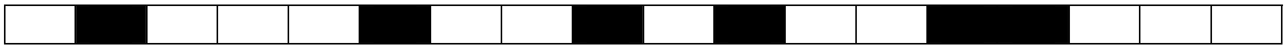
Answer Key
CSci 5511 Artificial Intelligence 1
October 7, 2008

Exam 1

1. Problem Representation

`20 points [graded by baylor]`

```
Consider the following problem: you are given a path of N white
and black squares. The exact configuration of white and black
squares and the length of the path vary with the instance of the
problem you are given to solve.
```



```
You start on the left-most square and the goal is to move off to
the right end of the path in the least number of moves. If you
are on white, you can move 1 or 2 squares right. If you are on a
black square, you can move 1 or 4 squares right.

1.1. Describe how you would represent the state space, including
the states, successor function and goal test.
```

State Space	array of colors, size 1-N
Solution	array of actions, size $\leq$ N
State	an integer (position in the problem). Let's call it `position`
Goal Test	`position` > N

Operators	`move1(position): position = position + 1`
	`move2(position): position = position + 2` when `onWhite(position)`
	`move4(position): position = position + 4` when `onBlack(position)`

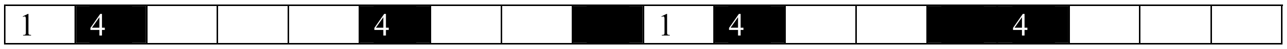
Discussion

State Space Representation
The world is a one-dimensional array so we'll represent it that way. A four square world of all black will look like [B,B,B,B] and an alternating black and white world will look like [B,W,B,W].

Solution Representation
For this problem, we want to find the sequence of moves to the right so we need to track each move we make. To represent that sequence, we can use complete-state or incremental-state representations. For this problem, incremental state makes the most sense. We'll start with an empty array of actions []. When we're done, the array will hold the (ordered) set of moves (actions) needed to solve the problem. The length of the array depends on the number of actions needed to solve the problem but, in the worst case, it will be size of the state space.

For completeness, let's look at how we could have done this as a complete-state formulation. We begin (and end) with an array the size of the state space. For each state, we'd store a Boolean saying whether we pick that move or not. Obviously, the incremental-formulation is always smaller than or equal to the size of the complete-state formulation and contains all the same information. Since the incremental-formulation is more compact and the complete-state formulation offers no advantages (for this problem), i prefer the incremental-formulation.

You were not required to solve the problem, you just had to show how to represent it. Still, i think seeing the solution will help you understand why we choose one representation over another. For the image above, the shortest path is [1,4,4,1,4,4]. Technically, the solution is [move1, move4, move4, move1, move4, move4] but since the number version is shorter and unambiguous, i'm going to abbreviate the method names to the number of steps they represent. Just keep in mind those numbers in the solution are function names, not actual numbers. Here's what [1,4,4,1,4,4] looks like in our world:



If we wanted to do this in complete-state formulation, the solution would be [T,T,f,f,f,T,f,f,f,T,T,f,f,f,T,f,f,f]. See why i prefer the incremental-formulation for this problem?

Another solution, and one that always exists for this problem regardless of the number of squares or their colors, is [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]. This is the longest possible solution to this problem. This is the only time the incremental-state and complete-state formulation are of equal length (the complete-state solution is [T,T,T,T,T,T,T,T,T,T,T,T,T,T,T,T,T,T]).

Many students did not specify how they would represent the solution. Since the problem did not explicitly ask for this, no points were taken off if you didn't do this. However, students were required to formally write a goal test. This often involves checking a value in either the solution or state arrays. Some students did not have the necessary information in their state representations and did not specify how (or if) solutions were tracked and as a result lost point on their goal test.

State
Our actions depend on our current state, so we need to track that. Our world is a line. The only thing that our successor state functions depend on is our position on that line. Our state, then, is a single number representing which square in the line we're on. To make the rest of this explanation easier to understand, let's give this number a name: `position`.

Note: We don't technically *have* to explicitly track our position. Our solution contains actions representing how many squares we've moved. Since we always start at 1 and actions are deterministic (if we say we move four, we move four), it would be easy to create a function that calculates position for us.

Goal Test
The goal test is:
```
      position > N
```

or, if you prefer code:

```
      Boolean atGoal(int position) { return position > N; }
```

where N is the number of squares in the state space. Since we are trying to move past the last square in the world, the goal test is that our position (which is returned by the successor state functions) is a larger number than the length of our world (how much larger is irrelevant).

Several students seemed to think that the goal was to land on the last square. It was not. It was to move beyond the last square. i didn't take any points off for it because i figured if you knew enough to land on the last square, you probably knew enough to get past it and had probably just read the question wrong.

