Fundamenta Informaticae XX (2015) 1–33
DOI 10.3233/FI-2012-0000
IOS Press

Factored Cost-Optimal Planning Using Message Passing Algorithms*

Loïg Jezequel
Université de Nantes, IRCCyN, UMR CNRS 6597
Nantes, France
loig.jezequel@irccyn.ec-nantes.fr

Eric Fabre
INRIA Rennes Bretagne Atlantique
Rennes, France
eric.fabre@inria.fr

Abstract. This paper proposes an approach to solve cost-optimal factored planning problems. Planning consists in organizing actions in order to reach some predefined goal. In factored planning one considers several interacting planning problems and has to design an action plan for each of them. But one must also guarantee that all these local plans are compatible: actions shared among several problems must be jointly performed or jointly rejected. We enrich the problem with the extra requirement that the global plan computed in this modular manner must also minimize the sum of all action costs. A solution is provided to this problem, based on classical message passing algorithms, known as belief propagation in the setting of Bayesian networks. Here, messages carry complex information under the form of weighted (or $(\min, +)$) automata, and all computations are performed with these objects. At the time our first paper on this topic was published, this method was the only one to solve cost-optimal factored planning problems in a modular way. Since then, new approaches were proposed. Experiments on classical benchmarks show that it is a valuable alternative to existing methods.

Keywords: Factored planning, weighted automata, networks of automata, concurrency

*This paper regroups results from two conference papers [13, 14], with the addition of the proofs of these results and of new material (the extensions presented in Sections 4 and 5).

1. Introduction

Planning consists in organizing a set of actions in order to reach some predefined objective. Each action consumes and produces resources, that can be modeled as discrete variables that are read and modified. A state of a planning problem is thus a function that associates a value to each resource (or state variable), and actions modify locally these states, i.e. modify a subset of the state variables. In this vision, one can equivalently consider a planning problem as a graph, where vertices represent states and edges represent actions. Planning then consists in finding a path in this graph (equivalently a sequence of actions, or a *plan*), from the initial state to one of the possible goal states. *Cost-optimal planning* assumes that each action incurs a cost, and one looks for a path reaching a goal state at minimal cost. In practice, planning problems would induce huge graphs, so one would like to avoid the full exploration of this state-space, and to find a path/plan as quickly as possible. For cost-optimal planning, it is even more important to master the search effort, and to balance the computational effort with the expected plan cost reduction.

The current approaches to cost-optimal planning are mainly based on variants of the celebrated A^* algorithm, and use heuristic functions to guide the graph exploration. Heuristic functions associate to any state an estimate of the best cost to reach the goal from that state. With an accurate heuristic, this approach avoids visiting expensive paths, and thus reduces the exploration of the state-space to a narrow beam directed towards the goal. More recently, alternate strategies were proposed to reduce the explored portion of the state graph, under the generic name of *factored planning*. The idea consists in splitting a planning problem into several sub-problems, relying either on a partition of the state variable set or on a partition of the action set. One then solves all sub-problems, with a method ensuring that their local solutions are compatible, i.e. that they can be assembled into a valid global plan. So far, the existing factored planning approaches are limited to ordinary planning problems, and can not address cost-optimal planning problems. Moreover, they generally take the shape of semi-algorithms, that incrementally adjust bounds on the length of paths/plans, so they can hardly decide the non-existence of a solution. The aim of this paper is to propose a general framework for factored planning, which addresses these two limitations.

The rest of this introduction formalizes planning and factored planning problems, gives an overview of the existing factored planning methods and of the one proposed here. Section 2 proves the soundness of our approach, and Section 3 explains how it can be implemented using a weighted automata calculus. Sections 4 and 5 propose two extensions of this method, to accommodate either complexity issues or specific shapes of planning goals. Finally, Section 6 gives some experimental results obtained on classical planning benchmarks.

