',T)$ must tend to zero if $e' > e$, and to a positive value if $e' < e$. That way, for sufficiently small values of T, the system will increasingly favor moves that go "downhill" (to lower energy values), and avoid those that go "uphill". In particular, when T becomes 0, the procedure will reduce to the greedy algorithm—which makes the move only if it goes downhill.

In the original description of SA, the probability $P(e,e',T)$ was defined as 1 when $e' < e$ — i.e., the procedure always moved downhill when it found a way to do so, irrespective of the temperature. Many descriptions and implementations of SA still take this condition as part of the method's definition. However, this condition is not essential for the method to work, and one may argue that it is both counterproductive and contrary to its spirit.

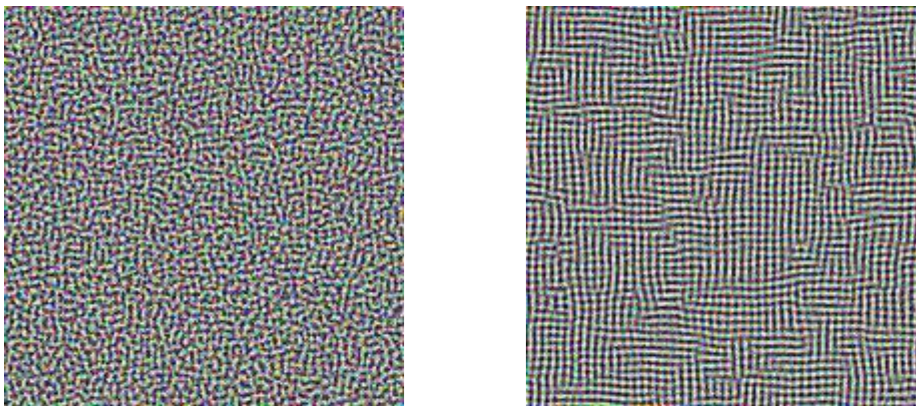
The P function is usually chosen so that the probability of accepting a move decreases when the difference $e' - e$ increases—that is, small uphill moves are more likely than

large ones. However, this requirement is not strictly necessary, provided that the above requirements are met.

Given these properties, the evolution of the state s depends crucially on the temperature T. Roughly speaking, the evolution of s is sensitive to coarser energy variations when T is large, and to finer variations when T is small.

The annealing schedule

Another essential feature of the SA method is that the temperature is gradually reduced as the simulation proceeds. Initially, T is set to a high value (or infinity), and it is decreased at each step according to some *annealing schedule*—which may be specified by the user, but must end with $T = 0$ towards the end of the allotted time budget. In this way, the system is expected to wander initially towards a broad region of the search space containing good solutions, ignoring small features of the energy function; then drift towards low-energy regions that become narrower and narrower; and finally move downhill according to the steepest descent heuristic.



Example illustrating the effect of cooling schedule on the performance of simulated annealing. The problem is to rearrange the pixels of an image so as to minimize a certain potential energy function, which causes similar colours to attract at short range and repel at slightly larger distance. The elementary moves swap two adjacent pixels. The images were obtained with fast cooling schedule (left) and slow cooling schedule (right), producing results similar to amorphous and crystalline solids, respectively.

It can be shown that for any given finite problem, the probability that the simulated annealing algorithm terminates with the global optimal solution approaches 1 as the annealing schedule is extended. This theoretical result, however, is not particularly helpful, since the time required to ensure a significant probability of success will usually exceed the time required for a complete search of the solution space.

Pseudocode

The following pseudocode implements the simulated annealing heuristic, as described above, starting from state `s0` and continuing to a maximum of `kmax` steps or until a state with energy `emax` or less is found. The call `neighbour(s)` should generate a randomly chosen neighbour of a given state s; the call `random()` should return a random value in the range [0,1). The annealing schedule is defined by the call `temp(r)`, which should yield the temperature to use, given the fraction `r` of the time budget that has been expended so far.

```
s := s0; e := E(s)                              // Initial state, energy.
sb := s; eb := e                                // Initial "best" solution
k := 0                                          // Energy evaluation
count.
while k < kmax and e > emax                     // While time remains &
not good enough:
  sn := neighbour(s)                            //   Pick some neighbour.
  en := E(sn)                                   //   Compute its energy.
  if en < eb then                               //   Is this a new best?
    sb := sn; eb := en                          //     Yes, save it.
  if P(e, en, temp(k/kmax)) > random() then    //   Should we move to it?
    s := sn; e := en                            //     Yes, change state.
  k := k + 1                                    //   One more evaluation
done
return sb                                       // Return the best
solution found.
```

Actually, the "pure" SA algorithm does not keep track of the best solution found so far: it does not use the variables `sb` and `eb`, it lacks the first **`if`** inside the loop, and, at the end, it returns the current state `s` instead of `sb`. While saving the best state is a standard optimization, that can be used in any metaheuristic, it breaks the analogy with physical annealing — since a physical system can "store" a single state only.

Saving the best state is not necessarily an improvement, since one may have to specify a smaller `kmax` in order to compensate for the higher cost per iteration. However, the step `sb := sn` happens only on a small fraction of the moves. Therefore, the optimization is usually worthwhile, even when state-copying is an expensive operation.

Selecting the parameters

In order to apply the SA method to a specific problem, one must specify the following parameters: the state space, the energy (goal) function `E()`, the candidate generator procedure `neighbour()`, the acceptance probability function `P()`, and the annealing schedule `temp()`. These choices can have a significant impact on the method's effectiveness. Unfortunately, there are no choices of these parameters that will be good for all problems, and there is no general way to find the best choices for a given problem. The following sections give some general guidelines.

Diameter of the search graph

Simulated annealing may be modeled as a random walk on a *search graph*, whose vertices are all possible states, and whose edges are the candidate moves. An essential requirement for the `neighbour()` function is that it must provide a sufficiently short path on this graph from the initial state to any state which may be the global optimum. (In other words, the diameter of the search graph must be small.) In the traveling salesman example above, for instance, the search space for $n = 20$ cities has $n! = 2432902008176640000$ (2.5 quintillion) states; yet the neighbour generator function that swaps two consecutive cities can get from any state (tour) to any other state in maximum $n(n - 1) / 2 = 190$ steps.

Transition probabilities

For each edge (s,s') of the search graph, one defines a *transition probability*, which is the probability that the SA algorithm will move to state s' when its current state is s. This probability depends on the current temperature as specified by `temp()`, by the order in which the candidate moves are generated by the `neighbour()` function, and by the acceptance probability function `P()`. (Note that the transition probability is not simply $P(e,e',T)$, because the candidates are tested serially.)

Acceptance probabilities

The specification of `neighbour()`, `P()`, and `temp()` is partially redundant. In practice, it's common to use the same acceptance function `P()` for many problems, and adjust the other two functions according to the specific problem.

In the formulation of the method by Kirkpatrick et al., the acceptance probability function $P(e,e',T)$ was defined as 1 if $e' < e$, and $\exp((e - e') / T)$ otherwise. This formula was superficially justified by analogy with the transitions of a physical system; it corresponds to the Metropolis-Hastings algorithm, in the case where the proposal distribution of Metropolis-Hastings is symmetric. However, this acceptance probability is often used for simulated annealing even when the `neighbour()` function, which is analogous to the proposal distribution in Metropolis-Hastings, is not symmetric, or not probabilistic at all. As a result, the transition probabilities of the simulated annealing algorithm do not correspond to the transitions of the analogous physical system, and the long-term distribution of states at a constant temperature T need not bear any resemblance to the thermodynamic equilibrium distribution over states of that physical system, at any temperature. Nevertheless, most descriptions of SA assume the original acceptance function, which is probably hard-coded in many implementations of SA.

Efficient candidate generation

When choosing the candidate generator `neighbour()`, one must consider that after a few iterations of the SA algorithm, the current state is expected to have much lower energy than a random state. Therefore, as a general rule, one should skew the generator towards

candidate moves where the energy of the destination state s' is likely to be similar to that of the current state. This heuristic (which is the main principle of the Metropolis-Hastings algorithm) tends to exclude "very good" candidate moves as well as "very bad" ones; however, the latter are usually much more common than the former, so the heuristic is generally quite effective.

