Utrecht University

Utrecht University

BSc Artificial Intelligence

Bachelor Thesis 7,5 ECTS

Static versus dynamic generation of local solutions in Distributed Constraint Satisfaction Problems

Rein Bressers

Supervised by:

Dr. T.B. Klos

Second evaluator:

Dr. F.W. Adriaans

June 30, 2019

Abstract

Solutions to difficult real-world problems, like logistics planning and timetable scheduling, are often bound by certain constraints that need to be satisfied. As such, solutions to these problems can be found by modeling and solving the problems as Constraint Satisfaction Problems (CSP). In this thesis, we will research algorithms for solving a distributed version of CSP, DisCSP, where the problem is divided amongst multiple agents. In order to solve these problems, agents must find a solution to their individually assigned subproblem, and communicate with other agents to ensure their local solutions do not interfere with each other. We will look at two algorithms, one having agents generate all possible local solutions before communicating with other agents, so called static generation, the other having agents generate local solutions only after certain agents have communicated theirs, so called dynamic generation. We will test these algorithms by using them to solve Calcudoku, a logic puzzle with a great parallel to planning problems, on which we will expand first. We compare the algorithms based on the computational effort and the amount of communication they require to solve Calcudokus of different sizes. Our results show that when solving Calcudokus of the chosen dimensions, the algorithm using static generation outperforms the algorithm using dynamic generation in terms of both computational effort and amount of communication. However, our results suggest that if the DisCSP to be solved gets more complex, dynamic generation of local solutions might be preferred.

Contents

1. Introduction

Many modern problems, like planning problems for example, prove to be very difficult to solve by hand or by brute force, if not impossible. A great deal of research is being done in the field of artificial intelligence, developing and comparing ways to model and solve these kinds of problems efficiently. A common way to do this is by describing these problems as Constraint Satisfaction Problems. However, it is sometimes more practical to divide these problems into subproblems and distribute those amongst different entities that interact with each other. Take, for example, an organization where multiple planners of different departments are responsible for their own planning problem, in addition to having to cooperate with each other in order to create a non-conflicting schedule. It might also be desirable to divide and distribute a problem when privacy is of importance, in order to limit the amount of knowledge a solver has. Because of this, Distributed Constraint Satisfaction Problems [1] have gradually become a more common research topic in AI.

(Distributed) Constraint Satisfaction Problems

Constraint Satisfaction Problems (CSPs) are defined by a set of variables, a set of domains (one for each variable), and a set of binary constraints. These constraints consist of relations between variables, such as $v_1 > v_2$ or $v_1 \neq v_2$. A solution to a CSP is an assignment of values to the variables, so that each variable has a value contained in its domain and all constraints are satisfied. Because a lot of problems can be modeled like this, methods of finding solutions to these CSPs are widely researched [2] [3]. Generally, these methods make use of a single central solver that has complete knowledge of the problem and solves it globally. In a Distributed CSP (DisCSP), the variables are distributed amongst multiple agents. Each agent is responsible for finding a solution to its local problem, and has to communicate this with other agents in order to make sure constraints between variables belonging to different agents are satisfied as well.

Both CSPs and DisCSPs are commonly solved by algorithms using backtracking. In the case of CSPs, the variables are assigned a value one by one. If a variable has no value in its domain that is consistent with its constraints and the value assignments of other variables, backtracking occurs, and the algorithm will try to find a new value for the previously assigned variable. In the case of DisCSPs [4], each agent is assigned a priority, and agents will try to find a local solution consistent with assignments of higher priority agents, informing lower priority agents when such an assignment is found. If no consistent local solution can be found, a backtrack message is sent to the agent previous in order, which in turn will try to find a new local solution.

The order in which the variables are assigned values can have a great impact on the number of backtracks required to find a global solution. If a variable is assigned a value that is not consistent with any global solution in an early stage of solving a (Dis)CSP, a lot of backtracking can occur between the other variables before eventually backtracking to the source of the problem. In CSPs, ordering the variables based on the Most Constrained Variable (MCV) heuristic has proven to be a very effective way of reducing the number of backtracks needed to find a solution [5]. The MCV heuristic orders the variables in a CSP based on increasing domain size, because a variable with a small domain is generally less likely to assign an inconsistent value than a variable with a large domain, and is therefore placed higher in order.