Many students believed that the goal test was to be both beyond the end of the array (basically, a buffer overflow) *and* to have done it in the fewest number of steps possible. This last part is wrong. The goal test says "i have landed in the goal state, yes or no". It's a function your search algorithm (A*, etc.) will call. It has no way of knowing whether or not you got there in the shortest number of moves. Which is fine – it's the search algorithm's job to see if you reached the goal with the least cost. The goal test just determines whether your current state is the goal state.

Successor State Functions

We want to find the solution through search, which means we need a function that tells us what our options are at each step – for our current state, what are the available (legal) actions and what will happen as a result of those actions. In other words, we need successor state functions (in this key, i use the terms successor state functions, operators, actions, methods and functions interchangeably). For this problem, our only actions are to move 1, 2 or 4 squares to the right:

```
move1(position): position = position + 1
move2(position): position = position + 2 when onWhite(position)
move4(position): position = position + 4 when onBlack(position)
```

As always, there is no standard way to represent successor functions. When writing them, pretend you're writing a set of requirements for some programmer to implement. The programmer lives in a cave and has no cell phone so he can't call you and ask you to clarify anything so you have to make sure what you write is completely unambiguous.

A few people have had problems understanding how to write successor functions. If none of the other formats made sense to you, i was willing to accept functions written in C. For example:

```
int move1(int position) { return position += 1; }
int move2(int position)
{
      //--- To save space, let's return noop rather than throw an exception
      if (!isWhite(position))
            return position;
      return position += 2;
}
int move4(int position)
{
      if (!isBlack(position))
```

```
            return position;
      return position += 4;
}
```

Even after the last homework and exercise, there were still several problems with the operators.

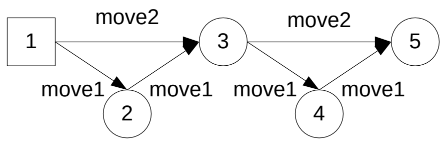
One problem is that people did not name their operators or list a purpose for them. If one were writing code, this would be the equivalent of writing the code for a function without actually writing the function name. What is search supposed to pick? i didn't take any points off but it seems an odd way to think about programming a computer.

Another problem was that constraints were left off. You cannot move 4 when you are on white. If you don't say when a function is legal, it's presumed to always be legal and your search ends up with an illegal solution (or crashes).

A third problem was saying that there was an operator called `move(numerOfSpaces)` that took an arbitrary move value. Assuming our search program was smart enough to make up parameters, the search algorithm could call `move(19)` and immediately solve the problem. You have to give the search algorithm a finite set of legal moves or else your branching factor becomes infinite.

1.2. Is the search space a tree or a graph?

This problem is a graph. Specifically, it's a DAG (directed acyclic graph). It has a finite length, you can only move in one direction and there are no loops. The search space for the problem [W,W,W,W] is:



There are four possible solutions: [move2, move2], [move2, move1, move1], [move1, move1, move2] and [move1, move1, move1, move1].

3 points were taken off if you said it was a tree.

1.3. What is the branching factor?

Two. At any given node, you only (and always) have two legal actions.

4 points were taken off if you thought the answer was four. The branching factor is how much wider the tree will grow per node at each level. Here, that means how many choices you have. If you are on white, your choices are move1 and move2. On black, it's move1 and move4. Either

way, a given state has two possible paths leading out of it. Therefore the answer is two. The branching factor is in no way related to the number of spaces you move.

1.4. Propose a non-trivial heuristic for the problem. Is it admissible? Monotonic?

The goal is to move off the edge of the world in the fewest moves possible. The heuristic, then, must estimate, for a given position, the number of moves to the goal. If it can't guess it precisely, it must guess low.

Answer 1
Given the three actions, the largest number of moves you can make is four. One heuristic, then, is to divide the distance to the goal by four, rounding down. We'll define the goal as the square right after the end of the state space, although the real goal might actually be 1-4 spaces beyond the end, depending on what our final operator is. In the first example above, we start on square 1 and the goal is square 19. Our heuristic would be:

$$h = \left\lfloor \frac{(N+1) - position}{4} \right\rfloor = \left\lfloor \frac{19-1}{4} \right\rfloor = 4$$

and our cost would be:

$$f = g + h = 0 + 4 = 4$$

This heuristic is admissible because it returns a number that's either equal to or less than the real answer (which, in our example, is six). It is consistent/monotonic because the estimated cost to the goal from a given state is less than or equal to the sum of the cost of getting to the next node plus the estimated cost to the goal from that node. Meaning, as you get closer to the goal, the f cost doesn't decrease. In our example above, at position one, the f cost is 4. If we move one square to position two (a black square), our f cost is $f = g+h = 1+\lfloor (18-1)/4 \rfloor = 1+4 = 5$.

