Constructive Disjunction Revisited

Jörg Würtz[1] and Tobias Müller[2]

[1] German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, Email: wuertz@dfki.uni-sb.de
[2] Universität des Saarlandes, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, Email: tmueller@dfki.uni-sb.de

Abstract. Finite Domain Programming is a technique for solving combinatorial problems like planning, scheduling, configuration or timetabling. Inevitably, these problems employ disjunctive constraints. A rather new approach to model those constraints is constructive disjunction, whereby common information is lifted from the alternatives, aiming for stronger pruning of the search space. We show where constructive disjunction provides for stronger pruning and where it fails to do so. For several problems, including a real-world college timetabling application, benefits and limitations of constructive disjunction are exemplified. As an experimental platform we use the concurrent constraint language Oz.

1 Introduction

Constraint Logic Programming over finite domains, i.e., finite sets of natural numbers, has shown up to be able to handle real-life problems like planning, scheduling, configuration or timetabling (e.g. [5, 17, 7]). Constraints allow to prune the search space a priori. A constraint problem is solved by interleaving pruning by constraints and speculative assignment (choices).

Inevitably in most real-world applications are disjunctive constraints. Hence, it is natural that disjunctive constraints play a crucial role in artificial intelligence fields like planning, configuration, knowledge representation or expert systems. For example in timetabling there is the constraint that two lectures must not overlap in time while in computer hardware configuration we may have the constraint that two cards must not overlap in space.

Disjunctive constraints can be dealt with in several ways. They may be modelled as choice-points where constraints become active only after choosing one alternative [5] or by reasoning about the truth value of alternatives, without making choices, i.e., checking whether a clause of the disjunction is true or false (the so-called reified approach). A rather new approach is called constructive disjunction, which lifts common information from the alternatives. As an example consider the disjunction $A + 7 \leq B \ \vee \ B + 7 \leq A$ with $A, B \in \{1, \ldots, 10\}$.

In G. Görz and S. Hölldobler, editors, *20th German Annual Conference on Artificial Intelligence*, volume 1137 of *Lecture Notes in Artificial Itelligence*, pages 377–386, Dresden, Germany, September 1996. Springer-Verlag.

Its meaning might be that the task A with duration 7 must precede task B or, alternatively, that task B with duration 7 must precede task A, i.e., A and B must not overlap in time. In the left alternative, A can only take the value 1, 2 or 3, and B the value 8, 9 or 10, and vice versa for the right alternative. The reified approach only checks if an alternative becomes inconsistent, in which case the other alternative is installed. Hence, no further pruning arises in our example. But one can do better. We can derive that neither A nor B will take the value $4, 5, 6$ or 7. This information is lifted by constructive disjunction resulting in (possibly dramatic) pruning of the search space.

While there exist several papers on constructive disjunction [3, 8, 9, 17], there is not much published experience about the usefulness of constructive disjunction for practical applications. But this experience is crucial for considering constructive disjunction worthwile for one's own applications. The given examples in the literature are sometimes incorrect or misleading in that the problems could be better solved by other approaches or do not scale up expectedly. Some readers might feel this view this to be too negative, but to state cases where constructive disjunction does not pay off can prevent developers from getting stuck or to waste a lot of efforts. This is especially important for new techniques.

By comparing the different concepts of modelling disjunctive constraints in Section 3 we extract useful applications for constructive disjunction but state also cases (like scheduling) where more advanced constraint techniques are known which beat it by orders of magnitude. There are also occasions where constructive disjunction can be useful in principle (e.g. packing problems) but where examples in literature are better tackled by reified constraints. Experiences in [8] and our work on college timetabling [7] supports our thesis that constructive disjunction is a powerful and flexible means to improve search (speedup of an order of magnitude) in cases, where the problem is rather complex and one needs to find new heuristics for guiding the search.

As an experimental platform we use Oz [15, 16], which allows to evaluate the different concepts of modelling disjunctive constraints in a single system. Oz is a new language providing for concurrency and object-oriented programming, which makes it well-suited for applications in AI. But what makes Oz unique is its expressiveness and flexibility for problem-solving. By means of a user-accessible search combinator, search strategies can be individually programmed and problem-solving can be guided by inventing and exploring various heuristics. Moreover, a rich set of constraints allows to prune the search space in an efficient way.

2 Computation in Oz

2.1 Computation Model

The central notion in Oz is a computation space [14]. A computation space consists essentially of a constraint store and a set of associated tasks.