However, when solving DisCSPs, we do not know the number of possible local solutions each agent has beforehand. Thus, in order to make use of an MCV heuristic, or in this case a "Most Constrained Agent" heuristic, the agents will have to generate the solutions to their local problems in advance [6]. As these local problems are often complex problems themselves, generating all possible solutions can require a lot of computing time and space.

Research question

In this thesis, we will research the effectiveness of this MCA heuristic by comparing two algorithms for solving DisCSPs. One algorithm will have agents generate the solutions to their local problem beforehand, and assign them a priority based on the number of generated solutions. Then, the agents will try to solve the problem by using this list of generated local solutions. The other algorithm will assign the agents a priority based on an estimate of the number of possible local solutions, and will only generate a solution when necessary. We are interested in how these algorithms compare in terms of efficiency, specifically computing time and amount of communication. We will research this by answering the following question:

How does dynamic generation of local solutions compare to static generation of local solutions regarding computational effort and amount of communication?

Structure

We will try to answer the research question by looking at the DisCSPs of choice first, Calcudokus. We will explain what a Calcudoku is and why we have chosen this problem. After that, we will go into detail about the algorithms we will be implementing, expanding on the difference between them, the way efficiency and communication are measured, and the size and number of the Calcudokus we will be testing on. Next, we will present the results of our experiment, after which we will analyze and discuss these results, and how they compare to our hypothesis. Then, we will answer our research question and state our conclusion. Finally, we will look at possible future works.

2. Calcudokus

We have decided to test the two solving algorithms on Calcudokus modelled as DisCSPs, in order to gather the information needed to answer our research question. This chapter will explain what a Calcudoku is, how a Calcudoku can be modelled as a DisCSPs, how the algorithms solve the given DisCSPs, and in what way the algorithms differ from each other.

What is a Calcudoku?

A Calcudoku [7] is a puzzle which consists of a n by n grid, where n is the dimension of the Calcudoku, divided into cages of varying shape and size. Written within each cage is an arithmetic operator ($+, -, \times, \div$) and a value. The goal of the puzzle is to fill the cells of the grid with numbers 1 to n, such that each row and each column contains only one occurrence of each number, and all of the cages' arithmetic clues are satisfied. We will explain the arithmetic clues using the Calcudoku in figures 1 and 2 below:

- A $+$ clue means that the numbers in the cage must add up to the given value. For example, the numbers in the cage with $13+$ as clue must sum to 13.
- Similarly, a $\times$ clue means that the product of the numbers in the cage must be equal to the given value.
- A – clue means that the largest number in the cage minus the other numbers in the cage must equal the given value. In the Calcudoku below, the $0-$ cage has numbers 1, 2 and 3, which is valid because $3 - 2 - 1 = 0$.
- Similarly, a $\div$ clue means that the largest number divided by the other numbers must be equal to the given value. We see that in the top left cage, the numbers 2 and 6 satisfy the $3\div$ clue, because $6 \div 2 = 3$.

If a cage consists of only one cell, the cage has no operator and the number in that cell must be the given value. Note that a cage may contain the same number more than once, given the numbers don't share a row or column, as seen in the $108\times$ cage below.

