Heuristic search

A *heuristic* is a "rule of thumb"

using domain knowledge

Heuristics may be incorporated

-in next-state rules

-in control

In the latter case:

Used to guide search

Examples:

eight puzzle

Distance of pieces from goal position

science: what to study

- likeliness of publication
- number of people interested
- how much will this answer a fundamental question

chess

- control of center
- material advantage
- lead to attack on king

Heuristics

Can divide search algorithms based on *informedness*

Uninformed: Make no use of domain knowledge

(also known as *blind search*

Breadth-first search, Depth first search

Iterative Deepening (ID)

Informed: Make use of heuristics

Hill climbing (gradient descent)

Best-first search: A and A*

IDA*: heuristic version of ID

Hill climbing

1. Guess a possible solution.

2. If it is one, quit.

3. Use all applicable operators to generate

   a set of candidate possible solutions

4. Now, evaluate all solutions (usually via a heuristic)

   for "closeness to the goal"

5. Take the closest. Go to step 2.

Problems:

local maxima, plateaus, ridges

First Aid:

backtracking, long jumps, blind movement, probabilistic movement

General Tree search algorithm
(OBVIOUSLY NOT IN PURE LISP!!!)

(ignores bookkeeping for graphs and keeping track of the path):

/* Initialize */

1. OPEN = {Start_node}; CLOSED = NIL;

While TRUE **do**

2. **If** empty(OPEN) **then return**(*FAIL*);

/* Expand next node */

3. CURRENT = **car**(OPEN);

4. CLOSED = **cons**(CURRENT, CLOSED);

5. **If** Goal(CURRENT) **Then Return**(PATH)

/* apply operators to get a list of successors */

6. SUCCESSORS = Expand(CURRENT);

/* Heuristics go here, in how things are inserted */

7. OPEN = Insert(SUCCESSORS, OPEN);

od

Best first search:

Using an evaluation function

Best first search is a heuristic search:
Use an *evaluation function* to determine *priority* in OPEN

How it works:

> Put nodes in OPEN in order of evaluation of "goodness"
>
> **car**(OPEN) == node with *lowest* f value

Thus, we will be opening the "best" node first.

Of course, this is really the "seemingly-best" node!

Best first search Example 1:
Greedy Search

Greedy search uses a *heuristic function* h(n)

that estimates the distance to the goal from node n.

In the "find a route" problem, this could be the straight-line distance to the goal.

In the 8-puzzle, this could be an estimate of the number of moveds required to get to the goal.

Greedy search:

- Is *very* goal-directed

- Is *not* optimal (won't find the shortest path in all cases).

- often minimizes *search cost* though

Best first search Example 2:
Algorithm A

Note: Uniform cost search is *not* goal-directed

It expands from the start node as "evenly" as possible, like ripples on a pond.

But it is optimal.

Would be nice to combine UCS and Greedy search:

f(n) = cost of getting to n

+ estimated cost of getting to the goal from here

= g(n) + h(n)

Thus, f(n) is:

an estimate of the cost of a path through node n.

This is called "Algorithm A".

Search: Evaluation functions

g(n) and f(n) are often estimates - true values are g*(n) and f*(n).

We estimate g*(n) with g(n) == actual cost of getting to here

I.e., sum of costs of *arcs* in path to root

For *uninformed search methods* h(n) == 0

If all arcs have cost -1:

Depth first search

But usually, we assume arcs costs are >= 0

If all arcs have cost 1:

Breadth first search

Otherwise, Uniform cost search

(finds shortest path to a goal)

Search: Evaluation functions

Informed algorithms make use of h:

heuristic function

A good heuristic function:

-is cheap to evaluate

-constrains the search tree

to be long and stringy

-finds nearly optimal solutions

NOTE TRADEOFF:

between cost of evaluating h and constraining search

between optimality and constraining search

Search: Evaluation functions

An *admissible* algorithm is one that is:

guaranteed to find optimal path

It turns out that if h(n) is a lower bound on h*:

$h(n) <= h^*(n)$

Then algorithm A is admissible.

This is called algorithm A*

However, for combinatorial problems:

Inadmissible algorithms may be the only way to get a solution!
h(n) may exceed h*(n)

Search: Evaluation functions

Informedness:

Given two A* algorithms

A_1 using h_1
A_2 using h_2

if $h_1 > h_2$ for *all* nodes:

Then algorithm A_1 is *more informed* than A_2

We can prove that A_2 will always open every node opened by A_1 (i.e., A_1 will always open *fewer* nodes).

Search: Evaluation functions

Recap:

Uniform cost search: Distance from source is known.
(generalization of BFS)

Uniform cost search + estimate of distance to goal (*h*)
= heuristic search = Algorithm A

Uniform cost search + special estimate of distance to goal
= Algorithm A*

Admissibility: An algorithm is admissible if it always terminates in an optimal path (this is the same notion as *optimality* that we used before).

Special estimate: One that never exceeds true distance
=> we will always find an optimal path, i.e. A* is admissible.

An algorithm A_1 is *more informed* than A_2
if $h_1 > h_2$

We can prove that A_2 will always open every node opened by A_1 (i.e., A_1 will always open *fewer* nodes).

Search: Evaluation functions

Sketch of Optimality of A*:

1. Termination of A*

2. Show that there is always a node on the optimal path to the goal in OPEN, call it n*

3. Show A* must terminate in the optimal path

   proof by contradiction

   Assume we found a non-optimal path terminating in n

   then $f(n) > f(n^*)$ so how did it get off OPEN first?

A heuristic algorithm that saves space: IDA*

IDA* is Iterative Deepening using an
f-cost limit instead of a depth limit.

First, need to change Depth-bounded depth first search to use f-costs instead.

When it reaches something over the depth limit,
it needs to update the next f-limit, "next-f"
(which needs to be a global!)

The following ignores bookkeeping for solution path
(assumes it is stored with node):
In C-ish lisp:

```
DFS-contour(node,f-limit)
(cond      ((f-cost(node) > f-limit)
           next-f = min(next-f, f-cost(node));
           return(nil);)
        (goal(node) return(node);)
        (t
           successors = expand(node);
           foreach node in successors,
              DFS-contour(node,f-limit)))
```

A heuristic algorithm that saves space: IDA*

Now, IDA* is:

DFS-contour(start, f-cost(start));

(also ignores absolute failure...)

This will cost more time than A*,

but will use less space....

Problem:

If f-costs go up in very small increments,
this could take an inordinate amount of time.

E.g., in Traveling Salesperson Problems...

So, use an "epsilon" increment - guarantees
solution within epsilon of optimal.