CIS 391/521 Spring 2012 : HW 2 - Search

1 Instructions

Due at 10:30 a.m. on Tuesday, Feb 7th. All submissions will be via Blackboard. There are two parts to this homework. Part 1 is short answer questions that are to be written up individually (.pdf or .txt formats are fine; please do not use .doc or .docx, as incompatibilities can arise). Part 2 is python programming that may be completed in pairs (no more than 2 per group). For the python programming you will submit (a) answers to all the questions as described below (b) a .zip file of the code you used to generate it.

Let us know if you have any questions, check the discussion board, and remember that you can always come to Office Hours, even if only to keep the Professor and TAs company.

2 Written Portion (35 points)

1. Consider the following problem: You have two jugs of water with capacities 4 and 13 liters. You also have an infinite supply of water. Can you use the two jugs to get exactly 2 liters of water? Cast this as a search problem: what is your state space, initial state, action space, goal condition, and costs of actions?

2. Describe a search space in which iterative deepening search performs much worse than depth-first search. In this situation, give big-O descriptions of the running time for iterative deepening and for depth-first search.

3. Sometimes there is no good evaluation function for a problem, but there is a good comparison method: a way to tell if one node is better than another, without assigning numerical values to either. Show that this is enough to do a best-first search. (a) Describe in words and pseudo code how you would implement this method. (b) Show what the differences in time and memory would be (if any) compared to a standard greedy search where you know the particular value associated with each node.

4. There are n people 0,1,...,n-1. They all need to cross the bridge but they have just one flashlight. A maximum of two people may cross at a time. It is nighttime, so someone must carry the single flashlight during each crossing. They all have different speeds such that the time taken to cross the bridge for person i is given by $T(i)$ minutes. You may assume without loss of generality that for $i < j$, $T(i) \geq T(j)$. The speed of two people crossing the bridge is determined by the slower of the two. You need to find the shortest amount of time in which all the people can cross the bridge.
   As an example suppose there are 4 person and time taken $(T(i))$ are 1,2,5 and 10 for $i = 0, 1, 2, 3$ respectively. A solution to the problem is:

   (a) 0 and 3 go across 10 minutes

(b) 0 goes back with light 1 minute

(c) 0 and 2 go across 5 minutes

(d) 0 goes back with light 1 minute

(e) 0 and 1 go across 2 minutes

(f) Total 19 minutes

Note: This is not the optimal solution for this problem.

(a) Describe three possible heuristic functions for this search problem, and state for each whether it is admissible and whether it is consistent. You should include at least one consistent heuristic function.

5. Which of the following are admissible, given admissible heuristics h_1 and h_2?

   (a) $h(n) = min(h_1(n), h_2(n))$

   (b) $h(n) = w\, h_1(n) + (1 - w)h_2(n)$, where $0 \leq w \leq 1$

   (c) $h(n) = max(h_1(n), h_2(n))$

6. Which of the following are consistent, given consistent heuristics h_1 and h_2?

   (a) $h(n) = min(h_1(n), h_2(n))$

   (b) $h(n) = w\, h_1(n) + (1 - w)h_2(n)$, where $0 \leq w \leq 1$

   (c) $h(n) = max(h_1(n), h_2(n))$

7. Assume we have a very large search space with a very large branching factor at most nodes, and we do not have any heuristic function. What search method would be good to use in this situation? Why?

   The above questions are taken from AIMA, but many have been modified.

2.1 Coding portion: Searching for Sudoku (35 points)

In the next section, you will be asked to implement two different types of solvers for the Sudoku problem introduced in the previous homework: uninformed graph based search and simulated annealing. For graph based search, you will represent the problem much like humans do: a given state is just a partially filled in solution to the initial Sudoku board. *Please note: your answers to these questions can be very brief. One sentence should suffice for each.*

1. Suppose that your successor function is to choose the next available square (reading the board left-to-right, up-to-down, like English) and write down a number in that square. What is the branching factor of this successor function?

2. Will Depth First Search (DFS) be **finite** on the Sudoku problem? Let r be the number of empty positions on the initial Sudoku board. What is the maximum search depth of DFS in terms of r?

3. What is the **best-case** run time complexity of DFS, in terms of r? What is the **worst-case?**

4. What are the **best-case** and **worst-case** run times for Breadth First Search (BFS)?

5. We saw in class that the (worst-case) **space** complexity of BFS is often quite large. In the case of Sudoku, suppose you try to run BFS for a board with $r = 40$ on your laptop. Assume (optimistically) that storing a state in memory requires only 1 byte. How much memory might your laptop need by the time it reached the solution?

6. In simulated annealing, we use a **local** representation of the problem. For Sudoku, the local representation is a complete assignment to the r originally empty positions on the board. What is the size of this entire state space, in terms of r?

7. What is a successor function you might use for simulated annealing?

3 Programming Portion

For this part of the assignment, you will write two Sudoku solvers in Python, and test them on real Sudoku problems. It is mostly up to you how to code them, as we will not be running any automated tests.

