Towards a Closer Integration of Dynamic Programming and Constraint Programming

S. D. Prestwich[1], R. Rossi[2], S. A. Tarim[3], and A. Visentin[1]

[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] Department of Management, Cankaya University, Ankara, Turkey
`at@cankaya.edu.tr`

Abstract

Three connections between Dynamic Programming (DP) and Constraint Programming (CP) have previously been explored in the literature: DP-based global constraints, DP-like memoisation during tree search to avoid recomputing results, and subsumption of both by bucket elimination. In this paper we propose a new connection: any discrete DP algorithm can be directly modelled and solved as a constraint satisfaction problem (CSP) without backtracking. This has applications including the design of monolithic CP models for bilevel optimisation. We show that constraint filtering can occur between leader and follower variables in such models, and demonstrate the method on network interdiction.

1 Introduction

Two major areas of research dealing with optimisation and decision making are Constraint Programming (CP) and Dynamic Programming (DP). CP emerged in the 1990s from the field of Artificial Intelligence. It has also been extended to Stochastic Constraint Programming (SCP) to handle problems involving uncertainty. Dynamic Programming (DP) emerged in the 1950s from the field of Operations Research, and is still indispensible for a wide range of modern, large-scale, deterministic and stochastic problems.

CP and DP are distinct fields with different techniques, applications and research communities. Like Mathematical Programming, SAT and some other problem classes, CP has languages, development tools and solvers. DP has few or none of these and is rather an approach and a collection of techniques. But unifying distinct fields of research often leads to new insights and techniques, and it is worth looking for ways of combining CP and DP. CP/OR integrations are surveyed in [9, 10].

We know of three existing connections between the two fields. Firstly, the DP feature of solving each subproblem once only has been emulated in CP [4] and in Constraint Logic Programming languages including Picat [21], by remembering the results of subtree searches via the technique of *memoization* (or *tabulation*) which remembers results so that they need not be recomputed. This can improve search performance by several orders of magnitude. Secondly, DP has been used to implement several efficient global constraints within CP systems [17]. Thirdly, bucket elimination [6] is a very general framework that can model DP and CP problems and others.

In this paper we explore a new connection between DP and CP: by explicitly representing DP states as CP variables, we can implement a DP as a Constraint Satisfaction Problem (CSP), which we call a *Dynamic Program Encoding* (DPE). A DPE is solved by constraint propagation without backtracking and can form part of a larger constraint model, allowing

DP to be seamlessly integrated within CP. This includes Stochastic DP (SDP), allowing some multistage stochastic optimisation problems to be modelled as CSPs. Furthermore, replacing some constants by variables in a DPE *parameterises* it, leading to flexible global constraints that can be called in multiple modes, and monolithic CP models for discrete bilevel optimisation problems.

The paper is organised as follows. Section 2 describes the DPE and its applications. Section 3 describes parameterised DPEs and their applications. Section 4 considers the bilevel optimisation application in detail, by modelling and solving a set of network interdiction problems. Section 5 concludes the paper and discusses future directions.

2 The dynamic program encoding

In this section we define the DPE and demonstrate some of its applications. As mentioned in Section 1 a DPE models every possible DP state by a CP variable, and the seed values and recurrence relations are modelled by constraints. We illustrate this by examples.

2.1 DPE for Fibonacci numbers

Suppose we want to use CP to generate the n^{th} Fibonacci number. Though very simple, this example has been used many times. It illustrated the use of ILOG Script as an imperative programming language in [1]. It was also used as an example of memoisation in (Constraint) Logic Programming, for example in Picat [12]. Finally, it was used as an example of Constraint Handling Rules [7], where a naive formulation takes exponential time because of repeated recursive calls, but a better formulation takes linear time.

To find the n^{th} Fibonacci number we need n states $f_1, l \ldots, f_n$. We need seed values:

$$f_1 = 1 \quad f_2 = 1$$

and recurrence relations:

$$f_i = f_{i-1} + f_{i-2} \quad (i = 3, \ldots, n)$$

If we create n real variables f_i in a CP system, and post the seed values and recurrence relations as *seed constraints* and *recurrence constraints*, all f_i values are immediately computed via constraint propagation. The 1st and 2nd constraints assign 1 to f_1 and f_2. As soon as the 3rd constraint is posted it assigns 2 to f_3. In turn when the 4th constraint it assigns 3 to f_4. And so on until we obtain $f_{10} = 55$.