Answer 2
Several people gave the answer `h=(N+1)-position`, where N is the number of squares. This equates to the solution of moving right one square at a time, which also happens to be the worst possible solution. It's like paying full price when you have a pocket full of coupons. There is *always* a better solution – even a problem with nothing but white squares can solve the problem in `N/2` steps. This is not a heuristic anyone should have used.

i debated whether this should count as one of those trivial heuristics Maria said not to use. In the end, i let people use it since the problem didn't ask for an admissible solution. However, to get full credit, you had to point out that this solution was inadmissible – it *always* overestimates by a minimum of twice the actual cost.

i'm a big fan of testing theories. Had people plugged some numbers into this heuristic, it would have been obvious that this heuristic is in admissible. Consider the example given in the problem. The number of squares, N, is 19. At the start of the problem, you're on square one. Therefore, `h = 19-1 = 18`. If the best case, every square is black, letting you move four squares at a time, so the shortest the path could be is `18/4 = 4½`, so 5 moves. In the worst case, every square is white, letting you move two squares at a time, so the shortest path is `18/2 = 9`, so 9 moves. Therefore the optimal solution to the problem must be between five and nine moves. The `N-position` heuristic gives 18, which is twice as large as the largest possible optimal solution. It overestimates. To be admissible, you need to underestimate, so this heuristic is inadmissible.

3 points were taken off if you used this heuristic and said it was admissible.

Other Answers
A few people described other heuristics. As the question mentions, these heuristics have no be non-trivial. The example given for a trivial heuristic was `h=0`. On the exam, i had people give the heuristic `h=1`. Changing the 0 to 1 does not make the heuristic better. Some people gave `h=N`. This is both trivial (it's a constant) and inadmissible (it overestimates in every state but start). Another choice was `h=C*`, where C* is the optimal answer. That one made me cry.

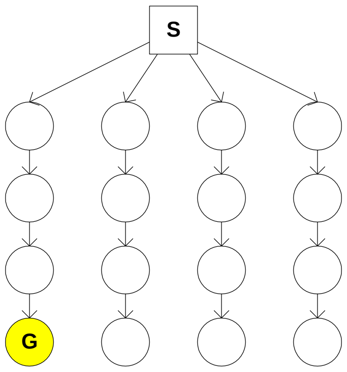
2. Search

10 points [graded by Matt]

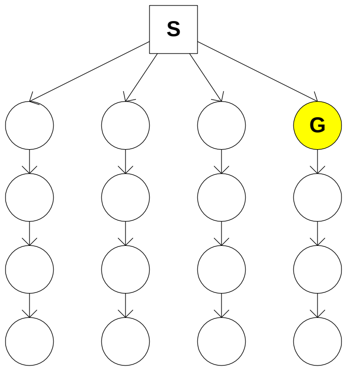
2.1. Can you construct a graph in which A* will expand more nodes than depth-first search? Draw and explain an example.

Yes. A* can expand more nodes if its heuristic is non-monotonic, since the first child, the one expanded by depth-first, might have a larger h value than the sibling nodes. A* might end up expanding many other nodes before choosing the first child.

How A* behaves depends on the heuristic it uses. If h is a constant number, A* will behave like uniform cost search. Depth first will find the solution faster (and expand fewer nodes). For example, in the following tree, assuming all link costs are 1,



depth-first search will expand four nodes and A* (acting like uniform-cost) will expand 13. The situation is reversed for the following graph:



A common mistake in this problem was to say that A* is optimally efficient and wouldn't expand more nodes than any other algorithm – but that's not what optimally efficient means. Optimally efficient means that no other search algorithm is *guaranteed* to expand *fewer* nodes than A*. Under circumstances, like those described above, A* will expand more nodes than other algorithms.

2.2. Could the same happen for uniform-cost? Explain.

There were a couple ways to interpret this problem. The intended way was to see if uniform-cost would expand more nodes than DFS:

Yes. Uniform cost can get into the same problem if the cost of the initial step in the depth first solution is much higher than the cost of the siblings. They will get expanded until the first child has a smaller cost.

Another way to interpret this problem was to see if A* would expand more nodes than uniform-cost:

Again, the answer is yes. On certain search graphs, A* could have a non-admissible heuristic that could steer it away initially from a more direct path that uniform-cost would explore first.

