# Modeling the p-Dispersion Problem with Distance Constraints

**Panteleimon Iosif**
University of Western Macedonia, Kozani, Greece

**Nikolaos Ploskas**
University of Western Macedonia, Kozani, Greece

**Kostas Stergiou**
University of Western Macedonia, Kozani, Greece

**Dimosthenis C. Tsouros**
KU Leuven, Belgium

## Abstract

We study the p-dispersion problem with distance constraints (pDD), which is a variant of the well-known p-dispersion problem. In a pDD, we seek to locate a set of facilities in an area, so as to maximize the minimum distance between any two facilities, subject to the satisfaction of constraints that specify the minimum allowed distances between facilities. Two CP models for the pDD have recently been proposed. The first explicitly models the objective function and links it to the decision variables, allowing any standard CP solver to solve a pDD through Branch&Bound. However, as the size of the problem grows, this model becomes increasingly inefficient due to memory and cpu time issues. The second CP model is a simple one that does not explicitly model the objective function, and therefore, does not link it to the decision variables, meaning that propagation power is diminished and Branch&Bound is not applicable. This model essentially treats the pDD as a satisfaction problem where all solutions are seeked, simply recording the best solution found within the allowed time limit. Despite its simplicity, if this model is implemented efficiently, it is able to handle instances of larger sizes, with the downside being that often, only solutions that are far from the optimal are discovered. In this paper we first present a detailed examination of the two CP models on problems of varying size, analyzing their pros and cons. Then, we demonstrate how a rather forgotten CSP technique, standard backjumping, coupled with a simple and rather unconventional propagation method, can be used to compensate for the weak propagation in the simple model, allowing a solver to mimic Branch&Bound, and to reach much improved solutions.

**2012 ACM Subject Classification** Author: Please fill in 1 or more \ccsdesc macro

**Keywords and phrases** Author: Please fill in \keywords macro

**Digital Object Identifier** 10.4230/LIPIcs...

## 1 Introduction

Maximum diversity problems arise in many practical settings, from facility location to telecommunications and social networking analysis [7, 13, 3]. The most famous such problem is the maxmin p-dispersion problem, initially introduced by Shier as early as 1977 [18]. In this problem, which is NP-hard on general networks for any given $p$, we are given a set of candidate locations $P = \{1, 2, \dots, n\}$ for $p$ facilities and an n × n matrix $D[i, j], i, j \in P$ with distances between candidate locations $i$ and $j$. The goal is to select $p$ items from $P$ to locate the facilities such that the minimum distance between any pair of facilities is maximized.

In practice, p-dispersion usually becomes relevant whenever the close proximity of a set of facilities is dangerous or for other reasons undesirable. This is typically the case when the facilities to be located are (ob)noxious, e.g. power plants, prisons, dump sites, etc., but similar principals also apply in other location scenarios. For example, dispersing military installations makes it harder for an adversary to neutralize all of them at once, whereas franchises of the same company may be spaced apart to limit direct competition.

The first integer linear programming (ILP) formulation for this problem was put forward by Kuby [10], while Erkut [6] developed the first dedicated algorithm for solving it. More recently, ILP-based strategies developed by Sayyady & Fathi [17] and Sayah & Irnich [16] have been shown to efficiently address large-scale instances.

A variant of the p-dispersion problem, which has begun to receive renewed attention, is the p-dispersion problem with distance constraints (pDD). In this problem, in addition to the objective of p-dispersion, there exist distance constraints between the facilities. Moon and Chaudhry were the first to systematically study location problems with distance constraints and coined the term p-dispersion [12]. While they recognized the pDD as a practical challenge in real-world scenarios, they did not propose any solutions. Later, Dai et al. revisited this issue within the broader context of circle (i.e. facility) dispersion in non-convex polygons [4].

Recently, ILP and CP models were proposed for the pDD [14, 9]. These models can be written into a format suitable for any standard MIP or CP solver, allowing for the pDD to be solved to optimality through Branch&Bound. However, experimental results indicated that the size of the ILP and CP models grows rapidly with the number of facilities and potential location points, primarily due to the introduction of numerous auxiliary variables and/or constraints required to model the distance constraints. This often leads to memory exhaustion and system crash.

Hence, a heuristic CP approach, which utilizes a very simple model and a greedy heuristic to prune branches within a dedicated CP solver, was also proposed, and was shown to significantly outperform the exact approaches on hard instances [14, 9]. The simple CP model includes only $p$ decision variables, each representing a facility with a domain containing all potential location points. Distance constraints are enforced through an arc consistency propagation algorithm, while the objective function is not explicitly captured. This model essentially views the pDD as a satisfaction problem, meaning that a solver that employs it will simply search for all solutions and keep the best it can find within the allowed time. The absence of an explicit objective function means that Branch&Bound is not applicable, and therefore, the solver may discover solutions that are better, equal, or worse than the current best one, while search progresses. Despite its simplicity, this approach was shown to be necessary when dealing with instances with large numbers of $p$ and $|P|$, because of the low memory requirements.

In this paper, we focus on CP models for the exact solving of the pDD. We describe the main "optimization" CP model, with two options for the distance constraints, as well as the simple "satisfaction" one. We first experiment with pDDs of small size, demonstrating that, as expected, the optimization model reaches much better solutions than the satisfaction one within 1 hour of cpu time, with both models being implemented in state-of-the-art solvers such as OR-Tools and CP Optimizer. We also give results from our custom solver implemented from scratch. We then experimentally demonstrate that standard CP solvers with either the optimization or the satisfaction model are unable to handle large pDDs because of the overwhelming memory requirements. We identify the formulation of the distance constraints as the main problem, and show that a simple implementation of these constraints, that bypasses CP modeling constructs such as the Element constraint, does not face any memory problems, even for very large pDD instances.

As our final, and main contribution, we utilize a simple observation regarding the maxmin objective of p-dispersion to significantly improve the performance of a solver that uses the satisfaction model for the pDD. Specifically, as noted by Shier [18] and further elaborated by Kuby [10], given a location of the p facilities with cost $d$, it is not possible to improve it, unless at least one of the two facilities that are located at distance $d$ is relocated. In the context of a CP approach, where variables corresponding to facilities are assigned in a depth-first manner, once a solution with cost $d$ is discovered, it cannot be improved unless the solver backtracks to the level where one of the variables determining the cost was assigned. These variables may have been assigned way up the search tree, meaning that standard chronological backtracking will result in fruitless exploration of a (possibly exponentially sized) portion of the search space.

To amend this, we propose a simple backjumping scheme that backtracks to the deepest among the two variables determining the cost, as soon as a solution that improves the objective is discovered. This scheme, which we call *Solution Based Backjumping* (SBJ), becomes (slightly) more complex in case there is more than one pair of variables that determine the cost. But even if SBJ is used, a CP solver that utilizes the satisfaction model still suffers from a serious drawback: It does not guarantee that any newly discovered solution is better than all previously found ones. This is because this model results in weaker pruning compared to the optimization CP model. To amend this, we propose a simple, slightly unconventional, propagation technique, which we call *max-min pruning*, and prove that through its use only improving solutions can be discovered. The combination of SBJ and max-min pruning allows for a CP solver that uses the satisfaction model to efficiently mimic Branch&Bound search without explicitly modeling the objective function.

