University of Tartu
Faculty of mathematics and computer science
Institute of computer science
Specialty of computer science

Pavel Tomozov

Crossword Construction using Constraint Satisfaction and Simulated Annealing

Bachelor thesis (6 EAP)

Supervisor: Mare Koit

Author: …..... …. mai 2013
Supervisor: …..... …. mai 2013

Allowed to defense
Professor: …..... …. mai 2013

Tartu 2013

Table of contents

Introduction

The main goal of this thesis is to create a crossword generating program, which uses two different algorithms, and then carry out an analysis of its effectiveness. The material will be used in the Artificial Intelligence I course in order to demonstrate students AI problems and their possible solutions, precisely constraint satisfaction method and simulated annealing. And also show the difference in their work and compare their effectiveness.

Chapter I begins with some general words about a crossword and algorithms, giving short remark on their history and development. Then the chapter proceeds to the short thesis description (why it is done, what the purposes and goals to achieve are). In the paragraph 1.4 there will be given some examples of similar problems and their possible solutions.

Chapter II will deal with program and its work description, giving a closer look at what program is capable of. In the second part a detailed description of algorithms that program uses will be given , from general to specific.

Chapter III gives an overview of the analysis. It begins with general description of test cases in paragraph 3.1. In paragraph 3.2 the results of these test along with the results of the analysis, algorithms effectiveness comparison will be presented.

Chapter IV contains the author's thoughts on the future development of the program.

Chapter V gives a brief overview of the work done and the results achieved.

The short overview of the thesis in Estonian language is provided in the end.

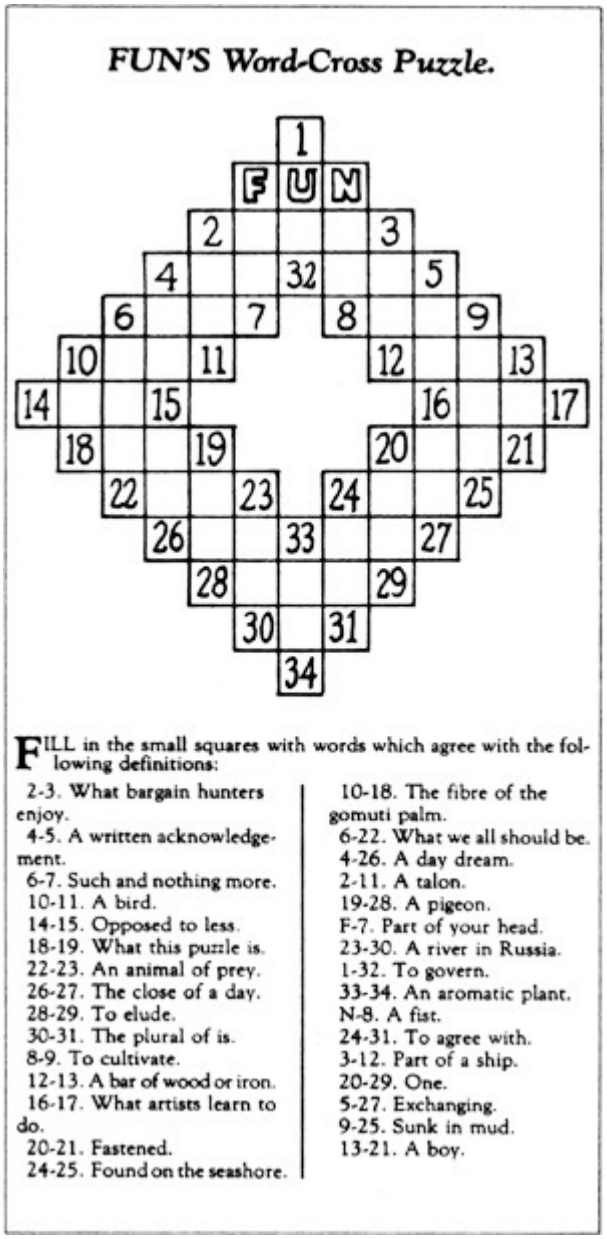
Program, user guide, tested grids and dictionaries are provided on the DVD.

1. Background and problem statement

1.1 About Crosswords

Crossword puzzles are said to be the most popular and widespread word game in the world, yet have a short history. The first crosswords appeared in England during the 19th century. They were of an elementary kind, apparently derived from the word square, a group of words arranged so the letters read alike vertically and horizontally, and printed in children's puzzle books and various periodicals. In the United States, however, the puzzle developed into a serious adult pastime.

The first known published crossword puzzle was created by a journalist named Arthur Wynne from Liverpool, and he is usually credited as the inventor of the popular word game. December 21, 1913 was the date then this game appeared in a Sunday newspaper, the *New York World*. Wynne's puzzle (see Figure 1 below) differed from today's crosswords in the way that it was diamond shaped and contained no internal black squares. During the early 1920's other newspapers picked up the newly discovered pastime and within a decade crossword puzzles were featured in almost all American newspapers. During this period crosswords began to assume their familiar form. Ten years after its rebirth in the States it crossed the Atlantic and re-conquered Europe.



FUN'S Word-Cross Puzzle.

FILL in the small squares with words which agree with the following definitions:

2-3. What bargain hunters enjoy.
4-5. A written acknowledgement.
6-7. Such and nothing more.
10-11. A bird.
14-15. Opposed to less.
18-19. What this puzzle is.
22-23. An animal of prey.
26-27. The close of a day.
28-29. To elude.
30-31. The plural of is.
8-9. To cultivate.
12-13. A bar of wood or iron.
16-17. What artists learn to do.
20-21. Fastened.
24-25. Found on the seashore.
10-18. The fibre of the gomuti palm.
6-22. What we all should be.
4-26. A day dream.
2-11. A talon.
19-28. A pigeon.
F-7. Part of your head.
23-30. A river in Russia.
1-32. To govern.
33-34. An aromatic plant.
N-8. A fist.
24-31. To agree with.
3-12. Part of a ship.
20-29. One.
5-27. Exchanging.
9-25. Sunk in mud.
13-21. A boy.

Figure 1 : First crossword [19]

The first appearance of a crossword in a British publication was in *Pearson's Magazine* in February 1922, and the first *Times* crossword appeared on February 1st 1930. British puzzles quickly developed their own style, being considerably more difficult than the American variety. In particular the cryptic crossword became established and rapidly gained popularity. [18] Currently there exist four major crossword types: American, British, Swedish (or Scandinavian) and Japanese-style crosswords, which differ in grid construction, clue types and filling rules (with respect to the language). *Note:* the program made for this thesis is capable of constructing American, British and Japanese-style grids.

With the appearance of computers and the internet crossword became even more easily accessible, but on the other hand less popular. There are several reasons for such loss of popularity. First of all, the design of the puzzle itself is not very suitable for the computer screen, especially for the mobile devises. The whole grid with clues will not fit on the screen forcing user to scroll up and down numerous times, which is not a pleasant activity. Secondly,

internet offers a variety of other spare-time activities that attract people more than crossword solving. As a proof take a look at how many people play “Farm Frenzy” or “Angry Birds” and compare with a number of people solving crosswords. Thirdly, crossword is a difficult puzzle, that requires knowledge in different areas, thus scaring away even more people.

In conclusion, crossword puzzle is a good spare-time activity, which unfortunately is not that widespread now as it was in the past.

