1.0 Setting Up The Environment

The backbone of the puzzle solver algorithms is the `WordPuzzle` class:

```
public class WordPuzzle{
      public int puzzleSize;
      public Character[] puzzle;
      public HashMap<String, Integer[]> smallWords;

      // Methods omitted
}
```

A `WordPuzzle` is constructed by reading in a text file line by line. The first line of “puzzle{X}.txt” gives us puzzleSize. The remaining lines provide us with a `String` that will be used as our key for `smallWords HashMap`, and `Integer` indices that will be saved into an `Integer[]`.

The `WordPuzzle` works in conjunction with the simple `WordList` class:

```
public class WordList {
      public HashMap<String, ArrayList<String>> wordList;

      private void addWords(String filename){ /* Details omitted */ }

      // Other methods omitted
}
```

A `WordList` is constructed by reading all files that were saved in a wordlist directory. Each file’s `filename` is a `String` that is used as a key in our `wordList HashMap`. The actual words in the file our saved as values in our `wordList HashMap` as an `ArrayList<String>`.

1.1 Word Puzzles

Letter-Based Assignment

```
public class LetterBasedSolver{
      Integer [] mcv; //mcv = Most Constrained Value.

      public ArrayList<Character[]> findSolution(WordPuzzle wordPuzzle,
            WordList wordList){ /* Details omitted */ }

      public int selectUnassignedVariable(Character[] assignment){
            /* Details omitted */
      }

      public boolean assignmentValid(WordPuzzle wordPuzzle, WordList
            wordList, Character[] assignment){ /* Details omitted */ }

      // Other methods omitted
}
```

findSolution is a wrapper class that is an implementation of the “Recursive-Backtracking” algorithm that is simply a Depth-First-Search (DFS) for Constraint Satisfaction Problems (CSP) with single-variable assignments. findSolution sets up the parameters to call a recursive version of findSolution to find all solutions to the puzzle.

In each level of our tree, we use selectUnassignedVariable to make sure we do not branch on variables. We only branch on values. selectUnassignedVariable uses “Most Constrained Value” (details explained later) to select an index in the puzzle to assign a letter to.

assignmentValid is used to make sure we have not violated any constraints as we add letters to our solution. We backtrack if any constraints have been violated.

(1) Variables, Domains, Constraints

a. Variables

- The variables are puzzleSize number of indices which is given in the 1st line of each “puzzle{X}.txt”

b. Domains

- The domains for each of puzzleSize variables are the 26 capital letters. Specifically: ‘A’ to ‘Z’.

c. Constraints

- The constraints are provided by wordList. A WordPuzzle tells us which constraints in wordList apply to a given problem (given in each “puzzle{X}.txt”). Each puzzle has 5-7 constraints. Each constraint is formed by 3 indices to puzzle that creates a three-letter word belonging to a specific category, such as “clothing”. The word created by these 3 indices must be present in a list of candidate words presented in wordList.

d. Checking/Inference to make search more efficient

- **Forward Checking**: The recursive search algorithm was coded in such a way such that we terminate a path (and backtrack) when a variable has no legal values.
- **Most Constrained Variable**: To speed up our search, MCV is used to select which indices to fill with letters first. The index in the puzzle that is part of the most word categories is the MCV. This greatly improved our search speed.

(2) All Possible Solutions

***** solve by LETTER *****

puzzle1.txt

```
NNEMANDYE
NWEMANDYE
NNESAYDYE
NWESAYDYE
```

puzzle2.txt

```
HSIAIWNCS
HSIAIWNPS
HSIOIWNDS
HSIOIWNYS
```

puzzle3.txt

```
ASULPEA
ASULPIE
```

puzzle4.txt

```
HEDITYRE
HELITYRE
HETITYRE
```

puzzle5.txt

```
IHTTNOIEN
IHTTYOIEN
THTTNOIEN
THTTYOIEN
```

(3) Letter-Based Traces

Search order: For all 5 searches, selecting the Most-Constrained Variable (MCV) which corresponds to the most word-categories an index is part of. (Index counting starts at 0)

puzzle1.txt

```
Search Order (MCV): 8 -> 4 -> 2 -> 3 -> 6 -> 7 -> 0 -> 1 -> 5

root -> E -> A -> E -> M -> D -> Y -> N -> N -> N (found result: NNEMANDYE)
                                        -> W -> N (found result: NWEMANDYE)
                       S -> D -> Y -> N -> N -> Y (found result: NNESAYDYE)
                                        -> W -> Y (found result: NWESAYDYE)
```

puzzle2.txt

```
Search Order (MCV): 1 -> 3 -> 0 -> 4 -> 5 -> 8 -> 2 -> 6 -> 7

root -> S -> A -> H -> I -> W -> S -> I -> N -> C (found result: HSIAIWNCS)
                                             -> P (found result: HSIAIWNPS)
             O -> H -> I -> W -> S -> I -> N -> D (found result: HSIOIWNDS)
                                             -> Y (found result: HSIOIWNYS)
```

puzzle3.txt

```
Search Order (MCV): 4 -> 0 -> 1 -> 3 -> 5 -> 6 -> 2

root -> P -> A -> S -> L -> E -> A -> U (found result: ASULPEA)
                         -> I -> E -> U (found result: ASULPIE)
```

puzzle4.txt

```
Search Order (MCV): 0 -> 1 -> 3 -> 7 -> 5 -> 6 -> 2 -> 4

root -> H -> E -> I -> E -> Y -> R -> D -> T (found result: HEDITYRE)
                                   -> L -> T (found result: HELITYRE)
                                   -> T -> T (found result: HETITYRE)
```

puzzle5.txt

```
Search Order (MCV): 8 -> 7 -> 5 -> 2 -> 3 -> 6 -> 0 -> 1 -> 4

root -> N -> E -> O -> T -> T -> I -> I -> H -> N (found result: IHTTNOIEN)
                                             -> Y (found result: IHTTYOIEN)
                                   -> T -> H -> N (found result: THTTNOIEN)
                                             -> Y (found result: THTTYOIEN)
```