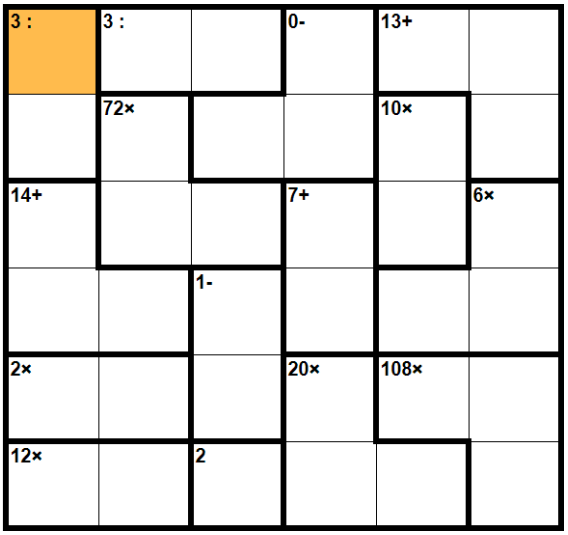

Figure 1. An unsolved 6x6 Calcudoku.

Figure 2. The solution to the Calcudoku in figure 1.

Calcudoku as Distributed Constraint Satisfaction Problems

For the algorithms to solve these Calcudokus, the Calcudokus will have to be modelled as DisCSPs. A DisCSP can be formally defined as a tuple (χ, D, C, A, ϕ) [8], where:

- $\chi = \{x_1, \dots, x_n\}$, the set of variables.
- $D = \{D(x_1), \dots, D(x_n)\}$, the set of domains, where $D(x_i)$ is the non-empty finite domain of variable x_i.
- $C = \{c_{i,\dots,j} \mid x_i, \dots, x_j \in \chi\}$, the set of constraints between variables, where $c_{i,\dots,j}$ is a relationship between variables x_i,…,x_j.
- $A = \{a_1, \dots, a_m\}$, the set of agents.
- $\phi = \chi \to A$, a function that maps each variable to an agent.

In our DisCSP model of Calcudokus, each cell in the Calcudoku's n by n grid is a variable and the domains of these variables will be the numbers 1 to n. Cells in the same row or column have a "not equal" constraint between them, meaning that if x_i and x_j are in the same row or column, they cannot be given the same number. Each cage will be represented by an agent, which is tasked with assigning numbers to the variables in that cage, so that the numbers satisfy the cage's arithmetic clue and no constrains between the variables are violated. If two agents have variables that have constraints between each other, these agents are neighbours. In order to solve the Calcudoku, agents will have to calculate solutions to their own arithmetic clue, and communicate these solutions to their neighbours, to make sure their assignments are consistent with each other. By modelling Calcudokus like this, where each cage is represented and solved by a different agent, we have a great parallel to complicated planning problems. Therefore, by researching efficient ways to solve Calcudokus, we can get a better understanding of how to efficiently solve complex distributed problems.

Solving Algorithm

One of the algorithms used to solve DisCSPs is the Asynchronous Forward Checking (AFC) algorithm [9]. In this algorithm, the agents are ordered, and there is always one "assigning agent". This agent assigns values to its variables, and in order to make sure these values are consistent, the agent sends forward checking messages to the lower ordered agents, which in turn check if they can still find consistent values. If they can, the agent next in order will become the assigning agent, until the lowest order agent has been reached. The agents receiving forward checking messages can check for consistency concurrently, hence the name Asynchronous Forward Checking.

To make sure the comparison is fair, the two algorithms we will be looking at both make use of a slight variation on AFC, which works as follows:

Each agent has an AgentView that consist of the solutions of neighboring agents, which is empty initially. At the start of the algorithm, each agent calculates a priority value, and shares this priority with the other agents. The agent with the highest priority is the first "assigning agent", and tries to find a solution to its arithmetic clue. When it does, it will send this assignment to its lower priority neighbours. These agents update their AgentView with the received assignment, and check if there still is a local solution consistent with their AgentView. If the AgentViews of all lower priority neighbours are consistent, the assigning agent will inform the agent next in order, which in turn becomes the assigning agent. When the lowest agent in order is the assigning

agent and finds a consistent assignment, all of the agents' assignments are consistent, thus a solution has been found and the algorithm terminates. In order to communicate, agents send other messages. These messages can be of the following types:

- ForwardCheck-message

  If the assigning agent has found an assignment consistent with its higher priority neighbours, it will send ForwardCheck-messages to its lower priority neighbours, containing this assignment.

