Utrecht University

Kunstmatige Intelligentie
Bachelor Thesis
7.5 ECTS

Dynamic ordering between Asynchronous Backtracking algorithms

Christopher Huijting

supervised by
Dr. Tomas Klos
second evaluator:
Dr. Gerard Tel

June 29, 2018

Abstract

A great number of problems, like scheduling shifts in a hospital, or dividing up resources for manufacturing, can be reduced to a problem called Constraint Satisfaction Problem (CSP). These CSPs make use of constraints placed on variables to determine if a problem can be satisfied. In this thesis, the effect of ordering variables on a backtracking algorithm used to solve a CSP will be explored. A number of graphs used to model a CSP problem will be solved, dynamically changing the order of the agents in between every run. Results found indicated a decline in median number of edges in the order of agents as they ranked from a high priority to a lower priority, as well as a reduced number of backtracks required on a lower decay rate. Conclusively, the order of agents seems to matter with respect to the median number of edges per agent, but where the decay rate barely has any influence on the median number of edges, they do greatly influence the number of backtracks required.

Contents

1 Introduction

1.1 Background

Arguably the greatest boon of the computer is its automation of certain time consuming processes. Where before we had to hire full-time planners to schedule shifts, or calculate by hand what the cheapest way is to manufacture products with available resources, or calculate the fastest route by trial and error, we now have computers to do this for us. This however brings a whole new class of problems with it. How does one model these problems so a computer can understand and efficiently solve these problems. One of the ways to model for example planning or resource distribution problems, is by defining them as a Constraint Satisfaction Problem (CSP).

The Constraint satisfaction problem (CSP) is a common problem in the AI research field. It is a widely researched problem, since many problems can be generalized to a CSP. Planning, natural language processing or more simple theoretical problems like solving a Sudoku are some of these problems.

1.2 Rest of thesis

In the next chapter we will be taking a closer look at CSP and the variant used in this thesis, the Distributed CSP. After that, there will be an explanation of the algorithm used and the design of the experiments.
In chapter 4 we will present our research question and hypothesis, as well as the variables to be tested. In the next chapter we will analyze the results. Finally, we will discuss the conclusion to the thesis and any future works based on this research.

2 Constraint Satisfaction Problem

2.1 CSP

CSP is a problem that consist of three components, a set of variables, a set of domains (one for each variable) and a set of constraints. The domains define for each variable what values are allowed, and the constraints define the allowed or disallowed relation between two or more variables. To solve a CSP, a solution must be found which assigns every variable with a value from its domain, which do not violate any constraints.
To find a solution to a CSP, several options are available. One of these is local search, where an algorithm iteratively tries to improve on a complete assignment of variables. A small number of variables are adjusted and the new assignment is checked to see if it still holds to the constraints, and if it has improved on the previous assignment. Local search however, is incomplete, meaning that a local search algorithm might not find a solution.
In this thesis, we will be using the backtrack option to solve a CSP. Backtracking works by step by step choosing a variable, trying out every value in the domain of this variable, and moving on to the next variable if a value has been found. If no value for this variable has been found, as every value violates one or more constraints, the algorithm will backtrack, moving back to the previous variable and checking if a new value can be found. In contrast to local search, backtracking is complete meaning that if there is a solution to be found, backtracking will find it.

2.2 Distributed Constraint Satisfaction Problem

2.2.1 Formal Definition

Of particular interest in this thesis is the distributed variant of CSPs, abbreviated to DisCSPs [5]. A DisCSP consists of a tuple of (χ, D, C, A, ϕ), formally defined as follows:

- $\chi = \{x_1, ..., x_n\}$, which stands for the set of variables.
- $D = \{D(x_1), ..., D(_2)\}$, which stands for the respective finite domains.
- $C = \{c_{12}, ..., c_{ij}\}$ where x_i and x_j have constraint c_{ij}.
- $A = \{1, ..., p\}$, which stands for a set of agents p.
- $\phi = \chi \rightarrow$ A which is a function that maps each variable to its agent.

