Optimization

Burr H. Settles
CS-540, UW-Madison
www.cs.wisc.edu/~cs540-1
Summer 2003

Announcements

- Project groups and preliminary topic ideas will be due on 6/30
  - A week from Monday
  - Be thinking about what you’d like to do
  - Try to find others in the class who might be interested in the same topic!
- We’re almost ready to start using the class discussions on the mailing list

Last Time

- Someone asked why, in the textbook’s example on page 98, A* search looks backward to nodes already explored
- The answer is: they don’t appear to be using a closed list
- If \( g(n) \geq 0 \) for all \( n \), closed lists prevent such unnecessary work (back-tracked states will always be toward the end of the queue)
- However, if \( -\infty \leq g(n) \leq \infty \) (i.e. costs can be negative: or rewards?), you would want to allow such moves, and ignore the closed list
- Closed lists aren’t necessary, but are often useful

Searching: So Far

- We’ve discussed how to build goal-based and utility-based agents that search to solve problems
- We’ve also presented both uninformed (or blind) and informed (or heuristic) approaches for search
- What we’ve covered so far are called partial search strategies because they build up partial solutions, which could enumerate the entire state space before finding a solution

Complete Searching

- In complete search strategies, each state or node already represents a complete solution to the problem at hand
  - We aren’t concerned with finding a path
  - We don’t necessarily have a designated start state
- The objective is to search through the problem space to find other solutions that are better, the best, or that meet certain criteria (goal)

Optimization

- Problems where we search through complete solutions to find the best solution are often referred to as optimization problems
- Most optimization tasks belong to a class of computational problems called NP
  - Non-deterministic Polynomial time solvable
  - Computationally very hard problems
  - For NP problems, state spaces are usually exponential, so partial search methods aren’t time or space efficient
Optimization Problems

- The $k$-Queens Problem
  Of course this isn’t *real* chess

- Traveling Salesman Problem (TSP)
  Perhaps most famous optimization problem

---

Optimization Problems

- As it turns out, many real-world problems that we might want an agent to solve are similarly hard optimization problems:
  - Bin-packing
  - Logistics planning
  - VLSI layout/circuit design
  - Theorem-proving
  - Navigation/routing
  - Production scheduling, supply/demand
  - Learning the parameters for a neural network
    (more in the machine learning part of the course)

---

Satisfiability (SAT)

- Classic NP problem
- Belongs to a specific class of problems in the complexity hierarchy called NP-Complete
  - Any other NP problem can be converted to a SAT problem in polynomial time
  - These are the hardest problems we know of

---

Satisfiability (SAT)

- Given:
  - Some logical formula
  - Array of binary variables in the formula

- Do:
  - Find a truth assignment for all variables such that the formula is *satisfied* (true)
Satisfiability (SAT)

- For example, given the following formula with 8 clauses and 10 variables:
  \((x_1 \lor x_2 \lor \neg x_3) \land (x_2 \lor \neg x_4) \land (\neg x_2 \lor x_5) \land (\neg x_4 \lor x_6) \land (\neg x_5 \lor x_7) \land (\neg x_6 \lor x_8)\)

- We need to find a 10-bit array that makes the formula logically true
  - There are \(2^{10} = 1024\) possible binary arrays
  - Only 32 of them (~3%) are solutions to this formula

A bit of notation: \(\lor\) and \(\land\) are logical “or” and “and” operators, respectively. A clause \((x_i \lor \neg x_j)\) is true if either \(x_i = 1\) or \(x_j = 0\). The formula is satisfied when all of its clauses are true.

Greedy Search for SAT

- A state is a 10-bit array \(x\)
  - e.g. \(x = “01010101”\)
  - For this array, \(x_1 = 0, x_2 = 1\), etc.
- Our actions are to toggle any single bit in the array to generate a new one
- Our heuristic (or objective function) will be to minimize the number of clauses in the formula that are unsatisfied by the candidate string
  - We are trying to satisfy them all
Greedy Search for SAT

- Greedy search does the right thing in that it does find a solution, and quickly
- However, it only expanded 4 out of the 35 that are generated in the search (i.e. placed in the open list)
  - You may work it all out yourself if you wish
- It also found a direct route, and we don’t need to remember the path, so storing all those extra states pretty much wasted space!

