An exact (re)optimization framework for real-time traffic management

Carlo Mannino[1,2] and Giorgio Sartor[*1]

[1]SINTEF Digital, Oslo, Norway
[2]University of Oslo, Norway

December 23, 2020

Abstract

In real-time traffic management, a new schedule for the vehicles must be computed whenever a deviation from the current plan is detected, or periodically after some time. If this time interval is relatively small, then each two consecutive instances are likely to be similar. We exploit this aspect to derive an exact reoptimization framework for dynamic scheduling problems based on the *Path&Cycle* formulation. The Path&Cycle (PC) is a non-compact formulation for job-shop scheduling, recently introduced as an effective alternative to the classic big-M and time-indexed formulations. In our approach, PC is solved via delayed row generation and the constraints separated while solving previous instances can be adapted to provide a warm start for the current instance. In other words, we maintain over time a pool of valid constraints that adapts to the evolving underlying systems, drastically reducing the computation time of similar consecutive instances. We systematize this novel approach by giving a simple and more direct derivation of PC (also extending its original interpretation), and describing an application to a relevant problem in air traffic management.

1 Introduction

In many real-life applications one needs to solve a sequence of instances of an optimization problem while the underlying system evolves over time. Each new instance is usually only "slightly" different from the previous one, and one can exploit any knowledge acquired by solving previous instances to help solving the subsequent ones. In this paper we address warm-starting mixed integer problems arising in real-time traffic management, but the ideas developed here can be extended to other relevant scheduling problems.

Traffic management (TM) optimization applies to several, critical real-life problems that involve scheduling and re-scheduling of vehicles (see, for example, [1, 19, 20, 29]). In this type of problems, vehicles need to be scheduled in such a way that some capacitated, shared resources are never over-occupied. [22] show that a basic version of TM can be modelled as a job-shop scheduling problem with blocking and no-wait constraints, linking TM to the more general and central theory of job-shop scheduling.

Machine scheduling and job-shop scheduling play a central role in combinatorial optimization and integer programming (see, for example, [23]). The associated decision problems are known to be NP-complete in the general case, but also in the special case of blocking and no-wait constraints ([22, 23]). The complexity stems from the fact that jobs (e.g., vehicles) must be sequenced on shared resources, which corresponds to the solution of a disjunctive program ([5]). A number of integer programming formulations for such disjunctive programs have been studied in the literature

[*]giorgio.sartor@sintef.no

(computational comparisons may be found in [17]). Two major basic models have been adopted and studied in the past decades: *big-M* formulations and *time-indexed* formulations ([24]). Big-M formulations have weak linear relaxations, resulting in large branch&bound trees; recent attempts to strengthen this formulation do not seem particularly promising from a computational standpoint (see [7]). Time-indexed formulations ([12]) have stronger relaxations, but a larger number of variables and constraints, which dramatically increases the time spent to compute the relaxations.

A viable alternative to these two mainstream formulations for the class of job-shop scheduling problems with non-decreasing costs has been recently introduced in [18]. The *Path&Cycle* (PC) formulation is derived by strengthening and lifting the Benders' reformulation of a classical big-M disjunctive formulation, and thus it has the same set of binary variables plus few continuous variables and no big-M constraints. The basic constraints of the PC formulation are in correspondence with the paths and cycles of the disjunctive graph ([5]) associated with the instance. Even if the number of constraints can in principle grow significantly, large instances can be tackled by applying delayed row generation ([18, 30]).

Since the PC formulation represents a new approach to job-shop scheduling problems, it opens up for new research directions, some discussed in the conclusive Section 5. In this paper we attack on some of these potential lines. First, we introduce a more general but simpler version of the PC formulation, which may include routing and other extensions to the basic job-shop scheduling problem. Next, with Lemma 2, we give a different and more "direct" derivation of the PC formulation, using only combinatorial arguments. But the central new development of this paper concerns an effective application of the PC formulation to *reoptimization*.

Reoptimization may occur whenever we solve a sequence of instances of an optimization problem, where the input data only change moderately from an instance to the next[1]. Then, one can exploit the solution to the previous instance(s) to speed up the solution to the next. In some cases, the previous solution is also part of the problem, because one may want the new solution to be "as close as possible" to the previous (so the problem also includes a transition cost, see [31]). In [26], iterative algorithms and real-time optimization are reported as major applications of reoptimization. As for real-time optimization, many real-life applications involve systems that change dynamically over time, as observed in [31]. Several such applications are discussed in [26, 31], including airline disaster recovery, deterministic and stochastic vehicle routing, combinatorial auction, production planning, crew scheduling and facility location, and storage systems.

Despite the large number of relevant practical applications, the great majority of reoptimization papers are devoted to theoretical or algorithmic issues, since the early studies on column generation ([11]) and dynamic graphs ([13]). For a survey on ad-hoc results on graph reopitmization, see for instance [10]. In contrast, we are interested in reoptimization in branch & bound algorithms and Mixed Integer Linear Programming (MILP), see [15,16,21,25,26]. All these papers use information collected in the branching nodes to perform the so called *warm start* to help solve the next MILP program. In particular, [25, 26] focus mainly on the definition of a value function by exploiting the optimal basis of the linear relaxation of the problem associated with each branching node. In [26] this is integrated with primal information to compute an upper bounding function. In [15,16] there is a somehow more direct exploitation of the branching nodes visited at the previous stage. The algorithm solves sequences of MILPs which change only on the objective function coefficients. In particular, the next solution process starts from the leaves of the previous branch and bound tree, suitably adjusted to take into account the new objective. Similarly, [21] provides a framework for recomputing dual bounds of the leaves nodes that are "valid" at the next iteration of the algorithm, even in presence of model disturbances.

