Applying AI techniques to solve Crossword Puzzles

Student Name: Kevin Lampis

Supervisor Name: Magnus Bordewich

Submitted as part of the degree of Software Engineering to the

Board of Examiners in the School of Engineering and Computing Sciences, Durham University.

ABSTRACT

Context
Crosswords are a popular intellectual pastime for humans and so too serve as an interesting testbed for examining various Artificial Intelligence techniques and their efficiency. Some of the challenges an automated crossword solver would face include finding suitable solutions to natural language clues, making sure these solutions mesh correctly with each other, noticing when mistakes have been made and correcting them when they are.

Aims
The aim of this project is to develop an automated crossword solving system and using a combination of literature review and evaluative tests, investigate ways it can be improved in both speed and accuracy while only utilising consumer grade hardware.

Method
A simple crossword solver will be created from scratch. Then tweaks to the algorithm and additional features will be implemented to observe their affect on the solver's performance. Quantitative characteristics will be considered and used to evaluate the results which will include measurements for the speed and accuracy of various components as well as the system as a whole.

Results
The finished system can solve real world crossword puzzles pulled from the Internet with an average grid coverage of 75%. The solving process itself can take anywhere between a few seconds and several minutes depending on how many correct answers to clues can be found. The thesaurus worked very efficiently as a word source however the Google and Clue database websearch features added many more correct words with even fewer incorrect ones.

Conclusion
The solution is adequate for the task of solving crossword puzzles and sheds light on methods that could be expanded in order to increase the effectiveness of similar systems in the future.

Keywords – AI, Artificial Intelligence, constraint satisfaction problem, depth first search, backtracking, forward tracking, arc consistency, crossword puzzle, data mining

I. Introduction

Project Domain

Crosswords are popular intellectual puzzles usually found in newspapers. They are made up of a grid, typically 15x15 squares [1] with some of the squares blacked out. Crosswords are symmetric meaning when the grid is rotated 180 degrees, the locations of the black squares are preserved. Clues are given to help the user find an appropriate word to fit into each row and column. Where rows and columns overlap both words must share a letter at the overlap point. Without the clues crosswords would not be unique; many different words could take the same row and still finish the puzzle. Multiple words may be used to fill in a row or column but the adjoining space, comma or dash is not counted as a letter i.e. RAW BAR is entered as RAWBAR. Clues should also pass a substitution test, where the clue and the actual word can be used interchangeably in a sentence and the meaning stays the same i.e. for the clue "poet's performance" one could write "I've seen a video of e.e. cummings giving a poet's performance." or "I've seen a video of e.e. cummings giving a reading".

An automated crossword solver may face a number of challenges including finding suitable solutions to natural language clues, making sure these solutions fit onto the grid correctly, recognising when mistakes have been made and taking appropriate action to rectify them when they are.

Like a human solver, an automated system would need to guess the answers to the given clues and then attempt the fit those answers onto the crossword grid. Some of the key issues with finding solutions to clues involves parsing the natural language of clues and occasionally coping with odd gimmicks such as "See three across" or "Author of two down, three across and five across". Additionally there is a vast sea of various special interest topics or references to current affairs or popular culture that may be mixed in with the clues.

Actually laying words on the crossword grid is a slightly different affair for an automated solver. Since it cannot be as confident of clue solutions as a human solver, an automated system is likely to have a much wider pool of candidate words to choose from when it comes to filling out the grid. An intelligent way of guessing which combination of clues will make up a valid solution is essential in order to coax the best possible performance from such a system.

Project Purpose

The purpose of this project is to create an automated crossword solving system which builds upon past research within the context of modern technology and explore different methods for enhancing this system's performance. Advances in the efficiency of commercial grade hardware and the speed of public internet access means that certain techniques used by past researchers may prove to be more effective if correctly adapted to this modern technological environment.

System performance was evaluated at various points in the project to provide guidance as to which techniques should be expanded upon to improve the system. By the end of the project we had a system that was capable of taking a standard crossword puzzle from a reputable source such as the Guardian website and returning its most complete solution within the space of a few minutes. Initially we were aiming for the system to achieve a completeness of greater than 50%, since 50% can trivially be achieved by inserting words into only down columns or only across rows with no overlapping words. By the end of the project the system was capable of finishing crosswords with an average grid coverage of 75%.

Deliverables

The project was broken down into three broad stages: minimum, intermediate and advanced objectives. These were self contained objectives and although some of them built on top of previous ones, the system had functional capabilities that were evaluated at each point. Below is a list of the original objectives, all of which were achieved.

Minimum objectives – Create a basic system that will read in crosswords form a textfile, look up clue words in a dictionary & thesaurus and implement a simple backtracking algorithm to find a solution with the candidate words it has generated.

Intermediate objectives – Build upon the basic system by adding multitreading support, appending Forward Checking to the backtracking algorithm and creating a simple Graphical User Interface to improve user interaction.

Advanced objectives – Add web search module as an additional word source and improve the system's ability to maximise completeness after a deadend has been reached.

II. Related Work

This section outlines part of the research we conduction prior to undertaking this project. It was very important to survey previous attempts at creating automated crossword solving systems to not only find a starting point for designing our own system but also to identify the general trends of what approaches worked best and areas that could benefit from further work or being re-examined from a new angle.

All of the previous work pertaining to crossword solving systems that we have investigated all used a similar approach centred around modelling the crossword problem as a Weighted Constrain Satisfaction Problem (WCSP). A WCSP differs from a traditional CSP with the addition of "soft constraints" which are defined like the pirate's code, "more what you'd call 'guidelines' than actual rules." The added dimension of soft constraints alters the goal slightly from finding a solution that does not violate given constraints (CSP) to one that does not violate given constraints and satisfies the soft constraints as much as possible (WCSP). This fits in perfectly with crossword puzzles which have both hard constraints and what can be described as soft constraints. The hard constraints being the length of the words and how they mesh with other words horizontally and vertically. The soft constraints pertaining to how well the selected words match the given clues.

Previous Approaches

In their paper, Solving Crossword Puzzles as Probabilistic Constraint Satisfaction (1999)[2], Shazeer et al discussed different methods of evaluating the effectiveness of different solving methods. Since there is only one correct answer but several possible solutions that do not break the rules of the game there needs to be a way of determining how appropriate or "fit" a given solution is. In particular two solutions are mentioned in the article, the maximumprobability solution and the maximum expected over- lap solution. The former is an all or nothing measurement, either the whole solution matches the correct answer or it doesn't. The latter is measured by how many individual words in the solution match the individual words in the correct answer.

The main focus of their paper[2] is to think of the crossword as a constraint satisfaction problem. Words must be found that are the correct length and mesh with nearby across or down words. The word must also correspond with a given clue. This creates a large scope of candidate words to choose from for each row or column. The method detailed in the article assigns a probability to each candidate word and an A* search is used to solve the constraint satisfaction problem using these probabilities as heuristics. This was the first recorded approach that modelled crosswords as a WCSP and sparked a succession of further work.