In the traveling salesman problem above, for example, swapping two *consecutive* cities in a low-energy tour is expected to have a modest effect on its energy (length); whereas swapping two *arbitrary* cities is far more likely to increase its length than to decrease it. Thus, the consecutive-swap neighbour generator is expected to perform better than the arbitrary-swap one, even though the latter could provide a somewhat shorter path to the optimum (with $n - 1$ swaps, instead of $n(n - 1) / 2$).

A more precise statement of the heuristic is that one should try first candidate states s' for which $P(E(s),E(s'),T)$ is large. For the "standard" acceptance function P above, it means that $E(s') - E(s)$ is on the order of T or less. Thus, in the traveling salesman example above, one could use a `neighbour()` function that swaps two random cities, where the probability of choosing a city pair vanishes as their distance increases beyond T.

Barrier avoidance

When choosing the candidate generator `neighbour()` one must also try to reduce the number of "deep" local minima — states (or sets of connected states) that have much lower energy than all its neighbouring states. Such "closed catchment basins" of the energy function may trap the SA algorithm with high probability (roughly proportional to the number of states in the basin) and for a very long time (roughly exponential on the energy difference between the surrounding state and the bottom of the basin).

As a rule, it is impossible to design a candidate generator that will satisfy this goal and also prioritize candidates with similar energy. On the other hand, one can often vastly improve the efficiency of SA by relatively simple changes to the generator. In the traveling salesman problem, for instance, it is not hard to exhibit two tours A, B, with nearly equal lengths, such that (0) A is optimal, (1) every sequence of city-pair swaps that converts A to B goes through tours that are much longer than both, and (2) A can be transformed into B by flipping (reversing the order of) a set of consecutive cities. In this example, A and B lie in different "deep basins" if the generator performs only random pair-swaps; but they will be in the same basin if the generator performs random segment-flips.

Cooling schedule

The physical analogy that is used to justify SA assumes that the cooling rate is low enough for the probability distribution of the current state to be near thermodynamic equilibrium at all times. Unfortunately, the *relaxation time*—the time one must wait for the equilibrium to be restored after a change in temperature—strongly depends on the "topography" of the energy function and on the current temperature. In the SA algorithm,

the relaxation time also depends on the candidate generator, in a very complicated way. Note that all these parameters are usually provided as black box functions to the SA algorithm.

Therefore, in practice the ideal cooling rate cannot be determined beforehand, and should be empirically adjusted for each problem. The variant of SA known as thermodynamic simulated annealing tries to avoid this problem by dispensing with the cooling schedule, and instead automatically adjusting the temperature at each step based on the energy difference between the two states, according to the laws of thermodynamics.

Restarts

Sometimes it is better to move back to a solution that was significantly better rather than always moving from the current state. This is called *restarting*. To do this we set `s` and `e` to `sb` and `eb` and perhaps restart the annealing schedule. The decision to restart could be based on a fixed number of steps, or based on the current energy being too high from the best energy so far.

Related methods

- Quantum annealing uses "quantum fluctuations" instead of thermal fluctuations to get through high but thin barriers in the target function.

- Stochastic tunneling attempts to overcome the increasing difficulty simulated annealing runs have in escaping from local minima as the temperature decreases, by 'tunneling' through barriers.

- Tabu search normally moves to neighboring states of lower energy, but will take uphill moves when it finds itself stuck in a local minimum; and avoids cycles by keeping a "taboo list" of solutions already seen.

- Stochastic gradient descent runs many greedy searches from random initial locations.

- Genetic algorithms maintain a pool of solutions rather than just one. New candidate solutions are generated not only by "mutation" (as in SA), but also by "combination" of two solutions from the pool. Probabilistic criteria, similar to those used in SA, are used to select the candidates for mutation or combination, and for discarding excess solutions from the pool.

- Ant colony optimization (ACO) uses many ants (or agents) to traverse the solution space and find locally productive areas.

- The cross-entropy method (CE) generates candidates solutions via a parameterized probability distribution. The parameters are updated via cross-entropy minimization, so as to generate better samples in the next iteration.

- Harmony search mimics musicians in improvisation process where each musician plays a note for finding a best harmony all together.

- Stochastic optimization is an umbrella set of methods that includes simulated annealing and numerous other approaches.

Adaptive simulated annealing

Adaptive simulated annealing (ASA) is a variant of simulated annealing (SA) algorithm in which the algorithm parameters that control temperature schedule and random step selection are automatically adjusted according to algorithm progress. This makes the algorithm more efficient and less sensitive to user defined parameters than canonical SA. These are in the standard variant often selected on the basis of experience and experimentation (since optimal values are problem dependent), which represents a significant deficiency in practice.

The algorithm works by representing the parameters of the function to be optimized as continuous numbers, and as dimensions of a hypercube (N dimensional space). Some SA algorithms apply Gaussian moves to the state, while others have distributions permitting faster temperature schedules. Imagine the state as a point in a box and the moves as a rugby-ball shaped cloud around it. The temperature and the step size are adjusted so that all of the search space is sampled to a coarse resolution in the early stages, whilst the state is directed to favorable areas in the late stages.

Result: The program for simulated annealing is implemented.

Experiment No: 5

Aim: Write a program to implement 8 puzzle problem.

Introduction:

Write a program to solve the 8-puzzle problem (and its natural generalizations) using the A* search algorithm.

The problem

The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in order. You are permitted to slide blocks horizontally or vertically into the blank square. The following shows a sequence of legal moves from an initial board position (left) to the goal position (right).

```
    1  3        1     3        1  2  3        1  2  3        1  2  3
 4  2  5   =>   4  2  5   =>   4     5   =>   4  5      =>   4  5  6
 7  8  6        7  8  6        7  8  6        7  8  6        7  8

 initial                                                      goal
```

Theory:

Best-first search: We now describe an algorithmic solution to the problem that illustrates a general artificial intelligence methodology known as the A* search algorithm. We define a *state* of the game to be the board position, the number of moves made to reach the board position, and the previous state. First, insert the initial state (the initial board, 0 moves, and a null previous state) into a priority queue. Then, delete from the priority queue the state with the minimum priority, and insert onto the priority queue all neighboring states (those that can be reached in one move). Repeat this procedure until the state dequeued is the goal state. The success of this approach hinges on the choice of *priority function* for a state. We consider two priority functions:

- *Hamming priority function.* The number of blocks in the wrong position, plus the number of moves made so far to get to the state. Intuitively, states with a small number of blocks in the wrong position are close to the goal state, and we prefer states that have been reached using a small number of moves.
- *Manhattan priority function.* The sum of the distances (sum of the vertical and horizontal distance) from the blocks to their goal positions, plus the number of moves made so far to get to the state.

For example, the Hamming and Manhattan priorities of the initial state below are 5 and 10, respectively.

```
 8  1  3        1  2  3     1  2  3  4  5  6  7  8    1  2  3  4  5  6
 7  8
 4     2        4  5  6     ----------------------    -----------------
 -----
 7  6  5        7  8        1  1  0  0  1  1  0  1    1  2  0  0  2  2
 0  3

 initial          goal         Hamming = 5 + 0          Manhattan = 10
+ 0
```

We make a key oberservation: to solve the puzzle from a given state on the priority queue, the total number of moves we need to make (including those already made) is at least its priority, using either the Hamming or Manhattan priority function. (For Hamming priority, this is true because each block out of place must move at least once to reach its goal position. For Manhattan priority, this is true because each block must move its Manhattan distance from its goal position. Note that we do not count the blank tile when computing the Hamming or Manhattan priorities.)

Consequently, as soon as we dequeue a state, we have not only discovered a sequence of moves from the initial board to the board associated with the state, but one that makes the fewest number of moves. (Challenge for the mathematically inclined: prove this fact.)

A critical optimization: After implementing best-first search, you will notice one annoying feature: states corresponding to the same board position are enqueued on the priority queue many times. To prevent unnecessary exploration of useless states, when considering the neighbors of a state, don't enqueue the neighbor if its board position is the same as the previous state.

```
 8  1  3      8  1  3     8  1  3
 4     2      4  2        4     2
 7  6  5      7  6  5     7  6  5

 previous      state      disallow
```

The input will consist of the board dimension *N* followed by the *N*-by-*N* initial board position. The input format uses 0 to represent the blank square. As an example,

```
3
 0  1  3
 4  2  5
 7  8  6

    1  3
 4  2  5
 7  8  6

 1     3
```

```
 4  2  5
 7  8  6


 1  2  3
 4     5
 7  8  6

 1  2  3
 4  5
 7  8  6

 1  2  3
 4  5  6
 7  8

Number of states enqueued = 10
Number of moves = 4
```

Note that your program should work for arbitrary *N*-by-*N* boards (for any *N* greater than 1), even if it is too slow to solve some of them in a reasonable amount of time.