Our approach to reoptimization differ from the previous ones, in that rather than storing branching-nodes related information, we preserve and adapt valid constraints generated at previous iterations. The idea is also mentioned in [25, 26] as a possible option, but, surprisingly, we could not find any further discussion. Indeed, preserving cuts seems a natural option already in standard branch and cut algorithms, and it becomes even more compelling when the generated cuts are

[1]One may find formal definitions and some theoretical results related to reoptimization in [31], but such formalism and results go beyond our scope

necessary to the formulation - as in delayed row generation methods. This is the case, for instance, of the sub-tour elimination inequalities for the TSP polytope, and the path and cycle inequalities of the Path&Cycle formulation for the job-shop scheduling problem discussed in this paper.

2 The Path&Cycle formulation

Scheduling problems can be represented by means of an *event graph* $G = (V, E)$. The nodes V are associated with events, for instance an operation starting on a machine, or a vehicle entering a transport network segment. Each (directed) edge in E is associated with a temporal precedence between two events (e.g., the head event can only start after the tail event) with minimum time span specified by the length of the edge. The set V also contains a special node o (the *origin*) to represent a fictitious event that precedes all other events. For every node $u \in V$, a real variable t_u represents the time in which event u occurs. So, an edge $(u, v) \in E$ with length l_{uv} is associated with the (time precedence) constraint on the t variables:

$$t_v - t_u \geq l_{uv}.$$

Since $t_u \geq t_o$ for all $u \in V \setminus \{o\}$, then E always contains edges (o, u) of length $l_{ou} = 0$ for all $u \in V \setminus \{o\}$. Note that G may have parallel edges. In the sequel, for a directed graph $G = (V, E)$ with edge lengths $l \in \mathbb{R}$, and a directed path or cycle Q in G, we let the *length* $l(Q)$ of Q be the sum of the lengths of the edges in Q.

In scheduling problems, precedence constraints between events (i.e., edges in the event graph) may be binding only when certain conditions are satisfied, or when certain decisions are taken. For instance, in disjunctive graphs for job-shop scheduling (see, for example, [4,22,28]), a pair of anti-parallel edges (also called a *disjunctive arc*) represents the alternative orders of two operations on the same machine. When the order is decided (i.e., an edge in the pair is selected) the associated precedence constraint becomes binding while the other one is discarded.

To describe such general scheduling problems, we let $y \in \{0,1\}^E$ be the incidence vector of a set of edges $E(y) \subseteq E$, and we let $G(y) = (V, E(y))$ be the (partial) sub-graph of G induced by $E(y)$. Not all subsets of edges are feasible, and we denote by Y the set of incidence vectors of the feasible subsets. The generic scheduling problem can now be written as:

$$\begin{array}{lll} \min & c(t) & \\ \text{s.t.} & t_v - t_u \geq l_{uv} & (u, v) \in E(y) \\ & y \in Y, t \in \mathbb{R}^V. & \end{array} \tag{1}$$

For many well-studied problems, the set Y can be described by a family of "natural" linear constraints. For instance, for a fixed precedence constraint associated with edge e, we have that all $y \in Y$ satisfy $y_e = 1$. If $\{(u, v), (w, z)\}$ is a disjunctive arc, we have $y_{uv} + y_{wz} = 1$. Another example: in problems involving routing, different edges may be associated with alternative routes and then the corresponding y variables must satisfy a family of multi-commodity flow constraints.

Observe now that, for a fixed $\bar{y}$, problem (1) reduces to the following timing problem ([8,32]):

$$\begin{array}{lll} \min & c(t) & \\ \text{s.t.} & t_v - t_u \geq l_{uv} & (u, v) \in E(\bar{y}) \\ & t \in \mathbb{R}^V, t_o = 0. & \end{array} \tag{2}$$

where we explicitly assume that the event associate with the origin starts at time 0. Then Lemma 1 follows from well-known results (see, for instance, [6]).

Lemma 1. *Program (2) has a solution if and only if $G(\bar{y})$ does not contain strictly positive directed cycles. Then, a feasible solution is given by $t^*_u = L_u(\bar{y})$, $u \in U$, where $L_u(\bar{y})$ is the length of a longest path from o to u in $G(\bar{y})$. Finally, if $c(t)$ is non-decreasing, then t^* is an optimal solution.*

From now on we will consider only non-decreasing cost functions. Note that this condition is normally satisfied in most traffic management problems. In fact, cost usually increases with delay which, in turn, increases with time. For similar practical reasons, we may also assume that $c(\cdot)$ is a non-negative function. Moreover, delays are measured on specific events, such as the arrival of a vehicle at a control point. So, from now on we also assume that $c(t)$ is separable and can be written as $c(t) = \sum_{u \in V} c_u(t_u)$, where $c_u(\cdot)$ is non-decreasing and non-negative. Under these assumptions, it follows from Lemma 1 that solving program (1) is equivalent to solving the following problem:

$$\min_{y \in Y} \left\{ q(y) = \sum_{u \in V} c_u(L^*_u(y)) : \ C \nsubseteq E(y),\ C \in \mathcal{C}_+ \right\}, \tag{3}$$

where $\mathcal{C}_+$ denotes the set of strictly positive directed cycles of $G = (V, E)$. Namely, we want to find $y \in Y$ such that $G(y)$ does not contain a strictly positive directed cycle and the sum of the cost (of the length) of the longest paths in $G(y)$ from o to every node $u \in V$ is minimized.