Shazeer et al's original work grew into a full scale project called Proverb [3]. This system is split into two, candidate generation and grid filling. The grid filling part is mostly based on their original work[2], but the former is new. The candidate generation part takes a clue and generates a list of candidate words using a variety of resources. Each candidate is assigned what they call a "confidence-score". The grid filling part takes the candidate lists and fits them into the crossword

grid so as to maximise the overall “confidence-score”.

The candidate generating process is made up of a number of “expert modules” which gather data from different sources. The type of modules are: word list, which is a crude lookup of which words match the required length in a dictionary; Crossword Database, which is a database of crossword clue and their answers; Information retrieval which looks for answers within text such as encyclopaedias; Database, which extracts data from domain specific databases i.e. imdb for movies; and syntactic modules which return answers for fill in the blank clues.

WebCrow, a 2005 system, shares similar design properties with Proverb [4] and splits the problem into two parts: candidate generation with confidence scores and grid filling to maximise the confidence score of the puzzle. However, unlike Proverb, Webcrow is a system that uses a different approach to the precompiled database lookup model. The core feature to this system is a Web Search Module, using the internet to find possible solutions to clues. The fact that the exact length of the required word is known in crossword puzzles, web search can be a powerful way of solving clues. Also since WebCrow is an Italian project, emphasis was placed on being language neutral; by using the Internet, WebCrow can solve crossword puzzles in different languages, not just the language the developers focused on.

The web search module works by utilising a search engine like Google. The clue is first processed by the system to remove uninformative words and generalise some of the words i.e. change tense and plurals. Once the search is made it is necessary to download the entire document as it was empirically observed that the answer to a clue was very rarely found in the small abstract of the document that search engines provide. The system downloads several documents in parallel and implements a strict timeout rule for websites that do not reply quickly enough. If this happens then a cached page from the search engine is sought for. If this does not exist then the document is declared missing. Overall 50 documents was deemed the optimal number to balance time and accuracy.

Each document is converted (from html) to plaintext. Words of the correct length are extracted to form an unweighted list. Adjacent words that fit together to form the right length are also extracted but only if they appear more than once. To assign confidence scores, a statistical phase uses a combination of the search engine rank of the document containing the word and tf*idf.

The most recent crossword solving project is Dr Fill in 2010 [5]. Like Proverb and other systems before it, Dr Fill models the crossword as a Constraint Satisfaction Problem and uses a variety of techniques to find answers to clues.

Dr Fill searches for clue answers from a number of different sources: crossword databases, dictionaries, thesaurus and wikipedia. Crossword Databases hold records of crossword clues and their corresponding answers to provide straightforward lookup functionality . Two dictionaries are used by the system, a small one to contain common words and large one to contain “everything” and is compiled from different sources including Moby. A database of synonyms was collected from an online thesaurus and also grammatical information from the WordNet project. This includes words along with their roots and parts of speech i.e. “walk” is the root of “walked”. The last source of data is Wikipedia. Dr Fill uses a list of every pair of consecutive words appearing in the online encyclopaedia as well as useful names and phrases to help with phrase development and fill in the blank type clues.

Candidate Words are given a confidence score based on 5 criteria. A word scores highest if it and its clue match an entry in the Crossword Database; it has already been used in another crossword. The second criterion is part of speech analysis. If the clue gives an indication as to what part of speech the answer should be then words matching that part of speech score higher. Word merit is the third and is based on abstract standards such as the word's Scrabble score, obscurity and use of rare letters. The forth criterion is abbreviation. If the clue is abbreviated then the answer may be too. The last criteria pertains to fill in the blank type clues where the clue is generally a common phrase with a word missing and potential answers are pooled from Wikipedia.

The word filling part of the system relies on filling in easy clues first and leaving the harder ones or multiword fills till last to avoid unnecessary backtracking. The search algorithm used is a

Limited Discrepancy search and not a Branch and Bound approach.

WCSPs are usually solved with branch and bound. The algorithm works by searching a tree where each node represents a variable assignment. During the search the best solution so far is kept and called the Upper Bound. At each node the best solution in the subtree below is calculated and called the Lower Bound. If the Lower Bound is bigger than or equal to the Upper Bound then the subtree below the current node is left since it cannot improve upon the current best solution.

However, Dr Fill's creator Ginsberg argues that Branch and Bound suffers from the "early mistake" problem where an erroneous selection early on has a significant effect on the rest of the solution following the mistake. If an error is made very early on then the time taken to fix the original error was unacceptably long within the scope of Dr Fill and crossword puzzles in general.

Limited Discrepancy search works by having some kind of heuristic. The "discrepancy" of a solution is the number of times it violates the heuristic. The algorithm iteratively uses a depth first search in order of discrepancy. Each pass it relaxes the permitted discrepancy in the solution thus allowing less favourable solutions to be attempted without searching the entire search space.

Constraint Satisfaction Problems

An article by Vipin Kumar [6] includes an in depth discussion of various algorithms used to solve a CSP. The most obvious method of solving is called generate and test, where values to variables and selected and combined to see if they satisfy the constraints. This is a crude and inefficient method.

Backtracking is a more sophisticated algorithm that assigns values to variables in a specific order. If ever a point where the constraints are violated is met, backtracking is performed to go back to a variable that had possible alternative values and try one of those. This effectively cuts out chunks of the search tree where an initial combination of values did not satisfy the constraints and therefore anything leading on from that point need not be searched. The main drawback of backtracking is it is prone to thrashing i.e. searching through values a given variable keeps failing for the same reason which is not resolved until backtracking is performed again to an earlier part of the search tree.

CSPs can have unary, binary or n-ary constraints [7]. An unary constraint is a constraint on the value of individual variables. i.e. 3 down must be 5 letters long. A binary constraint is a constraint relating to a pair of variables; the Cartesian product of their potential values i.e. 3 down and 4 across must share a common letter at a specific point. Because the variable domains are finite in size, both the unary and binary constrained domains are also finite. This means there is no danger in using a Depth First Search since there will always be a bottom of the tree where the search will return from and not go on indefinitely.

However within the context of CSPs Depth First Search can be very inefficient since it will continue searching past a point where the constraints of the problem have already been violated. To streamline searching for a CSP one can use Backtracking to check if any of the current variable assignments violate a constraint before proceeding with the next variable selection. If such a condition is detected then the current solution is rolled back to a point where the variable values are valid.

A further problem is that a particular variable assignment may eliminate all possible values for another variable further down the line. With Backtracking alone this wont be detected until the second variable is reached and there are no possible values to choose from. The time spent searching until that point is wasted and the solution needs to be backtracked up to the variable assignment that cause the difficulty. Forward checking can be used to fix this ahead of time. After each variable assignment Forward checking will eliminate all the values of future variables that do not meet the new constraints set by the last variable assignment. It is then possible to immediately tell if a particular variable assignment will cause the problem to be unsolvable later and fix it before continuing with the search.

Arc consistency is a further improvement that can be made. This extended Forward checking by testing all the potential variable values to see if they can be assigned without violating the constraints and removing those that don't. The test needs to be run again after removing a value

because some other value may now be useless without the one that was just deleted. Arc Consistency is usually used as a preprocessing step be can also be used through the search. Some problems may be solved with Arc Consistency alone if all that is left are single values for each variable.