2.2.2 Example DisCSP

So what does this formal description look like in an actual real life setting?
Lets take for an example a Sudoku problem. In a common Sudoku, there is a grid of 9 by 9 cells, where each cell can contain 9 numbers. To solve a Sudoku, a solution must be found where a number does not occur twice in the same column, in the same row and in the same "block", which is a 3 by 3 area in the grid. There are 9 blocks in a 9 by 9 grid, the first one being row and column 1, 2 and 3.
In this case, we will assume every agent has only a single variable its responsible for. This means that we have 9 x 9, or 81 agents in a normal Sudoku. For this example, we will define an agent by naming it a_{ij}, where i stands for the row and j stands for the column. If we take a closer look at the first block, it will consist of the following agents: a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32} and a_{33}.
Take for example agent a_{11}. This agent will have a domain consisting of the numbers 1 - 9 from which it can choose a new value. It will have constraints with agents a_{12} and a_{13} for being in the same row, agents a_{21} and a_{31} for being in the same column, and constraints with the remaining agents in this block for being in the same block. These constraints ensure that no agent can have the same value.
The difference with a CSP and DisCSP is that in a CSP, there would effectively only be one agent, which controls every variable. In a DisCSP, these variables are divided over a number of agents, so that they can work together to find a solution.

2.2.3 Description

The idea is to divide the constraints in a CSP into different parts, to be solved individually by agents, who remain in contact to determine if the constraints still hold. If agents can process their constraints at the same time as all the other agents, in other words working synchronously, this could vastly improve computation time for algorithms such as backtracking and local search. Instead of having to backtrack through the entire state space, only the relevant agents have to backtrack.
One of the earliest algorithms for DisCSPs, was introduced by Makoto Yokoo [4, 9, 10], namely Asynchronous Backtracking (ABT). In this algorithm, agents act concurrently and asynchronously to find a solution to a decision problem by backtracking. Agents are ordered from top to bottom, 1 being the top and the last agent number being the bottom, and from top to bottom messages are sent and received to find a solution.
Asynchronous algorithms, while potentially not as fast as synchronous algorithms [11, 12], also reduce the need for having to revisit the entire state space [6]. In addition, in a large system where small changes might occur, such as a work schedule for a hospital with multiple wings, these changes would mean a synchronous algorithm would have to redo the entire solution, whilst an asynchronous method only needs to adjust the changed agents.

2.2.4 Assumptions

In this thesis we assume that every agent in A has a single variable and all agents have access to the same values for the variable. For the graph-colouring problem it means that the agents have access to a certain number of colours, determined beforehand to be the least number of colours necessary to satisfy the specific problem the agents have to handle.
We also assume that there is no lag between sending and receiving messages.

2.3 DisCSP to be solved

In this thesis, we will be looking at solutions to the map colouring problem. In a map colouring problem, there exist variables, usually described as countries or regions, which are placed on a map. Every country has to be assigned a colour, and every country has access to the same range of colours. The only constraint is that if two countries border each other, they cannot have the same colour assigned. This problem can easily be converted to a graph, where the countries or regions (hereafter named agents) exist as nodes, and the border relation is defined with edges between the pairs of nodes. [7]
The question we are researching is how the order of the agents influences the number of backtracks required to solve the algorithm. In previous research [8, 10] this has been explored during the run time of the algorithm, but not between runs of the algorithm. This can be useful information if an algorithm has to run many times and where small changes are made, like in the planning or resource distribution examples.