3. Agent Design

`10 points [graded by baylor]`

`Why is it better to design performance measures for an agent according to effects in the environment instead of behaviors of the agent?`

Behaviors are non-deterministic – they don't always result in the intended effect. If an agent's behavior always results in the intended effect, it might be OK to measure just the behavior. But for most things, especially anything in the real world, behaviors sometimes fail. Suppose you have a robot that's trying to get to class. Maybe the robot tries to open the door by turning the door handle and fails. Maybe the robot thinks it's on the second floor and it's wrong. Maybe the robot gets caught in a bear trap. Behaviors and actions can fail.

A different, but common, answer was that, if you measure the behaviors, you might get some bad behavior. The common example was a vacuum cleaner that got points for picking up dirt. It had an incentive to pick up all the dirt, dump it back on the ground and pick it up again. A better measure, they argued, was to measure whether the environment was clean. i'm not convinced this answer is correct (if you used different behavioral and environment measures, the outcomes might be very different) but i realize the bad vacuum cleaner is an example Maria covers in class and is in the book so i gave it full credit.

One relatively common answer is that behavior is only rational in a given context and that context is the environment. As an example, a robot can take a short path or a safe path and the robot might be biased to take the shortest path, which presumably is bad. This answer isn't quite right, for two reasons. First, judging which action an agent takes in a given context is a judgment of behavior, not environment. A context-sensitive behavior is still a behavior. Second, in this example, there is no clear environmental measurement. If the goal is to get somewhere and two behaviors achieve the same thing in different ways, what in the environment would you measure?

4 points was taken off if your argument was secretly arguing for behavior and not environment.

A few people argued that it was better to measure behaviors, not environment. For example, suppose you had a vacuum cleaner that had the two actions vacuumRoomQuickly and vacuumRoomEfficiently. Both kept vacuuming until the room was clean (or, alternately, they went back and forth until the entire room was traversed, leading to comprehensive coverage). One did so quickly, the other used less battery power. The end result is the same, so there is no environmental measure you can use. The difference is in the behaviors, not the environment, so it makes sense to judge the behaviors. The question didn't give you the option of saying "they should measure the behaviors, not environment" but if you made a good case for it i gave you credit.

Some people said that you measure the environment rather than the behavior because the agent doesn't control the environment or because the agent might not do well in one environment but not another. These answers are vague and potentially wrong (depending on what the student meant to say). Neither of these answers show that the student understands AI, which is bad as that's the goal of answering any AI test and homework questions.

As this question was worth 10 points, 6-8 points were taken off if an answer was vague.

4. Search

25 points [graded by Matt]

4.1. Why any node in OPEN with f(n)< C* (the cost of the optimal solution path) will eventually be selected for expansion by A*?

The key factors:

- The open list is a list of nodes sorted by f values, smallest to largest.
- The f values are a mixture of actual cost (g) and estimated cost (h). Because the estimated cost is less than or equal to the actual cost, the f value is less than or equal to the actual cost.
- When you first start a search, actual cost (the amount you've moved) is 0 and f purely reflects the estimated cost. As you move closer to the solution, the actual cost grows and the estimated cost shrinks until the goal is found, at which point the estimated cost is 0 and the f cost is equal to the actual cost (g). Therefore f(n) = g(n) = C*.
- A* processes nodes one at a time, always selecting the first/top (lowest valued) node.

Since the solution node will have an f value equal to the actual cost C*, it will be placed in the open list below those nodes that have smaller f values. Since A* always processes the first (lowest cost) node in the list, all nodes with an f value less than the actual value C* must be processed before nodes with a cost C* can be.

4.2. Why does A* remain admissible if you remove from OPEN any node with f(n) > C*?

The solution here is the same as on the answer key for last week's exercise.

4.3. Is it true that all admissible heuristics are equal, in the sense that A* will search the states in the same order no matter what the admissible heuristic is?

No. Consider a map navigation problem with two heuristics. h1 is the straight line (Euclidean) distance, h2 always returns 0. h1 will cause the search algorithm to go directly to the goal, ignoring all nodes not on the solution path (assuming no obstacles). h2 will cause the search algorithm to become uniform cost search. When all costs are equal, this will cause the algorithm to become breadth first search, which floodfills out from the start node. It eventually finds the goal but only after first evaluating every single node closer to the starting position than the goal is.

4.4. In what sense is IDA* preferable to A*?

IDA* uses less memory. Which is good as A* can be quite memory intensive.

`4.5. Is breadth-first search complete if the state has infinite depth but a finite branching factor?`

Yes. If the width is finite, breadth-first will complete that level and move to the next. Since the solution must be on one of the levels, breadth-first search will eventually reach that level and find the solution.

