Modelling Dynamic Programming-Based Global Constraints in Constraint Programming

Andrea Visentin[1], Steven Prestwich[1], Roberto Rossi[2], and Armagan Tarim[3]

[1] Insight Centre for Data Analytics, University College Cork, Ireland
s.prestwich@cs.ucc.ie,andrea.visentin@insight-centre.org
[2] University of Edinburgh Business School, Edinburgh, UK
Roberto.Rossi@ed.ac.uk
[3] Cork University Business School, University College Cork, Ireland
armagan.tarim@ucc.ie

Abstract. Dynamic Programming (DP) can solve many complex problems in polynomial or pseudo-polynomial time, and it is widely used in Constraint Programming (CP) to implement powerful global constraints. Implementing such constraints is a nontrivial task beyond the capability of most CP users, who must rely on their CP solver to provide an appropriate global constraint library. This also limits the usefulness of generic CP languages, some or all of whose solvers might not provide the required constraints. A technique was recently introduced for directly modelling DP in CP, which provides a way around this problem. However, no comparison of the technique with other approaches was made, and it was missing a clear formalisation. In this paper we formalise the approach and compare it with existing techniques on MiniZinc benchmark problems, including the flow formulation of DP in Integer Programming. We further show how it can be improved by state reduction methods.

Keywords: Constraint programming · Dynamic programming · MIP · Encoding

1 Introduction

Constraint Programming (CP) is one of the most active fields in Artificial Intelligence (AI). Designed to solve optimisation and decision problems, it provides expressive modelling languages, development tools and global constraints. An overview of the current status of CP and its challenges can be found in [9].

The Dynamic Programming (DP) approach builds an optimal solution by breaking the problem down into subproblems and solving each to optimality in a recursive manner, achieving great efficiency by solving each subproblem once only.

There are several interesting connections between CP and DP:

– DP has been used to implement several efficient global constraints within CP systems, for example [17, 10] . A tag for DP approaches in CP is available in the global constraint catalogue [1].

- [8] used CP to model a DP relaxed version of the TSP after a state space reduction.
- The DP feature of solving each subproblem once only has been emulated in CP [5], in Constraint Logic Programming languages including Picat [19], by remembering the results of subtree searches via the technique of *memoization* (or *tabling*) which remembers results so that they need not be recomputed. This can improve search performance by several orders of magnitude.
- DP approaches are widely used in binary decision diagrams and multi-value decision diagrams [3].

Until recently there was no standard procedure to encode a DP model into CP. If part of a problem required a DP-based constraint that is not provided by the solver being used, the modeller was forced either to write the global constraint manually, or to change solver. This restricts the usefulness of DP in CP.

However, a new connection between CP and DP was recently defined. [16] introduced a technique that allows DP to be seamlessly integrated within CP: given a DP model, states are mapped to CP variables while seed values and recurrence equations are mapped to constraints. The resulting model is called a *dynamic program encoding* (DPE). Using a DPE, a DP model can be solved by pure constraint propagation without search. DPEs can form part of a larger CP model, and provide a general way for CP users to implement DP-based global constraints. In this paper we explore DPEs further:

- We provide a formalization of the DPE that allows a one-to-one correspondence with a generic DP approach.
- We compare the DPE with a widely known *variable redefinition* technique for modelling DP in Mixed Integer Programming (MIP) [13], and show its superior performance.
- We show that the performance of a DPE can be further improved by the application of state reduction techniques.
- We show that is possible to utilize it to model some DP-based constraints in MiniZinc.

The paper is organised as followed. Section 2 formalises the DPE technique to allow a one-to-one mapping of the DP approaches. Section 3 applies DPE to the shortest path problem in MiniZinc. We study its application to the knapsack problem and show how it can strongly improve the way we represent DP in CP, and how to use state reduction techniques to improve performance. Section 4 concludes the paper and discusses when this technique should be used.

2 Method

In this section we formalize the DPE. As mentioned above, it models every DP state with a CP variable, and the seed values and recurrence relations with constraints. [16] introduces the technique informally, and here we give a more formal description based on the definition of DP given in [4].