Detecting infeasible puzzles: Not all initial board positions can lead to the goal state. Modify your program to detect and report such situations.

```
3
 1  2  3
 4  5  6
 8  7  0
```

Hint: use the fact that board positions are divided into two equivalence classes with respect to reachability: (i) those that lead to the goal position and (ii) those that lead to the goal position if we modify the initial board by swapping any pair of adjacent (non-blank) blocks. There are two ways to apply the hint:

- Run the A* algorithm simultaneously on two puzzle instances - one with the initial board and one with the initial board modified by swapping a pair of adjacent (non-blank) blocks. Exactly one of the two will lead to the goal position.
- Derive a mathematical formula that tells you whether a board is solvable or not.

Deliverables: Organize your program in an appropriate number of data types. At a minimum, you are required to implement the following APIs. Though, you are free to add additional methods or data types (such as `State`).

```
public class Board {
   public Board(int[][] tiles)        // construct a board from an N-
by-N array of tiles
   public int hamming()               // return number of blocks out of
place
   public int manhattan()             // return sum of Manhattan
distances between blocks and goal
   public boolean equals(Object y)    // does this board equal y
   public Iterable<Board> neighbors() // return an Iterable of all
neighboring board positions
   public String toString()           // return a string representation
of the board

   //  test client
   public static void main(String[] args)
}


public class Solver {
   public Solver(Board initial)   // find a solution to the initial
board
   public boolean isSolvable()    // is the initial board solvable?
   public int moves()             // return min number of moves to
solve the initial board;
                                  // -1 if no such solution
   public String toString()       // return string representation of
solution (as described above)

   //  read puzzle instance from stdin and print solution to stdout (in
format above)
   public static void main(String[] args)
}
```

Result: The program is implemented to solve 8 puzzle problem.

Experiment No: 6

Aim: Write a program to implement A* program.

The A* (pronounced A-star) algorithm can be complicated for beginners. While there are many articles on the web that explain A*, most are written for people who understand the basics already.

Introduction:

The Search Area

Let's assume that we have someone who wants to get from point A to point B. Let's assume that a wall separates the two points. This is illustrated below, with green being the starting point A, and red being the ending point B, and the blue filled squares being the wall in between.

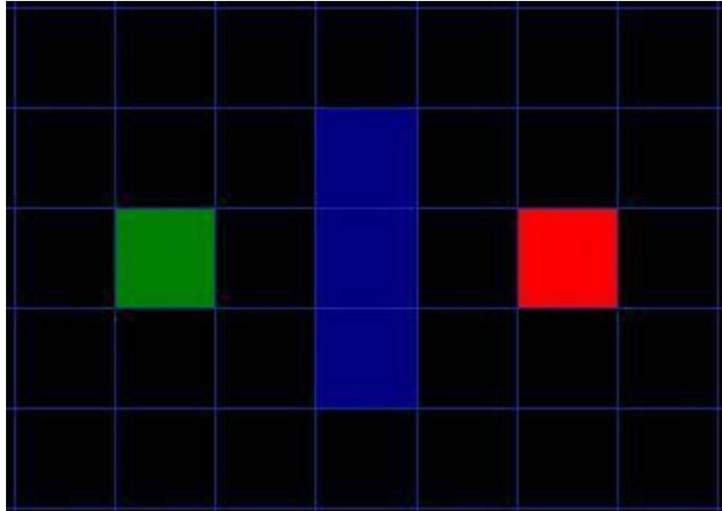


Fig. 1

The first thing you should notice is that we have divided our search area into a square grid. Simplifying the search area, as we have done here, is the first step in pathfinding. This particular method reduces our search area to a simple two dimensional array. Each item in the array represents one of the squares on the grid, and its status is recorded as walkable or unwalkable. The path is found by figuring out which squares we should take to get from A to B. Once the path is found, our person moves from the center of one square to the center of the next until the target is reached.

These center points are called "nodes". When you read about pathfinding elsewhere, you will often see people discussing nodes. Why not just call them squares? Because it is possible to divide up your pathfinding area into something other than squares. They could be rectangles, hexagons, triangles, or any shape, really. And the nodes could be placed

anywhere within the shapes – in the center or along the edges, or anywhere else. We are using this system, however, because it is the simplest.

Starting the Search

Once we have simplified our search area into a manageable number of nodes, as we have done with the grid layout above, the next step is to conduct a search to find the shortest path. We do this by starting at point A, checking the adjacent squares, and generally searching outward until we find our target.

We begin the search by doing the following:

1. Begin at the starting point A and add it to an "open list" of squares to be considered. The open list is kind of like a shopping list. Right now there is just one item on the list, but we will have more later. It contains squares that might fall along the path you want to take, but maybe not. Basically, this is a list of squares that need to be checked out.
2. Look at all the reachable or walkable squares adjacent to the starting point, ignoring squares with walls, water, or other illegal terrain. Add them to the open list, too. For each of these squares, save point A as its "parent square". This parent square stuff is important when we want to trace our path. It will be explained more later.
3. Drop the starting square A from your open list, and add it to a "closed list" of squares that you don't need to look at again for now.

At this point, you should have something like the following illustration. In this illustration, the dark green square in the center is your starting square. It is outlined in light blue to indicate that the square has been added to the closed list. All of the adjacent squares are now on the open list of squares to be checked, and they are outlined in light green. Each has a gray pointer that points back to its parent, which is the starting square.

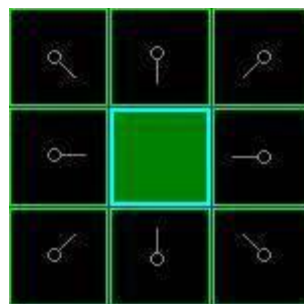


Fig. 2

Next, we choose one of the adjacent squares on the open list and more or less repeat the earlier process, as described below. But which square do we choose? The one with the lowest F cost.

Path Scoring

The key to determining which squares to use when figuring out the path is the following equation:

$F = G + H$

where

- G = the movement cost to move from the starting point A to a given square on the grid, following the path generated to get there.
- H = the estimated movement cost to move from that given square on the grid to the final destination, point B. This is often referred to as the heuristic, which can be a bit confusing. The reason why it is called that is because it is a guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). You are given one way to calculate H in this tutorial, but there are many others that you can find in other articles on the web.

Our path is generated by repeatedly going through our open list and choosing the square with the lowest F score. This process will be described in more detail a bit further in the article. First let's look more closely at how we calculate the equation.

As described above, G is the movement cost to move from the starting point to the given square using the path generated to get there. In this example, we will assign a cost of 10 to each horizontal or vertical square moved, and a cost of 14 for a diagonal move. We use these numbers because the actual distance to move diagonally is the square root of 2 (don't be scared), or roughly 1.414 times the cost of moving horizontally or vertically. We use 10 and 14 for simplicity's sake. The ratio is about right, and we avoid having to calculate square roots and we avoid decimals. This isn't just because we are dumb and don't like math. Using whole numbers like these is a lot faster for the computer, too. As you will soon find out, pathfinding can be very slow if you don't use short cuts like these.

Since we are calculating the G cost along a specific path to a given square, the way to figure out the G cost of that square is to take the G cost of its parent, and then add 10 or 14 depending on whether it is diagonal or orthogonal (non-diagonal) from that parent square. The need for this method will become apparent a little further on in this example, as we get more than one square away from the starting square.

H can be estimated in a variety of ways. The method we use here is called the Manhattan method, where you calculate the total number of squares moved horizontally and vertically to reach the target square from the current square, ignoring diagonal movement, and ignoring any obstacles that may be in the way. We then multiply the total by 10, our cost for moving one square horizontally or vertically. This is (probably) called the Manhattan method because it is like calculating the number of city blocks from one place to another, where you can't cut across the block diagonally.

Reading this description, you might guess that the heuristic is merely a rough estimate of the remaining distance between the current square and the target "as the crow flies." This isn't the case. We are actually trying to estimate the remaining distance along the path (which is usually farther). The closer our estimate is to the actual remaining distance, the faster the algorithm will be. If we overestimate this distance, however, it is not guaranteed to give us the shortest path. In such cases, we have what is called an "inadmissible heuristic."

