Automated generation and analysis of logical puzzles

The flats puzzle

T.K. Cocx

2004

Master thesis, Universiteit Leiden

CONTENTS

ABSTRACT

In the Netherlands, logical puzzles are published in puzzle booklets for leisure purposes. Because of the logical nature of these puzzles it may be possible for computers to play a role in building, verifying, generating and analysis of these publishable puzzles. A puzzle lifecycle is defined to identify the places where computer intervention might be possible. Puzzelsport, the leading publisher of logical puzzles in the Netherlands, mainly creates puzzles by hand. When they do use programming to generate puzzles they usually take a rule based approach, but the effectiveness, extra possibilities, possible speed up and peculiarities of this method were never researched. In the scientific world little attention has gone to publishable puzzles, but what has been done already has not lead to complete, usable results. In order to investigate constraints, difficulties and possibilities of computer usage in this field, the flats puzzle is used as a test case. The flats puzzle handles around the building of flats when one only has the view from the different sides at his disposal. The test case is used to test different approaches to different kinds of computer intervention. The research mainly targeted the solving, generation and analysis of the flats puzzle and is used to determine what value the computer has in the puzzle creation arena and the scientific research on logical puzzles. Some results are presented and discussed about puzzles in common and about the flats puzzle. Special attention went into the research of the 5 sized flats puzzle, which lead to some surprising conclusions. Finally some conclusions are drawn about computer usage in the puzzle generation and analysis arena.

Chapter 1
PREFACE

"A man does what he can until his destiny reveals itself"
Nathan Algren, The Last Samurai

It is rare for a last year student to reach a scientific breakthrough, change the world or make sure he or she will always be remembered for his academic achievements. Although thoroughly aware of this fact, it still seems like your duty to at least try and be original. After not coming up with sell suited ideas or too ambitious ones, one feels reluctant to accept an offer to research logical puzzles when this has almost been an in-house standard, an easy scientific way to keep writing those articles with usually very low social relevance.

To some settling for a less ambitious project must sound like a defeat.
Like not reaching your full potential and searching for an easy way out. Working on this project however has never felt like a failure; on the contrary. Examining a logical puzzle, trying to see all its angles, analyzing it and trying to embed the results in test-software with real-world relevance proved challenging enough and to be anything but 'ivory tower science'.

Before you lies neither the document of a lifetime nor the scientific find of the century. It is a scientific report about research on logical puzzles and marks the end of a well used period on Leiden University; the life and time of a lifetime.

The thesis deals with the situation of computer intervention in the solving, creation and analysis of all sorts of logical puzzles with varying difficulty and appearances. To this end a test case is examined that will try to incorporate above activities for the flats puzzle. Much of the research was done based upon information gained in an interview with the editorial staff of Puzzelsport.

This thesis is the result of research that was performed by the author in the year 2004, forming the 'graduation project' in computer science at the Leiden Institute of Advanced Computer Science (LIACS), Leiden University. The research was carried out under the supervision of Dr. W.A. Kosters and Dr. J.M. de Graaf.

T.K. Cocx started studying computer science in 1998. While working his way from the beginning to the end at a steady pace he took some classes on teaching and, while not being religious, on religion and philosophy. This thesis forms his first academic report.

Chapter 2
LOGICAL PUZZLES

'Logic is the beginning of wisdom; not the end'
Spock, Star Trek VI Final Frontier

The concept of logical reasoning has been the basis of philosophical and scientific work since the era of the ancient Greeks and probably since long before. Although it has met a lot of criticism on its path, drawing conclusions from the facts at hand and converting them into new facts is now commonly accepted as the only pure basis of scientific research in the beta-field. Apart from the scientific and debating use, logic has always been an area where people can train there minds and have fun while improving there reasoning skills. Games, riddles and mathematical problems have amazed and amused people throughout the centuries. Strangely enough, it has only recently been incorporated in a profitable concept with the rise of logical puzzle-booklets.

In order to be able to correctly discuss logical puzzles, we first establish what is meant by both the words 'logic(al)' and 'puzzle' in the scope of this thesis.

Logic is the science that occupies itself with the study of correct reasoning [..]. It dissects human reasoning into combinations of unchanging patterns; so-called deductions [i]. Therefore

Def 1) In a puzzle context, logical deductions will be referred to as rules

Def 2) Logical puzzles concern the employment of rules.

A puzzle is an assignment meant for leisure purposes; a problem. We will focus on the first of the two. Puzzles have a solution that needs to be found by the puzzler (m/f).

Def 3) When a puzzle's input field resembles the intended solution perfectly we call the puzzle solved.

Def 4) Logical puzzles are leisure meant assignments that need solving.

Because this is still a very large set of possible dilemma's we will narrow the scope down a bit further for this thesis.

In the next few chapters we will discuss those logical puzzles that

- Have exactly one solution.
- Do not employ the use of undoable actions (like towers of Hanoi).
- Are meant to be solved on paper.
- Are solvable by one person (a one person game).
- Do not include the use of natural language
- Do not employ the use of prior knowledge, both trivial and scientific.

In other words: this paper deals with puzzles that people take with them on their vacation to have a good time while using their brain.

The Netherlands are one of the world's leading creators of this kind of logical puzzles and has the world's largest player community. In total 7.5 out of 16 million people occasionally puzzle and 3.5 million people puzzle at least once every two weeks. Although logical puzzling doesn't take a large part in this, puzzlers seem to be a large target group in the Netherlands. In some cases, the titles of puzzle types with a Dutch origin are preserved in Dutch, like the "Tentje-Boompje" puzzle (See Figure 1). Other nations that do well in this branch of leisure puzzles are Japan and Israel. Although this is a small group from the world's perspective, the popularity of logical puzzles is rising and more and more people start to do logical puzzles every day.

Production of the logical puzzle in the Netherlands is mainly in hands of publishers Denksport and Puzzelsport of whom the latter is the market leader in both the number of subscribers and the number of produced puzzles. Between the different companies a lot of ideas 'exchange'. Some types of logical puzzles appear in magazines published by both. But since only a single instance of one type can be copyrighted there is nothing that can be done about this kind of 'corporal theft'.

The most famous kinds of logical puzzles include: Battleship, Tentje-Boompje, Letter-window, Logigram (Language based) and the upcoming Japanese Puzzle. Most logical puzzles of this type tend to employ a grid basis that needs 'filling' of some sort.
Although many people are only familiar with one or two of the above, the range of different puzzles is very large and more types are created everyday. On national and world championships that have been around for approximately 13 years contenders get confronted with a large collection of new puzzle types.

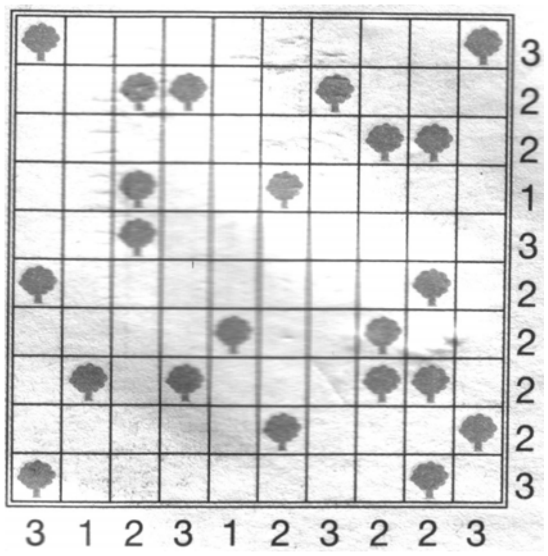


Figure 1. A Tentje-Boompje Puzzle. Place the given number of tents in every row and column where no tents touch each other and all tents touch a tree.

It seems inherent to this kind of puzzles that computers have a special edge. They can ‘see’ all the angles and can apply rules they ‘know’ in every circumstance they can be applied: the speed that a computer can employ in finding places where rules can be applied can lead to an exhaustive approach that checks all possible places for rule application and thus find all application places. A computer however needs programming by a human being. This paper deals with the involvement of the computer in solving, creating and analyzing these leisure meant logical puzzles from a human point of view. The next chapters deal with the question where and when the computer can and should be used to aid humans in this puzzle arena.

Chapter 3 COMPUTER INTERVENTION POINTS IN PUZZLE DYNAMICS

'I am a machine vastly superior to humans.'
Colossus, Colossus: The Forbin Project

3.1. Puzzle Life Cycle

A computer can intervene in a lot of phases during puzzle creation; the process where a single puzzle instance of some type is being created. In order to be able to talk about those phases efficiently, the computer intervention points will be linked to a so called puzzle lifecycle. Below we will show what such a life cycle would look like.

Def 5) The puzzle lifecycle describes what phases a puzzle under creation goes through and what characteristics are under evaluation at what point in time.

Def 6) The steps in a puzzle lifecycle are called the puzzle dynamics.

The puzzle lifecycle and its proposed computer intervention points look as in Figure 2:

In this lifecycle we distinquish six different puzzle dynamics. They are discussed in some detail in the following chapters. The proposed computer intervention points as laid out in Figure 2 will be discussed below the dynamics they affect.

3.2. Building

The first step in creating a puzzle can be seen as laying the foundation of a house. Just like a house depends on its fundaments to exist, a puzzle depends on its solution to make an impression. In practice not all puzzles are created by starting with a solution as will be discussed in 4.1, but most of them are. Because many puzzles are matrix based, most of the time building the solution means the same as filling the puzzle's matrix.

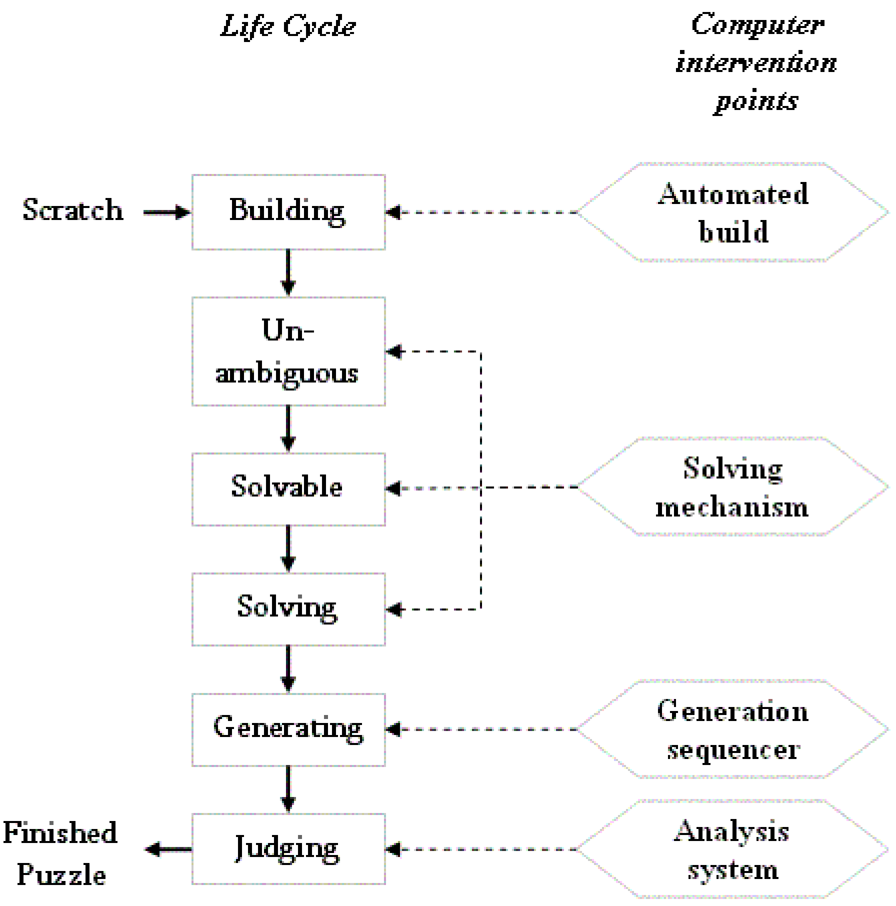


Figure 2. The puzzle lifecycle and its computer intervention points

Of course, a lot depends on the development of the solution. It can make a puzzle 'not-solvable' or even 'unambiguous'. It can make a puzzle easy or hard, fun or not fun.

The time to build a 'well-suited' solution may vary greatly between different types. This depends on the nature of the solution. A solution can as well be a randomly filled matrix as a complex structure of forms, in figures placed into some matrix. The 'problem' of different types of builds has its effects on the automation of the building process (see Figure 4).
Apart from this problem there are also puzzles that need to have a specific solution, because the solution in itself means something. This is the power of attraction of such puzzles but introduces a hard problem for the building phase. The most well known puzzle of this kind is the Japanese Puzzle (see Figure 3).

After we have built the puzzle's solution it is time to calculate the puzzle's characteristics, which are also part of the building phase.

Def 7) The information calculated from the puzzle's solution and available to the puzzler is called the puzzle's characteristics.

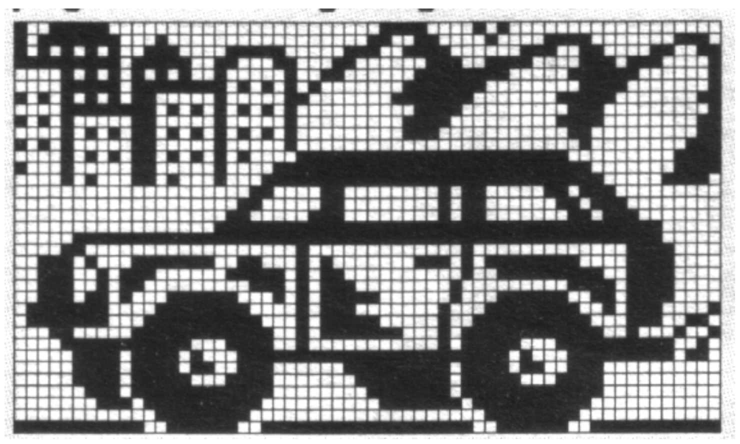


Figure 3. Japanese puzzles have a strict solution layout

Since most of the puzzles are matrix-based, deriving the characteristics is often simply calculating some kind of 1-dimensional projection of the matrix, but this can be more complex.
Although the Puzzle's characteristics are created in this phase, they can be changed later in the lifecycle (See 3.6 and 3.7).

After the solution and the characteristics have been developed the solution should be deleted to complete the building phase; Solving a puzzle that with an already complete input is trivial. This leaves us with the puzzle's characteristics. We will call the puzzle that now remains the puzzle's build. In some cases we leave some solution information behind to keep the puzzle solvable, but this is certainly not the default way to reach the end of the lifecycle.

3.3. Automated build

In a lot of cases it is desirable to automate the puzzle building process. It can be boring work filling up matrices and when implemented, a computer automated approach can be a considerable speed-up in going through the lifecycle. It should also give an edge to the puzzle-creator when he or she is disappointed with the first results. The computer would offer a simple click-on-the-button possibility to give an entire new result.
The puzzle characteristics are created even faster with computer assistance. Calculating the characteristics is pretty easy for a computer and saves the puzzle's creator a lot of time.

The results of computer intervention seem very promising, but the above mentioned problem gives rise to the inherent problems with computer programs doing human tasks; it is hard for a computer to 'come up' with an

original and challenging design, in this case the solution design. Puzzles that depend largely on this property seem therefore to lean heavily on this phase and make this one of the most important phases in the lifecycle (see Figure 2).
The fact that some different types of builds exist is not the only difficulty in automating the build process. Because of the importance of the solution it can in fact be pretty hard to build a basis that can satisfy the requirements put on the created puzzle to be which are described in the next few puzzle dynamics. Puzzles that have special solution layouts have entirely different demands than regular puzzles and therefore need a special approach. How this process works is not always clear (see 4.1.5.1).

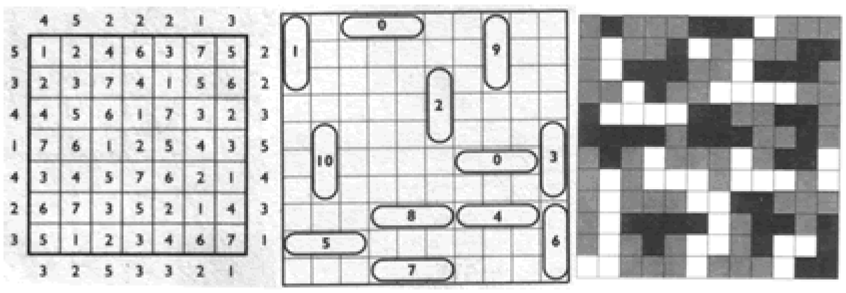


Figure 4. A large difference in a puzzle's solution type, ranging from easy (left) to difficult (right).

3.4. Unambiguous, solvable, and solving

3.4.1. Unambiguous

The first and in fact the most important demand we lay on the puzzle's build is that it is unambiguous. After the puzzle has been created, its characteristics have been setup and the solution has been deleted, there should still only be one solution possible; the one that we created in the building phase.
It is not that trivial for a puzzle to be unambiguous. In a large number of randomly developed builds the puzzle will turn out to be ambiguous. This depends however again on the puzzle's type. Some types will obviously be more open to ambiguity than others.

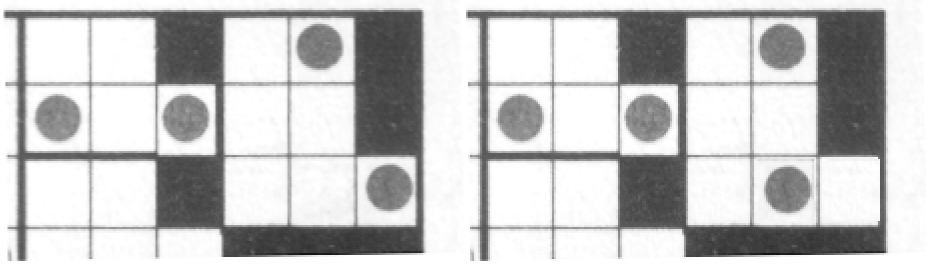


Figure 5. If both satisfy the characteristics this puzzle is ambiguous

When a puzzle turns out to be ambiguous one has to start the lifecycle again from the building phase. It is of course also possible to make small ambiguity repairs; in other words: adapting the puzzle's build in a number of small ways to make it unambiguous.
The most promising approach however would be to take the ambiguity phase under consideration during the building step, so no repairs or total refits would be necessary. This can turn out to be pretty hard in practice; especially when using the computer (see 3.5).

3.4.2. Solvable

Unlike the ambiguity phase the solvable phase deals more with the human component. Of course, when one has unlimited time and resources every unambiguous puzzle is also solvable. Just trying every possibility will eventually give the puzzler the one sought solution, but this method is hardly efficient, fun or doable. Instead of accepting the unambiguousness of a puzzle as the only heuristic we define the solvability phase as follows:

Def 8) A puzzle is solvable when an experienced puzzler can solve it in a reasonable time, without largely using the brute force way.

Def 9) The amount of difficulty that such a puzzler has in solving the puzzle defines the difficulty of that puzzle.

Of course these definitions are themselves ambiguous. It is not instantly clear what is meant by 'largely', 'reasonable' 'experienced puzzler' and 'amount of', but this is not really a problem, because anybody can make a somewhat consistent guess. For this thesis we will follow the following directives:

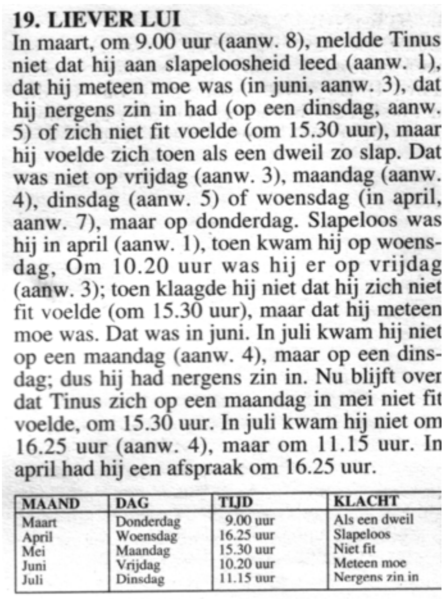
- A reasonable time is approximately between 30 and 45 minutes
- An experienced puzzler is a person that solved a number of puzzles of this type before.
- Largely using the brute force way means using the brute force way as main way to solve the puzzle.

Whether or not a puzzle is solvable depends largely on the subjective opinion of the puzzle's creator or the computer program's programmer or designer. They are the ones who define what the reasonable amount of difficulty actually is.

If a puzzle doesn't turn out to be solvable one has the same two options as when the ambiguity phase fails: either build a new puzzle or make repairs to the existing one. Of course one can also make sure during the build phase that all build puzzles are solvable. But as said before, this can be very difficult to accomplish.

3.4.3. Solving

It is inherent to a solvable puzzle that it can be solved. It is the task of the puzzle creator to make sure it is solvable by solving it himself. In that way he can test the solvability and can familiarize himself with the way in which he solves it. In this phase we will actually solve the puzzle that has just been built and been proved solvable, in order to give back information about how the puzzle can be solved. With some types of puzzles it is customary to present an explanation on the solution pages instead of just the plain solution (see Figure 6). For those types of puzzles this phase should be done with extreme care, because the creator should be able to present the puzzler with a clear overview of the rules that lead the puzzler to the right solution.


19. LIEVER LUI
In maart, om 9.00 uur (aanw. 8), meldde Tinus niet dat hij aan slapeloosheid leed (aanw. 1), dat hij meteen moe was (in juni, aanw. 3), dat hij nergens zin in had (op een dinsdag, aanw. 5) of zich niet fit voelde (om 15.30 uur), maar hij voelde zich toen als een dweil zo slap. Dat was niet op vrijdag (aanw. 3), maandag (aanw. 4), dinsdag (aanw. 5) of woensdag (in april, aanw. 7), maar op donderdag. Slapeloos was hij in april (aanw. 1), toen kwam hij op woensdag, Om 10.20 uur was hij er op vrijdag (aanw. 3); toen klaagde hij niet dat hij zich niet fit voelde (om 15.30 uur), maar dat hij meteen moe was. Dat was in juni. In juli kwam hij niet op een maandag (aanw. 4), maar op een dinsdag; dus hij had nergens zin in. Nu blijft over dat Tinus zich op een maandag in mei niet fit voelde, om 15.30 uur. In juli kwam hij niet om 16.25 uur (aanw. 4), maar om 11.15 uur. In april had hij een afspraak om 16.25 uur.

MAAND	DAG	TIJD	KLACHT
Maart	Donderdag	9.00 uur	Als een dweil
April	Woensdag	16.25 uur	Slapeloos
Mei	Maandag	15.30 uur	Niet fit
Juni	Vrijdag	10.20 uur	Meteen moe
Juli	Dinsdag	11.15 uur	Nergens zin in

Figure 6. An extended solution explanation (in Dutch)

Actually solving the puzzle is not a mere tool to come up with the used solving rules. It is an important phase as preparation for analyzing the puzzle, for it can play a large role in determining the difficulty and how much fun it is (see 3.8).

3.5. Solving mechanism

Because of the above mentioned directives on reasonability, the set of solvable puzzles can be seen as a subset of the set of unambiguous puzzles. It would therefore be sufficient to prove a puzzle solvable to accept it as a doable puzzle. Because you also have to do the solving anyway one can just as easy combine above mentioned dynamics into one computer intervention moment.
For humans this is not as trivial as it is for computers. One can sometimes provide a solvable puzzle more easily than one can solve that puzzle itself, because of the human gut-feeling. In essence, this relates to creating a software program that you 'know' to be correct without ever proving it so. It is not always obvious or easy to explain how one determines whether or not a program is correct or in this case a puzzle is solvable. For humans it is therefore easier to produce a solvable puzzle then to show the way its solution is reached. For computers on the other hand solvability and solving are in essence the same thing.