Of course we need not post the constraints in any particular order. If we post them in reverse order they are suspended by the CP system until the seed constraints are posted, then woken in the correct order. In this way CP systems provide some support for DP implementation: we do not need to implement a DP algorithm, merely describe its states. However, a DPE cannot handle billions or more states because CP systems they are not typically designed for such large applications.

It should be emphasised that this CSP is not recursive in any sense: it simply relates a set of variables by constraints. Each value is computed only once (this is efficient: f_{100} is solved in a small fraction of a second). This is equivalent to solving a DP by forward recursion.

We call such a CSP a *Dynamic Program Encoding* (DPE). DPEs are unlike the use of *memoisation* in CP, where recursive solutions are stored and reused in a DP-like way. Memos are invisible whereas DP state variables can interact with other problem variables via constraints. Our modeling of DP states is similar to that used by Régin to implement a subset-sum global constraint [17], which modelled the DP used earlier in [20] to implement the same constraint in a CP system. However, the technique was presented as an approach to a (non-DP) technique for encoding automata, rather than as a general CP/DP integration technique. DPEs are also related in spirit, though not in detail, to the *variable redefinition* approach used to integrate Mixed Integer Programming (MIP) and DP [14, 16].

2.2 DPE for the change problem

Fibonacci numbers are a trivial example that does not really require a shift to DP-style thinking, as there is really only one way of modelling them. As a less obvious example consider the *change problem*: find the minimum number of coins adding up to a given total S, given available denominations $d_1 \dots d_D$. This is slightly different to the *coins problem* of [3], which is the same as the *change making problem* of [8]. In that problem we must find a minimal set of coins that can sum to *all* values $\leq S$. In the change problem the solution for S might not contain all coins in the solution for $S-1$, so it is not necessarily a solution to the coins problem.

A constraint programmer might model the change problem in the following way:

$$\min \sum_i x_i \quad \text{s.t.} \quad \sum_i x_i d_i = S$$

where the x_i are finite domain variables denoting the number of coins from denomination i. This might require an exponential number of backtracks, but it is well-known that the problem can be solved in pseudo-polynomial time by DP. Consider the following simple DP method that takes $O(S^2)$ time (though there are more efficient methods taking $O(S \log S)$). The seed value is

$$n_0 = 0$$

and the recurrence relation is:

$$n_i = \min \begin{cases} n_{i-d_1} + 1 \\ \vdots \\ n_{i-d_D} + 1 \end{cases} \qquad (i = 1, \dots, S)$$

This is not quite correct because the recurrence relation refers to $n_{-1} \dots n_{1-M}$ so we add further seed values:

$$n_i = S \quad (i = -1, \dots, 1 - M)$$

where $M = \max_i(d_i)$. S is used here as a large value that will never be chosen by the DP min function.

We now use the DPE to model this DP as a CSP, leading to a backtrack-free CP model. First create a CP variable for each DP state (these could be real or finite domain variables, and here we use reals):

$$n_i \in \mathbb{R} \quad (i = 0, \dots, S)$$

Then the seed values and recurrence relation can be written using arithmetic constraints:

$$\begin{aligned} &n_0 = 0 \\ &n_i = \min\{n_{i-d_k} + 1 \mid k = 1, \dots, D\} \end{aligned}$$

The result of the DP computation is the value of n_S. In a CP model the expression n_{i-d_1} might need to be replaced by n_j where i, j, k are auxiliary variables related by additional constraints, and we did this in our model. The model does not directly tell us how many coins of each denomination should be used, but this can easily be reconstructed from the DP state variables.

2.3 Application to multistage stochastic optimisation

The DPE is not restricted to deterministic DP, and SDP can be modelled in the same way. As an example we use a 3-period stochastic inventory control problem. At the start of each period we must decide how many units of a product should be produced. If production takes place for x units ($x > 0$) we incur a production cost $c(x)$ which has a fixed and a variable component:

$$c(x) = \begin{cases} 0 & \text{if } x = 0 \\ 3 + 2x & \text{if } x > 0 \end{cases}$$

Production in each period cannot exceed 4 units. Demand in each period takes two possible values: 1 or 2 with equal probability (0.5). Demand is observed in each period only after production has occurred. After meeting the current period's demand, a holding cost of 1 per unit is incurred for any item carried over to the next period. Because of limited capacity, the inventory at the end of each period cannot exceed 3 units. All demand should be met on time (no backorders). Any units still in stock at the end of period 3 (the planning horizon) can be salvaged at 2 per unit. The initial inventory is 1 unit.