Experimental results demonstrate that SBJ and max-min pruning have a quite significant effect on instances that only contain a few variables (10-20), and can have an astounding effect on larger instances that are out of reach for standard CP solvers. Using the proposed techniques, the solver is able to obtain solutions of profoundly improved cost, mainly because on large instances, lengthy backjumps are achieved, skipping the exploration of very large portions of the search space.

## 2 Background

In this section, we define the problem and give the necessary notation.

### 2.1 Problem definition and notation

In a *p-dispersion problem with distance constraints* (pDD), $p$ facilities in a set of facilities $F$ are to be placed on $p$ nodes (points) of a weighted network $G$ [12]. Hence, we have a discrete/network location problem. We assume that the set of nodes (candidate facility sites) $P$ is known. Between each pair of facilities $f_i$ and $f_j$ there is a binary distance constraint specifying that the distance between the points where the facilities $f_i$ and $f_j$ are to be located must be greater than $d_{ij}$, where $d_{ij}$ is a constant. Notice that $d_{ij}$ may vary from constraint to constraint, as we deal with the case of heterogeneous facilities, following the work of [14, 9].

The distance between two points can be given by the Euclidean distance, e.g. for the location of hazardous facilities, or by the shortest path in a street network, e.g. for the location of franchises, or by any other suitable metric. As is common in the literature, we assume that the pairwise distances between all candidate facility location sites are given in a 2-D distance matrix $D$ (i.e. $D[i, j]$ is the distance between points $i$ and $j$). The goal in a pDD is to locate the $p$ facilities so that the minimum distance between any two facilities is maximized (a maxmin objective), subject to the satisfaction of all the distance constraints.

**Further notation.** In the sections below, we use the following notation:

- $obj_{best}$: At any point during search, this denotes the best objective value found so far.
- $A$: A complete assignment $A = \langle x_1 = v_1, \ldots, x_p = v_p \rangle$, with $A[x_i]$ denoting the projection of $A$ on $x_i$ (i.e. the value that $x_i$ takes in the assignment).
- $A_{pr}$: A partial assignment $A_{pr} = \langle x_1 = v_1, \ldots, x_n = v_n \rangle, n < p$.
- $obj_A$: Objective value of a complete assignment $A$ that satisfies all constraints (a solution).
- $X_c(A)$: A set containing pairs of variables that dictate the cost of a solution $A$. That is, for any pair $(x_i, x_j) \in X_c$, $D[x_i, x_j] = obj_A$. We call such pairs and variables "culprit".
- $x_{cr}$: The current variable under consideration during search.
- depth$(x_i)$: Given a complete assignment $A$, depth$(x_i)$ is the depth in the search tree where $x_i$ was assigned.
- $X_x^-$: The set of all variables $x_i$ such that depth$(x_i) <$ depth$(x)$, where $x_i, x \in X$.
- $X_x^+$: The set of all variables $x_i$ such that depth$(x_i) >$ depth$(x)$, where $x_i, x \in X$. During search, any unassigned variable is considered as having greater depth than $x_{cr}$.

**max-min pruning.** In a CSP, constraint propagation only removes values from domains when it is deemed that they are inconsistent, i.e. they cannot participate in any solution. This is typically achieved by applying some local consistency property, such as arc (domain) consistency or bounds consistency. However, in a Constraint Satisfaction and Optimization Problem (CSOP), consistent values can also be removed during search, if it is deemed that they cannot participate in any solution that is better than the incumbent solution (i.e. the best one discovered so far). We now formalize this, in the context of a pDD, by introducing the notion of *max-min consistency*.

▶ **Definition 1** (max-min consistency)**.** A value $v_i \in Dom(x_i)$, $x_i \in X$, is *max-min consistent* iff $\forall x_j \in X$, $i \neq j$, $\exists v_j \in Dom(x_j)$, s.t. $D[v_i, v_j] > obj_{best}$. In this case, $v_j$ is a *max-min support* of $v_i$. A variable $x_i$ is max-min consistent iff $\forall v_i \in Dom(x_i)$, $v_i$ is max-min consistent.

A value $v_i$ that has no max-min support in a domain $Dom(x_j)$ is *max-min inconsistent* and can be removed from $Dom(x_i)$. Practically, this means that given the assignment of $v_i$ to $x_i$ there is no way to assign $x_j$ with a value from its domain so as to improve the value of $obj_{best}$ (the distance between $x_i$ and $x_j$ will always be less or equal to $obj_{best}$). Hence, $v_i$ can be removed. We refer to the test of whether two values $v_i \in Dom(x_i), v_j \in Dom(x_j)$ satisfy the condition $D[v_i, v_j] > obj_{best}$ as a *max-min consistency check*.

## 3 CP Models for the pDD

We now present the CP models for the pDD. We first give two variants of a satisfaction (CSP) model and then the corresponding optimization (CSOP) ones. The two variants differ in the way they model the distance constraints. The first one uses the Element constraint to model them, while the second uses the Table constraint. Both of these types of global constraints are offered by all state-of-the-art CP solvers.

We make use of the following additional notation:

- $X = \{x_0, x_1, \ldots, x_{p-1}\}$, where $p = |X| = |F|$, is the set of decision variables representing the facilities. The domain of each variable $x_i \in X$, denoted $\text{Dom}(x_i)$, is the set of possible locations, i.e., $\forall x_i \in X : \text{Dom}(x_i) = P$.
- $Y = \{y_{ij} \mid 0 \leq i < j < p\}$ is a set of auxiliary variables where each $y_{ij}$ takes as value the distance between facilities/variables $x_i$ and $x_j$.

- $T = \{T_{ij} \mid 0 \leq i < j < p\}$ is the set of allowed tuples for the distance constraints between alls pairs of variables. Each $T_{ij}$, corresponding to a distance constraint between $x_i$ and $x_j$, contains pairs of assignments such that for any $(v_1, v_2) \in t_{ij}$, with $v_1, v_2 \in P$, we have $D[v_1, v_2] > d_{ij}$.

## 3.1 Modeling the pDD as a CSP

The two satisfaction models for the pDD problem are as follows:

### 3.1.1 Modeling with Element constraints

$$\begin{aligned} & \texttt{Alldifferent}(X) && (1)\\ y_{ij} &= \texttt{Element}(D, [x_i, x_j]) \quad \forall y_{ij} \in Y,\ 0 \leq i < j < p && (2)\\ & y_{ij} > d_{ij} \quad \forall y_{ij} \in Y,\ 0 \leq i < j < p && (3)\\ & && (4) \end{aligned}$$

This model contains $p$ decision variables, $\frac{p(p-1)}{2}$ auxiliary variables, $\frac{p(p-1)}{2}$ `Element` and $\frac{p(p-1)}{2}$ unary "greater-than" constraints. The Element constraint is used to access the distance matrix $D$ using the values of variables $x_i$ and $x_j$ as indices. The use of the `AllDifferent` constraint on all decision variables is not mandatory, since the distance constraints already propagate the fact that facilities should be placed at different locations, as they all have bounds (i.e. $d_{ij}$) greater than zero. We include them in this model and the ones that follow, although experimental results have demonstrated that they only have a slight impact on run times.