- OK- and NotOK-messages:

  When an agent receives a ForwardCheck-message, it will update its AgentView and try to find a consistent assignment. If it succeeds to do so, it will send an OK message to the sender of the ForwardCheck-message. If not, it will send a NotOK-message instead. If the assigning agent receives a NotOK message, it will try to find a new consistent assignment.

- Assign-message:

  If the assigning agent has found a consistent assignment, and has received OK-messages from all its lower priority neighbours, it will send an Assign-message to the agent next in order, informing him it's his turn to try to find an assignment.

- Backtrack-message:

  If the assigning agent cannot find an assignment because it received NotOK-messages on all its local solutions, it will send a Backtrack-message to the agent previous in order, informing it that its current assignment is not consistent. The agent receiving the Backtrack-message will become the assigning agent again, and will try to find a new assignment. When the agent highest in order tries to send a Backtrack-message, the problem does not have a solution and the algorithm terminates.

Calculating local solutions and ordering the agents

The difference between the two algorithms we will be testing lies in the moment the agents generate solutions to their local problems and the way agents calculate their priorities. The first algorithm (henceforth referred to as "Algorithm 1") will have agents generate all their local solutions beforehand, using numbers 1 to n for their variables. The other algorithm (henceforth referred to as "Algorithm 2") will have agents generate solutions only when necessary, that is when receiving a ForwardCheck-message or when the agent is the assigning agent. This way, the agent only uses non-constrained numbers, resulting in a smaller search space.

In both algorithms, the agents will calculate their priorities based on the number of solutions to their local problem, giving higher priority to agents with fewer local solutions. In Algorithm 1, this is easy, because the agents generate all their local solutions beforehand. However, in Algorithm 2, the agents will not generate any solutions before they need to. Because of this, we will have the agents estimate the number of possible solutions to their problem, and calculate their priority based on that approximation.

In the case of a $+$ clue, you could compare the variables in a cage of size s with s n-sided dice. The number of possible solutions to this cage's problem is equal to the number of ways you can throw the cage's value with s n-sided dice. As dice rolls are independent random variables, we can apply the central limit theorem, which states that when independent random variables are added, their sum tends toward a normal distribution. This means that the sum of s dice rolls approaches a normal distribution $N(s \cdot \mu, s \cdot \sigma^2)$, where:

$$\mu = \frac{1}{n} \cdot \sum_{i=1}^{n} i$$

is the mean of the roll of a single n-sided die, and

$$\sigma^2 = \frac{1}{n} \cdot \sum_{i=1}^{n} (i - \mu)^2$$

is the associated variance. The probability density function of a variable with a normal distribution $N(s \cdot \mu, s \cdot \sigma^2)$ is equal to:

$$pdf_{Normal}(x) = \frac{1}{\sqrt{2\pi \cdot s \cdot \sigma^2}} \cdot e^{-\frac{(x - s \cdot \mu)^2}{2 \cdot s \cdot \sigma^2}}$$

Thus, the number of possible solutions to a $+$ cage with value v and size s in a Calcudoku with dimension n can be approximated by multiplying the approximated probability of that value with the total number of possible ways to fill that cage, in other words:

$$Number\ of\ possible\ solutions\ to\ a + cage =$$

$$Add(v, s, n) \sim pdf_{Normal}(v) \cdot n^s$$

This is applicable for $-$ cages as well. For example, take a $2-$ cage with three variables in a 5 by 5 Calcudoku. The solutions to this cage are either of the form of a 5 and two numbers that sum to 3, as $5 - 3 = 2$, or a 4 and two numbers that sum to 2, as $4 - 2 = 2$. The cage has to include a 5 or 4, because if 3 was the highest number in the cage, we could not fill in the other two variables as the result would always be less than 2, because $3 - 1 - 1 = 1$. This means that the cage always has to have a value of $v + s - 1$ or higher. Also, we have to consider the possible ways to place the 5 and 4, thus the total number of solutions is equal to:

$$Number\ of\ ways\ to\ place\ a\ 5 \cdot number\ of\ ways\ to\ sum\ to\ 3\ with\ two\ variables +$$

$$Number\ of\ ways\ to\ place\ a\ 4\ \cdot number\ of\ ways\ to\ sum\ to\ 2\ with\ two\ variables$$