Computers are well known for their solving capabilities. The old computer tactics have served us well in solving discrete mathematical problems as well as in solving other discrete, more information and programming based, issues. The somewhat new AI-way of looking at problems has introduced new ways to handle these kind of more complex problems. In contrast to the old algorithmic approach an Artificial Intelligence engine would handle the problem much more like a human would do it; exactly what we want for our puzzle solving mechanism. It is obviously important to make sure humans can do the same as the computer that assisted the creator during the lifecycle did. The most straightforward conclusion would therefore be that an AI approach would operate much better than a classic one.

It is clear that computers can handle the solving phase a lot faster than humans can, whatever approach they take. Considerable speed-up can be reached in proving unambiguity and solvability and, depending on the approach, a set of rules used to solve this puzzle instance can almost instantly be generated. It can however be one hell of a job to program the computer and to make sure it is not accepting unsolvable puzzles or throwing away solvable ones.
Computer intervention can also present a new problem: the computer can be too good in doing its job. Two problems may arise here:

- Approaches that use own learning or advancement methods (Genetic Algorithms, neural nets) may introduce rules that no man has or can ever come up with and thus introduce a new measure of solvability.

- Although the programmer inserted 'human' rules himself, the computer still has the edge of seeing all the angles. It can and will spot every location where the appropriate rule can be used to get nearer to the final solution and thus introduce a new standard for solvability as well.

Both these problems could be dealt with in the analysis phase (see 3.8.1).

Integrating the solving system into the automated build gives us a lot of problems and gives rise to the question if the gain is worth the trouble. It would surely make the building of flawed puzzles unnecessary but there are too many questions unanswered:

- When does the checking take place?
- How does one detect problems before the puzzle is finished?
- How does one recover from an arising problem?

These are questions that strongly support the approach where there are two different intervention phases.
On most machines and with the most puzzles this should not present any difficulty. Building and solving should not give rise to difficult and long lasting computation. When performance, however, is becoming an issue, this approach should come under revision.

3.6. Generating

The creation of a puzzle actually consists of going through the earlier dynamics until you reach a candidate puzzle that complies with every requirement. By now we will have a puzzle that has a solution, characteristics and is solvable by humans.

After completion of the first dynamics with a positive result one can wonder if this puzzle is all it can be and reach its full difficulty and excitement potential. It might be possible to elaborate on this puzzle and see if we can 'spice it up' a bit.
The least we know is that the current solution gives rise to a solvable puzzle. We now have puzzle that is considered finished. It is, however, also possible that the same solution can still be the basis for a solvable puzzle whilst giving the puzzler less information. Since we already got a finished puzzle we can optionally introduce new difficulties or fun. The main thing that can be done to achieve this is to change the quantity of characteristics. The easiest way to do this is to strip the given characteristics or scrap some other kind of information. The puzzle is Figure 7 is an example of how stripping the characteristics can make a puzzle harder, more fun or both.

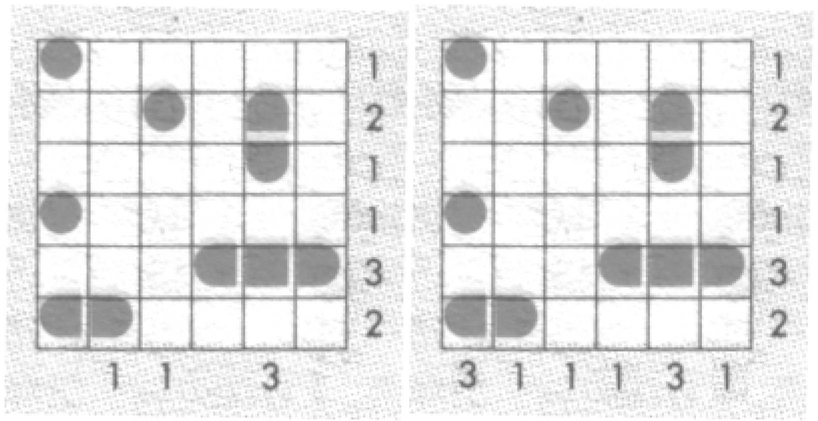


Figure 7. A (solvable) stripped battleship puzzle and its fully filled original

Creating a puzzle can be seen as entering a loop in which one builds and solves until a good puzzle is found where after one enters a loop of stripping and solving until the puzzle is as the creator likes. This can obviously be an intensive job.

3.7. Computer generation sequencer

For a human, working in a seemingly unending circle can be quite tedious and cost a lot of time. The computer can do this task of endlessly building and solving puzzles fast and efficient, providing the earlier dynamics have been programmed efficiently. Depending on how good the building and solving algorithms are and how many puzzles of a given 'size' are considered solvable the Generation sequencer should find a satisfying puzzle within a short time. It is necessary to keep the above mentioned size-problem in mind. For example: a battleship puzzle of size 40x40 will almost never be solvable and therefore be un-creatable. This counts for humans as well as for computers, but it is mentioned because people tend to believe that computers will eventually find such a puzzle and not consider the limitations at all. People will give up much sooner when creating a puzzle by hand.

After a suitable puzzle has been found the computer can try its hand at stripping the characteristics. A lot of algorithms exist that try a lot of possibilities and thus try to find some kind of maximum, most of which are recursive. This can take a very long time. Another approach would be to constructively strip the characteristics by first determining if one and what characteristic can be deleted. This, however, would need a predictive power build in the algorithm that would be very hard to design. From all phases, completing the information strip would be the most time absorbing.

It is, of course, also possible to let the computer settle for a less than maximal strip, just like humans would. This however requires insight into what a near maximum is (see 6.6.2.1).

3.8. Judging

After a puzzle is created it is time to judge its fitness for publication. Obvious questions would be:

- Is it good enough to make publication?
- If so, in what booklet should we publish it?
- What is this puzzle's target group?

These criteria are hardly measurable but if we take a closer look they can all be answered when we know:

- How hard the puzzle is
- How much fun the puzzle is
- How interesting a puzzle looks.

We will discuss them below.

3.8.1. How hard

It is always interesting to know how hard a puzzle will be for a human being. When beginning puzzlers get to solve too hard puzzles they will stop buying the booklets that contain them. The same goes for the experienced players trying too easy ones. Analyzing the new puzzle on its difficulty will make sure it ends up in the right booklet or gets published in the right magazine.

3.8.2. How much fun

Since the puzzles under consideration are meant for leisure purposes it is very important for a puzzle to be fun to do. It is absolutely imperative that a puzzle creator determines the fun rating of the puzzles he creates.

3.8.3. Graphically interesting

Since puzzles get published in external magazines and shiny booklets it can make a difference if a puzzle looks good. Although physical appearance doesn't really add to the puzzling experience it can be a factor when people browse for a good booklet in the bookstore.

A puzzle can have an entirely different set of rules that solve it than another and still look really alike (see Figure 8). Having a lot of puzzles that look the same doesn't add to your booklet's appeal. Puzzles that appear out of balance or look uninteresting at first sight should be discarded if high priority is given to a quick sell or to retain subscribers.

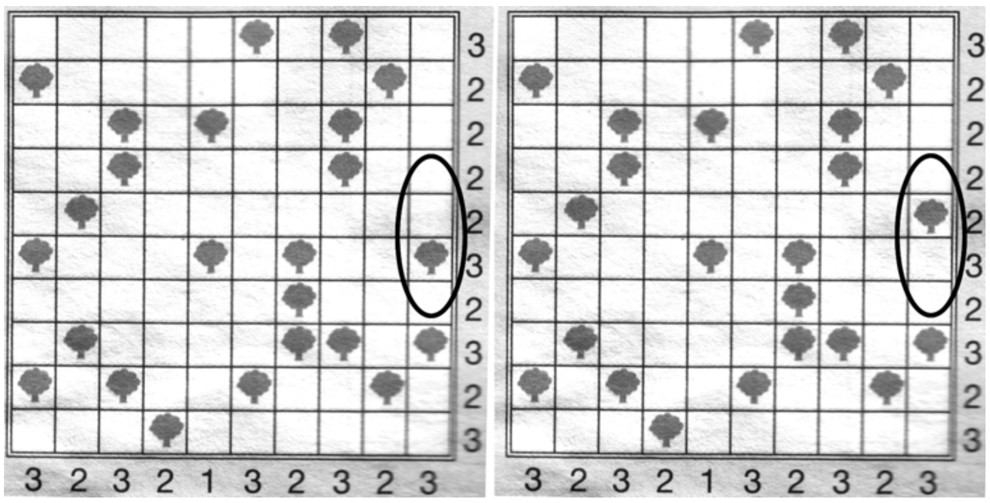


Figure 8. Two totally different puzzles that look really alike

Which one of the above is important depends on your target group. A holiday puzzler would probably want easy, fun and good looking puzzles in his booklet; a subscriber would probably want the same except for a higher level and at the world championships physical appearance and fun are not important at all, focusing only at the high difficulty. It is up to the puzzle creator to determine for whom he is making this puzzle, **so no puzzles should be discarded based on the analysis itself.**

3.9. Computer analysis system.

3.9.1. Discarding (primary) subjectivity

During human puzzle evaluation the threat of subjectivity and inconsistencies in the puzzle's creator decisions is always present. Ideally, a computer system would overcome this problem and judge puzzles as an average puzzler would judge it, thus outruling disappointment and frustration during the puzzling process. In the next best case, the computer would at least enable the creator to draw the same conclusions on the same facts at hand, coming to a more consistent ruling. Therefore, the computer could in essence provide a very good way to achieve the ratings proposed in the above section, by introducing less subjectivity than a human puzzle creator.

3.9.2. Classes of puzzles

From a mostly scientific point of view it would be interesting to try if we can categorize the current puzzle into a class with other generated puzzles that are a lot like it. Doing this we can learn more about the nature of the puzzle and see if there is more behind it than meets the eye.
This is something that only a computer can do. A number of methods are known today that can extract information from a pool of data and cluster it accordingly (data mining). Whether or not this classification provides us with any interesting information can unfortunately not be predicted beforehand.

3.10. Puzzle variations

Some puzzle types are known to have some variations. The battleship puzzle for example usually consists of one simple fleet, but also has the variation of two larger fleets and even a value based version.
It might be interesting to know whether or not the puzzle type under investigation can be transformed into an adaptation and to see if we are getting the most out of the current concept. Of course it is imperative that those new puzzle kinds remain solvable and fun to do. The computer can obviously play a part in this.

Chapter 4
CURRENT APPROACHES TO THE LIFECYCLE

'You've only got their word on it…
- It's what we depend on!'
Rosencrantz and Guildenstern, Rosencrantz and Guildenstern are dead

4.1. PuzzelSport

4.1.1. About Puzzelsport

Puzzelsport is a subdivision of Sanoma Publishers bv., the previous VNU magazines. They sell 4 million editions a year to both men and women aged approximately between 30 and 60 years. They own 37% of the puzzle-market. Next to their own brand Puzzelsport, they own the A-brands '10 voor Taal' and 'Jan Meulendijks'. Every Puzzelsport division has got its own redactional formula. One of them is the logical puzzle division, which is mainly run by one editor in chief, two editors and a bunch of freelancers.

Jan Meulendijks 10 VOOR TAAL Puzzelsport

Figure 9. The Puzzelsport brands

Puzzelsport's logical division is a member of the world puzzle federation and holds national tournaments under their flag. Once a year they organize the Dutch championships. The best players of this championship are sent to the world championships. Last year Puzzelsport organized that tournament in the Netherlands.
The tournaments and the membership are important contact possibilities for Puzzelsport. These provide them with new puzzle inspiration and give them contact with their target group, which is more like a community that gets together at championships and helps the Puzzelsport people. Contact with this group is maintained with a lot of care. Puzzelsport also uses its website for this purpose.
Puzzelsport sells puzzles to other divisions of Sanoma Publishers or sells logical puzzles to extern bodies, but the most part of their efforts go into their two booklets: Breinbrekers that comes out every three months and Logimix that comes out six times a year. While the first consists of a lot

of different styles and varies every edition, the latter encompasses only the five most well known types of logical puzzles: “Tentje-Boompje”, Battleship, Letter-window, Swedish number diagram and Japanese Puzzles.

{ LogiMix }

BREIN BREKERS

Figure 10. Puzzelsport's logical editions

4.1.2. Manual or obscure creation

Most of the puzzles in the Breinbrekers booklet are still created by hand. One man sits behind his desk and starts creating puzzles in an instinctive and somewhat unstructured manner. Although this may not seem real cost-effective, it in fact is. It is apparently not worth the while to invest in puzzle creation programming. Most of the puzzles in this booklet get published only a few times a year and the total collection of different kinds of puzzles is very large, so one would never reach the breakeven point when going for a programming approach.
Another part of these puzzles is created by freelancers. Puzzelsport has no influence over the way these puzzles are created and has no clear concept of how these freelancers do their work. They may or may not create the puzzles with computer assistance or may only use the computer for the unambiguity or solvable phase.
Most of the work pool of freelancers came out of a small community of puzzle hobbyists. They come up with new puzzle types, create puzzles for the booklet editions, participate in championships and help the Puzzelsport staff in extending their puzzle know-how. Obviously, this is one of the reasons Puzzelsport maintains close ties with their puzzler community.
All created puzzles are tested extensively by a large group of testers on solvability. This is also the group that helps in determining the puzzle’s difficulty.

4.1.3. Programs

In sharp contrast to the above mentioned puzzles, Puzzelsport owns computer creation programs for all the puzzles in its Logimix editions. Some of them were acquired from freelancers, some were in house

products and others just came from the internet and other unknown sources. All programs have been used for some time and never failed during testing. All produced puzzles, however, are still checked on solvability and given a difficulty degree.
In some of these programs the creator can select a couple of settings that helps in determining the difficulty degree before the creation process and help in making sure the puzzle has also a visual appeal. Some programs are elaborately designed and offer exporting possibilities, other programs operate in shell boxes and need complete copying before publishing.
Most of the in-house applications use a rule-based approach to the solving and the creation phase. At Puzzelsport they think rule-based approaches to be the closest to humans as can be and on the other hand be capable enough to do the tasks at hand quickly and most importantly: perfect. Although getting partial solutions might be interesting in the scientific world, Puzzelsport considers them useless. They are only interested in using perfect solution programs and wonder what people would ever want with an incomplete or partially wrong solution.
The rules that are used in the rule-base programs tend to have an inherit difficulty associated with each of them. Creating a puzzle of a particular difficulty then becomes the same as selecting appropriate rules for that difficulty that the program may use during puzzle creation time. Only selecting easy rules creates an easy puzzle, also or only selecting hard ones makes the puzzle difficult.

Def 10) Two solving rules are disjunctive when additions to the puzzle's field made by one rule do not prohibit application possibilities of the second rule

If two rules are non-disjunctive the following may happen. The first rule alters a place in the puzzle's field. Because of this, the situation has become more specific. The first rule, only applicable in a more general situation now fails to work. These rules are therefore non disjunctive.
The rules in the programs owned by Puzzelsport are all disjunctive: they can all be applied in every situation and it does not matter in what order they are evaluated by the program. It is not clear if this is a property inherit to the types of puzzles the programs create or if it is just clever rule developing.

4.1.4. Non-Logic Puzzle creation

Although it falls out of the scope of this thesis, it is worth mentioning that, strangely enough, computer programs are extensively used in the creation of non-logic puzzles at Puzzelsport. Especially in the language based areas computers are the only means of puzzle creation. Those puzzles are created fast and cheap and thus take a large part in corporal profits. The advantage those puzzles have over logical puzzles is that Puzzelsport can fill a whole edition with the same type of puzzle without people labeling them as boring. This is definitely not the case with logical puzzles.

4.1.5. Special creation methods

4.1.5.1. Japanese Puzzles.

The Japanese Puzzles are becoming a more important part of Puzzelport's logical puzzle department every day and enjoy a lot of popularity nation-wide. However, they do not have the in-house knowledge to create these puzzles themselves. The puzzles are bought from Conceptis [ii] and they maintain a strict copyright on their creation methods. It is therefore somewhat of a mystery how these Japanese puzzles are created.

4.1.5.2. Logigram

Although Logigrams are natural language based and therefore fall beyond the scope of this thesis, the way these puzzles are created is worth mentioning. They are an example of puzzles that have a different approach in the creation of characteristics. After the solution has been built the characteristics are created adding little bits after little bits until the creator deems the puzzle solvable. This can be a very tedious job, but since the caption of natural language by computers is not that advanced, it seems like the only solution for now.

4.2. Science

Although a lot of effort of the computer science departments has gone into solving small mathematical problems and puzzles in the past, the arena of publishable puzzles has been widely neglected thus far. Of course, the limited area where those puzzle booklets are sold, and therefore where the puzzles it contains are known, has a large part in this.
The master thesis of Joost Batenburg [iii] deals with the solving of Japanese puzzles through a mathematical foundation that was extended for this puzzle. It tries to approach the perfect solution by a genetic method and sometimes succeeds. A lot of puzzles however will not be solved by the program. The solving of such a puzzle would take a very long time (hours)

and does not provide certainty whether or not a correct solution can be provided.

Chapter 5
REVIEWING THE CURRENT APPROACH

'Now, let's just take a second here and take a hold of the situation and review our options.'
Robert Boyd, Very Bad Things

5.1. Difficulties and programming gains in creation

Although Puzzelsport claims that creating puzzles by hand is more cost-effective than a programming approach, it may be worth the time to investigate how much time it would really cost to develop a program and start using it for puzzle creation. Some speedup may be reached or even when the breakeven point will never be reached, it opens the door for a more elaborate version of Logimix, which can now as easily contain the puzzle type created by the new program or even a new type of issue. Therefore, next to the possibility of a gain in efficiency more advantages can possibly be found to programming puzzle creation software.
Next to the puzzles that are now created by hand, it is not clear how good the rule-based approach is or can be for puzzles that are created using such programs. It may as well leave a whole class of good puzzles unsolved and therefore uncreated. It may therefore be a good idea to check how such a system performs compared to other approaches like a brute force strategy.

5.2. Difficulties and programming gains in analysis

As was mentioned above, determining the difficulty of a puzzle often happens in an obscure way, by hand, or by determining what rules are to be used during puzzle creation time. All approaches have their respective problems.
As long as the determination of a puzzle's difficulty is done by a single person that estimates what the opinion of the user group would be on unclear reasons, a number of problems are introduced. First, the creator, being an experienced puzzler by nature, may not be capable of deducing how hard a puzzle would be to a beginning puzzler. Second, knowing not what reasons lay behind a decision may well introduce the subjectivity problem as discussed in 3.9.1.
It is not inherit to logical puzzles to have solving rules associated with them that encompass a natural difficulty rating. The fact that most puzzles created by Puzzelsport programs tend to have that property does not mean that the majority of puzzle types share it. Trying to attach a difficulty rating

to a newly created puzzle while not neglecting the previously mentioned subjectivity and inconsistency bias can be a hard job. Although it may be smart to limit your program to puzzle types that do include such a difficulty inheritance, it can limit a creator in his options. Developing an automated rating system can avoid the above biases and still rate all kinds of puzzles in a (semi-)objective manner.

5.3. Difficulties in Scientific approaches

Scientific approaches always seem to have their own truths and values associated with them. As was mentioned earlier (see 4.2), examples exist where also partial solutions to a puzzle can have meaning and are still considered ‘good’ approaches. Integrating scientific approaches into real-life situations requires a more goal oriented way of thinking instead of the inherit value of the research done. Seeking scientific solutions to everyday problems requires an everyday judging of the provided solution. It should therefore not be forgotten that only fully correct software provides any solution to normal man or business.

Chapter 6
PUZZLE TEST CASE: THE FLATS PUZZLE

'World's worst place to get a flat, huh?'
Jim, 28 days later

6.1. Puzzle Choice

Selecting a puzzle for the test case was somewhat difficult since the most well suited puzzles could already be created by computer programs. The search for a puzzle ended when the booklet revealed the Flats puzzle. This type of puzzle was well suited for making a test application because:

- It is challenging to solve it, even for experienced puzzlers.
- Building it is not that hard.
- Calculating all the characteristics still leaves a good puzzle.
- Solving rules do not tend to have a difficulty associated with them.
- The core of the puzzle is mathematically interesting (see 6.3.2).

6.2. The Flats Puzzle

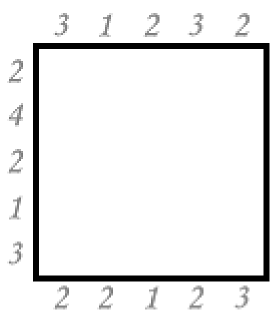


Figure 11. A typical Flats puzzle

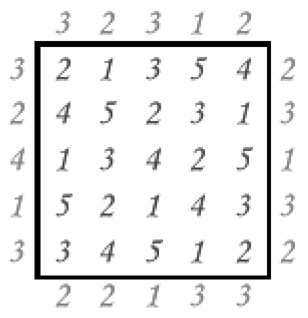


Figure 12. A solved Flats puzzle

The flats puzzle deals with a square area designated for flat construction. All flats take the same amount of ground space but are of different heights. All floors are of the same height, so a building of size 2 is exactly 1 floor lower than a flat of size 3. The heights of the flats to be built range from height 1 to height 5 (for a square of size 5).
The project management has a few demands on where the flats are built:

- Every possible location is used for flat construction
- When the area is divided into rows and columns every row and column should contain exactly 1 flat of all possible heights.
- From an esthetic point of view, the city ordinates how many flats a person should be able to see when he passes that row or column.

(Note that a flat of, for example, height3, blocks the flats of height 1 and 2 in that row or column from viewing (see Figure 13)).

- As constructor, it is up to the puzzler to determine where all the flats are to be built.

In Puzzelsport the flats puzzle is described as follows: a diagram displays a group of flats. In every row and column the heights *1* to *5* (or *1* to *6* or *1* to *7*) appear exactly one time. De numbers outside the diagram designate how many buildings are visible in the considered row or column from that side (see Figure 13).

Of course it is up to the puzzler to find the only corresponding placement of the flat buildings and fill in the diagram accordingly. From now on until the end of this chapter we will mean a Flats puzzle every time we use the word puzzle.

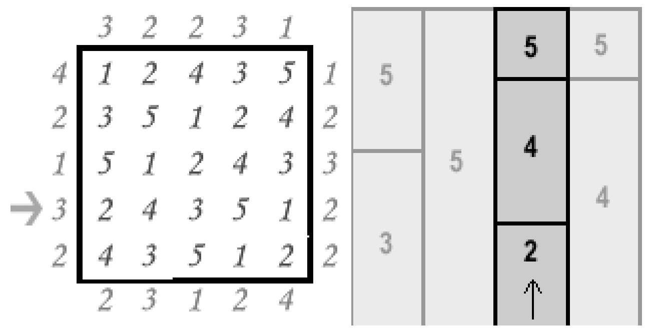


Figure 13. The derivation of the view characteristic (puzzle view and 3D)

6.2.1. Mathematical definition

More exact, we can state that the flats puzzle consists of:

- An integer *size* є {5, 6, 7}.
- A Latin square *(size x size)* as **solution** (definition: see 6.3.2).
- A matrix *(size x size)* as **field**.
- The projections (views) of the matrix on all four of its edges put into four *size*-sized arrays that represent the flat-view from all four possible directions(see Figure 13).

According to Def 3) the flats puzzle is called solved when the **field** matrix is the same as the **solution** matrix.

6.3. Puzzle domain

6.3.1. Views

The puzzle's views range obviously from *1* to *size,* for the simple reason that at least one flat is always visible (the flat with height *size*), and that we can see *size* flats at maximum (all the flats).

The mathematical domain of the contents of any puzzle's view arrays is therefore *{1, …, size}*.

The boundaries of the view domain describe a special situation when they appear as a view from any direction. We will describe them in 6.5.6.2 and 6.5.6.3.