Define $f_t(i)$ as the minimum expected total cost incurred in periods $t, t+1, \ldots, 3$ when i units of inventory are available at the start of period t. The aim is to compute $f_1(1)$. The DPE model has variables

$$f_t(i) \in \mathbb{R} \quad (t = 1, 2,\ i = 0, \ldots, 3)$$

with constraints for $t = 1, 2$:

$$\begin{array}{ll} f_t(i) = \min_{x \in \{0,\ldots,4\} \,|\, 2-i \le x \le 4-i} & \left\{ c(x) + \frac{1}{2}(i + x - 1) + \frac{1}{2}(i + x - 2) + \right. \\ & \left. \frac{1}{2} f_t(1)(i + x - 1) + \frac{1}{2} f_t(1)(i + x - 2) \right\} \end{array}$$

and for $t = 3$:

$$\begin{array}{ll} f_3(i) = \min_{x \in \{0,\ldots,4\} \,|\, 2-i \le x \le 4-i} & \left\{ c(x) + \frac{1}{2}(i + x - 1) + \frac{1}{2}(i + x - 2) + \right. \\ & \left. \frac{1}{2} 2(i + x - 1) + \frac{1}{2} 2(i + x - 2) \right\} \end{array}$$

Again constraint propagation from the seed constraints immediately yields the result: $f_1(1) = 16.25$. It is easy to generalise the model to arbitrary costs and number of time periods in a constraint specification language. Stochastic DPEs can also be used as the basis for new global constraints.

3 Parameterised DPEs

In a DPE some constants can be replaced by CP variables, creating a *parameterised DPE*. In this section we discuss applications of this idea.

3.1 Parameterised Fibonacci numbers

Suppose we parameterise the DPE of Section 2.1 for the 10th Fibonacci number, replacing the seed values by variables, then set f_{10} to some values and hope to infer (or at least constrain) the seed values. (In Section 4 we parameterise constants in recurrence constraints.)

If we use real variables then all the constraints are suspended until further variables assignment occur. However, if we use finite domain variables (say with domains 1–100) we immediately obtain a solution without backtracking. Setting $f_{10} = 55$ (which is the correct value of the 10th number) we obtain $f_1 = f_2 = 1$, and if we instead set $f_{10} = 56$ then CP detects failure without backtracking. This example shows that using finite domain variables in a parameterised DPE can be much more powerful than using real variables. However, parameterised DPEs are not in general guaranteed to be solved without backtracking.

3.2 Parameterised change problem

The change problem DPE of Section 2.2 can be also parameterised. For example we could replace the denomination constants by finite domain variables and assign a value to S, then solve the resulting DPE to find a set of values leading to a given minimum total S. This is similar to the *currency design problem* of [3] but, as with the change problem, it is not exactly the same.

Parameterisation is not always trivial, and for this example we found it necessary to add new variables and constraints. We model the DP states by finite domain variables:

$$n_i \in \{0, \ldots, S\} \quad (i = 1 - M, \ldots, S)$$

and add denomination variables:

$$d_j \in \{1, \ldots, M\} \quad (j = 1, \ldots, D)$$

and auxiliary variables $a_{i,j}$ to represent n_{i-d_j}:

$$a_{i,j} \in \{0, \ldots, S\} \quad (i = 1, \ldots, S,\ j = 1, \ldots, D)$$

The auxiliary variables are connected to the other variables via channeling constraints:

$$d_j = x \rightarrow a_{i,j} = n_{i-x} \quad (i = 1 - M, \ldots, S,\ j = 1, \ldots, D,\ x = 1, \ldots, M)$$

A more compact way of expressing the channeling constraints would be via `element` constraints, but this constraint must be posted with a list of integers whereas we require a list of finite domain variables. This is one reason that the parameterisation is nontrivial. The seed values are:

$$\begin{array}{l} n_0 = 0 \\ n_i = S \quad (i = 1 - M, \ldots, -1) \end{array}$$

The recurrence relation is expressed via the auxiliary variables:

$$n_i = \min\{a_{i,j} + 1 \mid j = 1, \ldots, D\} \quad (i = 1 - M, \ldots, S\}$$