1.2 History of algorithms

An algorithm is a specific set of instructions for carrying out a procedure or solving a problem, usually with the requirement that the procedure terminates at some point. Specific algorithms sometimes also go by the name method, procedure, or technique. The word "algorithm" is a distortion of al-Khwārizmī, a Persian mathematician who wrote an influential treatise about algebraic methods. The process of applying an algorithm to an input to obtain an output is called a computation.

Until 1920 all algorithms were considered to be concrete and positive, thus no formal definition existed. It was given in 1936 by Alan Turing and Alonzo Church and now is known as Church-Turing thesis, which states that a function is algorithmically computable if and only if it is computable by a Turing machine (or by using λ-calculus, or by using recursive functions). As a side remark, Church-Turing thesis is actually a hypothesis, since it has not been formally proven, but now it is near-universe accepted. This thesis allowed scientists to determine whether there exists an algorithm for solving given task or not, and if it exists, show how it can be computed [10,17].

Concerning computer algorithms, the first such algorithm was written by Ada Lovelace in 1842. It was an algorithm for analytical engine to compute Bernoulli numbers. But at that point of time there was no way to confirm if this algorithm worked properly and if it worked at all. The major steps were taken after first computers appeared in late 1940s [16]. Over the next couple decades a lot of different algorithms, that are now widely used, were designed. For example, such algorithms known to anyone acquainted with computer science like: Kruskal's algorithm, Dijkstra's algorithm, quicksort, A* and many others. Although it might seem that with so many algorithms we can solve any problem, it is not so. There are tasks which ca not be solved algorithmically and tasks for which algorithm has not been found yet or its absences confirmed.

1.3 State of the problem

The main goal of this thesis is to create a program that allows constructing crosswords, using two different algorithms. Given a grid and a text file with words (dictionary), the program should search for suitable words from a dictionary to fill the grid. The program should be able to complete this task in two different ways, in this case using constraint satisfaction method (CSM) with greedy algorithm and simulated annealing.

This task is considered to be a NP-complete problem, it is both in the set of NP and NP-hard problems. The abbreviation NP stands for non-deterministic polynomial time. A formal definition [4] :

A decision problem C is NP-complete if:

1) C is in NP
2) Every problem in NP is reducible to C in polynomial time

The concept of NP-completeness was introduced in 1971 by Stephen Cook, though the term NP-complete did not appear in his article. The name "NP-complete" was popularized by Alfred Aho, John Hopcroft and Jeffrey Ullman in their textbook [8]. The solution to any NP-complete problem is an algorithm that runs in superpolynomial time and it is unknown, whether there exists a faster algorithm or not. But the certain techniques, such as approximation, randomization, restriction, parametrization or heuristic, can be applied to give a rise to substantially faster algorithms. However, it still remains unclear whether all problems in NP can be solved as fast as they can be verified (in polynomial time) or there are some problems that ca not be solved in polynomial time, but whose solution can be verified in this time. This problem is called P versus NP problem and considered to be one of the major unsolved problems in computer science [4].

Afore-mentioned algorithms were chosen mainly for educational purposes, since construction of the fastest algorithm is not a goal of this work. Along with other similar Artificial Intelligence problems, like N queens problem, map colouring and Sudoku solving (which is also NP-complete problem), crossword construction is a good example of simple, yet nontrivial task.

The choice of CSM with greedy algorithm is obvious. If there are no constraints, the program will simply try to fill each entry by placing up to all, and that means also the words that are of inappropriate length, words in vocabulary until it finds first suitable or runs out of words. For example, by putting constraints on words length and already filled letters, the construction time can be drastically reduced.

The simulated annealing was chosen with intention to show that the same problem can be solved in different ways and also to illustrate the difference in algorithm processing and its effectiveness. In addition, simulated annealing is quite similar to greedy algorithm, thus making their comparison a bit easier, but more interesting.

1.4 Similar problems

In this section some examples of Artificial Intelligence problems will be given, which can be solved in a similar way.

1.4.1 N queens problem or more common eight queens problem

This is a problem of placing eight (n) chess queens on 8x8 (nxn) chessboard, so that there are no two queens sharing the same row, column or diagonal. The eight queen puzzle was designed by the chess player Max Bezzel in a year 1848. In 1850 the first solution was provided by Franz Nauck, who also generalized the problem to n queens. Over the years different solutions were developed using different techniques, such as constraint programming, logic programming and genetic algorithms.[6]

The algorithm design for solving this problem is quite similar to the crossword construction algorithm. It can be done with naïve brute-force search, but will require large amount of time and resources. Also it will not give any results for bigger n. It can be refined by restricting each queen placement to a single row or column. Also the solution can be found by using other algorithm work result (n rook placement on nxn board) and putting an additional constraint (diagonal attack) on it.

1.4.2 Four colour map theorem

The theorem states that, given any plane separated on arbitrary number of contiguous regions,

called map, at most four colours are required to colour this map in such way, that no two adjacent (regions are adjacent if they share a common border) regions are of the same colour. The five colour version of this theorem was proven in 1890 by Heawood, whereas the four colour theorem itself was proven in 1976 by Kenneth Appel and Wolfgang Haken. This was the first major theorem, which was proven using a computer.[5]

From this theorem comes a very common exercise in algorithm design. Given an arbitrary map, colour it with four colours, such that non two adjacent regions are of the same colour. This task can be solved by using CSP and forward checking.

1.4.3 Sudoku

The logical puzzle, which goal is to place numbers from 1 to 9 into 9x9 grid in such way, that every row, column and 3x3 sub-grid contains all digits from 1 to 9. An initial grid typically has some digits written, which results in the unique solution. First Sudoku-like puzzles appeared in late 19th century, more precisely on November 19, 1892 the first such puzzle was published and on July 6th, 1895 the improved version of it. The modern version of Sudoku was first published in 1979, but popularization began only in 1986 from Japan and became worldwide in 2005.[1]

Sudoku can be solved by using a simple brute-force algorithm that undoubtedly takes a lot of time and resources. On the other hand, it can be solved with the help of constraint programming and backtracking algorithm or as an alternative to constraint an exact cover can be used.

2. Program and algorithms description

In this chapter detailed description of the program and algorithms implied will be given.

2.1 Program description

The program was written in Java, an object-oriented programming language. Java was chosen since it is one of the most popular programming languages at this point of time and is used in teaching object-oriented programming at University of Tartu. Also it allows designing considerably intuitive user interface, which is an important part of this program because of the grid construction.

The program is capable of constructing a crossword grid on a field of arbitrary nxm size. By pressing a “New” button a user can set these parameters in a pop-up window. After it is done, an empty field of a set size is created and a user can then make entries in it by clicking on the initial point and then on the end point. When created, entries are coloured grey, while the rest of the grid is black and can not be filled with any symbols. The program as well allows a user to save created grids in .cw format file and, as follows, load saved grids if it is needed. This option contributes to faster testing and eases-up the demonstration.