6.3.2. Latin squares

6.3.2.1. Definition

A Latin square is an $n \times n$ table filled with n different symbols in such a way that each symbol occurs exactly once in each row and exactly once in each column [iv].

6.3.2.2. Number of squares

Table 1. Number of (reduced) Latin squares [v]

Size of Square	Number of Squares	Number of reduced squares
1	1	1
2	2	1
3	12	1
4	576	4
5	161280	56
6	812851200	9408
7	61479419904000	16942080
8	108776032459082956800	535281401856
9	Too large	377597570964258816
10	Too large	Too large

It is known for small integers *X* how many Latin squares of size *X* exist. Below is a table that describes these numbers. The numbers respectively are series A002860 and A000315 from the on-line encyclopedia of integer sequences [vi]. The last column will be described in 6.3.2.3. It is known that the number of Latin squares is found from the number of reduced Latin squares using the following formula:

n! · (n - 1)! · Number of reduced squares [ix]

6.3.2.3. Latin square generation

It is obviously difficult to work with numbers this large. It is nearly impossible to use all the possible squares in your puzzle lifecycle. The standard puzzle sizes already encompass more than 100.000 different squares. Puzzle generation on the other hand can be much simpler than generating all the possible different Latin Squares. Just switching a random number of rows and columns will give you a new square that is still Latin. However, in contrary to what one would expect, this trick doesn't yield all possible Latin squares. It turns out that this gives all possible Latin squares derived from a reduced square; a square with *1* to *X* on the first row and the first column. This means, that always starting with the same square in the previous mentioned method, limits the algorithm to just a small portion of all possible squares, by a factor ranging from *56* to *16942080.* No efficient algorithm is known to generate all possible reduced squares of size *X.*

6.3.2.4. Influence over domain

The fact that we can not, in any efficient way, generate all the possible Latin squares of size *size*, can obviously have a large effect on what puzzles we will be able to generate. As long as we always start with the same reduced Latin square we will always end up with a puzzle derived from that reduce square and it therefore create just a subset of all possible puzzles, although the influence of this incapability depends on the size. The following test case should also try to give an idea about the amount of problems this will add to our puzzle's lifecycle.

6.4. Application preliminaries

After establishing the mathematical domain we can start describing the goals of our test case application, that we will call 'Flats Unlimited' and start building it.

6.4.1. Application goals

The main goal of our application would obviously be to understand more of the puzzle lifecycle and try to incorporate scientific approaches into the puzzle creation process that would make it faster, cheaper and presents better feedback to the creator while trying to establish a general standard for puzzle creation. It would therefore answer some of the questions to the current approaches as posed in Chapter 5.
Subgoals for the application will therefore be to:

- Create a workable, efficient and understandable solving mechanism by trying different approaches.
- Incorporate such a mechanism into an efficient generation sequencer.
- Create an (semi-) objective analysis system to analyze puzzles for publication.
- Provide viewing, monitoring and setting options to help the puzzle creator (creating generation control).

6.4.2. Application functionality

In order to reach our goals, the application to be would need at least:
- An understandable user interface with enough feedback and customizing options (see 6.8).
- The possibility to save and load puzzles in order to come back to a specific puzzle later (see 6.8).
- A puzzle viewer (see 6.8).
- A very fast puzzle solver that can employ different kind of techniques (see 6.5).
- A reasonably fast puzzle generator (see 6.6).
- A reasonably fast puzzle analyzer to supply both creators and scientists with information (see 6.7).

6.5. Solving a Flats Puzzle

6.5.1. Approaches

There is more than one approach to take in creating the solving mechanism of the application. The main approaches in AI would lead us to a neural net, a genetic algorithm and a rule-based approach, among others [x].
Since we are looking for a solving algorithm the neural net seems like a very long shot because of its rigid input and output characteristics. Genetic algorithms have shown some results, although far from perfect,

in the past in trying to solve Japanese puzzles [iii]. Rule-based approaches have proven themselves in practice but need a lot of human input to work. The latter two will be incorporated into the application and judged on their performance.

6.5.2. Standards

We first need to set a standard for the solving mechanism to be built. Main question in setting these standards is:

- What is the minimum performance level the solving mechanism should have?

Since we have limited insights in the puzzle and do not know beforehand what puzzles are considered solvable, this main question leads us to the following performance demands:

- The mechanism should at least be capable of solving all the puzzles in the Puzzelsport booklets, since it is known that these are considered solvable. It should preferably solve all puzzles that are considered solvable or at least a large portion of them.
- It should not search endlessly or longer than a human for a solution.
- It should never 'solve' an unsolvable puzzle.
- It should preferably come up with a used rule set, to enable human checking.

6.5.3. Generating rules

The best thing a solving mechanism could do is solving the puzzle on itself, without entering human knowledge and while providing the creator with a used rule set. This would open the door for novices to learn how to solve a puzzle and help the creator to extend his understanding of the puzzle. The difficult and far from standard approaches that this effort would take is the reason why this pursuit falls out of the scope of this thesis.

6.5.4. Genetic approach

6.5.4.1. Algorithmic approach

Warned by the incomplete results provided by the above mentioned solving of Japanese puzzle,s we will try a small version of a genetic approach on our flats puzzle, before we decide to fully implement a not eventually non functional approach. This will contain a

generation sequencer in the form of a hillclimb operator, some survival strategies like elitary selection and a mutation operator. The fitness function will assign each puzzle a fitness based upon how many views of the suggested solution map to the intended solutions and how many times the Latin directive was violated.
We start with random Latin squares put in the puzzle's field. The mutation operator now switches two locations or mutates one location to another flat height, leaving us with an field that is no longer Latin, which will be difficult to repair while maintaining some the undergone change.
As we decided in 5.3 our system should only approve with puzzles that fulfill the puzzle characteristics perfectly.

6.5.4.2. Performance

Our small test case performed miserably:

- The algorithm takes way too much time to even come to a slightly better solution than the randomly generated solution.
- The puzzle always seems to get to a local maximum. As we could see in the approach used to solve Japanese puzzles, we are in some cases able to recover from the mistakes made by the mutation or crossover and to come to better candidate solution. As the example also showed this does not necessarily lead to a better chance in solving the puzzle. Even when the puzzle could be solved it took 10 hours due too all the necessary recovery work, which is far too long to be adapted in a professional business.

Next too the fact that it would mean inefficient working time, the flats puzzle has two main requirements to meet that will be distorted by mutation or crossover, where the Japanese puzzle has only one. Japanese puzzles only concern themselves with the order of some groups of black cells while the flats puzzle concerns itself with both the order in which the cells are filled and the way a single cell is filled. This destruction in 2 dimensions will only make it harder and even more inefficient to recover from a change in the field layout.

- It turns out that our small test case leads to the following results:

Table 2. Average results for the genetic test-case

Puzzle Size	View and/or Latin properties wrong after 10,000 Generations
5	9
6	13
7	18

As we can see in Table 2, after 10,000 generations a average puzzle of size 5 still fails to comply with the characteristics or the Latin directive in 9 places; a number that grows larger with the size of the puzzle.
The poor results above and the lessons we can learn from the Japanese example draw us to the conclusion that it is not worth much more time to pursue a genetic approach.

6.5.5. Backtracking approach

6.5.5.1. Algorithmic approach

Backtracking methods are used of old to solve (mathematical) problems of all sorts. They in essence research all candidate solutions and check them on correctness. This will eventually deliver all candidate solutions that fulfill the requirements and are thus considered true solutions. For this puzzle we can adapt the backtracking procedure to stop when more than 1 solution is found; meaning that this puzzle is ambiguous. When exactly 1 solution is found, the puzzle is solved. For this procedure it is customary to use a standard recursive function. We will in fact generate all possible squares row by row and backtrack whenever the Latin directive or a view is violated. We will give the function a stopping limit of more than 1 solution as mentioned above.

6.5.5.2. Performance

Backtracking methods are known for their thoroughness as well as their time consuming approach.
As expected, our algorithm reveals all the unambiguous puzzles and is able to solve all the puzzles provided by Puzzelsport and therefore scores 10 out of 10 for the completeness demand. The question remains, however, if the algorithm will not also solve unsolvable puzzles (unsolvable as defined in Def 8). Since the algorithm is unable to come up with a human way to solve the puzzle we can not rely on the solvability of the currently solved puzzle. Neglecting this information will also hinder the later analysis that finds its basis in solving information.
In contrary to its above mentioned time consuming reputation, the algorithm actually does its work pretty quickly; ranging from a very fast solving of the size 5 puzzle to a reasonably fast solving of the puzzles of size 7. We can consequently conclude that unambiguous puzzles tend to trigger the backtracking very fast.

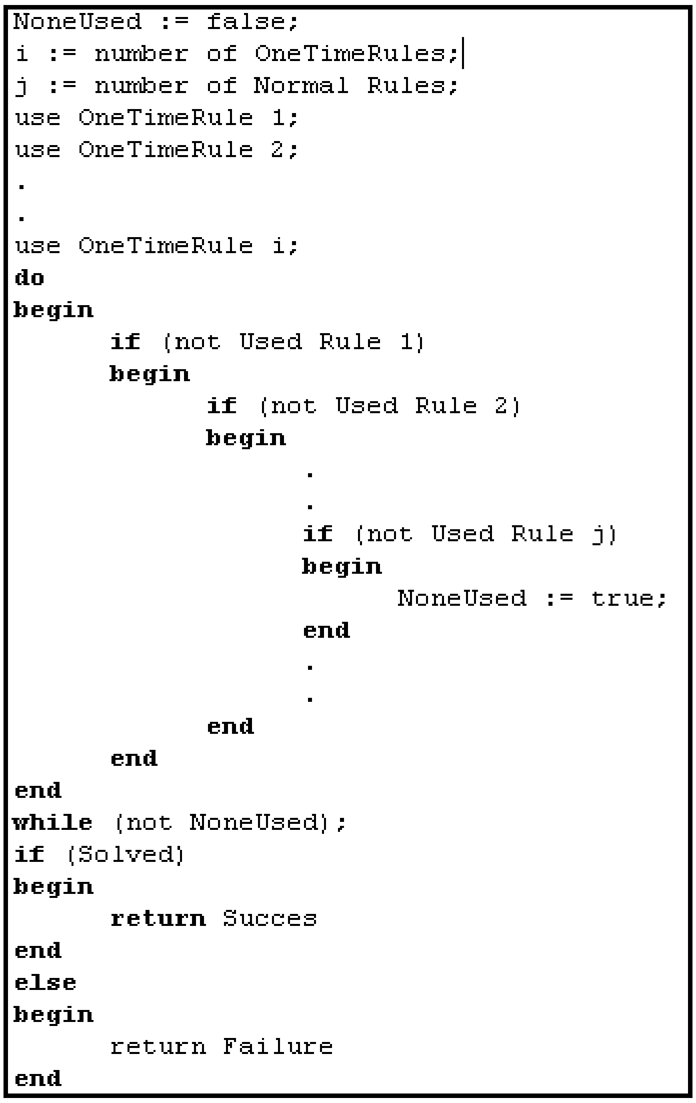
6.5.6. Rule-based

The rule-based approach depends on the knowledge that humans have on solving the puzzle, the capability of humans to put that knowledge down in words and the possibility to program those words into our mechanism. Although this doesn't seem like a major setback this proves to be harder in practice than it looks on paper.
Human knowledge is often stored into images that don't directly relate to a language based description and even when a description is found, it often turns out to be far too general; where humans can easily identify a special scenario they often lack the ability to define those situations beforehand and include them into the rule they provided [xi].
It is exactly those uncommon scenarios that make it hard to program a human rule. It can take a long time programming and is prone to have errors. Below is the rule-based layout as it was incorporated in 'Flats Unlimited' and which forms the core of this thesis. This is therefore the method that is used in most of the research.

6.5.6.1. Algorithmic approach

After entering the solving function a puzzle will be affected by the first set of rules. These rules affect the field of the puzzle only once. Thereafter it will enter the solving loop. The loop contains all rules that the programmer made available to the program. A puzzle leaves the loop only when it is solved or when the rules as a collective were unable to solve the puzzle. When the puzzle is deemed unsolvable by the system, this can be the result of an unsolvable puzzle (no do-able by humans) of an ambiguous one. The rules are sorted on perceived importance. When a rule is used, the loop starts over again with the first rules. When the last rule gets his try and fails the loop ends. A more precise formulation of the solving loop can be found in Figure 14.
The way the rules are evaluated is not trivial. Although the puzzles that are created by Puzzelsport programs allow a disjunctive rule-construction (see 4.1.3) pretty easily, the flats puzzle provides a more challenging basis for this approach. Not all rules are disjunctive and it will therefore matter in what way the rules are evaluated, since this will have its effects on the number of puzzles that are deemed solvable from a random set (see 6.5.6.17 and 6.6.1). Since we ultimately want to use the optimal order of our rules, we will research their respective solving quota during the analysis phase in 6.7.2.2.

The rules that were used in the application are described below. They are discussed in the order they are evaluated in the main solving loop during run-time.



```
NoneUsed := false;
i := number of OneTimeRules;
j := number of Normal Rules;
use OneTimeRule 1;
use OneTimeRule 2;
.
.
use OneTimeRule i;
do
begin
      if (not Used Rule 1)
      begin
            if (not Used Rule 2)
            begin
                  .
                  .
                  if (not Used Rule j)
                  begin
                        NoneUsed := true;
                  end
                  .
                  .
            end
      end
end
while (not NoneUsed);
if (Solved)
begin
      return Succes
end
else
begin
      return Failure
end
```

Figure 14. Main loop of the solving procedure

Next to the (temporary) field layout the algorithm also uses a so called scratch array that contains an entire 'scratch layer' for all possible flat heights. Such a scratch field has the same size as the field and contains for every flat and every location whether or not this flat can still be built here. The three options are:

- Possible
- Not possible
- Built

This scratch array provides the possibility to retain information derived from previous used rules and serves as the memory of our approach. Naturally, it is initialized with all cells on possible.

The scratch array can be seen as a 3 dimension representation of the puzzle under revision, where every floor contains the current solving information for one flat (see Figure 15).

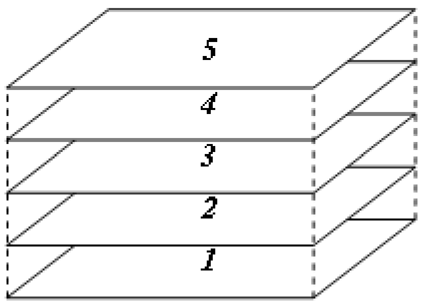


Figure 15. Nature of the scratch array

In the next examples we will use the following notation for the puzzle- and scratch fields:

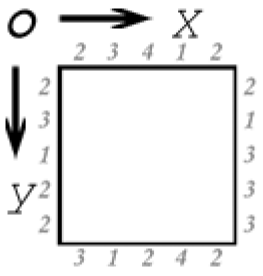


Figure 16. Origin and coordinates of puzzles

As origin for this matrix we use the top-left corner. From there, the coordinates *(x, y)* are given according to the arrows. Both these numbers are positive (see Figure 16).

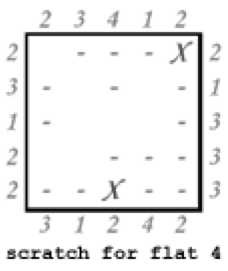


Figure 17. How the scratch-field is displayed

The Scratch-Fields are displayed as in Figure 17. As we can see below the puzzle’s field this is the scratch field for the flat with height 4. The empty spot in cell (1,1) denotes the fact that we have not yet got any information about that location for flat 4. If a large cross is displayed (like in cell (5,1)) we already know that this flat has been built on that location. A minus means that flat 4 cannot be built on that particular location.

Having established the preliminaries, we now describe the rules that were used in the above mentioned algorithm.

6.5.6.2. PlaceStair (One time rule)

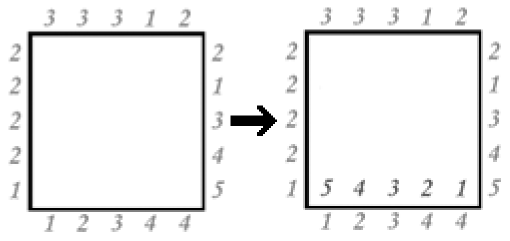


Figure 18. Using PlaceStair in row 5

This rule is one of the most obvious ones. It's only applicable in some puzzles, but when it can be used, it produces a considerable gain right at the beginning of the solving procedure.
The rule makes use of the special scenario where one of the views is *size*. If this is the case it is obvious that we can see all the flats in that row or column. They are thus neatly ordered from *1* to *size* (see Figure 18). The procedure is formulated more precise in Figure 19.

```
Do all directions
begin
      for i:=1 to size   (i->all rows and columns)
      begin
            if View in direction[i] = Size
            begin
                  place 1 to Size in row/column i;
            end
      end
end
```

Figure 19. Outline of the PlaceStair rule

6.5.6.3. PlaceN (One time rule)

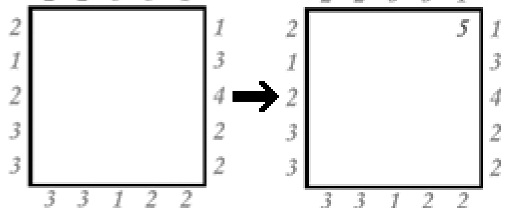


Figure 20. Using PlaceN on (5,1)

This rule is the one that novice puzzlers tend to discover first. It uses the fact that the largest flat can always be seen from all directions. If the largest flat is not the most outward placed flat in one direction you will always see more than one flat (the largest and at least the flat on the border of the field). If therefore the view from a direction is denoted as being *1*, this must mean that the largest flat (of size *size*) stands at the side (See Figure 20). The procedure is described in Figure 21.

```
Do all directions
begin
      for i:=1 to size
      begin
            if (View in direction[i] = 1)
            begin
                  Place N on side;
            end
      end
end
```

Figure 21. Outline of the PlaceN procedure

6.5.6.4. *Rule N+1 (One time rule)*

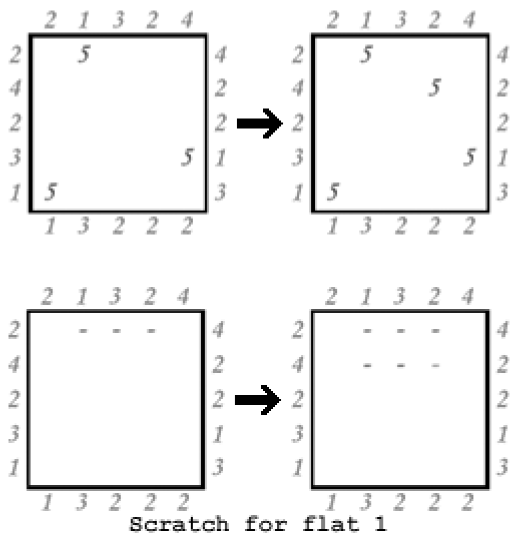


Figure 22. Using Rule N+1 in row 2

This Rule handles the situation where two opposing views summed equal *size+1*. This situation is usable because it denotes a double stair or pyramid shape in that column or row. This can be concluded from the following. The largest building can (and will) be seen twice in every row or column (i.e., once on every side); the smaller flats on the other hand can be seen only once or not at all, they are always

blocked to one side by the largest flat. This will result in a summed view of at most *((size – 1) + 2) = size + 1*.
Thus having a summed total of *size + 1* implies that all flats are being seen and that they are ordered in an ascending way from both sides to the top: the largest flat of height *size*. It is obvious that the PlaceStair rule in 6.5.6.2 is a more specific special case of this rule. PlacesStair however is both more powerful (more information is revealed) and less applicable (the situation does not occur that often as N+1).
Using Rule N+1 will result in two different advances:

- The puzzler can place the largest flat instantly. It can be placed in the *view[i]*d place from both sides, because of the pyramid shape of the row or column.
- The puzzler can also conclude where the smaller flats <u>cannot</u> be built. The smallest flat, for example, should always be placed at one of the first locations from either side. The other cells can therefore be scratched as possible locations. The second smallest flat cannot be placed at more than *2* locations from either side, and so on.

Both above mentioned advances are illustrated in Figure 22. The code layout for this rule can be found in Figure 23.

```
Do all rows and columns
begin
      for i:=1 to size
      begin
            if (Summed View in directions[i] = size+1)
            begin

                  Place largest flat on correct location;

                  for j:=1 to size
                  begin
                        if Distance to side > j
                        begin
                              scratch flat j in this location;
                        end
                  end
            end
      end
end
```

Figure 23. Outline of the Rule N+1 procedure

6.5.6.5. SideBlock (One time rule)

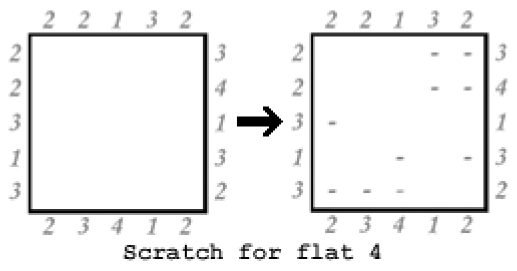


Figure 24. Using SideBlock

This is both a very simple and a very powerful rule. It is applicable in most puzzles and gives rise to a lot of advances in solving the puzzle. This rule is based upon the fact that large flats can make it impossible to see enough flats from some side. Building the flat of height *size* on the border of the field prohibits the view from that side to be any more than *1*; building the flat of height *size-1* there prohibits it to be more than *2*, and so on. Therefore, we can check whether a flat is too high to be placed on one of the outer cells of the field and if this is so, scratch that particular cell for that flat. The enormous results we can achieve in this way, early in the solution cycle, can be seen in Figure 24, the way this procedure works is laid out in Figure 25.

```
Do all cells
begin
      for x:=1 to size
      begin
            LeaveDistance := size - x + 1
            Do all directions
                  Distance := distance to current side;

                  if (View[direction] > Distance + LeaveDistance)
                  begin
                        scratch current cell for flat x;
                  end
            end
      end
end
```

Figure 25. Outline of the SideBlock procedure

6.5.6.6. SimpleScratch

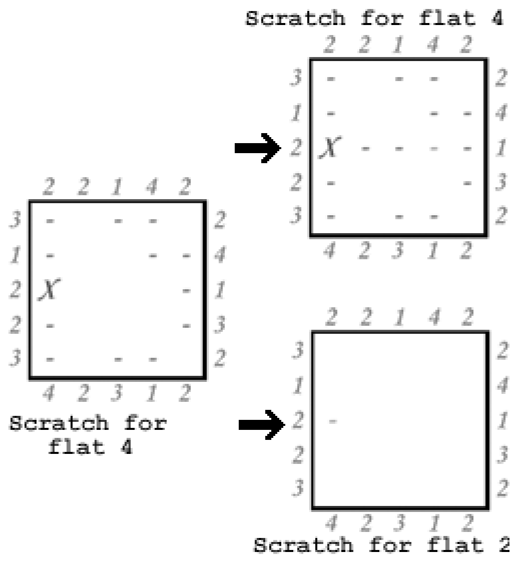


Figure 26. Using SimpleScratch after placing flat 4 on (1, 3)

SimpleScratch is the rule that handles the situation after a flat has been built. Its effects are very basic and are directly derived from the rules of the puzzle. In effect SimpleScratch scratches the cells for the flats that can no longer be built on that location. Its effects are therefore twofold:

- As soon as a flat has been built, it is no longer possible to build another flat on the same position. We can thus scratch this cell in the field for all other flats (see Figure 26, below).
- As soon as a flat has been built, it is no longer possible to build a flat of the same height in the same row or in the same column. All the cells in the corresponding row and column can therefore be scratched for the newly built flat (see Figure 26, above).

