# Constraint Satisfaction:

## Solving Word Puzzles using Backtracking Search

David Young

October 2015

# Contents

**1 Introduction** **2**
1.1 Project Overview . . . . . . . . . . 2
1.2 Problem Definition . . . . . . . . . . 2

**2 Background** **4**
2.1 Constraint Satisfaction Problems . . . . . . . . . . 4
2.2 Backtracking Search . . . . . . . . . . 4

**3 Overview of Source** **5**

**4 Backtracking Search: Letter-based Assignment** **6**
4.1 Defining the CSP . . . . . . . . . . 6
4.2 Solution Implementation . . . . . . . . . . 6
4.3 Puzzle Solution & Search Trace . . . . . . . . . . 8
4.4 Analysis & Discussion . . . . . . . . . . 10

**5 Backtracking Search: Word-based Assignment** **11**
5.1 Defining the CSP . . . . . . . . . . 11
5.2 Solution Implementation . . . . . . . . . . 11
5.3 Puzzle Solution & Search Trace . . . . . . . . . . 13
5.4 Analysis & Discussion . . . . . . . . . . 15

# 1 Introduction

## 1.1 Project Overview

This writeup summarizes the procedure and results of two particular solutions to a specific Constraint Satisfaction Problem (CSP). It also contains discussion of said results and attempts to provide some insight and reflection on the behavior of implemented algorithms. This report was produced for course "CS-440: Artificial Intelligence" at University of Illinois Urbana Champaign.

## 1.2 Problem Definition

I was tasked to solve word puzzles using backtracking search (one of many algorithms for constraint satisfaction). In these puzzles, an array of letter had to be filled such that certain subsets of the letters formed words from a given category. An example of a word puzzle with one possible solution is given below. On the left is the problem definition (represented graphically), and on the right is one possible solution.

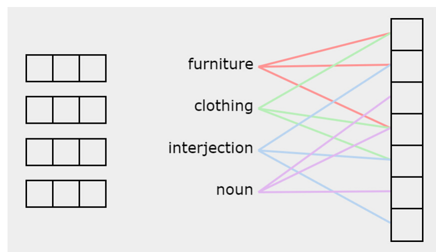


Figure 1: Example Problem

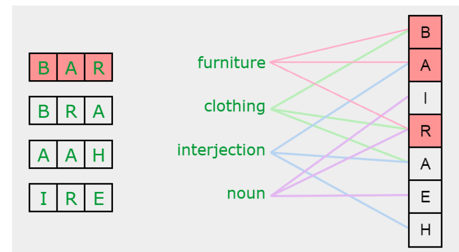


Figure 2: Example Solution

In this example, the letters connected to red lines should form a word from the "furniture" category and similarly, the letters connected to the green lines should form a word belonging to the "clothing" category. All the words have a length of 3 and the list of candidate words for each category is given in a word list file.

The goal was to implement backtracking search to find all solutions (each puzzle might have multiple solutions) to a few puzzles defined in text files. An example text file defining a problem is shown below.

```
9
emotion: 4, 5, 7
body: 3, 8, 9
adverb: 1, 5, 9
adjective: 2, 3, 9
interjection: 4, 5, 6
verb: 7, 8, 9
```

Figure 3: Example Puzzle File. First line is the number of letter/indices in the solution. Each subsequent line represents a category and the indices in the solution associated with that category.

In each input puzzle file, the first line specifies the size of the result array, and the rest of the file lists the category names and indices of the result array that correspond to the word of that category. This effectively describes the colored line connections shown in Figure 1.

Two versions of backtracking search are implemented here, one for each of two different formulations:

1. Letter-based assignment where the array is filled in one letter at a time.
2. Word-based assignment where the array is filled in one word at a time.

# 2 Background

## 2.1 Constraint Satisfaction Problems

In computer science, Constraint Satisfaction Problems (CSP) are problems defined as a set of objects whose state must satisfy a set of constraints. CSPs represent the entities in a problem as a collection of finite constraints over variables, which can be solved using constraint satisfaction methods.