We denote $\mathcal{P}_u$ as the set of all *simple* paths from o to u in $G = (V, E)$. For $u \in V$, we introduce a variable $\eta_u \in \mathbb{R}_+$ to represent the contribution of node u to the objective function value. Then, Lemma 2 defines the Path&Cycle formulation of problem (3).

Lemma 2. *Problem (3) is equivalent to the following program:*

$$\begin{array}{lllr} \min & f(\eta) = \sum\limits_{u \in V} \eta_u & & \\ \text{subject to} & \sum\limits_{e \in C} y_e - |C| + 1 \le 0, & C \in \mathcal{C}_+ & (i) \\ & c_u\big(l(P)\big) \left(\sum\limits_{e \in P} y_e - |P| + 1 \right) \le \eta_u, & P \in \mathcal{P}_u, u \in V & (ii) \\ & y \in Y, \eta \in \mathbb{R}_+^{|V|}. & & \end{array} \tag{4}$$

Proof. We first show that any optimal solution to (3) can be extended to a feasible solution to (4) with same value, and then show that any optimal solution to (4) can be projected to a feasible solution to (3) with (at most) same value. Constraints (4.i) are referred to as *cycle inequalities* whereas constraints (4.ii) are referred to as *path inequalities.*

1. Suppose (3) admits an optimal solution $\tilde{y}$, so $G(\tilde{y})$ does not contain strictly positive directed cycles. We show that $\tilde{y}$ can be extended to a feasible solution $(\tilde{y}, \tilde{\eta})$ of (4) with $f(\tilde{\eta}) = q(\tilde{y})$. To this end, for $u \in V$, we let $\tilde{\eta}_u = c_u(L^*_u(\tilde{y}))$. Then $f(\tilde{\eta}) = \sum_{u \in V} c_u(L^*_u(\tilde{y})) = q(\tilde{y})$. Next, we need to show that all cycle and path inequalities are satisfied by $(\tilde{y}, \tilde{\eta})$. Suppose there exists a violated (cycle) inequality of type (i) in correspondence with a strictly positive directed cycle $\bar{C}$ of $G = (V, E)$. Then we have

$$\sum_{e \in \bar{C}} \tilde{y}_e > |\bar{C}| - 1.$$

By the integrality of $\tilde{y}$ we have that $\sum\limits_{e \in \bar{C}} \tilde{y}_e = |\bar{C}|$ which implies $\bar{C} \subseteq E(\tilde{y})$ and so $\bar{C}$ belongs to $G(\tilde{y})$, a contradiction (because $\tilde{y}$ is a solution to program (3)). Now suppose that there exists a violated (path) inequality of type (ii), namely there exists $u \in V$ and a simple path $\bar{P} \in \mathcal{P}_u$ from o to $u \in V$ in G such that

$$c_u\big(l(\bar{P})\big) \left(\sum_{e \in \bar{P}} \tilde{y}_e - |\bar{P}| + 1 \right) > \tilde{\eta}_u. \tag{5}$$

Since $\tilde{\eta}_u, c_u(\cdot) \ge 0$, we have $0 < \sum\limits_{e \in \bar{P}_u} \tilde{y}_e - |\bar{P}_u| + 1 = 1$ (the equality holds because of the integrality of $\tilde{y}$). It follows that $\sum\limits_{e \in \bar{P}} \tilde{y}_e = |\bar{P}|$ and $\bar{P} \subseteq E(\tilde{y})$. But then $c_u\big(l(\bar{P})\big) > \tilde{\eta}_u = c_u\big(L^*_u(\tilde{y})\big)$, a contradiction since $L^*_u(\tilde{y})$ is the length of a longest path in $G(\tilde{y})$ and c_u is non-decreasing.

2. We show now that the reverse also holds, that is the projection $\tilde{y}$ of any optimal solution $(\tilde{y}, \tilde{\eta})$ to (4) is feasible for (3) and $q(\tilde{y}) = f(\tilde{\eta})$. Since $\tilde{y} \in Y$ satisfies all cycle inequalities (4.i), then $G(\tilde{y})$ does not contain strictly positive directed cycles and $\tilde{y}$ is feasible for (3). Also, (4.ii) implies that $\tilde{\eta}_u \geq c_u\big(l(\bar{P})\big)$ for any path $\bar{P}$ from o to $u \in V$ in $G(\tilde{y})$, which implies $\tilde{\eta}_u \geq c_u\big(L^*_u(\tilde{y})\big)$. It follows that $f(\tilde{\eta}) = \sum\limits_{u \in V} \tilde{\eta}_u \geq \sum\limits_{u \in V} c_u\big(L^*_u(\tilde{y})\big) = q(\tilde{y})$.

□

Note that the non-negativity of the cost function $c(\cdot)$ can be dropped by rewriting constraint (4.ii) as $(c_u\big(l(P)\big) - \underline{\mathrm{H}}_u)(\sum\limits_{e \in P} y_e - |P| + 1) + \underline{\mathrm{H}}_u \leq \eta_u$, where $\underline{\mathrm{H}}_u$ is a lower bound for η_u. Then Lemma 2 still holds with the small appropriate modifications.

