Heuristic Search

Heuristic Search

So far, we've been talking about *blind* search: All you are given is the state space: States, start state, successor operator, goal tester.

In *heuristic search* you are also given an *evaluation* function, a function mapping a state to a number.

Variants:

1. Cost function – low numbers are good; vs. Value function – large numbers are good. (Obviously they are interchangeable, but different metaphors are more natural in different situations.)
2. Flexible goal – optimal value is best but near-optimal is also good; vs. inflexible goal – best or never mind.
3. Known target value vs. unknown target value.

There are pretty much two general strategies

- Hill climbing, for discrete state spaces.
- Gradient descent, for continuous state spaces

Hill climbing

```
% Maximizing value of function f(S)
hillClimbing() {
    S = some starting state;
    while (true) {
        N = the neighbor of S for which f(N) is maximal
                (break ties arbitrarily);
        if (f(N) <= f(S)) return S;
        S=N;
     }
  }
```

Terminology: “Break ties arbitrarily”

“Break ties arbitrarily”: Any way that is most convenient for you to code.

(Usually either the first best candidate you encounter or the last, depending on how exactly you write the code.)

vs.

“Break ties randomly” – Collect all the values that are tied for best, and make a random choice with equal probability between them, using a random number generator.

Properties of hill-climbing

```
hillClimbing() {
    S = some starting state;
    while (true) {
        N = the neighbor of S for which
                        f(N) is maximal;
        if (f(N) <= f(S)) return S;
        S=N;
    }
}
```

- If the state space is finite, then it terminates. Proof: f(S) increases at each iteration, so it can't go into an infinite loop.
- The state returned is a local, non-strict, maximum (i.e. at least as good as any of its neighbors).
- Memory: Two states
- That's all one can say with certainty.

Properties of hill-climbing

In particular, if there is a local maximum (state that is as good as all its neighbors) that is not a global maximum (best state in the state space), then if it reaches the local maximum, it will return a non-optimal solution.

For complex problems, this almost always happens.

Still, in many cases, hill climbing gives pretty good solutions, particularly if (a) you have some way of choosing a good starting point; (b) reaching a pretty good but non-optimal solution is good enough for your purposes.

Example: Travelling salesman

Given: Complete directed graph with costs on the edges.

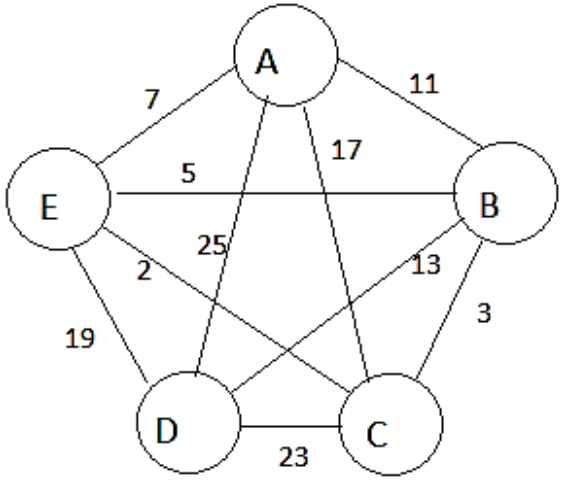
Find: The graph cycle of minimal total cost that includes every vertex once.

State: A cycle through the graph.

Evaluation function to be minimized: Total cost.

Neighbor function: Swap the order of two consecutive vertices.

Example



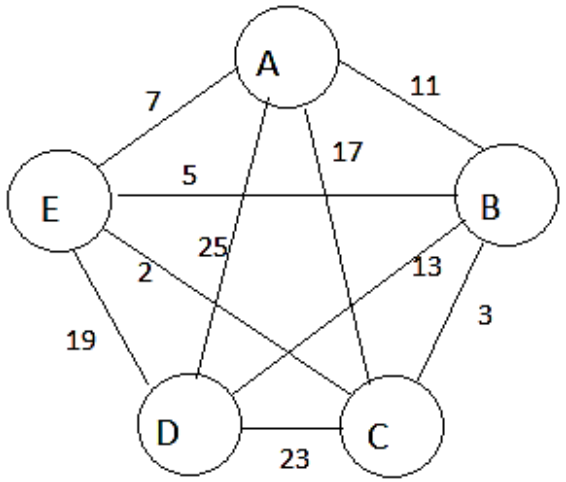
S = ABCDEA Cost=63

Neighbors: ACDEBA Cost=67
ACBDEA Cost=59
ABDCEA Cost=56
ABCEDA Cost=60
AEBCDA Cost=63

S=ABDCEA Cost=56

Neighbors: ADCEBA Cost=66
ADBCEA Cost=50
ABCDEA Cost=63
ABDECA Cost=62
AEBDCA Cost=65

Example, continued



S = ADBCEA Cost=50

Neighbors: AEDBCA: Cost=59
ABDCEA: Cost=56
ADCBEA: Cost=63
ADBECA: Cost=62
ABCEDA: Cost=60

Return ADBCEA

Travelling salesman

For the general case with N vertices:

A state has N neighbors.

Fixing the starting and ending point, the state space has size (N-1)!.

(Though if the graph is undirected, as here, then reversed path is identical, so only (N-1)!/2.)

The diameter (maximal distance between any two states) is (N-1)(N-2)/2. That is a lower bound on the worst case running time for hill climbing.

Travelling salesman

Other neighbor relations are possible. For instance, you could allow swaps in the path between any two elements.

For example, if S=ABCDEFGHA and you swap B and E you get the neighbor AECDBFGHA.

That way each state has N(N-1)/2 neighbors.