Constraints residing in the constraint store are equations between variables and/or values, as for instance atoms or integers, and constraints $x \in D$ where

D is a finite domain, i.e., a finite set of nonnegative integers. Oz provides efficient algorithms to decide satisfiability and implication for the constraints in the constraint store.

Tasks inspect the constraint store and are reduced if the store contains sufficient information. On reduction a task may impose further constraints on the store or spawn new tasks. The computation space a task is spawned in is called its host space. A typical task is a disjunction like


```
or X::3#6 X::4#10
[] Y::1000#1050
end
```

which spawns local computation spaces, e.g. the local store of the first clause holds the constraint `X`$\in \{3,\ldots,6\} \cap \{4,\ldots,10\}$ (juxtaposition is read as conjunction and, e.g., `X::3#6` denotes the constraint `X`$\in \{3,\ldots,6\}$). If the store of the host space implies for example `Y`$\in \{0,\ldots,10\}$, the second clause fails and, thus, the constraint `X`$\in \{4,\ldots,6\}$ will be added to the host space. If one alternative is implied, e.g. `Y` $= 1020$ holds, the disjunctive task simply ceases to exist.

2.2 Propagators

For more expressive constraints, like $x + y = z$, deciding their satisfiability is not computationally tractable. Such constraints are not contained in the constraint store but are modelled by so-called propagators.

A propagator P can be thought of as a long-lived task which amplifies the constraint store S. The propagator can tell the store a constraint C whenever the conjunction $S \wedge P$ implies the constraint C. A propagator must remain in a computation space until it is implied by the constraint store. For instance, assume a store containing `X,Y,Z` $\in \{1,\ldots,10\}$. The propagator `X+Y<:Z` amplifies the store to `X,Y` $\in \{1,\ldots,8\}$ and `Z` $\in \{3,\ldots,10\}$ (since the other values cannot satisfy the constraint).[3] Telling the constraint `Z=5` causes the propagator to strengthen the store to `X,Y` $\in \{1,\ldots,3\}$. Imposing `X=3` makes the propagator telling `Y=1`.

2.3 Disjunctive Constraints

In this section we discuss several ways to express disjunctive constraints in Oz for the example $|X - 1| = Y$ with $X \in \{1,\ldots,5\}$ and $Y \in \{0,1,5\}$. The constraint is equivalent to the disjunction $X - 1 = Y \vee 1 - X = Y$ and the Oz expressions `X::1#5 Y::0|1|5|nil`.

Disjunctive Tasks. The example can be formulated as

[3] An appended colon marks a finite domain propagator.

```
or X - 1 =: Y
[] 1 - X =: Y
end
```

Since the global information on variables is visible in the local stores, the propagator in the first clause amplifies the first local store to `X` $\in \{1, 2\}$ and `Y` $\in \{0, 1\}$. The store of the second clause contains the constraints `X=1` and `Y=0`. Imposing the constraint `X=2` makes the second clause fail. Thus, the remaining local computation space is lifted. The added propagator `X-1=:Y` imposes now the constraint `Y=1`.

Choice-Points. Oz provides also for disjunctive tasks, which can be used as choice-points (the keyword **or** is replaced by **dis**). The choice is delayed until no other computation (like constraint propagation) can take place, i.e., the computation space is stable (for details see [13]). The **dis**-task additionally prunes the search space by adopting the operational semantics of the **or**-task (distinguishing it from the choice-points as in `CHIP` [5]).

Reified Constraints. Reified constraints are propagators that reflect the validity of a constraint into a $\{0, 1\}$-valued variable. Because reified constraints avoid local computation spaces, they can be implemented more efficiently than disjunctive tasks.

Assume we want to reify a propagator P in a variable B (in Oz we write B`=(`P`)`). If B is constrained to 1 (resp. 0), then P (resp. its negation $\neg P$) is installed. Vice versa, if P is valid (resp. unsatisfiable), B is bound to 1 (resp. 0). Our example becomes:

```
R1 = (X - 1 =: Y)
R2 = (1 - X =: Y)
R1 + R2 >: 0
```

Since no propagator nor its negation is implied, the store is not changed. Telling the constraint `X=2` is inconsistent with the propagator `1-X=:Y`, which causes `R2` to be constrained to 0. The inequality `R1+R2>:0` amplifies the store by `R1=1` which in turn causes the propagator `X-1=:Y` to be installed. This propagator tells immediately the constraint `Y=1`.