[18] showed how the cycle inequalities (4.i) and the path inequalities (4.ii) can be derived by lifting and strengthening Benders' feasibility and optimality cuts[2], respectively, of a natural big-M formulation of (1). In particular, the cycle inequalities (4.i) correspond to the so called *combinatorial Benders' cuts* discussed in [9], and they are associated with the minimal infeasible subsystems of the original big-M formulation. By analogy, we will refer to constraints (4.ii) as *combinatorial Benders' optimality cuts.*

In the model that we just described, the combinatorial nature of these cuts gives rise to an interesting property: their intepretability. In fact, each cycle inequality in (4.i) reads as "These events cannot occur at the same time", while each path inequality in (4.ii) reads as "If these events occur, then the contribution to the objective function of node u will be at least this amount".

Traffic Management. A traffic management problem is characterized by a set F of vehicles and a transport network, the latter typically decomposed in elementary network resources or segments. In the event graph $G = (V, E)$, each node $u \in V \setminus \{o\}$ is associated with (the event of) a vehicle entering a specific network resource, while edges are associated with different types of precedence constraints.

Fixed constraints. The route of a vehicle is defined as an ordered sequence of network resources. We associate an edge $e = (u, v)$ with each two adjacent resources on the route of a given vehicle $f \in F$, and its length $l_{uv} \geq 0$ is the minimum time it takes vehicle f to traverse the resource associated with u. For every fixed edge (u, v) we have that $y_{uv} = 1$ for all $y \in Y$, because the associated constraint have to be satisfied in every feasible solution. Fixed edges are also used to represent lower bounds on departure times, connections among vehicles, etc. (see [18, 30]).

Non-fixed constraints. In most job-shop scheduling problems, and specifically in traffic management, resources have limited capacity. For example, a railway station can accommodate at most a given number of trains at a time. Consider two vehicles $f, g \in F$ sharing a network resource ρ with capacity $c_\rho \in \mathbb{Z}_+$. Let u, v be the nodes of G associated with f, g entering ρ, respectively. Let $n(u), n(v) \in V$ represent f, g entering the next resource in their routes, i.e., leaving ρ. Then exactly one of the following three possibilities holds:

1. f leaves ρ before g enters ρ. This case is represented by edge $e_1 = (n(u), v)$, with length $l_{e_1} = 0$.

2. g leaves ρ before f enters ρ. This case is represented by edge $e_2 = (n(v), u)$, with length $l_{e_2} = 0$.

3. f and g "meet" in ρ. Then f enters ρ before g leaves ρ and g enters ρ before f leaves ρ. This case is represented by a pair of edges: $e_3 = (u, n(v))$, with length $l_{e_3} = 0$ and $e_4 = (v, n(u))$, with length $l_{e_4} = 0$.

[2]For an extensive discussion on Benders' reformulation, see [33].

Taking one of the above three alternative sequencing decisions (therefore avoiding to exceed capacity) is the central task of traffic management problems. A decision is equivalent to selecting the corresponding edges in the disjunctive graph, and the set of feasible decisions can be modelled by linear (*capacity*) constraints in the y variables (see [18, 30]).

3 Combinatorial Learning

The number of inequalities of the PC formulation can grow exponentially with the nodes and edges of $G = (V, E)$. A suitable technique to cope with this hindrance is the well-known delayed row generation whereby a subset of constraints of the original formulation is generated on the fly only when they are violated by the current incumbent solution (see [3]). In many practical cases, only a fraction of the total number of constraints needs to be generated in order to prove optimality.

Combinatorial learning consists of extending a delayed row generation algorithm to within a reoptimization framework. Intuitively, the algorithm tries to reuse the rows generated while solving previous instances to warm start (maybe after small adjustments) subsequent instances, therefore populating a pool of (combinatorial) constraints that are likely to speed up the computation of similar consecutive instances. The hope is that this pool already contains a subset of the constraints that the delayed row generation algorithm would have generated on the fly if not warm started.

More formally, the Path&Cycle algorithm solves an instance of (4) iteratively by starting with a restricted version R of (4), with only a subset $\mathcal{C}_+^R$ of the constraints (4.i) and only a subset $\mathcal{P}_u^R$ of the constraints (4.ii), for $u \in V$. At iteration i, we solve the restricted problem R to optimality. Since R is a relaxation of (4), if R is infeasible, then (4) is infeasible and we are done. Otherwise, let (y^R, η^R) be the optimal solution to R (note that R cannot be unbounded), with $y^R \in Y$ and $\eta^R \in \mathbb{R}_+^V$. If (y^R, η^R) is feasible for (4), then it is also optimal for (4) and we are done. Otherwise there is at least a violated cycle inequality associated with a positive directed cycle $C \in \mathcal{C}_+ \setminus \mathcal{C}_+^R$ or a violated path inequality associated with a node $u \in V$ and a path $P \in \mathcal{P}_u \setminus \mathcal{P}_u^R$. In the first case, we add C to $\mathcal{C}_+^R$ and iterate, while in the second case we add P to $\mathcal{P}_u^R$ and iterate. The pseudocode is described in Algorithm 1.