The number of ways to place the 5 or the 4 is equal to the number of variables a cage has, and as seen above, we can approximate the number of ways to sum to a certain value with a given number of variables. We can generalize this by saying:

$$Number\ of\ possible\ solutions\ to\ a - cage =$$

$$Subtract(v, s, n) \sim s \cdot \sum_{i=v+s-1}^{n} Add(i - v, s - 1, n)$$

For the × and ÷ cages, we can apply this principle as well. The product of multiple independent random variables approaches a Log-normal distribution, as $\ln(a) + \ln(b) = \ln(a \cdot b)$. This means that we approximate the number of solutions to a × with a Log-normal distribution $Log\text{-}N(s \cdot \mu, s \cdot \sigma^2)$, where:

$$\mu = \frac{1}{n} \cdot \sum_{i=1}^{n} \ln(i)$$

is the mean of the roll of the natural logarithm of a single n-sided die, and

$$\sigma^2 = \frac{1}{n} \cdot \sum_{i=1}^{n} (\ln(i) - \mu)^2$$

is the associated variance. The probability density function of a variable with a Log-normal distribution $Log\text{-}N(s \cdot \mu, s \cdot \sigma^2)$ is equal to:

$$pdf_{Log\text{-}Normal}(x) = \frac{1}{x \cdot \sqrt{2\pi \cdot s \cdot \sigma^2}} \cdot e^{-\frac{(\ln(x) - s \cdot \mu)^2}{2 \cdot s \cdot \sigma^2}}$$

Thus, like in the case of + cages, the number of possible solutions to a × cage with value v and size s in a Calcudoku with dimension n can be approximated by:

$$Number\ of\ possible\ solutions\ to\ a \times cage =$$

$$Multiply(v, s, n) \sim pdf_{Log\text{-}Normal}(v) \cdot n^s$$

Again, like in the case of + and − cages, this can be extended to ÷ cages as well. For example, the number of possible ways to fill a 2÷ cage with three variables in a 6 by 6 Calcudoku is of the form of a 6 and two numbers which product is 3, a 4 and two numbers which product is 2, or a 2 and two numbers which product is 1. We can generalize this by saying:

$$Number\ of\ possible\ solutions\ to\ a \div cage =$$

$$Divide(v, s, n) \sim s \cdot \sum_{i=1}^{\left\lfloor \frac{n}{v} \right\rfloor} Multiply(i, s - 1, n)$$

These are still just approximations, and they will not be as accurate as the priorities calculated in Algorithm 1. However, as per the law of large numbers, these approximations become more precise as the size of the cages grow larger, and calculating all possible solutions will take longer, making these estimates more useful when solving large Calcudokus.

3. Experiment

In our introduction, we presented the following research question:

How does dynamic generation of local solutions compare to static generation of local solutions regarding computational effort and amount of communication?

In this chapter, we will formulate a hypothesis regarding our research question. We will explain how this hypothesis could be tested, and what the experiments will look like.

Hypothesis

We think that Algorithm 1 will use less communication overall, as the ordering heuristic that this algorithm uses is more precise. We believe this will result in fewer backtracks, and therefore less communication. When solving lower dimension Calcudokus, we think that Algorithm 1 will require slightly less computational effort than Algorithm 2, because we expect the cages to be relatively small. Thus, generating all possible local solutions should not take up as much resources, but does give the advantage of a more precise ordering heuristic. However, as the dimension increases, we expect the cages to grow larger, and thus that the number of possible ways to fill those cages will grow exponentially. We expect that, when solving higher dimension Calcudokus, using a more precise ordering heuristic does not compensate for the computational effort needed to generate all possible local solutions, and therefore that Algorithm 2 will perform better in terms of computational effort on higher dimension Calcudokus.

