Task scheduling of multiple industrial robots through constraint programming

Tommy Kvant
ada09tkv@student.lu.se

January 8, 2015

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Jacek Malec, jacek.malek@cs.lth.se & Maj Stenmark,
maj.stenmark@cs.lth.se

Examiner: Klas Nilsson, klas.nilsson@cs.lth.se

Abstract

Keywords: key word

Acknowledgements

Contents

Chapter 1

Introduction

1.1 Background

More and more of the production in today's society is getting automated. Manufacturers want to cut cost and make the production more effective by eliminating the human work and replace it with robots. But there are drawbacks; robots are expensive and robots does not have the versatility of a human. This puts pressure on the robot manufacturers to develop robots that are more versatile. Thus eliminating the need for manufacturers to have multiple robots to do multiple tasks and thereby lowering costs. And also by making the robots more versatile the close the gap of what a human and robots are able to do.

Current robot setups usually have one robot performing one task all the time, as oppose to flexible robots which will be performing many different tasks and assemblies. This poses the demand for the scheduling of such robots to be flexible as well. A scheduling of a robot can be a time consuming task. Since manufacturers want as effective assemblies as possible, it can take from days to weeks to perfect an assembly schedule. This is not feasible if you want to use the robot for many different tasks and assemblies. In this thesis we would like to try and automate this scheduling process in order to cut down on the scheduling time. To accomplish this we will be using Constraint programming, as it provides a general interface to solve problems without needing to build a complete framework from scratch. Also, scheduling is a classical constraint problem, thus constraint programming suits this problem well.

One of those robots are ABB's robot YuMi®(formerly known as FRIDA). YuMi®is a dual armed robot made to work along side humans and able to perform the some of the most complex tasks, such as mount a nut or thread a needle.[1] It accomplishes this by using a wide variety of sensors, e.g. force sensor, visual sensors, etc. Usually robot replaces humans to perform dangerous or heavy tasks, YuMi®is mainly designed for small parts assembly, i.e. usually humans roles in todays manufacturing environment.

1.2 Problem specification

1.2.1 The setup

What does our problem look like

1.3 Related work

[brucker 2009] mentions the solving of 15x15 benchmark(15 jobs with 15 operations) being solved without heuristics (although not necessarily in CP).[2]

[Thörnblad] concludes that when a cell is part of an assembly flow, the use of targeting due dates is to prefer. Because makespan can exacerbate an already unreliable flow. The assembly we perform is not a part of a flow, and such we do not concern ourselves with maintaining a stable flow through the cell, but only to optimize the assembly in the cell.[7]

[Garey] shows that job shop problems for size $m \geq 2$ and $n \geq 3$ are NP-complete [4]

[yuan] says pure CP is only effective on small problems of FJSP. To effectively perform larger sizes, methods such as discrepancy search, large neighborhood search(LNS) or iterative flattening search. It also shows LNS together with Hybrid Harmonic Search produces good results.[9]

[ejenstam] [andra test med solvers]

1.4 Report structure

Chapter 2
Approach

2.1 Constraint Programming

Constraint programing is a *declarative* paradigm. This means that in contrast to *imperative* paradigm languages, such as C or Java, the focus of solving problems using constraint programming is on specifying the solution and not the algorithm to solve it. This is done by specifying the solution using *domain variables*, or simply variables, and *constraints*. Variables have a domain of values, meaning they can represent each value in their domain. Variables can often be, depending on language and solver, either integers, floating-points, boolean or symbolic, symbolic being a text or label. For example a symbolic variable representing a week would have the domain
$\{Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday\}$.

