Key for Midterm 1

75 minutes == 75 points
open book and notes

1. *15 points*
   Propose an admissible (and not trivial, i.e. $h(n) = 0$ is not a valid answer) heuristic for the Missionaries and Cannibals problem. Assume there is one boat which can carry a maximum of 2 people, and that in the initial state the same number of missionaries and cannibals are on one side of the river. Explain why your heuristics is admissible.

   To come up with a heuristic we can try to solve a relaxed problem. If we do not take into account the possibility of cannibals eating missionaries, we can compute how many trips are needed to take everyone across. The boat takes two people, but after each trip across the boat has to come back to the starting side and so at least one person must paddle back. This will give us the following heuristic;

   $$h_1(n) = (NumberOfPeopleOnInitialSide) - 1$$

   This heuristic is admissible, since all boat trips (except the last one) can result in a net transfer of at most one person to the destination side.

   Another heuristic is

   $$h_2(n) = \frac{NumberOfPeopleOnInitialSide}{BoatCapacity}$$

   This heuristic is also admissible and is dominated by the first one, since $\forall\ n\ n/2 \geq n - 1$.

   Both heuristics have a local minimum, since every time the boat is sent back to the start side at least one person must go back, so each return trip increases $h(n)$.

   Common mistakes:

   - $h(n) = NumberOfBoatTrips$. This is not a heuristic, this is a solution!
   - Ignoring the restriction that cannibals can outnumber missionaries can be used to relax the problem and find a heuristic, but it is not a heuristic.

   Grading criteria:

   - Designed admissible heuristic and had correct explanation: 15 points
   - Designed admissible heuristic but had incorrect explanation: 10 points
   - Designed a non admissible heuristic but had correct explanation: 5 points
   - Some useful explanations: 2 points

2. *15 points*
   Suppose you want to use A with a heuristic function $h(n)$ which may under or overestimate the true cost of reaching the goal from state n. You know, however, that any overestimate is limited to no more than 10% of the true cost. Is there anything you can do to guarantee that*

the algorithm will find the optimal solution (if a solution exist)? If yes, explain how. If not, explain why not. Be precise.

If we modify the heuristic function to be

$$h'(n) = \frac{1}{1.1} \times h(n) = 0.9 \times h(n)$$

we can guarantee that the algorithm will find the optimal solution. The new heuristic is admissible since it underestimates the true cost to the goal.

Grading criteria:

- Designed admissible heuristic and had correct explanation: 15 points
- Designed admissible heuristic but had incorrect explanation: 10 points
- Designed a non admissible heuristic but had correct explanation: 5 points
- Some useful explanations: 2 points

3. *20 points*
Answer these questions on search algorithms:

   (a) *What kind of search does Greedy Best-First Search emulate when used with $h(n) = -2 \times g(n)$? Explain your reasoning. Show how the search works using a simple example.*
   The function $g(n)$ gives the cost of the path from the initial state to the node n. Using $h(n) = -2 \times g(n)$ as a heuristic function in Greedy Best-First Search will cause the algorithm to always select the node with the highest path cost so far (the largest g(n)) to expand next, since this will be the smallest h(n) (i.e. the most negative value). This type of search could be called worst-first or greatest-cost-first.
   If all operations have the same cost, then the largest $g(n)$ will always correspond to the longest path in the search tree. In this case Greedy Best-First Search with this heuristic will emulate Depth-First Search.

   (b) *Can you make A* behave like Breadth-First Search? If yes, explain how. If not, explain why not. Be precise and explain what you will use for $g(n)$ and for $h(n)$.*
   $A^\star$ evaluates nodes by adding $g(n)$, the cost to reach the node n, to $h(n)$, the estimate of the cost to get from node n to the goal: $f(n) = g(n) = h(n)$. To make $A^\star$ behave like Breadth-First Search we use $g(n) = depth(n)$ and $h(n) = 0$. From this follows that $f(n) = depth(n)$, which means that all paths at a given level of depth are considered to have the same cost associated with them. In this case $A^\star$ selects at each time step the shallowest node for expansion. This makes it equivalent to Breadth-First Search.

   Grading criteria:

   - Each individual questions is worth 10 points.
   - Explaining the general method in part A and B gives 5 points per question.
   - Showing an example of how the search works gives 5 points.
   - Explaining that Breadth-First Search assumes equal path costs gives 5 points.

4. *15 points*
Answer these questions briefly but precisely.

(a) *Would using a pattern database be a reasonable heuristics for solving Traveling Salesperson Problems? Explain why (or why not)*

This is not a viable choice for TSP for many reasons. First, unless we need to solve multiple TSP problems for the same set of cities, very few (if any) of the partial solutions will be reused. Computing partial solutions that will not be reused multiple times is a waste of effort. Second, constructing partial solution is not trivial since a TSP is a CSP for which a goal state is not known in advance. This implies that the usual method of starting from the goal node and recording the cost of each pattern found is not applicable. If computing partial solutions is as expensive as solving the problem the usefulness of building a pattern database becomes questionable.

Grading criteria:

- Full credit given for reasonably detailed explanations, even if answers were different.
- Answers that explained how to use a pattern database for TSP but missed the issue of reusability of partial solutions lost 1 point.
- Answers that were correct but generic on how to use a pattern database lost 3 points.

(b) *How would simulated annealing work if the temperature T is always fixed at 0?*

The algorithm (as given in the textbook) terminates with T=0 without doing any search! If the algorithm did not terminate (or if I had asked how the algorithm would behave with an infinitesimally small value of T) it would operate as hill-climbing search. It would always accept the best move found and never select a move that was not an improvement over the current situation, since with a very small T the probability of selecting a different move is basically 0.

Grading criteria:

- Full credit given for answers that noticed the termination condition.
- Answers that explained the correct idea on how the algorithm would behave, but that missed the termination condition lost 1 point.
- Answers that were incorrect or unclear lost 3 to 4 points

(c) *Explain briefly how a ridge in the search space may appear to be a local maximum to a hill-climbing algorithm.*

A ridge causes problems when states along the ridge are not directly connected, so that the only choices at each point on the ridge require to go downhill. Figure 4.13 in the textbook illustrates this graphically.

Grading criteria:

- Full credit given for answers that answered with clearly the question.
- Answers that were generic, or incorrect (some answered what happens in a plateau not on a ridge) lost 2 to 4 points

5. *10 points*
Write a function, `add-dupl`*, to duplicate all occurrences of a given element in a list. It should work like this:*

```
(add-dupl 3 '(2 3 4 1 3)) ==> (2 3 3 4 1 3 3)
(add-dupl 3 '(2 3 3 1)) ==> (2 3 3 3 3 1)
(add-dupl 5 '(2 3 4 1 3)) ==> (2 3 4 1 3)
```

```
;;; recursive definition
(defun add-dupl (el lst)
  (cond ((null lst) nil)
        ((eql el (car lst))
         ;; need to insert the element twice in the resulting list
         (cons el (cons el (add-dupl el (cdr lst)))))
        (t (cons (car lst) (add-dupl el (cdr lst))))))

;;; iterative definition
(defun add-dupl (el lst &aux (result nil))
  (dolist (x lst)
          (cond ((eql el x)
                 ;; need to insert the element twice in the resulting list
                 (push x result) (push x result))
                (t (push x result))))
  (nreverse result))
```

Grading criteria:

- Full credit given for answers that were correct even if inefficient or too complex.
- Answers that used incorrectly list/cons/append lost 2 points.
- Answers that were partially incorrect (e.g., duplicated only the first occurrence of the element) lost 3 to 6 points.