Technically, in this example, the Manhattan method is inadmissible because it slightly overestimates the remaining distance. But we will use it anyway because it is a lot easier to understand for our purposes, and because it is only a slight overestimation. On the rare occasion when the resulting path is not the shortest possible, it will be nearly as short.

F is calculated by adding G and H. The results of the first step in our search can be seen in the illustration below. The F, G, and H scores are written in each square. As is indicated in the square to the immediate right of the starting square, F is printed in the top left, G is printed in the bottom left, and H is printed in the bottom right.

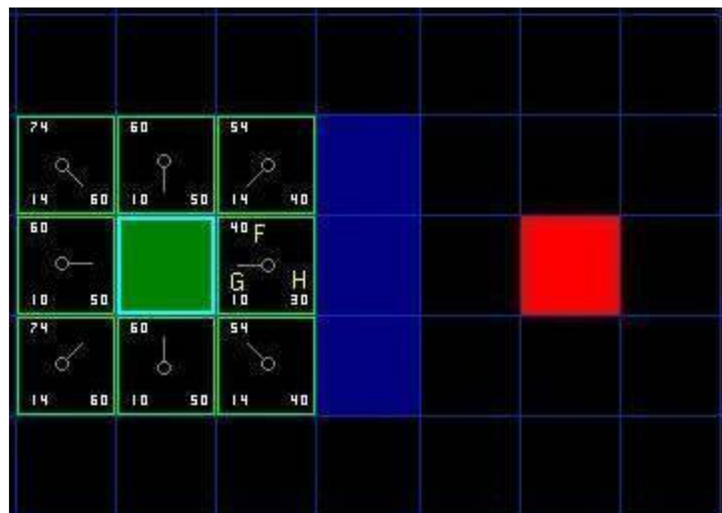


Fig. 3

So let’s look at some of these squares. In the square with the letters in it, G = 10. This is because it is just one square from the starting square in a horizontal direction. The squares immediately above, below, and to the left of the starting square all have the same G score of 10. The diagonal squares have G scores of 14.

The H scores are calculated by estimating the Manhattan distance to the red target square, moving only horizontally and vertically and ignoring the wall that is in the way. Using this method, the square to the immediate right of the start is 3 squares from the red square, for a H score of 30. The square just above this square is 4 squares away (remember, only move horizontally and vertically) for an H score of 40. You can probably see how the H scores are calculated for the other squares.

The F score for each square, again, is simply calculated by adding G and H together.

Continuing the Search

To continue the search, we simply choose the lowest F score square from all those that are on the open list. We then do the following with the selected square:

4) Drop it from the open list and add it to the closed list.

5) Check all of the adjacent squares. Ignoring those that are on the closed list or unwalkable (terrain with walls, water, or other illegal terrain), add squares to the open list if they are not on the open list already. Make the selected square the "parent" of the new squares.

6) If an adjacent square is already on the open list, check to see if this path to that square is a better one. In other words, check to see if the G score for that square is lower if we use the current square to get there. If not, don't do anything.
   On the other hand, if the G cost of the new path is lower, change the parent of the adjacent square to the selected square (in the diagram above, change the direction of the pointer to point at the selected square). Finally, recalculate both the F and G scores of that square. If this seems confusing, you will see it illustrated below.

Okay, so let's see how this works. Of our initial 9 squares, we have 8 left on the open list after the starting square was switched to the closed list. Of these, the one with the lowest F cost is the one to the immediate right of the starting square, with an F score of 40. So we select this square as our next square. It is highlight in blue in the following illustration.

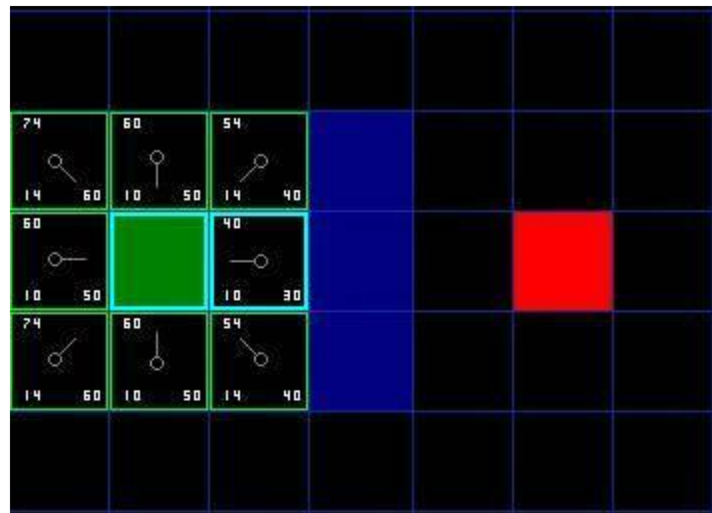


Fig. 4

First, we drop it from our open list and add it to our closed list (that's why it's now highlighted in blue). Then we check the adjacent squares. Well, the ones to the immediate right of this square are wall squares, so we ignore those. The one to the immediate left is the starting square. That's on the closed list, so we ignore that, too.

The other four squares are already on the open list, so we need to check if the paths to those squares are any better using this square to get there, using G scores as our point of reference. Let's look at the square right above our selected square. Its current G score is 14. If we instead went through the current square to get there, the G score would be equal to 20 (10, which is the G score to get to the current square, plus 10 more to go vertically to the one just above it). A G score of 20 is higher than 14, so this is not a better path. That should make sense if you look at the diagram. It's more direct to get to that square from the starting square by simply moving one square diagonally to get there, rather than moving horizontally one square, and then vertically one square.

When we repeat this process for all 4 of the adjacent squares already on the open list, we find that none of the paths are improved by going through the current square, so we don't change anything. So now that we looked at all of the adjacent squares, we are done with this square, and ready to move to the next square.

So we go through the list of squares on our open list, which is now down to 7 squares, and we pick the one with the lowest F cost. Interestingly, in this case, there are two squares with a score of 54. So which do we choose? It doesn't really matter. For the purposes of speed, it can be faster to choose the <u>last</u> one you added to the open list. This biases the search in favor of squares that get found later on in the search, when you have gotten closer to the target. But it doesn't really matter. (Differing treatment of ties is why two versions of A* may find different paths of equal length.)

So let's choose the one just below, and to the right of the starting square, as is shown in the following illustration.

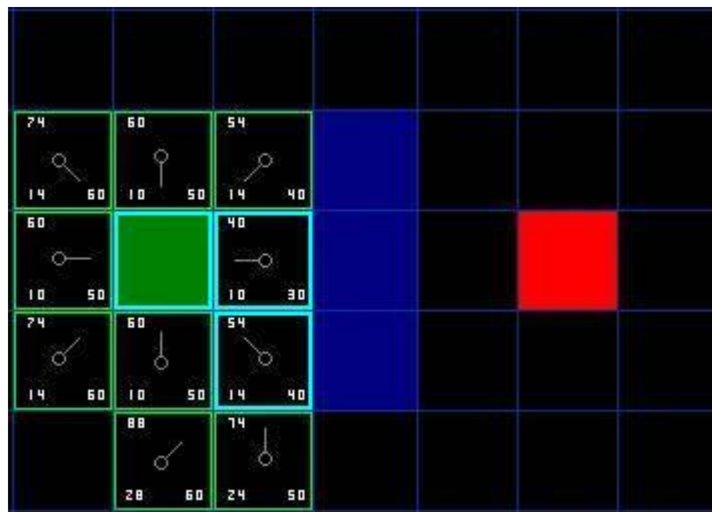


Fig. 5

This time, when we check the adjacent squares we find that the one to the immediate right is a wall square, so we ignore that. The same goes for the one just above that. We also ignore the square just below the wall. Why? Because you can't get to that square directly from the current square without cutting across the corner of the nearby wall. You really need to go down first and then move over to that square, moving around the corner in the process. (Note: This rule on cutting corners is optional. Its use depends on how your nodes are placed.)

That leaves five other squares. The other two squares below the current square aren't already on the open list, so we add them and the current square becomes their parent. Of the other three squares, two are already on the closed list (the starting square, and the one just above the current square, both highlighted in blue in the diagram), so we ignore them. And the last square, to the immediate left of the current square, is checked to see if the G score is any lower if you go through the current square to get there. No dice. So we're done and ready to check the next square on our open list.

We repeat this process until we add the target square to the closed list, at which point it looks something like the illustration below.