Design of the experiment

To test our hypothesis, we will be running both algorithms on 200 Calcudokus of various dimensions. In order to measure the amount of communication, we will be tracking the number of messages between agents, including backtracks. To measure computational effort, we will be tracking the total number of solutions generated, the total running time of the algorithms, the number of backtracks, and the number of Non-Concurrent Constraint Checks (NCCCs) [10]. NCCCs are constraint checks that do not happen simultaneously with other constraint checks, effectively making the other agents "wait" for the agent that is still checking constraints to finish. For example, after a ForwardCheck-message, all agents receiving that message will check if there is still a consistent solution for their local problem. They do this by either generating new solutions, or checking already generated solutions, which both require checking constraints. The agents act concurrently, so it's not that interesting to look at the sum of individual number of constraint checks, but rather to look at the agent with the highest number of constraint checks needed, and add that number to the count. This way, we get a better picture of how the algorithms compare.

Calcudokus to be tested

As the dimension of the Calcudokus grows, we expect the cage sizes to grow as well, which means that the number of ways to fill these cages grows exponentially. Because of this, we think it is more interesting to focus on higher dimension Calcudokus than smaller dimension. Therefore, we will be testing the algorithms on 50 Calcudokus of dimension 4, 50 Calcudokus of dimension 6, 50 Calcudokus of dimension 8 and 50 Calcudokus of dimension 9. These Calcudokus will be generated using the "Keen" puzzle in Simon Tatham's Portable Puzzle Collection [11], which is an online implementation of Calcudoku.

4. Results

In this chapter, we will present the results of our experiment and give an analysis of these results.

Results

The table below shows the results of the experiment. Each column is headed by the dimension of the tested Calcudokus along with the average number of variables in a cage, under which the average number of messages, backtracks, generated solutions, NCCCs and runtime in milliseconds per tested Calcudoku is stated, for Algorithms 1 and 2. Beneath that, the ratio of the results of Algorithm 1 and 2 is displayed. Because the runtimes for the Calcudokus of dimension 4 were too low to accurately record, they are not included in this table.

Algorithm	*Average cage size*	**4 by 4** *2,18*	**6 by 6** *2,19*	**8 by 8** *2,21*	**9 by 9** *2,23*
1	Messages	130	1404	25054	422993
	Backtracks	1	31	585	8042
	Generated solutions	35	121	292	557
	NCCCs	162	1161	18097	411060
	Runtime (ms)	-	0,04	1,48	24,34
2	Messages	153	1587	28156	447351
	Backtracks	2	36	620	8732
	Generated solutions	17	60	135	214
	NCCCs	191	1654	25344	445339
	Runtime (ms)	-	0,08	2,75	43,24
Ratio $^1/_2$	Messages	0,85	0,88	0,89	0,95
	Backtracks	0,5	0,86	0,94	0,92
	Generated solutions	2,06	2,02	2,16	2,6
	NCCCs	0,85	0,7	0,71	0,92
	Runtime (ms)	-	0,5	0,54	0,56

Analysis

We can see that, unsurprisingly, Algorithm 2 generated fewer solutions on average, for all the given dimensions. This is however the only category were Algorithm 2 outperforms Algorithm 1, as Algorithm 1 uses fewer messages, backtracks, NCCCs and computing time when solving the tested Calcudokus, regardless of dimension. This implies that for Calcudokus of the tested dimensions, the advantages of dynamic generation of local solutions do not outweigh the advantages of static generation of local solutions.

When we look at the relative differences, we can see that the ratio of the results of Algorithm 1 and 2 in terms of number of messages and backtracks gets closer to 1 as the dimension grows. This indicates that the difference in orderings in Algorithm 1 and 2 becomes smaller, and thus that the approximation of the total number of solutions a cage has, gets more accurate as the number of cages and the size of those cages increase. The ratio of the runtime of the algorithms also gets closer to 1, though not as rapidly as the number of messages and backtracks. After initially dropping going from dimension 4 to dimension 6, the ratio between the average number of NCCCs of Algorithm 1 and the average NCCCs of Algorithm 2 shows a rising trend towards 1 as well. The ratio of the number of generated solutions also increases as the dimension grows, meaning that in that regard, Algorithm 2 will outperform Algorithm 1 to an even greater degree as the Calcudokus get bigger.

