Session 3

I. Questions (Homework/LISP/Class)

AIMA

- What is the AIMA library.
- How can you use it

II. Issues Not In Class

Learning – the process of modification of each component of an agent to make the components agree closer with the available feedback thereby improving the agent’s performance.

- learning element – responsible for making improvements
- performance element – responsible for selecting external actions… the agent being modified.
- critic – provides feedback on the agent’s performance and suggests improvements.
    - distinguishes the reward in the percept by providing direct feedback on quality of agent’s performance.
- problem generator – suggests actions that will lead to exploration.

Bidirectional Search – simultaneous searches from the initial state forward and from the goal state backwards that stop when the 2 searches meet. Encouraged by the fact that $b^{d/2} + b^{d/2} \ll b^d$

- complete & optimal (with uniform step costs) if both algorithms are BFS.
- Checking a node for membership in the other search tree can be done in constant time via a hash table, but requires that 1 search tree be in memory.
  - Time-complexity: $O\left(b^{d/2}\right)$  Space-complexity: $O\left(b^{d/2}\right)$
- Bidirectional search requires that the predecessors of a node be efficiently computable:
  - Easy when actions are reversible. Otherwise…
- To deal with several (explicitly listed) goal states, we make them all have a successor of a single dummy goal state.
III. N-QUEENS

Russell’s Code

What is N-queens?

- Suppose we have N queens on a chess board of $N \times N$ squares.
  - Queens are allowed to move in any straight vertical, horizontal, or diagonal line indefinitely across the board to capture another piece.
  - We want to place N of them on the board so that no queens can be captured in a single move.

- What is the problem description (PEAS)? Okay, maybe this problem is a bit simple for the rigors of PEAS, but it’s a good habit to always write out your problem description first.
  - Environment $\rightarrow$ Hmmm… the chess board and the queens.
  - Actions $\rightarrow$ Placing the queens.
  - Sensors $\rightarrow$ Rule checks that ensure no queens are in danger.
  - Performance Measure $\rightarrow$ Number of queens that can be captured in one move.

- How do we solve the problem?
  - Work problems for $N=3,4,5…$ Below is a solution for $N = 8^*$

* The following image was taken from [http://www.eudoxus.com/mp9609f1.gif](http://www.eudoxus.com/mp9609f1.gif)
All Unique Solutions to the 8-Queens Problem

<table>
<thead>
<tr>
<th>Sol.Nbr.</th>
<th>Row 1</th>
<th>Row 2</th>
<th>Row 3</th>
<th>Row 4</th>
<th>Row 5</th>
<th>Row 6</th>
<th>Row 7</th>
<th>Row 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>6</td>
<td>3</td>
<td>7</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>6</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>8</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>7</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>7</td>
<td>3</td>
<td>6</td>
<td>8</td>
<td>5</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

† Table taken from http://www.durangobill.com/N_Queens.html
Uninformed Strategies

Which uninformed strategy would be ideal for N-Queens?
- **Breadth-First Search** – A bad idea for this problem. We are guaranteed to expand all nodes of depth less than \( N \) nodes. We’ll never reach any goals until \( N \)-th level.
- **Uniform Cost Search** – Not worth mentioning… no costs on our edges.
- **Depth-Limited Search** – Ideal for this problem. ALL GOALS are at depth \( N \) so we can halt search there! Moreover Goals are Dense.
- **Bidirectional** – Problem formulating a goal state is hard in this case.
  - If you knew the goal state, you’ve already solved the problem!
    - The states are cumulative (encapsulating all previous states) since we need to know the entire path to check whether we’re in a goal.
    - However, we could have a global state and specify from both directions. BUT this is equivalent to any other ordering of piece placement – The placements are COMMUTATIVE.

In-depth look at problem

- It might be ridiculous to place the columns (rows) in order from left to right (top to bottom). What if other orders of placement were more efficient?
  - Not so bad actually. If we always were going in a left to right placement order, we should continue to do so.
  - The columns with the most constraints on their values are the leftmost since eventually diagonals run off the board.

What are the simplest facts we can glean from the game?
  - Every queen must have its own column... but every queen must have its own row as well. If we think about this for a second, this means that every feasible N-queens solution must be a permutation of the list: \(\{1, 2, \ldots, N\}\).
    - A **permutation** of a list is another list with the same elements in a different order!
      
      e.g. \(N=4\) \(\pi_a = \{2, 4, 1, 3\}\)
    - Now to incorporate a diagonal constraint. This can be formulated mathematically as
      \[ \forall s < t \quad |\pi(s) - \pi(t)| \neq t - s. \]
    - Thus we have a way to write this problem mathematically; it must be a permutation that obeys the above constraint.