CSPs use a factored representation for each state: a set of variables, each of which has a value. The problem is solved when each variable has a value that satisfies all the constraints on that variable. The big idea behind CSP algorithms is to eliminate large portions of the search tree by identifying variable & value combinations that violate constraints.

When setting up problems as CSPs, it can be useful to use a standard notation. Most commonly this involves defining variables, domains and constraints. These are commonly written as:

- X = set of variables
- D = set of domains (one for each variable) where each domain is a set of allowable values for a variable
- C = set of constraints specifying allowable combinations of variables

## 2.2 Backtracking Search

Backtracking search is one of many algorithms for constraint satisfaction that can be used to solve CSPs. At a high level, backtracking search is basically just depth-first search. In action, backtracking search chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, the recursive algorithm returns failure, causing the previous call to try another value. There are many possible modifications to improve performance including pre-processing done prior to the search or improvements intertwined with the search.

The basic pseudo-code for backtracking search is as follows.

```
function RECURSIVE-BACKTRACKING(assignment, csp)
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp)
     if value is consistent with assignment given CONSTRAINTS[csp]
        add {var = value} to assignment
        result ← RECURSIVE-BACKTRACKING(assignment, csp)
        if result ≠ failure then return result
        remove {var = value} from assignment
  return failure
```

Figure 4: Pseudo code for a recursive backtracking search.

# 3 Overview of Source

**Obtaining the source code**
The entirety of the code written for this project can be found at the following repository:

https://github.com/dcyoung/WordPuzzleConstraintSatisfaction

**Summary of source code**
The following source files were written from scratch. All code is well commented with Javadocs; it should be no burden to browse for specific details.

| Filename | Description |
|---|---|
| TestRunner.java | Provides examples running the backtracking algorithm on various puzzles. |
| WordDatabase.java | Holds all of the words as hashmap with categories as keys. |
| Puzzle.java | Holds the data for a puzzle in the form of 2 hash maps. |
| WordFileReader.java | Reads properly formatted .txt files into a word database or puzzle structure. |
| LetterBasedCSP.java | Solves a constraint satisfaction problem described by a puzzle object. |
| WordBasedCSP.java | Solves a constraint satisfaction problem described by a puzzle object. |

A quick summary of the code: The structure and high level pseudo code for both versions of the backtracking algorithm are nearly identical (see figure 4). The implemented backtracking algorithm can be found in the method *"recursiveBacktracking()"* in both *WordBasedCSP.java* and *LetterBasedCSP.java*). The main difference between the two implementations is how the helper functions select unassigned variables, order domain values and check if constraints are violated. The puzzle definition however, remains the same regardless of the type of constraint solution (letter or word based assignment) used to solve the puzzle. A puzzle object was created to hold all the information defining an unsolved puzzle. The chosen data structures revolve around 2 hashmaps; the first is used to map between categories for words and their respective puzzle indices, while the second is used to map between the puzzle indices of the solution and their respective categories. In this way we have a mapping of categories – indices and indices – categories.

# 4 Backtracking Search: Letter-based Assignment

## 4.1 Defining the CSP

**Variables, Domains and Constraints**
For the letter based assignment, X,D & C were defined as:

- Variables X = index (of the solution array)
- Domain D = permit-able letters for a given variable
- Constraints C = a letter must be able to form a word from a connected category given previous specified letters

## 4.2 Solution Implementation

**Backtracking algorithm**
This is the main constraint satisfaction solver. It effectively conducts depth first search on the state space of possible assignments for a CSP, halting any dives that violate constraints. The recursive-backtracking pseudo code returned a result, but here it should look for all possible solutions. Therefore this implementation returns nothing, while any found solutions are stored in the instance variable "results".

```

public void RecursiveBacktracking(ArrayList<Character> assignment){
  int index = SelectUnassignedVariable(assignment);
  for(char c : this.OrderDomainValues(index)){
    // Add it to the assignment
    assignment.set(index, c);
    //if value is consistent with assignment given constraints
    if(CheckIfConsistent(index, assignment)){
      if (!assignment.contains(null)) {
        // add it to solution set
        this.results.add(DeepCopyCharArrayList(assignment));
      } else {
        //dive deeper into the tree (the passed in assignment here contains the
    char c)
        RecursiveBacktracking(assignment);
      }
    }
    // Remove from assignment, keeping the tree at the current depth
    assignment.set(index, null);
    // removing the character ensures the next loop iteration is searching breadth
  }
}
```

