Artificial Intelligence

CS482, CS682, MW 1 – 2:15, SEM 201, MS 227
Prerequisites: 302, 365
Instructor: Sushil Louis, sushil@cse.unr.edu, http://www.cse.unr.edu/~sushil
Non-classical search

- Path does not matter, just the final state
- Maximize objective function
Local optimum

- Heuristic: Number of pairs of queens attacking each other directly
- Movement: only within your column

H = 17 but all successors have > 1
We have a black box “evaluate” function that returns an objective function value.

Application dependent fitness function
Local Hill Climbing

function HILL-CLIMBING(problem) returns a state that is a local maximum

current ← MAKE-NODE(problem.INITIAL-STATE)
loop do
    neighbor ← a highest-valued successor of current
    if neighbor.VALUE ≤ current.VALUE then return current.STATE
    current ← neighbor

• Move in the direction of increasing value
• Very greedy
• Subject to
  • Local maxima
  • Ridges
  • Plateaux
• 8-queens: 86% failure, but only needs 4 steps to succeed, 3 to fail
Hill climbing

• Keep going on a plateau?
  • Advantage: Might find another hill
  • Disadvantage: infinite loops $\rightarrow$ limit number of moves on plateau
    • 8 queens: 94% success!!
• Stochastic hill climbing
  • randomly choose from among better successors (proportional to obj?)
• First-choice hill climbing
  • keep generating successors till a better one is generated
• Random-restarts
  • If probability of success is $p$, then we will need $1/p$ restarts
  • 8-queens: $p = 0.14 \approx 1/7$ so 7 starts
  • 6 failures (3 steps), 1 success (4 steps) = 22 steps
  • In general: Cost of success + $(1-p)/p$ * cost of failure
  • 8-queens sideways: 0.94 success in 21 steps, 64 steps for failure
    • Under a minute
Simulated annealing

function SIMULATED-ANNEALING(problem, schedule) returns a solution state
inputs: problem, a problem
         schedule, a mapping from time to “temperature”

current ← MAKE-NODE(problem.INITIAL-STATE)
for t = 1 to ∞ do
    T ← schedule(t)
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← next.VALUE − current.VALUE
    if ΔE > 0 then current ← next
    else current ← next only with probability $e^{-\frac{\Delta E}{T}}$

• Gradient descent (not ascent)
• Accept bad moves with probability $e^{-\frac{\Delta E}{T}}$
• T decreases every iteration
• If schedule(t) is slow enough we approach finding global optimum with probability 1
Beam Search

Idea: keep $k$ states instead of 1; choose top $k$ of all their successors
Not the same as $k$ searches run in parallel! Searches that find good states recruit other searches to join them
Problem: quite often, all $k$ states end up on same local hill
Idea: choose $k$ successors randomly, biased towards good ones
Observe the close analogy to natural selection!
Genetic Algorithms

- Stochastic hill-climbing with information exchange
- A population of stochastic hill-climbers

```plaintext
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
    inputs: population, a set of individuals
            FITNESS-FN, a function that measures the fitness of an individual

    repeat
        new_population ← empty set
        for i = 1 to SIZE(population) do
            x ← SELECTION(population, FITNESS-FN)
            y ← SELECTION(population, FITNESS-FN)
            child ← REPRODUCE(x, y)
            if (small random probability) then child ← MUTATE(child)
            add child to new_population
        population ← new_population
    until some individual is fit enough, or enough time has elapsed
    return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
    inputs: x, y, parent individuals

    n ← LENGTH(x); c ← random number from 1 to n
    return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```
More detailed GA

- Generate pop(0)
- Evaluate pop(0)
- T=0
- While (not converged) do
  - Select pop(T+1) from pop(T)
  - Recombine pop(T+1)
  - Evaluate pop(T+1)
  - T = T + 1
- Done
Generate pop(0)

Initialize population with randomly generated strings of 1’s and 0’s