1.1. Automated planning

A planning problem is a tuple $P = (A, O, i, G)$. A is a set of *atoms* (or binary variables), that represent the resources of the problem. They are either present (true) or absent (false), so a state of the planning problem is actually a subset of A. $O \subseteq 2^A \times 2^A \times 2^A$ is a set of *operators* on these atoms, $i \subseteq A$ is an initial state, and $G \subseteq 2^A$ is a set of goal states. An operator of O takes the form $o = (pre, del, add)$ where these three subsets of A are called *precondition* (pre), *additive effects* (add), and *deleting effects* (del). The semantics is as follows: from a state $s \subseteq A$ the operator $o = (pre, del, add)$ is firable if and only if $pre \subseteq s$, and the firing of o leads to the new state $s' = s \setminus del \cup add$. The objective in such a problem is to find a sequence of operators which, when fired, allows one to reach a state in G, starting from the initial state i. Such a sequence is called a *plan*. Notice that we define A as a set of

atoms, however, other standard representations exist: set of boolean variables, or even set of variables $\{v_k\}_{1\leq k\leq K}$, each one with its own domain D_k.

A planning problems can be transformed into a path finding problem in a directed graph. The states of the planning problem (i.e. the subsets of A) are the vertices of the graph, and the labeled edges correspond to the operators of the planning problem: there is an edge labeled by o from vertex/state s to vertex/state s' if and only if o is firable from s and its firing results in s'. Clearly the labels of a winning path in this graph, leading from the vertex corresponding to i to a vertex corresponding to an element of G, define a plan for the corresponding planning problem, and conversely.

Cost-optimal planning problems are special instances of planning problems where each operator has a cost: $P = (A, O, i, G, c)$ with $c : O \to \mathbb{R}_+$. The objective is no longer to find a plan but to find one with minimal cost, where the cost of a plan $o_1...o_N$ is defined as the sum of the costs of its operators $\sum_{n=1}^{N} c(o_n)$. In graph terms, cost-optimal planning thus reduces to a shortest path problem, where each edge representing operator o takes weight $c(o)$.

The standard method to solve (cost-optimal) planning problems is to use the well-known A^* algorithm proposed by Hart et al. in 1968 [16] (or one of its variants). This algorithm explores the state-space of a problem using a heuristic function, i.e. a function that associates to each state s an estimate $h(s)$ of the cost to reach G from s. A^* proceeds by always extending the search through the most promising state s, among those that were visited so far. The ranking is performed on the basis of a function $f(s)$ which adds up the best known cost $g(s)$ to reach s plus the estimated cost $h(s)$ to go from s to the goal. A^* converges, as it is clearly a variant of breadth-first search. But its main property is elsewhere. If the heuristic function is admissible, i.e. if it associates to each s a lower bound on the best cost to reach the goal from s, then the first valid plan that is obtained is a cost-optimal plan. Given this background, the art of planning concentrates on two essential difficulties: the efficient modeling of a real life planning problems into a representation suited to the A^* algorithm, and the design of good heuristic functions. Numerous contributions have addressed these questions, for various families of planning problems [33, 18, 4, 11, 22, 19, 29, 21].

1.2. Factored planning

The complexity of some planning problems has motivated the exploration of divide and conquer strategies. Some of them consist in looking for plans with given landmarks, where possible intermediate sub-goals are identified before trying to fill in the gaps between such successive landmarks. More recent works have proposed *factored planning* methods, exploiting the structure of a planning problem. The idea consists of splitting the planning problem into several sub-problems, as independent as possible, i.e. with as few interactions as possible. One then solves each of these sub-problems almost independently (as interactions must be taken care of), and merges all these *local plans* into a *global plan*, that is a plan for the original problem. There are essentially two ways of splitting a planning problem: by taking a partition of the set A of atoms [40, 7], or by taking a partition of the set O of operators [2, 6]. The two methods are very similar and rely on the same principles. As the present paper focuses on the first approach, we only describe this one.