While constructive disjunction strengthens the pruning power of constraints, reified constraints add more expressivity to a system (like soft constraints or preferences in cost functions of branch & bound optimization). The cardinality combinator of [6], which states that the number of true constraints of a given set must be in a given interval of integers, can be modelled with reified constraints straightforwardly.

Constructive Disjunction. Assume a computation space containing a store S. A disjunctive combinator with n clauses spawns n local computation spaces, which consist of tasks $T_1, \ldots, T_n$ and constraint stores $S_1, \ldots, S_n$, respectively. For making the disjunction constructive we have to lift common information from

the clauses. We merge each store S_i with the store S and call S'_i the resulting store after computation has terminated. Let L be the smallest set of constraints such that all S'_i imply L. We now lift L by adding it to S. For finite domains this means to compute the union of the domains of the occurring variables. In [8, 17] a more general form of constructive disjunction is proposed where also the tasks T_i are merged with the tasks T of the host space. In [3] it was shown that this general approach is very expensive but gains only a little compared to the variant we provide. Our approach is also justified by the performance results obtained in [10]).

Oz syntactically supports constructive disjunction (called CD in the sequel) by the keywords **condis** and **end**. The clauses can contain arbitrary finite domain propagators. Picking up our example, we obtain

```
condis X - 1 =: Y
[] 1 - X =: Y
end
```

But in contrast to the previous versions of disjunctions, `X` and `Y` are immediately constrained to $\texttt{X} \in \{1,2\}$ and $\texttt{Y} \in \{0,1\}$. This is the result of lifting common information from the clauses, i.e., $\texttt{X} \in \{1,2\}_{first} \cup \{1\}_{second}$ and $\texttt{Y} \in \{0,1\}_{first} \cup \{0\}_{second}$. Telling `X=2` fails the second clause and promotes the first clause, which results in telling `Y=1`.

3 Applications

In this section we point out what kinds of constraints and problem solving techniques benefit from CD. Of course, we cannot examine all possible applications, but the chosen three problems allow to gain important insights for the use of different models for disjunctive constraints.

3.1 General Remarks

Due to its definition, CD prunes the search space only for those variables occurring in all alternatives of the disjunction. It may tighten the bounds of variables like `Z` in

```
condis X + XDur =<: Z
[] Y + YDur =<: Z
end
```

where `X`,`Y` might denote start times of tasks and `XDur`, `YDur` their respective durations, i.e., the task `Z` must be delayed until `X` and `Y` are finished.

Assume the distance of X and Y to be 4, i.e., the constraint $\mid X - Y \mid = 4$, and the domains $X, Y \in \{1, \ldots, 5\}$. CD leads to $X, Y \in \{1, 5\}$, i.e., holes are cut in the domains. But for the sake of efficiency many constraint languages reason mainly on the bounds of domains (for instance in Oz, we approximate s`=:`t by s`=<:`t s`>=:`t). Thus, CD may prune the search space, but other constraints may not profit from the occurring domain reduction. Only those constraints that

reason on the whole domain benefit from these holes. As an example consider the constraint that at least one of X and Y must be 3. This constraint fails with the distance-constraint $\mid X - Y \mid = 4$, if $X, Y \in \{1, \ldots, 5\}$ holds.

Because constraint propagation is usually incomplete, choices must be made to assign values to variables (called labelling). Labelling strategies, which reason on the size of the domains, like first-fail (choose the variable with the smallest domain first), may benefit from CD's domain pruning: More information on variables is made available.

But in any case, one has to be aware that CD is computationally more expensive than reified constraints (see also the following sections). Hence, if the effects of CD cannot be exploited, it may slow down an application.

3.2 Square Packing

The problem is to pack a given set of squares into a master-rectangle such that all squares are used and no squares overlap [17]. The constraint that two squares at (XA, YA) and (XB, YB) with sizes SA and SB must not overlap (note that the coordinates are finite domains) employs CD:

```
condis XA + SA =<: XB   % X-clause
[] XB + SB =<: XA       % X-clause
[] YA + SA =<: YB       % Y-clause
[] YB + SB =<: YA       % Y-clause
end
```

The X-clauses (resp. Y-clauses) express that squares do not overlap horizontally (resp. vertically). As soon as only the two X- or the two Y-clauses are left (because the others are failed), the domains may be reduced by CD. For example $XA + 8 \leq XB \vee XB + 8 \leq XA$ with $XA, XB \in \{1, \ldots, 10\}$ leads to pruning XA and XB to $\{1, 2, 9, 10\}$. As an additional constraint we have that for each X- (and Y-) coordinate `P` the sum of the square sizes $\texttt{S}_i$ intersecting this coordinate must be less than the respective length `L` of the rectangle: $\sum \texttt{B}_i\texttt{*S}_i\texttt{=<:L}$, where $\texttt{B}_i\texttt{=(X}_i\texttt{::P-S}_i\texttt{+1\#P)}$. The occurring holes by CD propagation can only fail the reified constraint, i.e., $\texttt{B}_i\texttt{=0}$. Thus, CD does not lead to further pruning for this problem.

