TIPS ON A2

Some issues have arisen in office hours that might be relevant to your assignment. Below is a quick distillation of advice and clarifications about assignment 2. If you're stuck or confused, this might just help.

1) What functions you actually need to write and what should they do?

Much of the code for solving search and csp problems is written for you in the code for the textbook (AIMA). You will need to add the following ***along with several helper functions to produce the appropriate functionality***:

Tracks problem:

1. Loose ends

```
(defun count-loose-ends (track) …)
```

3. Weekly connected components

```
(defun count-wccs (track) …)
```

4. Formulate as a search problem
Just like the problem set you completed, formulating a search problem requires an initial state, goal test, successor function, and cost function. Remember that for local search problems, h-cost need not underestimate the distance to the goal like in tree search. Here, the successor function is decomposed into an action function and a result function. Note that some search algorithms only care about random actions (like simulated annealing), so to use those, you don't even need an action function if you have a properly written random-action function.

```
(defstructure (track-local-problem (:include problem)) … )

(defmethod h-cost ((problem track-local-problem) state) …)

(defmethod goal-test ((problem track-local-problem) state) …)

EITHER:
(defmethod action ((problem track-local-problem) state) …)
OR
(defmethod random-action ((problem track-local-problem) state) …)

(defmethod result ((problem track-local-problem) action state) … )
```

Sudoku Problem:

9. Define sudoku as a CSP
You'll need to define the csp type (often specifying an alternate constructor so that you can set up the variables, constraints, and domains appropriately) and one or more constraint types, along with an allowed? function for each. See the n-queens example for help.

```
(defstruct (sudoku-csp (:include csp) (:constructor create-sudoku-csp))
…)

(defun make-sudoku-csp () …)

(defstruct (<a-constraint-type> (:include constraint))

(defmethod allowed? (vars values (constraint <a-constraint-type>))

10. Backtracking search with guessing penalty
```

```
Make a new backtracking search function that either returns two values
(the solution and the number of guesses) or a list containing the
solution and num guesses.

(defun backtracking-search-with-cost (csp
                      &optional (assignment (initial-csp-state csp))
                      &key (select-unassigned-variable #'next-variable)
                           (order-domain-values #'default-order))


11. Backtracking search with guessing penalty and extra inference
You need to write code that eliminates possible values for variables
before backtracking search decides what variable to select and
instantiate.  Ideally, you let the backtracking search code that is
already written take care of actually adding and deleting variables from
the current assignment.  Just focus on constraining the result of
(current-legal-values) by pruning the set of allowable values in each
variable’s domain.  This is what forward checking and arc consistency
do.  There are specialized inference functions for sudoku, like
scanning.  See the references  in the assignment for a description of
these techniques.

(defun backtracking-search-with-inference (csp
                      &optional (assignment (initial-csp-state csp))
                      &key (select-unassigned-variable #'next-variable)
                           (order-domain-values #'default-order))
```

2) How to build up a function incrementally

In writing LISP code, *DO NOT* write your entire program then start plugging in test cases, thereby initiating the debugging process. One of LISP’s weaknesses is it’s lack of large-scale debugging utilities.

However, one of the strengths of LISP is its functional approach to programming. You should write and test functions *INDEPENDENTLY*. Since functions return values, you can simply test your function on “corner cases” – hard examples that demonstrate that the program is functioning as it is supposed to.

Another advantageous aspect of LISP is its Bottom-Up/Top-Down programming style. While this class will not use all aspects of this, you should build small (easy) functions that do common actions (e.g. iota). You should test these functions thoroughly, and then use them to implement your program.

Summary - Rapid Prototyping

1. Write a specification for a function
2. Write the function
    a. Implement dependent functions with **stubs** to be done upon completion of this program.
3. ***Test the functions individually*** – do not proceed until each function works independently; debugging an entire project at once in LISP is a painstaking. *Printing out values of variables in loops and other crucial locations can reveal key aspects of your program.*

4. After building and testing your functions, integrate them by implementing stubs in the same manner. Continue until entire program is implemented and correct.

Example: N-Queens CSP – writing the make-nqueens-csp method

1. Write and test the iota method
2. Write a stub for make-nqueens-csp:
```
            (defun make-nqueens-csp (&key (n 8))
```
3. Check the variable creation (make sure it returns the values you want):
```
            (iota n 1)
```
4. Add :variables into your n-queens constructor:
```
            (defun make-nqueens-csp (&key (n 8))
              (create-nqueens-csp
               :n n :variables (iota n 1)))
```
5. Check the domain creation (make sure it returns the values you want):
```
            (mapcar #'(lambda (var) (cons var (iota n 1)))
                  (iota n 1))
```
6. Continue in this manner defining the entire method
```
(defun make-nqueens-csp (&key (n 8))
  (create-nqueens-csp
   :n n :variables (iota n 1)
   :domains (mapcar #'(lambda (var) (cons var (iota n 1)))
                (iota n 1))
  :constraints (mapcan #'(lambda (var)
                      (mapcar #'(lambda (var2)
                        (make-nqueens-constraint :variables (list
var var2)))
                      (iota (- n var) (1+ var))))
            (iota n 1))))
```
7. Check your function make-nqueens-csp for several “important” values of n.
8. Continue writing all your functions and their supporting methods in this manner.

3) Dealing with the AIMA code:

You’ll need to get the latest version of the class code in order to do this assignment. The code has changed since A0 – search and csp algorithms have been added and several bugs have been corrected. To do this, repeat the steps of the lisp tutorial that involve copying over the code-2e-188.ZIP archive, unpacking it, and editing aima.lisp to point to your directory.

You’ll also need some code related directly to this assignment. Specifically, you’ll need to download:
- tracks.lisp
- track-display.lisp
- sudoku-puzzles.lisp
- print-sudoku.lisp

All of these can be found through links from the A2 page. You’ll need to load them all, along with the AIMA code, in order to get your functions to work. If things don’t load, make sure you’re loading them in the right order: AIMA code first, then the files as they are listed above. Other orders might work, but some don’t.

Finally, make sure you compile your code with (aima-compile). This will make everything run much faster. After you type (load “aima.lisp”) but before you type (aima-load ‘search), type (aima-compile). You only need to do this once. If you ever get new versions of the class code, you’ll need to do it again. Compiling takes a minute, but it will save you many in the long run.

4) Helpful tips from a fellow student:

1. The description of the connections? function for the railroad stuff is misleading. connections? returns T if a piece CAN have connections at a given edge. To test whether it DOES, you need to see if the adjacent piece can also have connections at that edge.

2. To access a piece, use (aref name-of-track x-coord y-coord).

3. A track is a 2 dimensional array, so to get the width or the height, use

    (array-dimensions name-of-track) gives a 2 member list with width and height

    (array-dimension name-of-track 0) gives the width

    (array-dimension name-of-track 1) gives the height

4. iota makes a list of numbers from 1 to n.

5. It's very useful to know how to use loop (a lot of people who've only had cs61a have never used it, let alone loops within loops).

6. The most important thing to know is that this project takes FOREVER, so get started early, especially since you'll need to understand all the code and figure out how to debug when nothing will run.