### 3.1.2 Modeling with Table constraints

$$\begin{aligned} & \texttt{Alldifferent}(X) && (5)\\ & \texttt{Table}(T_{ij}, [x_i, x_j]) \quad 0 \leq i < j < p && (6)\\ & && (7) \end{aligned}$$

This model contains $p$ decision variables and $\frac{p(p-1)}{2}$ `Table` constraints. Each distance constraint is captured as a Table constraint. This option may seem wasteful in terms of memory, but hopefully it can take advantage of efficient Table constraint implementations within solvers.

## 3.2 Modeling the pDD as a CSOP

The models that capture the pDD as the optimization problem that it really is, are as follows:

### 3.2.1 Modeling with Element constraints

$$\begin{aligned} & \texttt{Alldifferent}(X) && (8)\\ y_{ij} &= \texttt{Element}(D, [x_i, x_j]) \quad \forall y_{ij} \in Y,\ 0 \leq i < j < p && (9)\\ & y_{ij} > d_{ij} \quad \forall y_{ij} \in Y,\ 0 \leq i < j < p && (10)\\ & z = \min(Y) && (11)\\ & maximize(z) && (12) \end{aligned}$$

This model contains $p$ decision variables, $\frac{p(p-1)}{2} + 1$ auxiliary variables, $\frac{p(p-1)}{2}$ `Element` and $\frac{p(p-1)}{2}$ unary "greater-than" constraints, plus the constraint $z = \min(Y)$ forcing the auxiliary variable $z$ to be equal to the minimum distance among all pairs of variables, and the objective function that maximizes the value of $z$.

### 3.2.2 Modeling with Table constraints

$$\texttt{Alldifferent}(X) \tag{13}$$
$$y_{ij} = \texttt{Element}(D, [x_i, x_j]) \quad \forall y_{ij} \in Y,\ 0 \leq i < j < p \tag{14}$$
$$\texttt{Table}(T_{ij}, [x_i, x_j]) \quad 0 \leq i < j < p \tag{15}$$
$$z = \min(Y) \tag{16}$$
$$maximize(z) \tag{17}$$

This model contains $p$ decision variables, $\frac{p(p-1)}{2} + 1$ auxiliary variables, $\frac{p(p-1)}{2}$ `Element` and $\frac{p(p-1)}{2}$ `Table` constraints, and the constraint $z = \min(Y)$ plus the objective function. In this case the Element constraints are necessary in order to link the objective variable $z$ to the decision variables.

Although these problems are relatively easy to model, the resulting formulations introduce a large number of auxiliary variables and constraints, with complexity on the order of $\mathcal{O}(p^2)$. As the number of facilities increases into the hundreds and the number of potential location points—determining domain sizes and the size of `Table`/`Element` constraints—reaches into the thousands, solvers often run out of memory due to the large size of the model. This limitation becomes particularly problematic when attempting to handle large-scale instances, as will be clearly illustrated in the experimental section.

## 3.3 A custom implementation

We have also implemented a custom lightweight solver for the pDD, in order to investigate the modeling and algorithmic options in more detail. This solver is basically a straightforward MAC implementation using the first satisfaction model. The difference is that a custom implementation does not require the use of generic CP constructs/global constraints like the Element constraint to access the distance matrix $D$. We simply use the values of the assigned variables as indices to matrix $D$, given that we have direct access to the solver's internal data structures. Hence, instead of an Element constraint and an auxiliary variable $y_{ij}$, for each pair of variables $(x_i, x_j)$, there is a distance constraint $c_{x_i x_j} : D[x_i, x_j] > d_{ij}$, specifying that the distance between the points where $x_i$ and $x_j$ are located must be greater than $d_{ij}$.

As the objective function is not explicitly given in the model, we simply store its value (i.e. the minimum distance between any two facilities in the best solution found so far) and compute the cost of any new solution found so as to determine if this cost is better than the current value of the objective. If so, then the value of the objective is updated.

To leverage such a simplified model in a generic CP solver, which is something that we are currently working on, access to the solver's source code is necessary. This will allow direct indexing into the distance matrix $D$ using the current variable assignments, avoiding the need for Element or Table constraints and/or auxiliary variables.

We now briefly describe how propagation in our solver works, to set the stage for the enhancements detailed below. The propagation technique used by this solver during search is depicted by Function *Propagate* (Algorithm 1). It applies arc consistency on the distance constraints. The algorithm uses a queue to insert and then process variables that have their domain filtered. It is called when a change occurs in the domain of the current variable $x_{cr}$ (e.g. through a value assignment), initializing the queue with this variable. Thereafter, when a variable $x_i$ is removed from the queue, then for each unassigned variable $x_j$ constrained with $x_i$, and each value $v_j \in Dom(x_j)$, it checks if there exists a value $v_i$ in $Dom(x_i)$ s.t. the two values satisfy the distance constraint between $x_j$ and $x_i$ (arc consistency check). If no such $v_i$ exists then $v_j$ is deleted from $Dom(x_j)$, and variable $x_j$ is inserted in the queue to propagate the deletion.

As we demonstrate with an example below, the pruning achieved in this way, is weak because there is no explicit objective function in the model. Hence, an update in the value of $obj_{best}$ is not propagated to the decision variables in $X$. As a result, a solver that uses this model may discover solutions with worse or equal cost to $obj_{best}$, as search progresses. In contrast, the optimization models given above explicitly include the objective function and link it to the decision variables through auxiliary variables and constraints. Hence, any update to $obj_{best}$ will be "fully" propagated.

**Algorithm 1** $Propagate(X, Dom, C, x_{cr})$

support $\leftarrow$ true;
$Q \leftarrow \{x_{cr}\}$;
**while** $Q \neq \emptyset$ **do**
    Select and remove $x_i$;
    **for** $x_j$ where $c_{x_j x_i} \in C, x_j \in X^+_{x_{cr}}$ **do**
        **for** $v_j \in Dom(x_j)$ **do**
            support $\leftarrow$ false;
            **for** $v_i \in Dom(x_i)$ **do**
                **if** $D[v_j, v_i] > d_{ij}$ **then**
                    support $\leftarrow$ true; **break**;
            **if not** support **then**
                $Dom(x_j) \leftarrow Dom(x_j) \setminus \{v_j\}$;
                **if** $Dom(x_j) = \emptyset$ **then**
                    **return** *false*;
        **if** values have been removed from $Dom(x_j)$ **then**
            $Q \leftarrow Q \cup \{x_j\}$;
**return** *true*;

## 4 Experiments with CP Models

The evaluation of the models focuses on memory consumption, CPU time overhead, and solution quality. In our experiments, we considered all models presented in Section 3.