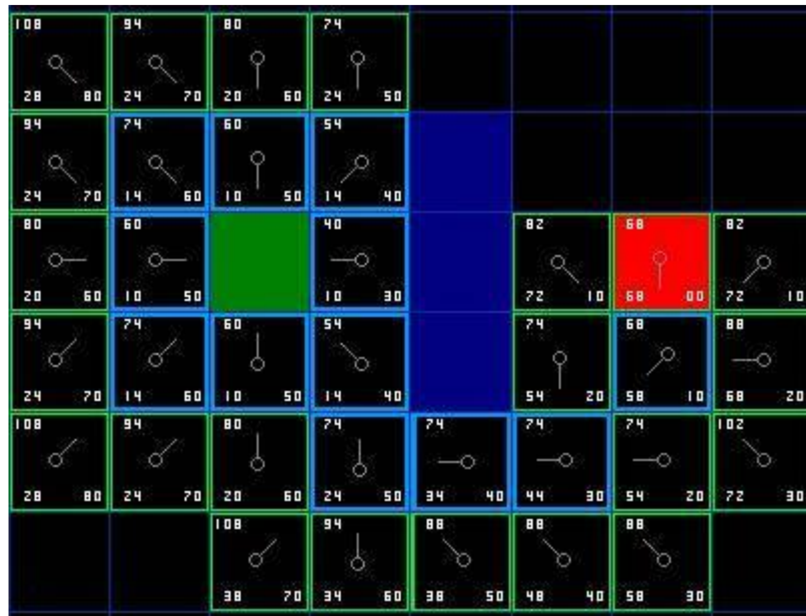


Fig. 6

Note that the parent square for the square two squares below the starting square has changed from the previous illustration. Before it had a G score of 28 and pointed back to the square above it and to the right. Now it has a score of 20 and points to the square just above it. This happened somewhere along the way on our search, where the G score was checked and it turned out to be lower using a new path – so the parent was switched and the G and F scores were recalculated. While this change doesn't seem too important in this example, there are plenty of possible situations where this constant checking will make all the difference in determining the best path to your target.

So how do we determine the path? Simple, just start at the red target square, and work backwards moving from one square to its parent, following the arrows. This will eventually take you back to the starting square, and that's your path. It should look like the following illustration. Moving from the starting square A to the destination square B is simply a matter of moving from the center of each square (the node) to the center of the next square on the path, until you reach the target.

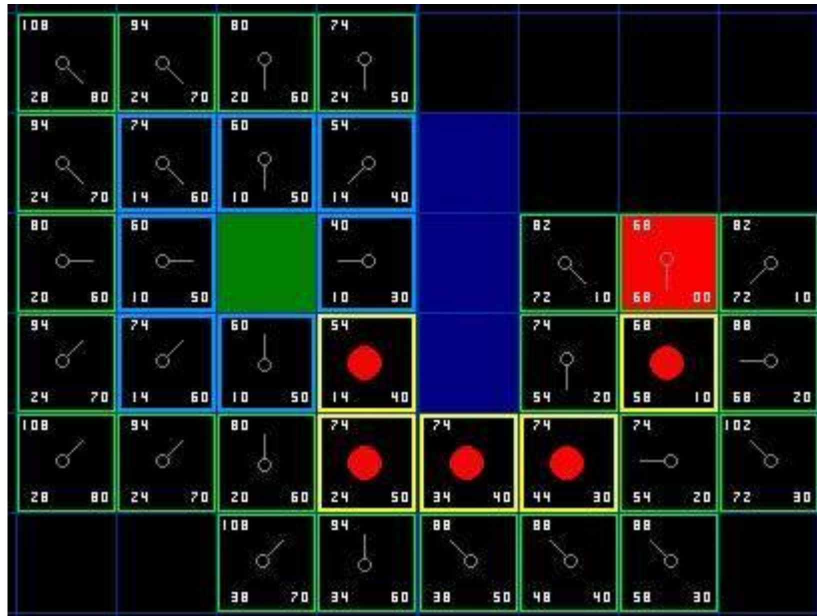


Fig. 7

Summary of the A Method*

Okay, now that you have gone through the explanation, let's lay out the step-by-step method all in one place:

1) Add the starting square (or node) to the open list.

2) Repeat the following:

a) Look for the lowest F cost square on the open list. We refer to this as the current square.

b) Switch it to the closed list.

c) For each of the 8 squares adjacent to this current square …

- If it is not walkable or if it is on the closed list, ignore it. Otherwise do the following.
- If it isn't on the open list, add it to the open list. Make the current square the parent of this square. Record the F, G, and H costs of the square.

- If it is on the open list already, check to see if this path to that square is better, using G cost as the measure. A lower G cost means that this is a better path. If so, change the parent of the square to the current square, and recalculate the G and F scores of the square. If you are keeping your open list sorted by F score, you may need to resort the list to account for the change.

d) Stop when you:

- Add the target square to the closed list, in which case the path has been found (see note below), or
- Fail to find the target square, and the open list is empty. In this case, there is no path.

3) Save the path. Working backwards from the target square, go from each square to its parent square until you reach the starting square. That is your path.

Notes on Implementation

Now that you understand the basic method, here are some additional things to think about when you are writing your own program.

1. Other Units (collision avoidance): If you happen to look closely at example code, you will notice that it completely ignores other units on the screen. The units pass right through each other. Depending on the game, this may be acceptable or it may not. If you want to consider other units in the pathfinding algorithm and have them move around one another, we suggest that you only consider units that are either stopped or adjacent to the pathfinding unit at the time the path is calculated, treating their current locations as unwalkable. For adjacent units that are moving, you can discourage collisions by penalizing nodes that lie along their respective paths, thereby encouraging the pathfinding unit to find an alternate route (described more under #2).

If you choose to consider other units that are moving and not adjacent to the pathfinding unit, you will need to develop a method for predicting where they will be at any given point in time so that they can be dodged properly. Otherwise you will probably end up with strange paths where units zig-zag to avoid other units that aren't there anymore.

You will also, of course, need to develop some collision detection code because no matter how good the path is at the time it is calculated, things can change over time. When a collision occurs a unit must either calculate a new path or, if the other unit is moving and it is not a head-on collision, wait for the other unit to step out of the way before proceeding with the current path.

2. Variable Terrain Cost: In this tutorial and accompanying program, terrain is just one of two things – walkable or unwalkable. But what if you have terrain that is walkable, but at a higher movement cost? Swamps, hills, stairs in a dungeon, etc. – these are all

examples of terrain that is walkable, but at a higher cost than flat, open ground. Similarly, a road might have a lower movement cost than the surrounding terrain.

This problem is easily handled by adding the terrain cost in when you are calculating the G cost of any given node. Simply add a bonus cost to such nodes. The A* pathfinding algorithm is already written to find the lowest cost path and should handle this easily. This simple example describes, when terrain is only walkable or unwalkable, A* will look for the shortest, most direct path. But in a variable-cost terrain environment, the least cost path might involve traveling a longer distance – like taking a road around a swamp rather than plowing straight through it.

An interesting additional consideration is something the professionals call "influence mapping." Just as with the variable terrain costs described above, you could create an additional point system and apply it to paths for AI purposes. Imagine that you have a map with a bunch of units defending a pass through a mountain region. Every time the computer sends somebody on a path through that pass, it gets whacked. If you wanted, you could create an influence map that penalized nodes where lots of carnage is taking place. This would teach the computer to favor safer paths, and help it avoid dumb situations where it keeps sending troops through a particular path, just because it is shorter (but also more dangerous).

Yet another possible use is penalizing nodes that lie along the paths of nearby moving units. One of the downsides of A* is that when a group of units all try to find paths to a similar location, there is usually a significant amount of overlap, as one or more units try to take the same or similar routes to their destinations. Adding a penalty to nodes already 'claimed' by other units will help ensure a degree of separation, and reduce collisions. Don't treat such nodes as unwalkable, however, because you still want multiple units to be able to squeeze through tight passageways in single file, if necessary. Also, you should only penalize the paths of units that are near the pathfinding unit, not all paths, or you will get strange dodging behavior as units avoid paths of units that are nowhere near them at the time. Also, you should only penalize path nodes that lie along the current and future portion of a path, not previous path nodes that have already been visited and left behind.

3. Handling Unexplored Areas: Have you ever played a PC game where the computer always knows exactly what path to take, even though the map hasn't been explored yet? Depending upon the game, pathfinding that is too good can be unrealistic. Fortunately, this is a problem that is can be handled fairly easily.