3 Method

3.1 Algorithm explained

The Asynchronous Backtracking Algorithm (ABT) used in this thesis, is slightly more optimized than the first algorithm as devised by Makoto Yokoo [10]. We will be using an algorithm that optimizes the nogood list, to reduce the space complexity from exponential to polynomial [1].
In this algorithm, every variable is defined as an independent agent. The algorithm will run in cycles, where each cycle involves every agent checking its list of received messages. If it has any messages, the agent processes these messages and may send messages out of its own. These cycles continue until there is a cycle where no agent has any messages, indicating that a solution has been found as no agent has a constraint that is violated and/or feels the need to backtrack.
Every agent has the following:
• A list of received messages, that will be processed in the current cycle.
• A "nogood" list, where it stores nogoods. A nogood is a combination of agents and their values, and its own value. This indicates that if the agents still have the values as stored in the nogood, the agent cannot have that specific value. This prevents the agents from getting stuck in a loop, as nogoods that are still accurate will remain in the nogood list, preventing a single agent from trying out the same few values in a loop.
• A list of agents higher than itself in the priority list, and a list of agents lower than itself in the priority list.
• A list of agents which are connected to this agent in the graph, indicating that they cannot share the same values.

3.2 Cycle of the algorithm

At the start of the algorithm, every agent sets an initial value and informs every agent in its lower agents list of the new value. These messages do not arrive until every agent has "had its turn", or until this cycle has ended. In the next cycle, the agents will receive the messages sent in the previous cycle and process them. This continues until not a single agent has any messages left.
Agents communicate using messages that contain information about the type of message that is being sent. The following types of messages can be sent:

- Ok-message
  - 1.1. Every time an agent changes its value, it has to send a ok-message to every lower priority agent that is connected to this agent. This message contains the new value that the agent has changed to, and the type "ok".
  - 1.2. Once an agent receives an ok-message, it adds the new value to its agent view and checks if the constraints are inconsistent with its own value.
- Nogood-message
  - 2.1. If an agent has no possible value to choose from its domain, because of its constraints, the agent has to backtrack. From its list of higher priority

agents, it determines the lowest priority agent and removes this agent and its value from the agent view. It sends a nogood message, containing a nogood composed of all the nogoods in its nogood list that caused the agent to backtrack. The agent then backtracks, and tries to find a value that is consistent with its agent view, now that it has a constraint less to consider. If it can't find a new value, it backtracks again.

2.2. Once an agent receives this nogood-message, it checks if the nogood in the message is still coherent with its own agent view. If it is, the agent adds this nogood to its own nogood list and tries to find another value to change to. If it isn't, yet the value in the nogood concerning the receiving agent is the same as its actual value, it sends another ok-message. If the value is not the same, it must mean that there is already an ok-message underway and the agent does nothing.

- AddLink-message

3.1. If an agent receives a nogood from a lower priority agent, it can contain values of agents that are not directly linked to the agent that received the nogood. If the priority of these agents is higher than the receiving agent and the nogood is coherent with its own agent view, it sends a AddLink message to the higher priority agent and adds it to the higher agents list.

3.2. Once an agent receives an AddLink-message, it adds the agent to its list of lower agents and sends it a ok-message.

In the referenced paper, another message can be sent in the form of Stop, which halts the algorithm as there is no solution to be found. Because of the fact that we used predetermined colour limits, we found there was no need for this type of message.

The major difference with this version [1] and the original version [10], is that the nogoods list gets reviewed every time a value in the agent view gets changed. Only nogoods that are coherent with the agent view remain, while others are removed.

3.3 Design of experiments

Every experiment will be focused on solving a random graph colouring problem. For this thesis, 25 agents and 100 edges will be considered per experiment. Every experiment will be randomly generated by a package named igraph [3] in R code, using the $G(n, M)$ Erdős–Rényi model (see 3.4 for a description). The main focus of this experiment is the order of the agents and whether this impacts solving times. Therefore, in this thesis at the start of each experiment, the order of the agents will be randomized.