:

**Selecting an unassigned variable**
A helper function for the main backtracking algorithm, this method selects an unassigned variable from the assignment. A more efficient implementation would likely consider the most constrained variable first, in order to fail fast and prune large portions of the search tree. But, currently, the naive implementation simply looks for the next unassigned index.

```
private int SelectUnassignedVariable(ArrayList<Character> assignment){
  //naive
  for(int i = 0; i <assignment.size(); i++){
    if (assignment.get(i) == null){
      return i;
    }
  }
  return -1;
}
```

:

**Ordering the values in the variable's domain**
A helper method for the main backtracking algorithm, this method returns a domain of possible values (permit-able letters) for the variable index. For efficiency, it should intelligently order the domain of possible values for the variable index, but currently the naive method simply returns the normal alphabet of uppercase characters. This is not efficient, but will work because the alphabet is finite and the performance impact is minimal considering the large portion of the alphabet typically included in associated categories.

```
private ArrayList<Character> OrderDomainValues(int index){
  //naive: return a-z since there will be lots of chars anyways
  return this.alphabet;
}
```

:

**Consistency Check**
The following consistency check ensures that constraints have not been violated. As outlined in the CSP definition for letter based assignments, this involves checking that the char c proposed for the specified index is able to form a word from every connected category given previous specified letters.

```
private boolean CheckIfConsistent(int index, ArrayList<Character> assignment){
  //for each category linked to the index
  for(String category: this.puzzle.getIndexCategoryMap().get(index)){
    String partialWord = GetWordRegEx(category, assignment);
    boolean WordExistsInCategory = false;
    //for each word in that category
    for(String word : db.getWordMap().get(category)){
      //could the partial word construct word
      if(Pattern.matches(partialWord,word)){
        WordExistsInCategory = true;
        break;
      }
    }
    if(!WordExistsInCategory)
      return false;
  }
  return true;
}
```

:

## 4.3 Puzzle Solution & Search Trace

### Puzzle #1:

```
9
emotion: 4, 5, 7
body: 3, 8, 9
adverb: 1, 5, 9
adjective: 2, 3, 9
interjection: 4, 5, 6
verb: 7, 8, 9
```

### Puzzle #1 Solution:

```
(Soln #0: NNEMANDYE)  adjective: NEE  emotion: MAD  interjection: MAN  verb: DYE  body: EYE  adverb: NAE
(Soln #1: NNESAYDYE)  adjective: NEE  emotion: SAD  interjection: SAY  verb: DYE  body: EYE  adverb: NAE
(Soln #2: NWEMANDYE)  adjective: WEE  emotion: MAD  interjection: MAN  verb: DYE  body: EYE  adverb: NAE
(Soln #3: NWESAYDYE)  adjective: WEE  emotion: SAD  interjection: SAY  verb: DYE  body: EYE  adverb: NAE
```

### Puzzle #1 Search Trace

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

### Puzzle #2:

```
9
pronoun: 1, 3, 9
palindrome: 2, 5, 9
math: 2, 5, 7
interjection: 1, 4, 6
verb:2, 4, 6
noun: 2, 4, 8
```

### Puzzle #2 Solution:

```
(Soln #0: HSIAIWNCS) palindrome: SIS  pronoun: HIS  interjection: HAW  verb: SAW  noun: SAC  math: SIN
(Soln #1: HSIAIWNPS) palindrome: SIS  pronoun: HIS  interjection: HAW  verb: SAW  noun: SAP  math: SIN
(Soln #2: HSIOIWNDS) palindrome: SIS  pronoun: HIS  interjection: HOW  verb: SOW  noun: SOD  math: SIN
(Soln #3: HSIOIWNYS) palindrome: SIS  pronoun: HIS  interjection: HOW  verb: SOW  noun: SOY  math: SIN
```