The answer is to create a separate "knownWalkability" array for each of the various players and computer opponents (each player, not each unit -- that would require a lot more computer memory). Each array would contain information about the areas that the player has explored, with the rest of the map assumed to be walkable until proven otherwise. Using this approach, units will wander down dead ends and make similar wrong choices until they have learned their way around. Once the map is explored, however, pathfinding would work normally.

4. Smoother Paths: While A* will automatically give you the shortest, lowest cost path, it won't automatically give you the smoothest looking path. Take a look at the final path calculated in our example (in Figure 7). On that path, the very first step is below, and to the right of the starting square. Wouldn't our path be smoother if the first step was instead the square directly below the starting square?

There are several ways to address this problem. While you are calculating the path you could penalize nodes where there is a change of direction, adding a penalty to their G scores. Alternatively, you could run through your path after it is calculated, looking for places where choosing an adjacent node would give you a path that looks better.

5. Non-square Search Areas: In our example, we used a simple 2D square layout. You don't need to use this approach. You could use irregularly shaped areas. Think of the board game Risk, and the countries in that game. You could devise a pathfinding scenario for a game like that. To do this, you would need to create a table for storing which countries are adjacent to which, and a G cost associated with moving from one country to the next. You would also need to come up with a method for estimating H. Everything else would be handled the same as in the above example. Instead of using adjacent squares, you would simply look up the adjacent countries in the table when adding new items to your open list.

Similarly, you could create a waypoint system for paths on a fixed terrain map. Waypoints are commonly traversed points on a path, perhaps on a road or key tunnel in a dungeon. As the game designer, you could pre-assign these waypoints. Two waypoints would be considered "adjacent" to one another if there were no obstacles on the direct line path between them. As in the Risk example, you would save this adjacency information in a lookup table of some kind and use it when generating your new open list items. You would then record the associated G costs (perhaps by using the direct line distance between the nodes) and H costs (perhaps using a direct line distance from the node to the goal). Everything else would proceed as usual.

6. Some Speed Tips: As you develop your own A* program, or adapt the one we wrote, you will eventually find that pathfinding is using a hefty chunk of your CPU time, particularly if you have a decent number of pathfinding units on the board and a reasonably large map. If you read the stuff on the net, you will find that this is true even for the professionals who design games like Starcraft or Age of Empires. If you see things start to slow down due to pathfinding, here are some ideas that may speed things up:

- Consider a smaller map or fewer units.
- Never do path finding for more than a few units at a time. Instead put them in a queue and spread them out over several game cycles. If your game is running at, say, 40 cycles per second, no one will ever notice. But they will notice if the game seems to slow down every once in a while when a bunch of units are all calculating paths at the same time.

- Consider using larger squares (or whatever shape you are using) for your map. This reduces the total number of nodes searched to find the path. If you are ambitious, you can devise two or more pathfinding systems that are used in different situations, depending upon the length of the path. This is what the professionals do, using large areas for long paths, and then switching to finer searches using smaller squares/areas when you get close to the target.
- For longer paths, consider devising precalculated paths that are hardwired into the game.
- Consider pre-processing your map to figure out what areas are inaccessible from the rest of the map. we call these areas "islands." In reality, they can be islands or any other area that is otherwise walled off and inaccessible. One of the downsides of A* is that if you tell it to look for paths to such areas, it will search the whole map, stopping only when every accessible square/node has been processed through the open and closed lists. That can waste a lot of CPU time. It can be prevented by predetermining which areas are inaccessible (via a flood-fill or similar routine), recording that information in an array of some kind, and then checking it before beginning a path search.
- In a crowded, maze-like environment, consider tagging nodes that don't lead anywhere as dead ends. Such areas can be manually pre-designated in your map editor or, if you are ambitious, you could develop an algorithm to identify such areas automatically. Any collection of nodes in a given dead end area could be given a unique identifying number. Then you could safely ignore all dead ends when pathfinding, pausing only to consider nodes in a dead end area if the starting location or destination happen to be in the particular dead end area in question.

7. Maintaining the Open List: This is actually one of the most time consuming elements of the A* pathfinding algorithm. Every time you access the open list, you need to find the square that has the lowest F cost. There are several ways you could do this. You could save the path items as needed, and simply go through the whole list each time you need to find the lowest F cost square. This is simple, but really slow for long paths. This can be improved by maintaining a sorted list and simply grabbing the first item off the list every time you need the lowest F-cost square. This will work reasonably well for small maps, but it isn’t the fastest solution. Serious A* programmers who want real speed use something called a binary heap, and this is what we use in this code. This approach will be at least 2-3 times as fast in most situations, and geometrically faster (10+ times as fast) on longer paths.

Another possible bottleneck is the way you clear and maintain your data structures between pathfinding calls. we personally prefer to store everything in arrays. While nodes can be generated, recorded and maintained in a dynamic, object-oriented manner, we find that the amount of time needed to create and delete such objects adds an extra, unnecessary level of overhead that slows things down. If you use arrays, however, you will need to clean things up between calls. The last thing you will want to do in such cases is spend time zero-ing everything out after a pathfinding call, especially if you have a large map.

We can avoid this overhead by creating a 2d array called whichList(x,y) that designates each node on map as either on the open list or closed list. After pathfinding attempts, we do not zero out this array. Instead we reset the values of onClosedList and onOpenList in every pathfinding call, incrementing both by +5 or something similar on each path finding attempt. This way, the algorithm can safely ignore as garbage any data left over from previous pathfinding attempts. we also store values like F, G and H costs in arrays. In this case, we simply write over any pre-existing values and don't bother clearing the arrays.

Storing data in multiple arrays consumes more memory, though, so there is a trade off. Ultimately, you should use whatever method you are most comfortable with.

8. Dijkstra's Algorithm: While A* is generally considered to be the best pathfinding algorithm (see rant above), there is at least one other algorithm that has its uses - Dijkstra's algorithm. Dijkstra's is essentially the same as A*, except there is no heuristic (H is always 0). Because it has no heuristic, it searches by expanding out equally in every direction. As you might imagine, because of this Dijkstra's usually ends up exploring a much larger area before the target is found. This generally makes it slower than A*.

So why use it? Sometimes we don't know where our target destination is. Say you have a resource-gathering unit that needs to go get some resources of some kind. It may know where several resource areas are, but it wants to go to the closest one. Here, Dijkstra's is better than A* because we don't know which one is closest. Our only alternative is to repeatedly use A* to find the distance to each one, and then choose that path. There are probably countless similar situations where we know the kind of location we might be searching for, want to find the closest one, but not know where it is or which one might be closest.

The pseudo code for A algorithm is:*

1	Create a node containing the goal state node_goal
2	Create a node containing the start state node_start
3	Put node_start on the open list
4	while the OPEN list is not empty
5	{
6	Get the node off the open list with the lowest f and call it node_current
7	if node_current is the same state as node_goal we have found the solution; break from the while loop
8	Generate each state node_successor that can come after node_current
9	for each node_successor of node_current
10	{
11	Set the cost of node_successor to be the cost of node_current plus the cost to get to node_successor from node_current
12	find node_successor on the OPEN list
13	if node_successor is on the OPEN list but the existing one is as good or better then discard this successor and continue
14	if node_successor is on the CLOSED list but the existing one is as good or better then discard this successor and continue
15	Remove occurrences of node successor from OPEN and CLOSED
16	Set the parent of node_successor to node_current
17	Set h to be the estimated distance to node_goal (Using the heuristic function)
18	Add node_successor to the OPEN list
19	}
20	Add node_current to the CLOSED list

Result: The A* algorithm is implemented for finding the least cost path from a given initial node to one goal node.

Experiment No: 7

Aim: Write a program to implement Hill climbing Algorithm

In computer science, *hill climbing* is a mathematical optimization technique which belongs to the family of local search. It is relatively simple to implement, making it a popular first choice. Although more advanced algorithms may give better results, in some situations hill climbing works just as well.