Algorithm 1 Path&Cycle algorithm($\bar{\mathcal{C}}_+, \bar{\mathcal{P}}$)

```
$R \leftarrow$ restricted version of (4) where $\mathcal{C}_+^R \leftarrow \bar{\mathcal{C}}_+$, $\mathcal{P}_u^R \leftarrow \bar{\mathcal{P}}_u$, for $u \in V$        ▷ Initialization
loop
    1. Solve $R$
    2. if $R$ is infeasible then break        ▷ Problem infeasible
    3. $(y^R, \eta^R) \leftarrow$ optimal solution to $R$
    4. if $(y^R, \eta^R)$ is feasible for the original problem (4) then break        ▷ Optimal solution
    5. if $y^R$ violates the cycle inequality (associated with) $C \in \mathcal{C}_+ \setminus \mathcal{C}_+^R$ then $\mathcal{C}_+^R \leftarrow \mathcal{C}_+^R \cup \{C\}$
    6. if $(y^R, \eta^R)$ violates the path inequality (associated with) $P \in \mathcal{P}_u \setminus \mathcal{P}_u^R$ then $\mathcal{P}_u^R \leftarrow \mathcal{P}_u^R \cup \{P\}$
return $y^R, \eta^R, \mathcal{C}_+^R, \mathcal{P}^R$
```

Assume now that the set Y can be expressed as the set of vectors in $\{0, 1\}^E$ satisfying a family of linear inequalities. Then, at each iteration, the problem R solved at step 1 is a MILP. The identification of a violated cycle inequality at step 4 amounts to looking for a positive directed cycle in graph $G(y^R)$. Assuming now $G(y^R)$ does not contain strictly positive directed cycles, the identification of a violated path inequality at step 5 amounts to computing a maximum path tree in $G(y^R)$. In both case, the identification of a violated inequality can be performed efficiently ([2]). Note that Algorithm 1 receives as input subsets of valid path and cycle constraints. If these initial sets are empty, then Algorithm 1 is basically an implementation of the classic delayed row generation approach.

Consider now an "extreme" case in which we are solving an instance of (4) twice. The first time we use Algorithm 1 with $\bar{\mathcal{C}}_+ = \emptyset$ and $\bar{\mathcal{P}}_u = \emptyset$, for $u \in V$. The second time we solve the same instance by passing as input to Algorithm 1 the sets $\mathcal{C}_+^R$ and $\mathcal{P}_u^R$, for $u \in V$, returned by the first computation. If we assume that problem R is solved by some deterministic technique, then no additional constraints will be added the second time and R is solved with a single iteration of the internal loop of Algorithm 1.

Indeed, in a reoptimization framework, two consecutive instances are usually very similar. Therefore, the majority of the constraints generated during the solution of the previous instance are likely to be still valid for the subsequent one, possibly after some manipulation. We exploit this observation when generating the input inequality sets for Algorithm 1. In our context, checking for the validity of constraints or adjusting them is straightforward and involves only simple combinatorial operations. Because this process can be viewed as "learning" valid (combinatorial Benders') cuts of the current instance from the past instance(s), we call it *combinatorial learning*.

In particular, let $G = (V, E)$,l be the edge-weighted graph associated with an instance of (4), and let $\tilde{G} = (\tilde{V}, \tilde{E})$, $\tilde{l}$ be the graph associated with the previous instance. Nodes $u \in V$, $\tilde{u} \in \tilde{V}$ are *sibling nodes* if they represent the same event in both instances. Similarly, edges $e = (u, v) \in E$, $\tilde{e} = (\tilde{u}, \tilde{v}) \in \tilde{E}$ are *sibling edges* if $u, \tilde{u}$ and $v, \tilde{v}$ are sibling nodes. Note that the lengths $l_e, \tilde{l}_{\tilde{e}}$ associated with sibling edges $e, \tilde{e}$ may, in general, be different.

Next, we show how to build the input sets $\bar{\mathcal{C}}$, $\bar{\mathcal{P}}$ to Algorithm 1 when solving the current instance, from the sets $\tilde{\mathcal{C}}$, $\tilde{\mathcal{P}}$ returned by Algorithm 1 after solving the previous instance.

Recall that each binary variable y_e appearing in the constraints of (4) is associated with the edge $e \in E$. First, if one or more edges of a cycle $C \in \tilde{\mathcal{C}}_+$ (or of a path $P \in \tilde{\mathcal{P}}$) in $\tilde{G}$ does not have a sibling in G, then the corresponding cycle (or path) inequality is not valid anymore and the corresponding cycle (path) will not belong to $\bar{\mathcal{C}}_+$ ($\bar{\mathcal{P}}$). Otherwise, generates the new (sets of) inequalities as follows:

- for a cycle $C \in \tilde{\mathcal{C}}_+$, compute $l(C)$ by using the lengths $l \in \mathbb{R}^E$ of the sibling edges in G. If $l(C) > 0$, then $C \in \bar{\mathcal{C}}_+$;
- for a path $P \in \tilde{\mathcal{P}}_{\tilde{u}}$, compute $l(P)$ by using the lengths of its sibling edges in G and update the value $c_u\big(l(P)\big)$ for the sibling node u. Then $P \in \bar{\mathcal{P}}_\sqcap$.

While solving a sequence of consecutive instances, the pool of valid path and cycle constraints grows, shrinks, and adapts following the evolution of the underlying system. This is the essence of combinatorial learning. Moreover, by keeping track of which events triggered the separation of these constraints (e.g., a meeting of some vehicles in a resource), one could gather information for identifying bottlenecks in the system. For example, if the only constraints ever separated refer to the meeting/precedence of some vehicles in a particular resource, then this resource could constitute a bottleneck in the system.

4 An example: hotspot resolution in air traffic management

The Path&Cycle formulation was introduced in [18] to solve a real-time train scheduling problem on a single-track railway line in Norway with 16 stations and up to a hundred trains. The results showed significant improvements in computation time when comparing PC to the original big-M formulation. Depending on the objective function (convex, non-convex, step), average speed-ups were up to six times. Further improvements were obtained in [30], where the PC was applied to solve a static version of the Hotspot Problem. In this section, we consider a dynamic version of such problem, where input scenarios evolve over time and a new instance is generated (and solved) at every minute.

