**Informed Methods**

**Best-FirstSearch**

Order nodes on the nodes list by increasing value of an evaluation function, f, that incorporates domain-specific information in some way. This is a generic way of referring to the class of informed methods.

**Greedy Best-First Search**

Use as an evaluation function f(n) = h(n), sorting nodes by increasing values of f

Selects node to expand that is believed to be closest (hence it's "greedy") to a goal node (i.e., smallest f value)

Not complete

Not admissible, as shown in the following example:

h=3

/\

/ \

h=2 h=4

| |

| |

h=1 h=1

| |

| |

h=1 goal

|

|

h=1

|

|

goal

Assuming all arc costs are 1, then Greedy Best-First search will find the left goal, which has a solution cost of 5, while the optimal solution is the path to the right goal, which has a cost of 3.

**Beam Search**

Use an evaluation function f(n) = h(n), but the maximum size of the nodes list is k, a fixed constant

Only keeps k best nodes as candidates for expansion, and throws the rest away

More space efficient than Greedy Best-First Search, but may throw away a node that is on a solution path

Not complete

Not admissible

Algorithm A  
Use as an evaluation function f(n) = g(n) + h(n), where g(n) is as defined in Uniform-Cost search. That is, g(n) = minimal cost path from the start state to the current state n.

Adds a "breadth-first" component to the evaluation function by including the g term

Ranks nodes on the search frontier by the estimated cost of a solution that goes from the start node through the given node to a goal node. That is, g(n) is the cost from the start node to node n, andh(n) is the estimated cost from node n to a goal.

Not complete since if a node n is on the solution path but h(n) = infinity, then node n may never be expanded

Not admissible

Algorithm A\*  
Use the same evaluation function as used by Algorithm A except add the constraint that for all nodes n in the search space, h(n) <= h\*(n), where h\*(n) = the true cost of the minimal cost path from n to a goal.

When the condition h(n) <= h\*(n) holds, we say that h is admissible.

Using an admissible heuristic guarantees that a node on the optimal path can never look so bad that you bypass it forever

A\* is complete whenever the branching factor is finite, and every operator has a fixed positive cost

A\* is admissible

If h(n) = h\*(n) for all n, then only the nodes on the optimal solution path will be expanded. So, no extra work will be performed.

If h(n) = 0 for all n, then this is an admissible heuristic and results in A\* performing exactly as the Uniform-Cost Search does

If h1(n) < h2(n) <= h\*(n) for all n that are not goal nodes, then h2 is a better heuristic than h1 in the sense that if A1\* is a version of the A\* algorithm which uses h1, and A2\* is a version of the A\* algorithm which uses h2, then every node expanded by A2\* is also expanded by A1\*. In other words, A1\* expands at least as many nodes as A2\*. We say that A2\* is better informed than A1\*.

The closer h is to h\*, the fewer extra nodes that will be expanded

**A\* Algorithm**

Put the start node S on the nodes list, called OPEN

If OPEN is empty, exit with failure

Remove from OPEN and place on CLOSED a node n for which f(n) is minimum

If n is a goal node, exit (trace back pointers from n to S)

Expand n, generating all its successors and attach to them pointers back to n. For each successor n' of n

If n' is not already on OPEN or CLOSED estimate h(n'),g(n')=g(n)+ c(n,n'), f(n')=g(n')+h(n'), and place it on OPEN.

If n' is already on OPEN or CLOSED, then check if g(n') is lower for the new version of n'. If so, then:  
(i) Redirect pointers backward from n' along path yielding lower g(n').  
(ii) Put n' on OPEN.   
If g(n') is not lower for the new version, do nothing.

Goto 2

Example

Consider the following search space where the start state is S and the goal state is G. The left figure shows the arcs labeled with the costs of the associated operators. The right figure shows the states labeled with the value of the heuristic function, h, if it is ever applied at that state.

S ... Initial State S 8

/|\ /|\

1/ 5 \8 / | \

/ | \ / | \

A B C 8 A B 4 C 3

/|\ | / /|\ | /

3/ 7 9 4 /5 / | \ | /

/ | \|/ / | \|/

D E G .... Goal State D E G

\* \* 0

Edge Costs Heuristic = Estimated Costs = h(n)

Summary of g(n), h(n), f(n) = g(n) + h(n), as well as h\*(n), the hypothetical perfect heuristic:

n g(n) h(n) f(n) h\*(n)

S 0 8 8 9

A 1 8 9 9

B 5 4 9 4

C 8 3 11 5

D 4 inf inf inf

E 8 inf inf inf

G 10/9/13 0 10/9/13 0

Notice that since h(n) <= h\*(n) for all n, h is admissible

Optimal path = S B G

Cost of the optimal path = 9

Greedy Best-First Search: f(n) = h(n)

node

expanded nodes list

---- ------------------

{ S(8) }

S { C(3) B(4) A(8) }

C { G(0) B(4) A(8) }

G { B(4) A(8) }

Solution path found is S C G. #nodes expanded = 3.

See how fast the search is!! But it is NOT optimal.

A\* Search: f(n) = g(n) + h(n)

node

expanded nodes list

---- ------------------

{ S(8) }

S { A(9) B(9) C(11) }

A { B(9) G(10) C(11) D(inf) E(inf) }

B { G(9) G(10) C(11) D(inf) E(inf) }

G { C(11) D(inf) E(inf) }

Solution path found is S B G. #nodes expanded = 4.

Still pretty fast. And optimal too.