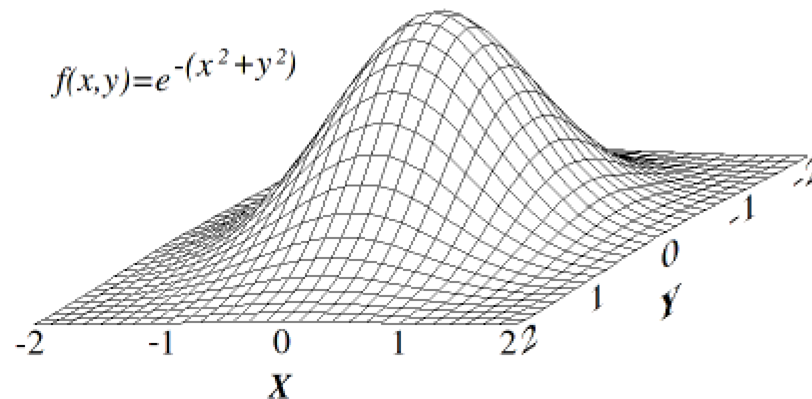
Hill climbing can be used to solve problems that have many solutions, some of which are better than others. It starts with a random (potentially poor) solution, and iteratively makes small changes to the solution, each time improving it a little. When the algorithm cannot see any improvement anymore, it terminates. Ideally, at that point the current solution is close to optimal, but it is not guaranteed that hill climbing will ever come close to the optimal solution.

For example, hill climbing can be applied to the traveling salesman problem. It is easy to find a solution that visits all the cities but will be very poor compared to the optimal solution. The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited. Eventually, a much better route is obtained.

Hill climbing is used widely in artificial intelligence, for reaching a goal state from a starting node. Choice of next node and starting node can be varied to give a list of related algorithms.

Mathematical description

Hill climbing attempts to maximize (or minimize) a function $f(x)$, where x are discrete states. These states are typically represented by vertices in a graph, where edges in the graph encode nearness or similarity of a graph. Hill climbing will follow the graph from vertex to vertex, always locally increasing (or decreasing) the value of f, until a local maximum (or local minimum) x_m is reached. Hill climbing can also operate on a continuous space: in that case, the algorithm is called gradient ascent (or gradient descent if the function is minimized).*.



<u>Variants</u>

In *simple hill climbing*, the first closer node is chosen, whereas in *steepest ascent hill climbing* all successors are compared and the closest to the solution is chosen. Both forms fail if there is no closer node, which may happen if there are local maxima in the search space which are not solutions. Steepest ascent hill climbing is similar to best-first search, which tries all possible extensions of the current path instead of only one.

Stochastic hill climbing does not examine all neighbors before deciding how to move. Rather, it selects a neighbour at random, and decides (based on the amount of improvement in that neighbour) whether to move to that neighbour or to examine another.

Random-restart hill climbing is a meta-algorithm built on top of the hill climbing algorithm. It is also known as *Shotgun hill climbing*. It iteratively does hill-climbing, each time with a random initial condition x_0. The best x_m is kept: if a new run of hill climbing produces a better x_m than the stored state, it replaces the stored state.

Random-restart hill climbing is a surprisingly effective algorithm in many cases. It turns out that it is often better to spend CPU time exploring the space, than carefully optimizing from an initial condition.

<u>Problems</u>

Local maxima

A problem with hill climbing is that it will find only local maxima. Unless the heuristic is convex, it may not reach a global maximum. Other local search algorithms try to overcome this problem such as stochastic hill climbing, random walks and simulated annealing. This problem of hill climbing can be solved by using random hill climbing search technique.

Ridges

A ridge is a curve in the search place that leads to a maximum, but the orientation of the ridge compared to the available moves that are used to climb is such that each move will lead to a smaller point. In other words, each point on a ridge looks to the algorithm like a local maximum, even though the point is part of a curve leading to a better optimum.

Plateau

Another problem with hill climbing is that of a plateau, which occurs when we get to a "flat" part of the search space, i.e. we have a path where the heuristics are all very close together. This kind of flatness can cause the algorithm to cease progress and wander aimlessly.

Pseudocode

Hill Climbing Algorithm

```
currentNode = startNode;
loop do
   L = NEIGHBORS(currentNode);
   nextEval = -INF;
   nextNode = NULL;
   for all x in L
      if (EVAL(x) > nextEval)
           nextNode = x;
           nextEval = EVAL(x);
   if nextEval <= EVAL(currentNode)
      //Return current node since no better neighbors exist
      return currentNode;
   currentNode = nextNode;
```

Result: The Hill Climbing algorithm is implemented. The algorithm terminates when goal state is reached from the starting node.

Experiment No: 8

Aim: To study JESS expert system

Theory:

1. Introduction

1.1. Compatibility

Jess is compatible with all versions of Java starting with Java 1.1. In particular, this includes JDK 1.2 (or "Java 2" as it is now known.) All versions of Jess later than version 4.x are no longer compatible with Java 1.0.

1.2. Mailing List

There is a Jess email discussion list you can join. To get information about the jess-users list, send a message to majordomo@sandia.gov containing the text

```
help
info jess-users
end
```

as the body of the message. There is an archive of the list at http://www.mail-archive.com/jess-users@sandia.gov .

1.3. Bugs

This is the final release of Jess 5.2. There may still be bugs. Please report any that you find at ejfried@ca.sandia.gov so we can fix them for a later release.

1.4. Assumptions

Jess is a programmer's library. The library itself is written in Java. The library serves as an interpreter for another language, which we will refer to in this document as the Jess language. The Jess language is very similar to the language defined by the CLIPS expert system shell, which in turn is a highly specialized form of LISP.
Therefore, we are going to assume that you, the reader, are a programmer who will be using either one or both of these languages. we will assume that all readers have at least a minimal facility with Java. You must have a Java compiler and runtime system, and you must know how to use it at least in a simple way. You should know how to use it to

- compile a collection of Java source files
- run a Java application
- deal with configuration issues like the CLASSPATH variable

If you do not have at least this passing familiarity with a Java environment, then may we suggest you purchase an introductory book on the topic.

For those readers who are going to program in the Jess language, we assume general familiarity with the principles of programming. we will describe the entire Jess language, so no familiarity with LISP is required. Furthermore, we will attempt to describe, to the extent possible, the important concepts of rule-based systems as they apply to Jess. Again, though, we will assume that the reader has some familiarity with these concepts and more. If you are unfamiliar with rule-based systems, you may want to purchase a text on this topic as well.

Many readers will want to extend Jess' capabilities by either adding commands (written in Java) to the Jess language, or embedding the Jess library in a Java application. Others will want to use the Jess language's Java integration capabilities to call Java functions from Jess language programs. In sections of this document targeted towards these readers, we will assume moderate knowledge of Java programming. we will not teach any aspects of the Java language. The interested reader is once again referred to your local bookstore.

1.5. Getting ready

1.5.1. Unpacking the Distribution

If you download Jess for UNIX, you can extract the files using tar and gunzip:

```
gunzip Jess52.tgz
tar xf Jess52.tar
```

If you downloaded Jess for Windows, you get a .zip file which should be unzipped using a Win32-aware unzip program like WinZip. Don't use PKUNZIP since it cannot handle long file names.

When Jess is unpacked, you should have a directory named `Jess52/`. Inside this directory should be the following files and subdirectories:

`docs/`	This documentation
`jess/`	If you have a source distribution, this is directory containing the `jess` package. There are many source files in here that implement Jess's inference engine. Others implement a number of Jess GUIs and command-line interfaces. `Main.java` implements the Jess command-line interface. `Console.java` is a very simple GUI console for Jess; `ConsoleApplet.java` is an applet version of the same. If you have a binary distribution, this directory contains only the Java examples.
`jess.jar` *(optional)*	The compiled .class files that make up the Jess library and applications. Only the binary distribution contains this file.
`examples/`	A directory of tiny example Jess files.
`jess/examples`	A directory of more complicated examples, containing example Java source files. Some of these require Java 2 (JDK 1.2 or later.)
`Makefile`	A simple makefile for Jess.

1.5.2. Compiling Jess

If you have a source distribution, Jess comes as a set of Java source files. You'll need to compile them first. If you have a `make` utility (any UNIX-like `make`; you could use the CygWin environment on Windows), you can just run `make` and the enclosed makefile will

build everything. You will have to edit it a bit first to specify the path to your Java compiler. Otherwise, on Windows, the commands:

```
javac -d . jess\*.java
javac -d . jess\awt\*.java
javac -d . jess\factory\*.java
```

would work just fine, given that you have a Java compiler named javac (as in Sun's JDK) and that `Jess52/` is your current directory. On UNIX, you'll need to change the backslashes to forward slashes. If you have problems, be sure that if you have the `CLASSPATH` environment variable set, it includes '.', the current directory. *Don't* try to compile from inside the `Jess52/jess/` directory; it won't work. You must use a Java 1.1 or later compiler to compile Jess. The resulting code will run on a Java 1.1 or later VM. Jess works great with Java 1.2.