The code for the SimpleScratch procedure is described below.

```
Do all built flats
begin
      do entire rest of row and column
      begin
            scratch current cell for built flat
      end

      do all other flats
      begin
            scratch built cell for current flat;
      end
end
```

Figure 27. Outline of the SimpleScratch procedure

6.5.6.7. FillGaps

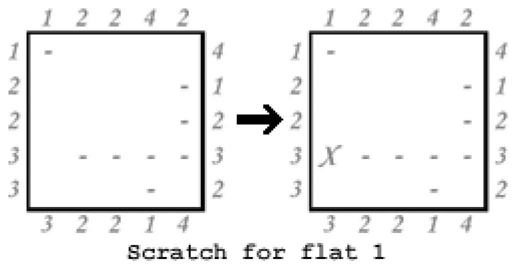


Figure 28. Using FillGaps in row 4

FillGaps is based upon the fact that a flat is placed once in every row and column. Therefore, if there is only one place available for a flat to be built in any row or column, it must be built in that particular location. This is the main way to come to building flats by scratching impossible cells. The effects of this rule can be seen in Figure 28. The underlying pseudo-code is given in Figure 29.

```
Do all rows and columns
begin
      for x:=1 to size
      begin
            if only one cell available
            begin
                  build flat x in that cell
            end
      end
end
```

Figure 29. Outline of the FillGaps procedure

6.5.6.8. BuildLocation

BuildLocation is in a way very similar to the above mentioned rule FillGaps. It also fills gaps in the scratch fields, but as the Fillgaps rule goes about it in a flat, 2-dimensional way, BuildLocation goes through the entire scratch array, from one flat level to another (i.e.: the vertical direction is displayed in Figure 15). In contrary to the above mentioned rule where a puzzler fills in the gaps from a flat's point of view, the BuildLocation algorithm tries to locate and fills gaps from a location's point of view.

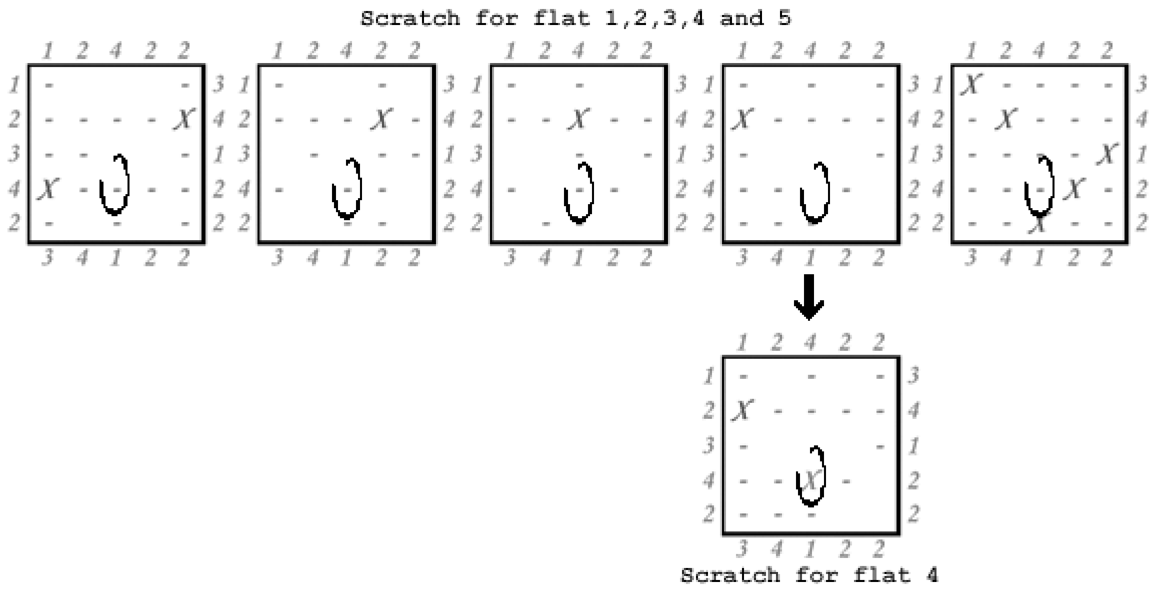


Figure 30. Using BuildLocation in (3,4)

Since no location can remain empty, if at any location all flats but one are impossible, this one remaining flat must be built there. As we can see in Figure 29, if flats 1, 2, 3 and 5 cannot be built on location (3, 4) flat 4 must be built at that location. The way the procedure works is laid out in Figure 31.

```
Do all locations
begin
      if all but one flats cannot be build
      begin
            place remaining flat at location;
      end
end
```

Figure 31. Outline of the BuildLocation procedure

6.5.6.9. DiffersOne

The DiffersOne rule is a collection of a number of sub-rules that all handle around a similar situation. They deal with the special scenario where a view from a direction is exactly *1* more than the current view from that same direction. A number of conclusions can be drawn from this situation depending on the other characteristics of the situation. Most of the situations handle around the largest and the smallest flat that have not been placed yet in that row or column.

Def 11). The 'largest (smallest) not placed' is the largest (smallest) flat that has not been built yet in a given row or column.

The different special situations that arise from the common one are described below.

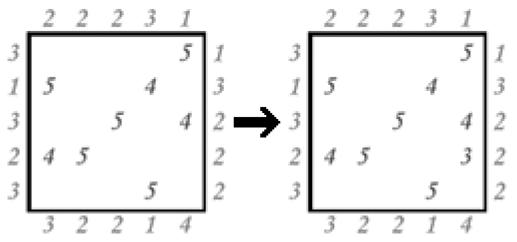


Figure 32. Using DiffersOne (1) in row 4 from the East

The first situation that arises is when the largest not placed cannot be 'hidden' behind a larger flat that has already been built, so that it cannot be seen from the given direction. If this is the case the largest not placed must be built at the first empty spot that is available. This can be concluded from the following: if the largest not placed will be seen (because it cannot be hidden) it will already fulfill the existing difference of *1*. If it is not built at the first available location, at least one other, smaller (because this flat is the largest not placed) building will also be seen because it will be built in front of the newly built flat. This would exceed the difference of *1 (i.e.: (view -1) + 1 + 1)* and make the field incorrect. How the procedure works can be seen in Figure 33.

```
Do all directions
begin
      possible := false;
      x:= location with > LargestNotPlaced;
      while x < size
      begin
            if possible build LargestotPlaced
            begin
                  Possible := true;
            end
            x:= location with > LargestNotPlaced;
      end
      if not Possible
      begin
            build LargestNotPlaced on first free location;
      end
end
```

Figure 33. Outline of DiffersOne procedure (1)

Opposite to the above mentioned, it also possible to conclude that the largest not placed should be hidden if it cannot be built on the first open location.

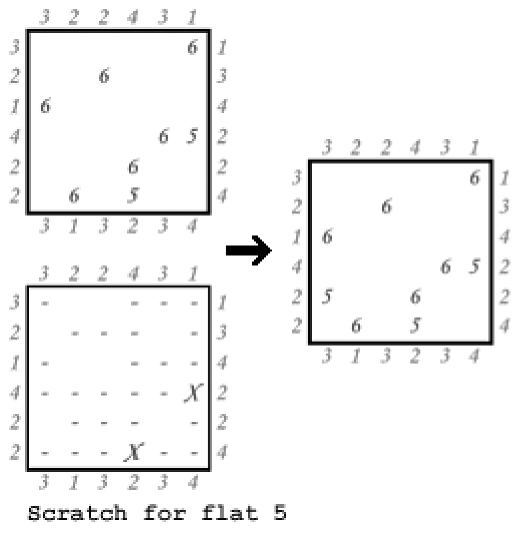


Figure 34. Using DiffersOne(2) in row 5 from the East

As can be seen in Figure 34, the flat with height 5 cannot be placed on location (6,5). Since this flat is the largest not placed viewed from the east on row 5 it cannot be built on location (5,5), for this would raise the number of viewed flats above the given view according to the above mentioned argument. Therefore, this flat must be hidden behind the flat with height 6. Since there is only one possible location left behind this flat, we can determine where flat 5 should be built.
It is also possible that there is more than 1 location where this flat can be hidden. If this is the case we can at least scratch all the locations for this flat from the second cell from the east to the cell before the first flat that is larger than the largest not placed.
The code below will describe this method more abstractly.

```
Do all directions
begin
      x:= first free location;
      if Scratch[LargestNotPlaced][x] = NO
      begin
            if Number of Hide possibilities > 1
            begin
                  for y:=first free to Field[x]> LargestNotPlaced
                  begin
                        Scratch[LargestNotPlaced][y] := NO;
                  end
            end
            else
            begin
                  build LargestNotPlaced on hide location;
            end
      end
end
```

Figure 35. Outline of DiffersOne procedure (2)

Just like conclusions can be reached about the largest not placed, it is also possible to learn about the smallest not placed. Examining the next example in Figure 36 would reveal new information about the possible locations for this small flat in row 5.

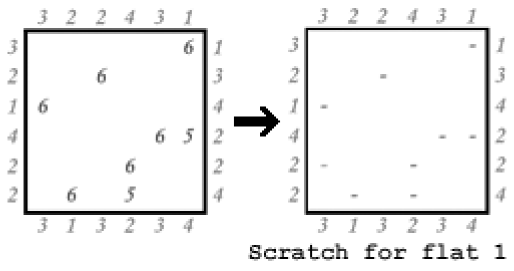


Figure 36. Using DiffersOne(3) in row 5 from the West

If the flat with height 1 would be built on location (1,5) it would already complete the view from the west on row 5. Since this building is the smallest not placed, any building in the following 2 cells would be higher and at least 1 of them would be seen as well, resulting in a larger view than the demanded western view *(i.e.: (view – 1) + 1 + 1)*. This would result in an erroneous field. The location (1,5) can thus be scratched for flat 1.

```
Do all directions
begin
      Found := false;
      Count = 0;
      x:= first free location + 1;
      while Field[x] is not > SmallestNotPlaced
      begin
            if Field[x] is not empty
            begin
                  Found = true;
            end
            else
            begin
                  Count = Count + 1;
            end
            x := next location;
      end
      if not Found and Count > 0
      begin
            scratch first free cell for SmallestNotPlaced;
      end
end
```

Figure 37. Outline of DiffersOne procedure (3)

A note of caution in using this deduction would be that the algorithm needs checking if there is no smaller flat already built that contributes to the current view and that will be blocked from viewing by the smallest not placed. If this is the case, this rule cannot be used. The algorithm also needs to check if there is at least one free cell after the first free cell. The Rule and this note of caution can be found in Figure 37.

6.5.6.10. DiffersIsPossible

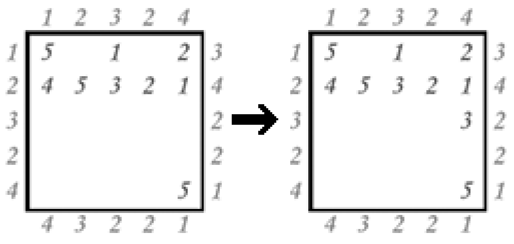


Figure 38. Using DiffersIsPos in column 5

Just like deductions can be made about the situation where the current view lacks just 1 visible flat it is also possible to reason about the situation where the number of free locations from a certain direction before the flat with size 'largest not placed + 1' is the same as the difference between the current view and the requested view; in other words: all the foremost empty locations must contain a flat that will be visible in the future, while keeping the flats that are already built visible..

Using this foundation, we can conclude that, if the smallest not placed cannot be hidden behind a larger flat, it should be built at the first free location. On the other hand, if the smallest not placed can be hidden, we can at least always scratch the locations between the first free cell and the first flat that hides the smallest not placed. In Figure 38 it is clear that the flat with height 3 must be placed in cell (5,3) to make sure that the requested view in column 5 can still be made, when placing the other flats. Because, if it was not placed there, its sight would be blocked by the flat with height 4. The code beneath demonstrates how this procedure works.

```
Do all directions
begin
      possible := false;
      x:= location with > SmallestNotPlaced;
      while x < Size      // Can SmallestNotPlaced be hidden?
      begin
            if possible build SmallestNotPlaced
            begin
                  possible := true;
            end
            x := x + 1;
      end
      if not possible
      begin
            build SmallestNotPlaced on first free location;
      end
      else
      begin
            for x := first free to Field[x] > SmallestNotPlaced
            begin
                  scratch SmallestNotPlaced on position x;
            end
      end
end
```

Figure 39. Outline of the DiffersIsPos procedure

6.5.6.11. Blocks

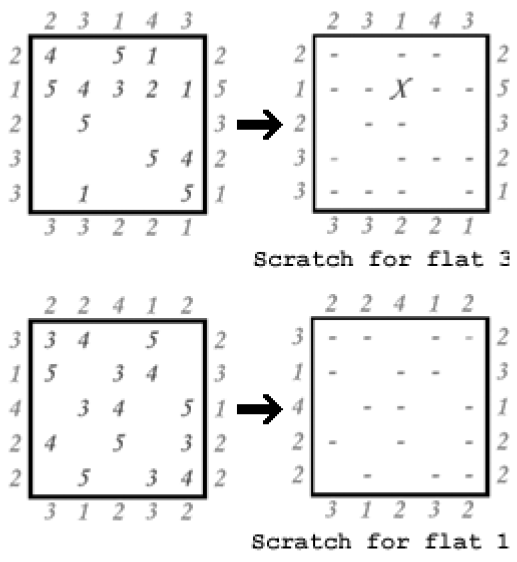


Figure 40. Using Blocks

The Blocks Rule is an extension of the SideBlock rule in 6.5.6.5. It is in fact the application of that rule in a context where some locations are already filled. The rule consists of two parts:

- LargeBlocks is the part of this rule that handles the situation where building a high flat in some location would make it impossible to reach the demanded view from a direction. As can be seen in Figure 40 (above), placing the flat with height 3 in location (1,4) would fix the view in row 4 from the west at 2 because no higher building can be built in the remaining 2 locations (3 is the largest not placed). This flat can therefore not be placed on that particular location.
- SmallBlocks is the part of this rule that deals with small flats completing the view that leave open spaces that would create a view-overflow. As can be seen in Figure 40 (below) flat 1 cannot be placed in location (5,1) because that would complete the view from the north in column 5 and still leave a location that can only contain a larger flat that would also be seen. Flat 1 can therefore not be placed in that location.

The procedure is described in Figure 41.

```
Do all directions
begin
     Differs := View - CurrentView;
     Free := first free location;
     for x:= 1 to Differs - 1 + Free;
     begin
          Scratch LargestNotPlaced on location x;
     end

     Positons := positions to first filled location;

     if Positions > Differs
     begin
          Scratch position 1 for smallestnotplaced;
     end
end
```

Figure 41. Outline of the Blocks procedure

6.5.6.12. SuperFillGaps

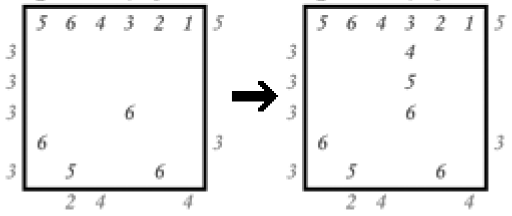


Figure 42. Using SuperFillGaps

As the name claims, this rule is an extension to the existing rule FillGaps in 6.5.6.7. It has the purpose of filling gaps that can only have a certain filling. If we take Figure 42 as example we can see in column 4 from the north that having the flat with height 3 at the front and only two spaces free before flat 6 (the maximum flat height for this puzzle) only leaves us with the possibility of building flat 4 and 5 in between in ascending order to complete the view. Other (lower) flats would have been hidden behind flat 3 and disappeared from our view from the north. The code for this procedure can be found in Figure 43.

```
Do all directions
begin
      Differs := View - CurrentView;
      Positions := free locations before heighest flat;
      if heighest flat is built and
         Positions = Differs and

         // Difference between heighest flat and flat
         // on first location must be 1 less than the
         // number of positions between them
         First Field - Heighest flat -1 = Positions
      begin
            for x:=Field[1] to Heighest - 1
            begin
                  Next Free Cell:=x;
            end
      end
end
```

Figure 43. Outline of the SuperFillGaps procedure

6.5.6.13. Rule N-1

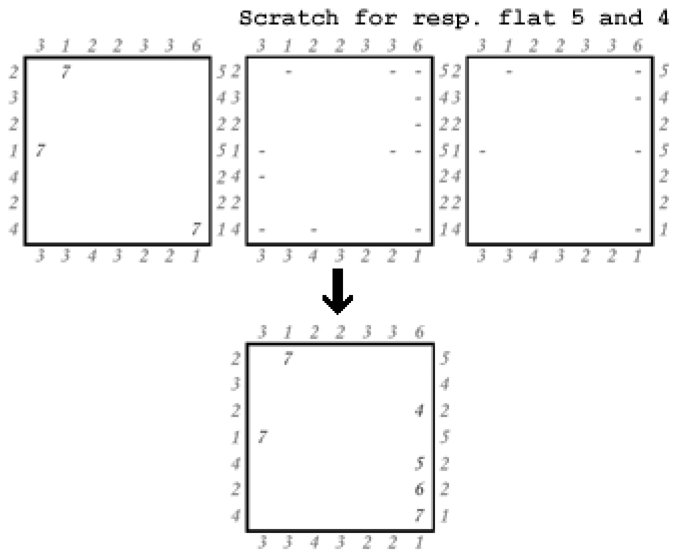


Figure 44. Using Rule N-1

The Rule N-1 is actually a pretty strong one, but is placed at the near end of the solving loop. This is done from a performance point of view because of the quantity of its application possibilities. It deals with a special case scenario that appears not that often in a randomly generated puzzle and therefore would often mean an unnecessary performance setback and loss of time. When it is applied, however, it can lead to surprisingly good results.

When a certain view from a certain direction is exactly *size* – 1 it is possible to deduct what form the flat layout should take. It resembles the stair layout greatly with exactly one difference: one of the flats is built one or more positions to the rear. In that way only the moved building will disappear from the view and leave all the other buildings visible. The conclusions we can draw from this can take many forms, but the most common one is depicted in Figure 44, that also shows the power of this rule by automatically placing 3 flats in one rule. As we can see from the middle scratch-field on the top level, flat 4 cannot be built on its own location (4th position from the north), which means that it is either the one flat that has 'bubbled' back to the rear or is one of the flats that is put 1 position to the front to hide the bubbling flat from view. If the first was the case, flat 5 would be built at the fourth location to hide the moving flat 4.

```
Do all directions
begin
      if view[direction] = size -1
      begin

            scratch positions <= y - 2 for all flats y;

            for x:=1 to size
            begin
                  if flat x cannot be build on position x
                  begin
                        if flat x+1 cannot be build on position x
                        begin
                              build flat x on position x - 1;
                              build all flats > x on own position;
                        end
                        if flat x cannot be build on position x-1
                        begin
                              build all flats < x on own position;
                        end
                  end
            end
      end
end
```

Figure 45. Outline of the Rule N-1 procedure

Since we can see that this is also impossible, the second possible situation must be true. So we can build the fourth flat on the third position hiding flat 1, 2 or 3.
Also, since the flat with height 5 could not be built on the fourth position we know that the moving flat stopped bubbling on the fourth position after the flat with height 4. We can thus conclude that flats 5, 6 and 7 must be at their own position and place them in the field.
In Figure 45, the code for Rule N-1 is laid out.

6.5.6.14. Try

The try rule is the strongest rule of all, but also the least usable by humans. It deals with the situation where all possible outcome situations for a certain row are evaluated and conclusions are drawn from them. They can for example yield that in all the possible outcomes the flat with height 3 was never built in location (3,1) (See Figure 46).

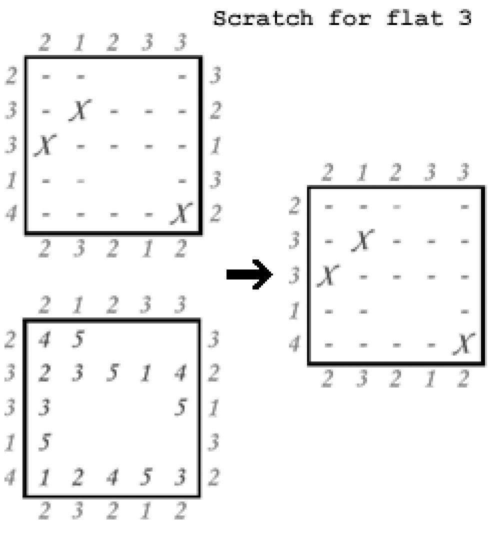


Figure 46. Using Try

Try uses the fact that all permutations of the collection *{1 ,…, size}*. are exactly the forms a row or a column in the flats puzzle can take. We will use this principle and try to extract scratch information from them. The algorithm tries to falsify most of the permutations per row or column in order to leave only a few possible combinations. For example: in the above Figure, the permutation 4,5,1,2,3 is impossible for the first row because the flat with height 3 cannot be built on position 5. As is the permutation 5,4,2,3,1 because flats 5 and 4 are already placed in another location. The first step of

the try procedure is therefore the elimination of permutations based on the current scratch array.
After permutations have been eliminated based upon the scratch fields, we can also eliminate permutations because they do not comply with the demanded view; this is step 2 of the try procedure.
After all non-usable permutations have been eliminated the program checks if some flats do not appear in some cells in the remaining permutations, meaning: in all remaining positions a certain flat *x* never appears on a certain position *y*. If this is the case, that particular flat *x* can be scratched on that location *y*. The conclusions we draw and the effects they have on the scratch array is step 3 in the Try process.

```
Do all rows and columns
begin
      Do all permutations of size size
      begin
            for x:=1 to size
            begin
                  if scratch[permutation[x]][x] = NO
                  begin
                        Falsify current permutation;
                  end
            end
      end
      for y:=1 to size
      begin
            for z:=1 to size
            begin
                  Found[z] := false;
                  Do all remaining permutations
                  begin
                        if permutation[y] = z
                        begin
                              Found[z] := true;
                        end
                  end
                  if not Found[z]
                  begin
                        scratch[z][y] := NO;
                  end
            end
      end
end
```

Figure 47. Outline of the Try procedure

The above mentioned steps will be taken for all rows and columns.
Using the rule can create problems for humans, both because it is hard to create all permutations in ones head and to remember which ones have been falsified yet. A difference is made between using Try

over a few locations and using Try over many locations (Try Large). The border has been drawn between 3 and 4 open locations left. This is done to enable the puzzle's creator to keep puzzles solvable by humans.
In Figure 47 the pseudo-code for this procedure is shown.

6.5.6.15. Simple Try

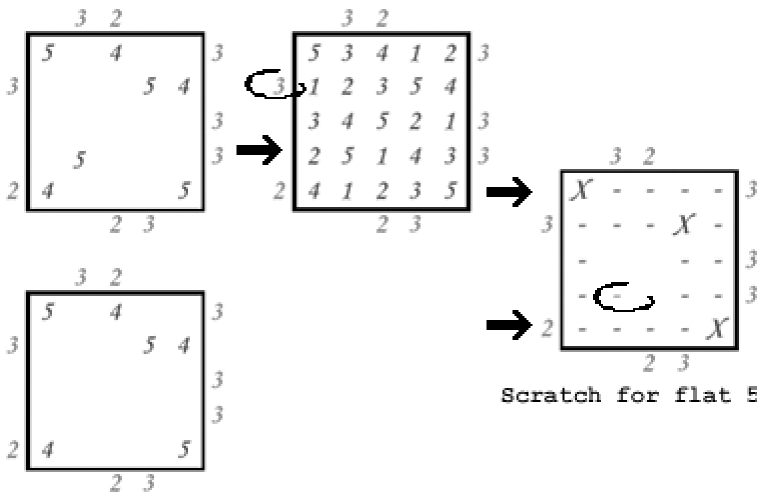


Figure 48. Using SimpleTry

SimpleTry offers a limited backtracking solution to the situation where no other rule is able to go one step ahead in the solving procedure. We will take Figure 48 as example to explain the way this function works, which is in fact really simple.
The figure on the left side below is the puzzle as we now know it. As we can make out quickly, the flat with height 5 in column 2 can only be built on position 3 and 4. We will approach the situation as follows:

1. Fill in the flat with height 5 on one of the two available positions.
2. Check if the new puzzle situation solved by the remaining other rules creates a puzzle situation that contradicts the characteristics. Contradictions that are detected in this respect are:
   - A certain row or column doesn't hold for the Latin square principle.
   - A current view in a certain column or row is already larger than the requested view.
3. If a contradiction exists (like there is on the second row of the above middle puzzle) scratch the flat with height 5 for that

position. If no contradiction exists for this puzzle try the other free location in that row or column.