Many problems can be solved with a DP approach that can be modelled as a shortest path on a DAG, for example the knapsack problem [11] or the lot sizing problem [7]. We decided to directly use the shortest path problem, which was already used as a benchmark for the MiniZinc challenge [18]. One of the most famous DP-like algorithms is used to solve this problem: Dijkstra's algorithm. We will use Figure 1 to help the visualization of the problem.

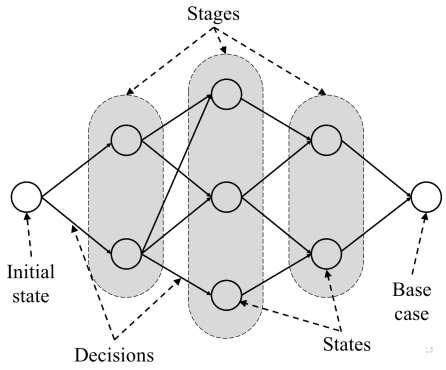

Fig. 1. Graph relative to a generic shortest path problem.

Most DPs can be described by their three most important characteristics: *stages*, *states* and *recursive optimization*. The fundamental feature of the DP approach is the structuring of optimization problems in *stages*, which are solved sequentially. The solution of each stage helps to define the characteristics of the next stage problem. In Figure 1 the stages are represented in grey. In the DPE the stages are simply represented as groups of states, and it is important that the stages depends only on the next stage's states. Since the graph is acyclic we can divide the stages into group of nodes that can not access the nodes of the previous stages, and can not be accessed by the nodes of the next stages.

To each stage of the problem are associated one or more states. These contain enough information to make future decisions, without regard to how the process reached the current state. In the DPE the states are represented by CP variables and they contain the optimal value for the subproblem represented by that state. In the graphical representation they are the nodes of the graph. In Dijkstra's algorithm these variables contain the length of the shortest path from that node to the sink. We can identify two particular type of stages: the *initial state* that contains the optimal solution for the whole problem and no other state solution is based on its value; and *base cases* or final states that are the solution of the smallest problems. The solutions for these problems do not depend on any other state. In Figure 1 they are represented by the source and the sink of the graph.

The last general characteristic of the DP approach is the *recursive optimization* procedure. The goal of this procedure is to build the overall solution by

solving one stage at time and linking the optimal solution to each state to other states of subsequent stages optimal solution. This procedure is generally based on a *backward induction* process. The procedure has numerous components, that can be generally modelled as a *functional equation* or *Bellman equation* [2] and *recursion order*.

In the DPE the functional equation is not a single equation, but is applied to every state via a constraint. This constraint contains an equality that binds the optimal value obtainable to that states to the ones of the next stages involved. For every state we have a set of feasible *decisions* that can lead to a state of the next stage, which in the graph are represented by the edges leaving the associated node: if used in the shortest path it means that decision is taken. In the constraint is included also the *immediate cost* associated with each decision, which is the value that is added to or subtracted from the next stage state variables. In Figure 1 these costs are represented by the weights of the involved edges. In the shortest path problem, the constraint applied to each (non-sink) state assigns to the node's CP variable the minimum of the reachable with one edge node's CP variables, plus the edge cost.

The important difference between the encodings is the order in which the states are explored and resolved. In DP they are ordered in such a way that each state is evaluated only when all the subsequent stages are solved, while in the encodings the ordering is delegated to the solvers. In the MIP flow formulation it is completely replaced by a search on the binary variables, while in the DPE it is done by constraint propagation, which depends on the CP solver implementation of the propagators. This approach is more robust than search, which in the worst case can spend an significant time exploring a search subtree. The optimality of the solution is guaranteed by the correctness of the DP formulation.