### 4.1 Benchmarks

Following [14, 9], we experimented with instances generated in two different ways. The first uses the benchmark library MDPLIB 2.0 [11] as basis to create pDDs, while in the second we seek to locate facilities on a grid. Experiments were performed on a machine with an Intel Xeon Gold 6230 with 20 CPU cores at 2.10 GHz and 28 GB of main memory. The system features an L1 cache of 1,281 KB, an L2 cache of 20 MB, and an L3 cache of 27.5 MB. The experiments were carried out on an Ubuntu-based operating system, with a time limit of 3,600 seconds for all reported experiments.

The MDPLIB collects a large number of p-dispersion benchmark instances divided into various classes. As in [14], we used some of these instances as basis, to produce pDDs of varying size. We generated 10 instances for each class by randomly adding distance constraints between facilities, using an interval of [0, max/t], where *max* is the maximum distance between any two points and $t$ is the level of tightness for the constraints. In order to produce feasible pDDs (especially with high values of $p$) we have set $t = 8$. We have tried pDDs with 100-2000 candidate facility locations and 10-200 facilities.

Again following [14], we also generated pDDs using the *grid generation model*, which takes the following parameters: $n$, $p$, $|P|$, $t$. We first randomly select $|P|$ among the $n \times n$ nodes of a grid to place the potential facility locations and fill the matrix $D$ with the Manhattan distances between them. For each distance constraint $D[x_i, x_j] > d_{ij}$ between facilities $x_i$ and $x_j$, $d_{ij}$ is randomly set to an integer number in the interval $[0, max/t]$, with $t = 8$.

## 4.2 Experimental results

Tables 1, 2 and 3 demonstrate the performance of the custom CP solver (CP$_c$) and the solvers CP-SAT OR-Tools and CP Optimizer using the models described in Section 3. In Table 2 we give the results obtained using the two satisfaction models 3.1.2 and 3.2.1 (e.g. ORt$_{s1}$ and ORt$_{s2}$, respectively). The geometric mean of the best objective value obtained for all 10 instances of a class, is denoted in columns $obj_b$ while the corresponding geometric mean of cpu time (in seconds) taken for a solver to reach that solution is given in $t_b$ columns. If a solver was unable to find a solution on some instances of a class, we calculate the mean over instances where at least one solution was discovered. We denote the number of such instances using a subscript. In case a solver did not find any solution in all instances of a class, we leave columns $obj$ and $t_b$ blank. Finally, columns *mem* give **approximations** of the memory consumption for each solver in each class. We denote the case of a system crash due to memory exhaustion with X in the mem columns. The same holds for Table 1, where results from our solver are given, and Table 3, where we give results obtained using the two optimization models (e.g. ORt$_{o1}$ and ORt$_{o2}$, respectively) with OR-Tools and CP Optimizer. We do not report total CPU times because all solvers reached the cut off limit of 1 hour in all instances.

**Table 1** Evaluating CP$_c$ in small MDPLIB and grid pDDs.

| **Class** | **CP$_c$** | | |
|---|---|---|---|
| ($\|P\|$,$p$) | $obj_b$ | $t_b$ | mem |
| **MDG** | | | |
| a1 (100,10) | 4.26 | 636 | 2MB |
| a1 (100,20) | 1.57 | 871 | 2MB |
| **GKD** | | | |
| d1 (100,10) | 32.91 | 1,809 | 2MB |
| d1 (250,10) | 29.68 | 168 | 3MB |
| **GRID** | | | |
| g1 (10,80,30) | 1.23 | 0 | 2MB |
| g2 (20,150,50) | 1 | 0 | 2MB |

There are some observations that can be made by looking at the results in the tables. First of all, as was of course expected, the optimization models reached solutions of better quality compared to the satisfaction ones. Looking at Tables 1 and 2, results are mixed regarding solution quality and cpu times. OR-Tools fared better with the second model (Table constraints) in solution quality, cpu time and memory consumption, whereas CP Optimizer fared better with the first model (Element constraints) in terms of solution quality. Note that in class d1 (250,10) OR-Tools (with the first model) reached the time limit during presolve and was not able to locate any solution in all instances of the class. Our solver

**Table 2** Solving satisfaction models in small MDPLIB and grid pDDs.

| Class | $\mathbf{ORt}_{s1}$ | | | $\mathbf{ORt}_{s2}$ | | | $\mathbf{CPopt}_{s1}$ | | | $\mathbf{CPopt}_{s2}$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $(\|P\|,p)$ | $obj_b$ | $t_b$ | mem | $obj_b$ | $t_b$ | mem | $obj_b$ | $t_b$ | mem | $obj_b$ | $t_b$ | mem |
| **MDG** | | | | | | | | | | | | |
| a1 (100,10) | 2.75 | 1,359 | 2GB | 4.06 | 487 | 300MB | 3.79 | 1,157 | 30MB | 2.49 | 35 | 210MB |
| a1 (100,20) | $0{,}93_9$ | 2,339 | 5GB | 1.92 | 886 | 160MB | 1.42 | 862 | 55MB | 1.69 | 695 | 715MB |
| **GKD** | | | | | | | | | | | | |
| d1 (100,10) | 23.87 | 1,832 | 2GB | 33.26 | 1,099 | 300MB | 31.83 | 355 | 30MB | 25.15 | 142 | 210MB |
| d1 (250,10) | - | - | 7GB | 29.91 | 624 | 350MB | 28.35 | 1,128 | 120MB | 18.17 | 94 | 1GB |
| **GRID** | | | | | | | | | | | | |
| g1 (10,80,30) | 1 | 665 | 9GB | 1.07 | 13 | 350MB | 1 | 5 | 99MB | 1 | 1 | 76MB |
| g2 (20,150,50) | - | - | X | 1 | 26 | 2GB | 1 | 29 | 1.5GB | 1 | 8 | 610MB |

**Table 3** Solving optimization models for MDPLIB and grid pDDs.

| Class | $\mathbf{ORt}_{o1}$ | | | $\mathbf{ORt}_{o2}$ | | | $\mathbf{CPopt}_{o1}$ | | | $\mathbf{CPopt}_{o2}$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $(\|P\|,p)$ | $obj_b$ | $t_b$ | mem | $obj_b$ | $t_b$ | mem | $obj_b$ | $t_b$ | mem | $obj_b$ | $t_b$ | mem |
| **MDG** | | | | | | | | | | | | |
| a1 (100,10) | 4.41 | 1,003 | 2GB | 4.66 | 328 | 2GB | 4.68 | 63 | 40MB | 4.68 | 117 | 245MB |
| a1 (100,20) | 0.84 | 2,777 | 6GB | 1.72 | 1,432 | 8GB | 1.78 | 2,185 | 85MB | 1.91 | 732 | 810MB |
| **GKD** | | | | | | | | | | | | |
| d1 (100,10) | 33.03 | 1,370 | 2GB | 34.03 | 1,297 | 2GB | 34.06 | 219 | 47MB | 34.06 | 144 | 245MB |
| d1 (250,10) | - | - | 7GB | - | - | 10GB | 36.09 | 1,736 | 156MB | 36.51 | 1,204 | 1.3GB |
| **GRID** | | | | | | | | | | | | |
| g1 (10,80,30) | - | - | X | - | - | X | 2 | 119 | 115MB | 2 | 50 | 200MB |
| g2 (20,150,50) | - | - | X | - | - | X | 3 | 58 | 845MB | 3 | 66 | 1.5GB |

was competitive in terms of solution quality and cpu times, and crucially, it used negligible amounts of memory compared to the other two solvers.