One of the d_j is set to 1 so that there is a solution for n_1 (we choose $j = 1$):

$$d_1 = 1$$

The final DP state variable must equal the total value:

$$n_S = C$$

To break permutation symmetry and ensure that all denominations are distinct we strictly order the d_j :

$$d_j < d_{j+1} \quad (1 \leq d_i < d_{i+1} \leq D)$$

As an example we set $S = 55$, $C = 3$, $D = 5$ and $M = 19$. We assign the d_j variables, then the n_i, then the $a_{i,j}$ in ascending order of index, and choose smaller domain values first. This and other models are implemented and executed in the Eclipse constraint logic programming language [3]. In less than 1 second we find a solution for $M = 19$ with denominations 1, 16, 17, 18 and 19. This is correct: if denominations are limited to 1–19 we need at least 3 coins to total 55, and it is easily verified that 3 coins can total 55 with these denominations: $18+18+19 = 55$. It is also optimal: if we set $M = 18$ even 3 coins with maximum denomination 18 fail to total 55.

However, to show unsolvability by backtracking with $M = 18$ takes at least several hours whereas a simple CP model would immediately detect this by reasoning on bounds. This indicates that the level of consistency achieved by the DPE is low. Channeling the DPE to a standard CP model would immediately detect unsolvability in this case, but the model becomes slightly complicated and we shall not pursue this problem further. A simple but less general way to detect unsolvability on this example is to add the implied constraint $Cd_D \geq S$.

3.3 Application to bilevel optimisation

In bilevel optimisation problems [5] there are two or more decision makers. In a *two-player Stackelberg game* there is a *leader* (or *outer*, or *upper-level*) problem, and a *follower* (or *inner*, or *lower-level*) problem. The leader must optimise an objective, under the constraint that the follower optimises a different objective. The two decision makers may have the same objective function, or opposite objectives, or anything between. The leader is assumed to know the follower's objective and must optimise its objective in that knowledge. Though their relationship is asymmetric, the leader and follower influence each other: the follower reacts to the leader's decisions, while the leader makes decisions taking the follower's objective into account.

Stackelberg games originate from economic game theory and mathematical programming. Applications include toll setting, structural optimization, defense applications, road network planning, optimal chemical equilibria, environmental economics, and water resource management. A standard mathematical formulation is:

$$\begin{array}{rl} \min_{\vec{x},\vec{y}} & F(\vec{x},\vec{y}) \\ \text{s.t.} & \vec{y} \in \operatorname{argmin}_{\vec{y}}\{f(\vec{x},\vec{y}) \,|\, g_j(\vec{x},\vec{y}) \leq 0,\ j = 1\ldots J\} \\ & G_k(\vec{x},\vec{y}) \leq 0,\ k = 1\ldots K \\ & \vec{x} \in X,\ \vec{y} \in Y \end{array}$$

where $\vec{x}$ represents the leader decision variables and $\vec{y}$ the follower decision variables. The condition $\vec{y} \in \operatorname{argmin}_{\vec{y}}$ forces $\vec{y}$ to be an optimal solution to the follower problem, and can be considered as a computationally expensive constraint.

In a *generalised Stackelberg competition model* [18] there are multiple leaders and followers. Leaders make their decisions independently of each other, then followers make their decisions independently of each other. Several classes of such problems have known solution strategies [13], but in the general case bilevel optimisation problems are considered hard to solve, and discrete problems are a relatively unexplored class that are in NP or PSPACE. Evolutionary algorithms are often used [19] but can be computationally expensive. Benders decomposition is a natural approach, but it is not always possible to generate strong constraints because the programmer is not free to choose the decomposition: the leader must be the Benders master problem, and the follower must be the slave.

To the best of our knowledge CP has been applied to bilevel optimisation only once: a scheduling problem in [2] that used a Benders decomposition-like approach. A point discussed in that paper is optimistic and pessimistic cases when follower problems have multiple optimal solutions. In the optimistic/pessimistic case the leader assumes that the follower will choose the best/worst solution from the leader's point of view. Our approach is like most others in that it handles the easier optimistic case, but the difference is irrelevant to the bilevel problem we tackle.

Parametrised DPEs have a natural application to bilevel optimisation, because a follower problem can be thought of as a parameterised optimisation problem whose parameters are chosen by the leader. Thus we have a method for modelling and solving discrete bilevel optimisation problems in which the leader is a constraint program and the follower is a parameterised dynamic program: we simply add a parameterised DPE for the follower to the leader's CP. Section 4 provides a detailed example.