```c
for(i = 0 ; i < popSize; i++){
    for(j = 0; j < chromLen; j++){
        Pop[i].chrom[j] = flip(0.5);
    }
}
```
Genetic Algorithm

- Generate pop(0)
- Evaluate pop(0)
- T=0
- While (not converged) do
  - Select pop(T+1) from pop(T)
  - Recombine pop(T+1)
  - Evaluate pop(T+1)
  - T = T + 1
- Done
Evaluate pop(0)

Decoded individual → Evaluate → Fitness

Application dependent fitness function
Genetic Algorithm

• Generate pop(0)
• Evaluate pop(0)
• T=0
• While (T < maxGen) do
  • Select pop(T+1) from pop(T)
  • Recombine pop(T+1)
  • Evaluate pop(T+1)
  • T = T + 1
• Done
Genetic Algorithm

- Generate pop(0)
- Evaluate pop(0)
- T=0
- While (T < maxGen) do
  - Select pop(T+1) from pop(T)
  - Recombine pop(T+1)
  - Evaluate pop(T+1)
  - T = T + 1
- Done
Selection

• Each member of the population gets a share of the pie proportional to fitness relative to other members of the population
• Spin the roulette wheel pie and pick the individual that the ball lands on
• Focuses search in promising areas
int roulette(IPTR pop, double sumFitness, int popsize)
{

    /* select a single individual by roulette wheel selection */

    double rand, partsum;
    int i;

    partsum = 0.0; i = 0;
    rand = f_random() * sumFitness;

    i = -1;
    do{
        i++;
        partsum += pop[i].fitness;
    } while (partsum < rand && i < popsize - 1);

    return i;
}
Genetic Algorithm

- Generate pop(0)
- Evaluate pop(0)
- T=0
- While (T < maxGen) do
  - Select pop(T+1) from pop(T)
  - Recombine pop(T+1)
  - Evaluate pop(T+1)
  - T = T + 1
- Done
Crossover and mutation

Mutation Probability = 0.001
Insurance

Xover Probability = 0.7
Exploration operator
Crossover helps iff substrings are meaningful components
void crossover(POPULATION *p, IPTR p1, IPTR p2, IPTR c1, IPTR c2)
{
    /* p1,p2,c1,c2,m1,m2,mc1,mc2 */
    int *pi1,*pi2,*ci1,*ci2;
    int xp, i;

    pi1 = p1->chrom;
    pi2 = p2->chrom;
    ci1 = c1->chrom;
    ci2 = c2->chrom;

    if(flip(p->pCross)) {
        xp = rnd(0, p->lchrom - 1);
        for(i = 0; i < xp; i++) {
            ci1[i] = muteX(p, pi1[i]);
            ci2[i] = muteX(p, pi2[i]);
        }
        for(i = xp; i < p->lchrom; i++) {
            ci1[i] = muteX(p, pi2[i]);
            ci2[i] = muteX(p, pi1[i]);
        }
    } else {
        for(i = 0; i < p->lchrom; i++) {
            ci1[i] = muteX(p, pi1[i]);
            ci2[i] = muteX(p, pi2[i]);
        }
    }
}
int muteX(POPULATION *p, int pa)
{
    return (flip(p->pMut) ? 1 - pa : pa);
}
How does it work

<table>
<thead>
<tr>
<th>String</th>
<th>decoded</th>
<th>f(x^2)</th>
<th>fi/Sum(fi)</th>
<th>Expected</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>01101</td>
<td>13</td>
<td>169</td>
<td>0.14</td>
<td>0.58</td>
<td>1</td>
</tr>
<tr>
<td>11000</td>
<td>24</td>
<td>576</td>
<td>0.49</td>
<td>1.97</td>
<td>2</td>
</tr>
<tr>
<td>01000</td>
<td>8</td>
<td>64</td>
<td>0.06</td>
<td>0.22</td>
<td>0</td>
</tr>
<tr>
<td>10011</td>
<td>19</td>
<td>361</td>
<td>0.31</td>
<td>1.23</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td></td>
<td>1170</td>
<td>1.0</td>
<td>4.00</td>
<td>4.00</td>
</tr>
<tr>
<td>Avg</td>
<td></td>
<td>293</td>
<td>0.25</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Max</td>
<td></td>
<td>576</td>
<td>0.49</td>
<td>1.97</td>
<td>2.00</td>
</tr>
</tbody>
</table>
# How does it work cont’d

<table>
<thead>
<tr>
<th>String</th>
<th>mate</th>
<th>offspring</th>
<th>decoded</th>
<th>f(x^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110</td>
<td>1</td>
<td>2</td>
<td>01100</td>