Summery of Related Word

For the purpose of creating a basic system to work with we decided it would be best to heed Ginsberg's warning[5] and avoid the branch and bound algorithm. Instead we will opt for Depth First Search with Backtracking. This will give us the opportunity to extend the system through various means suggested in Kumar's paper such as Forward Checking[7].

One particular approach we liked for extending the basic system we envision was that of Webcrow[4]. The Internet is an ever expanding resource of data, much of it for more relevant to the socio-polititcal nature of crossword clues than traditional dictionaries and thesaurus. One of the main reasons this approach could benefit from being reexaminied in our own research is the advance of communication technologies. Home broadband today is far quicker than it was 8 years ago, making web search potentially more practical and effective than ever.

III. SOLUTION

This section outlines the design and implementation process of the solution. The solution is an automated crossword solving system with the features discussed in the introduction of this paper. Initially a basic system was created and then optimisations and a graphical user interface were added later. This section is split into 4 parts, Word Generation, Solving Process, Optimisations and Development Process.

High Level Description

Several existing systems split the problem of solving crosswords into two logical parts [3,4]. The first part involves generating lists of candidate words in response to clues. The second is laying out those words on the crossword grid in a way that does not break the rules of the game.

Initially a system was required to take a text file from the user which contained the clues, their location on the grid and the number of letters expected. For this we designed a bespoke file format that was easy for both humans and machines to understand. To avoid tediously generating crossword puzzles by hand for testing we wrote a script to automate the process of downloading puzzles from online sources, parsing them and then writing the data to disk in our file format.

For word generation the system must take the clue and the number of letters expected and run these through several word sources to generate a list of words with the right length that are related to the clue.

For the solving part, the problem is essentially a Constraint Satisfaction Problem[2]. The most popular search algorithm for solving this type of problem is a depth first search with backtracking. The system takes the lists of candidates for each clue and performs the depth first search while making sure that each variable assignment does not contradict the constraints imposed by the existing nodes in the solution. Since there are a finite number of word combinations there is no risk of a bottomless search tree. In addition to this a backtracking algorithm is used to climb back up the tree after reaching a dead end. Dr. Fill [5] used a system like this as opposed to branch and bound, which the author argues suffers from the early mistake problem, where a mistake early on will not be recognised until much later causing much wasted effort.

If after searching the entire search space a solution cannot be found then the system informs the user and instead displays a handful of the system's best attempts at a solution. The system can improve upon these best attempts by skipping over clues for which no solution can be found. This shifts the system's focus from finding a complete solution as quickly as possible to finding the most complete solution it can, which are very different goals, especially when Forward Checking is used. We call this method *keep going* and it will be examined in more detail later on.

Basic Word Generation

Word generation works by aggregating words of an appropriate length to match a given clue from several different sources. Initially sources stored locally on disk or in memory were used but later a web based source was introduced to examine the affect on the system.

The first source used was a straightforward thesaurus. Each word in a clue is looked up in the thesaurus and any words returned that match the length requirement are added to the list of candidate words. This thesaurus was compiled by extracting data from multiple online thesauruses and stored locally on the hard drive as a flat file.

The second source is a reverse dictionary. To explain what we mean by a reverse dictionary, first imagine a regular dictionary for a moment. One would look up the word “tomato” in this dictionary and find the definition “An edible, fleshy, usually *red* fruit.”. Looking up “strawberry” would yield: “fruit that consists of a *red* fleshy edible receptacle and numerous seedlike fruitlets.” For our purposes it would be much more useful if we could look up an entry and receive a list of words that contain this entry in their definitions. For example, looking up “red” would give us “tomato, strawberry, etc...”; essentially a crude list of things that are red or associated with red. This is what we called a reverse dictionary. We generated the reverse dictionary from a number of normal ones and it takes the form of a file directory containing numerous text files, each named with a word and containing a list of words associated with its file title. Now if we were posed with a crossword clue “red fruit” we would look up “red” and “fruit” in our reverse dictionary and, among other things, have returned “tomato” and “strawberry”.

Advanced Word Generation

The web-search function provides an extra source of candidate words to be pooled. The user will have the option of including web-search as a source or not at runtime. Web-search works by retrieving a page from the internet and parsing it for relevant words. It was deemed that regular expressions were the best way of doing this since we're only seeking to extract words that do not appear inside HTML tags. A sax parser would be impractical because many websites do not use valid XHTML, so every page would need to be validated and possibly fixed before parsing. Since the documents are only needed once, a DOM parser is also impractical due to the inherent overhead.

The web search module has a strict 60 second timeout. The object that performs the actual document fetching is run in a separate thread, if it does not complete in time then the thread is killed and an empty string is returned to the calling function.

Several special purpose online resources are utilised in the web-search module, in total there are three types of web-search included in the system.

The first is Wikipedia and its “list of” pages. Words appearing in the clue are looked up on Wikipedia to check if there is a list entry for that item. For example, if the clue contained the word “composer” searching Wikipedia would return a “List of Composers”. This webpage is then parsed and words of the right length are added to the list of candidate words.

The second web-search source is an online database of popular crossword clues and solutions. Several existing crossword solving systems use this approach such as Proverb[3] and Dr Fill [5]. In both these systems, words found in the Clue Database carried the highest confidence scores and were attempted first during the solving stage.

The third web-search source is a much more broad. The whole clue is used to query a search engine and the first several documents are downloaded and parsed for words of the right length. By crossreferenceing which words appear the most over several documents the system gets a good idea of how relevant each potential solution is to the clue words. This method was intended to work particularly well with fill in the blank type clues where the missing word is part of a popular phrase or idiom.

Solving

Algorithms

Once the system has a list of candidate words for each clue it must “solve” the puzzle by finding a

combination of words that will mesh together correctly on the crossword grid. A number of Artificial Intelligence techniques are utilised to make the searching process more efficient.

The crudest method of searching would be to take one candidate word assigned to each clue and try to squash them onto the grid. If they do not fit together, then replace the words one at a time until a solution has been found or until every combination of words has been attempted. Although this will surely work it is very inefficient and will take much longer than it needs to.

By analysing the problem itself we can infer specific properties that will help us solve it more efficiently. The key element of a crossword puzzle is that words for different clues must share common letters at certain points. This is a constraint that is placed on any potential solution so we call this a constraint satisfaction problem.

To find a solution for our crossword puzzle we will need to search through different combinations of words to find a solution. The search algorithm used to do this is a Depth First Search. Since there is a finite number of word combinations there is no danger of a bottomless search tree. A depth first search assigns a word to each clue until either all clues have a word or a clue is reached that cannot be assigned a word. A technique called Backtracking is used in cases where the latter occurs. The system will go back to a previous assignment and replace the word given to that clue with a different word, then continue the search. This way the search does not get stuck and can keep going after setbacks.