4 Network interdiction

In this section we explore in detail an application of the DPE mentioned in Section 3.3: bilevel optimisation. We use a particular and well-known example: *network interdiction*. In this class of problems the follower must move materiel through a (transportation, communication, biological) network as quickly as possible, while the leader attempts to disrupt the follower's efforts by *interdicting* chosen arcs. Interdiction might completely destroy an arc, increase its (fixed or expected) length, or reduce its capacity. Each arc has an associated interdiction cost and the leader has a total interdiction budget.

4.1 The MXSP variant

In the *Maximising the Shortest Path* (MXSP) problem [11] the follower takes a shortest path through a network which is a weighted digraph, but before that occurs the leader (the *interdictor*) increases the length of certain edges by a given amount. Each edge interdiction has a cost and the leader has a limited budget. The leader's objective is to maximise the follower's minimum path length so the two decision makers are in conflict. MXSP is NP-complete. Previous approaches to the MXSP are discussed in [11].

Let V be the vertices and E the edges of the graph (V, E). Let the length of edge (i, j) be $c_{i,j}$ ($0 \leq c_{i,j} < \infty$) which increases to $c_{i,j} + d_{i,j}$ ($d_{i,j} > 0$) on interdiction, let the cost of interdicting edge (i, j) be $r_{i,j}$, and let the total interdiction budget be r_0. Define binary decision variables $x_{i,j}$ such that $x_{i,j} = 1$ iff edge $(i, j) \in E$ is interdicted by the leader. All constants are assumed to be integers, "r" stands for *resource*, and r_k is typically a small integer indicating how many weapons are needed to interdict the edge, so r_0 is the total number of available weapons. We assume that V contains an origin vertex s and a sink vertex t. Below we shall denote resource r_k for edge $k = (i, j)$ by $r_{i,j}$.

4.2 Shortest path DPEs

If the network has no negative cycles we can apply DP to find shortest paths in linear time, as follows. Define a DP state variable z_i for each vertex $i \in V$ denoting the length of the longest path from i to t. For the follower shortest path problem the seed value is

$$z_t = 0$$

and the recurrence relation is

$$z_j = \min_{i \in P_j} [z_i + c_{i,j}] \quad (\forall j \in V \setminus \{t\})$$

where $c_{i,j}$ is the length of arc (i,j), and $P_j = \{(i,j) \in E\}$ is the set of all (*predecessor*) edges of j in the DAG. The length of the shortest path is then z_s. This is an application of Dijkstra's algorithm.

4.3 Basic CP model for interdiction

We now parameterise the above DPE by making each arc length dependent on a binary interdiction variable $x_{i,j}$:

$$z_j = \min_{i \in P_j} [z_i + c_{i,j} + d_{i,j}x_{i,j}] \quad (\forall k, i)$$

with all $x_{s,j} = x_{i,t} = 0$ because these arcs are non-interdictable. Then our CP model for network interdiction is simply:

$$\begin{array}{l} \max z_s \ \text{ s.t.} \\ \quad z_t = 0 \\ \quad z_j = \min_{i \in P_j} [z_i + c_{i,j} + d_{i,j}x_{i,j}] \quad (\forall k, i) \\ \quad \sum_{k=(i,j) \in E} x_{i,j} r_{i,j} \leq r_0 \end{array}$$

where the decision variables are x, z. We did not find it necessary to directly model the shortest path computed by the follower, but this can easily be done. Alternatively, it can be inferred from the z.

4.4 Improved CP model

Now that we have a monolithic (single-level) constraint model, we can apply standard CP techniques to improve performance. We find it useful to define an additional set of variables z' to complement the z, measuring the distance from the source to each node:

$$\begin{array}{l} z'_s = 0 \\ z'_j = \min_{i \in P_j} [z'_i + c_{j,i} + d_{j,i}x_{j,i}] \quad (\forall k, i) \end{array}$$

This allows us to base constraints (see below) on the shortest path length that includes arc (i,j) via the expression $z'_i + c_{i,j} + d_{i,j}x_{i,j} + z_j$.[1] We add a constraint

$$z_s = z'_t$$

as both measure the same shortest path length under any interdiction plan. We also add

$$z_i + z'_i \geq z_s \quad (\forall i \neq s, t)$$

We exploit a dominance by posting constraints to prevent the interdiction of any arc that can not lead to a shortest path under the interdiction plan:

$$(z_j + z'_i + c_{i,j} > z_s) \rightarrow (x_{i,j} = 0) \quad (\forall i, j)$$

We also post dominance constraints to explot an interdiction version of the Floyd-Warshall triple operation:

$$(c_{k,j} \geq c_{k,i} + d_{k,i}x_{k,i} + c_{i,j} + d_{i,j}x_{i,j}) \rightarrow (x_{k,j} = 0)$$

[1] We could instead obtain these lengths by encoding the Floyd-Warshall algorithm, but that requires $O(|V|^3)$ states whereas our z, z' approach only requires $O(|V|)$.

