Problem Set 2: Game Search, GAs, Constraints

6.034 Fall 2004

`http://www.csail.mit.edu/courses/6.034f/`

Due: Tuesday, October 12th, 2004

(We strongly recommend you have 1-5 done by 10/08.)

For this problem set, either download ALL the files from the `ps2` *directory, or download an unpack the* `ps2.tar.gz` *file. Fill in your answers in* `ps2.scm`*; this will be the only file we collect.*

While wrestling with rules and search at Ur-Bits (and Louis Reasoner's maddening management style), Ben Bitdiddle picked up Go as a displacement activity. After completing Louis's assignments, Ben decided to take a good long vacation in Tahiti, bringing along only his Go set, his laptop, and a bottle of fine gin, and apply himself equally to the study of each. Being a hacker at heart, Ben rapidly came to the conclusion that the key to his success - and his freedom from Louis's tyranny - lay in the development of a champion computer Go player.

Before diving into Go, however, Ben decided he should dust off his 6.034 game search skills, by applying them to much simpler problems. Ben decided to warm up by remembering exactly how the minimax algorithm worked, and began furiously scribbling in the sand with his finger.

Problem 1: Minimax

Give the result of performing minimax on the following trees, assuming that the player at the root node is trying to maximize her score. Note that a tree of uneven depth might occur if the search has heuristics that can prune certain branches a priori (e.g. a blunderstopping device/novice strategy in chess telling you to never put your queen at risk) but this doesn't change the fundamental structure of the algorithm. Also note that you are not required to code up solutions to this problem, but problem 4 asks you to implement `minimax`*, and you can use it here if you want.*

1. `(((1 4) (3 6)) ((8 4) (2 7)))`
2. `((((4 6) (8 12)) ((13 2) (7 1))) (((9 5) (6 8)) (2 6) (14 7)))`
3. `((2 1) ((4 7) (6 (8 2))))`
4. `((4 8 7) ((2 3) 9) (10 (0 11)) ((14 6) (12 2)))`

Some of the other beachcombers were starting to give Ben odd looks, possibly because Ben was wasting effort by using an algorithm that must consider every leaf node. "Yeah, yeah", Ben muttered to himself, "I'll fix you all." He got to work on remembering alpha-beta pruning.

Problem 2: Alpha-Beta Pruning

For the following trees, give a list whose first element is the list of values the static evaluator would produce, in left-to-right order, and whose second element is the final score of the tree, if the tree was processed by the alpha-beta pruning algorithm. Again assume the player at the root node is trying to maximize his score. Look at the first test case if you are confused.

1. `((7 3) (4 7))`
2. `((9 8 7) (6 5 4) (3 2 1))`
3. `((1 2 3) (4 5 6) (7 8 9))`

"There! Finished! Now on to the real stuff!" Ben said, standing up, stretching, and gazing at the swath of sand he'd covered with scribbles. All of a sudden, however, he felt a tap on his shoulder. Turning around, he noticed the glint of a brass rat, and, looking up, the friendly face of Alyssa P. Hacker, grinning at him and arching an eyebrow.

"Enjoying your vacation?" She asked.

"Yeah, sure," Ben affirmed, motioning vaguely at his spot on the beach, his bottle of gin, and his handiwork (in that order). "I just finished with basic alpha beta pruning, and I'm ready to tackle Go."

"Ha! Alpha Beta has its subtleties, you know. I wonder if you remember them," Alyssa challenged. They were soon lost in conversation.

Problem 3: Game Search Questions

Answer the following true/false questions; fill in `#t` *if the answer is true, and* `#f` *otherwise.*

1. When using minimax with alpha-beta pruning, rearranging the branches coming from a given node has no effect on the eventual answer that is reached.
2. When using minimax with alpha-beta pruning, rearranging the branches coming from a given node has no effect on which leaves must be statically evaluated.
3. The use of alpha-beta pruning can never reduce the total number of static evaluations needed by more than a factor of 2 (from the number needed by pure minimax).
4. On any given search tree, the time spent using progressive deepening to ensure that an answer is always ready is always about one-half of the total computation time.
5. Even for a simple game such as Tic Tac Toe, a human being cannot feasibly draw out a search tree representing every possible sequence of game states on pencil and paper (in a reasonable amount of time, say as part of a final exam question).

While Ben and Alyssa argued, the sun set and night fell. Sighing at Ben, Alyssa told him she had to leave, and that she hoped she'd convinced him Go was out of his reach, at least for the moment. Ben remained, spending a while staring glumly at the waves, before deciding he would implement minimax search to make sure he knew what he was talking about. Trained to work through the break of dawn, Ben shook off his weariness and his encroaching hangover and sat down to code.

Problem 4: Implementing Minimax Search

Implement the minimax function, assuming it is only externally called by a player seeking to maximize his/her score. It should take a tree (of the form from problem 1) as an argument and return a list of the maximum score attainable and the branch to take to get there (as valid input to `list-ref`*). See the public test cases if you're confused. If minimax is called on a number, the branch to report is* `-1`.