There are a number of optional example source files in the subdirectories `Jess52/jess/examples/` that aren't compiled if you follow the instructions above. Note that some of these examples require a Java 2 (JDK 1.2 or later) compiler and runtime.

You can issue a set of commands like:

```
javac -d . jess\examples\simple\*.java
javac -d . jess\examples\pumps\*.java
javac -d . jess\examples\xfer\*.java
```

Again, *don't* set your current directory to, for example, Jess52/jess/examples/pumps/ to compile the pumps example: it will *not* work. The compiler will report all sorts of errors about classes not being found and the jess package not being found. Compile everything from the Jess52 directory. we can't stress this enough: this is by far the most common problem people have in getting started with Jess!

1.5.3. Jess Example Programs

There are a few trivial example programs (in the examples/ directory) that you can try to confirm that you have properly compiled Jess. including `fullmab.clp`, `zebra.clp`, and `wordgame.clp`. `fullmab.clp` is a version of the classic Monkey and Bananas problem. To run it yourself from the command line, just type (if you have a source distribution)

```
java jess.Main examples/fullmab.clp
```

or (if you have the binary distribution)

```
java -jar jess.jar examples/fullmab.clp
```

and the problem should run, producing a few screens of output. Any file of Jess code can be run this way. Many simple CLIPS programs will also run unchanged in Jess. Note that giving Jess a file name on the command line is like using the `batch` command in CLIPS. Therefore, you generally need to make sure that the file ends with:

```
(reset)
(run)
```

or no rules will fire. The `zebra.clp` and `wordgame.clp` programs are two classic CLIPS examples selected to show how Jess deals with tough situations. These examples both generate huge numbers of partial pattern matches, so they are slow and use up a lot of memory. Other examples include `sticks.clp` (an interactive game), `frame.clp` (a demo of building a graphical interface using Jess's Java integration capabilities), and `animal.clp`. Note that `animal.clp` is hardwired to expect a data file to exist in a subdirectory examples/ of the current directory.

In the jess/examples/* subdirectories, you will find some more complex examples, all of which contain both Java and Jess code. As such, these are generally examples of how to tie Jess and Java together. The *Pumps* examples is a full working program that demonstrates how Jess rules can react to the properties of Java Beans.

1.5.4. Command-line Interface

Jess has an interactive command-line interface. Just type `java jess.Main` (or java -jar jess.jar, if you have the binary-only distribution) to get a `Jess>` prompt. To execute a file of CLIPS code from the command prompt, use the `batch` command:

```
Jess> (batch examples/sticks.clp)
Who moves first (Computer: c Human: h)?
```

Note that in the preceding example, you type what follows the `Jess>` prompt, and Jess responds with the text in bold. we will follow this convention throughout this manual.

You can use the Jess `system` command to invoke an editor from the Jess command line to edit a file of Jess code before reading it in with `batch`. `system` also helps to allow non-Java programmers to integrate Jess with other applications. Given that you have an editor named `notepad` on your system, try:

```
Jess> (system notepad README &)
TRUE
```

The `&` character makes the editor run in the background. Omitting it will keep the system command from returning until the called program exits.

The class `jess.Console` is a graphical version of the Jess command-line interface. You type into text field at the bottom of the window, and Output appears in a scrolling window above. Type `java jess.Console` (or `java -classpath jess.jar jess.Console`) to try it.

1.5.5. Jess as an Applet

The class `jess.ConsoleApplet` is a generic Jess applet that uses the same display as the `jess.Console` class. It can be used in general question-and-answer situations simply by embedding the applet class on a Web page. The applet accepts two applet parameters. The value of an `INPUT` parameter will be interpreted as a Jess program to run when starting up. Note that when this program halts, the Jess prompt will appear in the applet window. The applet also accepts a `COMPACT` parameter. If present, `ConsoleApplet` will contain only a bare-bones version of Jess (no optional functions will be loaded).

Note that the ConsoleApplet and ConsoleDisplay classes now use the Java 1.1 event model, which is still not supported by much of the installed base of Web browsers. Don't use them if you want to deploy highly portable applets! Actually, the idea of deploying Jess as an applet makes less and less sense these days; a much better alternative is to run Jess on the server side (as a servlet, for example) and run only the GUI on the client. Good applets are generally very small (a few tens of kilobytes), while Jess's class files now occupy hundreds of kilobytes.

1.6. What makes a good Jess application?

Jess can be used in two overlapping ways. First, it can be a rule engine - a special kind of program that very efficiently applies rules to data. A rule-based program can have hundreds or even thousands of *rules* , and Jess will continually apply them to data in the form of a *knowledge base* . Often the rules will represent the heuristic knowledge of a human expert in some domain, and the knowledge base will represent the state of an evolving situation (an interview, an emergency). In this case, they are said to constitute an *expert system.* Expert systems are widely used in many domains. Among the newest applications of expert systems are as the reasoning part of *intelligent agents*, in enterprise resource planning (ERP) systems, and in order validation for electronic commerce.
But the Jess language is also a general-purpose programing language, and furthermore, it can directly access all Java classes and libraries. For this reason, Jess is also frequently used as a dynamic scripting or rapid application development environment. While Java code generally must be compiled before it can be run, a line of Jess code is executed immediately upon being typed. This allows you to experiment with Java APIs interactively, and build up large programs incrementally. It is also very easy to extend the Jess language with new commands written in Java or in Jess itself, and so the Jess language can be customized for specific applications.
Jess is therefore useful in a wide range of situations. One application for which Jess is not so well suited is as an applet intended for Internet use. Jess's size (a few hundred kilobytes of compiled code) makes it too large for applet use except on high-speed LANs. Furthermore, some of Jess's capabilities are lost when it is used in a browser: for example, access to Java APIs from the Jess language may not work at all due to security restrictions in some browsers. When building Web-based applications using Jess, you should strongly consider using Jess on the server side (in a servlet, for example.)

1.7. About Jess and performance

Jess's rule engine uses an improved form of a well-known algorithm called Rete (latin for "net") to match rules against the knowledge base. Jess is actually faster than some popular expert system shells written in C, especially on large problems, where performance is dominated by algorithm quality.
Note that Rete is an algorithm that explicitly trades space for speed, so Jess' memory usage is not inconsiderable. Jess does contain some commands which will allow you to sacrifice some performance to decrease memory usage. Nevertheless, Jess' memory usage is not ridiculous, and moderate-sized programs will fit easily into Java's default 16M heap.

1.8. Command-line, GUI, or embedded?

As we've discussed, Jess can be used in many ways. Besides the different categories of problems Jess can be applied to, being a library, it is amenable to being used in many different kinds of Java programs. Jess can be used in command-line applications, GUI applications, servlets, and applets. Furthermore, Jess can either provide the Java `main()` for your program, of you can write it yourself. You can develop Jess applications (with or without GUIs) without compiling a single line of Java code. You can also write Jess

applications which are controlled entirely by Java code you write, with a minimum of Jess language code.
The most important step in developing a Jess application is to choose architecture from among the almost limitless range of possibilities. One way to organize the possibilities is to list them in increasing order of the amount of Java programming involved.

1. Pure Jess language scripts. No Java code at all.
2. Pure Jess language scripts, but the scripts access Java APIs.
3. Mostly Jess language scripts, but some custom Java code in the form of new Jess commands written in Java.
4. Half Jess language scripts, with a substantial amount of Java code providing custom commands and APIs; `main()` provided by Jess.
5. Half Jess language scripts, with a substantial amount of Java code providing custom commands and APIs; `main()` written by you.
6. Mostly Java code, which loads Jess language scripts at runtime.
7. All Java code, which manipulates Jess entirely through its Java API. This option is not fully supported at this time, but will in a future release.

Examples of some of these types of applications are package with Jess. The basic examples like `wordgame.clp`, `zebra.clp`, and `fullmab.clp` are all type 1) programs. `draw.clp` and `frame.clp` are type 2) programs. The `pumps` example is packaged two ways. If you run it using the script file `pumps.clp`, it is a type 4) program; if you run it using `MainInJava.java`, it is a type 6) application.
Your choice can be guided by many factors, but ultimately it will depend on what you feel most comfortable with. Types 4) and 5) are most prevalent in real-world applications.

Result: JESS is the rule based expert system. The various aspects of JESS expert system such are Rete algorithm and syntax for applying rules to knowledge base are studied.