4. If no contradictions arise it may be due to puzzle ambiguity, but this is not necessarily so. It can also be the effect of un-solvability or flaws in the rule-based approach. Therefore, no conclusions can be drawn from this situation.

NOTE: we limited this rule to the situation where no more than 2 locations are still possible for a certain flat in a row or column. The rule in itself is obviously already stretching the limits of human capability, let alone the situation where 3 different possibilities should be totally worked out.

NOTE: SimpleTry will **not** use itself when trying to solve the adapted puzzle. This is incorporated into the rule for the following three reasons: first, it would be way too hard for a human to apply backtracking within a backtracking method, second, it would mean a unreasonable performance setback and third, it would be un-doable to deliver a solving-path after the puzzle is solved because of its complexity.

Below is the algorithm described in pseudo code:

```
do all directions
begin
    for CurrentFlat := 1 to Size
    begin
        if Number of free positions = 2 then
        begin
            for CurrentPosition := 1 to 2
            begin
                Fill in CurrentFlat on CurrentPosition;
                Try to solve new puzzle situation;
                CheckContradiction(Con); (checks contradictions)
                if con = true then
                begin
                    Scratch CurrentFlat on CurrentPosition
                    CurrentFlat := Size;
                    CurrentPosition := 2;
                end
            end
        end
    end
end
```

Figure 49. Outline of the SimpleTry procedure

6.5.6.16. Rule Relations

As it was pointed out a number of times in the past 14 sections, some rules are special cases of one another or include the other as a whole. We will now deal with this situation and try to establish an analytical basis that points out how the various rules are interrelated. It appears to be the case that certain rules, on their own or combined with another rule contain another rule. This means that the conclusions drawn from these rules in the third rule's situation are exactly the same as the third rule's conclusions. For example: using only SideBlock and FillGaps in a PlaceStair situation would also fill the entire row, exactly as PlaceStair itself would do it. The containment is defined as below.

Def 12) A rule is contained by a number of other rules when the other rules combined have the same effects as the original rule in any situation where the original rule was applicable.

The fact that some rules contain others gives rise to the existence of so called elementary rules:

Def 13) Elementary rules are rules in a rule based system that are not contained by any combination of other rules.

We can see in Figure 50 how the rules in our rule-based system are interrelated. The elementary rules are printed bold and, as is clearly visible, contain all other rules. They are therefore able to solve exactly the same puzzles as the entire set of rules.

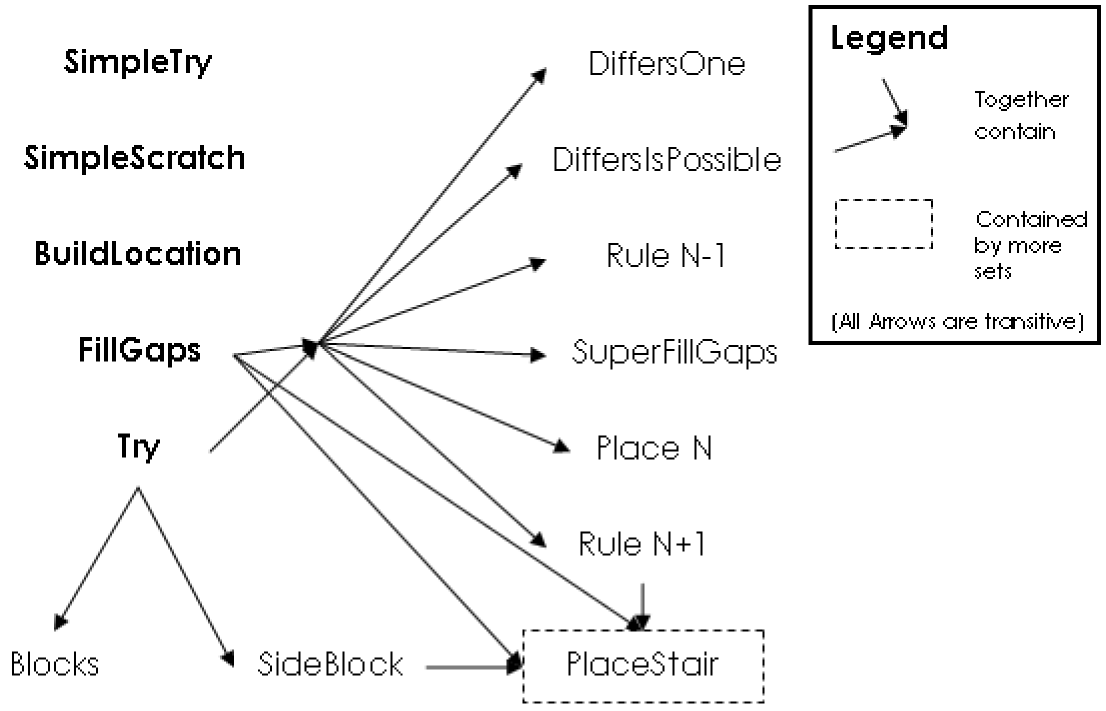


Figure 50. Relations of the rules in the rule-based system

Since there are only 5 elementary rules, it is reasonable to ask whether all the other rules are in fact necessary to include in the system. The answer to this is definitely yes; the rules at 'the end' of the line are the rules best known and best applied by humans and should therefore never be neglected.

It is also interesting to know how many of the elementary rules are actually necessary to include in your rule set to be able to solve **any** puzzle and why. In trying to answer this we will only discuss the 5 elementary rules (all other rules are contained) minus SimpleTry, because this is not really rule-based in nature (it is backtracking based) and can therefore never be necessary in a rule-based approach.
This leaves us with the following 4 rules: SimpleScratch, FillGaps, BuildLocation and Try. We will follow a deductive approach to establish what rules are necessary. That way, we will also know why the necessity holds for a certain rule.
First of all, the rules that deal with the basic game rules of the flats puzzle are obviously considered necessary. In this case those game rules are:

- One building in every cell
- Comply with the Latin square principle
- Comply with the view characteristic

As mentioned in 6.5.6.14 Try mainly falsifies its candidates on an incorrect current view in a certain row or column. Since no other elementary rules deal with the view characteristic, this rule is necessary.
The two other basic puzzle requirements are dealt with in the SimpleScratch rule. As was described in 6.5.6.6 SimpleScracth scratches the rows and columns for a flat that has already been built in there and scratches the other flats for the location that has currently be filled. Since this rule is the only remaining basic rule that deals with these facts, it is also necessary.
Now all basic rules have been dealt with but since none of these rules ever actually builds a flat, but only forbids them on certain locations we need at least one other rule to do this job. As far as basic deductions are concerned either of the two remaining rules are good enough. As was mentioned before (see 6.5.6.8) both rules are in essence the same and differ only in the direction they operate (in the flat or in a 3 dimensional way). We can now come to the following conclusion about rule necessity: these rules are necessary to be able to solve **any** puzzle:

$$Try \wedge SimpleScratch \wedge (BuildLocation \vee FillGaps)$$

It is to be expected that including FillGaps will solve more puzzles than including BuildLocation. I.e. most of the rules contained in Try handle around a flat situation and the progress they reach will therefore inherently be more usable to FillGaps than to BuildLocation. This hypothesis will be experimentally tested in 6.7.2.1.

6.5.6.17. Performance

Given that the rule-based approach is based upon certainties and there from concludes what action to take, we eventually reach a certain solution. This leaves no possibility for another solution that realizes the characteristics. This certainty-principle thus provides the demand that no unsolvable or ambiguous puzzles will be solved.

The Rule-based approach turns out to have some very attractive properties, a major setback and some mild disadvantages.
The greatest objection to using a rule-based approach would be that all the knowledge about the puzzle's domain has to be already there, in the mind of the algorithm programmer. Knowledge that he or she doesn't have about solving the puzzle will not be there in the program. Next to the need for existing human knowledge, it takes a lot of time to program the rules we humans already know into the application.
A slight disadvantage would be that experiments showed that not all the puzzles from the Puzzelsport magazine were solved by the program, when not including the backtracking rule SimpleTry. 83.5% of the magazine puzzles, however, were even then still solved by the program, which is still a very high percentage. When SimpleTry is included in the used set of rules, all published puzzles encountered were solved by our approach.
Next to the disadvantages there are three major advantages to the rule-based approach. First, the program can offer complete trace-ability to the puzzles creator and shows which rules are used. The creator can see instantly that there is a way to solve the puzzle, what one of the possible solution paths is, which rules are used and which ones are not. He can also check where solving stops when trying to solve an unsolvable puzzle. This way the creator is able to extend his knowledge about the puzzle and can use complete information on every step of the solving procedure.
The second advantage is the flexibility. This can be used mainly during the generation phase (see 6.6). The creator can indicate whether or not he would like to use certain rules in his puzzle to be. In this way, he can for example decide not to use the Try Large rule

to make the puzzle easier for humans or decide to use just a small subset of the rules to test the puzzler's in-depth knowledge of the puzzle.

The last advantage is that the known solving procedure and rule statistics can be used during the analysis phase. Certain rules may introduce fun or difficulty or decrease those dimensions. These aspects of the rule-based algorithm will be discussed in 6.7.3.

It would be interesting to know what rules are being used often and which rules will be used seldom and thus establish their worth to the program or the puzzler. Questions around this topic will be dealt with in 6.7.2.3.

Overall the rule-based approach is a time consuming but powerful way of solving a logical puzzle. If solving 100% of the solvable puzzles is important one should probably not use this algorithm, because the creator inherently doesn't possess all the knowledge himself. If generating puzzles is more important or user feedback plays a large role, this algorithm will serve the user very well.

Some more issues concerning performance will be discussed in the analytical section (see 6.7.2).

6.6. Puzzle Generation

6.6.1. Approach

As mentioned in 3.6 generating a puzzle needs to be nothing more then coming up with candidate puzzles until the creator finds a puzzle that meets his demands. The most important demand that goes for every puzzle instance is obviously the fact that it needs to be solvable. Having developed a solving mechanism in the last section, generating a puzzle can be accomplished reasonably easy.

The approach we will take includes five steps.

1. Build a random puzzle of size *size*.
2. Try to solve the puzzle.
3. If the puzzle is unsolvable go to step 1.
4. Solve the puzzle and show the way this is done.
5. Save the puzzle.

This can be easily incorporated when the function that checks if the puzzle can be solved returns a Boolean value. We now produce a loop that incorporates step 1 to 3 using the previously mentioned function. This generation function would look as in Figure 51:

```
Puzzle := GeneratePuzzle;
Temp := SolvePuzzle(Puzzle);   //(boolean->is puzzle solvable)
while Not Temp do
begin
      Puzzle := GeneratePuzzle;
      Temp := SolvePuzzle(Puzzle);
end
DisplaySolveInformation(Puzzle);
SavePuzzle(Puzzle)
```

Figure 51. The puzzle generation function

6.6.2. SideStrip©

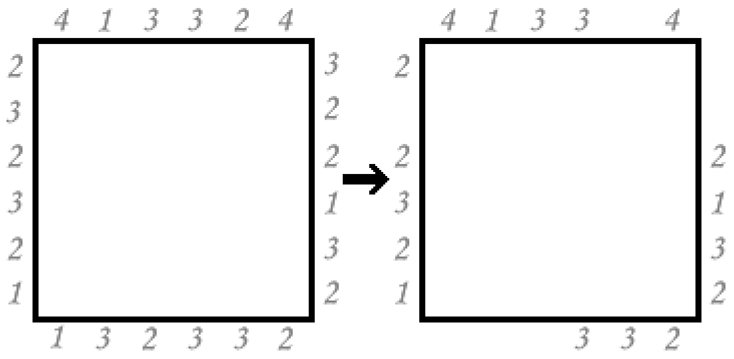


Figure 52. SideStripping a flats puzzle

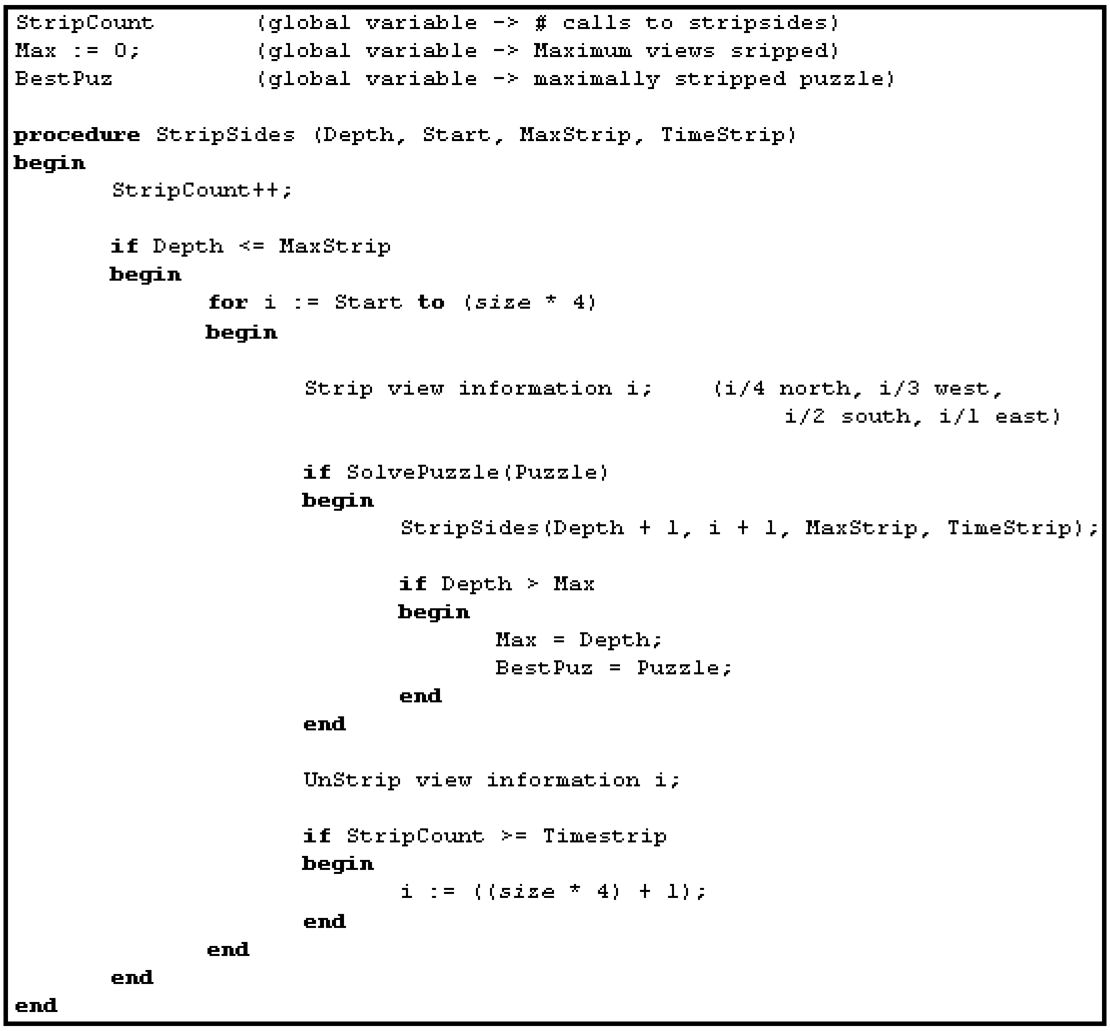
We can integrate a step 3a in the approach we took in the above section that will strip the puzzle's characteristics according to what was mentioned in 3.7. As noted, stripping the puzzles characteristics can make a puzzle more interesting or harder than its un-stripped counterpart.
Of course it would be ideal to do the stripping during the creation phase, but no efficient way is known to develop an algorithm that combines the solving and editing procedure in such a way that solvable puzzles are found reasonably easy and the characteristics are stripped simultaneously.
The best approach would therefore be to generate a puzzle, that we know to be solvable and then strip its characteristics in such a way that it remains solvable throughout the procedure. The obvious way to come to such a system is to strip a small part of the characteristics (for this puzzle one of the views), to check if it is still solvable and go on with the next part if it is. If it is not we backtrack to the last good known configuration of the characteristics and try another part for removal. The obvious way to approach such a challenge is to use a recursive function.
It is however necessary to build in the possibility of stopping the recursive function before it ends. The recursive loop would take far too

long to let the puzzle's creator wait for the stripping to complete. Furthermore, it is far from certain that a lot of progress will be made after the first large improvements. Research on this will be laid out in 6.6.2.1.

There are a few settings that will influence the way SideStrip works:

- Maximal Stripping time: this actually is the amount of recursive function calls that will be done before the function terminates and returns the most stripped result up to that moment. This is done for above mentioned reasons.
- Maximal Stripping amount: this determines the maximal amount of characteristic information (here: views) that should be stripped. This is done to enable the creator to change the puzzle's difficulty, fun and the way it looks.

Both will stop the function if the desired level is reached but inspect a different variable. The way SideStrip works is disclosed in Figure 53.



```
StripCount        (global variable -> # calls to stripsides)
Max := 0;         (global variable -> Maximum views sripped)
BestPuz           (global variable -> maximally stripped puzzle)

procedure StripSides (Depth, Start, MaxStrip, TimeStrip)
begin
       StripCount++;

       if Depth <= MaxStrip
       begin
              for i := Start to (size * 4)
              begin

                     Strip view information i;     (i/4 north, i/3 west,
                                                        i/2 south, i/1 east)

                     if SolvePuzzle(Puzzle)
                     begin
                            StripSides(Depth + 1, i + 1, MaxStrip, TimeStrip);

                            if Depth > Max
                            begin
                                   Max = Depth;
                                   BestPuz = Puzzle;
                            end
                     end

                     UnStrip view information i;

                     if StripCount >= Timestrip
                     begin
                            i := ((size * 4) + 1);
                     end
              end
       end
end
```

Figure 53. Outline of the StripSides procedure

6.6.2.1. SideStrip findings

After reviewing some of the puzzles produced using the SideStrip method some noticeable puzzle properties emerged.

There seems to be a trend in how many views can be maximally stripped using the rule-based algorithm and the complete backtracking SideStrip procedure. Independent of the size of the puzzle it turns out that the maximal number of views that can be scratched is approximately 14. It may seem unlikely that a puzzle of size 7 can lose as much views as a puzzle of size 5 although it has 8 (28 – 20) more views. However it is not: solving the puzzle using the rule-based approach depends for a crucial part on the usage of low or high flats, so having a large gap between them (i.e.: 7-1 = 6) makes a puzzle much less likely to be solvable. The puzzles that have those large gaps would therefore need to supply more information to the puzzler. So although the larger puzzles have more information that could be stripped they need equally much more information to retain their solvability. However, the fact that the number of stripped views is approximately around 13 or 14 for all sizes still seems a coincidence.
Next to the maximal stripping amount, we can also look at the optimum we can reach in a near optimal time-span. It turns out that setting a maximum stripping amount of 10 or 11 will usually complete the SideStrip procedure in a time under 1 second (on an AMD2400+ machine). The difference between the maximal amount and the fast result is therefore very reasonable (especially in the case of the 7 sized puzzle, where the relative difference is only 10%). It is therefore strongly recommended to set the maximal stripping amount to this number to achieve a reasonably efficient creation time whilst ensuring a very reasonable result. The backside to this approach would be that this would often create a puzzle that becomes ‘out-of-balance’ from an esthetic point of view; the SideStrip function first strips all of the north, then all of the west and so on. Setting a maximum of 10 or less would often only strip the first ‘visited’ directions. Restructuring the for-loop could deal with this problem.

6.6.3. Puzzle generation findings

The puzzles that are generated by our generation sequencer turn out to be remarkably hard. Setting a somewhat experienced puzzler to the task of solving some of them would give him a huge task. Apparently the computer produced puzzles can be harder than just the sum of the difficulty of the used strategies. As was mentioned before, the computer has an edge in solving puzzles, because it can 'see all the angles' (see 3.5). This can be an advantage of the computer as a solving engine but poses a problem for the computer as creator; if seeing all the angles is used to solve a puzzle, it can as well be necessary to see all the angles, which could mean that the puzzle would become nearly unsolvable or at least very hard. It will therefore remain up to the puzzle's creator to select the puzzle for publication or not. He can be helped in making the right decision by the proposed analysis system that gives the created puzzle a difficulty rating among other ratings (see 6.7.3.3).

Using the rule-based approach for the generation procedure it can take certain time for the system to discover a solvable puzzle. Of course, this depends on what rules were selected to be used in the solving procedure. We take a closer look at this for two selections of rules: the situation where all rules are selected and the situation where only the TryLarge rule and SimpleTry are not (all times are measured on an AMD2400+ machine). The average time to 'discover' a solvable puzzle (time) and the average number of times the generation loop is taken (tries) for both rule sets are described in Table 3.

Table 3. Experimental data for RuleBased puzzle Generation over 10,000 generated puzzles

Size	All rules selected		All rules but TryLarge and SimpleTry selected	
	Avg Tries	Avg Time(sec)	Avg Tries	Avg Time(sec)
5	1.321	0.092	1.776	0.025
6	11.550	3.79	26.213	0.480
7	85.097	180.480	391.546	9.826

From the experimental data we can conclude that the generation of solvable puzzles can be really fast but grows exponentially slower as the size increases.

The number in the second column of Table 3 can be used for more purposes than just establishing the speed and efficiency of the generation approach, but also encapsulates the percentage of random puzzles that this approach deems solvable. These are the following percentages:

Table 4. Percentage of puzzles solvable by the rule-based approach using all rules

Size	Percentage of solvable puzzles
5	75.7 %
6	8.7 %
7	1.2 %

We can conclude that the 5 sized puzzle has a high solvability rate, but that solvability of a random build decreases exponentially with its size.
We can also make a comparison between the proposed rule-based system and the backtracking approach. Although the backtracking approach is not very suitable for solving (and therefore generation), we can use the approach to determine how the rule-based approach is performing. In Table 5, Figure 54 and Figure 55 we can see the experimental results for a backtracking approach in comparison to the rule-based results.