These results indicate that while Algorithm 1 clearly outperforms Algorithm 2 when solving Calcudokus of the tested dimensions, the performance of Algorithm 2 gets closer to the performance of Algorithm 1 as the Calcudokus grow in size, regarding all test-parameters except for the number of generated local solutions. It is likely that given a high enough dimension, Algorithm 2 performs on par with, or even better than Algorithm 1.

5. Discussion and Conclusion

We will begin this chapter by discussing how the results of our experiment compare to our hypothesis, after which we will reflect on possible limitation of our experiment. Next, we present a conclusion to this thesis, by answering our research question and discussing how these results might apply to problems similar to Calcudoku. Lastly, we will consider in what ways our experiment could incite future works.

Discussion

As stated in our hypothesis, we expected Algorithm 1 to use less communication overall, which was the case. We also expected Algorithm 2 to outperform Algorithm 1 in terms of computational effort on Calcudokus of the higher dimensions, which did not prove true. Though there was a rising trend in the ratio of the results of Algorithm 1 and 2, this ratio never became greater than or equal to 1. There could be multiple reasons for our hypothesis differs from the results that we found.

One reason could be that we simply did not test the algorithms on Calcudokus of high enough dimension. While the average size of the cages did increase as the Calcudokus got bigger, it did so at a much lower rate than we expected. This meant that the local problems in Calcudokus of dimension 8 and 9 were less complex than we initially presumed, thus generating solutions to these problems required less computational effort than we imagined. Because the generated Calcudokus are intended to be solved by humans, it is possible that the growth rate of the cage sizes is purposely kept low, to ensure that higher dimension Calcudokus are not too difficult for humans to solve. Had we tested the algorithms on Calcudokus of dimension higher than 9, it is possible that due to the increase in complexity of the local problems, Algorithm 2 would perform better than Algorithm 1 in terms of computational effort.

Another reason could be the nature of the problem. The arithmetic problems the agents must solve are relatively simple and solving them does not require much computational effort, so it is possible that generating solutions to these problems beforehand does not have a large enough impact on computational effort to favor Algorithm 2. Also, checking if there is a previously generated solution that is consistent with an agent's AgentView requires almost the same number of constraint checks as generating a new solution, so despite Algorithm 1 generating at least twice as many local solutions as Algorithm 2, solving Calcudokus using Algorithm 2 still required more NCCCs and runtime on average. This might not be the case for DisCSPs where generating local solutions is much harder than checking solutions, where the number of NCCCs and runtime needed to find a global solution could depend more heavily on the number of generated local solutions.

Also, the fact that our algorithms did not run as a truly concurrent system of agents could possibly have had an effect on the results, and could therefore be a reason why our results did not entirely match our expectations. In Algorithm 1, instead of having the agents generate their local solutions at the same time, we had the agents do it one by one and counted the highest number of constraint checks and the longest runtime. We used this technique when an agent sent out ForwardCheck-messages as well, in both algorithms, counting the highest number of

constraint checks and longest runtime if all ForwardChecks were successful. If a ForwardCheck was unsuccessful, we counted the lowest number of constraint checks and shortest runtime, considering only the agents that sent a NotOK-message. This means that, along with the fact that the Stopwatch class in C# does not measure runtime very accurately, our recorded number of NCCCs and in particular our recorded runtime may not represent the number of NCCCs and runtime used by the tested algorithms entirely accurate.

Conclusion

When solving the tested Calcudokus, Algorithm 1 outperformed Algorithm 2 on all the test-parameters, except for the number of generated solutions. This indicates that algorithms that use static generation of local solutions perform better than algorithms that use dynamic generation of local solutions in terms of computational effort and amount of communication.

