# CS 440: CSPs and Games

## Part 1: Constraint SatisFaction Problems

### Word Puzzles

In this problem, we were tasked with finding solutions to the given word puzzles. In order to do so, we set up an infrastructure that would be helpful in both letter-based assignment and in the word-based assignment. Here is a quick list of the infrastructure decisions that we made to enable us to effectively solve this problem.

* Maintained a dictionary that maps a given category (string) to the list of possible words for that category (strings).
* Maintained a dictionary that maps a given category (string) to the indices of that category’s slots in the solution (integers).
  + For example, in Puzzle 1, the category ‘emotion’ would be mapped to the integers 4, 5, 7
* Maintained a dictionary that maps a category (string) to its category id (integer)
* For each index of the solution, we maintained a list of category ids (integers) that could place a letter at that location.
* Maintained the number of categories that were being used for the given puzzle
* Maintained a list that would ultimately hold the solution.
  + Elements of this list are initially assigned to hold the integer 0, and as letters or words get assigned, the elements of this list become characters. When this array contains a solution, it is formatted and printed.

As you may notice above, categories are sometimes referred to as a string or by their category id (integer). We chose to do this as strings are easy to read from the user end, but manipulating integers is much easier than manipulating strings. To ensure consistency between category id (integer) and the category (string) itself, we ensured that we maintained a dictionary that defined this mapping scheme.

**NOTE:** Search trace data for each example and each formulation is appended at the very end of this file.

### For the letter-based assignment:

* **Variables:** Indices of the solution array
* **Domains:** All letters for a given index (See section on inference detection)
* **Constraints:** Each category has 3 indices in the solution array. The letters of these three indices (taken in ascending order) MUST make up a word from that category. This must hold true for every category. (See section on constraint checking)
* **Inference Detection:** We implemented inference detection when generating the domain for an index. The naïve solution would merely use all 26 letters as the domain for each index of the array. In order to reduce the search trace, we chose our domain more carefully. Here is our methodology for generating the domain for a given index. We looked at each category that maps to the index. For each category, we determined which letter (1st, 2nd, or 3rd) of the word corresponded to the index. We then made a list of every letter that exists at the given position for that category. After generating these lists for every single category, we then took the intersection of the lists to generate the domain for the given index. This was done for each index of the solution array. This proved to be very effective. It limited the possible values for each index to only be letters that could exist in the solution rather than all letters. Our domains ended up being very small in comparison to having domains of all letters.
* **Constraint Checking:**
* **An Important Note About Our Search Iterations:** We originally did not implement strong constraint checking when doing our letter based implementation. We tested our code to determine how efficient it was before implementing constraint checking, and then we also tested it to see how efficient it was after implementing constraint checking. The differences are unbelievable. Here are some details on this data.

Our letter-based assignment found the following solutions.

**Puzzle 1 - Letter**

(Found result: NNEMANDYE)

(Found result: NNESAYDYE)

(Found result: NWEMANDYE)

(Found result: NWESAYDYE)

**Puzzle 2 - Letter**

(Found result: HSIAIWNCS)

(Found result: HSIAIWNPS)

(Found result: HSIOIWNDS)

(Found result: HSIOIWNYS)

**Puzzle 3 - Letter**

(Found result: ASULPEA)

(Found result: ASULPIE)

**Puzzle 4 - Letter**

(Found result: HEDITYRE)

(Found result: HELITYRE)

(Found result: HETITYRE)

**Puzzle 5 - Letter**

(Found result: IHTTNOIEN)

(Found result: IHTTYOIEN)

(Found result: THTTNOIEN)

(Found result: THTTYOIEN)

### For the word-based assignment:

* **Variables:** The categories
* **Domains:** All words for a given category (See section on inference detection)
* **Constraints**: Each category has 3 indices in the solution array. You must pick a word for each category, and assign the letters of that word to the the 3 indices (in ascending order) for the given category. You must ensure that each category has a word in the solution array at that category’s given indices. (See section on constraint checking)
* **Inference Detection**: We implemented inference detection when generating the domain for a given category. The naïve solution would set the domain for each category to be all possible words. As a result, the domain of a given category would contain words that were not even in that category. The domains of each category would be homogenous. We chose to implement inference detection for the domains. The domain of a given category would only be the words in that category. This drastically reduces the size of your domain for each category.
* **Constraint Checking:** We chose to implement constraint checking here to minimize the size of our search trace. Before expanding a potential word, we implemented our forward checking. Using one of our mapping schemes, we would look up the category that word belonged to and would find the indices that the given category could place values in. Then, we looked at each index of the solution array that a category could place a letter in. For this problem, we would always look at three indices since a category could only place letters at three indices. If a letter had already been assigned to that index and the current word would change the letter at that index, our constraint checking would tell the backtracking search to not expand that word. If a letter had not been assigned to that index yet or if assigning the given word would not change the letter at that index, then the index was consistent. If all three indices were consistent, then the constraint checking would tell the backtracking search to continue; the backtracking search would then assign the word and continue on the search path.