Table 5. Experimental data for Backtracking puzzle Generation over 1,000 generated puzzles

Size	Avg Tries	Avg Time(sec)	Percentage of solvable puzzles
5	1.315	0.008	76.0 %
6	9.990	0.730	10.0 %
7	84.000	897.000	1.2 %

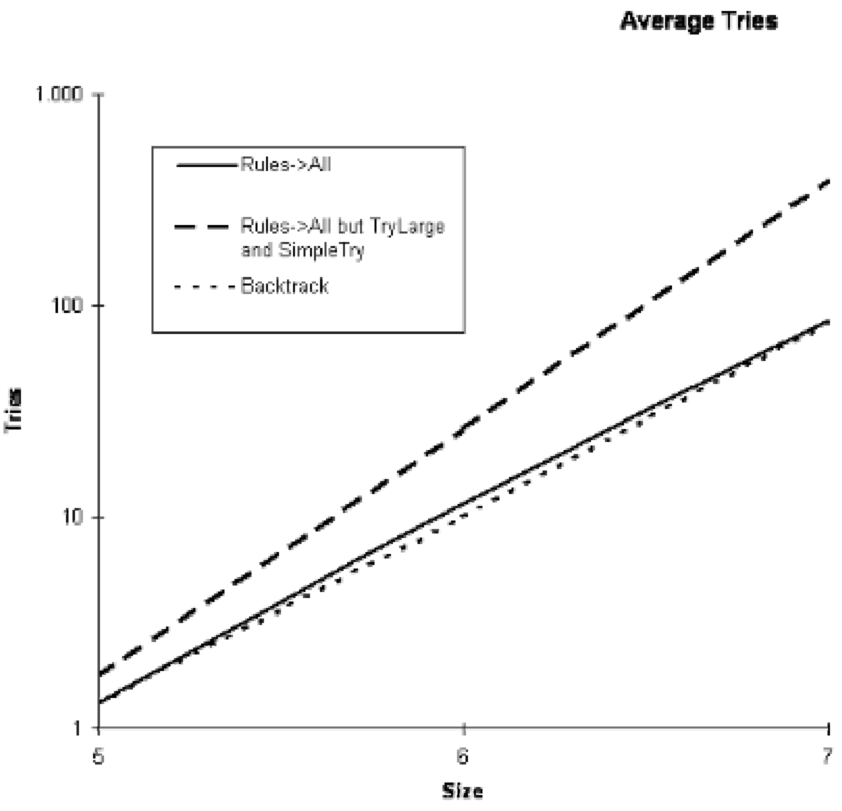


Figure 54. Difference in tries between Rule-based and Backtracking

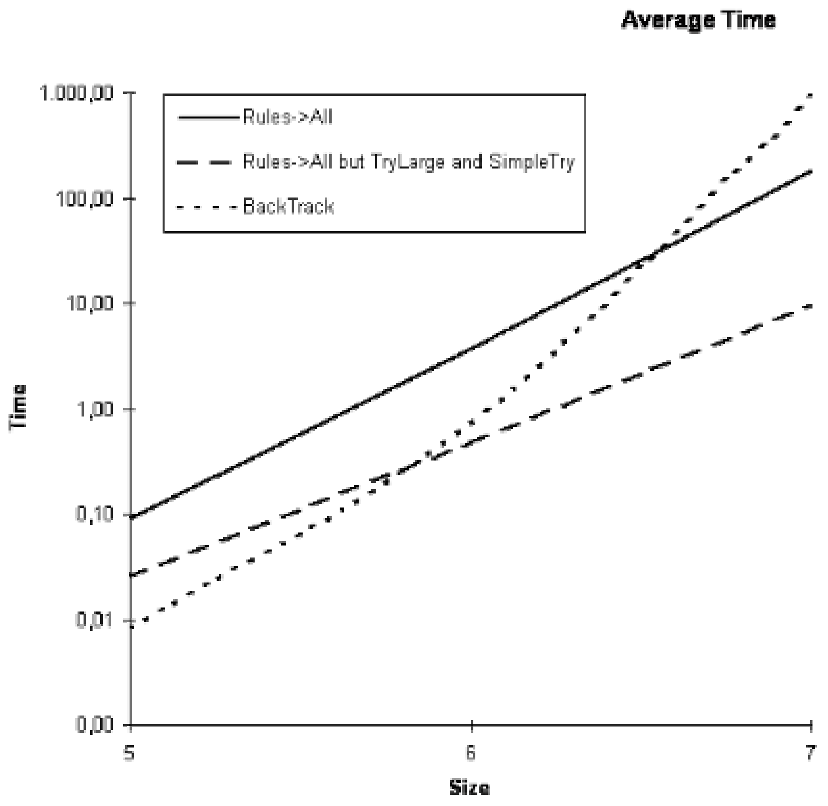


Figure 55. Difference in time between Rule-Based and Backtracking

As can be seen in the above figures, there is only a slight difference in the number of puzzles that are deemed solvable by both strategies, while there is a large difference in time for the larger puzzles (note the logarithmic scale). We can therefore conclude that the rule-based strategy proves better in performance while its delivered results differ only marginally from the unambiguous puzzles generated by the backtracking procedure (these puzzles don't even necessarily need to be solvable).

Below are the percentages of unambiguous puzzles that are categorized as solvable by the rule-based system.

Table 6. Percentage of unambiguous puzzles deemed solvable by rule-based system in comparison to the backtracking approach

Size	Percentage
5	99.5 %
6	86.5 %
7	98.7 %

As was mentioned before, these results don't necessarily dictate a lack of detection but will also include the standard difference between unambiguity and solvability. Which part of the percentage is caused by what reason, is unknown.

6.7. Puzzle Analysis

6.7.1. Test set generation

In the following analysis phase we will use certain tools that employ a test set. The ultimate test set would contain approximately 1000 puzzles to ensure a valid statistics outcome, extra information about the way the puzzle is solved to come to analytical conclusions about its properties and user opinions about the puzzle to include a statistical means of abstracting subjectivity.
Unfortunately such a test set is unavailable for the Flats puzzle and we can therefore only meet the first two demands: a large number of (solvable) puzzles and extra information about its solvability. Although this makes certain approaches for analysis impossible, there are still a lot of possible puzzle properties to research.
We can generate the following test sets:

- Random test set (Try on/off, TryLarge on/off, SimpleTry on/off, SideStrip on/off, random MaxStrip, random Size, both solvable and unchecked).

- Size ordered test set (same as random but with evenly divided puzzle sizes).
- 'As Set' test set (random solvable puzzles with user given size, rule set and stripping information).

6.7.2. Solve method analysis

6.7.2.1. Elementary rules and FillGaps vs. BuildLocation

To test our hypothesis on how the rules in the rule-based system are related we perform the same test as in 6.6.3 with just the elementary five rules selected and will generate 10,000 puzzles using FillGaps or BuildLocation as necessary rule (see 6.5.6.16).
Next to the common test on equality between the entire rule set and the elementary one we can also test our hypothesis in the necessity issue between BuildLocation and FillGaps. As we claimed in 6.5.6.16 the nature of the other rules could give FillGaps an edge because it operates on the same (flat) dimension.
Below are the test results for elementary puzzle generation.

Table 7. Experimental average number of tries using different rule sets over 10,000 generated puzzles

Size	All rules selected	All elementary rules selected	Only necessary rules (BuildLocation)	Only Necessary Rules (FillGaps)
5	1.321	1.319	1.400	1.410
6	11.550	11.683	12.81	12.73
7	85.097	84.898	94.61	94.49

When taking the standard deviation into account, we can learn from Table 7 that we will indeed reach the same quality levels when using the elementary rule set as we reach using the full arsenal of solving procedures. The test results create a solid basis under the theory of rule-relations.
The match between BuildLocation and Fillgaps ended in an unexpected draw. Both perform about evenly good during the shakedown test. Why the apparent edge Fillgaps has doesn't cash itself out is unclear but undermines the thought of an inherit 'flat' nature of the solving routines under consideration. Effects obviously take as much place between flats as they take place on one flat's field information.

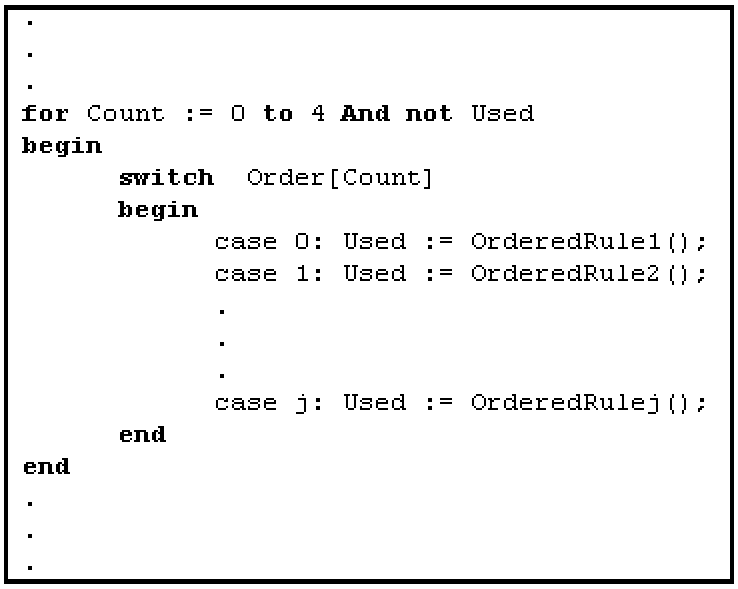
6.7.2.2. Rule Order

Since the order of the evaluated rules in the rule-based approach is not trivial (see 6.5.6.1), it is worth the time to analyze the current order and see if it can be optimized. Some of the rules however are obviously used in all puzzles and do not influence the other rules, are difficult for humans of have a performance issue/ These rules are left out of the optimization and remain on their initial position, such as FillGaps, SimpleScratch, Try and SimpleTry.

First we need to modify the main solving procedure to enable switching of the order in which the rules are evaluated. Then we need to let loose all the different orders on the same unchecked puzzle collection to see which one of them solves the most puzzles. At last we select the best orders and try to solve a test set that was already proved solvable by another (out ruled) order to test them on completeness. Then we can select (at random) one of the remaining best orders to use in further program testing.

The Rules that are selected for re-ordering are DiffersOne (0), DiffersIsPossible (1), Blocks (2), SuperFillGaps (3) and Rule N-1 (4).

The inside of the solving loop will be adapted as in Figure 56 between BuildLocation and Try.



```
.
.
.
for Count := 0 to 4 And not Used
begin
      switch  Order[Count]
      begin
            case 0: Used := OrderedRule1();
            case 1: Used := OrderedRule2();
            .
            .
            .
            case j: Used := OrderedRulej();
      end
end
.
.
.
```

Figure 56. Adaptation of the main solving loop

After going through a test set of 1000 random, unchecked puzzles with all possible 120 orders of these 5 rules, the following orders considered the most puzzles solved and therefore performed best in the first selection by a difference of approximately 5 out of 1000 rules:

```
10234, 10243, 10324, 10342, 10423, 10432, 13024, 13042,
13402, 14023, 14032, 14302, 31024, 31042, 31402, 34102,
41023, 41032, 41302, 43102.
```

It becomes clear from the above orders that certain patterns or their transitive closure occur in every one of them. The following can be concluded from the set.

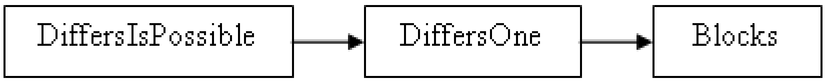


Figure 57. The transitive order that occurs in every best test set

After the first selection process we will subdue our solvable test set to the different orders. The following remaining orders performed best (and equally well):

```
10324, 10342, 10432, 13024, 13042, 13402, 14032, 14302,
31024, 31042, 31402, 34102, 41032, 41302, 43102.
```

We can now update our previously created transitive order diagram (see Figure 58).

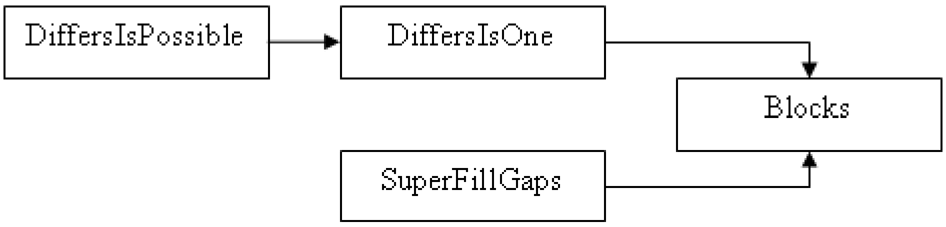


Figure 58. Updated transitive order

A small remark can be placed upon these results from the theories described in 6.5.6.16. It may well be the case that the fact that Blocks is contained by Try alone, makes it a weaker rule than its three competitors. The fact that DiffersIsPossible apparently has an edge over DiffersOne can not be validated through that theory.

After our selection process we have selected the 15 rule orders out of the original 120 that performed best. Since the first one is the most like the original order we pursued, we will use it during further testing of our rule system. The order will be therefore be adapted as follows.

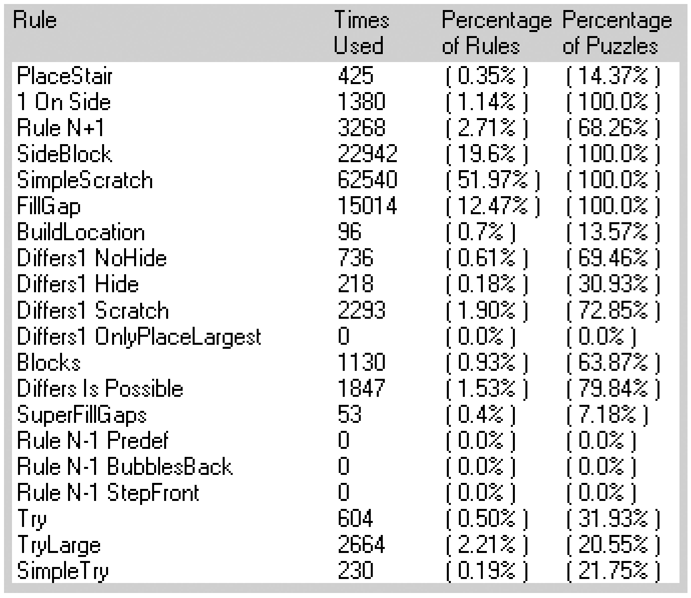
PlaceStair, PlaceN, Rule N+1, Sideblock, SimpleScratch, FillGaps, BuilLocation, DiffersIsPos, DiffersOne, SuperFillGaps, Blocks, Rule N+1, Try and as last SimpleTry.

6.7.2.3. Rule Usage

One may wonder if all the rules in the rule-based system are as important as other rules. Of course this is not the case. Some rules are extraordinarily more powerful than others and can (and will) be applied in a large percentage of puzzles. Some rules on the other hand may become completely obsolete because of a rule earlier in the solving cycle or are almost fruitless on their own because of their low applicability rate. It may be interesting to establish the merit of each rule.
In order to come to a comparison between rules we need to have a test set at our disposal that contains information about what rules are used in every solvable puzzle. As described in 6.7.1 we can use such the random, solvable test set to investigate rule usage..
We will collect three statistics on rule usage:

- General rule usage that describes how many times a rule is used
- Percentage of rule usage that describes what percentage of used rules, this rule is accountable for (this establishes the power of this rule in the current rule order) and
- Percentage of puzzle usage that describes in what percentage of puzzles this rule is used (this establishes the applicability of this rule in the current rule order)

The research gave rise to the following data:



Rule	Times Used	Percentage of Rules	Percentage of Puzzles
PlaceStair	425	(0.35%)	(14.37%)
1 On Side	1380	(1.14%)	(100.0%)
Rule N+1	3268	(2.71%)	(68.26%)
SideBlock	22942	(19.6%)	(100.0%)
SimpleScratch	62540	(51.97%)	(100.0%)
FillGap	15014	(12.47%)	(100.0%)
BuildLocation	96	(0.7%)	(13.57%)
Differs1 NoHide	736	(0.61%)	(69.46%)
Differs1 Hide	218	(0.18%)	(30.93%)
Differs1 Scratch	2293	(1.90%)	(72.85%)
Differs1 OnlyPlaceLargest	0	(0.0%)	(0.0%)
Blocks	1130	(0.93%)	(63.87%)
Differs Is Possible	1847	(1.53%)	(79.84%)
SuperFillGaps	53	(0.4%)	(7.18%)
Rule N-1 Predef	0	(0.0%)	(0.0%)
Rule N-1 BubblesBack	0	(0.0%)	(0.0%)
Rule N-1 StepFront	0	(0.0%)	(0.0%)
Try	604	(0.50%)	(31.93%)
TryLarge	2664	(2.21%)	(20.55%)
SimpleTry	230	(0.19%)	(21.75%)

Figure 59. Statistics on rule usage

A note must be placed on the above data. Internal name giving for the rules was used for debugging purposes. Therefore DiffersOne is denoted as Differs1 and PlaceN is portrayed as 1 on Side. Also, some rules were split up in sub rules for this purpose. Try and TryLarge are exclusive and therefore Try will describe all the usage of Try for the lower number of free locations and TryLarge displays the usage for the higher number of free locations.
A second note must be placed on the above data. The lowest three values were used during randomizing of the random testset, because of their probable effects on difficulty and fun. The corresponding values correlate to the respective randomize settings (1/3, 1/5, 1/4), although the SimpleTry is used slightly less often than its usage was allowed by the randomizer.
It is apparent that some of the rules are used in all puzzles and for most it is clear why they are. PlaceN (1 on side) is always used because it is on top of the list and the 1 always appears at least 2 times in each un-stripped puzzle. SideBlock is always used because of its high position in the list and the fact that with every number larger than 1 in the view characteristic (which always happens) at least 1 flat will block the side. In this place it plays a large role as contained rule in the necessity of the Try rule. SimpleScratch in itself is a necessary rule without straight rules it contains, so it is clear why this one is used in all puzzles. FillGaps probably takes away all the thunder just by being put before Buildlocation in the list, but owes it usage rating to its (shared) necessity.
Rule N-1 unfortunately is not used at all. Although it in itself is not contained by rules that are all placed above it in the solving order, it apparently has such a low applicability that the few times it could be applied it, it finds itself countered by other rules above it in the solving order. However, it can still remain a powerful rule when special (for example: stripped) puzzles arise.
DiffersOne OnlyPlaceLargest as subpart of the DiffersOne rule setturned out to be superfluous all together and was therefore scrapped from the solving list.
All other rules performed around their expectation point. It could be expected that the higher a rule is placed in the order, the more it is used. This goes for almost all remaining rules except for SuperFillGaps which is used far less than all its colleagues. It is not clear why this rule was placed higher in the solving loop than Blocks after the rule order test, but is used far less often.

6.7.3. Puzzle rating analysis approaches

6.7.3.1. Puzzler and strategy decomposition

In order to understand what ratings we could assign to a puzzle and how these ratings are constructed we need to establish how a publisher would market his product first. We will therefore construct a market cycle and define where puzzle rating (and thus, the computer) can intervene.

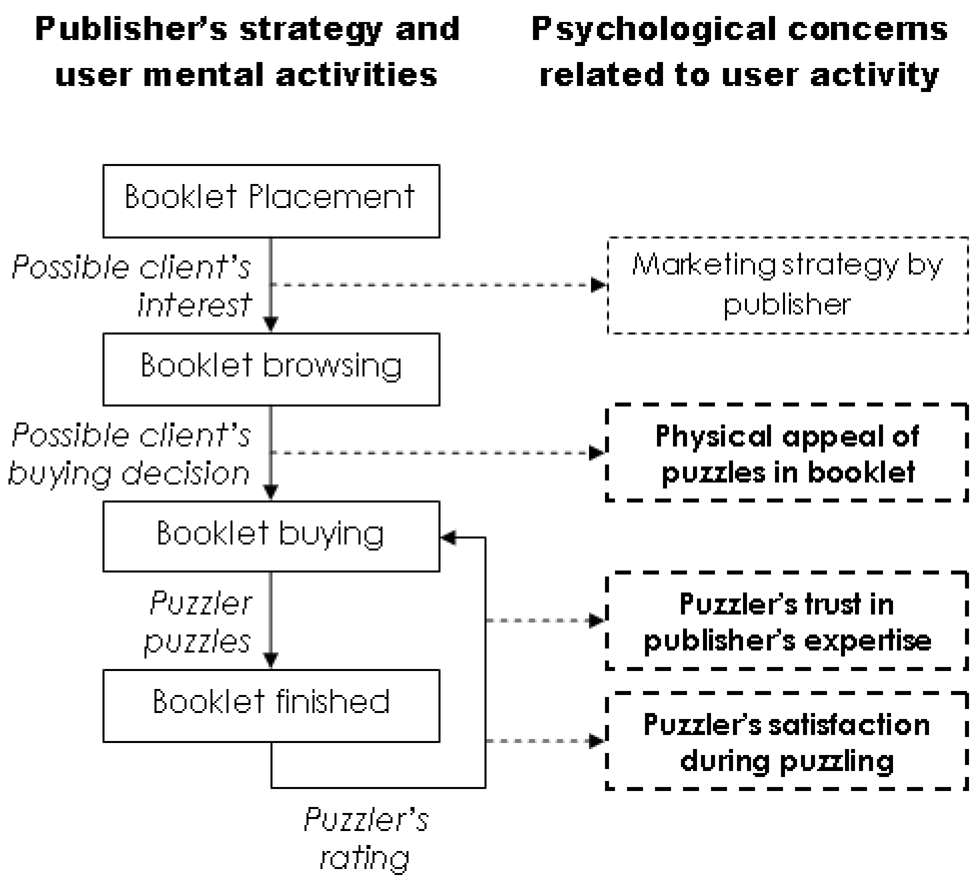

Publisher's strategy and user mental activities
Psychological concerns related to user activity
Booklet Placement
Possible client's interest
Marketing strategy by publisher
Booklet browsing
Possible client's buying decision
Physical appeal of puzzles in booklet
Booklet buying
Puzzler puzzles
Puzzler's trust in publisher's expertise
Booklet finished
Puzzler's satisfaction during puzzling
Puzzler's rating

Figure 60. Booklets market cycle and user concern constructs

In Figure 60 is depicted how the publisher would want its booklets to travel through the market. The user mental state during each phase is placed near each transition including the way how we can influence that or the user constructs associated with them. The bold printed user constructs are the areas where the puzzle rating during creation time can play a role. They are described below:

- Physical appeal: it is desirable to rate a puzzle's physical appeal before actually publishing it. As long as we consider only the inherit puzzle layout and not the way it is physically printed (i.e.:

colors, icons and so on), this is a construct we can probably rate directly from the puzzle itself.

- Puzzler's trust: it is desirable to give the puzzler the impression that the publisher is an authority in creating these puzzles. This is mainly established by providing solvable puzzles and assigning the correct difficulty to the puzzles in a certain booklet.
- Puzzler's satisfaction: it is desirable to leave the puzzler with a good feeling about the puzzling sensation and himself. A publisher can provide this feeling by providing challenging puzzles (creating perceived difficulty will create personal pride when a puzzle is solved.) fun puzzle type's (out of the scope of this test-case) and fun puzzle instances.

Summarizing the above constructs we want to automate the following puzzle ratings in our system:

- Difficulty
- Fun (perceived difficulty and common pleasure)
- Physical appeal

For this thesis we will follow the following definitions:

Def 14) Difficulty of a certain puzzle is the amount of time and the amount of mental and physical work a puzzler has to undertake(in his own view after solving the puzzle) in order to solve it.

Def 15) Perceived difficulty is the amount of difficulty a user mentally assigns a puzzle before trying to solve it.

6.7.3.2. Approach

Most approaches to instructing or learning a computer to act like a human require a test set containing those human opinions. A neural net approach would be extremely well suited to the task of rating puzzles on subjective manners while still maintaining a certain consistency. Unfortunately such and other similar approaches require the before mentioned type of test set and we don't possess one. Of course these test sets are developable, but currently the puzzle publishers have not given the acquiring of such a set high priorities.

Since we cannot use the more advanced methods of rating puzzles we will have to resort to an analytical approach. One of the best suited approaches would be to formulate an expert opinion about what factors constitute a certain rating. These dimensions can be straightforwardly taken from puzzle statistics, but can also be complexly aggregated from them. After we have formed these dimensions we can scale them to each other, assign importance

ratings to them and combine them to a value. We can then take some clustering approach to divide them into groups of evenly rated puzzles (3 groups if we want to rate a puzzle 1 to 3 on a rating). We can store the cluster values if the occasion arises in order to prevent doing the entire clustering anew every single time. In that case we can rate the current puzzle alone without re-clustering the old ones.
The clustering algorithm will first assign a cluster value to each individual that can now be viewed as clusters with one element. After all individuals have been assigned a value the algorithm joins two of the clusters with the smallest difference between cluster values and joins them into one. The cluster value of this newly created cluster will be the average of all the cluster values of the elements it contains.
Below we will describe the dimensions as we used them for our rating mechanism.