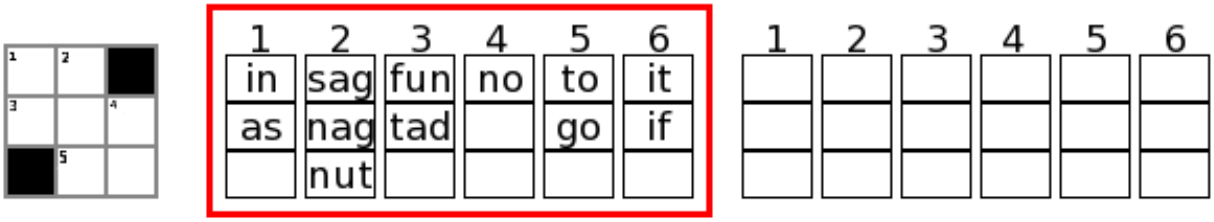
Since this is a constraint satisfaction problem there are more techniques we have applied to improve efficiency. When a word is assigned to a clue, this affects the choices that can be made when assigning words to other clues. For each clue we have a list of potential words that can be used. Before assigning one of these words to a clue, the system checks that none of them conflict with words already assigned to other clues. This is something we can only do because it is a constraint satisfaction problem. By using this technique we can be sure that at any point during the search, the partial solution that is currently being built upon is valid or "consistent" with the constraints of the problem. This is a far better approach than waiting until all the clues have words and then checking if there are problems.

Kumar discusses in his paper [6] the inefficiencies of Depth First Search alone when applied to Constraint Satisfaction Problems. He suggests that a technique called forward checking should be applied to address these shortcomings and this is the last technique we implemented. The purpose of forward checking is to spot dead ends as early as possible in the search. Every time the system assigns a word to a clue, it checks the candidate words for other clues and removes words that conflict with the recent assignment. If no possible words remain for a particular clue then this means a dead end will ensue at some point in the future. Instead of continuing with the search the system undoes the assignment that will cause the dead end and tries a different word.

Details

In this section we will go through how the algorithms discussed above were implemented in more detail. Each clue has a stack of candidate words associated with it called the master stack and another, empty one, called the solver stack. As seen in Figure 1. In our example stack 6 is used for clue 1 down.

Figure 1, showing the Master Stack with all the candidate words for each clue highlighted red, the empty solver stack to the right, and the crossword grid.



The system copies the contents of the first master stack into the first solver stack. It then copies the contents of the second master stack into the second solver stack but omits all words that conflict with the head of the previous stack. Figures 2 and 3 below show an example of this.

Figure 2, showing the population of the first solver stack corresponding to clue 1 Across.

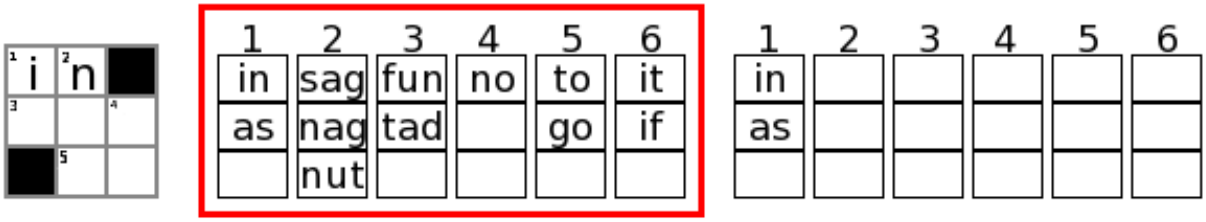
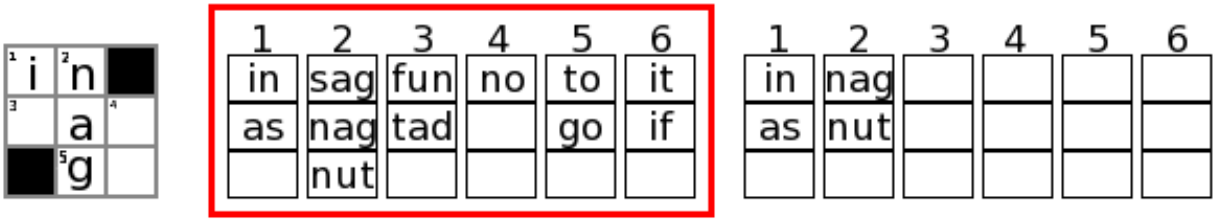


Figure 3, showing the population of the second solver stack. Note that the word "sag" has been omitted because it conflicts with the assignment of "in" for clue one.



If the system reaches a point where no words can be carried over from the master stack to the solver stack then this is a dead end. The system will backtrack to the previous stack, pop the first word and then continue with the new head as demonstrated in Figure 4. If the last word on the stack has been popped then the system will backtrack again to the stack before that. If the system cannot backtrack any further i.e. all the words on the first solver stack has been popped then this means there is no solution using these words. If, on the other hand, all the solver stacks have at least one word in them then a solution has been found and comprises of the head of each stack. Figure 5 shows this.

Figure 4, backtracking. Here we see that for clue 4, the word "no" does no fit. This is a dead end. To resolve this the system will pop "tad" from stack 3. Now that this stack is empty it must backtrack again to stack 2 and pop "nag". Now it will continue the search using the word "nut" for clue 2.

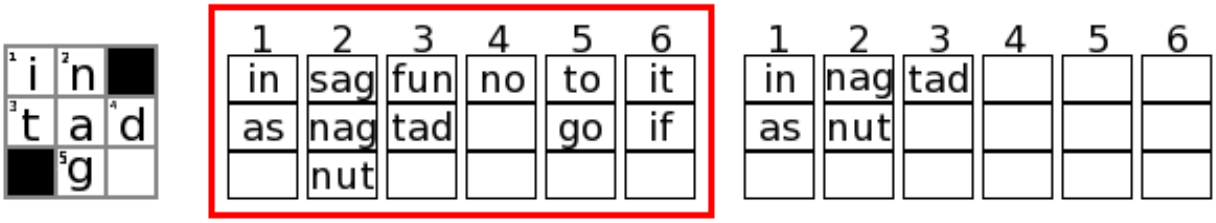
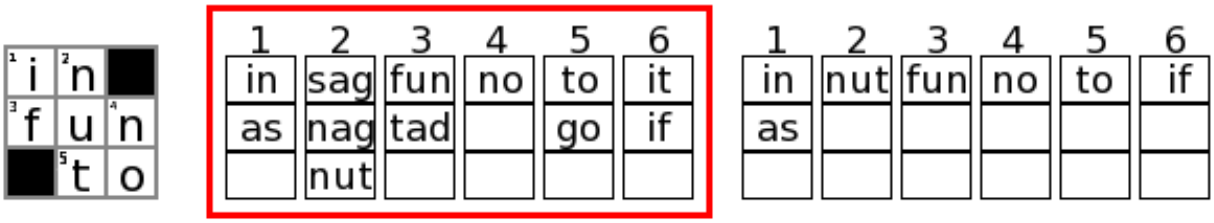
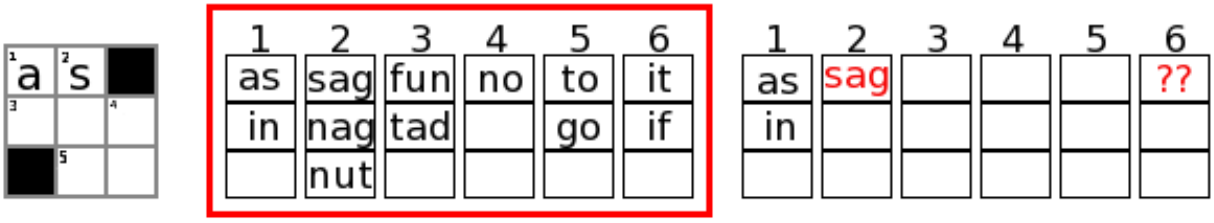


Figure 5, a solution has been found.