**A* Search**

- First a little bookkeeping about yesterday's lecture.
- **Consistent (Monotonic) Heuristic** – \(h(n)\) is not more than the cost through \(n\) to \(n'\) plus \(h(n')\). Thus, a general triangle inequality:
  \[ h(n) \leq c(n,a,n') + h(n') \]
- Can A* do the job efficiently?
  - Heuristic Functions (In the Incremental Formulation).
    - Choosing a column to place. As discussed above, choosing a column to place is simple. Leftmost is probably most constrained.
    - Choosing a value for the column. Probably want a value that limits the fewest other columns.
      - This fails! All values have the same limitations. If we move on diagonal off the board we bring another one on.

**Incremental vs Complete Formulation**

- Which formulation is most convenient for the N-queens problem – incremental or complete.
  - **incremental formulation** – variables are assigned one at a time such that the assignment remains consistent.
    - Allows us to simply start with an empty board and add queens one column at a time – similar to the human approach.
    - Leads to a lot of backtracking (we come to final columns and realize there are no legal placements).
  - **complete-state formulation** – all variables are assigned initially and changed incrementally in attempts to make the assignment consistent.
    - valid since the path by which a solution is reached is irrelevant.
    - NO BACKTRACKING
      - When we have illegal queens, we simply move a single queen to remove possible captures.

- **Why do some problems fit well into incremental formulations and others into complete formulations?**
A*-Incremental N-Queens

- What are the possible moves in this formulation?
  - Move a queen in its column?
  - Swap a pair of queens across columns!!
- Now what are good heuristic function?
  - Number of Queens in Conflict → overestimates.

Local Search Solutions to N-Queens

- Simulated Annealing – why?
- Genetic Algorithm

CSP Solutions

TBD next time
Solution Density

How Common are the N-queens solutions? The following table came from http://www.durangobill.com/N_Queens.html and shows the number of solutions (and unique solutions) along with their probabilities. These probabilities are “inflated” in that I assumed the queens each had to be in separate rows or columns (N! such configurations) whereas, there are far more dumb solutions (N^2 choose N ~ O(N^{2N})).

<table>
<thead>
<tr>
<th>Order (“N”)</th>
<th>Ordinary Queens Total Solutions</th>
<th>Ordinary Queens Unique Solutions</th>
<th>Probability of Total Solutions</th>
<th>Probability of Unique Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0.0833333333</td>
<td>0.041666667</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>2</td>
<td>0.0833333333</td>
<td>0.016666667</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>1</td>
<td>0.005555556</td>
<td>0.001388889</td>
</tr>
<tr>
<td>6</td>
<td>40</td>
<td>6</td>
<td>0.007936508</td>
<td>0.001190476</td>
</tr>
<tr>
<td>7</td>
<td>92</td>
<td>12</td>
<td>0.002281746</td>
<td>0.000297619</td>
</tr>
<tr>
<td>8</td>
<td>352</td>
<td>46</td>
<td>0.000970018</td>
<td>0.000126764</td>
</tr>
<tr>
<td>9</td>
<td>724</td>
<td>92</td>
<td>0.000199515</td>
<td>2.53527E-05</td>
</tr>
<tr>
<td>10</td>
<td>2,680</td>
<td>341</td>
<td>6.71397E-05</td>
<td>8.54277E-06</td>
</tr>
<tr>
<td>11</td>
<td>14,200</td>
<td>1,787</td>
<td>2.9645E-05</td>
<td>3.73068E-06</td>
</tr>
<tr>
<td>12</td>
<td>73,712</td>
<td>9,233</td>
<td>1.18374E-05</td>
<td>1.48273E-06</td>
</tr>
<tr>
<td>13</td>
<td>365,596</td>
<td>45,752</td>
<td>4.19366E-06</td>
<td>5.2481E-07</td>
</tr>
<tr>
<td>14</td>
<td>2,279,184</td>
<td>285,053</td>
<td>1.74293E-06</td>
<td>2.17985E-07</td>
</tr>
<tr>
<td>15</td>
<td>14,772,512</td>
<td>1,846,955</td>
<td>7.06049E-07</td>
<td>8.82748E-08</td>
</tr>
<tr>
<td>16</td>
<td>95,815,104</td>
<td>11,977,939</td>
<td>2.6938E-07</td>
<td>3.36755E-08</td>
</tr>
<tr>
<td>17</td>
<td>666,090,624</td>
<td>83,263,591</td>
<td>1.04038E-07</td>
<td>1.30051E-08</td>
</tr>
<tr>
<td>18</td>
<td>4,968,057,848</td>
<td>621,012,754</td>
<td>4.08406E-08</td>
<td>5.10512E-09</td>
</tr>
<tr>
<td>19</td>
<td>39,029,188,884</td>
<td>4,878,666,808</td>
<td>1.60422E-08</td>
<td>2.00529E-09</td>
</tr>
<tr>
<td>20</td>
<td>314,666,222,712</td>
<td>39,333,324,973</td>
<td>6.15894E-09</td>
<td>7.69869E-10</td>
</tr>
<tr>
<td>21</td>
<td>2,691,008,701,644</td>
<td>336,376,244,042</td>
<td>2.39413E-09</td>
<td>2.99267E-10</td>
</tr>
<tr>
<td>23</td>
<td>227,514,171,973,736</td>
<td>?</td>
<td>3.66693E-10</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>2,207,893,435,808,350</td>
<td>?</td>
<td>1.42342E-10</td>
<td></td>
</tr>
</tbody>
</table>
How easy are solutions?