### Puzzle #2 Search Trace

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

### Puzzle #3:

```
7
nature: 1, 4, 5
food: 5, 6, 7
animal: 1, 2, 5
interjection: 2, 3, 5
noun: 4, 6, 7
```

### Puzzle #3 Solution:

```
(Soln #0: ASULPEA)  nature: ALP  interjection: SUP  animal: ASP  noun: LEA  food: PEA
(Soln #1: ASULPIE)  nature: ALP  interjection: SUP  animal: ASP  noun: LIE  food: PIE
```

### Puzzle #3 Search Trace

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

### Puzzle #4:

```
8
body: 2, 6, 8
pronoun: 1, 2, 7
computer: 1, 4, 5
interjection: 1, 2, 6
verb: 3, 4, 8
noun: 4, 7, 8
```

### Puzzle #4 Solution:

```
(Soln #0: HEDITYRE)  computer: HIT  pronoun: HER  interjection: HEY  verb: DIE  noun: IRE  body: EYE
(Soln #1: HELITYRE)  computer: HIT  pronoun: HER  interjection: HEY  verb: LIE  noun: IRE  body: EYE
(Soln #2: HETITYRE)  computer: HIT  pronoun: HER  interjection: HEY  verb: TIE  noun: IRE  body: EYE
```

### Puzzle #4 Search Trace

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

### Puzzle #5:

```
9
number: 3, 8, 9
container: 4, 7, 9
music: 3, 7, 8
body: 4, 6, 8
adverb: 5, 6, 9
animal: 2, 8, 9
noun: 1, 6, 9
```

### Puzzle #5 Solution:

```
(Soln #0: IHTTNOIEN) container: TIN  number: TEN  music: TIE  animal: HEN  noun: ION  body: TOE  adverb: NON
(Soln #1: IHTTYOIEN) container: TIN  number: TEN  music: TIE  animal: HEN  noun: ION  body: TOE  adverb: YON
(Soln #2: THTTNOIEN) container: TIN  number: TEN  music: TIE  animal: HEN  noun: TON  body: TOE  adverb: NON
(Soln #3: THTTYOIEN) container: TIN  number: TEN  music: TIE  animal: HEN  noun: TON  body: TOE  adverb: YON
```

### Puzzle #5 Search Trace

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

## 4.4 Analysis & Discussion

**Potential Improvements**

The implemented algorithm closely follows the pseudo code provided in the background section, differing only by storing all solutions rather than returning just one. As for the helper functions, most were left with naive implementations as the time required to solve the given puzzles was only a few seconds at most. The naive methods are not as efficient in how they order variables or values, but they return complete and optimal results. That is to say all valid solutions were found and the algorithm never returned invalid solutions.

To improve the algorithm further, the ordering of unassigned variables could be altered to prioritize the most constrained variables. With this change, the backtracking algorithm would try options likely to conflict first, and in doing so eliminate large portions of the search tree early on. If the goal was to find any solution as rapidly as possible, the ordering of the values in the variable's domain would be altered to prioritize the least constrained value such that more likely components of the solution are attempted first. In this situation however, the goal was to find all solutions. Therefore the ordering of values was irrelevant. Lastly, the current implementation does not include any early checking of failure, such as an arc-consistency check.

Even without these enhancements, for small puzzles this backtracking algorithm is reasonably fast.

# 5 Backtracking Search: Word-based Assignment

## 5.1 Defining the CSP

### Variables, Domains and Constraints

For the word based assignment, X,D & C were defined as:

- Variables X = a *wordVar*
- Domain D = all potential *wordVals* from the linked category
- Constraints C = for every *wordVar*, the *wordVal* assigned to that *wordVar* must exist in the *wordVal*'s category

Here a *wordVar* is the three indices linked to a category (as defined by the puzzle definition) and a *wordVal* is the assigned letters at the indices linked to a category.