3 Computational Results

We aim to make all our results replicable by other researchers. Our code is available online at: https://github.com/andvise/dpincp. We also decided to use only open source libraries. In the first experiment we used MiniZincIDE 2.1.7, while the second part is coded in Java 10. We used 3 CP solvers: Gecode 6.0.1, Google OR-Tools 6.7.2 and Choco Solver 4.0.8. We used as MIP solvers: COIN-OR branch-and-cut solver, IBM ILOG CPLEX 12.8 and Gurobi 8.0.1. All experiments are executed on an Ubuntu system with an Intel i7-3610QM, 8GB of RAM and 15GB of swap memory.

3.1 Shortest path in MiniZinc

MiniZinc is a standard modelling language for CP. It provides a set of standard constraints, with ways of decomposing all of them to be solved by a wide variety of solvers. To achieve standardization the MiniZinc constraints catalogue is very limited. Only constraints that are available in all its solvers, or that can be decomposed into simpler constraints, are included. The decomposition is generally

done in a naive way, causing poor performance. This is true of all the DP-based constraints in particular.

In this section we focus on applications of the DPE in MiniZinc. We aim to apply this new technique to the shortest path problem and solve it with the DPE of the Dijkstra algorithm we used as an example in the method section. The shortest path was one of the benchmarks of the MiniZinc challenge [18]. The current reduction is based on a flow formulation on the nodes of the graph, which regulates the flow over each node and requires a binary variable for each edge indicating whether an edge is used or not. This is the same encoding proposed by [13].

Our implementation is based on Dijkstra's algorithm: every decision variable contains the shortest distance to the sink node. The formulation is shorter and more intuitive than the previous one.

We compared the methods on the 10 available benchmark instances. We used the MiniZincIDE and Gecode as solver, with 20 minutes as a time limit. Table 1 shows the results of the computations. When the flow formulation finds a good or optimal solution quickly, the DPE is approximately twice as fast. However, the flow formulation requires search that can take exponential time, and it is unable to find a solution before timeout occurs. The most interesting result is that, by using only constraint propagation, DPE performance is robust and only marginally affected by the structure of the instances. In some cases, for example instance 7, the flow formulation finds an optimal solution but takes a long time to prove optimality, in which case the DPE is more than 4 orders of magnitude faster.

Table 1. Time required to complete the computation of the 10 benchmark instances in Gecode. '-' represents a timeout.

CP solver	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Dijkstra	23 ms	19 ms	18 ms	17 ms	24 ms	20 ms	25 ms	23 ms	20 ms	29 ms
Flow formulation	-	50 ms	60 ms	571 ms	46 ms	-	47 ms	1 182 s	4 504 ms	-

The DPE requires a smaller number of variables, since it requires only one for each node. On the contrary, the flow formulation requires a variable for each edge. This is without taking in account the number of additional variables created during the decomposition.

The DPE cannot rival a state-of-the-art shortest path solver in terms of performance, in the case of *parameterised* shortest path problems, in which the costs of the edges are influenced by other constraints. However, the DPE allows a more flexible model than a specific global constraint and a more efficient model in MiniZinc.

We repeated the above experiment using a MIP solver instead of CP. Table 2 contains the results of the 10 instances solved using COIN-OR branch-and-cut solver. Interestingly, the situation is inverted: the flow formulation performs efficiently while the DPE fails to find an optimal solution in many cases. This

is due to the high number of auxiliary discrete variables needed by the MIP decomposition of the *min* constraint. Because of this the DPE loses one of its main strengths: DP computation by pure constraint propagation. Moreover the MIP can take advantage of the unimodularity of the matrix, as mentioned before. We therefore recommend the usual flow-based formulation for MIP and the DPE for CP.

Table 2. Time required to complete the computation of the 10 benchmark instances in CBC COIN. '-' represents a timeout.

MIP solver	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Dijkstra	375 s	64 ms	-	-	-	20 667 ms	61 ms	-	138 ms	303 ms
Flow formulation	31 ms	39 ms	34 ms	40 ms	46 ms	35ms	36 ms	40ms	37 ms	53ms

3.2 Knapsack Problem

We now apply the DPE to the *knapsack problem* [11] because it is a widely known NP-hard problem, it has numerous extensions and applications, there is a reduction in MiniZinc for this constraint, and it can be modelled with the technique proposed by [13]. We consider the most common version in which every item can be packed at most once, also known as the 0-1 knapsack problem [15]. Research on this problem is particularly active [12] with many applications and different approaches.