It turns out, for big N, solutions to the N-queens no longer look like intricate puzzles with clever tricks – they look like the simplest lines we could think of (Below is a solution for N=46 produced by a Java applet: http://www.apl.jhu.edu/~hall/NQueens.html).
Russell's Code for N-Queens

n-queens as a search problem.

We give both an incremental formulation [2e p 66] and a complete-state formulation [2e p 110-111]. We also provide the methods required for applying genetic algorithms to the complete-state formulation.

Incremental formulation: add one queen at a time, avoiding illegal choices.

(defstruct (nqueens-incremental-problem (:include problem) (:constructor create-nqueens-incremental-problem))
  n ;; the number of queens (n x n board)
)

(defun make-nqueens-incremental-problem (&key (n 8))
  "Returns an nqueens problem instance with an empty board. In general, a state is an n-element vector of queen positions, one per column."
  (create-nqueens-incremental-problem
    :n n
    :initial-state (make-sequence 'vector n :initial-element nil)))

(defun copy-state ((state vector)) (copy-seq state))

(defun goal-test ((problem nqueens-incremental-problem) state)
  "Return true if all queens have been placed, i.e., last queen is non-nil."
  (elt state (1- (nqueens-incremental-problem-n problem))))

(defun actions ((problem nqueens-incremental-problem) state)
  "Generate the possible moves from an nqueens-incremental state. A move is simply the row position of the queen in the next column."
  (let ((n (length state))
        (next-col (position nil state)))
    ;; For each possible square, check if attacked by previously placed queens
    (loop for row from 0 to (1- n) do
      (unless (some #'(lambda (col)
                       (let ((q (elt state col)))
                         (or (= q row) (= (- next-col col) (abs (- q row))))))
                         (iota next-col))
      (push row actions))))

(defun result ((problem nqueens-incremental-problem) action state)
  (let ((outcome (copy-state state)))
    (setf (elt outcome (position nil state)) action)
    outcome))

(defun h-cost ((problem nqueens-incremental-problem) state)
  "Number of unfilled columns."
  (let ((next-col (position nil state)))
    (if next-col (- (nqueens-incremental-problem-n problem) next-col) 0)))

(defun print-nqueens-state (state)
  "Print out nqueens board state."
  (let ((n (length state)))
    (loop for j from (1- n) downto 0 do
      (format t "\n")
      (loop for i from 0 to (1- n) do (format t (if (= (elt state i) j) "Q" ".")))))

Complete-state formulation: start with all queens on the board, pick any queen and move it in its column.

(defstruct (nqueens-complete-problem (:include problem) (:constructor create-nqueens-complete-problem))
  n ;; the number of queens (n x n board)
)
(defun make-nqueens-complete-problem (&key (n 8))
  "Returns an nqueens problem instance with all n queens placed randomly, one per column."
  (create-nqueens-complete-problem :n n :initial-state (random-nqueens-complete-state n)))

(defun random-nqueens-complete-state (n)
  "Return a random complete state with n queens, one per column."
  (let ((state (make-sequence 'vector n)))
    (loop for i from 0 to (1- n) do
      (setf (elt state i) (random n)))
    state))

;;; Methods for genetic algorithms applied to complete-state nqueens

(defun fitness ((problem nqueens-complete-problem) individual)
  "Return the number of non-attacks between queens."
  (let ((n (length individual)))
    (- (/ (* n (- n 1)) 2) (h-cost problem individual))))
Appendix: Problem