There will be 100 experiments, which means 100 randomly generated graphs. For every individual graph, the code we've written will figure out the lowest number of colours required to solve it, and use this lowest number to solve the graph. We have introduced a certain decay rate, clarified further in chapter 4. There are four decay rates, namely 0, 0.3, 0.7 and 1. Every one of these increments will have 100 "iterations". During an iteration, which is the successful

solving of the problem, the number of backtracks per agent is tracked. After the iteration is complete, the order of the agents is rearranged based on the total number of backtracks this agent has, based on previous iterations as well as on the just finished iteration. This is also where the decay rate comes into play. A decay rate of 1 means that none of the previous iterations are remembered, as they have all "decayed". A higher decay rate means that as time goes on, previous iterations start to matter less. If a certain agent has 10 backtracks in iteration 1, a decay rate of 0.7 means that in iteration 2 these iteration 1 backtracks now only count for seven.
Every experiment has 100 iterations per decay rate. After these are all finished, the final order of the agents is registered so we can determine where agents with a higher number of edges end up in the final order of every decay rate.

3.4 Erdős–Rényi model

The graphs used for this experiment were generated by using the Erdős–Rényi model. First introduced in 1959 [2], this model has two closely related variants, namely the $G(n, M)$ model and the $G(n, p)$ model. In this thesis the $G(n, M)$ model was chosen, where n stands for the number of agents and M stands for the number of edges. This model was chosen to make sure that all the graphs have an equal number of edges, as the $G(n, p)$ has a probability of p to create a new edge which results in graphs with differing number of edges.
The $G(n, M)$ model creates a graph by choosing a graph uniformly at random from all the graphs which have n nodes and M edges.

4 Hypothesis

The research question is how the priority order of the agents influences the number of backtracks. We believe that agents who are connected to many other agents, will be required to backtrack more, as their number of constraints are higher. If these were to be in a higher priority, the algorithm would need a reduced number of backtracks to be completed, as these agents have more conflicting values with other agents. The hypothesis of this thesis is as follows: Agents with a high number of edges, will have more backtracks and thus end higher up the priority list than agents with a low number of edges.

4.1 Variables

As explained in the theoretical background, we will be looking at the order of the agents. According to our hypothesis, an agent with more edges will have a higher order on this final list.
To find this final list, we keep count of the backtracks each agent has done. Unfortunately this presents a problem, as certain orderings of the agent list can mean a very high number of backtracks for a single iteration. If a single iteration has an extreme number of backtracks, it would be hard for future iterations to break this order, seeing as they need to have enough backtracks to break the balance, which might not be the case. This can be seen as a local maximum. To prevent this local maximum, we will be introducing a decay rate to lessen the impact of outliers.

5 Analysis of results

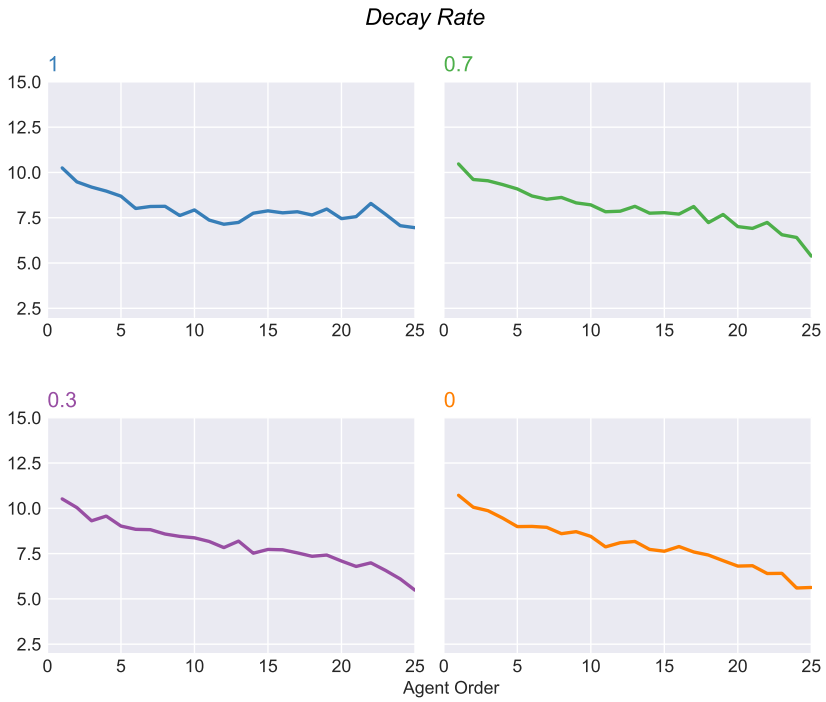

Figure 1: Decay rate for 100 random experiments with 25 agents. X-axis is for the order of the agents, as in agent 1 is the first agent in the final order of the experiment with the above mentioned decay rate. Y-axis is the median of the number of edges, for the agents of all the 100 experiments.