However, the difference in results between the two algorithms gets smaller as the size of the Calcudokus grew larger, with Algorithm 2 performing almost as well as Algorithm 1 on Calcudokus of dimension 9 regarding all test-parameters, except runtime. This might indicate that in order to solve Calcudokus of a large enough size, algorithms that use dynamic generation of local solutions could require as little, if not less, computational effort and communication as algorithms that use static generation of local solutions.

Our research suggests that in terms of computational effort and amount of communication, generating solutions only when necessary does not outweigh the advantage of having a more precise ordering heuristic when solving simple DisCSPs. Though, as the complexity of the DisCSPs and its local problems grow, the difference between static and dynamic generation of local solutions regarding computational effort and amount of communication, gets smaller. It seems that when a DisCSP is complex enough, the impact of a more precise ordering heuristic on the computational effort needed to find a global solution, does not make up for the resources required to generate all possible local solutions beforehand. To expand on this, we will conclude this thesis with some suggestions for possible future works.

Future works

For future research, it would be interesting to test algorithms on Calcudokus having higher average cage sizes or with dimensions greater than 9, to find out if the performance of Algorithm 2 will eventually match or even surpass that of Algorithm 1, with regard to computational effort and amount of communication. Likewise, testing the algorithms on DisCSPs where local problems are more complex than the arithmetic problems in Calcudoku, or where the complexity of the local problems scales faster, could prove interesting future research as well. It is possible that when solving these problems, the difference between the number of local solutions generated by Algorithm 1 and the number of local solutions generated by Algorithm 2 has a greater impact on the required computational effort than when solving Calcudokus. Researching the difference between the two orderings as the Calcudokus grow could give a better insight on the effectiveness of the MRA heuristic, and an idea of how precise the approximated ordering heuristic must be to favor Algorithm 2. Anyhow, it would be advised to use a truly concurrent system of agents in future works, to ensure the measure of NCCCs and runtime is as accurate as possible.

References

[1] M. Yookoo, E. Durfee, T. Ishida and K. Kuwabara, "The Distributed Constraint Satisfaction Problem: Formalization and Algorithms," *IEEE Transactions on Knowledge and Data Engineering,* vol. 10, no. 5, pp. 673-685, 1998.

[2] S. Brailsfor, C. Potts and B. Smith, "Constraint Satisfaction Problems: Algorithms and Applications," *European Journal of Operational Research,* vol. 119, no. 3, pp. 557-581, 1999.

[3] V. Kumar, "Algorithms for Constraint Satisfaction Problems: A Survey," *AI Magazine,* vol. 13, no. 1, pp. 32-44, 1992.

[4] M. Yokoo and K. Hirayama, "Algorithms for Distributed Constraint Satisfaction: A Review," *Autonomous Agents and Multi-Agent Systems,* pp. 185-207, 2000.

[5] J. C. Beck, P. Prossers and R. J. Wallace, "Toward Understanding Variable Ordering Heuristics for Constraint Satisfaction Problems," *Proceedings of the 18th Irish Conference on Artificial Intelligence and Cognitive Science,* pp. 11-16, 2003.

[6] A. Armstrong and E. Durfee, "Dynamic Prioritization of Complex Agents in Distributed Constraint Satisfaction Problems," *The Fifteenth International Joint Conference on Artificial Intelligence,* pp. 620-625, 1997.

[7] W. Shortz, "A New Puzzle Challenges Math Skills," *New York Times,* February 8, 2009.

[8] M. Wabbi, "Algorithms and Ordering Heuristics for Distributed Constraint Satisfaction Problems," 2012.

[9] A. Meisels and R. Zivan, "Asynchronous Forward-Checking for DisCSPs," *Constraints,* vol. 12, pp. 131-150, 2007.

[10] A. Meisels, E. Kaplansky, I. Razgon and R. Zivan, "Comparing Performance of Distributed Constraints Processing Algorithms," 2002.

[11] S. Tatham, "Simon Tatham's Portable Puzzle Collection," [Online]. Available: https://www.chiark.greenend.org.uk/~sgtatham/puzzles/.