The *Hotspot Problem* (HP) arises in air traffic management. The airspace shared by a set of flights F is partitioned into a set S of volumes called *sectors*. The airplanes flying through a sector $s \in S$ are guided by an air traffic controller, who can oversee only a limited number c_s of flights at a time. We let c_s be the capacity of s. Flight schedules are usually available around two hours before their departure time, sometimes even less. Clearly, the introduction of new flight schedules in the model may induce overcrowded sectors. We say that a set of flights $\{f_1, \dots, f_n\} \subseteq F$

produces a hotspot in sector $s \in S$, if $c_s < n$, and their planned schedule admits a point in time in which they are simultaneously in s (see Figure 1).

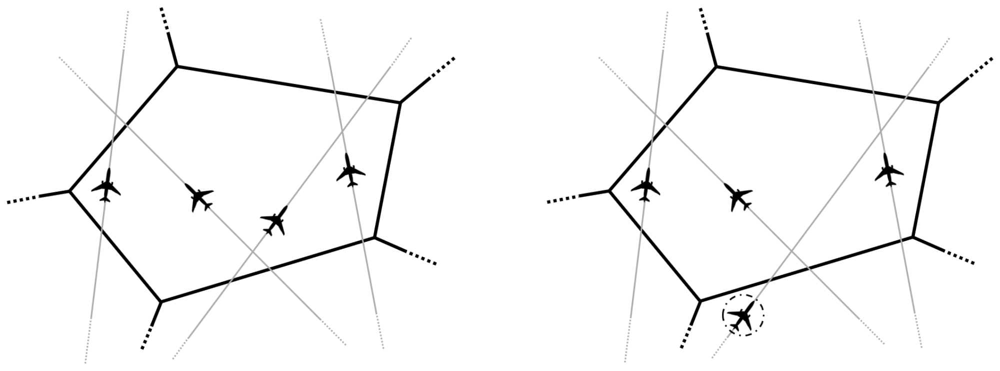
Figure 1: A sector with capacity equal to 3. The original schedule (left) produce an hotspot since 4 airplanes fly through the sector at the same time. By delaying one airplane, the new schedule (right) is hotspot-free.

The Hotspot Problem consists of computing new (possibly delayed) departure times for all the flights in F such that the new schedule is hotspot-free and the total delay (in respect the original schedule) is minimized. Note that in our model flights cannot be delayed after they have departed. This is usually the case in real life as well.

The formulation for HP follows directly from the one described in Section 2. The set Y of (4) is described by a set $\mathcal{H}$ of capacity constraints (in this case called *hotspot* constraints) as discussed at the end of Section 2. In the dynamic setting, HP is solved iteratively after a short interval of time. While time passes, new flight schedules may become available, some flights may depart, and some others may land. When considering a small interval of time, only few of these events may occur. Therefore, the models corresponding to two consecutive iterations are usually very similar.

In the Hotspot Problem, the path and cycle constraints added during one iteration of the Path&Cycle algorithm are eligible for being used to warm start the next iteration[3]. In these computational experiments a set of 10 instances has been randomly generated to represent realistic scenarios where more than 200 airplanes where scheduled to depart from 20 airports in a 3 hour time span. Each schedule was available only 2 hours before departure, and modifications were allowed up until departure. The Path&Cycle algorithm was run at each minute of this 3 hour window in two variants: one that made use of rows generated at previous iterations (PC+), and one that solved the problem each time from scratch (PC). In this experiments the MILP subproblem (called R in Algorithm 1) was solved with CPLEX 12.8 with default settings[4].

In Figure 2 we report relevant statistics for each iteration of one of these instances. The peaks in the computation time correspond to new unexpected events (e.g., the appearance of a new flight schedule), and they initially affect both approaches. However, the computation time of PC+ drops immediately at the following iterations, since the cuts generated in response to those events have already entered the pool of constrains used for the warm start. The number of path, cycle and hotspot constraints used by PC+ grows larger than the one of PC over time, because it includes many of the constraints separated during previous iterations, plus new constraints. It shrinks by the end of the 3 hour interval when many of the airplanes have departed or landed (and the corresponding constraints are not needed anymore). Remarkably, more than 50% of the constraints separated by PC at a given iteration are usually already present in the pool used to warm start PC+ (at the same iteration).

[3]In our experiments, we treated the hotspot constraints similarly to the path and cycle constraints, by separating them only when violated and considering them at the warm start. For more details, see [30].

[4]In the actual implementation, the path and cycle inequalities are separated as lazy constraints at the integral nodes of the branching tree, rather than waiting for the optimal solution to the current MILP.

In Table 1 we report the aggregated results of all 10 instances. The experiments show a significant advantage of using previously separated constraints to warm start subsequent iterations. Note that the "extra" constraints of PC+ introduced by the warm start led to a drastic drop in the number of nodes visited in the branching tree and a substantial reduction in computation time.

Table 1: Static (PC) versus dynamic (PC+) approach on 10 randomly generated problems. The reported numbers represent the average across the 180 sequential iterations.