2.1.1 Constraints

Constraints are set up as relationships between the variables, and thereby limiting the domains of the variables. Integer domains are often used for variables, so for the rest of this section we will assume variables have integer domains. For this domain the following function symbols can be used: $+$, $\times$, $-$ and $\div$. The constraint relation symbols are $=$, $<$, $\leq$, $>$, $\geq$. Together with the function symbols and the constraint relation symbols, one can create simple constraint, called *primitive constraints*. An example of a primitive constraint is $X < Y$, i.e. the values in X's domain has to lower than in Y's. Primitive constraints can be used to create more complex constraints using the conjunctive connective $\wedge$. An example of this is $X < Y \wedge Y < 10$, i.e. Y has to be less than 10 and X has to be less than Y. Since all constraints has to hold when the model is evaluated, all constraints are joined in a conjunction.(**?**) The disjunctive connective $\vee$ is also available and can be used in the same way as $\wedge$. In some cases the logical implications; reverse implication ($\leftarrow$), forward implication ($\rightarrow$) and bi-implication ($\leftrightarrow$) are also available. Here, for example, the

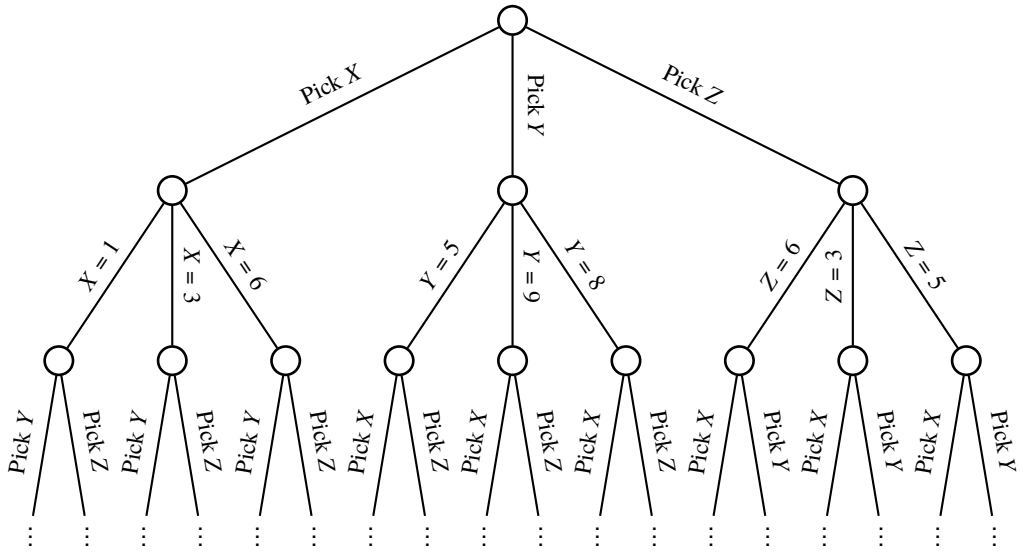

Figure 2.1: The beginning of the search space for the variables X, Y, Z, where $X = \{1, 3, 6\}$ $Y = \{5, 9, 8\}$ $Z = \{6, 3, 5\}$

implication works by taking two logical statement and if the first one evaluates to *true* the other statement should hold as well. For example $X = 0 \rightarrow Y > 5$ says that if $X = 0$ then Y should be larger than 5.

For example, we have a problem with the variables x and y. $x = 4$ and $y = \{1..10\}$. Here the x has the value 4 and can thereby only assume the value 4. y on the other hand can assume the values 1 to 10. This means a solution to this problem can be $x = 4$ and $y = 1$ or likewise $x = 4$ and $y = 5$, they are equally correct.
On this problem we can impose a constraint, for example $y > x$. Now we have set the constraint that y needs to be larger than x. And since x has a fixed known value we can directly see that $y > 4$, since $x = 4$. Now with this constraint, we can get rid of the lower part of y's domain and now $y = \{5..10\}$ instead. And now a viable solution can be $x = 4$ and $y = 7$, but not $x = 4$ and $y = 3$.