Let $A = A_1 \cup \cdots \cup A_N$ be a partition of the atom set of problem $P = (A, O, i, G)$. For clarity, let us also assign distinct names to the actions of O: each action now takes the form $o = (\lambda, pre, del, add)$ where the name λ is a label taken in some finite set Λ. Each A_n induces the sub-problem $P_n = (A_n, O_n, i_n, G_n)$, $1 \leq n \leq N$, where O_n gathers the operators of O interacting with A_n, and trun-

cated to these atoms: for each $o = (\lambda, pre, del, add) \in O$ such that $(pre \cup add \cup del) \cap A_n \neq \emptyset$, one creates $(\lambda, pre \cap A_n, del \cap A_n, add \cap A_n)$ in O_n. In practice, two actions of O_n that only differ by their name should not be distinguished (or should be merged), but for simplicity this detail is ignored. $i_n = i \cap A_n$ is the restriction of the initial state to A_n, and $G_n = \{g \cap A_n \mid g \in G\}$ is the restriction of the set of goal states to A_n. In such a decomposition of $P = (A, O, i, G)$, the objective is to find a local plan p_n in each sub-problem $P_n = (A_n, O_n, i_n, G_n)$ such that these p_n are all compatible: there is a plan p in $P = (A, O, i, G)$ whose restriction to each O_n is exactly p_n, where the restriction of p simply amounts to removing actions that have no effect on A_n, or equivalently actions which names do not appear in O_n. Figure 1 shows some examples of compatible and non-compatible local plans.

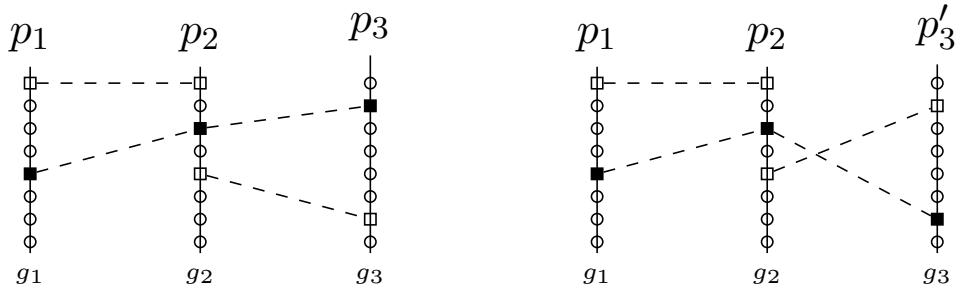

Figure 1. The p_n represent local plans, i.e. sequences of actions, read from top to bottom. Circles represent operators (or action names) appearing in only one sub-problem $P_n = (A_n, O_n, i_n, G_n)$. White boxes stand for operators appearing in two sub-problems, and black boxes stand for operators involving three sub-problems. Dashed lines join local operators with the same name. On the left hand side: the three local plans are compatible. On the right hand side: p_2 and p'_3 are not compatible: the operators they share are not used in the same order.

The methods used to solve factored planning problems usually take the shape of semi-algorithms. They involve some bound on the possible plans that are explored. For example, Brafman and Domshlak [7] bound by K the length of the sequence of shared operators, i.e. operators that appear in more than one subproblem. Given K, the problem is reduced to a constraint satisfaction problem [10] (which aims at finding a sequence of shared operators with length K) coupled with local planning (aiming at filling the gaps between two shared operators in each subproblem P_n with operators appearing only in O_n). If no solution is found, one proceeds with a larger K.

At the time of the publication of our first paper on cost-optimal factored planning [13], no other approaches were able to perform cost-optimal planning in a modular way. Since then, however, a few other solutions have been proposed, allowing several agents to perform cost-optimal planning in a distributed way. One can notice in particular two adaptations of A* to a multi-agent setting [26, 36] and a work on planning with self-interested agents [37].