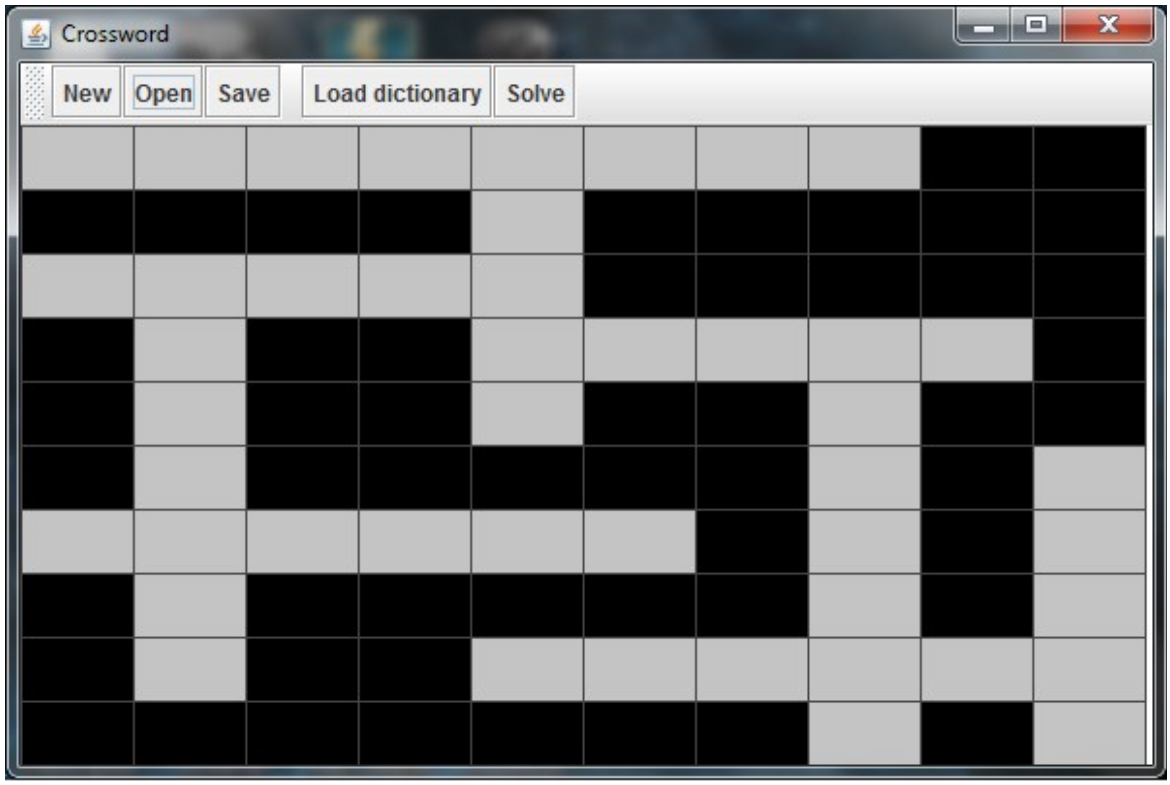


Figure 2: A program in work

After grid is created a user should press the dictionary button and choose a dictionary from which his grid will be filled with words. The program works with dictionaries in .txt format, where every word appears on a new line. After dictionary is chosen it is loaded in memory and ready to use. The size of dictionary is not limited, but user must keep in mind that more input data always leads to more time and system resources required to complete the task.

Once a grid is done and a dictionary is loaded a user should press the “Solve” button and choose a method with which the grid shall be filled. If his choice is CSM, then a user should

choose the corresponding entry in the first drop box. For simulated annealing a user should do exactly the same. Then a user should choose heuristic (if needed). After that the program proceeds to work. If it is possible to fill the given grid with words from a chosen dictionary, the program will end its work with a filled grid and no additional messages. Otherwise, a pop-up window will appear reporting that algorithm failed to fill the given grid, thus one must create a new grid or change a dictionary.

2.2 Algorithms description

In this section a detailed description of implemented algorithms will be given.

2.2.1 Constraint Satisfaction Method

Often referred to as constraint satisfaction problem (CSP). As it may be inferred from the name, this set of problems deals with constraints. These constraints are not different from those we encounter in a real world. Constraints surround us every day, such as temporal constraints (managing work and home life), or tangible constraints (making sure we do not go over budget), and we figure the ways to deal with them to varying success. If we are not satisfied with the result and run into problems, especially with solutions that may work, but due to our limited capacity to deal with a large amount of data, can not be resolved. This is an area where computers, and more specifically, constraint satisfaction problems (CSPs), are necessary.

Like most problems in artificial intelligence (AI), CSPs are solved through search. What makes CSPs unique, however, is the structure of the problem. Unlike other AI problems, there is a standard structure to CSPs that allows general search algorithms using heuristics (with knowledge about the structure of the problem and not necessarily domain-specific knowledge) to be implemented for any CSP. In addition to this, all CSPs are also commutative - they can be searched in any order and still give the same result. These special and defining characteristics make CSPs both interesting and worthwhile to study.

CSPs are very useful, when dealing with temporal or combinatorial problem, solving logical puzzles, among other things. Further some examples of CSPs application in various areas are given:

- Operations research (scheduling, timetabling)
- Bioinformatics (DNA sequencing)
- Electrical engineering (circuit layout)
- Telecommunications (CTVR @ 4C)
- Hubbell telescope/Satellite scheduling

Generally speaking, CSPs are a rather recent formulation. There is not extensive published literature on the subject, but they are widely studied and their applications will continue to increase.

The formal definition of CSP includes variables, their domains and constraints. Assume that we have a set of variables $[X_1, X_2, \dots X_n]$. Each variable has a domain $[D_1, D_2, \dots D_n]$, such that all variables X_i have a value in their respective domain D_i. There is also a set of constraints $[C_1, C_2, \dots C_n]$, such that constraint C_i restricts (puts a constraint on) the possible

values in the domains of subset of some variables. A solution to a CSP is an assignment of every variable with some value in its domain, such that every constraint is satisfied. Therefore, each assignment (a state change or step in a search) of a value to a variable must be consistent: it must not violate any of the constraints [11].

As in any AI search problem, there can be multiple solutions (or none). To address this, a CSP may have a preference of one solution over another using some preference constraints (as opposed to all absolute constraints), want all solutions, or the optimal solution, given by an objective function. Optimizing a CSP model will be explained later in the thesis.

This explanation of constraint programming will only touch on problems that have finite domain variables. This means that the domains are a finite set of integers, as opposed to a real-valued domain that would include an infinite number of real-values between two bounds.

As mentioned, the structure of the CSP is the most important part of it since the same algorithms can be used to search any CSP. Since we know that the structure is standard across all CSPs, we can take a look at heuristics that are able to operate on all different types of problems. However, this does not mean that all algorithms are equally tractable and efficient on all sorts of problems. Currently, the decision of the use of an algorithm for a certain problem is determined empirically.

A constraint is considered as *n*-ary if it involves *n* variables. So if a constraint affects just a single variable, it is considered as unary. Unary constraints can be dealt with as a preprocessing step. Constraints that involve two variables are binary constraints and are of particular interest for two reasons. The first reason is that they can be modeled as a constraint graph, where the nodes of the graph represent the variables and an edge connects two nodes if a constraint exists between the two variables. The second reason is that a constraint of higher arity (the number of variables involved in a constraint) can always be reduced to a set of binary constraints. However, it does not mean that this is always a good idea. In some cases, the number of binary constraints for a problem can be exponential, thus creating an intractable model. More complex constraints, with arity > 2, are called global constraints. A simple example of a global constraint is the *Alldifferent* constraint; this constraint forces all the variables it touches to have different values (Note: It is easy to see how this particular global constraint could be decomposed into many "not equal" binary constraints.) [11].

Deciding on the variables to be included in a model of your problem is usually not too difficult: there are the obvious variables that must be assigned values for a solution to exist (decision variables) and variables that help to make the problem more efficient or contribute to some objective function. While tricks can be used (such as was earlier when the queens were represented as rows) to increase performance, they are just that.