In [3] disjunctive constraints are modelled by CD and reified constraints. The authors claim that the constructive approach solves the problem with less choice-points for first-fail labelling, i.e., the occurring holes lead the search to a solution earlier. But this result is incorrect[4]; for one reported example both CD and reified constraints lead to the same number of choice-points, while for the other example, the reified approach needs half of the choice-points as needed

[4] We assume that this is due to a compiler error since replacing the suspicious code with semantical equivalent code allows us to reproduce in Agents [3] the same results as in Oz. For the first example we have rectangle length `L=10` and the sizes are `[6 4 4 4 2 2 2 2]`, and for the second we have `L=20` and `[9 8 8 7 5 4 4 4 4 3 3 3 2 2 1 1]` for the sizes.

by the constructive approach. If one uses naive labelling, the number of choice-points is the same for both approaches and examples. For this application, CD is not a good choice.

If one uses a special labelling strategy [17] (since first-fail does not scale up for larger problems), the number of choice-points are the same for all approaches. The following table gives the runtimes taken on a Sparc10 for two examples.

	CD	Reified	**or**	CD/Rei	CD/**or**
Expl. 1	780ms	670ms	770ms	1.16	1.01
Expl. 2	2190ms	1640ms	2970ms	1.34	0.74

While CD does not pay off for this application, more complex packing problems may benefit from it (e.g. if a minimum stock must be guaranteed, like $\sum$`B`$_i$`*S`$_i$`>:L`; see also 3.4).

3.3 Scheduling

The constraint that two tasks `X` and `Y` with durations `XDur` and `YDur`, respectively, must not overlap in time if they require the same resource, occurs often in scheduling:

```
condis X + XDur =<: Y
[] Y + YDur =<: X
end
```

As seen above, this constraint may cut holes in domains. In [8, 3], the bridge-problem is used as an example. The problem is to find the optimal schedule for building a bridge with limited resources and some additional time constraints. In [3] choice-points without active pruning, reified constraints and CD are compared for finding the first solution by first-fail labelling. The number of labelling steps grows from CD via choice-points to reified constraints. But this example is misleading because the optimal solution (in which one is interested for this problem) cannot be found after more than 10 million choices also with the CD-approach.

By the choice-point approach the optimal solution can be found, but reordering the choices leads to runtimes which are several magnitudes worse. Hence, this approach is not robust against different formulations of the problem

Therefore, in [8] the choice-points are extended by constraint lifting, according to CD. They obtain the optimal solution and the proof of optimality in 881 backtracking steps selecting the next choice-point by a strategy considering the involved variable domains in the disjunctions (being robust against reordering).

But modeling the disjunctive constraints by CD is too weak to solve really hard scheduling problems. There are constraint techniques inspired from Operations Research, which allow to solve hard scheduling problems [4, 2] (for example the proof of optimality for the notorious MT10 problem (see [11]) needs about 2000 choices in Oz). These techniques exploit domain specific knowledge. While for hard problems techniques like task intervals [4] or cumulative constraints [1] are used, the bridge problem can be solved by a rather naive labelling strategy

and reified constraints needing only 176 backtracking steps. We choose the most demanded resource first and schedule it completely. For this resource we find the tasks which can be first on it, choose the one with the smallest possible start time and state constraints that the remaining tasks are scheduled after the chosen one. The disjunctions are modelled by reified constraints while the choices are made by the **dis**-task of Oz, i.e., we combine different ways of modelling disjunctive constraints.[5]

3.4 Real-world Applications

In [8] a more complex aircraft sequencing application is reported, where CD results in a speedup by a factor of 6.