1.3. Representation of plans

Most approaches to planning, and specifically those based on the A^* algorithm, compute plans as sequences of operators, which is probably the most intuitive representation. However, several authors noticed that it could be more relevant to consider plans as partial orders of operators: this is the case of

GRAPHPLAN presented in Blum and Furst [3] and more recently of the works of Hickmott et al. [23, 5] which rely on unfoldings of concurrent systems. The central observation is that operators generally involve few atoms, therefore two operators acting on different subsets of atoms can fire in any order, which characterises a notion of *concurrency* (or parallelism). Handling plans as partial orders of operators, where this concurrency is well represented, avoids the burden of dealing with all possible interleavings of these operators into sequences, which makes no real sense and generates an explosion of the plan space. For example, assume o_1 and o_2 have disjoint supports, i.e. read and modify different subsets of atoms. Then either both sequences o_1o_2 and o_2o_1 are valid plans, or none of them is, since they produce/reach the same final states. The partial order approach will only explore the unordered pair $\{o_1, o_2\}$ as a plan, and not the two sequences, meaning that these two operators must be used regardless of their order. In the general case, one should thus handle Mazurkiewicz traces [42] of operators rather than sequences.

Interestingly, factored planning is a natural setting for handling plans as partial orders. For example, let us represent local plans as sequences of action names and let $p_1 = \alpha b \delta$ and $p_2 = \alpha c \delta$ be two local plans in subproblems P_1 and P_2 resp., where b and c are private operators of P_1 and P_2 resp. (b only involves variables of P_1, and symmetrically). Then b and c are concurrent and in a global plan they can be fired in any order after α is fired, and before δ is fired. In other words, one could propose $\alpha\{b, c\}\delta$ as a partially ordered plan, instead of any of the two sequences $\alpha bc\delta$ or $\alpha cb\delta$. This is also equivalent to proposing the pair of local plans $(\alpha b\delta, \alpha c\delta)$ as a solution to the factored planning problem (P_1, P_2). We will call such a tuple a *factored plan* in this paper.

1.4. A new approach to factored planning

As the usual approaches to factored planning rely on semi-algorithms, they suffer from two related weaknesses. One is that they can not conclude about the absence of solution, but mostly that they are inadequate to cost-optimal factored planning problems, since when a (best) plan is found at some stage, there is no straightforward guarantee that a better plan could not be obtained at a later stage. The approach presented here adopts another strategy, where all local plans of a given sub-problem P_n are handled at a time, and one progressively removes those that are not compatible with local plans of the other sub-problems, until convergence. It is therefore a top-down approach, that proceeds by filtering out invalid local plans, whereas previous contributions adopt bottom-up approaches and try to *build* plans.

Our top-down approach is made possible thanks to two ingredients. One is the family of message passing algorithms (see [12] for ex.), that proceed with recursive filterings in a peer to peer manner: each sub-problem P_n communicates its constraints to its neighbors, and conversely takes their constraints into account in its own selection of local plans. The second ingredient is the representation of possibly infinite sets of local plans within a compact data structure, namely a regular weighted language, encoded as a weighted automaton. As shown below, one can combine message passing algorithms with computations on weighted automata. This approach is radically new, and of course still amenable to numerous improvements, some of which are suggested in conclusion. However, the objective of this paper is to convince the reader that this new strategy is sound, feasible and even promising.

The next section presents the main framework used in this paper, namely message passing algorithms, and explains how it can be instantiated to perform cost-optimal factored planning.

2. Message passing algorithms for planning

Message passing algorithms (MPA) appeared independently in several communities and at different dates and under different forms, for example to optimally decode convolutional error correcting codes (the Viterbi algorithm), to estimate the state of a Markovian dynamic system (Kalman filtering), to compute the posterior distribution of random variables in a Bayesian network, given the value of some of these variables (Pearl [38]), etc. In this former setting, they also appear under the name belief propagation (BP). The general idea is first to represent graphically the interactions of a set of variables: variables become the nodes of the graph, and edges encode the presence of interaction between the connected variables. For example, these interactions can be hard constraints, soft constraints (probabilities), or combinations of them. The sparser the graph, the less structured are the interactions, in the sense that (conditional) independencies between variables are more numerous, and the more manageable is the global system. As shown below, message passing algorithms solve an inference problem on these variables, i.e. explains how the complete field influences each or some of its variables. These MPA are distributed in nature: they combine messages exchanged on the edges between neighbor variables, and local computations at each node. Section 2.1 proposes a minimal algebraic setting that enables the deployment of these algorithms, as well as some intuition on the links between this setting and factored planning. Section 2.2 then shows formally how factored planning problems can be recast within this framework.