A part of any search algorithm is choosing a variable that has not been instantiated yet and assigning it a value from its domain. There are both static and dynamic variable ordering heuristics available to decide "*how"* to choose this next variable. One such heuristic is MRV (minimum-remaining values), which comes from the fail-first principle. The MRV heuristic selects from the set of unassigned variables the next variable with the fewest remaining values in its domain [11]. Essentially this allows us to discover a dead end sooner than we would have and as a result reduce the overall size of our search tree. This heuristic becomes much more useful when dealing with a problem with noticeable variances in the cardinality of domains, both during the preprocessing and (dynamically) as the search progresses.

Another heuristic for variable ordering, often used as a tie-breaker is the degree heuristic. This heuristic attempts to choose the unassigned variable that is involved in the most constraints with other unassigned variables. This reduces the number of children of each node

in the search tree by decreasing the domain sizes of other variables.

After we have a variable, we must assign it a value. The way in which we choose values, or value ordering, is also important because we want to branch as often as possible towards a solution (though value ordering is a waste of time if we are looking for all solutions). The most popular heuristic for choosing a value is LCV, or least-constraining value. The idea is to choose the value that would eliminate the fewest values in the domains of other variables and thus hopefully steer the search away from a dead end [11]. In other words, it leaves the most possible number of values to by assigned further on.

Searching a CSP involves firstly choosing a variable and then assigning a value to it. In a search tree, each node is a variable and branches leading away from that node are the different values that can be assigned to that variable. Therefore, a CSP with n variables will generate a search tree of depth n. Successors are generated for only the current variable and the state of the problem is appended each time the search branches.

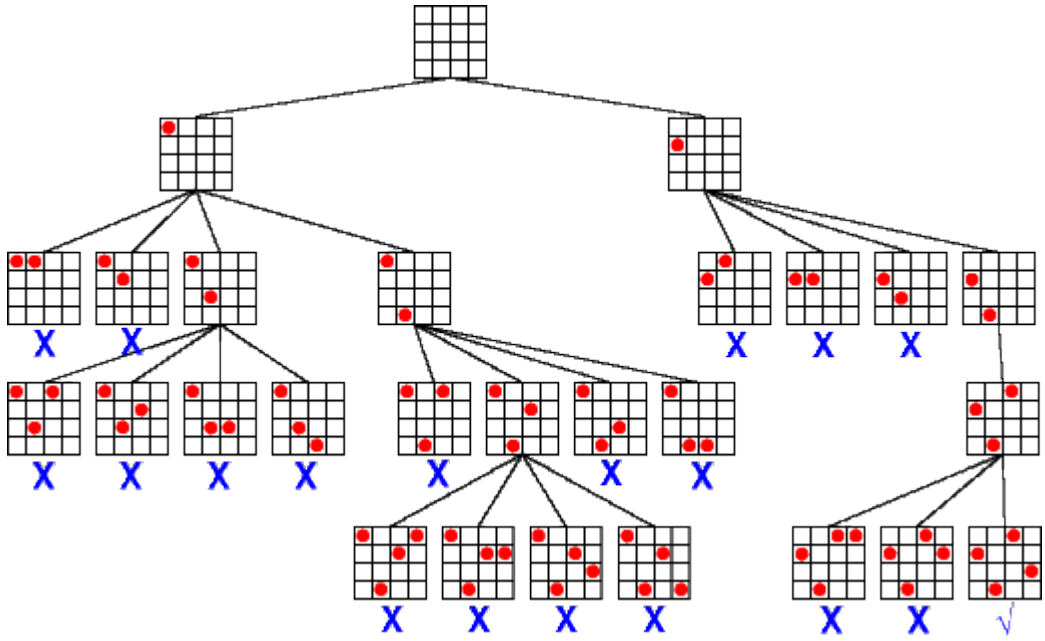


Figure 3: An example of backtracking for 4-queens problem [14]

If we consider a simple depth-first search on a CSP (Figure 3), we realize that because of the constraints we have imposed, at some point during our search we may be unable to instantiate a variable because its domain is empty. In the case that we arrive at a node where the goal test returns false (there are still unassigned variables) and there are no branches leading away from that node, we must go backward. This is called backtracking and it is the most basic of searches for CSPs. A variable is assigned a value and then the consistency of that assignment is checked. If the assignment is not consistent with the state of the problem, another value is assigned. When a consistent value is found, another variable is chosen and this is repeated. If all values in the domain of a variable are inconsistent, the algorithm will backtrack to the previous assignment and assign it a new value [11].

When backtracking search chooses a value for a variable, it checks to see whether that assignment is consistent with the constraints on given problem. It is clearly not very efficient. Consider a simple graph-colouring problem. If there is an edge between two nodes, then once we assign a colour to one of these nodes, we know that choosing the same colour for the other will not be consistent; therefore, we must temporarily remove the values from the domains of

yet unassigned variables that are not consistent with the current problem state with the new assignment.

The forward checking algorithm does just this. Every time an assignment is made to a variable, all other variables connected to the variable (that is currently being instantiated) by constraints are retrieved and the values in their domains which are inconsistent with the current assignment are temporarily removed [11] (Figure 4). In this way the domain of a variable can become empty and another value must be chosen for the current assignment. If there are no values left with which to assign the current variable, the search may need to backtrack, in which case those values that were temporarily removed as a result of the original assignment are reinstated to their respective domains. Forward checking is able to predict, in a sense, what assignments will lead to a failure and can act accordingly by pruning branches. It will, of course, encounter these inconsistencies much sooner than backtracking, especially when used in conjunction with the fail-first heuristic described earlier.

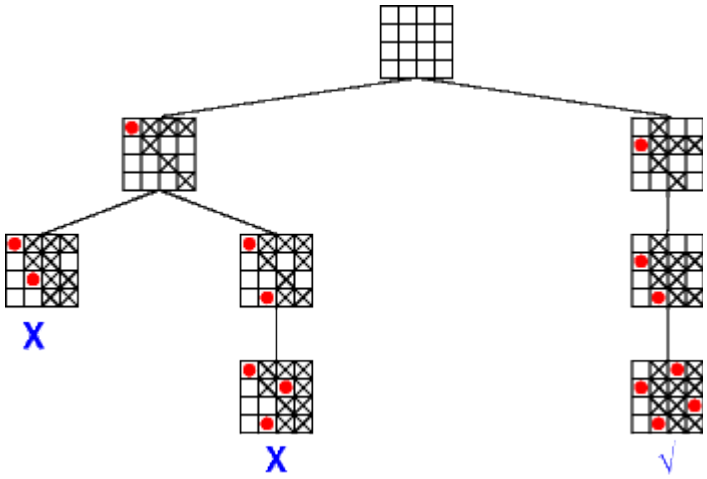


Figure 4: An example of forward checking for 4-queens problem [14]

Local search for AI problems involves making a complete assignment of variables and then switching (flipping) a value for a variable and checking to see if we have found a solution. It is not different in constraint programming. Assignments are made to all our variables, but these assignments will not be consistent with the constraints on the problem. In local search, each node in the search tree is a complete assignment, with branches involving flipping different variables within the complete assignment until a solution is found. Local search works very well for some types of CSPs. The most popular heuristic for local search in CP is min-conflicts. When min-conflicts flips one of the variables in the assignment, it will choose a value for that variable that results to the minimum number of conflicts with other assignments. Thus we have some idea of progress in our local search. The min-conflicts algorithm pseudocode presented below [11] :

```
algorithm MIN-CONFLICTS
     input: csp, a constraint satisfaction problem
            max_steps,the number of steps allowed before giving up
            current_state, an initial assignment of values for the variables
in the csp
     output: a solution set of values for the variable or failure
   for i=1 to max_steps do
       if current_state is a solution of csp then return current_state
          var <-- a randomly chosen variable from the set of conflicted
variables CONFLICTED[csp]
```

```
                    value <-- the value v for var that minimizes
CONFLICTS(var,v,current,csp)
        set var = value in current_state
    return failure
```

Forward checking utilizes a basic notion of consistency: an assignment to a variable is consistent with other assignments given a set of constraints. *K-consistency* is a term that defines the extent to which constraints are propagated. By definition, a CSP is *K-consistent* if for any subset of k - 1 variables in the problem, and for any consistent assignment to those variables, a consistent value can be assigned to any Kth variable.

