Instructor: R. Greiner
Due Date: Thurs, 4 October 2007 at start of class

The following exercises are intended to further your understanding of agents, policies, search, constraint satisfaction techniques, GSAT and other stochastic algorithms. Of course, be sure to explain your answers, etc etc etc.

Total points: 70 + 50 = 120

Submission: You should hand-in hardcopies of Problems 1 to 5. For Problem 6: use ASTEP to submit a gzipped tar file named as1.tgz such that unpacking it creates a directory called as1 with the following subdirectories that contain your solutions:

<table>
<thead>
<tr>
<th>Subdirectory</th>
<th>Contains your solution to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>as1/p6a</td>
<td>Problem 6(a)</td>
</tr>
<tr>
<td>as1/p6b</td>
<td>Problem 6(b)</td>
</tr>
<tr>
<td>as1/p6c</td>
<td>Problem 6(c)</td>
</tr>
<tr>
<td>as1/p6d</td>
<td>Problem 6(d)</td>
</tr>
</tbody>
</table>

Note each coding sub-question is in its own sub-directory. Typing ‘make’ in each subdirectory should compile the program. The name of the executable should be the same as the name of the subdirectory.

Problem 1 [10 points] (modified from [Russell/Norvig:Exercise 2.9 – p57])
The vacuum cleaning agent VCA described in [Russell/Norvig:Section 2.4] has 6 actions: it can move \{ Left, Right, Up, Down \}, can Suck, which picks up any dirt that appears at the position, and can TurnOff. Moreover, it can sense the dirtiness of (only) the square it is occupying. It operates in an otherwise-empty \( n \times m \) rectangular room (with no furniture or other obstacles), where each square initially has a 5% chance of containing dirt. VCA starts from the home \( (1, 1) \) position, and its goal is to (1) have a dirt-free room, and (2) be in the home \( (1, 1) \), with the power turned off. You may assume a square, once cleaned, will remain clean.

a [5]: Explain why it is impossible for a simple reflex agent reliably do this? What is the best a reflex agent can do?

b [5]: Now suppose the agent can have a “state”. Can it solve the problem now? If so, how many bits does it need?

(You may assume that the agent can “perceive” its \( (x, y) \) location, by measuring its distance to each wall.)
**Problem 2 [15 points] Uninformed Search**

a [5]: Construct a finite search tree for which it is possible that depth-first search uses more memory than breadth-first search. (Be sure to show the goal node(s) in your tree.)

Is there any tree and distribution of goal nodes for which depth-first search *always* requires more memory than breadth-first? Briefly justify your answer. Your search algorithm needs to include enough information to identify the path to the solution.

*Notes: Node ordering is a property of the search algorithm, not the tree. Trees are not necessarily binary. Do not confuse the asymptotic claim associated with $O(f(b,d))$, with the function $f(b,d)$...*

b [5]: (from [Russell/Norvig:Exercise 3.17b]) When considering uniform-cost search, [Russell/Norvig:page 62] disallows negative arc costs. Suppose there is a set of operators that form a loop, so that executing the set in some order results in no net change to the state. If all of these operators have negative cost, what does this imply about the optimal behavior for an agent following uniform-cost search in such an environment?

c [5]: (from [Russell/Norvig:Exercise 3.13]) Describe a class of search spaces in which iterative deepening performs much worse than depth-first search.

**Problem 3 [15 points] Heuristic Search**

Consider the graph

![Graph with labels](image)

with start state $S$ and goal state $Z$. The numbers on the arcs indicate the cost of traversing that arc.

a [5]: Seek the best (least cost) path, using uniform-cost search. (1) List the nodes in the order they would be expanded; (2) list the nodes that lie along the final correct path to the goal.

b [5]: Same as (a), but using greedy best-first search, based on the following $h(n)$ function:

<table>
<thead>
<tr>
<th>$n$</th>
<th>$S$</th>
<th>$F$</th>
<th>$E$</th>
<th>$C$</th>
<th>$D$</th>
<th>$B$</th>
<th>$A$</th>
<th>$G$</th>
<th>$Z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h(n)$</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(1)

c [5]: Same as (a), but using $A^*$ search with the heuristic function shown above in Equation 1.
Problem 4 [15 points]  Adversarial Search
Consider the following simple game. There are five (5) pennies on the table. You and your opponent take turns picking up 1, 2, or 3 coins until none is left. You get to keep each penny you pick up. But, if you pick up the last coin, you have to pay 2 cents to your opponent. The object of the game is to finish with the most money.