The problem consists of a set of items I with volumes $\boldsymbol{v}$ and profits $\boldsymbol{p}$. The items can be packed in a knapsack of capacity C. The objective is to maximize the total profit of the packed items without exceeding the capacity of the knapsack. The binary variables $\boldsymbol{x}$ represent the packing scheme, x_i is equal to 1 if the item is packed, 0 otherwise. The model is:

$$\max \boldsymbol{x} \cdot \boldsymbol{p} \tag{1a}$$

$$\text{s.t.} \quad \boldsymbol{x} \cdot \boldsymbol{v} \leq C \tag{1b}$$

$$x \in \{0, 1\} \tag{1c}$$

This model can be directly implemented in CP or in MIP. To solve the binary knapsack problem we the well known DP-like algorithm described in [11], refer to the source for the full description of the algorithm. Utilizing the structure of a DP approach in the previous section: $J[i, j]$ with $i \in I$ and $j \in [0, C]$ are the states of our DP. Each $J[i, j]$ contains the optimal profit of packing the subset of items $I_i = (i, \ldots, n)$ in a knapsack of volume j.

The formulation can be represented by a rooted DAG, in this case a tree with node $J[1, C]$ as root and nodes $J[n, j], j \in [0, C]$ as leaves. For every internal node $J[i, j]$ the leaving arcs represent the action of packing the item i, and their weight is the profit obtained by packing the i-th item. A path from the root to a leaf is equivalent to a feasible packing, and the longest path of this graph is the optimal solution. If we encode this model using a DPE, creating all the CP

variables representing the nodes of the graph, then it is solved by pure constraint propagation with no backtracking.

We use this problem to show the potential for speeding up computational times. With the DPE implementation we can use simple and well known techniques to reduce the state space without compromising the optimality. For example, if at state $J[i, j]$ volume j is large enough to contain all items from i to n (all the items I might pack in the next stages) then we know that the optimal solution of $J[i, j]$ will contain all of them, as their profit is a positive number. This pruning can be made more effective by sorting the items in decreasing order of size, so the pruning will occur closer to the root and further reduce the size of the search space.

We test the DPE on different type of instances of increasing item set sizes, and compare its performance with several other decompositions of the constraint:

- A CP model that uses the simple scalar product of the model (1a)-(1b) (**Naive CP**). The MiniZinc encoding of the knapsack contraint uses the same structure.
- The knapsack global constraint available in Choco (**Global constraint**). This constraint is implemented with scalar products. The propagator uses Dantzig-Wolfe relaxation [6].
- A CP formulation of the encoding proposed in this paper (**DPE**) solved using Google OR.
- A DPE with state space reduction technique introduced before (**DPE + sr**).
- A DPE with state space reduction technique with the parts sorted, (**DPE + sr + sorting**).
- The MIP flow formulation proposed by [13] (**Flow model Solver**) which we tested with 3 different MIP solvers: COIN CBC, CPLEX and Gurobi.

To make the plots readable we decided to show only these solutions, but others are available in our code.

As a benchmark we decided to use Pisinger's described in [14]. We did not use the code available online because it is not possible to set a seed and make the experiments replicable. Four different type of instances are defined, in decreasing correlation between items's weight and profit order: subsetsum, strongly correlated, weakly correlated and uncorrelated. Due to space limitation we leave the reader finding the details of the instances in the original paper. In our experiments we tested all the types, and we kept the same configuration of [14] first set of experiments. We increased the size of the instances until all the DP encodings were not finding the optimal solution before the time limit. A time limit of 10 minutes was imposed on the MIP and CP solvers, including variable creation overhead.