2.1. An abstract version of message passing algorithms

2.1.1. Variables, systems, composition, projection, and the reduction problem

Let $\mathcal{V}_{max}$ be a finite set of variables, each variable $V \in \mathcal{V}_{max}$ taking values in a domain $\mathcal{D}_V$. We consider abstract systems defined over these variables; these systems are generically denoted by S. Systems are provided with a *composition* (or product) operation, denoted as $S_1 \wedge S_2$, which is commutative and associative. We also provide this setting with a *projection* (or reduction) operator: the reduction of system S to a subset of variables $\mathcal{V}$ is denoted as $\Pi_{\mathcal{V}}(S)$. Intuitively, this is similar to the marginalization operation on probabilities, that discards variables ($\mathcal{V}_{max} \setminus \mathcal{V}$) from a joint distribution. The following axiom establishes that operators $\Pi_{\mathcal{V}}$ are projectors:

$$\forall \mathcal{V}_1, \mathcal{V}_2 \subseteq \mathcal{V}_{max}, \forall S,\ \Pi_{\mathcal{V}_1}(\Pi_{\mathcal{V}_2}(S)) = \Pi_{\mathcal{V}_1 \cap \mathcal{V}_2}(S) \tag{1}$$

Further, it is assumed that each system has a limited range, i.e. that it operates over (or is defined on) a limited set of variables:

$$\forall S, \exists \mathcal{V} \subseteq \mathcal{V}_{max},\ \Pi_{\mathcal{V}}(S) = S \tag{2}$$

Combined with (1), there is thus a minimal set $\mathcal{V}$ for which this holds, which can be considered as the domain of system S. The central axiom in this setting is the following. Let systems S_1, S_2 be defined over $\mathcal{V}_1, \mathcal{V}_2$ respectively, then

$$\forall\, \mathcal{V}_3 \supseteq \mathcal{V}_1 \cap \mathcal{V}_2,\ \ \Pi_{\mathcal{V}_3}(S_1 \wedge S_2) = \Pi_{\mathcal{V}_3}(S_1) \wedge \Pi_{\mathcal{V}_3}(S_2) \tag{3}$$

This property establishes that the interactions between two systems are fully captured by their shared variables. It can be considered as an algebraic counterpart of the conditional independence statements that define Bayesian networks: here one would say that $\mathcal{V}_1$ and $\mathcal{V}_2$ (or S_1 and S_2) are conditionally

independent given $\mathcal{V}_3$. Axiom (3) is central in the derivation of MPA. Finally, a last technical axiom is necessary: the existence of a neutral element 1 for composition

$$\forall S,\ S \wedge 1 = S. \tag{4}$$

One could further add the natural property $\forall S,\ \Pi_{\emptyset}(S) = 1$, but this is not necessary to the developments below.

To make this formal setting a little more intuitive, let us illustrate it on a simple example where systems define constraints on the value of variables. Other illustrations could be envisioned : systems could as well define local potential functions as in the case of Gibbs fields (see the definition of spin glasses in statistical physics), and the rest of this paper will actually illustrate the application of this formalism to factored planning.