a [3]: Consider building a game tree to solve the above problem. What would each state in the game tree represent? What is/are the operator(s)? (Can you go into debt here?)

b [5]: Show the (complete) resulting state tree for this problem. Each node should have an entire state description.

c [2]: What cost function (i.e., static evaluation function), do you use to evaluate terminal nodes?

d [3]: Show the (complete) minimax state tree for this problem; be sure to back up the values, etc.

e [2]: Your opponent courteously offers to let you go first. If you accept, what is your first move? How much will you earn, assuming optimal play?

Problem 5 [15 points]  WalkSat/ GSAT
Recall a clause is a disjunction of literals, where a literal is a Boolean variable or its negation. The core loops of GSAT is

\[
\text{while (existsUnsatisfiedClauses) }
\{
\quad \text{Flip the variable that produces the LARGEST number of satisfied clauses.}
\quad \text{// What if there's a tie? You decide.}
\}
\]

and of the general Mixed-WalkSat is

\[
\text{while (existsUnsatisfiedClauses) }
\{
\quad \text{With probability } p:\n\quad \quad \text{Flip the variable that produces the LARGEST number of satisfied clauses.}
\quad \quad \text{// What if there's a tie? You decide.}
\quad \text{With probability } 1 - p:\n\quad \quad \text{Flip a variable in a currently-unsatisfied clause.}
\}
\]

The “Pure-WalkSat” routine has \( p = 0 \). Each of these algorithms stops as soon as it hits the first satisfying assignment. For this problem, you should assume no random restarts, etc.

a [5]: Give a set of satisfiable clauses that would be difficult for GSAT to satisfy, starting from a random truth assignment.

b [5]: If a formula is satisfiable, can Pure-WalkSat always reach a satisfying assignment, starting from a random initial assignment? Can mixed-WalkSat?

c [5]: Can Pure-WalkSat reach \( \text{all} \) satisfying assignments starting from a random initial
Problem 6 [50 points] Implementation: GSAT and WalkSat

a [20]: Write and test the GSAT algorithm, extending the code provided in
http://ugweb.cs.ualberta.ca/~c366/gsat.c
This code, which implements the basic input/output functions, etc., takes as input a file of
the form

c this is a comment!
c The following formula is (easily) satisfiable
p cnf 5 7
-1 2 -3 0
1 -2 3 0
2 3 5 0
-1 2 -5 0
-3 -4 -5 0
-1 -3 4 0
-1 3 -4 0

Any (preliminary) row starting with “c” is a comment, to be ignored. The row starting
with “p cnf” states that this is a CNF formula; the following numbers means there are 5
variables (x_1 through x_5), within 7 clauses. Each of the next 7 rows represents a clause,
encoded as a list of non-0 integers, followed by 0. A positive value i indicates the positive
form of the variable x_i is included, and a negative number means the negation of the variable
\neg x_i; hence the first row “-1 2 -3 0” represents the clause \neg x_1 \lor x_2 \lor \neg x_3.

There are several subtleties to worry about: How many plateau walks before restarting?
How many restarts before giving up? During a plateau walk, there may be many different
variables that, once flipped, produce assignments with equivalent scores. Be sure to pick one
“fairly” — and in particular, avoid simply flipping var#1 each time, or avoid simply flipping
var#1, then var#2, then var#1, etc. (Hint: you may want to pick one of these variables at
random.)

If you want some challenge problems, look at
ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/benchmarks/cnf/
(Nota the code provided can read in these formulae.)

b [15]: Implement Mixed-WalkSat
Of course, it has to take the same i/o as the other algorithms. Again, feel free to use the
code provided for the I/O functions, etc.

c [10]: Compare these two algorithms over a range of (satisfiable) problems.

d [5]: Write your own algorithm BESTSat, for solving general SAT problem. It should
have the same I/O as GSAT (i.e., take the same input and return the same response). It
can be stochastic or systematic, and use any idea you wish.
We will then compare your algorithm with the ones written by others in the class, and announce the winner(s).

While bragging is fun, please note this problem is worth only 5 points, out of 50 (well, 120 for this assignment!) and so allocate your time accordingly.

Good luck!