Forward checking is a mechanism that will come into play each time a new word is assigned to a clue i.e. when a master stack is copied into a solver stack or when a word is popped from a solver stack. The system will check which unassigned clues intersect with the clue that has just been assigned a new value and check that at least one remaining candidate word for each clue. This is shown in figure 6.

Figure 6, forward checking. After assigning the word "as" to clue one across the system will perform forward checking. Clues 2 down and 1 down (stack 6) intersect with clue 1 across. We see that "sag" is still potentially okay for stack 2 however there are no words for stack 6. This indicates that "as" is not apart of a viable solution. Without forward checking this mistake would not have been realised until much later.



Dead Ends

We must be realistic about the capabilities of the system. Similar systems in the past such as Dr Fill [4] achieved a 70% success rate. This means dead ends where no complete solution can be found are inevitable. In these cases we want the system to return a handful of best solutions so far i.e. solutions that fit the most number of candidate words into the grid. At this point the system will attempt to improve this solution if prompted to do so by the user. This is called the "keep going" function and works by identifying which clues are lacking appropriate words in the best solution and rearranging the order in which the clues are attempted in such a way that the difficult ones are attempted last. It is already established at this point that there is no complete solution to the puzzle so there is little additional overhead in this rearrangement.

Optimisations

Word confidence

In its simplest form, we can be more confident of a candidate word if it appears in multiple sources. If the reverse dictionary returns "tomato" and the thesaurus returns "Communist" and "tomato" then we can be more confident in the "tomato" solution. Each word source has a "confidence-score" assigned to it and all words emanating from that source are given a respective score. If a word is found in multiple sources then the aggregate score is used to boost the system's confidence in that word. This allows the system to attempt the words it is more confident of first and reach complete solutions quicker.

The user has the option of altering the confidence scores of various word searches before the lookup is performed. This allows the user to experiment with the scores and have them reflect the

quality of each word source. For example the online clue database is far more likely to respond with the correct solution than a random Google search.

Stack Ordering

The depth first search algorithm iterates through the stacks of candidate words in a predefined order. Initially this order will be the same order in which the clues are presented to the system. However, changing this order can have a profound impact on the performance of the system. The system reorders the stacks twice prior to initiating to searching process to take advantage of this.

The first reordering is simply based on the number of candidate words each stack contains, ordering them in ascending order. Imagine our first stack has 10 candidate words and our last stack only has one and these two words must mesh on the first character of each. Like so

one tree
two
three
…
ten

With this ordering we will have to attempt
one – tree
two – tree
three – tree
…
ten – tree

Now a simple rearrangement of the stacks would put the one of only 1 candidate word first. Now it is apparent that only 3 of the previous 10 combinations are valid

tree – two
tree – three
tree – ten

The search space has therefore been cut by 70%.

The second rearrangement is a bit more complicated and based on a human-like trait. When attempting a crossword humans will pick a clue they are confident in and branch out from there. This makes the meshing operation easier and will avoid nasty sunrises where two good words have been found but then there is no way of adding a third that intersects with them both. The system accomplishes this by taking the first clue, then iterating through the rest of the clues until it finds one that intersects with the first. Then it takes the second clue, and goes through the remaining clues to find one that intersects with that. If none is found then it will look for some that intersects with the first. This way the stacks will be in a logical sequential order, having the stacks with the fewest candidate words closer to the front.

Multithreading

The bulk of the system runtime is taken up by the search algorithm which is purely a processing task. This means that a significant speed boost can be achieved on multicore processors by using threads.

In order to split the task fairly between a number of threads the system will take the first clue stack and compare the number of candidate words with the number of cores available on the CPU. For example an Intel i7 processor has 8 logical cores. If the first stack only has one candidate then it will check the next stack. If the second stack has 8 words then 8 sets of stacks are prepared, one for each core. They will all contain the same words except the second stack of each set, which will only contain 1 of the original 8 words. This way the task will be evenly split into 8 pieces, one

for each CPU core. If the second stack has 15 then 4 would go to the first 7 sets of stacks and 3 to the remaining one. If all the stacks have fewer candidate words than the number of cores i.e. 7 or less in our example, then the stack with the most is selected and the number of cores used is adjusted to that amount. This is the simplest way of dealing with such a situation and the time needed to search such a small search space would be trivial enough to not even need multiple threads anyway.

After these threads have been started a list of threads is checked every second. When a thread returns a result a loop will pick this up and remove it from the list of remaining active threads. When either all the threads are finished or an optimal solution has been found the system kills off any remaining threads and continues to the result reporting stage.

Graphical User Interface

Although it is be possible to operate the system from the command line, a Graphical User Interface provides better interaction with the user and offers many more options for visualising the solving process.

The GUI comprises of a drawing plane where the crossword grid is painted onto the screen. As the system progresses through the solution it will display key points on the grid, showing the current list of words that are being investigated. At the end a number of the best solutions will be available for the user to display on the crossword grid.

The GUI offers a field where the user can select a puzzle file to check or simply use parts of the system in a stand alone fashion. i.e. look up a word or phrase to see what candidate words the system generates.

The user interacts through pushbuttons. One to select a puzzle file, one to tell the system to generate candidate words for the given clues and one to initiate the solving process. There are also extra pushbuttons designed to allow the user to control the flow of the solving process by pausing it or skipping ahead. In addition to this the user has several widgets for supplying specific settings. There are comboboxes to choose how many threads to use and which thread to visualise. Tickboxes allowing the user to indicated which data sources the system should use to generate candidate words. And a textfield to change the delay between visualisation transitions.

While the solving process is taking place the GUI will paint the current combination of words being attempted. This is updated once a second to avoid squandering resources on it. If multiple threads are active then the user has the option of choosing which thread to follow.

Hardware and Software

The application is written in C++ and the GUI will written with the QT4 framework. C++ was chosen because it is a standard commercial language widely used for it's flexibility and performance. The QT4 framework is extensive, easy to use and crossplatform. The system adapts the number of threads it uses to the capabilities of the hardware it is currently running on at runtime.

Tools

The initial stages of the implementation were written using the Eclipse-cpp Integrated Development Environment however when the GUI was added a simple text editor was used instead since Eclipse and QT do not work well together. Debugging was done with GDB and the compiler used was GCC.