The results in Table 3 demonstrate that CP Optimizer is clearly a better option than OR-Tools, as the former obtained better solutions in all classes, and was able to handle the grid classes where the latter failed due to memory exhaustion. Regarding the two models in the case of CP Optimizer, there are no significant differences in terms of solution quality, but as was rather expected, the model with the Table constraints consumed a significantly larger amount of memory. The memory requirements of OR-Tools and CP Optimizer do not differ significantly between the two corresponding tables, indicating that the distance constraints are the main bottleneck memory-wise.

## 5 SBJ and max-min pruning

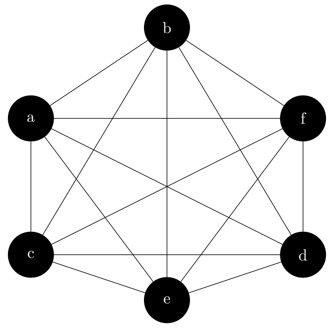
In this section, we show how the limitations of the satisfaction model for the pDD can be overcome through classical CP techniques such as backjumping and dedicated constraint propagation. The combination of our methods allows the solver to mimic Branch&Bound despite the absence of an explicit objective function, and thus to handle large instances efficiently with minimal memory consumption while still producing solutions of high quality. We first give a running example of a pDD and then we describe SBJ and max-min pruning in detail.

## 5.1 A running example of a pDD

▶ **Example 1.** Let us consider a small pDD instance with $p = |X| = 4$ and $|P| = 6$, where the four facilities are to be placed in the network shown in Figure 1b. Let $x_1, \ldots, x_4$ be the variables in $X$ and $Dom(x_i) = P = \{a, b, c, d, e, f\}, 1 \leq i \leq 4$. The distances between any two points in $P$ are given in the 2-D matrix $D$ of Figure 1a. Let a distance constraint $D[x_i, x_j] > 1$ exist between any two variables $x_i$ and $x_j$, $\forall x_i, x_j \in X, i < j$. Given the distances in matrix $D$, it is clear that all distance constraints are satisfied for any pair of distinct points. For simplicity, assume that lexicographic variable/value ordering is used.

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| a | 0 | 3 | 3 | 4 | 5 | 4 |
| b | | 0 | 4 | 6 | 5 | 2 |
| c | | | 0 | 4 | 4 | 4 |
| d | | | | 0 | 5 | 4 |
| e | | | | | 0 | 5 |
| f | | | | | | 0 |

**(a)** Distance matrix D




**(b)** Network instance

**Figure 1 Left.** Distance matrix $D$. **Right.** An example network $G$ for the small pDD instance.

Let us see how the custom CP Solver described in Section 3.3 operates. Variable $x_1$ will be assigned value $a$, and the propagation of the distance constraints will remove $a$ from the domains of all other variables. Similarly for the assignment of $b$ to $x_2$, and so on until the solver finds the first solution $A_1 = \langle x_1 = a, x_2 = b, x_3 = c, x_4 = d \rangle$ having $obj_{A_1} = D[A[x_1], A[x_2]] = D[A[x_1], A[x_3]] = 3$. Hence, $X_c(A_1) = \{\{x_1, x_2\}, \{x_1, x_3\}\}$. After finding the first solution, the solver will try the remaining values for $x_4$ ($e$ and $f$), reaching alternative solutions, but without being able to improve $obj_{best}$, which remains 3. It is clear that unless at least one of the variables in both culprit pairs in $X_c(A_1)$ changes value, $obj_{best}$ cannot be improved. However, the solver, being a chronological backtracker, will then backtrack to $x_3$, trying $x_3 = d$.

When the solver eventually backtracks to $x_2$, it will try the assignment $x_2 = c$, creating the partial assignment $A_{pr} = \langle x_1 = a, x_2 = c \rangle$. The propagation of the distance constraints will remove $c$ from the domains of the future variables. The domains are $Dom(x_2) = \{c\}$, $Dom(x_3) = Dom(x_4) = \{b, d, e, f\}$.

However, values $b$ and $c$ do not have a *max-min support* in $Dom(x_1) = \{a\}$, as $D[a, b] = D[a, c] = 3$. Consequently, they are *max-min inconsistent* and should be removed. The solver fails to detect these inconsistencies since there is no propagation of the updated objective bound. Therefore, the solver will inevitably explore unfruitful paths that lead to equal or worse solutions. Specifically, it will proceed by trying $x_3 = b$, creating $A_{pr} = \langle x_1 = a, x_2 = c, x_3 = b \rangle$, and $b$ will be removed from $Dom(x_4)$. This will lead to the discovery of the following solutions that fail to improve the value of $obj_{best}$: $\langle x_1 = a, x_2 = c, x_3 = b, x_4 = d \rangle$ and $\langle x_1 = a, x_2 = c, x_3 = b, x_4 = e \rangle$, both having cost 3. A new **improved** solution will be found only after the solver backtracks again to $x_2$, assigning $x_2$ with $d$, leading to $\langle x_1 = a, x_2 = d, x_3 = e, x_4 = f \rangle$ with $obj_{best}$ now becoming 4.

It is clear that the custom CP solver suffers from two shortcomings: **(1)** it unnecessarily explores some parts of the search space after finding solutions because it does not identify and exploit culprit pairs to backtrack non-chronologically, and **(2)** it fails to effectively prune domains once $obj_{best}$ is updated, as it does not apply *max-min consistency*. We now detail how these shortcomings can be addressed.

## 5.2 Solution-based backjumping

To address the first inefficiency stated above, we propose a backjumping technique [8, 15, 5] that we refer to as *Solution-Based Backjumping-SBJ*[1], taking advantage of the maxmin optimization criterion in pDDs. This technique is applied as soon as a new solution is discovered. Let us first detail the steps of this method and demonstrate how it affects search, using Example 1.

After a solution $A$ has been discovered, a solver that uses SBJ will follow four steps:

1. Create the culprit set $X_c(A)$.
2. For each culprit pair $x_c = \{x_i, x_j\} \in X_c(A)$, identify the "deepest" variable (e.g. if depth$(x_i) >$ depth$(x_j)$ then $x_i$ is the deepest). Add all such variables to a set $X_d$.
3. Among all $x_d \in X_d$, select the "shallowest" variable. That is, the one with minimum depth among the variables in $X_d$. This variable, denoted as $x_{bj}$, is the one where the backjump occurs.
4. Force search to non-chronologically backtrack to depth$(x_{bj})$.