Figure 2 shows the computational time in relation with the instance size. Due to space limitations we had to limit the number of plots. We can see that DPE clearly outperforms the naive formulation in CP or the previous encoding (flow formulation) solved with an open source solver, CBC. Normal DPE solved with an open source solver is computationally comparable to the flow formulation

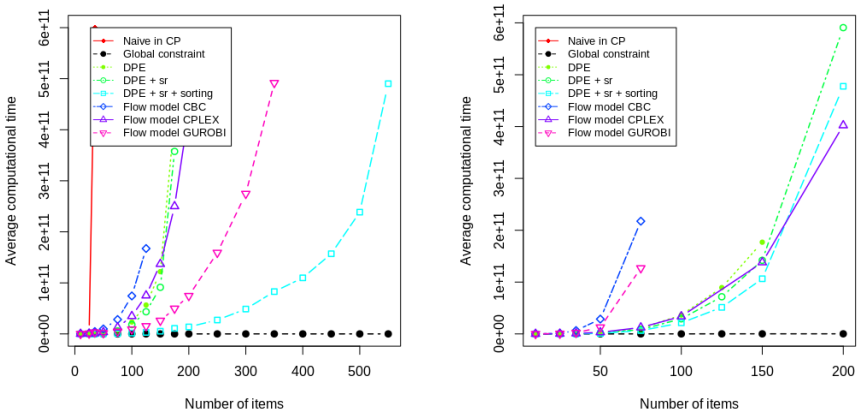

Fig. 2. Average computational time for: (a) subsetsum instances; and, (b) uncorrelated instances.

implemented in CPLEX, and outperform the one solved by Gurobi in instances where the correlation between weight and profit of the items is lower, even if the commercial MIP solvers use parallel computations. The DPE outperforms the variable redefinition technique in MIP, because of the absence of search. It is also clearly better that a simple CP model of the problem definition, which is the same model used for the MiniZinc constraint. The Choco constraint with ad-hoc propagator outperforms the DPE in most of the cases, confirming that a global constraint is faster than a DPE. A particular situation is the test on the strongly correlated instances, in this case the global constraint fails to find the optimal solution in many test instances even with a small number of items; probably the particular structure of the problem makes the search get stucked in some non optimal branches.

It is interesting to note the speed up from the space reduction technique. The basic DPE can solve instances up to 200 items but it has a memory problem: the state space grows so rapidly that a massive usage of the SWAP memory is needed. However, this effect is less marked when a state reduction technique is applied. This effect is stronger when the correlation between item profits and volumes is stronger. The reduction technique improves considerably when we increase the number of items needed to fill the bin, since the pruning occurs earlier in the search tree: see Figure 3.

In the case that the constraint has to be called multiple times during the solving of a bigger model, the DPE can outperform the pure constraint since the overhead to create all the variables is not repeated. This experiment demonstrates the potential of the DPE with state space reduction: even with a simple and intuitive reduction technique we can solve instances 10 times bigger than with a simple CP model. We can see that the behaviour of DPE is stable regardless of the type of the instance; on the contrary, the performance of the space

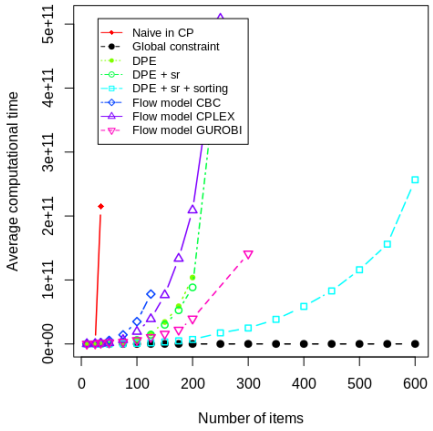

Fig. 3. Computational time in the subsetsum instances with volume per item reduced

reduction technique strongly depends on the instance type and the volume of the knapsack.

Of course we can not outperform a pure DP implementation, even if our solution involves a similar number of operations. This is mainly due to the time and space overhead of creating CP variables. In fact the DPE requires more time to create the CP variables than to propagate the constraints.

4 Conclusions

In this paper we have analysed a recently proposed technique for mapping DP into CP, called the dynamic program encoding (DPE), which takes advantage of the optimality substructure of the DP. With a DPE the DP execution is achieved by pure constraint propagation (without search or backtracking).