Testing

The system will be evaluated both as separate parts and as a whole. The main features that must be tested are the word generation accuracy and efficiency and the solving process's speed and completeness. Several tests will be devised to evaluate each part of the system. The tests will be automated and run on a large number of sample crossword puzzles to maximise the accuracy of the results. The purpose of these tests will not only be to demonstrate how effective the proposed solution is but also the effects of various optimisations. It will also serve as a tool for analysing any bottlenecks in the system and planning ways of addressing them.

IV. Results

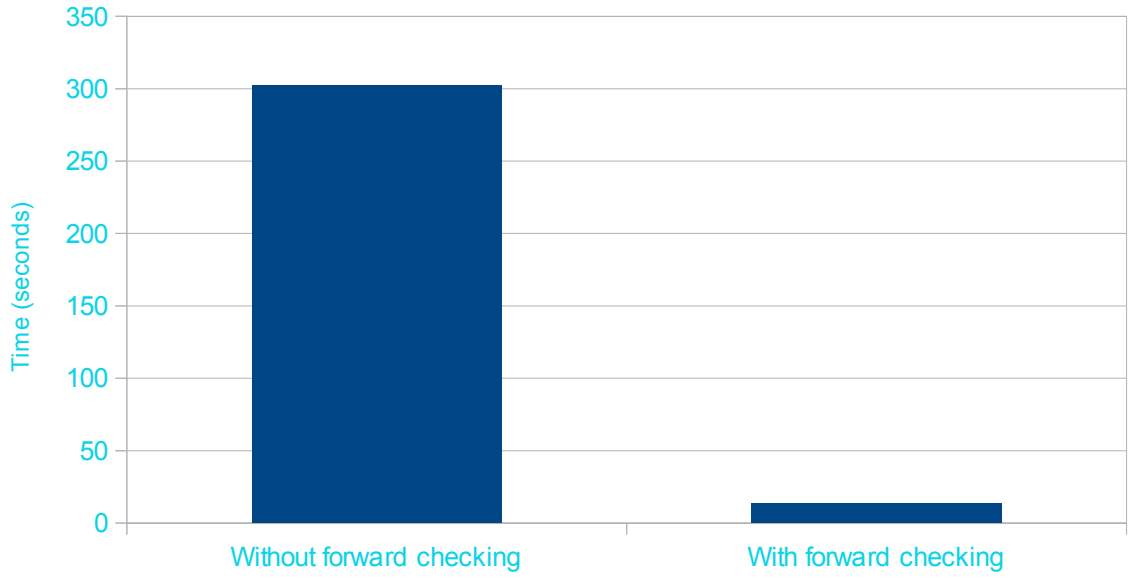
In this section we present the various tests we ran on our system along with the results of those tests. Four specific system properties were tested: Speed of the solving process, Accuracy of the word sources, the effect of stack size on solving speed and the aggregate speed and completeness of the system for different solving methods.

The tests were carried out on an Intel i7 920 desktop with 8 logical CPU cores[13]. All files used by the system were mounted in tmpfs[11] to minimise disc access time. Timing was calculated by making a call to TIME(1)[12], which returns the number of seconds since the UNIX epoch, at the start and end of each action to be timed and taking the difference of those two values. Since each process is expected to take a number of minutes the time taken to call the TIME method is negligible. All tests were performed on "Quick" crossword puzzles originally published on the Guardian's website [14]. Puzzles were selected from a variety of authors and publication dates ranging over ten years.

Speed

This is a straightforward runtime test to see how quickly the solving process performs. We created a number of contrived puzzles with a fixed number of candidate words per clue (100). Each puzzle had a full solution. The correct word was inserted at the end of each stack of candidate words. The two solving methods we compared were Depth first search with forward checking and without forward checking. These results are the average from 400 Guardian Quick crossword puzzles.

Figure 7 Speed

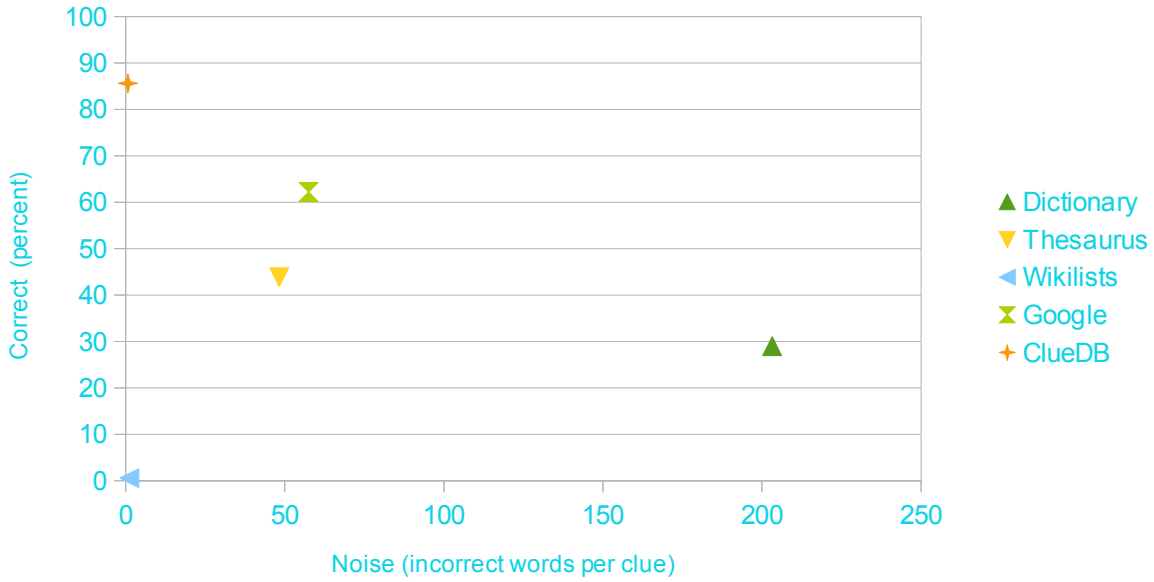


Accuracy of Sources

To test the accuracy of our word finding solution we devised a test of several clues extracted from puzzles along with their solutions. We rated each source of candidate words by how many correct solutions they found along with how many incorrect words they returned. These are important measurements because the total number of words that each source returns has a large impact of the solving time. Since there is only one correct word per clue, the ideal total number of words returned should be as close to one as possible. The word sources we compared were: Reverse Dictionary, Thesaurus, Websearch Wikipedia lists, Websearch Google, and Websearch clue database. In the graph below, the y axis denotes the percentage of correct words the source returned and the x axis

shows the noise which is calculated by removing the number of correct words from the total words a source returns and dividing by the number of clues. This is a count of the number of incorrect words per clue and ideally should be as close to zero as possible. These results are the average from 400 Guardian Quick crossword puzzles.