2.1.2 Global constraints

AllDifferent

Circuit

Cumulative

Global_Cardinality

2.1.3 Solver

A constraint programming program is consisting of many of these constraints and variables. When the solution is specified in a model, a *solver* runs the model. The goal of the solver is to satisfy all the constraints, i.e. set the domains of the variables so that the all follow the relationships of the constraints. This is called the *constraint satisfaction problem*, and can be defined as a triple $\langle Z, D, C \rangle$. $Z = \{x_1 \dots x_n\}$ is a finite set of all the variables in the solution, $D(x_i)$, $x_i \in Z$ is a set representing the domain of values the variable x_i can assume, C is the set of constraints imposed on the variables in Z. The solver accomplishes this by doing a *search* on the space of possibilities, i.e. the *search space*. The search space is in the form of a tree, where each branch is a selection of a variable where the variables domain is reduced into a smaller subset that conforms with the constraints. The solver traverses the tree in search for a solution. When all variables are set to conform with the constraints a solution is found. If the solver reaches a node where a variables domain becomes empty, it has to *backtrack* to a previous node from which it can choose a new variable to set, i.e. traversing a new branch of that node.

[8] [5] [6]

2.1.4 Reified

2.2 Job-shop scheduling problem

2.2.1 Flexible job-shop scheduling problem

2.3 MiniZinc

2.4 Model

This model is based on/inspired by the model in [ejenstam]. That model is centered around work performed in fixtures. So tasks can easily be labeled *tray* if it uses a tray, *fixture* if it uses a fixture, etc. This is common robot cell assembly procedures; take a component from a tray, put it in a fixture, get another component, mount the component on the the component in the fixture. But YuMi can perform much more complex tasks than that. We want to be able to schedule mounting tasks that does not incorporate a fixture. We

have used a similar way of generalizing tasks by labeling them with *tray*, *fixture*, etc. but extended it.

2.4.1 Input data

The solver takes a description of the robot cell in the form of a MiniZink data file. The file describes; the number of arms available, the tools available, the trays available, the fixtures available, etc. Here on after called *Cell Variables*. It also sets up a number of *Decision Varibles* which contains a set of values from which each *Decision Variable* can take.

Cell Variables

Decision Variables

2.4.2 Constraints

In this section some of the most important constraints for the model will be described. For a full list of used constraints see *Appendix A*, for the MiniZink code see *Appendix B*.

2.4.3 Filter

Domain Filter

Predecessor Filter

2.5 Solvers

2.5.1 G12

2.5.2 JaCoP

2.5.3 Gecode

2.5.4 OR-tools

2.5.5 Opturion CPX

2.5.6 Choco3

Chapter 3
Results

The results

G12

Filter	Pred & Dom		Pred		Dom		None	
Version	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1
Time (ms)	-	-	-	-	-	-	-	-
# of integer variables	174	292	174	313	154	269	154	290
# of boolean variables	162	97	162	97	142	106	142	106
# of arrays	32	46	32	46	30	44	30	44
# of constraints	2555	584	1018	584	2248	558	711	558
% refeid	7.16%	15.92%	17.97%	15.92%	7.25%	17.02%	22.92%	17.02%

JaCoP

Filter	Pred & Dom		Pred		Dom		None	
Version	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1
Time (ms)	635	-	1139619	-	-	-	-	-
# of integer variables	129	223	129	244	129	229	129	250
# of boolean variables	42	27	42	27	42	27	42	27
# of arrays	31	46	31	46	29	43	29	43
# of constraints	2318	425	781	425	2046	416	509	416
% refeid	2.71%	6.35%	8.06%	6.35%	3.07%	6.49%	12.37%	6.49%

Gecode

Filter	Pred & Dom		Pred		Dom		None	
Version	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1
Time (ms)	-	21	-	74019	-	65	-	73496
# of integer variables	129	223	129	244	129	250	129	450
# of boolean variables	42	27	42	27	42	27	42	27
# of arrays	31	50	31	50				
# of constraints	2316	421	779	421	1046	416	509	415
% refeid	2.72%	6.14%	8.08%	6.41%	3.07%	6.49%	12.37%	6.5%

or-tools

Filter	Pred & Dom		Pred		Dom		None	
Version	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1
Time (ms)	257	!	672	!	564	!	600	!
# of integer variables	129	223	129	244	129	229	129	250
# of boolean variables	42	27	42	27	42	27	42	27
# of arrays	31	50	31	50	29	43	29	43
# of constraints	2316	421	779	421	2046	416	509	416
% refeid	2.72%	6.41%	8.08%	6.41%	3.07%	6.49%	12.37%	6.49%