Important This assignment should be done in pairs. Only one version of the homework should be submitted from each group. The zipped file should be named "[person1's pennkey]_[person2's pennkey].zip"

3.1 Uninformed Graph Search

1. Implement Depth First Search (DFS) and Breadth First Search (BFS) using a FIFO/LIFO queue-based implementation (do not use a recursive implementation of DFS). Your state space and successor function should be similar to those that a naive human would use: a state is a partially filled in solution to the initial Sudoku board, and your successor function should fill in numbers one at a time, reading the board from left-to-right, top-to-bottom.

   As a truly uninformed search, this implementation should **only** check whether or not the next state on the queue is a valid, complete solution: if it is a solution, then the algorithm should **return** that solution, and if it is not, then the algorithm should **generate successors** (all numbers 1-9, even those that checking constraints might allow you to rule out) for that state and add them to the queue.

   Finally, the solver should keep track of and report two statistics: the number of times the successor function was called (**run-time** complexity), and the maximum queue length over the run of the entire algorithm (**space** complexity).

2. Run the uniformed BFS and DFS algorithm on `tiny.txt` and report time and space complexity.

3. There is a simple modification we can add to the graph based algorithms to take advantage of the structure of the Sudoku problem: given a partially filled board that *already* violates one of the Sudoku constraints (e.g., uses the same # more than once in a row, column, or square), we know that no successors of this board will *ever* be a solution to the Sudoku problem.

   Modify the DFS solver you implemented as follows. When checking the next state on the queue, the algorithm should **not expand** that state if the partial solution is found to violate any of the Sudoku constraints.

4. Re-run the DFS experiments from part 2 and report the new time and space complexities. Did you find any improvement? (Hint: it should be dramatic.) Run your **informed** algorithms on `easy.txt` and `medium.txt` and report time and space complexities.

5. Implement BFS by applying the same modification from part 3. The algorithm should not generate successors that violates any constraints. Run the algorithm on tiny.txt, easy.txt, and medium.txt and report time and space complexities. Did you find any improvement from normal BFS?

6. **Optional**: Try your informed solver on some **real** Sudoku boards and report the results. You can find a selection in the file `optional_boards.zip`.

7. **Hint**: The `copy` module has a useful function `deepcopy` to make deep copies of objects. The `sys` module allows you access to `sys.argv` (and therefore arguments passed on the command line.) The `Queue` module has a `Queue` object which is more efficient than using a `list` for a FIFO queue.

3.2 Simulated Annealing

Recall that in simulated annealing, we use a **local** representation of the problem. For Sudoku, the local representation is a complete assignment to the r originally empty positions on the board.

1. Implement simulated annealing for Sudoku problems. For your cost function, you should compute the number of "unused numbers" across all constraints on the board. That is, **for each constraint** (block, row, and column), you should compute how many numbers from the range $\{1, \ldots, 9\}$ are **not** being used up by the current board assignment, and then add these values **over all constraints.** For example, if a given row contains the values $\{1, 5, 2, 3, 3, 3, 6, 7, 5\}$, then the numbers 4, 8, and 9 are not being used, so this row constraint would add 3 to the total cost of the board assignment. Clearly, any solution to the Sudoku board will be optimal with zero cost.

   Furthermore, unlike Figure 4.5 in the textbook, you should not allow your algorithm to run indefinitely. Instead, your implementation should take as input a maximum number of iterations, T, and use the following schedule for randomization:

   $$p(t) = 0.99 \cdot p(t-1),$$

   where $p(t)$ is the probability of taking an "uphill" step on round t.

   Finally, initialization and successor function are key to the success of the annealing algorithm on this problem. You should initialize your annealing algorithm by randomly filling in unfilled squares so that the **column constraints** are all satisfied. (Since they are independent constraints and do not overlap, this should be easy). The successor function should then choose two **non-fixed** squares on the board and **swap** their values.

   Feel free to explore other initializations, successor functions, and cost functions if you desire (this algorithm can be much improved), but **you must *at least* report results using this exact algorithm to get full credit.**

   Your algorithm should record the following statistics: the number of "downhill" moves (greedy steps to minimize the objective), the number of rejected "uphill" moves, and the number of accepted "uphill" moves.

2. Run your simulated annealing code on `tiny.txt`, `easy.txt`, and `medium.txt`, with $T = 10^4$. Please run your algorithm at least three times on each of these T values, with different random seeds each time. Report whether the total number of steps for each run of the algorithm and whether or not the algorithm returned a solution, along with breakdowns into uphill and downhill steps.

3. **Hint**. The `random` package will be necessary for the randomized steps of the simulated annealing algorithm. A simple way to implement "with probability p, do A" is to sample a number q uniformly from $[0, 1]$ and take action A if $q \leq p$. Make sure to come to office hours if you feel you don't know how to get started on these problems.

4. **optional bonus question** There is a choice in implementing simulated annealing of what to do when a proposed change is entirely neutral. One could, in the limit, either reject or accept all such changes. (What does the algorithm above do?) Does learning happen slower or faster if you accept neutral changes? Why might you be getting this effect?