At DFKI we have solved a real-life college timetabling problem [7] with several complex constraints (like a limited number of rooms or that teachers must teach at most three days a week). A frequently occuring constraint is that lectures must not overlap in time. This constraint was already seen above and CD is worthwile to be considered here. Further we need to express that a certain number of lectures may overlap. Therefore non-overlapping is reified in the variable `B` :

```
condis B=:1  X+XDur>:Y  Y+YDur>:X
[] B=:0  X+XDur=<:Y
[] B=:0 Y+YDur=<:X
end
```

If `B=0` is known, CD pays off because the disjunction results in a simple non-overlapping constraint. Modelling these disjunctions constructively leads to stronger pruning because there occur reified constraints `B=(X::D)`. Ocurring holes may constrain `B` to 0. Because the `B`'s essentially occur in equations $\sum$`B`$_i$`=:L` (like that the lectures of a teacher must be on `L` days), CD pays off for propagation.

As a labelling strategy we use a modified first-fail to select the variables, which benefits also from the more pruned domains. In comparison to reified constraints, the resulting speedup by using CD is about an order of magnitude (see also [7]). For this example we have a combination of more pruning and benefitting labelling heuristics.

4 Conclusion

We have compared constructive disjunction with other ways to model disjunctive constraints and have shown that constructive disjunction does not pay off for problems like scheduling where domain specific knowledge can be used in a constraint setting. But it is very useful for applications which are rather complex, where a special purpose strategy is unknown but flexibility is required. Here it can prune the search space and, thus, allows for better heuristics by providing

[5] If we model the disjunctions by CD and **or**-tasks,respectively, we obtain the runtime relations CD/Reified=1.8 and CD/**or**=0.6.

more information on the problem variables. For real-world applications one needs also choice-points and reified constraints to evaluate the most efficient modelling of disjunctive constraints.

Remark and Acknowledgements

The research reported in this paper has been supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (FTZ-ITW-9105), the Esprit Project ACCLAIM (PE 7195) and the Esprit Working Group CCL (EP 6028). We thank Joachim Niehren for valuable comments on this paper and Björn Carlson for making a prototype of Agents available for testing. The documentation of the DFKI Oz system is available from the programming systems lab of DFKI through anonymous ftp from `ps-ftp.dfki.uni-sb.de` or through WWW from `http://ps-www.dfki.uni-sb.de/oz/`.

References

1. A. Aggoun and N. Beldiceanu, 'Extending CHIP in order to solve complex scheduling and placement problems'. Mathl. Comput. Modelling, volume 17, number 7, pp. 57–73, (1993).
2. P. Baptiste and C. Le Pape, 'A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling', in *IJCAI*, (1995).
3. B. Carlson and M. Carlsson, 'Compiling and executing disjuctions of finite domain constraints', in *ICLP*, pp. 117–131, (1995).
4. Y. Caseau and F. Laburthe, 'Improved clp scheduling with task intervals', in *ICLP*, (1994).
5. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier, 'The constraint logic programming language CHIP', in *FGCS*, pp. 693–702, (1988).
6. P. Van Hentenryck and Y. Deville, 'The Cardinality Operator: A New Logical Connective for Constraint Logic Programming', in *ICLP*, pp. 745–759, (1991).
7. M. Henz and J. Würtz, 'Using Oz for college time tabling', in *International Conference on the Practice and Theory of Automated Time Tabling*, pp. 283–296, (1995).
8. J. Jourdan and T. Sola, 'The versatility of handling disjunctions as constraints', in *PLILP*, pp. 60–74, (1993).
9. T. le Provost and M. Wallace, 'Generalized constraint propagation over the CLP scheme', *Journal of Logic Programming*, **16**, 319–359, (1993).
10. T. Müller and J. Würtz, 'Constructive Disjunction in Oz', in *11. Workshop Logische Programmierung*, (1995).
11. J.F. Muth and G.L. Thompson, *Industrial Scheduling*, Prentice Hall, 1963.
12. T. Le Provost and M. Wallace, 'Generalized constraint propagation over the CLP scheme', *The Journal of Logic Programming*, **16**(3 & 4), 319–359, (1993).
13. C. Schulte and G. Smolka, 'Encapsulated search in higher-order concurrent constraint programming', in *ILPS*, pp. 505–520, (1994).
14. G. Smolka, 'The definition of Kernel Oz', DFKI Oz documentation series, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, (1994).

15. G. Smolka, 'The Oz programming model', in *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, 324–343, Springer-Verlag, (1995).
16. *DFKI Oz Documentation Series*, eds., G. Smolka and R. Treinen, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1995.
17. P. Van Hentenryck, V. Saraswat, and Y. Deville, 'Design, implementation and evaluation of the constraint language cc(FD)', in *Constraints: Basics and Trends*, Lecture Notes in Computer Science, vol. 910, Springer Verlag, (1995).

This article was processed using the LaTeX macro package with LLNCS style