<td>12</td>
</tr>
<tr>
<td>1100</td>
<td>0</td>
<td>1</td>
<td>11001</td>
<td>25</td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>4</td>
<td>11011</td>
<td>27</td>
</tr>
<tr>
<td>10</td>
<td>011</td>
<td>3</td>
<td>10000</td>
<td>16</td>
</tr>
<tr>
<td>Sum</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Continuous spaces

Suppose we want to site three airports in Romania:
- 6-D state space defined by \((x_1, y_2), (x_2, y_2), (x_3, y_3)\)
- objective function \(f(x_1, y_2, x_2, y_2, x_3, y_3) = \text{sum of squared distances from each city to nearest airport}\)

Discretization methods turn continuous space into discrete space, e.g., empirical gradient considers \(\pm \delta\) change in each coordinate

Gradient methods compute

\[
\nabla f = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right)
\]

to increase/reduce \(f\), e.g., by \(x \leftarrow x + \alpha \nabla f(x)\)

- What is a good value for \(\alpha\) ?
  - Too small, it takes too long
  - Too large, may miss the optimum
Newton Raphson Method

Sometimes can solve for $\nabla f(x) = 0$ exactly (e.g., with one city). Newton–Raphson (1664, 1690) iterates $x \leftarrow x - H_f^{-1}(x) \nabla f(x)$ to solve $\nabla f(x) = 0$, where $H_{ij} = \partial^2 f / \partial x_i \partial x_j$. 
Linear and quadratic programming

• Constrained optimization
  • Optimize $f(x)$ subject to
    • Linear convex constraints – polynomial time in number of variables
      • Linear programming – scales to thousands of variables
    • Convex non-linear constraints – special cases $\rightarrow$ polynomial time
      • In special cases non-linear convex optimization can scale to thousands of variables
Games and game trees

• Multi-agent systems + competitive environment $\rightarrow$ games and adversarial search
• In game theory any multiagent environment is a game as long as each agent has “significant” impact on others
• In AI many games were
  • Game theoretically: Deterministic, Turn taking, Two-player, Zero-sum, Perfect information
  • AI: deterministic, fully observable environments in which two agents act alternately and utility values at the end are equal but opposite. One wins the other loses
• Chess, Checkers
• Not Poker, backgammon,
Game types

<table>
<thead>
<tr>
<th></th>
<th>deterministic</th>
<th>chance</th>
</tr>
</thead>
<tbody>
<tr>
<td>perfect information</td>
<td>chess, checkers, go, othello</td>
<td>backgammon monopoly</td>
</tr>
<tr>
<td>imperfect information</td>
<td>battleships, blind tictactoe</td>
<td>bridge, poker, scrabble nuclear war</td>
</tr>
</tbody>
</table>

Starcraft? Counterstrike? Halo? WoW?
Search in Games

“Unpredictable” opponent $\Rightarrow$ solution is a strategy specifying a move for every possible opponent reply

Time limits $\Rightarrow$ unlikely to find goal, must approximate

Plan of attack:

- Computer considers possible lines of play (Babbage, 1846)
- Algorithm for perfect play (Zermelo, 1912; Von Neumann, 1944)
- Finite horizon, approximate evaluation (Zuse, 1945; Wiener, 1948; Shannon, 1950)
- First chess program (Turing, 1951)
- Machine learning to improve evaluation accuracy (Samuel, 1952–57)
- Pruning to allow deeper search (McCarthy, 1956)
Tic-Tac-Toe

Figure 5.1  FILES: figures/tictactoe.eps (Tue Nov 3 16:23:55 2009). A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.
Minimax search

Perfect play for deterministic, perfect-information games

Idea: choose move to position with highest minimax value
     = best achievable payoff against best play

E.g., 2-ply game:

```
          MAX
            |
            |
            3
          /|\  /|
 A1  A2  A3
        /   /
     3   2  2
    / /   / /
 A11 A12 A13 A21 A22 A23 A31 A32 A33
   /   /   /   /   /   /   /   /   /
  3   12  8   2   4   6   14  5   2
```
Minimax algorithm

function MINIMAX-DECISION(state) returns an action
inputs: state, current state in game
return the a in ACTIONS(state) maximizing MIN-VALUE(Result(a, state))

function MAX-VALUE(state) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
v ← −∞
for a, s in Successors(state) do v ← MAX(v, MIN-VALUE(s))
return v

function MIN-VALUE(state) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
v ← ∞
for a, s in Successors(state) do v ← MIN(v, MAX-VALUE(s))
return v
3 player Minimax

- Two player minimax reduces to one number because utilities are opposite – knowing one is enough
- But there should actually be a vector of two utilities with player choosing to maximize their utility at their turn
- So with three players $\rightarrow$ you have a 3 vector
- Alliances?

---

Figure 5.4  FILES: figures/minimax3.eps (Tue Nov 3 16:23:11 2009). The first three plies of a game tree with three players ($A$, $B$, $C$). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.
Minimax properties

• Complete?
  • Only if tree is finite
    • Note: A finite strategy can exist for an infinite tree!

• Optimal?
  • Yes, against an optimal opponent! Otherwise, hmmmm

• Time Complexity?
  • $O(b^m)$

• Space Complexity?
  • $O(bm)$

• Chess:
  • $b \sim 35$, $m \sim 100$ for reasonable games
  • Exact solution still completely infeasible
Alpha-beta pruning
Alpha-beta
Alpha-beta
Alpha-beta
Alpha-beta
Alpha-beta

- Alpha is the best value (for Max) found so far at any choice point along the path for Max
  - Best means highest
  - If utility v is worse than alpha, max will avoid it
- Beta is the best value (for Min) found so far at any choice point along the path for Min
  - Best means lowest
  - If utility v is larger than beta, min will avoid it
Alpha-beta algorithm

function **Alpha-Beta-Decision**(\(state\)) returns an action
    return the \(a\) in **Actions**(\(state\)) maximizing **Min-Value**(**Result**(\(a, state\)))

function **Max-Value**(\(state, \alpha, \beta\)) returns a utility value
    inputs: \(state\), current state in game
    \(\alpha\), the value of the best alternative for **Max** along the path to \(state\)
    \(\beta\), the value of the best alternative for **Min** along the path to \(state\)
    if **Terminal-Test**(\(state\)) then return **Utility**(\(state\))
    \(v \leftarrow -\infty\)
    for \(a, s\) in **Successors**(\(state\)) do
        \(v \leftarrow \text{MAX}(v, \text{Min-Value}(s, \alpha, \beta))\)
        if \(v \geq \beta\) then return \(v\)
        \(\alpha \leftarrow \text{MAX}(\alpha, v)\)
    return \(v\)

function **Min-Value**(\(state, \alpha, \beta\)) returns a utility value
    same as **Max-Value** but with roles of \(\alpha, \beta\) reversed
Alpha beta example

• Minimax(root)
  • = max (min (3, 12, 8), min(2, x, y), min (14, 5, 2))
  • = max(3, min(2, x, y), 2)
  • = max(3, aValue <= 2, 2)
  • = 3
Alpha-beta pruning analysis

• Alpha-beta pruning can reduce the effective branching factor
• Alpha-beta pruning’s effectiveness is heavily dependent on MOVE ORDERING
 • 14, 5, 2 versus 2, 5, 14
 • If we can order moves well \( O(\frac{m}{b^{\frac{1}{2}}}) \)
 • Which is \( O((b^{1/2}).m) \)
• Effective branching factor then become square root of \( b \)
• For chess this is huge \( \rightarrow \) from 35 to 6
• Alpha-beta can solve a tree twice as deep as minimax in the same amount of time!
  • Chess: Try captures first, then threats, then forward moves, then backward moves comes close to \( b = 12 \)
Imperfect information

• You still cannot reach all leaves of the chess search tree!
• What can we do?
  • Go as deep as you can, then
  • Utility Value = Evaluate(Current Board)
  • Proposed in 1950 by Claude Shannon
Search

- Problem solving by searching for a solution in a space of possible solutions
- Uninformed versus Informed search
- Atomic representation of state
- Solutions are fixed sequences of actions