Name	$\|F\|$	$\|\mathcal{C}_+\| + \|\mathcal{P}\|$		$\|\mathcal{H}\|$		**# nodes**		**Time (s)**		**Speed-up**
		PC	PC+	PC	PC+	PC	PC+	PC	PC+	
ATM1+	323	298	705	18	22	14161	391	4.82	0.99	4.89x
ATM2+	236	37	142	8	11	46	6	0.25	0.1	2.57x
ATM3+	250	493	1092	17	20	6955	839	3.42	1.25	2.74x
ATM4+	215	166	368	15	16	9128	1012	1.65	0.49	3.36x
ATM5+	257	98	272	9	14	1956	511	0.75	0.46	1.63x
ATM6+	220	53	96	6	5	127	27	0.27	0.09	3.08x
ATM7+	218	43	124	10	14	453	83	0.41	0.12	3.45x
ATM8+	236	132	235	8	10	685	108	0.75	0.32	2.37x
ATM9+	238	56	139	10	14	218	52	0.35	0.16	2.22x
ATM10+	250	158	560	12	15	4490	1089	2.55	1.61	1.58x

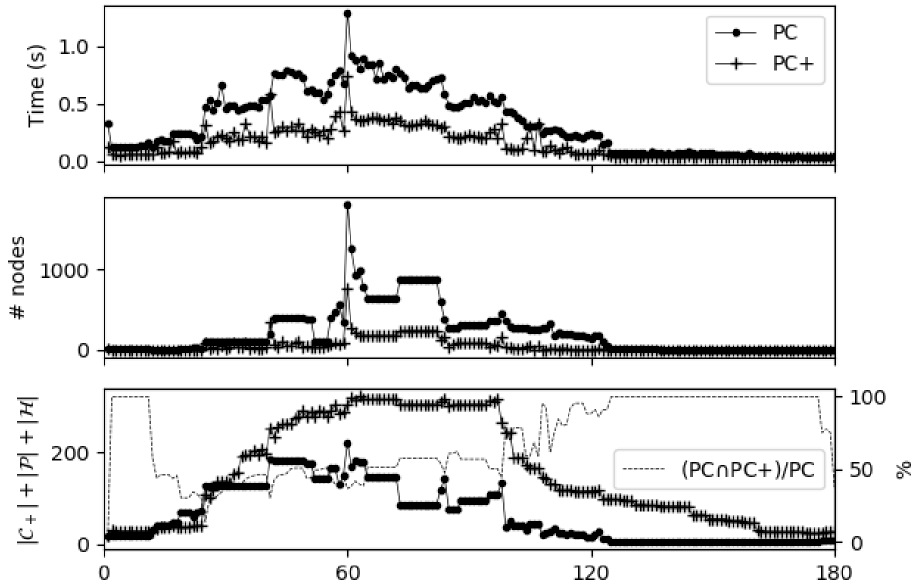

Figure 2: Computation time, number of visited nodes in the branch&bound tree, and total number of constraints for each iteration of the static (PC) and dynamic (PC+) approach when solving ATM9+. The dashed line represents the percentage of constraints of PC that also appear in the pool used to warm start PC+.

The efficiency of the Path&Cycle formulation when solved with Algorithm 1 relies on the fact that only a relatively small number of constraints need to be separated. Real-life instances of traffic management are usually favourable in this regard. Indeed, it is very likely that, at any point in time, vehicles already follow a schedule that is "almost feasible". Yet, it is possible to construct (artificial) instances that are very hard to solve for the Path&Cycle . Consider, for example, a situation where n equivalent vehicles are scheduled to traverse a unit capacity resource at the same time. Then Algorithm 1 will generate the path constraints for all $n!$ optimal sequences.

5 Developments

The Path&Cycle formulation represents a new approach to model and solve job-shop scheduling problems. In this paper we showed its application to traffic management problems, but numerous diverse problems could take advantage of it (e.g., the strip-packing problem). Natural improvements for this novel formulation could consist of identifying stronger valid (possibly facet defining) inequalities and symmetry breaking inequalities.

When solving a sequence of similar MILPs, the previous primal solution can be exploited to quickly generate a good solution to the next instance. Future developments could combine our dynamic "dual" approach with primal heuristics such as Proximity Search ([14]). Besides providing an upper bound, the previous solution can also be exploited to guide the search in the branching phase. This is even more pertinent when the objective function contains a term that penalizes the distance from the previous solution, a typical case in reoptimization problems. Additional research may point towards improving and/or stabilizing the row generation and to extend the overall approach to other delayed row generation algorithms.

A complementary direction of research can focus on studying the relation between this exact combinatorial learning approach and classic machine learning procedures. Each path and cycle constraint can be read as an if-then clause, which can also be represented by a node of a decision tree ([27]). An ad-hoc machine learning algorithm could then be used either as a primal heuristic or to identify relevant cuts that have been not yet separated. Moreover, these interpretable clauses encoded in the pool of valid logical constraints can provide relevant insights on the *physical* constraints of the underlying system. In our example, analysing these constraints can indicate which sector of the airspace should be considered for a capacity upgrade.

References

[1] R Abbou, JM Barman, C Martinez, and S Verma. Dynamic route planning and scheduling in flexible manufacturing systems with heterogeneous resources, a max-plus approach. In *2017 13th IEEE International Conference on Control & Automation (ICCA)*, pages 723–728. IEEE, 2017.

[2] Ravindra K Ahuja, Thomas L Magnanti, James B Orlin, and K Weihe. *Network flows: theory, algorithms and applications*. Prentice Hall, 1993.

[3] Dimitris Alevras, Manfred Padberg, and Manfred W Padberg. *Linear optimization and extensions: problems and solutions*. Springer Science & Business Media, 2001.