In addition to a problem being *K-consistent*, it can also be strongly *K-consistent*, which means it is consistent for K and all weaker consistencies less than K. The benefit of a strongly *K-consistent* problem is that we will never have to backtrack. As with most things CSP, determining the correct level of consistency checking for a given problem is done empirically. Below, popular (node, arc and path) consistencies are discussed [11].

Node consistency is the weakest consistency check and simply assures that each variable is consistent with itself; if a variable is assigned a value, the value must be in that variable domain[11].

Arc consistency (AC, 2-consistency) is the most popular consistency and can be used either as a preprocessing step, or dynamically as a part of the maintaining arc consistency (MAC) algorithm. The simple definition of arc consistency is: given a constraint C_{XY} between two variables X and Y, for any value of X, there is a consistent value that can be chosen for Y such that C_{XY} is satisfied, and vice versa. Thus, unlike forward checking, arc consistency is directed and is checked in both directions for two connected variables. This makes it stronger than forward checking. Also note than any k-consistent problem can be reduced to a set of arc consistencies.

When applied as a preprocessing step, arc consistency removes all values from domains that are inconsistent with one another. If it is applied dynamically as MAC, the same algorithm that is used to check AC for preprocessing is applied after every variable instantiation during the search.

One of the most popular algorithms for arc consistency is AC-3 (Arc Consistency Algorithm #3), developed by Alan Mackworth in 1977. This algorithm has the worst-time complexity of $O(ad^3)$ and space complexity of $O(a)$, where a is the number of arcs and d is the size of the largest domain [11].

The easiest way to think about path consistency is to consider a triangle, with three points labeled a, b, and c where edge(a, c) is not a solid line. It represents a problem where there are constraints between a and b, and b and c. Path consistency considers triples of variables, so that while a and c are not explicitly constrained, there is a constraint induced on them through the transitive nature of their constraints involving b [11]. Thus our triangle is an easy representation of this relationship. If the constraints are that $a > b$ and $b > c$, it is clear that there is an implicit constraint between a and c, such that $a > c$. 3-consistency, though obviously stronger than arc consistency, is not generally used. While arc consistency checks pairs of variables for consistency, path consistency must check all triples of variables for a large problem, it is easy to see that the number of combinations is potentially huge. In the worst-case time, the complexity is $O(d^3, n^3)$.

Studying the phase transition for a type of problem allows us to locate where the

fundamentally hard CSP and SAT problems are. In the context of a CSP, constraint tightness refers to the number of instances where, given a constraint between two variables X and Y, the pair of values for X and Y are inconsistent. A phase transition is a phenomenon that is observed by considering the graph of search effort against constraint tightness for many instances of a problem as the constraint tightness of the problem increases from 0 to 1. As this happens, the CSP will move from the part of the problem space that is underconstrained and where there are many solutions to a space that is overconstrained and where problems are not satisfiable.

The transition between the soluble and insoluble regions is referred to as the mushy region; a term coined by B Smith and is populated both with problems that have a solution and those that do not: it is in this region that the peak search effort is spent trying to find a solution that exists with low probability. Because of this, phase transitions are important in the study of NP-complete problems and can give some understanding, whether a problem is likely to be easy or difficult. Phase transitions are not algorithm-specific.[12]

While modeling a CSP, it is common that one may encounter symmetry. Symmetry is defined as an assignment, which is equivalent to another assignment; in other words, the assignments are interchangeable [11]. If you consider these instances, it is easy to see that if one of these assignments is consistent, then they all are. Hence it is possible to have classes of equivalent solutions where different symmetrically equivalent assignments can be interchanged and one can be assured a solution can be found for this "different" problem.

It is important to take notice of symmetries because they can be used to shorten search time (by not searching symmetrically equivalent branches). "Breaking" symmetries or, as this is called, can often be the difference in whether a problem is tractable or not. In order to exploit this feature, additional constraints (called symmetry-breaking constraints [11]) must be added to the model. These constraints (which are model-specific) must make sure that if one assignment does not work, all symmetrically equivalent sets of assignments are automatically ruled out (Figure 5).

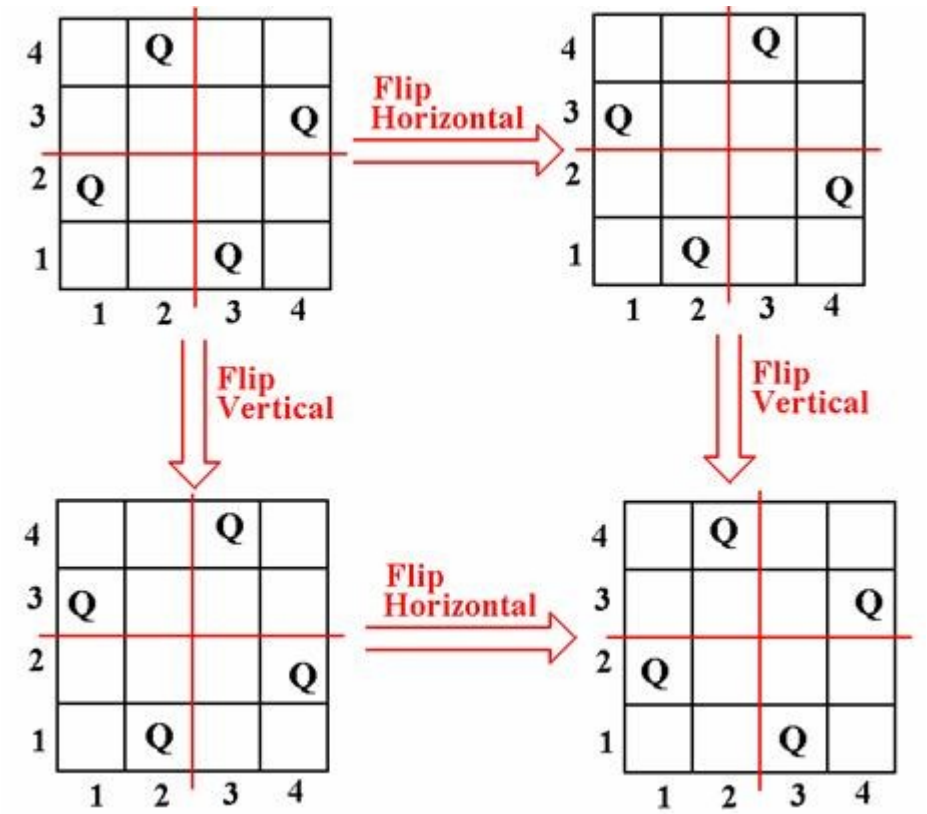


Figure 5: Example of symmetry in 4-queens problem [15]

Problems which must be optimized for a given constraint, represented by an objective function, are solved in the same way as other CSPs. When a solution is encountered, it will be associated with it some "ranking" value for the objective function. The search continues to find all solutions and then chooses the solution with the *most* optimal value. It can be said that for every solution found, if for some objective function the value is more optimal than the previous one found, that solution is saved until all solutions have been found. This idea is also referred to preferred constraints.