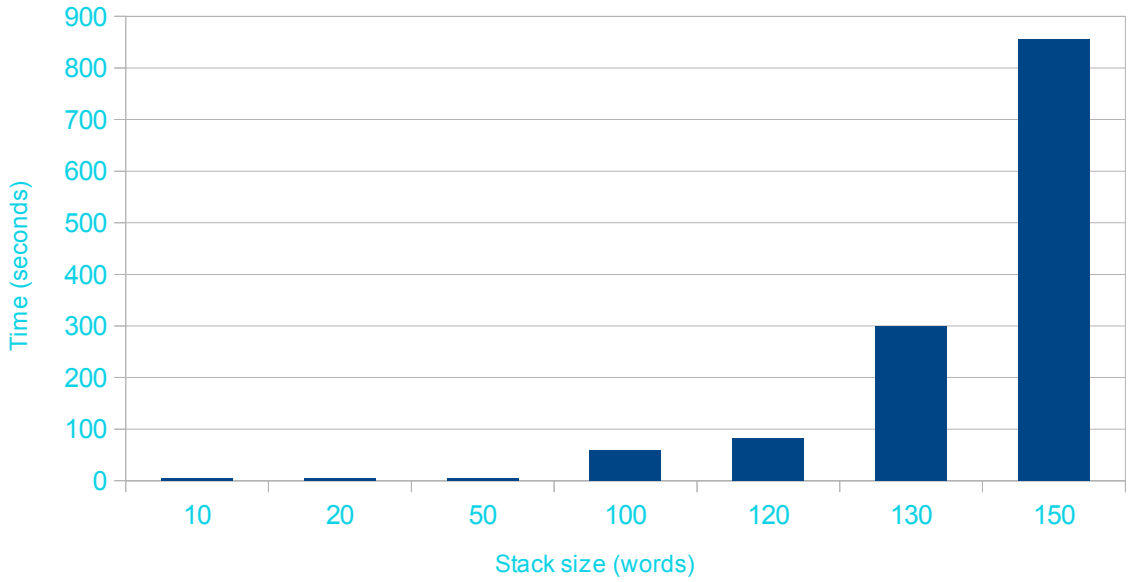
Figure 8 Accuracy of Sources graph



Stack Size and Speed

This test was to investigate the effects of increasing the number of candidate words on the solving time of the system. We swapped the system's legitimate word sources with a random letter generator since the focus of this test is not on solving puzzles correctly but on what effect increasing the number of candidate words per stack has on the speed of the system. We ran the test several times over a selection of 100 puzzles each time. The system curtailed the number of candidate words per stack to a different number for each test: 10, 20, 50, 100 and 150. These results are the average from 100 Guardian Quick crossword puzzles.

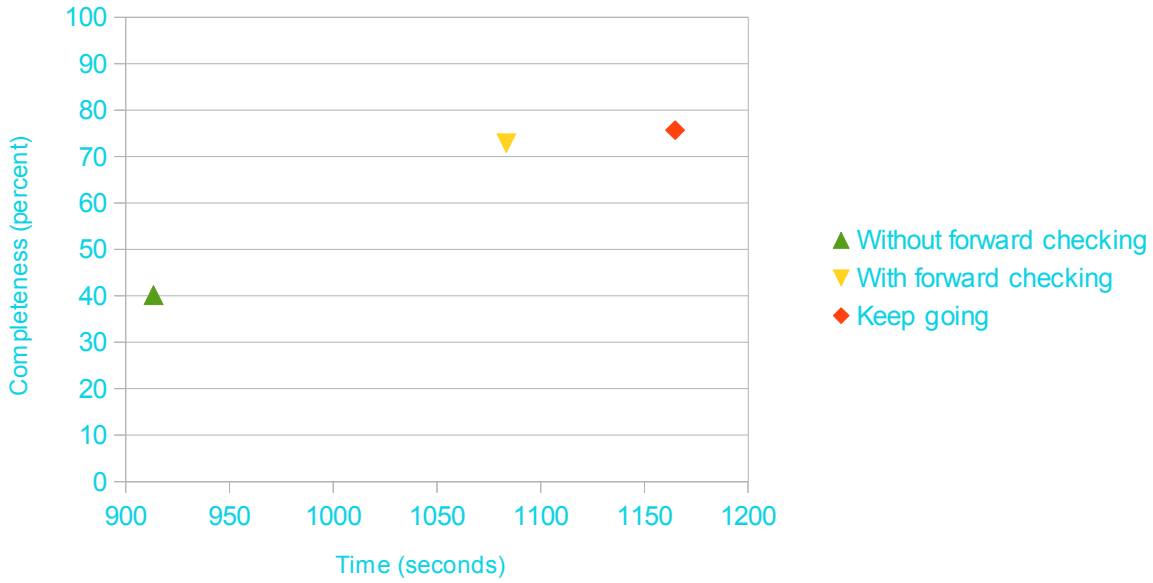
Figure 9 Stack Size and Speed



Speed and Completeness

In this test we used all word sources and compared the speed and completeness of different solving methods: Depth first search with forward checking, without forward checking, and with the *keep going* method. In the graph below the y-axis denotes the percentage of the grid filled and the x-axis shows the time in seconds. These results are the average from 50 Guardian Quick crossword puzzles.

Figure 10 Speed and Completeness



V. EVALUATION

In this section we will discuss the content of the Results section as well as the successes and organisation of the project as a whole.

Speed

Figure 7 shows that the addition of forward checking produced a significant increase in the solving speed. By looking ahead for future contradictions a lot of dead ends were caught early and no time was wasted investigating them. However this also means that less of the search space is covered and should the system not have the candidate words for a complete solution the forward checking method is likely to return shorter best attempts. In other words the system without forward checking will build bigger solutions before realising there is no complete solution, which takes more time. This produces a clear tradeoff between solving time and completeness, where no single method is best for both puzzles that have a complete solution and those that don't.

Stack Size and Speed

The number of words per stack has a profound impact on the solving time. The average time taken for 100 words per stack was 60 seconds and for 150 words 14 minutes. This suggests that we can tailor the expected speed of the solving process by curtailing the number of candidate words per stack. This would appear to create a tradeoff between solving time and accuracy however each word is assigned a confidence score so by removing words in order, starting from the on with the lowest confidence score, the effect on system accuracy ought to be minimal. For our test system the best results would come about from keeping the length of each stack under 150 words, or keeping the combined number of words across all stacks below 150 multiplied by the number of clues. The latter approach would be more accurate since not all sources will return the maximum number of words for each clue.

Accuracy

The accuracy of sources test shows that the clue data base was the most efficient source. It solved more clues than the other sources and returned an almost negligible number of incorrect answers. The next most effective source was the Google search followed by the thesaurus. Contrary to initial speculations, the Google search actually returned fewer words than three of the other sources, and bested only by the clue database. The Wikipedia lists source was completely useless since it returned almost no correct answers. The dictionary also returned a decent number of correct answers however it also returned more than twice the number of incorrect answers than any other source.