6.7.3.3. Difficulty

As we saw in 6.6.3, it is reasonably important to have a structured approach to determine how hard a puzzle will be for humans in order to secure delivery to the right target group. In contrary to what one might believe it is **not** the perceived difficulty that we will investigate here. This rating actually deals with the fact how hard the puzzle will actually be when a puzzler sets his mind to it. As was mentioned above, this is nevertheless interesting information because of the believability of the rating the publisher assigns its puzzles. It therefore partly contributes to the trust that is placed into the publisher to deliver the right puzzles to the customer and to give a good indication of the amount of time necessary to solve the puzzle. This rating is therefore important in delivering trust. Issues dealing with enthusiasm, motivation and experience are discussed in 6.7.3.4.
The following dimensions are most probable to constitute difficulty in a flats puzzle. We will also provide the importance scale (the way this dimension contributes to the rating difficulty rating (the higher, the more important)).

- Puzzle size (2): obviously the size of the puzzle plays a large role in determining how long it will take the puzzler to find a solution and therefore largely contributes to the puzzle's difficulty.
- Number of sides stripped (2): the more view characteristics are stripped, the less information a puzzler has at his disposal to solve the puzzle. This will obviously require more mental work than having more information available.

- Number of Rules used (1): the more rules are used during solving, the more time it will obviously take to solve the puzzle.
- Different Rules used (1): the more different rules are used, the more experience a user should have or gain before being able to extend his knowledge and solve the puzzle.
- Percentage of rules used (2): knowing a few rules is easy; knowing all rules is very hard. The amount of time it takes to 'discover' rules obviously rises strongly when nearing the 100% mark of being known with all rules. (It is therefore also far from sure that we have 'all possible' rules collected in our approach).
- Rule Alternation (2): it is obviously confusing for a user to keep switching in using different rules he knows. The more switching takes place, the more mental work a user has to do.
- SimpleTry (3) used: exploiting a backtracking approach would require a lot of mental work, not to mention the physical work of redrawing or erasing notations (solving an entire puzzle!). If this rule is used during generation of the puzzle under consideration, it has therefore a large impact on the difficulty. Next to the above mentioned, the fact that SimpleTry is last in the solving loop denotes the fact that at least once the other rules failed to lead to progress, constituting difficulty
- TryLarge (1) used: the usage of TryLarge may require a lot of mental work. Generating a complete set of possibilities can be tiresome mental work.

6.7.3.4. FunRate©

Clearly, it is important for a puzzle to be fun to do. Of course, the most part of the attribute fun is established by the type of puzzle, but there are probably some aspects to a single puzzle that determine the amount of fun that is related to it. In this section we try to establish a set of dimensions that constitute fun in a single flat puzzle.

- Perceived difficulty >= Actual difficulty (1): one of the most important aspects of puzzling fun is the feeling of pride that comes after solving the puzzle. The higher the difficulty is the puzzler mentally assigns to a puzzle the greater the pride will be after solving it. We will aggregate perceived difficulty from the following more concrete metrics:
    - Number of 1's (2): the less 1's are incorporated into the view characteristic, the more difficult the puzzle will be rated by a puzzler. It does not really matter if the absence really diminishes the chance to solve it.

    - Number of sides stripped (3): the more sides are stripped, the higher its difficulty will be. Since this is one of the only metrics that is visible to the puzzler at first, this will influence the perceived difficulty.
    - Size of puzzle (1): the other metric that is clearly visible to the puzzler is naturally the puzzle's size. It will therefore also have its effects on perceived difficulty.
  - User level <> difficulty (2): another metric that clearly identifies the amount of fun a player gets from this puzzle instance is if the puzzler did or did not get enough or too much challenge from it. Giving a player too much for his own level can be tedious and stressful instead of relaxing and going about it the other way around leaves an experienced player totally unsatisfied.
  - SimpleTry used (2): SimpleTry is a rule that requires a lot of extra work, mental and physical, and therefore means a lot of overhead that puzzlers generally are not that fond of. It therefore has a negative effect on how a puzzle's fun is rated.
  - Time Challenge (1): the time that a puzzler usually wants to spend on solving puzzles depends on his own level. Generally puzzlers that have a high expertise in a certain puzzle area like doing these puzzles and puzzlers that perform badly like to finish it quickly. The time that is needed for doing a puzzle is here heuristically taken as the number of used rules in the solving path.

6.7.3.5. Physical appearance

By far the hardest rating to measure without having a test set at your disposal is the physical appearance. It is hard to determine what people find physically appealing or what they dislike but the following two metrics will probably release information about the nature of a puzzle's physical appeal.

- Side balance (2): The balance of how the sides are stripped is definitely of interest in this area. A single direction containing all or most of the stripped views while the other directions are fully filled is not very appealing to the possible puzzler audience. It will take the puzzle out of balance.
- Side difference (1): The difference in the numbers at the side that are visible also plays a role. Puzzles that have extremely varied sides or that almost only have one typical number in the view characteristic definitely score better on the physical appeal scale than puzzles that do not show anything special.

6.7.3.6. Rating findings

Overall, the rating of the puzzles appears to be satisfying. Although no puzzler filled test set as described in 6.7.1 was available and we thus can not statistically verify the outcome on a large scale the first small personal circle results seem promising.
The difficulty of the puzzle seems to be pretty accurate. Most puzzles get rated medium, a third gets rated hard and only a very small portion of the current puzzle gets the easy degree. This expectedly compares to reality. Some small personal tests (5 people testing 20 puzzles) revealed that approximately 85% of the puzzles were rated the same as humans would.
The FunRate of the puzzle seems to be much more complex. 70% of the puzzles was rated correctly according to a small number of human test subjects. The fact that there was also a large difference between different humans rating the same puzzle indicates that assigning a FunRate to a puzzle is a difficult and not trivial task. Approximately 50% of the puzzles was rated Ok while a third was rated superb. Only a small percentage was rated boring. This is probably what one would expect it to be.

The physical appeal of the puzzles was rated poorly. A lot of people didn't even know how to rate a puzzle's physical appeal, but only 45 % of the people eventually gave the same answer as the rating the program provided. The physical appeal rating was therefore done with questionable results. It is remarkable that 60% of the puzzles is rejected on basis of Physical appeal, but since the trustworthiness of this rating is very low it is highly questionable whether or not this can be seen as a reasonable finding.
Following these statistics we conclude that difficulty can be incorporated in puzzle programming even when the programmer lacks a suitable test set. Funrate and the rating of Physical appeal however, should be used with extreme caution.
In Table 8 the percentages over all three ratings are collected.

Table 8. Outcome in percentages of all three ratings

Score	Difficulty	Fun	Physical Appeal
1 (easy, boring, ugly)	2 %	14 %	60 %
2	66 %	55 %	21 %
3 (hard, fun, nice)	31 %	31 %	19 %

6.8. Puzzle variants

There are some variants to the standard flats puzzle. One of them has even been widely discussed in some of the previous sections: the SideStripped version. This variant was mentioned before this section because of its already published status. We will now discuss another variant to the standard puzzle that can deliver a new approach to puzzle generation: the Filled Field variant.

The Filled Field variant is a newly created puzzle that has some flats already built. It can still be SideStripped if the creator would like it to or it can be left complete. This Filling of the field can be done to make a puzzle easier or to make a puzzle solvable, which is what we will investigate here. We will also take a look at how this new approach will influence fun, difficulty and physical appeal as well as examine its effects on performance. However, we will first see how to incorporate such an approach into the existing system.

6.8.1. Incorporating Filled Fields

Obviously it would be the best solution to take a combined approach and use both the standard generation technique and if necessary the Filled Fields variant to produce an optimal puzzle. The necessity for such an approach (which would probably mean a performance setback) will need to be established first, before trying to use that strategy. Therefore we will first incorporate the Filled Field approach on its own and see how it functions in comparison to the standard approach. In order to do this, we need to adapt step 1 to 3 (see 6.6.1) from the generation sequence slightly:

1. Build a random puzzle of size *size*
2. Try to solve the puzzle.
3. If the puzzle is solvable go to step 4.

3a. Reveal one of the correct flats on its position.
3b. If the puzzle is unsolvable go to step 3a
4. …

This will deliver a solvable puzzle with some of the field positions already filled in. Depending on how we select a cell to reveal this method will give some very different results. There are two main ways to do the cell selection for this variant:

- Random: the cell that is selected is chosen randomly from the set of unrevealed cells. The number of flats that has to be revealed in order to find a solvable situation is unpredictable. This unpredictability is to be expected since we make no effort at all to select an optimal cell to reveal. To make things worse, we do not remove an already revealed flat from sight, nor do we have any way of knowing that the deletion of already visible flats will not disrupt the certainty of the approach ever reaching a solvable situation.

- Heuristic revealing of flats: in order to reach the situation where we have a near minimum of revealed flats and still create a solvable puzzle in a reasonable amount of time we need to have a more structured approach than the random one. In order to accomplish this we need to determine how to approximate the optimum choice without too much loss in performance. The best way to do this appears to be the usage of a data structure that is already there: the scratch field. As we remember the scratch fields represent everything we know at a certain position in the puzzle solving path (see 6.5.6.1). We can use this information as heuristic for our Filled Field approach. After step 3b (or step 3) we have obtained a partially solved puzzle and its current state in the form of the mentioned scratch fields. We now randomly pick one of the cells, that we know nothing or very little about in a random way. This way we can be sure we will take the puzzle solving path at least one step ahead: we now not only reveal a flat, we reveal actual extra information. Revealing extra information will most likely lead to better or more complete conclusions. It is therefore to be expected that this will reveal only a near minimal amount of flats.

6.8.2. Filled Fields findings

Obviously, we want to know how our new way of producing puzzles performs from an analytical and performance point of view. We will test how the newly developed algorithm performs in comparison to the old, clean generation. We will compare the two on their performance with respect to the amount of tries necessary to find a solvable puzzle and the amount of time necessary for that job. Below are the empirical statistics for both types of puzzle generation.

Table 9. Comparison between experimental data for both generation methods

	Clean Fields generation		Filled Fields Generation	
Size	Avg Tries	Avg Time	Avg Tries	Avg Time
5	1.321	0.092	1.420	0.110
6	11.550	3.79	4.340	1.750
7	85.097	180.480	6.600	102.500

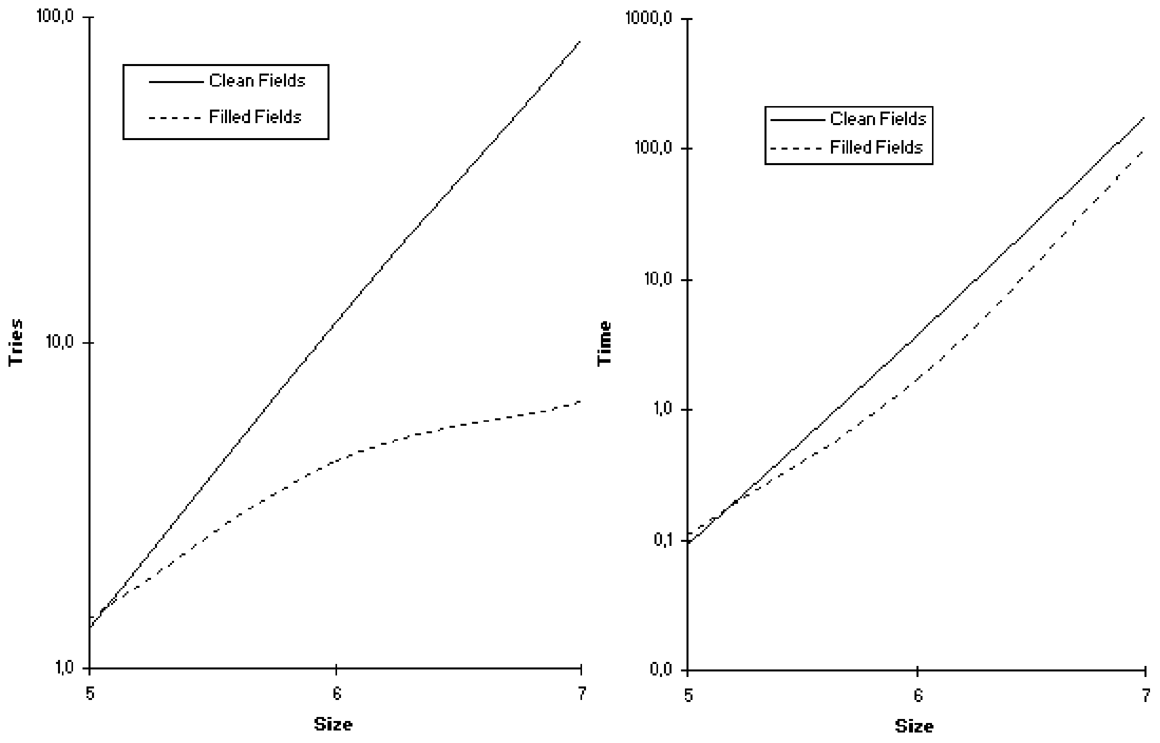


Figure 61. Graphs showing the difference in performance between Clean and Filled Fields

It is obvious that the Filled Field method performs much better than its clean counterpart in both tries (a very huge difference) and in time. These results are even better when not using SimpleTry. This can be explained by the fact that randomly generated builds are much less likely to produce a solvable puzzle than the summation of randomly revealed flats in a single random puzzle. It would therefore seem obvious for a creator to go for the Filled Fields method, certainly if performance is an issue. It is however far from certain that the results will be as good as the results from the Clean Fields method. Filling in flats might make a puzzle less appealing, but since this method has never been published, this has not been checked by the public. Results below and in 6.8.3 can guide the creator in deciding if he will start using the newest approach or if he sticks with the old one, but for the largest part the choice has to be made by his own personal feelings or even a market try-out.

We can also deduct from the experimental data how many flats will on average appear in a generated Filled Fields puzzle. Since the tries metric is the number of times the generation loop was taken, the number of flats placed in the puzzle will therefore on average be *(tries – 1)*. If we take a look at Table 9 for example, we will note that a puzzle of size *7* will have *5.6* flats already built on average when using the Filled Fields approach, which is analytically seen a very reasonable result.

6.8.3. Filled Fields effects on puzzle ratings

In order to say something about the actual ratings that these puzzles could get, one should probably adapt the rating system to account for flats already placed. This can be particularly interesting when looking for (in)balances in the physical appearance. Not having incorporated the possibilities of flats in the field in constructing this rating, Filled Fields will perform exactly the same on this rating as the standard created puzzles. It can be interesting to compare the results of both types of generated puzzles and see if the Filled Fields approach indeed scores lower as would be expected.

Table 10. Experimental findings in rating Filled Fields puzzles on difficulty

Score	Filled Fields	Clean Fields
Easy	30 %	2 %
Medium	58 %	66 %
Difficult	12 %	31 %

Table 11. Experimental findings in rating Filled Fields puzzles on Fun

Score	Filled Fields	Clean Fields
Boring	24 %	14 %
Medium	48 %	55 %
Fun	28 %	31 %

As can be seen in Table 10 and Table 11 some difference exists between the different variants in puzzle generation. Although the FunRate of Filled Fields is just marginally lower than the original variant and therefore reasonably acceptable, the difficulty drops largely after adding some flats to make a puzzle solvable.
The question remains if the above mentioned problem creates a publishing dilemma for the newly developed puzzle type. The addition of flats during creation time lowers the perceived difficulty of a puzzle. Experienced puzzlers would think this puzzle to be much easier than the regular type. Therefore it is not that unwanted that the actual test results exactly match this prejudice. Since the fun level of a puzzle drops only a little bit we can safely say that the Filled Fields variant is

excellent as a starting type for beginning flats puzzlers; especially when performance during creation time is an issue.

6.9. In depth exploration of the 5 sized puzzle

This section deals with a more complete exploration of the flats puzzle. We will research some properties and peculiarities of this puzzle type. Because of the large number of possible Latin squares we will limit the research to the flats puzzle of size 5. As we saw in Table 1, 161,280 different Latin Squares exist of this size. Creating, managing and researching this amount of puzzle builds is time consuming but still manageable. Doing this extensive research on a group of more than 800 million would be too costly and therefore is left outside the scope of this thesis. Nevertheless, research on the 5 sized flats puzzle provides some interesting results.

6.9.1. Generating the complete set

In order to learn more about the nature of the 5 sized puzzle, it is apparent that we need to generate all possible puzzle builds of size 5.
As was mentioned in 6.3.2.3 no efficient algorithm is known to generate all Latin squares of a given size. Since the number of Latin squares of size 5 is still not too large we can just use a backtracking approach to find and store them all. For this, we will use the backtracking approach as we used to find solutions to a puzzle (see 6.5.5.1) but only backtrack whenever the Latin directive is violated (i.e., not on views). All possible builds were generated in approximately 2 minutes.

6.9.2. How many different

A first question that is obviously interesting to research, now that we have the complete set at our disposal, is to know how many of all Latin squares actually lead to different flat puzzle builds. Whenever a puzzle has two or more possible solutions it is obvious that two different Latin squares correspond to that puzzle. Therefore, the other way around, two different squares can correspond to the same puzzle build (see Figure 62).

Now that we have established when two squares lead to the same build we can investigate how many of the 5 sized squares lead to different puzzles. It turns out that 102,398 of the 161,280 Latin squares lead to a different puzzle. This means that 63.5 % of the Latin squares yields a different build and therefore 36.5 % of the Latin squares yields an ambiguous puzzle.

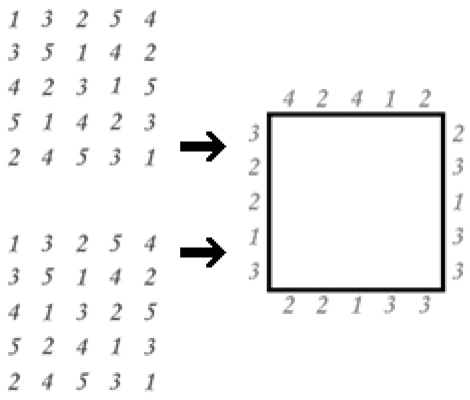


Figure 62. Two different Latin squares (row 3 and 4) delivering the same puzzle build

In relation to 6.3.2.4 and 7.1 we can state that it is therefore remarkable how many different puzzles we generated in our test case, starting from one single reduced square.

6.9.3. How many solvable

It is obviously interesting to know how many puzzle builds of the complete set of 5 sized Latin squares are deemed solvable by the rule based approach (without the SimpleTry rule). One would expect that it is only interesting to know how many of the different puzzles the program is able to solve, but this is not the case. Since all builds yielded by two or more different Latin squares are in essence ambiguous and therefore automatically unsolvable, these builds are automatically deemed unsolvable.
Of the unambiguous puzzles, an unexpectedly low number of puzzles turns out to be solvable. As we saw in 6.6.3, the rule based system was able to solve approximately 66 % of the randomly generated builds (from a single reduced square). However, the exact number for uniquely solvable 5 sized flats puzzles by the rule based system is 38,310 out of 161,280 constituting a percentage of 23.8. This is much lower than the amount of puzzles found by the generation sequencer in 6.6.3. We can, again, conclude that the reduced square we used for creating puzzle builds produces a very good result.

6.9.4. Most hard, easy, boring and fun

Naturally, it is interesting to know what the most extreme puzzles are when rated as described in 6.7.3. We will first discuss the puzzles that score extremely on the difficulty scale and second the puzzles that are most and less fun to do.

6.9.4.1. Difficulty

The hardest and easiest puzzles of size 5 were found easily. It turned out that there was 1 single puzzle that was considered the hardest and 1 single puzzle that was labeled easiest by the program. They are depicted in Figure 63 with the hardest one on the left and the easiest one on the right.

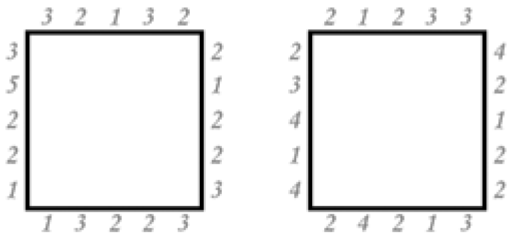


Figure 63. The hardest and easiest puzzle of size 5

Since all puzzles have the same size and are non-stripped, the main factors on which the different difficulty ratings are based are rule variation and amount of different rules used. These are obviously the factors that constitute the difference between the two above puzzles. The easy puzzle uses a very small amount of rules with very little alternation while the hard puzzle uses a lot of different rules and alternates a lot. In total the hardest puzzle was rated exactly 200% as hard as the easiest puzzle of size 5.

6.9.4.2. Fun

In contrast to the simple outcome of the difficulty experiment as described above, the puzzles of size 5 did not differ that much in fun. Exactly 5 puzzles of the 102,398 different puzzles were all rated as the most fun, and 2 puzzles were rated as being the most boring. The small difference that exists between the most fun and the most boring puzzles of size 5 and the reasonably large number of puzzles that are rated in those extremes can be explained because most of the aspects incorporated in the fun rating mechanism depend on a larger puzzle size. It is therefore imaginable that most of the 5 sized puzzles are boring in comparison to their larger variants.
Below is one of the five puzzles that was rated most fun (Figure 64) and one of the two puzzles that was rated boring (Figure 65).

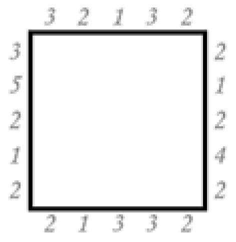


Figure 64. One of the five most fun puzzles of size 5

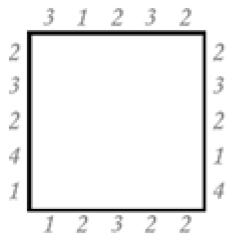


Figure 65. One of the two most boring puzzles of size 5

6.9.5. SideStrip special looks

One may wonder what the possibilities are when scratching away the puzzle characteristics. The number of views we are able to scratch are discussed in 6.6.2.1 and 6.9.6, so what we will discuss here is the possibility of scratching away certain patterns of views still leaving the puzzle solvable.
At first it may seem impossible to scratch away an entire direction (North, East, South, West) of view information while maintaining the puzzle's solvability. It is counter intuitive to humans to expect the existence of such a puzzle. Such a puzzle, however, is found very quickly. Usually one will find such a puzzle within 5 tries of generating a 5 sized, 5 stripped puzzle (see Figure 66 (left)).
A stronger demand would be to see if we can scratch an entire row of view characteristics for both the horizontal and vertical direction. In the above case we can still draw a lot of information from the non stripped direction, but in this case we seriously limit the way in which we can cross-deduct over both rows and columns. A puzzle with this property is however as easily found as the above mentioned one. An example can be seen in the middle of Figure 66.
Next to stripping two rows in different directions we can also try to strip two rows for a certain direction: stripping both the entire North and South or the entire East and West. One may doubt if stripping an entire direction will ever maintain the puzzle's solvability but like the above examples, a puzzle complying with these requirements was easily found (see Figure 66 (right) for an example).

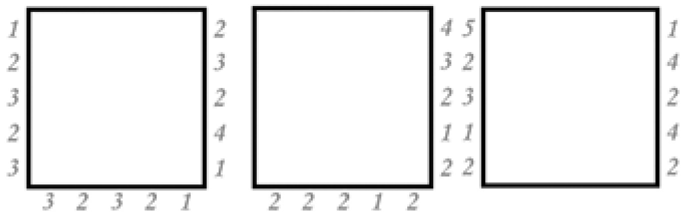


Figure 66. Specially stripped puzzles of size 5

One may wonder if it is even possible to strip away three entire sides of the characteristics. This is in fact possible and will be discussed in 6.9.6 because of that puzzle's other properties.
Of course, next to stripping away entire rows or columns of view information one can also wonder if only stripping one view can render an initially solvable puzzle unsolvable. This is in fact the case. In Figure 67 we can see a complete puzzle that is solvable and the same puzzle with one view stripped that is not uniquely solvable anymore.