The most common objective functions are minimize and maximize. These functions try to minimize a given constraint (or variable) or maximize it. An example of a constraint is a linear equation between two variables, x and y. The constraint may be that $x + y > 100$, but we want to maximize this constraint for x or y, so that we find the largest value(s), which satisfy the all the constraints of the problem.

Scheduling problems are obvious places where objective functions will pop-up. As an example, we could look at the person who works at a grocery store that is open from 7am to 2am. Certainly, that person would not want to work one night from 6pm-2am and then the next morning from 7am-3pm. Using an objective function to create a schedule where all shifts are covered by employees, but minimizing the number of consecutive night-morning shifts would certainly be very useful.

Branch and bound is an optimizing method for solving CSPs that are too large (whether it be in the number of variables and constraints or the complexity of the constraints) to search all the solutions. The idea borrows from that of partitioning; the first solution to the problem is found (and along with it, some evaluation) and a constraint is added on the objective function to form a "subproblem" of the original so that the subproblem will be searched for the first solution found again and repeated until some optimal solution for a minimizing/maximizing

function is found or there are no more solutions left. The creation of a new subproblem from the original is the branching part of the algorithm, while the bounding is the use of the evaluation for a solution to bound the new constraint. While this approach to large problems that need to be optimized is practical for real constraints like time and space, it is also more efficient. Once you have a bound, the search can stop before finding a solution if it knows that the evaluation will not be as low as one previously found.

2.2.2 Greedy algorithm

A greedy algorithm is an algorithm that follows the problem solving heuristic of making a locally optimal choice at each stage with hope of finding a global optimum. In many problems, a greedy strategy does not, in general, produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions, which then can be used to find a globally optimal solution.
In general, greedy algorithms have five components [2]:

1. A candidate set, from which a solution is created
2. A selection function, which chooses the best candidate to be added to the solution
3. A feasibility function, that is used to determine better candidates
4. An objective function, which assigns a value to a solution, or a partial solution
5. A solution function, which will indicate when we have discovered a complete solution

Greedy algorithms produce good solutions for the problems with following properties:

1. There is no need to reconsider algorithms decision
2. A problem has an optimal structure (an optimal solution to the problem contains optimal solutions to the sub-problems)

2.2.3 Implementation of CSP for crossword construction

CSP implementation is not that hard. Constraints are defined as follows: to fill the entry word must be of the same length, if entry contains any letters at some positions, then the candidate word must also contain these letters at the same positions. Thus, the search space, and as a result time, for finding every candidate word is reduced. As can be noticed there are no complex constraints defined. Generally, defining constrains for a problem is intuitive.

The search algorithm itself applies greedy heuristic search. Heuristics defined in this method are consonant count and pairwise distance. Consonant counting heuristic defines candidate word cost by counting the number of consonants in it. The more consonants in a candidate word, the higher its cost (following code illustrates this heuristic).

```
for(int i = 0; i < ls.length(); ++i) {
   char c = ls.charAt(i);

   if(c != 'a' && c != 'e' && c != 'i' && c != 'o' && c != 'u' && c != 'y')
      count++;
}

return count;
```

Here *ls* stands for a candidate word and count is the cost of this word, which will be used by algorithm to make a decision.

The pairwise distance heuristic uses the principle of hamming distance. It counts the numeric difference between the first word's letters and the second ones. This number is then added to the considered word cost. The operation continues until all candidates are considered. The following code shows this heuristic implementation :

```
public static int getHammingDistance(String s1, String s2) {
    int d = Math.abs(s1.length() - s2.length());
    int commonLength = Math.min(s1.length(), s2.length());

    for(int i = 0; i < commonLength; ++i) {
                        d += Math.abs(Character.getNumericValue(s1.charAt(i)) -
Character.getNumericValue(s2.charAt(i)));
    }

    return d;
}

@Override
public int getCost(String s) {
    int cost = 0;
    for(String m : matched) {
        cost += getHammingDistance(s, m);
    }

    return cost;
}
```

The first part of this code computes the hamming distance between two words. First of all, it makes sure that words of different length were not considered as words of the same length (e.g. clear and clearly). Then it proceeds to count the hamming distance as it was described earlier. The second part computes the *cost* of the considered word, *s* responds to this word in code.

Based on the chosen heuristic, greedy algorithm then takes an optimal solution, the one with highest cost, at each step (in this case a step is filling one entry). As a result, this algorithm does not guarantee that the found solution is optimal (that there is no better solution).

2.2.4 Simulated annealing

Simulated annealing is a technique for combinatorial optimization problems, such as minimizing functions of very many variables. Because many real-world design problems can be cast in the form of such optimization problems, there is intense interest in general techniques for their solution. Simulated annealing is one such technique which has recently appeared, it was introduced independently by Kirkpatrick, Gellat and Vecchi in 1983, by

Cerny in 1985 and is an adaptation of the Metropolis-Hastings algorithm, a Monte Carlo method to generate a sample states of a thermodynamic system, invented by Rosenbluth and published by Metropolis in 1953, with an unusual pedigree: it is motivated by an analogy to the statistical mechanics of annealing solids.[7,9,13] To understand why such a physics problem is of interest, consider how to coerce a solid into a low energy state. A low energy state usually means highly ordered state, such as a crystal lattice; a relevant example here is the need to grow silicon in the form of highly ordered, defect-free crystals for use in semiconductor manufacturing. To accomplish this, the material is *annealed*: heated to a temperature that permits many atomic rearrangements, and then cooled carefully, slowly, until the material freezes into a good crystal. Simulated annealing techniques use an analogous set of "controlled cooling" operations for nonphysical optimization problems, in effect transforming a poor, unordered solution into highly optimized, desirable solution. Thus, simulated annealing offers an appealing physical analogy for the solution of optimization problems, and more importantly, the potential to reshape mathematical insights from the domain of physics into insights for real optimization problems.

Interest in such solution techniques is intense because few important combinatorial optimization problems can be solved exactly in a reasonable time. Many optimization problems arising in practice are NP-complete: all known techniques for obtaining an exact solution require an exponentially increasing number of steps as the problems become larger. Hence, emphasis has been directed toward heuristic techniques for solving these important problems. The difference between a heuristic and an algorithm is that a heuristic is not guaranteed to get the optimum answer: heuristics are designed to give an acceptable answer for typical problems in a reasonable amount of time. In practice, however, the terms *algorithm* and *heuristic* are often used interchangeably. Moreover, simulated annealing is not an algorithm in the sense that it prescribes a mechanical sequence of computations to solve a specific problem, for example, in the sense that Gaussian elimination is an algorithm for matrix inversion. Rather, annealing is a strategy or style for solving combinatorial optimization problems. Specifically, simulated annealing is a heuristic solution strategy applicable to a wide variety of optimization problems.