Local Search

- Local search is a type of greedy, complete search that focuses on a specific (or local) part of the search space, rather than trying to branch out into all of it
- We only consider the neighborhood of the current state rather than the entire state space so far (so as not to waste time/space)

Beam Search

- One type of local search is beam search, which uses $f(n)$, as in other informed searches, but uses a “beam” with a width $w$ to restrict the possible search directions
- Only keep the $w$-best nodes in the open list, and throw the rest away
- More space efficient than best-first search, but can throw away nodes on a solution path

Beam Search Example

\[ f(n) = g(n) + h(n) \]
\[ w = 2 \]

\begin{tabular}{|c|c|c|}
\hline
State & Open & Closed \\
\hline
S & 5 & 5 \\
A & 8 & 8 \\
B & 9 & 9 \\
C & 1 & 1 \\
D & 4 & 4 \\
E & 5 & 5 \\
G & 10 & 10 \\
\hline
\end{tabular}
Hill-Climbing (HC)

- The most common local strategy is called hill-climbing, if the task is to maximize the objective function
  - Called gradient descent if we are minimizing

- We consider all the successors of the current node, expand the best one, and throw the rest away

Hill-Climbing for SAT

- How to represent the problem?
  - States, actions, and objective function

- We’ve seen this before...
  - States: binary array that correspond to variable truth assignments (e.g. “1010101010” for 10 variables)
  - Actions: toggle a single bit on/off
  - Objective function: to minimize the number of unsatisfied clauses
Hill-Climbing for k-Queens
- How to represent the problem?
  - States, actions, and objective function
- This is a little tricky…
  - States: k x k chess board with k queens
  - Actions: move a single queen to any of its legal positions up, down, or diagonally
  - Objective function: minimize the number of conflicts between queens

Hill-Climbing for TSP
- How to represent the problem?
  - States, actions, and objective function
- This is even trickier…
  - States: an n-city tour (e.g. 1-4-6-2-5-3 is a 6-city tour)
  - Actions: swap any two cities in the tour
  - Objective function: minimize the total cost or length of the entire tour

Hill-Climbing Issues
- The solution found by HC is totally determined by the initial state
  - How should it be initialized?
  - Should it be fixed or random?
  - Maybe we just want to get started somewhere in the search space…”
- Can frequently get stuck in local optima or plateaux, without finding the global optimum

Objective Surfaces
- The objective surface is a plot of the objective function’s “landscape”
- The various levels of optimality can be seen on the objective surface
- Getting stuck in local optima can be a major problem!

Example Objective Surface
\[ f(x,y) = - (|x| - 10) \cos(x) - y \times \cos(|y| - 10) \]

Escaping Local Optima
- Searching with HC is like scaling Mount Everest in a thick fog with one arm and amnesia
- Local optima are OK, but sometimes we want to find the absolute best solution
- Ch. 5 & 6 of How to Solve It: Modern Heuristics have a better discussion of techniques for escaping local optima than AI: A Modern Approach
Escaping Local Optima

- There are several ways we can try to avoid local optima and find more globally optimal solutions:
  - Random Restarting
  - Simulated Annealing
  - Tabu Search

Random Restarting

- If at first you don’t succeed, try, try again!

- The idea here is to run the standard HC search algorithm several times, each with different, randomized initial states

- Of course, depending on the state space, this can be a difficult task in and of itself… not all states that can be generated are legal for some problems

Random Restarting

- If the object is to find a particular solution (or that reaches a certain goal), we can stop when it is found

- If the aim is to find the best solution (optimize), repeat for a fixed number of trials and return the best found

Random Restarting

- As the number of trials increases, the probability of finding a solution (or the global optimum) approaches 1.0
  - This is for the trivial reason that, given enough restarts, we’ll ultimately generate the optimum

- But we don’t want to keep restarting ad infinitum
  - That would defeat the point of local search!

Random Restarting

- It turns out that, if each HC run has a probability $p$ of success, the number of restarts needed is approximately $1/p$

- For example, with 8-Queens, there is a probability of success $p = 0.14 = 1/7$

- So, on average, we would need only 7 randomly-initialized trails of the basic HC search to find a solution
Random Restarting

- Random restart approaches are built in to many state-of-the-art constraint satisfaction algorithms
- They’ve been shown especially useful in systems geared toward solving hard SAT problems
  - GSAT
  - Davis-Putnam (DPLL with restarts)

Simulated Annealing (SA)

- We don’t always want to take the best local move, sometimes we might want to:
  - Try taking uphill moves that aren’t the best
  - Actually go downhill to escape local optima
- We can alter HC to allow for these possibilities:
  - Modify how successor states are selected
  - Change the criteria for accepting a successor

Simulated Annealing (SA)

- With standard Hill-Climbing:
  - We explore all of the current state’s actions/successors
  - Accept the best one
- Perhaps we can modify this to account for the other kinds of moves we’d like to make:
  - Choose one action/successor at random
  - If it is better, accept it, otherwise accept with some probability $p$

Simulated Annealing (SA)

Concepts behind the SA analogy:

<table>
<thead>
<tr>
<th>Physical system</th>
<th>Optimization problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical state</td>
<td>Feasible solution</td>
</tr>
<tr>
<td>Energy</td>
<td>Objective function</td>
</tr>
<tr>
<td>Ground state</td>
<td>Goal or optimum</td>
</tr>
<tr>
<td>Rapid quenching</td>
<td>Local search</td>
</tr>
<tr>
<td>Temperature</td>
<td>Control parameter $T$</td>
</tr>
<tr>
<td>Annealing</td>
<td>Simulated annealing</td>
</tr>
</tbody>
</table>

Let $\Delta E = score(\text{NEXT}) - score(\text{CURRENT})$

\[ p = e^{\Delta E/T} \] (Boltzmann equation)

- $\Delta E \to -\infty, \ p \to 0$
  - The worse a move is, the probability of taking it decreases exponentially
- $\text{Time} \to \infty, \ T \to 0$
  - As time increases, the temperature decreases, in accordance with a cooling schedule
- $T \to 0, \ p \to 0$
  - As temperature decreases, the probability of taking a bad move also decreases
Simulated Annealing (SA)

```c
CURRENT = initialState  // initialize the search
for TIME = 1 to do {
    T = schedule(TIME)   // elapsed time effects schedule
    if T = 0 then         // T has totally "cooled"
        return CURRENT
    NEXT = random successor of CURRENT
    E = score(NEXT) - score(CURRENT)
    if E > 0 then         // take all "good" moves
        CURRENT = NEXT
    else
        CURRENT = NEXT with probability e^(-E/T)
    }
```

Simulated Annealing (SA)

- Can perform downhill and locally sub-optimal moves, unlike HC
- Chance of finding global optimum increased
- SA is fast in practice
  - Only one random neighbor generated per step
  - Only score one successor instead of whole neighborhood
  - Can use more complex heuristics

Simulated Annealing (SA)

According to thermodynamics, to grow a crystal:
- Start by heating a row of materials in a molten state
- The crystal melt is cooled until it is frozen in
- If the temperature is reduced too quickly, irregularities occur and it does not reach its ground state (e.g. more energy is trapped in the structure)

By analogy, SA relies on a good cooling schedule, which maps the current time to a temperature \( T \), to find the optimal solution
- Usually exponential
- Can be very difficult to devise

Simulated Annealing (SA)

SA was first used to solve layout problems for VLSI (very large-scale integration) computer architectures in the 1980s
- Optimally fitting hundreds of thousands of transistors into a single compact microchip

It is also proven useful for the TSP, and is used in many factory scheduling software systems

Tabu Search

- Tabu search is a way to add memory to a local search strategy, and force it to explore new areas of the search space

- We’ve seen state-based memory before with the closed list, but this memory:
  - Tracks actions taken rather than states expanded
  - Is designed to be a limited (short-term) memory

- Moves that have been seen or taken too recently or too often become tabu (or taboo)

Tabu Search

- We maintain an array \( M \) which tracks time-stamps of the actions we’ve taken
  - We store in location \( M_i \) the most recent time action \( i \) was taken in the search

- The key parameter of tabu search is the horizon: how long should a certain remain tabu?
  - If we set this too small, we may default to normal HC and stay stuck in local optima
  - If we set it too large, we may run out of legal moves!
  - Usually problem-dependent
Tabu Search

```java
CURRENT = initialState  // initialize search
BEST = CURRENT        // retain best solution so far
for TIME = 1 to MAX_TIME do {
    NEXT = best legal successor of CURRENT
    ACTION = action that generated NEXT
    M[ACTION] = tabu info based on horizon & TIME
    if score(CURRENT) better than score(BEST) then
        BEST = CURRENT
    }
return BEST
```

Tabu Search

- Instead of an array, memory can also be stored in a queue, or tabu list:
  - As a move is made, place it in the queue
  - When the queue becomes full, the oldest move is removed and becomes legal again
  - The size of the queue is the horizon

Tabu Search

- Since we take the best non-tabu move at each step, we are allowed to take backward steps or just OK moves, as with simulated annealing

- Tabu search can also be faster than standard HC, as it doesn’t have to evaluate all action/successors, just those that are legal

Summary

- Local search methods are more appropriate for solving complete search and optimization problems
  - State spaces can be prohibitively large
  - The goal is different than with partial search strategies

- However, basic hill-climbing can become stuck in local optima rather than finding the best solution

Summary

- There are several effective ways of escaping local optima for local searching, which exploit different properties:
  - Random restarting tries several times from different parts of the search space
  - Simulated annealing allows for a variety of moves by searching stochastically
  - Tabu search is deterministic, but incorporates memory to force exploration of the state space