Before moving on with Example 1, let us clarify Steps 2 and 3, which, as we prove below, guarantee that *SBJ* will not miss improving solutions after a backjump. Consider the simple case where after the discovery of the first solution, there is only one culprit pair $\{x_1, x_2\}$ in $X_c$, with depth$(x_1) <$ depth$(x_2)$, having the assignment $x_1 = v_1, x_2 = v_2$. As $\{x_1, x_2\}$ is the culprit pair, $D[v_1, v_2] = obj_{best}$. If we backjump to $x_1$ and undo the assignment $x_1 = v_1$, we may miss better solutions, as there may exist a value $v_2' \in Dom(x_2)$ such that $D[v_1, v_2'] > obj_{best}$. To avoid this, we must backjump to $x_2$ instead of $x_1$, i.e. to the deepest variable in the pair. In the general case, there may be many culprit pairs that determine the cost of a solution. Suppose that $X_c = \{\{x_1, x_2\}, \{x_1, x_3\}\}$ (with variables assigned in lexicographic order), where $D[v_1, v_2] = D[v_1, v_3] = obj_{best}$. In this case, the value of $obj_{best}$ can be improved only if the assignment of at least one of the variables in each culprit pair is undone. As mentioned, to avoid losing improving solutions, the deepest variable must be chosen from each pair, resulting in the set $X_d = \{x_2, x_3\}$. According to Step 3, we then perform a backjump to the shallowest variable in $X_d$, which is $x_{bj} = x_2$. This is because in general, backjumping to one of the other variables in $X_d$ risks leaving one or more culprit pairs unaffected (i.e. with their assignments intact). For example, selecting $x_3$ as the variable to backjump to would leave the assignment of the culprit pair $\{x_1, x_2\} \in X_c$ unchanged, thereby preventing any further improvement to $obj_{best}$.

Now consider again Example 1. After discovering the first solution $A_1 = \langle x_1 = a, x_2 = b, x_3 = c, x_4 = d\rangle$, the solver will find all culprit pairs and form the set $X_c(A_1) = \{\{x_1, x_2\}, \{x_1, x_3\}\}$ (Step 1). Then, it will create set $X_d = \{x_2, x_3\}$ by selecting the deepest variable from each pair in $X_c(A_1)$ (Step 2) and it will set $x_{bj} = x_2$ (Step 3) because $x_2$ is the shallowest variable in $X_d$. Thus, the solver will perform a backjump to $x_2$ and will skip all remaining nodes in the sub-tree rooted at $\langle x_1 = a, x_2 = b\rangle$. Therefore, it will not unnecessarily discover solutions $\langle x_1 = a, x_2 = b, x_3 = c, x_4 = e\rangle$ and $\langle x_1 = a, x_2 = b, x_3 = c, x_4 = f\rangle$ that are no better than $A_1$.

[1] Not to be confused with Solution Directed Backjumping for Quantified CSPs or QBF [1, 19].

We now prove that SBJ is sound, in the sense that it does not miss any solution that improves the value of the objective as search progresses.

▶ **Property 1.** After SBJ has been applied when a solution $A$ has been found, no solution with better cost than $A$ will be missed during the search process.

**Proof.** (By contradiction) Assume that SBJ is applied after finding a solution $A_1$ with cost $obj_{A_1}$, jumping back to variable $x_{bj}$. Right after backjumping and before unassigning $x_{bj}$, the algorithm restores the domains of all variables $x_j \in X^+_{x_{bj}}$ while keeping the assignments of the variables in $X^-_{x_{bj}}$ intact.

Suppose that there exists a solution $A_2$ such that $obj_{A_2} > obj_{A_1}$ that is missed due to backjumping. Such a solution $A_2$ will have the following property:

$$(A_1[x_i] = A_2[x_i], \forall x_i \in X^-_{x_{bj}} \cup \{x_{bj}\}) \wedge (\exists x_j \in X^+_{x_{bj}} \text{ s.t. } A_1[x_j] \neq A_2[x_j]) \tag{18}$$

meaning that $A_1$ and $A_2$ share the same assignments for the variables in $X^-_{x_{bj}} \cup \{x_{bj}\}$, and differ in at least one assignment for the variables that were restored after backjumping. However, $x_{bj}$ is guaranteed to be part of a culprit pair with a variable $x_i \in X^-_{x_{bj}}$, as $x_{bj} \in X_d$, and $X_d$ includes the deepest variables of each culprit pair. Thus, due to Eq.18, it follows that $obj_{A_2} \leq obj_{A_1}$, contradicting the assumption that $obj_{A_2} > obj_{A_1}$. As a result, it is proved SBJ does not miss any solution with a better cost than $A_1$. ◀

## 5.3 Applying max-min consistency

A property that CP solvers have when solving a pDD with an optimization model is incrementality with respect to the cost of the discovered solutions. That is, any solution discovered during search is guaranteed to be better than all previously discovered ones. This is a standard property of CP solvers that is typically achieved by linking the objective function to the decision variables through auxiliary variables/constraints, allowing for any update to the objective's value to be propagated to the decision variables. As Example 1 demonstrates, our CP solver (or any other solver that uses a satisfaction model) does not have this property. We will now show that the property of solution incrementality can be achieved in a satisfaction model through the application of max-min consistency.

First, we show that there is a very simple, slightly unconventional, way to apply max-min consistency and guarantee that all max-min inconsistent values will be removed. Specifically, we claim that this can be done if the propagation mechanism invoked at each node (after the first solution has been found) does the following : 1) adds **all** variables $x_i \in X$ to the queue in line 2 of Function *Propagate* (Algorithm 1). That is, we replace line 2 ($Q \leftarrow \{x_{cr}\}$) with:

$$Q \leftarrow \{x_i \mid x_i \in X\}.$$

**and** 2) this modified propagation method is called right after a backjump occurs and before trying the next value for $x_{bj}$. We call the modified propagation method *Propagate_maxmin*.

To illustrate how this works, let us consider again Example 1, after the solver backjumps to $x_2$. Function *Propagate_maxmin* gets called and all variables are inserted in $Q$. At some point, $x_1$ will be extracted and variables $x_j \in X^+_{x_2} \cup \{x_2\}$ will be revised. Therefore, checks $D[A_{pr}[x_1], v_2] > obj_{best}, \forall v_2 \in Dom(x_2)$ will detect the inconsistency between values $a$ and $c$, and remove $c$ from $Dom(x_2)$. Furthermore, as propagation goes on, all values $v_j \in Dom(x_j), \forall x_j \in X^+_{x_{bj}}$ will be checked for max-min consistency with $A_{pr}[x_1]$, leaving the domains $Dom(x_3) = Dom(x_4) = \{d, e, f\}$.

After propagation terminates, a new value for $x_2$ will be selected, that is $x_2 = d$, which indeed is max-min consistent with $x_1 = a$. Propagation will remove $d$ from $Dom(x_3)$ and $Dom(x_4)$ (due to distance constraints) and there will be no further removals, since values $e$ and $f$ are max-min consistent. This will lead to the discovery of a new **improved** solution $\langle x_1 = a, x_2 = d, x_3 = e, x_4 = f\rangle$ with $obj_{best} = 4$, having skipped all the intermediate solutions with equal or worse cost compared to the first one.