Having completed and tested minimax, Ben decided he would go to bed and sleep on the issue of how to scale up to Go. Compulsively checking his email one last time before sleep, however, Ben noticed a message labeled "UNBELIEVABLY URGENT: GAs ARE THE WAY AND THE LIGHT" from Louis Reasoner. Muttering about Louis' tendency to hit the bottle then send him email at wee hours, Ben decided to address the email before he went to bed.

Looking more closely, it appears Louis had read about genetic algorithms in *Daily Proactive Synergy: Buzzwords You Must Pretend to Know*, a dreadful management rag Louis had been obsessed with recently. He had come up with a scheme to use Ur-Bits's petty cash via online trading. His plan was to "evolve" a set of trading bots, whose code is represented by a simple bit string, to place trades. He offhandedly mentioned something about maybe treating the bit strings as piles of assembler instructions for a special Trading Bot Machine or "something like that", but "didn't elaborate because evolution would take care of all the rest".

Louis proposed "crossing" bots' programs as follows: Assume all the programs have length n. Select a random integer k ranging from 1 to n. Construct a "child" bit string from two parent bit strings A and B by selecting the first k bits from A, and the rest from B. Louis mentioned something about "needing random mutations to, you know, make progress", and proposed mutating a program by randomly flipping m of its bits, where m is the mutation rate. If pruning of the population becomes necessary, Luis suggested measuring the trading performance of each bot (say the average profit per day), and using that as the fitness metric; he didn't go into details of how exactly pruning would be done.

Louis wanted Ben to code up the GA search immediately. Ben rolled his eyes and realised he'd better disabuse Louis of his notions before GAs became his personal hell for the rest of his time at Ur-Bits (and watching the company go down the tubes in the process, unless they got exceedingly lucky). He started an angry email involving a few scenarios..

Problem 5: Genetic Algorithms

Fill in the blanks in your solutions scheme file for each of the following questions. True/False should be answered as for problem 3. If the question asks for a numerical answer, fill in the appropriate number. If the question asks for a yes or no answer, fill in the symbol yes (the text 'yes) for an affirmative answer, and the symbol no (text 'no) for a negative one.

1. Assume the current generation of bots at time $t = 0$ has two members, one described by the bit string 000 and the other described by the bit string 111. Also assume at each generation only crossover steps are applied, and in fact all possible crossovers (that is, for every pair in the population, all splits) are generated at each generation. How many more generations are needed to generate any 3-bit string?
2. Yes/No: Assume Ur-Bits' computer system can sustain a population of 2^{n-2} bots whose programs have bit-length n, even for reasonably large n (say 32). Assume the initial population has two elements, one whose program is all 0s, and the other whose program is all 1s. Without mutations, should Louis expect local minima to generally be a problem for the search?
3. True/False: Based on the information given, two bots whose bit strings are close according to the Hamming distance (that is, the number of bits that differ between them is small) should definitely exhibit similar average daily trading profits.
4. True/False: Based on the information given, two bots whose strings differ by only a single mutation should definitely exhibit similar average daily trading profits.
5. True/False: The structure of the mappings from bit string (genotype) to program (phenotype) to fitness will strongly impact the effectiveness of a GA search.
6. True/False: Based on the information given, a high-mutation-rate search with crossovers on a very small population (relative to bit size) will very likely perform better than a completely random search (generate many random bit strings; pick the best one).
7. Yes/No: Should Louis expect high mutation rates to help the GA search escape local minima (if the population is small relative to the bit string length)?

Sated, Ben Bitdiddle goes to sleep. The next day, he wakes up to find an email from his admiring cousin, Ike N. Graduate, who is entering his freshman year and planning to major in Course 6. Ike is sweating about all the course requirements the major presents; he wants to know how he can schedule his courses to ensure timely graduation, given e.g. constraints on the terms they're offered, prerequisites and such.

Ben glances at his trusty gin bottle, but finds it empty. He shrugs and decides he'll take the opportunity to brush up on constraint-based search while lounging on the beach. Rummaging around on his laptop, he finds code for part of a constraint-satisfaction engine dating back to his 6.034 days, but he can't remember exactly how it works. He decides to warm up by writing some basic arithmetic constraints.

After getting one completed example running, Ben decides to try to solve some simple constraint satisfaction problems, on the finite-domain, numerical variables a, b and c.

Problem 6: Simple Minded Net

Follow the instructions from the enumerated list below, in order. This problem is aimed at familiarizing you with the constraints package; none of the scheme you have to write is at all involved.

1. Look at the file `constraints-example.scm`. Evaluate it, carefully read the comments, and observe the results. It describes the constraints package you'll be using for the rest of the pset. Also read the `constraints-notes.txt` file.