Using finite domain z, z' variables improves performance because of bidirectional links in the network. A small example illustrates how this helps. If we use real variables and post constraints such as

$$x = \min(y + 5, 5) \qquad y = \min(x + 5, 5)$$

(similar to constraints for bidirectional arcs in our interdiction model) then the constraints are simply suspended until x or y are assigned a value. But if x, y are finite domain variables then the solver immediately infers $x = y = 5$ from these constraints. Along with the parameterised Fibonacci model of Section 3 this is another example of the advantage of using finite domain variables in a parameterised DEP.

4.5 Search strategy

Another aspect of CP that can have a large impact on performance is the choice of search heuristics, especially the variable and value orderings. For the value ordering we choose 1 then 0. We use a dynamic variable ordering, choosing $x_{i,j}$ in ascending order of shortest path length involving arc (i, j) using the mean of the lower and upper bound on $z'_i + c_{i,j} + z_j$, breaking ties randomly. These upper and lower bounds are extracted from the variable domains and change during search.

4.6 Filtering between leader and follower

We use the basic model of Section 4.3 to illustrate an important aspect of our bilevel CP models: that constraint filtering can occur between the leader and follower variables (which are x and z respectively in this model).

For example suppose that during search the leader assigns $x_{i',j} = 1$ for some edge (i', j), and that i' has the least lower bound for $z_i + c_{i,j} + d_{i,j}x_{i,j}$ among all $i \in P_j$. Then the lower bound for z_j is increased by filtering. If j has the same property with respect to some other edge (k, j) then the lower bound for z_k is also increased by filtering. By cascading through the network this can increase the lower bound on z_s. If this lower bound exceeds the upper bound imposed by branch-and-bound then backtracking will occur. Moreover, a sufficiently powerful filtering algorithm will detect this and remove value 1 from the domain of $x_{i',j}$.

This argument shows that a leader variable domain can be filtered by reasoning on the follower's optimality condition, which is encoded in the DPE. This was mentioned as a possible direction for future research by [2], and it occurs naturally when we use a DPE with sufficiently powerful filtering.

4.7 Experiments

An example of a network from [11] is shown in Figure 1, in which unbroken lines are interdictable and dotted lines are length 0 and non-interdictable. They created instances using various parameters. There are $m \times n$ "transshipment nodes" arranged in m rows and n columns, an arc from the leftmost node s to each node in the first column, and an arc from each node in the last column to the rightmost node t; the latter two sets are 0-length and non-interdictable. There are also arcs between columns as shown. Each arc has a length which is a randomly generated integer in the range 1–c, and this is increased by an integer d^+ if the arc is interdicted. Each arc also has an interdiction resource which is a randomly generated integer in the range 1–r, representing the cost of interdiction. The total interdiction budget is r_0.

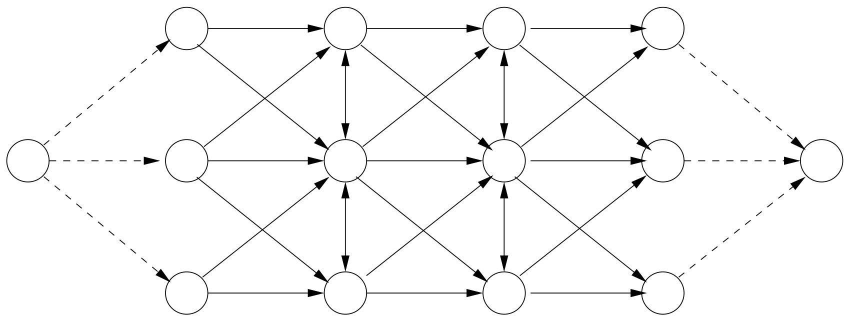

Figure 1: Example of a 3×4 network from [11].