Our word-based assignment found the following solutions:

**Puzzle 1 - Word**

(Found result: NNEMANDYE)

(Found result: NWEMANDYE)

(Found result: NNESAYDYE)

(Found result: NWESAYDYE)

**Puzzle 2 - Word**

(Found result: HSIAIWNCS)

(Found result: HSIAIWNPS)

(Found result: HSIOIWNDS)

(Found result: HSIOIWNYS)

**Puzzle 3 - Word**

(Found result: ASULPEA)

(Found result: ASULPIE)

**Puzzle 4 - Word**

(Found result: HEDITYRE)

(Found result: HELITYRE)

(Found result: HETITYRE)

**Puzzle 5 - Word**

(Found result: IHTTNOIEN)

(Found result: THTTNOIEN)

(Found result: IHTTYOIEN)

(Found result: THTTYOIEN)

An interesting design decision we made with this portion of the problem is that we did not tie the code down to each category being mapped to three letters and only three letters. Given the way we designed our solution, categories need not be symmetric in how many letters they assign. We could easily support words with more than three letters, given that we calculate lengths, rather than assigning them as three up front. Though this proves to be more work for the programmer, it increases the versatility of our code. Our code could even be run with a single puzzle where each category assigned a different number of letters. For example, the puzzle could have one category that assigned four letters, another category that assigned six, and a third category that only assigned two letters. Our code would still work on puzzles like that. This is for bonus points. ☺

### 1.2 Map Coloring (For Bonus Points)

Though we are three unit students, we chose to implement the Map Coloring problem for bonus points.

First, we implemented this without forward checking and did not have a heuristic that determines which variable to assign, and which value to assign to that variable. With this method, the search space will become increasingly large as the number of nodes in the graph increases. This implementation did, however, generate a solution to the problem. The solution is listed below.

After implementing the most basic backtracking search, we chose to implement forward checking. When assigning a color to a node, we would check if any of the adjacent nodes had already been assigned the same color. If that was the case, we would not assign that color.

Here are the graphs that we tested, and the solutions that accompanied them.

**Graph 1 – The Australia Problem**

Node 0 (WA) is red

Node 1 (NT) is yellow

Node 2 (SA) is green

Node 3 (Q) is red

Node 4 (NSW) is yellow

Node 5 (V) is red

Node 6 (T) is red

T -> {}

WA -> {NT, SA}

NT -> {WA, Q, SA}

SA -> {WA, NT, Q, NSW, V}

Q -> {NT, SA, NSW}

NSW -> {Q, SA, V}

V -> {SA, NSW}

**Graph 2**

Node 0 is red

Node 1 is yellow

Node 2 is yellow

Node 3 is red

Node 4 is green

Node 5 is green

Node 6 is blue

0 -> { 1, 4, 2, 5 }

1 -> { 0, 4, 6, 5 }

2 -> { 0, 4, 3, 6, 5 }

3 -> { 2, 4, 6 }

4 -> { 0, 1, 6, 3, 2 }

5 -> { 2, 6, 1, 0 }

6 -> { 2, 3, 4, 1, 5 }

**Graph 3**

Node 0 is red

Node 1 is yellow

Node 2 is red

Node 3 is red

Node 4 is yellow

Node 5 is yellow

Node 6 is red

Node 7 is red

Node 8 is yellow

Node 9 is yellow

Node 10 is red

Node 11 is green

0 -> {1, 4, 5}

1 -> {0, 3}

2 -> {}

3 -> {1}

4 -> {0}

5 -> {0,6,11}

6 -> {5}

7 -> {8,9,11}

8 -> {7}

9 -> {7,10}

10 -> {9}

11 -> {5,7}

## Part 2: War Game

### 2.1 Minimax and Alpha-Beta Agents