## 5.2 Solution Implementation

### Backtracking algorithm

This is the main constraint solver and again it differs slightly from the pseudo code. Instead of returning the first result encountered, the algorithm adds every encountered solution to a running list of results stored in an instance variable.

```
public void RecursiveBacktracking(ArrayList<Character> assignment, int depth){
  //wordVar here logically refers to 3 indices, but will be a category
  String category = SelectUnassignedVariable(assignment);

  for(String wordVal : this.OrderDomainValues(category)){
    //Remember the current assignment for later
    ArrayList<Character> old_assignment = DeepCopyCharArrayList(assignment);
    // Add it to the assignment
    AddToAssignment(assignment, category, wordVal);

    //if wordVal is consistent with assignment given constraints
    if(CheckIfConsistent(category, assignment)){
      if (!assignment.contains(null)) {
        //add it to solution set
        if(!DuplicateResultCheck(assignment)){
          this.results.add(DeepCopyCharArrayList(assignment));
          for (int i = 0; i < depth; i++) {
            System.out.print("        ");
          }
        }
      } else {
        //dive deeper into the tree (the passed in assignment here contains the wordVal)
        RecursiveBacktracking(assignment, depth+1);
      }
    }
    // Remove from assignment, keeping the tree at the current depth
    assignment = old_assignment;
    // removing the word ensures the next loop iteration is searching breadth
  }
}
```

:

**Selecting an unassigned variable**
A helper method for the main backtracking algorithm, this returns a category with any unsigned characters.

```
private String SelectUnassignedVariable(ArrayList<Character> assignment){
    //should return a category
    int index = 0;
    for(int i = 0; i <assignment.size(); i++){
      if (assignment.get(i) == null){
        index = i;
        break;
      }
    }
    return this.puzzle.getIndexCategoryMap().get(index).get(0);
  }
```

:

**Ordering the values in the variable's domain**
A helper method for the main backtracking algorithm, this returns a domain of possible values for a variable. For the word-based assignment, this is relatively simple as the word database intrinsically provides this information already.

```
private ArrayList<String> OrderDomainValues(String category){
    return this.db.getWordMap().get(category);
  }
```

:

**Consistency Check**
A helper method for the main backtracking algorithm this checks that a constraint has not been violated after an assignment. To follow the CSP designed for the word-based assignment, this means checking that the *wordVal* proposed for the specified category does not violate any constraints at indices shared with other categories.

```
private boolean CheckIfConsistent(String augmentedCategory, ArrayList<Character>
    assignment){
    //An efficient solution would check only categories linked to any index of the
    augmented category

    //naive, check every category
    for(String category: this.puzzle.getCategoryIndexMap().keySet()){
      String partialWord = GetWordRegEx(category, assignment);
      boolean WordExistsInCategory = false;
      //for each word in that category
      for(String word : db.getWordMap().get(category)){
        //could the partial word construct word
        if(Pattern.matches(partialWord,word)){
          WordExistsInCategory = true;
          break;
        }
      }
      if(!WordExistsInCategory)
        return false;
    }
    return true;
  }
```

:

## 5.3 Puzzle Solution & Search Trace

The puzzle definitions and their solutions are the same for both versions of the backtracking algorithm (letter and word based assignments). Therefore, only the traces will be included here for brevity. One can view the definitions and solutions in the previous section.

### Puzzle #1 Search Trace

```
Search order: adverb -> adjective -> interjection -> verb
root->  NAE ->  NEE ->  MAN -> DYE(found result: NNEMANDYE)
                            -> backtrack
                    ->  SAY -> DYE(found result: NNESAYDYE)
                            -> backtrack
                    ->  backtrack
            ->  WEE ->  MAN -> DYE(found result: NWEMANDYE)
                            -> backtrack
                    ->  SAY ->  DYE(found result: NWESAYDYE)
                            -> backtrack
                    ->  backtrack
            -> backtrack
    ->  NAW ->  NEE ->  MAN ->  DYE -> backtrack
                    ->  SAY ->  DYE -> backtrack
                    ->  backtrack
            ->  WEE ->  MAN ->  DYE -> backtrack
                    ->  SAY ->  DYE -> backtrack
                    ->  backtrack
            -> backtrack
    ->  backtrack
```