problem	k	$m \times n$	a	CP $d^+ = 5$	CP $d^+ = 10$
13	5	7×7	188	1.9	5.5
14	10	7×7	188	87	298
15	5	8×8	238	3.5	12
16	10	8×8	238	284	8/10
17	5	9×9	312	10	40
18	10	9×9	312	1128	2/10
19	5	12×12	594	323	267
20	10	12×12	594	0/10	0/10

Table 1: k-most-vital-arcs results

We first consider their problems 13–20 which are also *k-most-vital-arcs problems* because all arcs have the same interdiction cost of 1. Our results are shown in Table 1 with all figures means over 10 runs (in the case indicated by "—" more than half of the instances took over an hour to solve). Our results cannot be directly compared with those of [11] for several reasons: we used 10 different randomly generated instances; they used the CPLEX version 6.5 MIP solver whereas we used the Eclipse constraint solver; our machines are different. They did not specify their machine but after a gap of 15 years ours, a 2.8 GHz Pentium 4 with 512 MB RAM, is likely to be several times faster. On the other hand, Eclipse is not the fastest CP solver while CPLEX is a state-of-the-art MIP solver. However, we can compare results in an approximate way.

[11] applied 3 algorithms to the above problems, which they called MXSP-D, Algorithm 1 and Algorithm 2. MXDP-D applies branch-and-bound to a MIP model created by combining the dual of the follower problem with the leader problem; these models included variables representing shortest paths, and flow-balance constraints. Algorithm 1 is a basic Benders decomposition algorithm enhanced with *supervalid inequalities*. Algorithm 2 is based on an alternative decomposition that reduces MXSP to a feasibility-seeking set-covering problem. Algorithm 2 was motivated partly by the possibility for Algorithm 1 to terminate with an incorrect result, depending on the size of d^+, but it is also much more efficient. (They also describe enhanced Algorithms 1E and 2E which include an optimality tolerance.)

MXSP-D and Algorithm 1 solved some but not all instances within 1 hour each, and took mean times of approximately 1–1000 seconds on those they solved. Algorithm 2 solved all instances, taking between 0.3 seconds on problem 13 to 447 seconds on problem 20 (its runtime was independent of d^+). Interestingly, both their algorithms and ours found problems 14 harder than 15, 16 harder than 17, and 18 harder than 19.

We also tried the more general interdiction instances of [11]. Their problems 1–4 use $m = n = 10$, $r_0 = 10, 20, 30, 40$, random arc lengths in the range 1–$c = 10$, random interdiction increases in the range 1–$d = 10$, and random resources in the range 1–$r = 5$. MXSP-D solved problem 1 in 60 seconds and problem 2 in 651 seconds, but failed to solve problems 3 and 4 within an hour. Algorithm 1 only solved problem 1 in 219 seconds, but an enhancement solved problems 1–3 in 3, 26 and 135 seconds respectively. An enhanced Algorithm 2 solved all four in 4, 29, 191 and 1349 seconds respectively. Ours solved problem 1 in 120 sec, problem 2 in 8/10 of cases, problem 3 in 1/10 of cases, and problem 4 in 0/10 of cases.

Our approach seems roughly comparable to the more basic MIP-based methods, but considerably less efficient than the best ones. This might be due to the superiority of MIP over CP on many classes of problem. On other problem classes CP is superior and we expect to find bilevel problems on which our method is competitive, especially those with global constraints in the leader problem. However, perhaps other CP researchers can improve our interdiction method by adapting modeling techniques from [11], by finding better search heuristics, or by using global constraints.

5 Conclusion

We described a simple method called the DPE (dynamic program encoding) for modelling DP in CP, which enables us to model and solve multi-stage stochastic optimisation problems and bilevel optimisation problems in CP. Our treatment of network interdiction is, to the best of our knowledge, only the second work on bilevel optimisation in CP. The pioneer work in this area was that of [2] who modelled and solved a bilevel scheduling problem using a Benders decomposition approach. They cited complexity results to predict that "no direct encoding of discrete bilevel problems into CP or MIP can be expected", but we have shown that it is possible in the case of DP followers. They also proposed a direction for future work: filtering the leader's domain via inference on the follower's optimality condition. Our technique achieves this naturally as shown in Section 4.6.