Let the variable set $\mathcal{V} = \{A, B, C, D\}$ contain four variables, which respective domains are simply denoted as $\mathcal{D}_A = \{a_1, a_2, a_3, ...\}, \mathcal{D}_B = \{b_1, b_2, ...\}$ etc. A system represents possible tuple of values for some of these variables. Let S_1 be defined on variables $\{A, B\}$ for example, with $S_1 = \{(a_1, b_1), (a_1, b_2), (a_2, b_2)\}$. Projections can simply be defined as restrictions to some variables of the tuples permitted by a system. So projecting S_1 on variable A results in $\Pi_A(S_1) = \{a_1, a_2\}$, which means for example that this system does not allow the value a_3 for A. Observe that obviously $\Pi_{A,B}(S_1) = S_1$. Consider now S_2 operating on variables $\{B, C\}$ and defined as $S_2 = \{(b_2, c_2), (b_2, c_3), (b_3, c_3)\}$. One then has for $S = S_1 \wedge S_2$ a system operating on $\{A, B, C\}$ and defined as $S = \{(a_1, b_2, c_2), (a_2, b_2, c_2), (a_1, b_2, c_3), (a_2, b_2, c_3)\}$. Projecting S on the variables of S_1 yields $S'_1 = \Pi_{A,B}(S) = \{(a_1, b_2), (a_2, b_2)\} \subset S_1$, so the tuple (a_1, b_1) has vanished as it could not be extended with a value for C since S_2 has no tuple starting with b_1. Similarly $S'_2 = \Pi_{B,C}(S) = \{(b_2, c_2), (b_2, c_3)\} \subset S_2$ and the pair (b_3, c_3) has vanished. Nevertheless, observe that $S'_1 \wedge S'_2 = S$.

Fig. 2 illustrates a slightly larger example, still on constraint systems. All values of variables appear, components $S_1, ..., S_4$ are defined on pairs of variables (respectively $\{A, B\}, \{B, C\}, \{C, D\}, \{D, A\}$) and are thus represented as edges: edge (a_1, b_1) in S_1 means that S_1 allows this pair of values. Observe that any composition of 3 of these components leads to no "reduction" of the components : projecting for example $S_1 \wedge S_2 \wedge S_3$ on $\{A, B\}$ yields S_1, etc. However, the composition of all four components induces some reduction in the possible pairs of variables that remain valid in each component : $\Pi_{A,B}(S_1 \wedge ... \wedge S_4) = S'_1 \subset S_1$. In the remaining of this Section we formally define this notion of reduction and explain why it is usefull and how it can be computed, using what we call *message passing algorithms*.

2.1.2. The reduction problem

Let $S_1, ..., S_N$ be systems defined over $\mathcal{V}_1, ..., \mathcal{V}_N$ respectively, and consider the compound system $S = S_1 \wedge \ldots \wedge S_N$. The *reduction problem* that we consider here can be stated as follows: compute the projection $\Pi_{\mathcal{V}_n}(S)$ of the global system S on the variable set of each of its components, for $1 \leq n \leq N$. This amounts to understanding how each component S_n is modified once it is connected to all the others. In a Bayesian setting, for example, this can encode the computation of the posterior distribution of a subset of variables $\mathcal{V}_n$ given the observed value of some other variables. The efficient computation of these reduced components $S'_n = \Pi_{\mathcal{V}_n}(S)$ is precisely what the MPA does. In some situations, and quite surprisingly, the reduced components can be derived without first computing S itself. This is a crucial property because computing S can be a very expensive operation, not to mention the projection of the result.

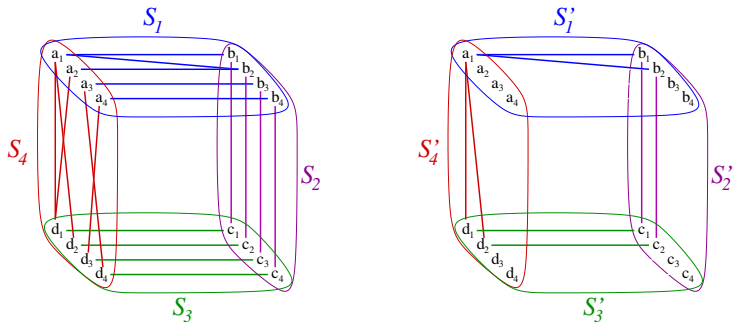

Figure 2. Left: components $S_1, ..., S_4$ defined respectively over variable sets $\{A, B\}, \{B, C\}, \{C, D\}$ and $\{A, D\}$. Each variable can take 4 values. Each component expresses which pairs of values are permitted, for example S_1 allows $\{(a_1, b_1), (a_1, b_2), (a_2, b_2), (a_3, b_3), (a_4, b_4)\}$. Right: the reduced components S'_i, that contain only the pairs of S_i that participate to a valid $4-tuple$. For example, (a_1, b_1) in S'_1 is the restriction to A, B of (a_1, b_1, c_1, d_1) that satisfies the constraints of all 4 components.