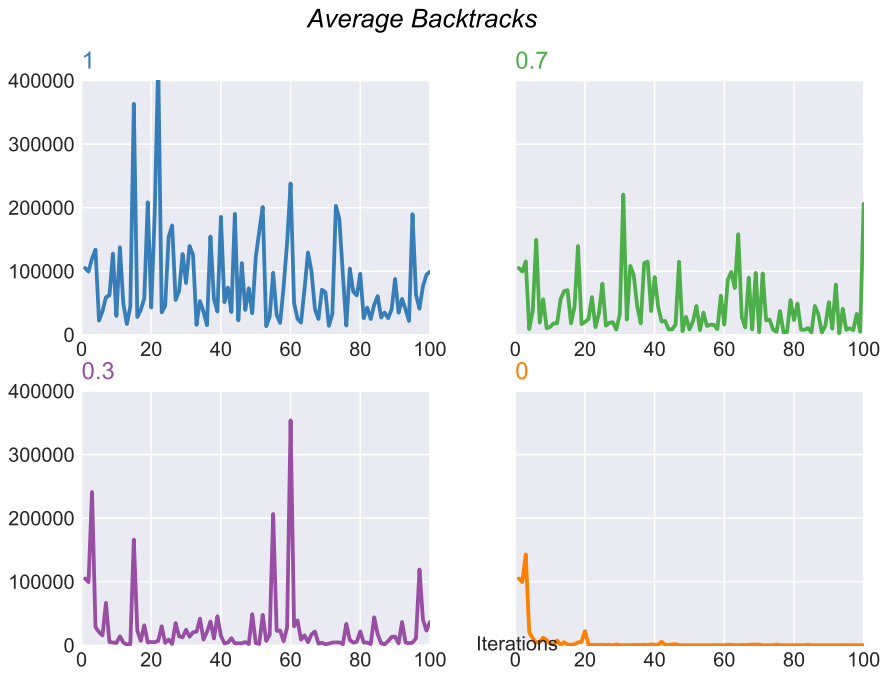

Figure 2: The average or mean number of backtracks. X-axis is for the iterations. Y-axis is the mean number of backtracks required for that iteration.

Our hypothesis was that agents with a higher number of edges will have more backtracks, meaning that they end up higher on the priority list. This means that there should be a visible decline in number of edges as we go down this priority list. We also introduced a decay rate, which was meant to combat local maxima.

In figure 1, we can see that with most of the decay rates, there is a clear decline in the median number of edges as the priority gets lower. As we have already seen [8, 10] that a dynamic order during the algorithm improves the run time compared to the normal ABT, we can now also confirm that changing this order between the runs also improve run time.
Interestingly enough the decay rate, with the exception of 1, doesn't seem to have an impact on the total priority order. Whilst total backtracks and thus speed are improved by using a lower decay rate, the median number of edges remain fairly consistent across the different decay rates. These results indicate that while the local maximum might not be optimal, it is still based on an ordering of high edges first, followed by lower number of edges.
As seen in figure 2, there is a huge decrease in backtracks required as we have a lower decay rate. Even though in figure 1 there is not much of a difference in the four different decay rates displayed, all decay rates with the exception of 1 still display a downwards trend in the number of median edges per agent.
This means that even though the median number of edges per agent in the order is somewhat the same in figure 1, the specific order in which they are placed means a lot, as seen with the decrease of backtracks required in figure 2.