Word-Based Assignment

```
public class WordBasedSolver{
      Integer [] mcv; //mcv = Most Constrained Value.

      public ArrayList<Character[]> findSolution(WordPuzzle wordPuzzle,
            WordList wordList){ /* Details omitted */ }

      public int selectUnassignedWord(WordPuzzle wordPuzzle, Character[] assignment){
            /* Details omitted */
      }

      public boolean assignmentValid(WordPuzzle wordPuzzle, WordList
            wordList, Character[] assignment){ /* Details omitted */ }

      // Other methods omitted
}
```

findSolution is a wrapper class that is an implementation of the “Recursive-Backtracking” algorithm that is simply a Depth-First-Search (DFS) for Constraint Satisfaction Problems (CSP) with single-variable assignments. findSolution sets up the parameters to call a recursive version of findSolution to find all solutions to the puzzle.

In each level of our tree, we use selectUnassignedWord to make sure we do not branch on variables. We only branch on values. selectUnassignedWord simply selects the next unused word to assign to the puzzle.

assignmentValid is used to make sure we have not violated any constraints as we add words to our solution. We backtrack if any constraints have been violated.

(1) **Variables, Domains, Constraints**

a. Variables

- The variables are puzzleSize number of indices which is given in the 1st line of each “puzzle{X}.txt”

b. Domains

- The domains for each of puzzleSize variables are the 26 capital letters. Specifically: ‘A’ to ‘Z’.

c. Constraints

- The constraints are provided by wordList. A WordPuzzle tells us which constraints in wordList apply to a given problem (given in each “puzzle{X}.txt”). Each puzzle has 5-7 constraints. Each constraint is formed by 3 indices to puzzle that creates a three-letter word belonging to a specific category, such as “clothing”. The word created by these 3 indices must be present in a list of candidate words presented in wordList.

d. Checking/Inference to make search more efficient

- **Forward Checking**: The recursive search algorithm was coded in such a way such that we terminate a path, and backtrack, when a 3-letter word has no legal assignments.

(2) All Possible Solutions

*** solve by WORD ***

puzzle1.txt

```
NNEMANDYE
NNESAYDYE
NWEMANDYE
NWESAYDYE
```

puzzle2.txt

```
HSIAIWNCS
HSIAIWNPS
HSIOIWNDS
HSIOIWNYS
```

puzzle3.txt

```
ASULPEA
ASULPIE
```

puzzle4.txt

```
HEDITYRE
HELITYRE
HETITYRE
```

puzzle5.txt

```
IHTTNOIEN
IHTTYOIEN
THTTNOIEN
THTTYOIEN
```

(3) Letter-Based Traces

puzzle1.txt

```
Search Order: adjective -> emotion -> interjection -> verb -> body -> adverb

root -> NEE -> MAD -> MAN -> DYE -> NAE -> (found result: NNEMANDYE)
            -> SAD -> SAY -> DYE -> NAE -> (found result: NNESAYDYE)
        WEE -> MAD -> MAN -> DYE -> NAE -> (found result: NWEMANDYE)
            -> SAD -> SAY -> DYE -> NAE -> (found result: NWESAYDYE)
```

puzzle2.txt

```
Search Order: palindrome -> pronoun -> interjection -> verb -> noun -> math

root -> SIS -> HIS -> HAW -> SAC -> SIN -> (found result: HSIAIWNCS)
                          -> SAP -> SIN -> (found result: HSIAIWNPS)
                      HOW -> SOD -> SIN -> (found result: HSIOIWNDS)
                          -> SOY -> SIN -> (found result: HSIOIWNYS)
```

puzzle3.txt

```
Search Order: nature -> interjection -> animal -> noun -> food

root -> ALP -> SUP -> LEA -> (found result: ASULPEA)
                   -> LIE -> (found result: ASULPIE)
```

puzzle4.txt

```
Search Order: computer -> pronoun -> interjection -> verb -> noun -> body

root -> HER -> HIT -> HEY -> DIE -> (found result: HEDITYRE)
               HEY -> HIT -> LIE -> (found result: HELITYRE)
                          -> TIE -> (found result: HETITYRE)
```

puzzle5.txt

```
Search Order: container -> number -> music -> animal -> noun -> body -> adverb

root -> TIN -> TEN -> HEN -> ION -> NON -> (found result: IHTTNOIEN)
                                 -> YON -> (found result: IHTTYOIEN)
        HEN -> TEN -> TIN -> TON -> NON -> (found result: THTTNOIEN)
                                 -> YON -> (found result: THTTYOIEN)
```