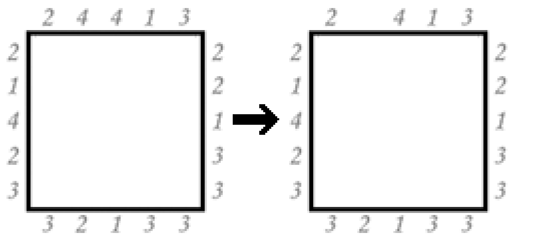


Figure 67. An unsolvable puzzle with one stripped view, derived from a solvable puzzle

6.9.6. SideStrip extremes

We now consider:

Def 16) An extreme set of stripped views is a set of views that can be stripped but allows no other views to be stripped while maintaining solvability.

In this section we try to find the smallest possible extreme set in the entire set of 5 sized puzzle builds.
All sides contain at least one view that is exactly 1, because all sides contain exactly one largest possible flat. This particular view can always be stripped when leaving the other part of that view row intact. This can be done because all other positions on the side will be ruled out by the SideBlock rule. This way, the information of a view of 1 will be superfluous, since we already know where to build the largest flat. We can therefore always strip at least 4 view characteristics and know that a minimum extreme set would at least contain 4 views. Searching for such

an extreme set over all 161,280 puzzle builds is very time consuming and therefore falls outside the scope of this thesis.
NOTE: the current smallest extreme set that was encountered by the author contained 10 items.

The maximum amount of views that can be stripped in this set is 16, only leaving 4 views and in contrast to what one would expect these views are all on the same side. The puzzle that remains solvable after stripping 16 sides is displayed in Figure 68. Naturally there are seven other puzzles that have the same property (the puzzles that have the same series of visible views but on another side). This is also the puzzle that has three entire sides stripped as was mentioned in 6.9.5.

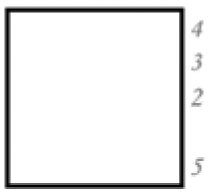


Figure 68. Maximally stripped puzzle of size 5

It turns out that this puzzle has a very special Latin Square as basis, which might very well be the reason that such a stripping amount and such a special view are possible. The Latin Square that resembles this puzzle's solution is displayed in Figure 69 and is special because this is in fact the reduced Latin square that we used in our generation program (see 6.3.2.3).

1	5	4	3	2
2	1	5	4	3
3	2	1	5	4
4	3	2	1	5
5	4	3	2	1

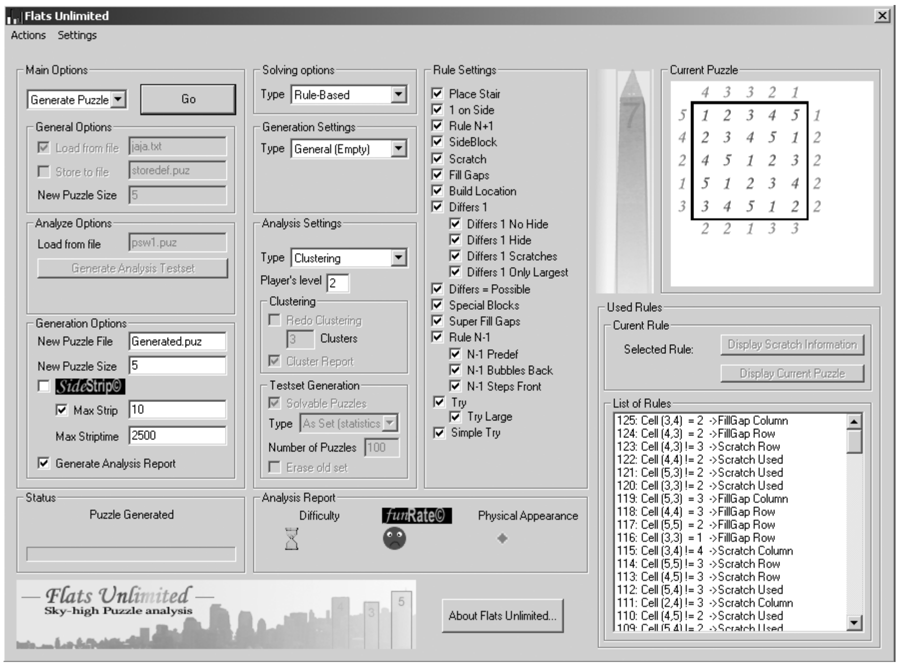
Figure 69. The solution for the maximally stripped puzzle of size 5 (and the reduced square used for puzzle generation)

6.10. The Flats unlimited application

The Flats Unlimited application is the Microsoft Windows application that was used in this thesis which contains all the used source code and combines an intuitive User Interface and an easily adaptable architecture to prov[1]ide sky-high puzzle analysis possibilities. During research time (and probably later) the reasonably up to date executable program is available on

[1] http://www.liacs.nl/~tcocx

the internet. The software can be used as an extensive scientific analysis tool as well as a business solution to creating flats puzzles (switch type by menu option). Below is a short description of how the program works. A complete manual comes with this thesis as attachment A.



Main Options: Generate Puzzle

General Options: Load from file: jaja.txt; Store to file: storedef.puz; New Puzzle Size: 5

Analyze Options: Load from file: psw1.puz

Generation Options: New Puzzle File: Generated.puz; New Puzzle Size: 5; Max Strip: 10; Max Striptime: 2500; Generate Analysis Report

Solving options: Type: Rule-Based

Generation Settings: Type: General (Empty)

Analysis Settings: Type: Clustering; Player's level: 2

Clustering: Redo Clustering; 3 Clusters; Cluster Report

Testset Generation: Solvable Puzzles; Type: As Set (statistics; Number of Puzzles: 100; Erase old set

Rule Settings: Place Stair; 1 on Side; Rule N+1; SideBlock; Scratch; Fill Gaps; Build Location; Differs 1; Differs 1 No Hide; Differs 1 Hide; Differs 1 Scratches; Differs 1 Only Largest; Differs = Possible; Special Blocks; Super Fill Gaps; Rule N-1; N-1 Predef; N-1 Bubbles Back; N-1 Steps Front; Try; Try Large; Simple Try

Status: Puzzle Generated

Analysis Report: Difficulty; funRate©; Physical Appearance

Current Puzzle

	4	3	3	2	1	
5	1	2	3	4	5	1
4	2	3	4	5	1	2
2	4	5	1	2	3	2
1	5	1	2	3	4	2
3	3	4	5	1	2	2
	2	2	1	3	3	

Used Rules: Curent Rule: Selected Rule:

List of Rules

```
125: Cell (3,4)  = 2  ->FillGap Column
124: Cell (4,3)  = 2  ->FillGap Row
123: Cell (4,3) != 3  ->Scratch Row
122: Cell (4,4) != 2  ->Scratch Used
121: Cell (5,3) != 2  ->Scratch Used
120: Cell (3,3) != 2  ->Scratch Used
119: Cell (5,3)  = 3  ->FillGap Column
118: Cell (4,4)  = 3  ->FillGap Row
117: Cell (5,5)  = 2  ->FillGap Row
116: Cell (3,3)  = 1  ->FillGap Row
115: Cell (3,4) != 4  ->Scratch Column
114: Cell (5,5) != 3  ->Scratch Row
113: Cell (4,5) != 3  ->Scratch Row
112: Cell (5,4) != 3  ->Scratch Used
111: Cell (2,4) != 3  ->Scratch Column
110: Cell (4,5) != 2  ->Scratch Used
109: Cell (5,4) != 2  ->Scratch Used
```

Figure 70. Screenshot of the Flats Unlimited application

6.10.1. Main Options

Flats Unlimited offers four options to a puzzle's creator or researcher, that can be selected via shortcut keys or the main selection box in the top left corner of the screen (see Figure 70 and Figure 71). The four options consist of:

- View Puzzle: view the chosen puzzle or a new one without the solution or field input
- Solve Puzzle: solve the chosen puzzle or a new one, show the end result and, possibly, the solve path.
- Analyze Puzzle: analyze the chosen puzzle according to the chosen option
- Generate Puzzle: generate a puzzle of the chosen size and characteristics and solve it afterwards.

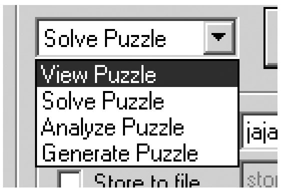


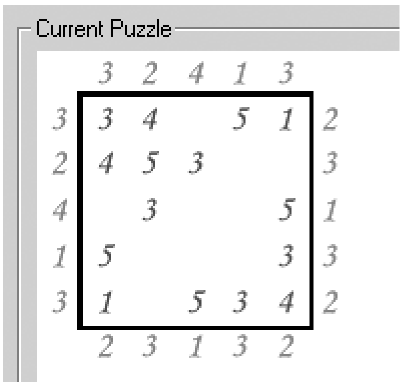
Figure 71. Selecting program options through the main input box

Some basic settings for all options are available on the left, while some more advanced settings are available in the middle of the window.

6.10.2. Main Feedback Information

Flats Unlimited provides four main ways of feedback to the user. They are:

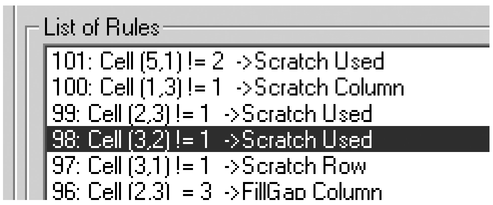
- Current puzzle viewer: in the top-right corner the user can view the puzzle that is now in use by the program. The puzzle is filled with a (partial) solution or may remain empty according to the user's choice.



	3	2	4	1	3	
3	3	4		5	1	2
2	4	5	3			3
4		3			5	1
1	5				3	3
3	1		5	3	4	2
	2	3	1	3	2	

Figure 72. Flats Unlimited's puzzle viewer

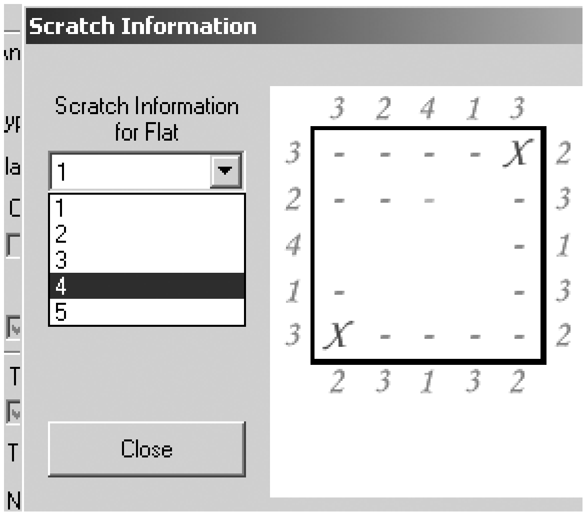
- Current solving path viewer: in the bottom-right the user can view and select the rules used in solving the current puzzle. He can then decide to view the current puzzle or scratch-field information at that point in the solving procedure.



```
101: Cell (5,1) != 2  ->Scratch Used
100: Cell (1,3) != 1  ->Scratch Column
99: Cell (2,3) != 1  ->Scratch Used
98: Cell (3,2) != 1  ->Scratch Used
97: Cell (3,1) != 1  ->Scratch Row
96: Cell (2,3)  = 3  ->FillGap Column
```

Figure 73. Flats unlimited's rule viewer

- Current scratch status viewer: in a pop-up window the user can see the scratch-field information at any point in the solving cycle. This enables debugging during research time and raises understanding of the solving procedure during user time.



	3	2	4	1	3	
3	-	-	-	-	X	2
2	-	-	-		-	3
4					-	1
1	-				-	3
3	X	-	-	-	-	2
	2	3	1	3	2	

Figure 74. Flats Unlimited's scratch viewer

- Current analysis viewer: the bottom middle of the window reveals the analysis window, which reveals the outcome of the quantative puzzle analysis. It shows how a puzzle was rated on three fronts: Difficulty, Fun and Physical appeal. All ratings on these dimensions are depicted by 1 to 3 icons each (below, the puzzle scores 1 'icon' for all ratings). The coloring changes with the number of pictograms.

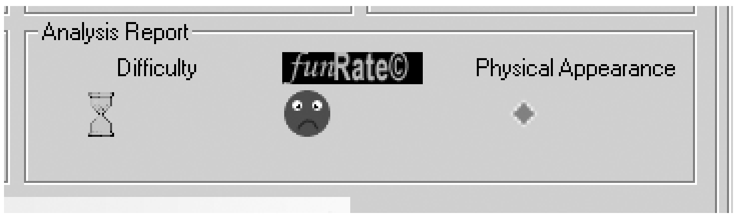


Figure 75. Flats Unlimited's analysis viewer

Chapter 7
TEST CASE CONCLUSIONS

'Hasty conclusion like hole in water, easy to make'
Charlie Chan, Charlie Chan in Paris.

Some aspects of the Flats Unlimited test case gave rise to some interesting discoveries about the computer intervention points and the flats puzzle itself. They will be discussed below.

7.1. Building process

As we can see in the Flats unlimited application the building of a single new unverified puzzle is very fast, even when done on a mediocre machine. The foundation of the flats puzzle (the Latin square) can be generated very fast and the views are calculated with the same amount of ease. Of course not all of the logical puzzles use a Latin square as basis, but since most of them use a similar type of mathematical construct it is to be expected that generating puzzle bases will in general not be the bottleneck of a computer system. Since programming the build system was not very time consuming as well, we can safely assume that an automated build will always enrich your puzzle's creation process.
A note to the above must be that we used the limited Latin square generation system as discussed in 6.3.2.4. This could in essence be a problem for the generation of this puzzle and could therefore be seen as a warning to other systems. The author and other people involved considered most puzzles to show a lot of variation in solving path, layout and physical appearance even after a large number of puzzles was generated. So, since no problems were found in variance between the different generated puzzle instances, it is to be expected that even generating a small number of possible variations of a mathematical construct can still result in a large number of different and differing puzzles.

7.2. Solvability testing

The creation of three basic types (genetic, backtracking and rule based) of solving the flats puzzle led to the conclusion that one of them already gave way in the demand that the approach should come to complete and correct solutions. The remaining two both solved most (all) of the puzzles and never solved incorrect ones. Best performing was the rule-based approach,

as is currently already used in some of the small programs used by Puzzelsport. It provided the most human way to solve the puzzle and was able to deliver a complete step-by-step way to solve a certain puzzle. It also enables the further analytical use of its used rules in order to establish puzzle rating and analysis. The backtracking approach, although solving just a few more puzzles, performed badly from a time point of view, and is therefore considered to be unfit from both a performance and a solvability point of view. It could however be used well to determine the performance of the rule-based approach.
The Rule-based algorithm turns out to perform almost as good in the number of puzzles it considers solvable as the backtracking approach which gives rise to the idea that almost all unambiguous puzzles are solvable by humans. The addition of a small backtracking routine (SimpleTry) into the rule-based approach got this number even closer to the results of the backtracking method. It is therefore to be expected that rule-based approaches don't need to be worse than more advanced or 'all seeing' approaches that are out there, while providing much more usable feedback and a better performance
Overall, the rule-based algorithm performs within reasonable parameters, but some notes are needed here. First of all, the addition of the SimpleTry rule (and that of TryLarge) had a serious impact on the amount of time needed to solve a puzzle, ranging from a factor *2* to a factor *5* in time. When these rules are not incorporated into the solving loop, the mechanism usually solves a puzzle in less than *1* second. Rule based approaches can therefore be very fast.
The addition of SimpleTry and TryLarge is not recommended for puzzle creation because of the amount of effort that humans need to put into using it. Their addition can however be invaluable to solve third party (for example Puzzelsport) puzzles that did require their effects to take place in order to solve the puzzle.
As was mentioned before, the order in which rules are evaluated can influence the number of puzzles that is considered solved. The Flats Unlimited application proved the existence of a gap in results belonging to different rule orders and thus the need to research the order in which rules are evaluated. Some rules were used very few times during the solving of a large test set of puzzles and some were not used at all. The Flats Unlimited application showed that it can be worth the while to scan for abundances in the solving loop. Deleting unused rules will probably raise its efficiency.
Next to the practical things shown by the rule-based approach, Flats Unlimited also tested the existence and truth of so called elementary rules that contain other rules. Their sole usage turned out to solve the same puzzles as the entire set. It is not recommended, however, to use only the elementary rules of a system to retain true solvability (remember the time constraints defined in 3.4.2) and performance. Their existence is interesting

from a scientific point of view and gives rise to questions about how humans solve puzzles and if they know how some rules correlate.

7.3. Puzzle generation

The demands we put on the generation of puzzles is only partly met. After the program has been written it will indeed:

- provide a very cheap puzzle generation possibility,
- gives great feedback in the form of solving information and analysis,
- generates puzzles with large speed (compared to humans) and
- assures the incorporation of a lot of control possibilities.

The main setback is that is not cheap to develop the program and even less fast. It takes a lot of time to develop the rules for the rule-based systems and to program them out in a bug-free way. It is therefore recommended to only use program controlled generation for any type of puzzle when the results will almost certainly raise the amount of money that can be earned from it.

Puzzle generation can be very fast, although the flats puzzle test case suggest that the time it takes to generate a puzzle rises exponentially with the size of the puzzle. Next to the fact that it is not even proven that a lot of puzzles of larger size than the standards now used in the booklets are even solvable, their generation is not recommended because of the high loss in performance. It is also recommended to put the algorithm through a used rule analysis and see if any abundances can be removed from the cycle to speed up generation.

The puzzles generated by the generation sequencer turn out to be of approximately the same quality as the puzzles provided by Puzzelsport, which is more than can be said of, for example, the current automated generation of language based puzzles. Using an automated generation sequencer that is rule-based therefore does not seem to produce less fun or less difficult puzzles.

It is highly questionable that some of Puzzelsport's puzzles apparently require the use of a backtracking approach to solve them. The usage of these types of rules is not considered to be fun by a large part of the puzzle community and their decision to incorporate it in some of their puzzles is therefore one that they probably should revise.

7.4. Stripping characteristics

Stripping the sides of a puzzle can be a time-consuming business, but when wisely applied it can mean a good possibility to provide puzzle variation with little then a couple of seconds extra creation time. It is always necessary to research the amount of characteristics that can be stripped without a great performance setback. For this test case this limit was approximately 10 or 11 but this may very greatly between the different types of puzzles. It remains strongly recommended that this research is done before using a SideStrip method in a puzzle generation program under development.

7.5. Puzzle rating

The lack of a good test set diminished the opportunity to try the best ways a computer can employ to rate puzzles. The results however were still reasonably promising when using a semi-subjective analytical approach. The opinion of the program creator and his associates is still seen as the true opinion about fun and difficulty, but at least it is applied on a solid, consistent, fast basis after every puzzle generation. The use of a test set however still remains highly recommended for future work.
The failure in trying to rate a puzzle on physical appearance can very well be due to the above mentioned absence of a test set, but the difficulty with which people themselves are able to assign a rating on this subject gives rise to the question if this rating should be incorporated in programs anyway. This probably is a job still best done by a human puzzle creator that has an eye for what its customers want to see.

7.6. Puzzle variant

The try-out of a variant to the standard flats puzzle was reasonably successful. Although there were some setbacks to the use of the Filled Fields method in difficulty and fun, this method proved largely better in performance. As was mentioned before, some variations to puzzles may therefore be used for beginners or special purpose puzzlers.
The addition of new variants can improve the amount of variation a puzzle can have and therefore retain the interest of puzzlers longer than without their existence. They apparently also provide a very fast and cheap way of generation and provide a fun way to introduce new people to one of the more difficult puzzles around (like the flats puzzle).
The time needed to modify the solving loop and program was so low in comparison to the creation of the program that it is highly recommended to take the time to put in some puzzle variants in every program. The easy addition of a variant is also a reason to develop programs in the first place.

Chapter 8
CONCLUSION & DISCUSSION

This thesis deals with the logical puzzle as published in leisure meant puzzle booklets. We established a puzzle lifecycle and defined the points where a computer can play a role during creation, testing and analysis of a single puzzle instance. We tried to establish how the publishing and scientific world now think (or not think) about computer intervention in the world of logical puzzles and determined where we could come to new or better strategies or provide a solid basis upon which old methods en techniques can be maintained.
In order to reach this goal a large amount of effort was put into the creation of a test case. The puzzle chosen for this was the Flats puzzle as published by Puzzelsport. The creation of the Flats Unlimited program gave rise to the conclusion that some of the old approaches used by the publisher perform good and can even be improved when taking a closer look at them. It may, however, be concluded that the time effort put into creating such rule based systems will not always be cost-effective. Scientific methods performed not great considering the demands the puzzle industry puts upon puzzle solvability and generation. Flats Unlimited showed some promising results within the arena of puzzle rating. Both difficulty and fun were rated pretty well. Physical appearance however, was not rated with the same high results.

Researching this subject and particularly the flats puzzle led to some interesting and fun results. The creation of the first automated rating system for these puzzles and the extensive control a researcher or creator can maintain over the analysis and generation of these puzzles seem to have been worth the effort put into the creation of the Flats Unlimited application. This thesis revealed some of the gaps in researching this arena (like investigating rule order and superfluities) and created a basis upon which further research can be done. Following researchers on this subject should best put their efforts in the research of rating via a test set. Obtaining such a test set and providing some AI techniques to incorporate them into a workable puzzle analysis program can largely improve the rating system. It is not recommended that future efforts should go into the application of more new techniques for the solving of puzzles as long as the actual puzzle market is concerned. Some mathematical results may be reached, but unless new ways perform just as good as the current rule based approach they appear to function miserably in the actual puzzle creation business.

Researchers on these subjects, even if they are of old subjects of the mathematical or computer science world, should always keep in mind how its outcome can influence reality, rather than find a truth of its own.

Chapter 9
ACKNOWLEDGEMENTS

The Author would like to thank Walter Kosters and Jeannette de Graaf for supervising this research project and providing some valuable comments and suggestions. Part of the research was done thanks to the invaluable starting information, graciously provided by the Puzzelsport staff. Editor Hans Eendebak is thanked in special. The famous movie quotes were verified and partly taken from IMDb [vii], which has always been an inspiration during college time. Last, but not least the author would like to thank Pascal Haazebroek and Daphne Swiebel for their shown interest in his work, their constructive feedback and helping hands, and his parents for mental and financial support during his educational period.

Chapter 10
REFERENCES

[i] J. F. A. K. van Benthem, H. P. van Ditmarsch, J. Ketting, W. P. M. Meyer-Viol. *Logica voor Informatici, Tweede editie.* Addison-Wesley, 1994.

[ii] Conceptis Ltd. *Conceptis Logic Puzzles – The art of logic*, http://www.conceptistech.com, 2004.

[iii] J. Batenburg. *An evolutionary Algorithm for Discrete Tomo-graphy.* Master thesis, Leiden University. 2003.

[iv] People of wordIQ.com. *Definition of Latin Square.* http://www.wordiq.com/ definition/Latin_square, 2004.

[v] Eric W. Weisstein. *Latin Square*. From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/LatinSquare.html, 2002.

[vi] N. J. A. Sloane. *Online Encyclopedia of integer sequences*. http://www.research.att.com/~njas/sequences/Seis.html, 2004.

[vii] Internet Movie Database Inc. *Internet Movie Database (IMDb).* http://www.imdb.com, 2004.

[viii] B.D. McKay and E. Rogoyski. *Latin Squares of Order 10. Electronic J. Combinatorics* 2, No. 1, R3, 1-4, 1995.

[ix] J.-Y. Shao. and W.-D. Wei. *A Formula for the Number of Latin Squares. Disc. Math.* 110, 293-296, 1992.

[x] Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach (Second Edition)*, Prentice Hall, 2003.

[xi] Avelino J. Gonzalez and Douglas D. Dankel, *The Engineering of Knowledge-Based Systems*, Prentice Hall, 1993