;;; -. Mode: Lisp; Syntax: Common-Lisp; -. File: problems.lisp

;;; Defining Problems

(defstruct problem

  ;; A problem is defined by the initial state, successor function, goal test, and path cost (defined, in turn, by step cost). [2e p 62]

  (initial-state (required)) ; A state in the domain

  (successor-fn ((problem problem) state)

    ;; When we define a new subtype of problem, we need to specify either
    ;; 1) a SUCCESSOR-FN method; or
    ;; 2) ACTIONS and RESULT methods.
    ;; If one or the other is not done, an infinite loop will result!
    ;; We may also need to define methods for GOAL-TEST, H-COST, and
    ;; STEP-COST, but they have default methods which may be appropriate.
    ;; In addition, there is a technicality: states and actions require
    ;; hash keys, although a default is provided that often works (see below).

    ;; Return a list of (action . state) pairs that can be reached from this state.

    (mapcar #'(lambda (action) (cons action (result problem action state)))
              (actions problem state)))

(defmethod actions ((problem problem) state)

  ;; Return an list of actions possible in this state;
  ;; use this default method only if successor-fn is independently defined!

  (mapcar #'car (successor-fn problem state)))

(defmethod result ((problem problem) action state)

  ;; Return the state resulting from executing action in state;
  ;; use this default method only if successor-fn is independently defined!

  (cdr (assoc action (successor-fn problem state)

    ;; test #'(lambda (a1 a2)
    ;;         (equalp (action-hash-key problem a1)
    ;;                 (action-hash-key problem a2))))))

(defmethod sequence-result ((problem problem) action-sequence state)

  ;; Return the state resulting from executing action-sequence in state.
  ;; Useful for checking that a proposed solution sequence achieves the goal.

  (if (null action-sequence)
      state
      (sequence-result problem (rest action-sequence)
                           (result problem (first action-sequence) state))))

(defmethod successor-states ((problem problem) state)

  ;; Return a list of states that can be reached from this state.
  ;; This ignores actions, and is appropriate only for offline local search.

  (mapcar #'(lambda (action) (result problem action state)))
              (actions problem state)))

(defmethod goal-test ((problem problem) state)

  ;; Return true or false: is this state a goal state?

  (declare-ignore state)

  (required))

(defmethod step-cost ((problem problem) state1 action state2)

  ;; The cost of going from state1 to state2 by taking action.
  ;; This default method counts 1 for every action. Provide a method for this if
  ;; your subtype of problem has a different idea of the cost of a step.

  (declare-ignore state1 action state2)

  1)
(defun path-cost (problem action-sequence &optional (state (problem-initial-state problem)) (cost 0))
  "Return the sum of step costs along the given action sequence."
  (if (null action-sequence)
      cost
    (let ((next-state (result problem (first action-sequence) state)))
      (path-cost problem (rest action-sequence) next-state
        (+ cost (step-cost problem state (first action-sequence) next-state))))))

(defun h-cost ((problem problem) state)
  "The estimated cost from state to a goal for this problem.
   If you don't overestimate, then A* will always find optimal solutions.
   The default estimate is always 0, which certainly doesn't overestimate."
  (declare (ignore state))
  0)

;;; The ability to generate a single random successor,
;;; rather than all successors at once, is important for
;;; local search algorithms in domains with large state
;;; representations and/or many successors.

(defun random-successor ((problem problem) state)
  "Return (a . s) for a random legal action a and outcome s."
  (let ((action (random-action problem state)))
    (cons action (result problem action state))))

(defun random-successor-state ((problem problem) state)
  "Return the outcome s of a random legal action."
  (result problem (random-action problem state) state))

(defun random-action ((problem problem) state)
  "Return a random legal action in state; typically this
  method must be defined specially for each domain."
  (random-element (actions problem state)))

;;; Hash keys for states and actions.
;;; States are hashed in the graph search algorithms; both states and actions
;;; are hashed in the enumerated-problem class. Two states or actions represented
;;; by complex data structures may not be EQUALP if the representation
;;; is not canonical, so we must define hash keys for them.
;;; For example, moves in backgammon can be written in any permutation
;;; and still be the "same" move. However, this situation is rare.
;;; In most cases, the state or action representation serves as its own hash key.

(defun state-hash-key ((problem problem) state)
  "Key to be used to hash the state; identical states must have EQUAL keys.
  Default is the state itself, i.e., assume a canonical representation."
  state)

(defun action-hash-key ((problem problem) action)
  "Key to be used to hash actions; identical actions must have EQUAL keys."
  action)