Although we did not explicitly test the speed of each source it is clear that the web based sources introduce significant time overhead compared to the local, in memory, sources. Each Google page returns 10 results, making 201 webpages that must be downloaded for a 20 clue puzzle. Each webpage can takeup to 10 seconds to download making the total time needed anything up to 30 minutes. This overhead introduces a need to balance time and completeness

Speed and Completeness

In figure 9 the results show that without forward checking the system takes less time to finish and returns less complete solutions. With forward checking the system returned more complete solutions. Although the average time with forward checking (1083s) is longer than without forward checking (913s) it should be noted that the quickest individual time with forward checking (698s) was quicker than the quickest individual time without forward checking (719s). This suggests that the two methods react differently in some situations since they were tested on the same set of 50 puzzles.

Surprisingly these results are contrary to our initial hypothesis i.e. that forward checking would be quicker but less complete. The reason we predicted this was because the forward checking method prunes the search tree by skipping ahead to find future contradictions. Without forward checking the system will cover more of the search space and should therefore conceivably take

longer to finish and find longer solutions. It is very anti-intuitive that with forward checking the system covers less of the search space but returns longer solutions. The reason our results show this happening is unclear; it could be down to experimental error, a bug in the system code or some other arcane reason.

As expected the keep going method slightly increases the word coverage in most cases with an insignificant increase to the solving time. However this method was not as effective as initially expected. On average it only increased the longest solution by 3%(approximately one word per puzzle). Ideally the longest solution should be the same length as the number of clues minus the number of clue solutions the system is missing. We think this may be down to a design oversight. The system does not attempt to identify which clue is causing trouble, instead it takes the best solution so far and blindly kicks the next stack to be attempted to the back of the queue of stacks. However this stack may not actually be the cause of the problem so it is more or less left to chance whether kicking this clue to the end of the list will help improve the best solution.

Organisation

The development process for this project was very rigid and plan driven. The objectives for the project were laid out early on and a development plan and schedule was drawn up. The plan was altered slightly in places during the process, for example websearch and gui were originally intended to be developed incrementally and in parallel but when the time came we instead decided that it would be better to finish the gui completely first and then move on to the web search. Although some slack time was utilised at the end to straighten out an obscure bug in the *keep going* module, the project was finished on time and with no schedule extensions or features dropped. Some argue that such an approach would stifle creativity due to its lack of flexibility [8] however I felt that the most important issue in this project was to do only exactly what was required and avoid any kind of function creep that would delay the project or add extra work.

For the architectural design of the system an Object Oriented approach was deployed due to general convention and with very little consideration of the implications. However this proved to be very unwieldy and made the system difficult to maintain. There was no opportunity to use inheritance and there was only one instance of every class. Pointers were passed everywhere to make sure all objects worked with the same instantiation of classes and certain features such as the static p_thread functions actually made it impossible to use multiple instances of a class in some sections. Overall there is great room for improvement to the system's architecture. Use of the singleton design pattern [9] could have made it much easier to manage single instances of classes and help them better integrate with each other. Alternatively a different paradigm to OO all together, such as imperative or Aspect Oriented Programming[10], may have proved to be a superior approach.

VI. Conclusion

The results of this project illuminate the vast potential for utilising the internet as a giant source of information as opposed to using static compiled databases of information. The use of depth first search with backtracking, forward checking, and various other optimisations proved to be a huge success, capable of churning through a vast number of candidate words very quickly.

It was unfortunate that the *keep going* method suffered from a design fault that limited its effectiveness. While it may have been straight forward to extend its functionality the project's rigid planning and scheduling did not permit this. It should be noted that this was an advanced objective so there always existed the possibility that it would not be implemented fully in the time available.

Despite this shortcoming of the *keep going* method, the search algorithm and its optimisations as a whole performed very well. However, the system cannot find lengthy solutions if the appropriate candidate words are not presented by the word sources in the first place. On reflection the project could have benefited from more time being spent refining the output of the word sources instead of the search algorithm. Overall the solution was suitable for the task since it

successfully solved real world crossword puzzles.

As discussed by Shazeer et al [2], there are two different objectives when solving a crossword puzzle: finding a complete solution or maximising the grid coverage. As this project has shown there is no single algorithm that will perform both functions equally well. Finding a complete solution as quickly as possible will not yield good grid coverage if there is no complete solution with the words available; even if there is only one word missing, the system will realise this very early on and abort the search. On the other hand, striving for maximum grid coverage from the start will take a much longer time to yield the complete solution if the correct words are available. The *keep going* method outlined in this project was an attempt to marry these two approaches, using one, then the other as the context of the running system changes.

There are a number of ways one could extend this project in the future, one of which would be to re-examine the web search word source, inspect why the Wikilists section failed to work as expected and tune the Google search section so that it supplies the correct words more often without increasing the number of incorrect words returned. It may also be interesting to observe if any speed increase can be gained from a hybrid approach where the system goes as far as it can using only the thesaurus then re-evaluates the remaining clues using the web search module.

REFERENCES

[1] Matthew L. Ginsberg (2011) Dr.Fill: Crosswords and an Implemented Solver for Singly Weighted CSPs. Available at: http://jair.org/media/3437/live-3437-6039-jair.pdf (Accessed 30/04/13)

[2] Noam M. Shazeer, Michael L. Littman and Greg A. Keim (1999) Solving Crossword Puzzles as Probabilistic Constraint Satisfaction. Available at: http://www.aaai.org/Papers/AAAI/1999/AAAI99-023.pdf (Accessed 30/04/13)

[3] Michael L. Littman, Greg A. Keim, and Noam M. Shazeer (1999) Solving Crosswords with Proverb. Available at: http://www.aaai.org/Papers/AAAI/1999/AAAI99-135.pdf (Accessed 30/04/13)

[4] Giovanni Angelini , Marco Ernandes & Marco Gori (2005) WebCrow : Solving Italian crosswords using the Web. Available at: http://webcrow.dii.unisi.it/webpage/docs/WebCrow_TR.pdf (Accessed 30/04/13)

[5] Matthew L. Ginsberg (2011) Dr.Fill: Crosswords and an Implemented Solver for Singly Weighted CSPs. Available at: http://jair.org/media/3437/live-3437-6039-jair.pdf (Accessed 30/04/13)

[6] Vipin Kumar (1992) Algorithms for Constraint- Satisfaction Problems

[7] Stuart Russell and Peter Norvig (1995) Artificial Intelligence A Modern Approach

[8] Richard Turner (2003) People Factors in Software Management: Lessons From Comparing Agile and Plan-Driven Methods

[9] J. B. Rainsberger (2001) IBM developerWorks: Use your singletons wisely. Available at: http://www.ibm.com/developerworks/webservices/library/co-single/index.html (Accessed 30/04/13)

[10] Gary Pollice (2004) IBM developerWorks: A look at aspect-oriented programming. Available at: http://www.ibm.com/developerworks/rational/library/2782.html (Accessed 30/04/13)

[11] Linux tmpfs documentation. Available at: https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt (Accessed 30/04/13)

[12] TIME(1) man page. Available at: http://linux.die.net/man/1/time (Accessed 30/04/13)

[13] Intel i7 920 specifications. Available at: http://ark.intel.com/products/37147 (Accessed 30/04/13)

[14] Guardian Quick Crosswords. Available at: http://www.guardian.co.uk/crosswords/series/quick (Accessed 30/04/13)