[4] Egon Balas. Machine sequencing via disjunctive graphs: an implicit enumeration algorithm. *Operations research*, 17(6):941–957, 1969.

[5] Egon Balas. Disjunctive programming. In *Annals of Discrete Mathematics*, volume 5, pages 3–51. Elsevier, 1979.

[6] Dimitris Bertsimas and John N Tsitsiklis. *Introduction to linear optimization*. Athena Scientific Belmont, MA, 1997.

[7] Pierre Bonami, Andrea Lodi, Andrea Tramontani, and Sven Wiese. On mathematical programming with indicator constraints. *Mathematical programming*, 151(1):191–223, 2015.

[8] Reinhard Bürgy and Kerem Bülbül. The job shop scheduling problem with convex costs. *European Journal of Operational Research*, 268(1):82–100, 2018.

[9] Gianni Codato and Matteo Fischetti. Combinatorial benders' cuts for mixed-integer linear programming. *Operations Research*, 54(4):756–766, 2006.

[10] Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F Italiano. Dynamic graph algorithms. In *Algorithms and theory of computation handbook*, pages 9–9. Chapman & Hall/CRC, 2010.

[11] Martin Desrochers and François Soumis. A reoptimization algorithm for the shortest path problem with time windows. *European Journal of Operational Research*, 35(2):242–254, 1988.

[12] Martin E Dyer and Laurence A Wolsey. Formulating the single machine sequencing problem with release dates as a mixed integer program. *Discrete Applied Mathematics*, 26(2-3):255–270, 1990.

[13] David Eppstein, Zvi Galil, and Giuseppe F Italiano. Dynamic graph algorithms. In *Algorithms and theory of computation handbook*, pages 181–205. CRC Press, 1998.

[14] Matteo Fischetti and Michele Monaci. Proximity search for 0-1 mixed-integer convex programming. *Journal of Heuristics*, 20(6):709–731, 2014.

[15] Gerald Gamrath, Benjamin Hiller, and Jakob Witzig. Reoptimization techniques for mip solvers. In *International Symposium on Experimental Algorithms*, pages 181–192. Springer, 2015.

[16] Benjamin Hiller, Torsten Klug, and Jakob Witzig. Reoptimization in branch-and-bound algorithms with an application to elevator control. In *International Symposium on Experimental Algorithms*, pages 378–389. Springer, 2013.

[17] Ahmet B Keha, Ketan Khowala, and John W Fowler. Mixed integer programming formulations for single machine scheduling problems. *Computers & Industrial Engineering*, 56(1):357–367, 2009.

[18] Leonardo Lamorgese and Carlo Mannino. A non-compact formulation for job-shop scheduling problems in transportation. *Operations Research*, 2019.

[19] Leonardo Lamorgese, Carlo Mannino, Dario Pacciarelli, and Johanna Törnquist Krasemann. Train dispatching. In *Handbook of Optimization in the Railway Industry*, pages 265–283. Springer, 2018.

[20] Elisabeth Lübbecke, Marco E Lübbecke, and Rolf H Möhring. Ship traffic optimization for the kiel canal. *Operations Research*, 2019.

[21] T. Marcucci and R. Tedrake. Warm start of mixed-integer programs for model predictive control of hybrid systems. Technical report, Computer Science and Artificial Intelligence Laboratory (CSAIL), Massachusetts Institute of Technology, 2019.

[22] Alessandro Mascis and Dario Pacciarelli. Job-shop scheduling with blocking and no-wait constraints. *European Journal of Operational Research*, 143(3):498–517, 2002.

[23] Michael Pinedo. *Scheduling. Theory, Algorithms and Systems.* Springer, 2012.

[24] Maurice Queyranne and Andreas S Schulz. *Polyhedral approaches to machine scheduling.* TU, Fachbereich 3, Berlin, 1994.

[25] TK Ralphs and M Güzelsoy. Duality and warm starting in integer programming. In *The Proceedings of the 2006 NSF Design, Service, and Manufacturing Grantees and Research Conference*. Citeseer, 2006.

[26] TK Ralphs, M Güzelsoy, A Mahajan, and S Oshkai. Warm starting for mixed integer linear programs. In *INFORMS Annual meeting, Seattle*, 2007.

[27] Lior Rokach and Oded Z Maimon. *Data mining with decision trees: theory and applications*, volume 69. World scientific, 2008.

[28] Bernard Roy and B Sussmann. Les problemes d'ordonnancement avec contraintes disjonctives. *Note ds*, 9, 1964.

[29] Marcella Samà, Andrea D'Ariano, Paolo D'Ariano, and Dario Pacciarelli. Air traffic optimization models for aircraft delay and travel time minimization in terminal control areas. *Public Transport*, 7(3):321–337, 2015.

[30] Giorgio Sartor and Carlo Mannino. The path&cycle formulation for the hotspot problem in air traffic management. In *ATMOS, Helsinki, Finland*, 2018.

[31] Baruch Schieber, Hadas Shachnai, Gal Tamir, and Tami Tamir. A theory and algorithms for combinatorial reoptimization. *Algorithmica*, 80(2):576–607, 2018.

[32] Thibaut Vidal, Teodor Gabriel Crainic, Michel Gendreau, and Christian Prins. Timing problems and algorithms: Time decisions for sequences of activities. *Networks*, 65(2):102–128, 2015.

[33] Laurence A Wolsey and George L Nemhauser. *Integer and combinatorial optimization*. John Wiley & Sons, 1999.