Many people said something along the lines of "no, if the goal is infinitely deep, BFS will not find it". This is wrong because completeness refers to the algorithm finding a solution *if one exists*. If the goal is infinitely deep, it essentially does not exist.

5. Agent Design

10 points [graded by baylor]

Write a function in Lisp to split a list in two sublists, one containing all the positive elements from the list, and the other with the negative elements.

```
;;; split a list in two sublists, one with the positive
;;; elements, the other with the negative.
;;; Non numbers should be discarded.
;;; (split '(8 -2 6 -1 -3)) => ((8 6) (-2 -1 -3))
;;; (split '(2 4 3)) => ((2 4 3) nil)
;;; (split '(2 a -3)) => ((2) (-3))

;;; iterative definition.  It is concise and easy to read.
(defun split (lst &aux (pos nil) (neg nil))
  (dolist (x lst)
          (if (numberp x)
              (if (> x 0)
                  (setf pos (cons x pos))
                  (setf neg (cons x neg)))
              ))
  (list (reverse pos) (reverse neg)))


;;; recursive definition using an auxiliary function
;;; many students wrote something similar to this.  It is efficient
;;; because it scans the list only once.
(defun split (lst)
  (let ((pos nil) (neg nil))
       (split-aux lst pos neg)))

(defun split-aux (lst pos neg)
  (cond ((null lst) (list (reverse pos) (reverse neg)))
        ((numberp (car lst))
         (if (> (car lst) 0)
             (split-aux (cdr lst) (cons (car lst) pos) neg)
             (split-aux (cdr lst) pos (cons (car lst) neg))))
        (t (split-aux (cdr lst) pos neg))))


;;; definition using a filter function (as done often in Scheme)
;;; this is a style of solution a few students have used
;;; it is not the most efficient, since it scans the list multiple
;;; times, but it is easy to write and understand
(defun split (lst)
  (list (filter #'positivep (filter #'numberp lst))
        (filter #'negativep (filter #'numberp lst))))

;; we can filter for numbers only once and be more efficient
(defun split (lst &aux (fltlst nil))
  (setf fltlst (filter #'numberp lst))  ; check for numbers once
  (list (filter #'positivep fltlst)
        (filter #'negativep fltlst)))
```

```
(defun filter (fn lst)
  (cond ((null lst) nil)
        ((funcall fn (car lst)) (cons (car lst) (filter fn (cdr lst))))
        (t (filter fn (cdr lst)))))

;; these are not predefined in Lisp
(defun positivep (x) (> x 0))
(defun negativep (x) (< x 0))


;;; definition using remove-if/remove-if-not built-in function
;;; remove-if is non destructive (contrary to delete-if)
;;; this style of solution has been used by a few students.  It is
;;; a good way to use a built-in function and accomplishes what the
;;; solution above using filter does without having to define filter
(defun split (lst &aux (fltlst nil))
  (setf fltlst (remove-if-not #'numberp lst))
  (list (remove-if-not #'positivep fltlst)
        (remove-if-not #'negativep fltlst)))
```

Common mistakes:

- You have to remember to return the result, i.e. the list of the positive numbers list and the negative numbers list.

- When returning the result you need to use list as in `(list pos neg)` instead of cons since `(cons pos neg)` will not return what you want.
  Example: `(cons '(8 6) '(-2 -1 -3))  => ((8 6) -2 -1 -3)`
  If you prefer to use cons recall that
  `(list pos neg)` = `(cons pos (cons neg nil))`

- If you use local variables to construct your positive and negative lists you have to use setf to assign the new value to a list every time you cons a value to it. Doing (cons x pos) does not change the value of pos. Using local variables with recursive calls in this case does not work, since you get new variables for each recursive call.

- If you write a recursive function you need to use an auxiliary function with additional arguments for building your positive and negative lists.

- You should not use append when attaching a new element to a list. Use cons and reverse the result if needed. It this case the question did not ask for the result to contain the elements in the order in the original list, so it was not necessary to reverse the positive and negative lists.

- A number of students wrote something like this:

```
(defun split (lst)
  (if (null lst)
      nil
      (if (numberp (car lst))
          (if (> (car lst) 0)
              (cons (car lst) (split (cdr lst)))
              (cons (car lst) (split (cdr lst)))))))
```

The problem with this function is that it returns the original list.
To see why just look at what the recursive calls do.

- I have seen a number of mistakes in parentheses and syntax but did not take off any points.

- I have seen a few programs where closing parentheses are listed one per line. I know that many TAs use this style in Scheme, but it is not the way professionals write programs. All the closing parentheses should be on the same line. Adding too many lines makes programs harder to read because they take too much space.