The main idea of simulated annealing is: given an arbitrary initial state s consider at random its neighbour state s', if a new state is better than the initial, then go there, otherwise use probability function P to decide, whether go there or not. This operation is repeated until an acceptable state is reached or algorithm used allowed number of steps. Eventually, a probability of going to a worse state decreases (similar to temperature cooling).

The neighbours of a state are new states of the problem that are produced by changing a given state in some well-defined way. The way in which the states are changed is called "move". Moves usually result in minimal changes of the last state in order to help the algorithm keep the better parts of the solution. Simple heuristics proceed by taking the best neighbour after the best neighbour and stop when they have reached a solution which has no neighbours that are better solutions (e.g. greedy algorithm). The problem of this approach is that the neighbours of a solution found this way may not have any better states among them, however, this does not mean that there is no better solution. That is why the best solution found this way is called "local optimum",while actual best solution – "global optimum". Compared to that, simulated annealing accepts worse solutions as well (with some probability $P()$), in order to avoid getting stuck in "local optimum". As a result, if the algorithm is run for an infinite amount of time, the global optimum will be found.

A probability of making a transition (move) from the current state s to a new state s' is specified by an acceptance probability function $P(e, e', T)$, where $e=E(s)$ and $e=E(s')$ are the energies of the states, and T is a global time variable called the temperature. States with

smaller energy are preferred. The probability function P must be positive even when $e'>e$. This feature prevents thr method from becoming stuck at a local minimum that is worse than the global one.

When T tends to zero, the probability P must tend to zero if $e'>e$ and to a positive value otherwise. For relatively small values of T, the method will favor moves that go "downhill" and avoid those that go "uphill". With $T=0$ the method reduces to the greedy algorithm, which makes only the downhill transitions. Originally, the probability P was equal to 1 when $e'<e$, the algorithm moved downhill when it found the way to do so, despite the temperature. This condition is not essential for the method to work, but is still used in many descriptions and applications of simulated annealing.

The function P is usually chosen in such way that the probability of accepting a worse move decreases while the difference $e'-e$ increases, so that no large moves uphill are made. This is an alternative requirement, to the previous one, for this method to work.

The temperature T plays a major role in defining probability P, precisely, the higher is the value of T, the higher is the probability P of changing a state to a worse one and other way. Temperature initially starts at high values, gradually decreasing with each algorithm step to zero. In this way, the system is expected to wander initially towards a broad region of the search space containing good solutions, ignoring small features of the energy function; then drift towards low-energy regions that become narrower and narrower; and finally move downhill.

The following pseudocode illustrates simulated annealing [3] :

```
s ← s0; e ← E(s)                          // Initial state, energy.
sbest ← s; ebest ← e                      // Initial "best" solution
k ← 0                                     // Energy evaluation count.
while k < kmax and e > emax                    // While time left & not good
enough:
  T ← temperature(k/kmax)                 // Temperature calculation.
  snew ← neighbour(s)                     // Pick some neighbour.
  enew ← E(snew)                          // Compute its energy.
  if P(e, enew, T) > random() then        // Should we move to it?
    s ← snew; e ← enew                    // Yes, change state.
  if enew < ebest then                    // Is this a new best?
     sbest ← snew; ebest ← enew              // Save 'new neighbour' to 'best
found'.
  k ← k + 1                               // One more evaluation done
return sbest                              // Return the best solution found.
```

In order to apply the simulated annealing to a specific problem, one must fix the following parameters: the state space, the energy function $E()$, the candidate generator *neighbour*(), the acceptance probability function $P()$, initial temperature and annealing schedule *temp*(). Each of those parameters has a significant influence on effectiveness of the method. Unfortunately, there are no universal choices for those parameters. One must set them empirically, using some general guidelines.

Simulated annealing may be modeled as a random walk on a search graph, whose vertices are all possible states and edges are the candidate moves. An essential requirement is that the diameter of the search graph has to be small. For each edge (s,s') of the search graph a transition probability must be defined (probability that the SA algorithm will move form the current state s to s'). This probability depends on the current temperature, order in which candidate moves are generated and acceptance probability function $P()$. The specification of *neighbour*(), $P()$, and *temp*() is partially redundant. In practice, it is common to use the same

acceptance function *P*() for many problems, and adjust the other two functions according to the specific problem.

2.2.5 Simulated annealing implementation

Simulated annealing is implied as it was described earlier. It selects a random neighbour of the current state. If it is a better solution, algorithm takes it as a current solution, otherwise it accepts it with some probability defined by the following function :

Math.exp((cost - currentSolutionCost) / t);

Where cost responds to the considered neighbour's cost and *t* is temperature. Both current and candidate solutions are evaluated by the TargeFunction, which implies pairwise distance heuristics. New candidates are found using chosen heuristic function, consonant or hamming distance heuristic (both described in CSP implementation). Eventually temperature decreases as well as the probability of accepting bad candidates. Temperature schedule is defined by the following code :

```
public boolean done() {
    return next < minimum;
}
public double nextTemperature() {
    double result = next;

    next *= alpha;

    return result;
}
```

Where alpha is a temperature decreaser and minimum is the temperature, reaching which an algorithm stops its work with solution it got to this point (if it got any). The danger here is that a temperature schedule is not defined well enough, the algorithm may not find an optimal solution in reasonable time (high initial temperature or slow cooling) or it may stuck in local optimum (low initial temperature or cooling too fast). As it was mentioned earlier, these parameters can only be determined empirically.

3. Analysis

3.1 Test cases

To determine algorithm effectiveness, one should run tests with different input data. In this case, input data is represented by grids and dictionaries of different size. For this thesis three different grids of size 10x10 (test2, test3, test4) and one grid of size 5x13 (test1) were made. Test2 grid has ten words, which have double interceptions. Test3 grid has six isolated words. Test4 grid has nine words and only single interceptions (one word intercepts only with one another). Test1 grid has nine words and is an example of a “non-standard” grid.

Three dictionaries were used in test, one containing 190 words (small), second – 999 (mid) and last one containing 58112 words (big). Grids were built in such way, that allows to show the difference between algorithm operation time and trace some dependencies in their work efficiency. Dictionaries were chosen to show how algorithm operation time depends on input data amount.

The tested grids and dictionaries are provided with program on the DVD.

3.2 Test results and analysis

The results of tests are provided in the table below. They illustrate operation time of program using different combinations of input date (grids, dictionaries), algorithms (CSP with greedy search, simulated annealing) and heuristics (consonant count, pairwise distance). The results are provided in seconds.

Algorithm/Grid	test 1	test 1	test 1	test 2	test 2	test 2	test 3	test 3	test 3	test 4	test 4	test 4
CSP1	0,8	0,3	61,2	0,4	8,2	215,7	0,06	0,07	3,6	0,3	7,8	1
CSP2	0,08	31,8	0,5	1,8	*	*	0,07	0,1	3,7	0,08	90	1,2
SA1	1,3	0,8	62,2	1	8,1	232,3	0,5	0,5	32,6	0,4	7,9	3
SA2	0,3	32,2	3,2	2,1	*	*	0,5	0,5	31,7	0,3	90,5	4,4
Dictionaries	small	mid	big	small	mid	big	small	mid	big	small	mid	big

Table 1: Test results (in seconds)

In this table CSP1 responds to CSP with greedy algorithm using consonant count heuristic, CSP2 is the same method, but with pairwise difference heuristic, SA1 is simulated annealing with consonant count heuristic and SA2 – same method with pairwise difference heuristic (the detailed description of algorithms and heuristics is given is sections 2.2.3 and 2.2.5). * means that program could not find a solution in reasonable amount of time.