Devising Heuristics

Good heuristics must be fast to compute, because if it takes so long to compute the value of a heuristic at a single node, it may have been preferable to have just expanded more nodes using a cheaper heuristic. For example, if the heuristic function is a breadth-first search to find a solution and its cost, then this is clearly too expensive to be useful.

Can often devise good heuristics by computing the cost of an exact solution to a simplified version of the problem. For example, in the 8-puzzle, if we relax the assumption about how a tile can be moved so that any tile can be moved in a single step from any position to any other position, then this means that a solution costs only the number of misplaced tiles since each misplaced tile can now be moved in one step to its goal position. This heuristic is admissible because at each move just one tile moves one position, so this is the minimum number of steps to get each of the misplaced tiles to their goal position.

Similarly, if we assume that tiles in the 8-puzzle are restricted to moving one square horizontally or vertically at a time, but we relax the assumption that only one tile can occur at a board position at a time, then each tile can be moved independently to its goal position, taking a number of steps equal to the Manhattan distance from its start position to its goal position. This leads to a heuristic which is the sum of the distances of the misplaced tiles to their goal positions. This heuristic is admissible.

The heuristic function h is an indicator of "adventurousness" in that in Algorithms A and A\* a good heuristic allows successive nodes on a single path to be expanded in succession even when several "good" steps are intermixed with a few "bad" steps.

Unfortunately, A\* often suffers because it cannot venture down a single path unless it is almost continuously having success (i.e., h is decreasing). Any failure to decrease h will almost immediately cause the search to switch to another path.

In order to devise an admissible heuristic, h must frequently be very simple and therefore resorts to (almost) uniform-cost search through parts of the search space.

If optimality is not required, i.e., a satisficing solution is enough, using a heuristic that occasionally overestimates the actual cost but is usually very close to the actual cost (over or under), will result in many fewer nodes being expanded to find a solution than using a provably admissible heuristic.

Iterative Improvement Algorithms

Rather than searching for a solution path and then executing the steps associated with the solution path, iteratively pick a next best move, make that move, and then repeat. Hence, irrevocably make a decision about one step at each iteration.

Best used in problems where all the information for a solution is contained in the node itself. For example, cryptarithmetic problems.

Rather than trying to find a solution with minimum value of the evaluation function, f, for historical reasons, we instead will attempt to maximize the function. That is, the goal is to find a state, n, such that f(n) >= f(i) for all states i in the state space.

Hill-Climbing Search

Look at all immediate successors of current state m

If there exists a successor n such that f(n) > f(m), and f(n) >= f(t) for all the successors t of m, then move from m to n. Otherwise, halt at m. (Note: This definition differs from the textbook in that we require f(n) to be strictly greater than f(m); the textbook's algorithm states that f(n) must only be greater than or equal to f(m). Their definition allows the algorithm to move through states that are equal in their values of f.)

Looks one step ahead to determine if any successor is better than the current state; if there is, move to the best successor.

Similar to Greedy Best-First search but Hill-Climbing does not allow backtracking or jumping to an alternative path since there is no nodes list of other candidate frontier nodes from which the search could be continued. Corresponds to Beam search with a beam width of 1 (i.e., the maximum size of the nodes list is 1).

Not complete since the search will terminate at "local maxima," "plateaus," and "ridges."

Algorithm visualized in terms of a surface in 3D:

^

|

| y

|

f | /

| /

| /

| /

|/

+----------------> x

Consider the state space to be the set of points in the x,y plane, and for each such point the height f is the value of the evaluation function for that state. This height function, f = f(x,y), defines a surface. The initial state corresponds to a point on this surface and the goal is to find a state where the height is a global maximum.

Hill-climbing (should be called Valley-Finding in this context where we are minimizing instead of maximizing a value) moves in the direction of steepest ascent since it moves to the successor (i.e., adjacent) node that increases f the most.

Notice that by considering the state space as a continuous space of points in the x,y plane, if the height surface is continuous (i.e., smooth so that derivatives are well-defined everywhere), then the direction of steepest ascent corresponds to the gradient direction = [df(x,y)/dx, df(x,y)/dy], and the search is called gradient ascent.

Simulated Annealing

Named after a metal-casting technique called annealing where molten metal is heated and then gradually cooled resulting in an even distribution of the molecules and a desired crystalline structure

Attempts to fix the problem with hill-climbing methods where the search gets stuck in a local maximum.

Basic idea: Instead of picking the best move, pick a random move; if the successor state obtained by this move is an improvement over the current state, then do it. Otherwise, make the move with some probability < 1. The probability decreases exponentially with the badness of the move.

Define a temperature function that decreases over time. At each move, compute the current temperature T, and use T to determine the probability with which to allow a move to a worse state. In the limit, T goes to 0 at which point the method is doing hill-climbing. Hence the probability is proportional to T.

If temperature is lowered slowly enough, simulated annealing is complete and admissible. Intuitively, this is the case because the temperature can be controlled so that it is large enough to move off a local maximum, but small enough to not move off a global maximum.

Algorithm  
Assume we are trying to find a state where some evaluation function f is a global maximum:

current = Initial-State(problem)

for t = 1 to infinity do

T = Schedule(t) ; T is the current temperature, which

; is monotonically decreasing with t

if T=0 then return current ; halt when temperature = 0

next = Select-Random-Successor-State(current)

deltaE = f(next) - f(current) ; If positive, next is

; better than current.

; Otherwise, next is

; worse than current.

if deltaE > 0 then current = next ; always move to

; a better state

else current = next with probability p = e^(deltaE / T)

; as T -> 0, p -> 0

; as deltaE -> -infinity, p -> 0

end