Also, notice that any removal of a value $v$ from $Dom(x_j)$ at some point during propagation, will lead to the insertion of $x_j$ to $Q$ (lines 15-16, Algorithm 1), so that the deletion will be propagated. This guarantees that any values that become max-min inconsistent during propagation because they lose all their max-min supports, will be also deleted.

We now give our main theoretical result, proving that if max-min pruning is applied, the solver can mimic the effects of Branch&Bound in the satisfaction model.

▶ **Property 2.** The application of Function *Propagate_maxmin* at each node after the first solution has been discovered, guarantees that any solution discovered thereafter will improve the value of $obj_{best}$.

**Proof.** (by contradiction) Assume that after a solution $A$ with $obj_A > obj_{best}$ has been found, and before a solution with better cost than $obj_A$ has been discovered, the solver finds another solution $A'$ with $obj_{A'} \leq obj_A$. Let $x_{bj}$ be the variable where the solver backjumps after discovering solution $A$, and $(x_i, x_j)$ be the culprit pair for $obj_{A'}$, with $A'[x_i] = v_i$ and $A'[x_j] = v_j$, i.e. $obj_{A'} = D[v_i, v_j]$ (the proof can easily be generalized to the case of more than one culprit pairs). Without loss of generality, assume that depth$(x_i)$ < depth$(x_j)$. Now consider that as soon as solution $A$ is discovered, SBJ forces the solver to backjump to depth$(x_{bj})$. It is not possible that depth$(x_j)$ < depth$(x_{bj})$, because in this case we would have $A[x_i] = v_i$ and $A[x_j] = v_j$, and therefore, $obj_A = D[v_i, v_j]$, meaning that $(x_i, x_j)$ would be a culprit pair for $obj_A$ and the solver would have backjumped to depth$(x_j)$. Hence, either depth$(x_i)$ < depth$(x_{bj}) \leq$ depth$(x_j)$ or depth$(x_{bj}) \leq$ depth$(x_i)$.

In the former case, when the solver backjumps to $x_{bj}$, $x_i$ will still be assigned to $v_i$ and $x_j$ will become unassigned. *Propagate_maxmin* will be called and at some point the pair $(x_i, x_j)$ will be revised. As $D[v_i, v_j] \leq obj_{best} = obj_A$, $v_j$ will have no max-min support in $x_i$ and will thus be deleted. Hence, it will not be possible to discover a solution with $x_j = v_j$, as long as $v_i$ is assigned to $x_i$. In the latter case, when search moves forward, extending the branch that will eventually correspond to solution $A'$, it will at some point assign variable $x_i$ with $v_i$. At this point, *Propagate_maxmin* will be called and, for the same reason as above, it will remove $v_j$ from $Dom(x_j)$. Hence, again it will not be possible to discover a solution with $x_i = v_i$, $x_j = v_j$. ◀

While the application of *Propagate_maxmin* enforces max-min consistency, initializing the queue with all variables at every search node can be computationally expensive. We now show that it is not necessary. Consider that after a backjump to depth$(x_{bj})$ has been carried out and propagation through *Propagate_maxmin* has been completed, no max-min inconsistent values will remain in any $Dom(x_j)$ for $x_j \in X^+_{x_{bj}} \cup \{x_{bj}\}$. Now, as search moves forward, all values in $Dom(x_j)$, $x_j \in X^+_{x_{bj}}$, will certainly remain max-min consistent with respect to past assignments, as long as there is no backtrack to a depth higher up the search tree than depth$(x_{bj})$. If there is no such backtrack then the only way in which a value of an unassigned variable can become max-min inconsistent is because of the propagation of the current assignment, meaning that, in this case, it suffices to initialize the queue with $x_{cr}$, as Algorithm 1 does. To take advantage of this and reduce the redundant consistency checks, we propose switching between *Propagate_maxmin* and *Propagate* (i.e. between initiliazing the queue with all variables and only $x_{cr}$), according to the depth of $x_{cr}$, compared to depth($x_{bj}$).

Specifically, after a backjump to depth($x_{bj}$), we propagate with *Propagate_maxmin* to eliminate all inconsistent values $v_j \in Dom(x_j)$ for $x_j \in X^+_{x_{bj}} \cup \{x_{bj}\}$. If, at some point, the solver backtracks to a depth $\leq$ depth($x_{bj}$), we again use *Propagate_maxmin* to propagate any new assignments. However, as long as the search proceeds within $X^+_{x_{bj}}$, we propagate any assignment using Function *Propagate*, and this suffices to maintain max-min consistency. We call this method that switches between the two propagation modes *Propagate_adaptive*.

A byproduct of this property is that the application of Function *Propagate_maxmin* also guarantees that any solution discovered will improve the value of $obj_{best}$.

## 6 Evaluating SBJ and max-min pruning

Tables 4 and 5 compare the custom solver described in Section 3.3 (CP$_c$) that uses Function *Propagate* (Algorithm 1) for propagation to an implementation of the same solver that uses SBJ and applies max-min pruning during search using *Propagate_adaptive* (solver SBJ-PA hereafter). Both use dom/wdeg [2] for variable ordering and lexicographic value ordering. The columns in the tables follow those of Tables 2 and 3. Again, we do not report total CPU times as all solvers reached the cut off limit of 1 hour in all instances, except for the case of SBJ-PA in **a1 (100,10)** and **d1 (100,10)** of Table 4 where the solver terminated in $\approx$ 3,081 and 136,25 seconds in each class, respectively.

We have also tried solving all instances of Table 5 using OR-Tools and CP Optimizer with the optimization models, but the CP solvers were unable to handle such classes, crashing or timing out on all instances due to the size of the constructed model (the same holds for the satisfaction models). This is denoted with X in mem columns. The crash/timeout occured either because of memory exhaustion (mainly in the case of OR-Tools) or because the solver took longer than 1 hour to load the model, due to its size (mainly in the case of CP Optimizer). In contrast, note that CP$_c$ and SBJ-PA only required 37 MB at most for any instance.

Evidently, SBJ-PA locates solutions of much higher quality than CP$_c$ in all classes in both tables, with the differences in the large classes of Table 5 being overwhelming. Focusing on classes MDG a1 (100,10) and GKD d1 (100,10), SBJ-PA was able to prove optimality in all instances, in contrast to the other solvers, terminating successfully within the time limit. On the other hand, it locates slightly worse solutions in class a1 (100,20). But most importantly, SBJ-PA is able to easily handle, memory-wise, the large classes where both OR-Tools and CP Optimizer fail with any of the considered models.