As one may see, given the same input parameters, simulated annealing method takes more time to find a solution, than CSP with greedy algorithm. This may be explained by the fact that SA has to perform additional actions, such as probability and temperature computations.

As it was expected, with more quantity of input data both algorithms operation time increases, but besides the input size, there are other parameters that have impact on algorithms operation time. These parameters are a grid structure (its size also impacts the operation time, but it is similar to the dictionary size), a dictionary structure and its content.

A grid structure increases operation time in the following way: the more interceptions are there, especially multi-interceptions (one word intercepts with two or more other words), the more time it will take to fill it,and vice versa. Also, SA works slower than greedy algorithm, with disconnected grids. That might be explained by the fact, that greedy algorithm just puts the first suitable word into entry, while SA tries to find an optimal solution for each entry.

A dictionary structure, precisely how words are sorted, also impacts the operation time. Time will increase, if the words considered by algorithm to have a highest cost are at the end of the dictionary, thus making the algorithm to consider all possible candidates, and the other way around.

A dictionary content will increase algorithms operation time if there is a lot of words of the same length, in addition, if they have the same number of consonants (for consonant count heuristic) or are quite of similar structure (for pairwise difference heuristic), the operation time will increase even more, thus making algorithms to consider large numbers of candidates, and vice versa.

4. Future plans

The program at its current state is a simple demonstration of possible solutions for crossword construction. There are a lot of improvements to be done and new features to add. First of all, optimize and improve algorithm, experiment with simulated annealing probability and temperature, add a dictionary sort with respect to each heuristic. Then improve the user interface, in particular grid part, so that it looks like a common crossword, published on paper. Also, for better understanding of algorithms work, visualization can be done, precisely; show how the algorithm is choosing the candidate words from a dictionary and then is placing them into the grid.

5. Conclusion

As a result of this thesis, a simple demonstrative program, that constructs crosswords using CSP with greedy algorithm and SA, was created. Efficiency of both algorithms was tested, compared and analysed. Conclusions about what might influence their operation time were made. Detailed algoritm descriptions along with some examples were provided. All together, it makes this thesis acceptable as a good educational material for Artificial Intelligence I course.

6. Addendum

All additional materials are provided on a DVD.

7. Ristsõna koostamine kasutades kitsenduste rahuldamist ja libalõõmutamist

Selle töö eesmärk on luua programm, mis koostab ristsõnu, kasutades kahte meetodit: kitsenduste rahuldamist (KR) ahne algoritmiga ja libalõõmutamist (LL), ning võrrelda nende meetodite efektiivsust. Tööd hakatakse kasutama õppematerjalina aine Tehisintellekt I õpetamisel.

Ristsõna koostamine on üks tehisintellekti probleemidest, mis kuulub NP-täielike klassi. Seega hea lahenduse leidmine nõuab palju ressursse ja aega. Aga eksisteerivad meetodid, mis võimaldavad lahenduse leidmise aega vähendada. Nende hulgas on ka KR ja LL.

KR kasutades seatakse antud ülesandele kitsendusi, mis teevad lahendamise lihtsamaks.
Ristsõna koostamisel kehtivad järgmised kitsendused:
1.Sõna ei saa olla lühem, kui ruutude järjend, kuhu seda pannakse.
2.Sõna ei saa olla pikem, kui ruutude järjend, kuhu seda pannakse.
3.Kui järjendis on mõned tähed juba olemas, siis sõna, mis pannakse sellesse järjendisse, peab neid tähti sisaldama täpselt nendel samadel positsioonidel ja ei saa sisaldada mingeid teisi tähti nendel positsioonidel.

Kui sõna rahuldab neid tingimusi, siis teda võetakse vastu ning ahne algoritm otsustab, kasutades heuristilist funktsiooni, kas see sõna on parim lahendus selles olukorras.Niiviisi püüab programm lõpliku sammude hulgaga optimaalse lahenduseni jõuda.

LL töötab nii: antud on suvaline algseisund *s,* leida tema naaberseisund *s′*, kui uus seisund on jooksvast seisundist parem, siis valida see, aga kui leitud seisund on jooksvast seisundist halvem, siis kasutada tõenäosus funktsiooni *P*, et otsustada, kas valida seda seisundit või mitte. Sellist operatsiooni korratakse kuni rahuldav lahendus on leitud või algoritm on juba teinud lubatud arvu samme. Tõenäosus, et algoritm valib uueks seisundiks halvema seisundi väheneb aja jooksul (kooskõlas nn temperatuuri alanemisega).

Meetodeid on testitud ja võrreldid, kasutades erinevaid heuristikuid.
Programm on kirjutatud keeles Java ja lisatud tööle DVD-l.

8. References

1.Christian Boyer “Sudoku's French ancestors”, Pour la Science, 2007.

2.T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, MIT, 2009.

3. S. Edelkamp, S. Schrödl, Heuristic search theory and application, Elsevier, 2012.

4. M.R. Garey, D.S. Johnson Computers and Intractability: A Guide to the Theory of NP-completeness, W.H. Freeman & Company, 1979.

5. P.J. Heawood "Map-Colour Theorem", Quarterly Journal of Mathematics, Oxford 24: 332–338, 1890.

6. E.J. Hoffman, J.C. Loessi, R.C. Moore Construction for the Solution of the m Queens Problem. Mathematics Magazine, Vol. XX (1969), p. 66–72.

7. S. Kirkpatrick, C. D. Gelatt Jr., M. P. Vecchi, (1983). Optimization by simulated annealing. Science, 220, p. 671-680

8. Donald Knuth, T. Larrabee, P.M. Roberts, Mathematical Writing § 25, 1989.

9. N. Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller,Edward Teller, (1953). "Equation of State Calculations by Fast Computing Machines". The Journal of Chemical Physics **21** (6): 1087

10. R. Prank. Matemaatiline loogika ja algoritmiteooria. TÜ kirjastus, 2004.

11. S. Russell, P. Norvig Artificial Intelligence A modern approach 3rd edition, Pearson Education , 2009.

12. B. Smith. “Phase Transition and the Mushy Region in Constraint Satisfaction Problems.” Proceedings of ECAI-94, pages 100-104, 1994.

13. V .Černý, (1985). "Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm". Journal of Optimization Theory and Applications **45**: 41–51.

14. Constraint Propagation.

http://ktiml.mff.cuni.cz/~bartak/constraints/propagation.html

15. The N-Queens Problem.

http://csc.columbusstate.edu/bosworth/SearchProblems/N_Queens.htm

16. Ada Lovelace, World's First Computer Programmer, Celebrated With Google Doodle

http://www.huffingtonpost.com/2012/12/10/google-doodle-ada-lovelace_n_2270668.html

17. Turing Machines.

http://plato.stanford.edu/archives/win2012/entries/turing-machine/

18. Inventor of the Week Archive.

http://web.mit.edu/invent/iow/crosswordhome.html

19. Fun's Word-Cross Puzzle

http://upload.wikimedia.org/wikipedia/commons/3/32/First_crossword.png

All links were last checked on 13.05.2013

9. Non-exclusive licence to reproduce thesis and make thesis public

I Pavel Tomozov

(date of birth: 12.02.1990),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

Crossword Construction using CSP and SA.

supervised by Mare Koit,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu , **13.05.2013**