2.1.3. Interaction and communication graphs

The first step towards MPA is to associate a graph to a compound system $S = S_1 \wedge \ldots \wedge S_N$. The (non-directed) *interaction graph* of S is defined as: $G = (V, E)$, were $V = \{S_1, \ldots, S_n\}$ and $E = \{(S_i, S_j) \mid \mathcal{V}_i \cap \mathcal{V}_j \neq \emptyset\}$. A *communication graph* of S is obtained by recursively removing *redundant* edges from G, until no more redundant edge remains. An edge (S_i, S_j) is said to be redundant in a graph iff there exists a path $S_i S_{k_1} \ldots S_{k_L} S_j$ in that graph such that $\mathcal{V}_i \cap \mathcal{V}_j \subseteq \mathcal{V}_{k_\ell}$ and $k_L \notin \{i, j\}$ for $1 \leq \ell \leq L$. Figure 3 illustrates these notions. If any communication graph of S is a tree then all the communication graphs of this system are trees [12]; S is then said to *live on a tree*.

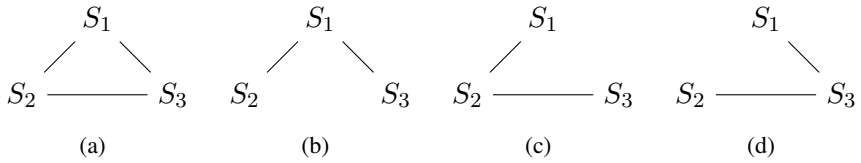

Figure 3. interaction graph of a system $S = S_1 \wedge S_2 \wedge S_3$ such that $\mathcal{V}_1 \cap \mathcal{V}_2 = \mathcal{V}_2 \cap \mathcal{V}_3 = \mathcal{V}_3 \cap \mathcal{V}_1 \neq \emptyset$ (a), and the three corresponding communication graphs (b) (c) (d).

2.1.4. Abstract message passing algorithm

Algorithm 1 is called *message passing algorithm*. It works on a communication graph $G = (V, E)$ of a compound system $S = S_1 \wedge \ldots \wedge S_n$. In this algorithm, $\mathcal{N}(S_i)$ denotes the set containing all the neighbors of S_i in G. The first loop is an initialization step. The second loop is the core of the algorithm: the idea is that each sub-system S_i propagates its knowledge of the compound system S to all its neighbors S_j using the messages $\mathcal{M}_{i,j}$. When messages no longer evolve, the main loop terminates. The third loop yields the reduced components systems S'_i using the messages computed before.

Algorithm 1 message passing algorithm (MPA)

forall $(S_i, S_j) \in E$ **do**
  $\mathcal{M}_{i,j} \leftarrow 1$
done
until stability of messages **do**
  select $(S_i, S_j) \in E$
  $\mathcal{M}_{i,j} \leftarrow \Pi_{\mathcal{V}_i \cap \mathcal{V}_j}(S_i \wedge (\wedge_{S_k \in \mathcal{N}(S_i) \setminus \{S_j\}} \mathcal{M}_{k,i}))$
done
forall $S_i \in V$ **do**
  $S'_i = S_i \wedge (\wedge_{S_k \in \mathcal{N}(S_i)} \mathcal{M}_{k,i})$
done

Theorem 2.1. ([12])
If $S = S_1 \wedge \ldots \wedge S_n$ lives on a tree, then the message passing algorithm converges in finitely many (useful) steps on any communication graph of S, and at convergence one has $\forall 1 \leq i \leq n$, $S'_i = \Pi_{\mathcal{V}_i}(S)$.