The DPE has application to global constraints. These are one of CP's most powerful features and several are based on DP: see for example [17] for a discussion. Suppose a CP programmer requires such a global constraint with variables replacing some of its parameters. Alternatively, a DP-based global constraint for a new problem might be needed. Global constraints are hard-wired into the CP system, and implementing new and more flexible global constraints is a nontrivial task usually performed by CP system developers. But a DPE enables the programmer to quickly implement such constraints.

Modelling DP as CP also has a software development advantage: it makes available for DP development a variety of CP tools for specification, tracing, debugging and visualisation. In fact all our models were implemented via the Kolmogorov constraint specification language [15]. This kind of high-level programming support has not previously been provided for DP.

There are other possible uses which we hope to explore in future work. We shall model and solve other bilevel problems, especially those for which CP has an advantage over MIP. Explicitly representing DP states by CP variables also allows us to answer new types of query. For example we could post constraints on different DP states in a parameterised DPE. Or we could model two separate optimisation problems as parameterised DPEs, and relate them by posting constraints on their DP states. This is not supported by the memoisation approach to

DP in CP.

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. S. Armagan Tarim is supported by the Scientific and Technological Research Council of Turkey (TUBITAK).

References

[1] Efficient modeling with the ibm ilog cplex optimization studio. Technical report, IBM Corporation, 2010. White paper.

[2] T. Kis A. Kovács. Constraint programming approach to a bilevel scheduling problem. *Constraints*, 16(3):317–340, 2011.

[3] K. R. Apt and M. Wallace. *Constraint Logic Programming Using Eclipse*. Cambridge University Press, 2007.

[4] G. Chu and P. Stuckey. Minimizing the maximum number of open stacks by customer search. In *Proceedings of the 15th International Conference on the Principles and Practice of Constraint Programming*, volume 5732 of *Lecture Notes in Computer Science*, pages 242–257, 2009.

[5] B. Colson, P. Marcotte, and G. Savard. Bilevel programming: a survey. *4OR*, 3(2).

[6] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113:41–85, 1999.

[7] T. Frühwirth. *Programming in Constraint Handling Rules*. 2005.

[8] S. Heipcke. Comparing constraint programming and mathematical programming approaches to discrete optimisation — the change problem. *The Journal of the Operational Research Society*, 50(6):581–595, 1999.

[9] B. Hnich, R. Rossi, S. A. Tarim, and S. Prestwich. A survey on cp-ai-or hybrids for decision making under uncertainty. In M. Milano and P. Van Hentenryck, editors, *Hybrid Optimization: the 10 Years of CP-AI-OR*, volume 45 of *Optimization and Its Applications*, pages 227–270. Springer, 2011.

[10] J. N. Hooker. Logic, optimization and constraint programming. *INFORMS Journal on Computing*, 14(4):295–321, 2002.

[11] E. Israeli and R. K. Wood. Shortest-path network interdiction. *Networks*, 40(2):97–111, 2002.

[12] H. Kjellerstrand. Picat: A logic-based multi-paradigm language. In *ALP ISSUE*, Feature Articles. 2014.

[13] J. Lu, C. Shi, and G. Zhang. On bilevel multi-follower decision making: General framework and solutions. *Information Sciences*, 176(11):1607–1627, 2006.

[14] R. Kipp Martin. Generating alternative mixed-integer programming models using variable redefinition. *Operations Research*, 35(6):820–831, 1987.

[15] S. D. Prestwich, S. A. Tarim, and R. Rossi. Constraint problem specification as compression. In *2nd Global Conference on Artificial Intelligence*, volume 41 of *EPiC Series in Computing*, pages 280–292, 2016.

[16] J. F. Raffensperger. The marriage of dynamic programming and integer programming. In *Proceedings of the ORSNZ 24th Annual Conference*, pages 49–58, 1999.

[17] J.-C. Régin. Global constraints: A survey. *Hybrid Optimization*, 45:63–134, 2010.

[18] A. Sinha, P. Malo, A. Frantsev, and K. Deb. Finding optimal strategies in a multi-period multi-leader-follower stackelberg game using an evolutionary algorithm. *Computers and Operations Research*, 41:374–385, 2014.

[19] El-G. Talbi. Metaheuristics for bi-level optimization. *Studies in Computational Intelligence*, 482, 2013.

[20] M. A. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals of Operations Research*, 118:73–84.

[21] N.-F. Zhou, H. Kjellerstrand, and J. Fruhman. *Constraint Solving and Planning with Picat*. Springer, 2015.