2. Fill in the blanks to use constraint search to find a solution to the constraint network from the example file (look at the stub procedure `go-simple-net-with-search`. Note that this part uses no constraint propagation, but returns all consistent answers; it simply takes more work than the combination of search and constraint propagation.

3. Fill in the blanks to implement the constraint-satisfaction problem shown below. To do this, you will have to implement a new constraint type, `diff<threshold`, to represent the last two constraints, and insert the constraint arcs corresponding to these new constraints.

$$\begin{aligned} a &\in \{23, 4\} \\ b &\in \{3, 5, 6\} \\ c &\in \{1, 2, 3, 4\} \\ a + b &> 11 \\ b + c &> 8 \\ a - c &< 20 \\ c - b &< -1 \end{aligned}$$

4. Fill in the blanks to implement the stopping condition `constraint-search-strategy::have-enough-successes?::one`. Compare (visually, nothing to turn in) the solutions you get from this strategy vs the `constraint-search-strategy::have-enough-successes?::all` strategy to the constraint search problem from the examples. (You can do this by looking at the results of the public test cases.)

Warmed up and with a completed constraint propagator, Ben feels ready to tackle Ike's schedule. He opts for a flexible framework, in which his user can specify how many terms they want to graduate in (Ben thinks Ike might take awhile), how many classes they're willing to take per semester, and what requirements they must fulfill. His program uses the constraint propagator to assess

Problem 7: Course Scheduling

Fill in the blanks (in your solutions file) for the course scheduling constraint system, meeting the specifiations of and using the framework provided in `courses.scm`*. Pay particular attention to the comments at the top, which specify what a required course is, what a course requirements list is, what variables/domains should be used, etc. Also pay attention to the public test cases and the comments in* `courses.scm`*, which you may find helpful in deciding how to structure your code.*

1. **Prerequisites only:** In the first part of this problem you are expected to build a network much like you did in the previous simple-minded net. You will have to parse the list of courses into the network and add the constraints for the prerequisites only. This means that in the representation you are given, you will include the slot term in the domain, but will not add constraints involving it. You are supposed to provide code for *instantiate-prereq-constraints, instantiate-course-variable.* The functions *create-prereqs-constraint-net,* and *generate-prereqsonly-graduation-plan* are provided for you to test your solution up to here.

2. **Scheduling:** As you may have noticed from the test cases, the network produced in the previous item does not allow for proper scheduling of courses. In fact some of its solutions would imply a massive amount of credits per term for Ike to take. Here is where having introduced an additional element in our representation, namely, the slot field, we can benefit from a better choice of representation. The slot field assumes that there is a maximum number of courses a student can take per semester (due to the finite nature of semesters that does not make justice to the infinite capabilities of our students). Your task will be to complete *create-schedule-constraint-net* to make sure that no two courses are allocated in the same slot by adding suitable constraints to the network. A good way to do so it to seek some inspiration from *create-prereqs-constraint-net.*

Note that both parts will require you to implement predicates defining the semantics of the constraints you're adding (analogous to e.g. `diff<threshold` in the simple-minded net problem).

With Ike on a safe path to graduation, Ben's mind drifts (briefly!) back to games. He remembers a classic problem in artificial intelligence (and in chess), where the task is to place N queens on an N by N chessboard, such that none of the queens are threatened. He furthermore remembers that constraint satisfaction dramatically simplifies the search for such arrangements, since a queen in a certain position immediately rules out many other positions for the other queens. He decides to hack together a quick N-queens solver using his constraint package before the flight home.

Problem 8: N-Queens

The file `Nqueens.scm` *contains almost the entire code needed to solve the N-queens problem. The only missing element is the threat constraint between queens - that is, the procedure that returns false if two queens are on the same row, the same column, or the same diagonal. Your job is to supply this function and use the completed code to solve N-queens for a few dimensions. A few words of advice are below.*

1. If you represent the domain of each queen as a coordinate pair, you have an enormous amount of work to do. (Think about it.) If, on the other hand, you identify queens with their own columns (as, at bare minimum, each queen must be on its own column), you only need to check for row/diagonal collisions.

2. The variable names in `Nqueens.scm` are the column indices of each queen (from 0 to N-1). Their domains are the rows of each queen (from 0 to N-1).

3. The pairwise threat constraints are equipped with two parameters, named `col1` and `col2`. The first contains the column of the queen at one end of the constraint; the second contains the column of the queen at the other end of the constraint. Combined with the values of each queen variable, this gives you the coordiantes of each queen, enabling you to detect horizontal and diagonal threats.

Wrap-up

When you have completed this problem set, you should be able to `ssh` *into* `athena.dialup.mit.edu` *and type:*

```
cd ~/6.034-files/ps2/
add scheme
scheme
```

```
(load "tester.scm")
(test-file "ps2" "ps2-publictest")
```

This will evaluate your problem set based on the public test cases; you should pass them all! Good luck!