We provided a standard way to model a DP into DPE. We have demonstrated the potential of the DPE in constraint modelling in several ways: we compared it with another DP-encoding technique using CP and MIP solvers; we showed how to use state reduction techniques to improved its performance; we showed that it outperforms standard DP encoding techniques in the literature, and greatly outperforms non-DP-based CP approaches to the knapsack problem; and we applied the DPE to MiniZinc benchmarks, showing how its performance is faster and more robust than existing CP techniques. We also showed a negative result: the DPE is unsuitable for use in MIP, where standard methods are much better.

To recap the potential applications of the DPE, it can be used when: a DP-based constraint is needed but also other constraints can affects states inside the DP; when the respective DP global constraint is not implemented in the specific solver; and when DP approaches are needed in MiniZinc as starting approach to decompone more complex problems in simpler instructions.

Acknowledgments This publication has emanated from research supported in part by a research grant from Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289 which is co-funded under the European Regional Development Fund.

References

1. Beldiceanu, N., Carlsson, M., Rampon, J.X.: Global constraint catalog, (revision a) (2012)
2. Bellman, R.: The theory of dynamic programming. Tech. rep., RAND Corp Santa Monica CA (1954)
3. Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.N.: Discrete optimization with decision diagrams. INFORMS Journal on Computing **28**(1), 47–66 (2016)
4. Bradley, S.P., Hax, A.C., Magnanti, T.L.: Applied mathematical programming. Addison Wesley (1977)
5. Chu, G., Stuckey, P.J.: Minimizing the maximum number of open stacks by customer search. In: International Conference on Principles and Practice of Constraint Programming. pp. 242–257. Springer (2009)
6. Dantzig, G.B., Wolfe, P.: Decomposition principle for linear programs. Operations research **8**(1), 101–111 (1960)
7. Eppen, G.D., Martin, R.K.: Solving multi-item capacitated lot-sizing problems using variable redefinition. Operations Research **35**(6), 832–848 (1987)
8. Focacci, F., Milano, M.: Connections and integrations of dynamic programming and constraint programming. In: CPAIOR 2001 (2001)
9. Freuder, E.C.: Progress towards the holy grail. Constraints **23**(2), 158–171 (2018)
10. Malitsky, Y., Sellmann, M., van Hoeve, W.J.: Length-lex bounds consistency for knapsack constraints. In: International Conference on Principles and Practice of Constraint Programming. pp. 266–281. Springer (2008)
11. Martello, S.: Knapsack problems: algorithms and computer implementations. Wiley-Interscience series in discrete mathematics and optimiza tion (1990)
12. Martello, S., Pisinger, D., Toth, P.: New trends in exact algorithms for the 0–1 knapsack problem. European Journal of Operational Research **123**(2), 325–332 (2000)
13. Martin, R.K.: Generating alternative mixed-integer programming models using variable redefinition. Operations Research **35**(6), 820–831 (1987)
14. Pisinger, D.: A minimal algorithm for the 0-1 knapsack problem. Operations Research **45**(5), 758–767 (1997)
15. Plateau, G., Nagih, A.: 0–1 knapsack problems. Paradigms of Combinatorial Optimization: Problems and New Approaches **2**, 215–242 (2013)
16. Prestwich, S.D., Rossi, R., Tarim, S.A., Visentin, A.: Towards a closer integration of dynamic programming and constraint programming. In: 4th Global Conference on Artificial Intelligence (2018)
17. Quimper, C.G., Walsh, T.: Global grammar constraints. In: International conference on principles and practice of constraint programming. pp. 751–755. Springer (2006)
18. Stuckey, P.J., Feydy, T., Schutt, A., Tack, G., Fischer, J.: The minizinc challenge 2008–2013. AI Magazine **35**(2), 55–60 (2014)
19. Zhou, N.F., Kjellerstrand, H., Fruhman, J.: Constraint Solving and Planning with Picat. Springer (2015)