6 Conclusion and future works

As established in the discussion, we found that our results seem to confirm our hypothesis in the scope of the problem of this thesis, namely the map colouring problem. Figure 1 indicates that the order only slightly differs with different decay rates, yet figure 2 indicated that a lower decay rate would result in a decrease of backtracks. This means that the priority order by itself does not have a large influence on the number of backtracks, as figure 1 has similar results for a decay rate of 0.7 and 0.0, yet as seen in figure 2 greatly differs in the number of backtracks.
There is at least a definite trend to see in the number of edges depending on the order of the agents, as well as a sharp decline in number of backtracks required by prioritizing agents with a high backtracking record. This all does mean we can conclude that the order influences the number of backtracks required by ensuring that the problematic agents with many backtracks are placed ahead of the agents with fewer backtracks.

As with any thesis, a lot of issues came up in hindsight. The used algorithm couldn't handle large number of agents or minimum number of colours, eventually limiting it to a maximum of 25 agents with 100 edges. A possible solution would have been to read all the messages first before processing them, instead of processing the messages individually.
Another would be the lack of concurrency knowledge on the author's part, as enabling the program to make use of the multicore processor would've sped up the process significantly, seeing as every agent acts independent of the other agents, as long as the cycle lasts. They only need to synchronize when all of them are done with processing their messages, as this means that the messages "in wait" can be processed. This would've allowed a higher number of agents and edges.
In figure 2, the number of backtracks is an average of the total number of backtracks. Usage of the average however, means that any extremely high number of backtracks will influence the graph, as seen for example in figure 2, decay rate 0.3. There just happens to be an experiment with a very high number of backtracks on position 60. A possible solution would be to use more experiments than the 100 we have used now.

For future research, changing the order based on the number of edges beforehand, not relying on any backtrack numbers, might prove an interesting case study to see if this would improve run-time and reduce the number of backtracks needed.
Usage of the Erdős–Rényi model results in the graph not having a heavy tail, something that usually does occur in real life problems. In a random graph one would expect a high number of agents with a low number of edges, tapering off to a low/nonexistent number of agents with a high number of edges. In real life problems, this is usually not the case, as there always is an number of agents who have a high number of edges, therefore these real life problems

have a "heavy tail". Usage of the Erdős–Rényi model means that these do not model real life problems as well as some other models would have. While that was not a focus of this thesis, as we looked at map colouring problems, this would probably mean that had we used another model, these results might have changed significantly. This could be explored in future research.

While backtracking can be slow, its guaranteed to find all solutions to a finite problem in a limited number of time, and thus remains a highly researched subject.

References

[1] Bessière, C., Maestre, A., Brito, I., and Meseguer, P. Asynchronous backtracking without adding links: a new member in the abt family. *Artificial Intelligence 161*, 1 (2005).

[2] Erdös, P., and Rényi, A. On Random Graphs, I. *Publicationes Mathematicae 6* (1959).

[3] igraph Core Team, T. igraph R package.

[4] Modi, P. J., Shen, W.-M., Tambe, M., and Yokoo, M. Adopt: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence 161*, 1–2 (2005).

[5] Nissim, R., Brafman, R. I., and Domshlak, C. A general, fully distributed multi-agent planning algorithm. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems* (2010).

[6] Petcu, A. *A Class of Algorithms for Distributed Constraint Optimization.* Amsterdam, Netherlands, 2009.

[7] Russell, S., and Norvig, P. *Artificial Intelligence: A Modern Approach*, 3rd ed. 2009.

[8] Silaghi, M. Framework for modeling reordering heuristics for asynchronous backtracking, 01 2006.

[9] Yokoo, M. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems, 01 1970.

[10] Yokoo, M., Durfee, E. H., Ishida, T., and Kuwabara, K. The distributed constraint satisfaction problem: formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering 10*, 5 (Sep 1998).

[11] Zivan, R., and Meisels, A. Synchronous vs asynchronous search on discsps.

[12] Zivan, R., and Meisels, A. Concurrent dynamic backtracking for distributed csps. In *Principles and Practice of Constraint Programming – CP 2004* (Berlin, Heidelberg, 2004), M. Wallace, Ed.