### Puzzle #2 Search Trace

```
Search order: interjection -> math -> pronoun -> noun
root -> AYE -> DAG -> backtrack
            -> backtrack
     -> FIE -> DAG -> backtrack
            -> backtrack
     -> HAW -> SIN -> HIS -> SAC(found result: HSIAIWNCS)
                          -> SAP(found result: HSIAIWNPS)
                          -> SOD(found result: HSIOIWNDS)
                          -> SOY(found result: HSIOIWNYS)
                          -> backtrack
                   -> backtrack
            -> backtrack
     -> HOW -> SIN -> HIS -> SAC -> SAP -> SOD -> SOY -> backtrack
                   -> backtrack
            -> backtrack
     -> HUM -> backtrack
     -> SAY -> SIN -> backtrack
            -> backtrack
     -> WOW -> SIN -> HIS -> SAC -> SAP -> SOD -> SOY -> backtrack
                   -> backtrack
            -> backtrack
     -> YAY -> SIN -> backtrack
            -> backtrack
     -> YOW -> SIN -> HIS -> SAC -> SAP -> SOD -> SOY -> backtrack
                   -> backtrack
            -> backtrack
     -> YUM -> backtrack
     -> backtrack
```

### Puzzle #3 Search Trace

```
Search order: nature -> interjection -> noun
root -> ALP -> SUP -> LEA(found result: ASULPEA)
                   -> LIE(found result: ASULPIE)
                   -> backtrack
            -> backtrack
     -> BOT -> backtrack
     -> ZHO -> backtrack
     -> backtrack
```

### Puzzle #4 Search Trace

```
Search order: computer -> body -> verb -> noun
root -> FAX -> backtrack
     -> HIT -> EYE -> DIE -> IRE(found result: HEDITYRE)
                          -> backtrack
                   -> LIE -> IRE(found result: HELITYRE)
                          -> backtrack
                   -> TIE -> IRE(found result: HETITYRE)
                          -> backtrack
                   -> backtrack
            -> backtrack
     -> WEB -> backtrack
     -> WWW -> backtrack
     -> backtrack
```

### Puzzle #5 Search Trace

```
Search order: noun -> animal -> music -> body -> adverb
root -> ION -> HEN -> TIE -> TOE -> NON(found result: IHTTNOIEN)
                                 -> YON(found result: IHTTYOIEN)
                                 -> backtrack
                          -> backtrack
                   -> backtrack
            -> backtrack
     -> TON -> HEN -> TIE -> TOE -> NON(found result: THTTNOIEN)
                                 -> YON(found result: THTTYOIEN)
                                 -> backtrack
                          -> backtrack
                   -> backtrack
            -> backtrack
    -> backtrack
```

### 5.4 Analysis & Discussion

Similar to the letter based solution, the helper functions are mostly naive. While the results are still complete, and all valid solutions were returned; search time could be improved with better ordering of variables.

For one, the ordering of the unassigned variables only involved searching through the solution array for an empty slot; then grabbing the first category associated with that slot. An improved solution could choose the most constrained category. Figuring which category is most constrained would likely involve looking at the number of overlaps the category has with other categories.

Ordering domain values just returns a list of categories. A better solution would organize domain values by least constraining words. One idea for finding the least constraining word is to keep a letter count for each index of each category; i.e. category 'x' has 5 words with the first letter of 'a'. These numbers could be used to decide which words least constrain the categories it intersects with.

The consistency check checks the consistency of every category. In reality, only the categories intersected by the category being assigned need to be checked for consistency. Practically, given the number of categories and the size of the solution array, this almost always equates to checking every category. Implementing a smarter consistency check for this problem would yield little, if any performance advantage. Additionally, the performance for these small puzzles was not noticeably slow; solutions appeared almost instantaneously after running the algorithm.