## 7 Conclusions

We evaluated variants of a CP model for the pDD problem that allows it to be modeled and solved by any CP solver. We observed that as instance sizes grow, these models scale poorly, often leading to memory exhaustion and system failures, even if the pDD is viewed as a satisfaction problem. This is due to the inefficient handling of the distance constraints offered by high-level modeling tools, such as the Element and the Table constraints. In contrast, a simple CSP model implemented within a custom CP solver avoids such issues, but at the cost of reduced propagation strength, resulting in lower-quality solutions. To address this trade-off, we have enhanced CP solving for the pDD through two simple but very effective techniques. *Solution Based Backjumping* takes advantage of the maxmin objective to skip

**Table 4** Comparing $CP_c$ and SBJ-PA on small grid and MDPLIB pDDs.

| **Class** | **CP$_c$** | | | **SBJ-PA** | | |
|---|---|---|---|---|---|---|
| ($n$,\|$P$\|,$p$) | $obj_b$ | $t_b$ | mem | $obj_b$ | $t_b$ | mem |
| **GRID** | | | | | | |
| g1 (10,80,30) | 1.23 | 0 | 2MB | 2 | 1 | 2MB |
| g2 (20,150,50) | 1 | 0 | 2MB | 3 | 1 | 2MB |
| **MDPLIB - MDG** | | | | | | |
| a1 (100,10) | 4.26 | 636 | 2MB | 4.68 | 67 | 2MB |
| a1 (100,20) | 1.57 | 871 | 2 MB | 1.65 | 1,664 | 2MB |
| **MDPLIB - GKD** | | | | | | |
| d1 (100,10) | 32.91 | 1,809 | 2MB | 34.06 | 16 | 2MB |
| d1 (250,10) | 29.68 | 168 | 3MB | 36.31 | 1,670 | 3MB |

**Table 5** Comparing $CP_c$ and SBJ-PA on large grid and MDPLIB pDDs.

| **Class** | **CP$_c$** | | | **SBJ-PA** | | | **ORt$_{o1}$** | **ORt$_{o2}$** | **CPopt$_{o1}$** | **CPopt$_{o2}$** |
|---|---|---|---|---|---|---|---|---|---|---|
| ($n$,\|$P$\|,$p$) | $obj_b$ | $t_b$ | mem | $obj_b$ | $t_b$ | mem | mem | mem | mem | mem |
| **GRID** | | | | | | | | | | |
| g1 (60,1K,100) | 1 | 27 | 12MB | 6.19 | 248 | 12MB | X | X | X | X |
| g2 (60,1K,200) | 1 | 297 | 13MB | 4 | 1,158 | 13MB | X | X | X | X |
| **MDPLIB - MDG** | | | | | | | | | | |
| b18 (500,100) | 2.89 | 783 | 5MB | 4.74 | 1,709 | 5MB | X | X | X | X |
| b40 (2K,100) | 4.06 | 26 | 36MB | 41.95 | 723 | 36MB | X | X | X | X |
| b40 (2K,120) | 3.76 | 34 | 37MB | 21.78 | 1,057 | 37MB | X | X | X | X |
| **MDPLIB - GKD** | | | | | | | | | | |
| d1 (500,100) | 1.35 | 4 | 5MB | 7.18 | 70 | 5MB | X | X | X | X |
| d1 (1K,100) | 1.59 | 9 | 12MB | 7.88 | 264 | 12MB | X | X | X | X |
| d1 (1K,200) | 0.88 | 74 | 14MB | 2.85 | 2,161 | 14MB | X | X | X | X |

an exponentially sized portion of the search space, whereas *max-min pruning* guarantees that despite the use of a basic CP model, only improving solutions are discovered as search unravels, by simply checking past assignments against unassigned variables at certain points during search. We experimented with pDDs having up to 2,000 potential locations and 200 facilities. Results demonstrate that applying SBJ in tandem with max-min pruning can result in profoundly improved solutions being discovered, especially in large hard instances that standard solvers cannot handle.

## References


1. Fahiem Bacchus and Kostas Stergiou. Solution directed backjumping for QCSP. In *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 148–163. Springer, 2007.
2. Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *ECAI*, volume 16, pages 96–97, 2004.
3. Richard L Church and Robert S Garfinkel. Locating an obnoxious facility on a network. *Transportation science*, 12(2):107–118, 1978.
4. Zhengguan Dai, Kathleen Xu, and Melkior Ornik. Repulsion-based p-dispersion with distance constraints in non-convex polygons. *Annals of operations research*, 307:75–91, 2021.

5 Rina Dechter and Daniel Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136(2):147–188, 2002. URL: https://www.sciencedirect.com/science/article/pii/S0004370202001200, doi:10.1016/S0004-3702(02)00120-0.
6 Erhan Erkut. The discrete p-dispersion problem. *European Journal of Operational Research*, 46(1):48–60, 1990. URL: https://www.sciencedirect.com/science/article/pii/037722179090297O, doi:10.1016/0377-2217(90)90297-O.
7 Erhan Erkut and Susan Neuman. Analytical models for locating undesirable facilities. *European Journal of Operational Research*, 40(3):275–291, 1989. URL: https://www.sciencedirect.com/science/article/pii/0377221789904207, doi:10.1016/0377-2217(89)90420-7.
8 John Gaschnig. A general backtrack algorithm that eliminates most redundant tests. In *IJCAI-1977*, volume 1, page 457, 1977.
9 Panteleimon Iosif, Nikolaos Ploskas, Kostas Stergiou, and Dimosthenis C. Tsouros. A CP/LS Heuristic Method for Maxmin and Minmax Location Problems with Distance Constraints. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:21, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2024.14, doi:10.4230/LIPIcs.CP.2024.14.
10 Michael J Kuby. Programming models for facility dispersion: The p-dispersion and maxisum dispersion problems. *Geographical Analysis*, 19(4):315–329, 1987.
11 Rafael Martí, Anna Martínez-Gavara, Sergio Pérez-Peló, and Jesús Sánchez-Oro. A review on discrete diversity and dispersion maximization from an or perspective. *European Journal of Operational Research*, 299(3):795–813, 2022. URL: https://www.sciencedirect.com/science/article/pii/S0377221721006548, doi:10.1016/j.ejor.2021.07.044.
12 I Douglas Moon and Sohail S Chaudhry. An analysis of network location problems with distance constraints. *Management Science*, 30(3):290–307, 1984.
13 David Pearce. Economics and genetic diversity. *Futures*, 19(6):710–712, 1987.
14 Nikolaos Ploskas, Kostas Stergiou, and Dimosthenis C. Tsouros. The p-Dispersion Problem with Distance Constraints. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:18, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2023.30, doi:10.4230/LIPIcs.CP.2023.30.
15 Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Comput. Intell.*, 9:268–299, 1993.
16 David Sayah and Stefan Irnich. A new compact formulation for the discrete p-dispersion problem. *European Journal of Operational Research*, 256(1):62–67, 2017. URL: https://www.sciencedirect.com/science/article/pii/S037722171630457X, doi:10.1016/j.ejor.2016.06.036.
17 Fatemeh Sayyady and Yahya Fathi. An integer programming approach for solving the p-dispersion problem. *European Journal of Operational Research*, 253(1):216–225, 2016. URL: https://www.sciencedirect.com/science/article/pii/S0377221716300637, doi:10.1016/j.ejor.2016.02.026.
18 Douglas R Shier. A min-max theorem for p-center problems on a tree. *Transportation Science*, 11(3):243–252, 1977.
19 Lintao Zhang and Sharad Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified boolean formula evaluation. In *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, volume 2470 of *Lecture Notes in Computer Science*, pages 200–215. Springer, 2002.