How does this state space search compare to the previous one?

Certainly, at each iteration, it is more computation to enumerate the neighbors.

Quite likely, this will either reduce the number of iterations or return a higher quality answer or both.

But there is no guarantee of an improvement in either respect.

The problem is NP-complete, so (presumably) you can’t actually win with hill-climbing or any other algorithm. No polynomial time algorithm always returns the best answer.

Random Restart

```
randomRestart() {
    Best = —∞;
    repeat {
        START = a random state in the state space;
        S = run hillClimbing starting from START;
        if (f(S) > Best) { BestState = S;  Best=f(S); }
      }  until (TerminationCondition)
    return BestState;
}
```

What is “TerminationCondition”?

TerminationCondition is what is called a “metaparameter” or a “hyperparameter” or (my own terminology) a “tunable parameter” of the algorithm. It’s a feature of the algorithm that has to be set when the algorithm is implemented.

Possible termination conditions for RandomRestart

A particular target value has been attained (e.g. Cost=0).

1000 iterations of the loop

1000 iterations since the last time f(P) improved

Over the last 1000 iterations, f(P) has improved by less than Δ (another metaparameter).

Replace “1000” above by q(Problem) where q() is some function of some kind of relevant features of the problem.

How do you choose the value of a metaparameter?

For AI problems, there is almost never any theory to guide you.

If you're running the program once, to get a particular answer, you guess, or you set it in terms of what your budget will afford.

If you want to sell a program that will work well over a wide space of examples, then you have to do

Empirical experimentation

How much and what kind of experimentation?

That depends on your circumstances: What is your budget, how much time do you have, how valuable are incremental improvements to the final answer?

Hill-climbing with sideways motion

```
% Looking for a state S where Cost(S)=0.

sideways(S) {
    NSide=0;
    repeat {
        NNS = the set of neighbors of S tied for smallest Cost;
        N = random choice from NNS;
        if (Cost(N)==0) return N;
        if (Cost(N) > Cost(S)) return "Fail";
        S = N;
        if (Cost(N) < Cost(S)) NSide=0; else NSide++;
       }  until(NSide > MaxSide);
    return "Fail";
}
```

Sideways motion vs. simple hill climbing

In simple hill climbing, you always have to improve: if you reach a state where you can’t improve, you quit.

In sideways motion, you are allowed to go sideways. You’re not allowed to get worse; if you reach a state that is better than all its neighbors, you quit. But you’re allowed to move to an equal state. But of course that would allow the program to go into an infinite loop, so you put a limit “MaxSides” on the number of consecutive sideways motions you’re allowed.

Sideways motion

The intuition is this: In many types of combinatorial problems (examples soon), there are large “plateaus” where the error function is constant, and only a few states in these plateaus allow you to drop down to a lower value of the error function.

In sideways motion, you wander around the plateau at random until you can find a state where you can drop down to a lower value of the error function.

The reason that it’s important to choose the neighbor at random among those of equal value is so that you (on average) do wander around the plateau. If you chose the neighboring state systematically, you could get stuck in a loop.

Example of search with sideways motion

State: Any placement of any number of queens on the NxN board.

Neighbor: Add a queen on an empty square or delete a queen. (Thus N^2 neighbors.)

Start state: Random configuration

Error function: Sum of two terms:

Number of pairs who attack one another plus
max(0,N-# of queens on the board) (Penalize for having too few queens, but don't reward for having too many.)

Search with sideways motion vs. blind search

The state space is incomparably larger: 2^{N^2} vs. < N!

(For N=8, that's 2^{64} vs. <40320)

Nonetheless, search with sideways motion plus random restart is *much* faster.

With DFS, you can solve around N=25, N=30 in reasonable time.

With hill climbing with sideways motion, you can solve N=200 in seconds. (Usually just a handful of restarts.)

Down side: If you don't get an answer, you don't know whether one doesn't exist or whether you were just unlucky.

Another example of error function

Exact set cover problem:

Given: A set Ω and a collection C of subsets of Ω.

Find: A subcollection D of C s.t. every element of Ω appears exactly once in D.

Example: Ω = {a,b,c,d,e,f,g,h}

C1={a,b,g}. C2={a,c,d}. C3={b,e}. C4={c,h}. C5={c,f,h}. C6={b,f,h}. C7={e,g}

State: Any subcollection of D.

Error: The number of repetitions of elements plus the number of elements uncovered.

For instance Error(C1,C2,C3) = 2 (a and b repeated) +

2 (f and h uncovered) = 4.

Another example of error function

Exact set cover problem:

Given: A set Ω and a collection C of subsets of Ω.

Find: A subcollection D of C s.t. every element of Ω appears exactly once in D.

Example: Ω = {a,b,c,d,e,f,g,h}

C1={a,b,g}. C2={a,c,d}. C3={b,e}. C4={c,h}. C5={c,f,h}. C6={b,f,h}. C7={e,g}

Neighbor relation: From a subcollection D, consider all ways of either adding one element of C not in D, or deleting an element of D.

For instance if D={C1,C3,C6}, the neighbors of D are

{C3,C6}, {C1,C2,C3,C6},

{C1,C6}, {C1,C3,C4,C6},

{C1,C3,C5,C6}, {C1,C3}.

Choosing a state space and an error function for sideways motion

The one absolute rule of the error function:

Error(S)=0 if and only if S is a goal state.

Generally:

- Any two states in the state space are connected by a path.
- The neighbor relation is symmetric, certainly between states of equal error.
- The branching factor is not small, though not enormous.
- The value of the error function on neighboring states tends to be close, though that’s not always the case.