Opturion CPX - no warm start

Filter	Pred & Dom		Pred		Dom		None	
Version	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1
Time (ms)	25163591	!	-	!	7332356	!	-	!
# of integer variables	28736	9530	28736	28875	28716	9507	28716	28852
# of boolean variables	57284	18654	57284	57322	57264	18702	57264	57370
# of arrays	32	4694	32	14366	30	28209	30	14382
# of constraints	102519	33021	100982	100675	102212	33024	100675	100678
% refeid	55.91%	56.26%	56.76%	56.86%	56.06%	56.24%	56.92%	56.85%

Opturion CPX - warm start

Filter	Pred & Dom		Pred		Dom		None	
Version	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1
Time (ms)	26318399	!	-	!	7231907	!	-	!
# of integer variables	28736	9530	28736	28875	28716	9507	28716	28852
# of boolean variables	57284	18654	57284	57322	57264	18702	57264	57370
# of arrays	32	4694	32	14366	30	28209	30	14382
# of constraints	102519	33021	100982	100675	102212	33024	100675	100678
% refeid	55.91%	56.26%	56.76%	56.86%	56.06%	56.24%	56.92%	56.85%

Choco3

Filter	Pred & Dom		Pred		Dom		None	
Version	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1	1.6	2.0.1
Time (ms)	-	-	-	-	-	-	-	-
# of integer variables	129	223	129	244	129	229	129	250
# of boolean variables	42	27	42	27	42	27	42	27
# of arrays	31	50	31	50	29	43	29	43
# of constraints	2316	421	779	421	2046	416	509	416
% refeid	2.47%	6.41%	8.08%	6.41%	3.07%	6.49%	12.37%	6.49%

Chapter 4
Discussion

The discussion of the results

Concurrent tasks vs. en task med två maskiner. Två tasks ger mer generella constraints, man kan säga att all tasks tar upp en maskin alltid och att man begränsar det med ett constraint att vissa måste ske samtidigt. Man kan ha constraintet att alla tasks kan ha en predecessor. En task gör att man måste kunna säga att tasks kan ta upp till så många maskiner som finns tillgängliga -> fler constraints. Man måste kunna säga att en task ska kunna ha flera predecessors -> fler constraints.

Chapter 5
Conclusions

The conclusions

5.1 Further work

I [ejenstam] kommer de fram till att local search fungerar dåligt.[3] [yuan 2013] menar de att Large neighborhood Search (LNS) tillsammans med Hybrid Harmony Search (HHS) fungerar mycket väl med flexible job shop problem och löser problemet med begränsningen för CP vid dessa problem.[9] Kan vara värt att titta vidare på

Bibliography

[1] ABB. YuMi. `http://new.abb.com/products/robotics/yumi`.

[2] Peter Brucker. *Job-shop Scheduling Problem.* Springer, 2009.

[3] Joakim Ejenstam. Implementing a time optimal task sequence for robot assembly using constraint programming. Master thesis, Uppsala University, 2014.

[4] M. R. Garey, D. S. Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976.

[5] Kim Marriott and Peter J. Stuckey. *Programming with constraints : an introduction*. Cambridge, Mass. : MIT Press, cop., 1998.

[6] The G12 Team. *Specification of Zinc and MiniZinc v.1.6*. NICTA, Victoria Research Lab, Melbourne, Australia, August 2011.

[7] Karin Thörnblad, Ann-Brith Strömberg, Michael Patriksson, and Torgny Almgren. An efficient algorithm for solving the flexible job shop scheduling problem. In *25th NOFOMA conference proceedings, June 3-5 2013, Göteborg, Sweden*, page 15, 2013.

[8] Edward Tsang. *Foundations of constraint satisfaction*. Academic Press, 1993.

[9] Yuan Yuan and Hua Xu. An integrated search heuristic for large-scale flexible job shop scheduling problems. *Computers & Operations Research*, 40(12):2864–2877, 2013.

Appendices

Appendix A
Model

Appendix B
Tool Manuals