In the until loop, an update of $\mathcal{M}_{i,j}$ is "useful" if at least one of the inputs on the right hand side has changed. The proof can be found in [12]. It makes use of the four axioms above, and relies on the following idea. Every edge (S_i, S_j) separates the tree into two parts. Let $S_{i<j} = \wedge_{k \leq i < j} S_k$ denote the product of components lying of the side of S_i from the standpoint of S_j, then the only stationary point of the algorithm is obtained for $\mathcal{M}_{i,j} = \Pi_{\mathcal{V}_i \cap \mathcal{V}_j}(S_{i<j})$. So each message arriving at node S_j summarizes the influence of a whole subtree on S_j. These messages are conditionally independent given $\mathcal{V}_j$ so axiom (3) justifies the final merge equation.

This observation reveals that convergence can be reached with exactly one update of each message. The idea is the following: the main loop should update a message only when all the other messages taking part in its computation were already updated, and no message should be updated twice. Message updates thus start at the leaves of the tree, and one easily sees that the algorithm can not stop, until all messages have been updated exactly once. They then contain the sub-tree summary mentioned above, which can be proved by recursion. This yields Algorithm 2.

Algorithm 2 MPA with efficient message computation

forall $(S_i, S_j) \in E$ **do**
  $\mathcal{M}_{i,j} \leftarrow 1$
done
until all messages were updated exactly once **do**
  select $(S_i, S_j) \in E$ s.t. $\forall S_k \in \mathcal{N}(S_i) \setminus \{S_j\}, \mathcal{M}_{k,i}$ was updated before
  $\mathcal{M}_{i,j} \leftarrow \Pi_{\mathcal{V}_i \cap \mathcal{V}_j}(S_i \wedge (\wedge_{S_k \in \mathcal{N}(S_i) \setminus \{S_j\}} \mathcal{M}_{k,i}))$
done
forall $S_i \in V$ **do**
  $S'_i = S_i \wedge (\wedge_{S_k \in \mathcal{N}(S_i)} \mathcal{M}_{k,i})$
done

2.2. Application to cost-optimal planning

As stated in Section 1.1, a planning problem can be reduced to a shortest path problem in a weighted, directed and labeled graph. Such a graph can be considered as an automaton where goal vertices correspond to marked/accepting states, so the shortest path problem is equivalent to finding an accepted word of minimal cost in this automaton. From this perspective, the set of all plans of a given planning problem can be represented as the (regular) language of a weighted automaton, i.e. a language where costs are associated to words.

This section relies on this formal language interpretation of an optimal planning problem, and shows that factored planning can be expressed in this setting: variables are actions (or action names), systems are weighted languages on this alphabet of actions, and the suitable product ($\times$) and projections ($\mathcal{P}$) are described below. In the following, we formally show our results on the application of weighted language calculus for factored cost-optimal planning, and we also demonstrate them on a running example.

Running example. As a running example we consider the following planning problem $P = (A, O, i, G)$. The set of atoms is $A = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}, \mathtt{G}, \mathtt{H}, \mathtt{I}\}$. The set of operators is $O = \{a, b, c, \alpha, \beta\}$ and the five operators are such that: $a = (\{\mathtt{A}\}, \{\mathtt{A}\}, \{\mathtt{B}, \mathtt{C}\})$, $b = (\{\mathtt{A}\}, \{\mathtt{A}\}, \{\mathtt{B}, \mathtt{D}\})$, $c = (\{\mathtt{H}\}, \{\mathtt{H}\}, \{\mathtt{I}\})$, $\alpha = (\{\mathtt{B}, \mathtt{E}\}, \{\mathtt{B}, \mathtt{E}\}, \{\mathtt{F}\})$, and $\beta = (\{\mathtt{F}, \mathtt{I}\}, \{\mathtt{F}\}, \{\mathtt{G}\})$. The initial state is $i = \{\mathtt{A}, \mathtt{E}, \mathtt{H}\}$ and the goal states in G are all the states containing atom $\mathtt{G}$. As we are interested in cost-optimal planning, we consider a weighted version of this problem, with the cost function c defined as follows: $c(a) = 1$, $c(b) = 2$, $c(c) = 1$, $c(\alpha) = 2$, $c(\beta) = 3$. We also consider a factored version of this problem, defined by the three subsets of atoms $A_1 = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\}$, $A_2 = \{\mathtt{E}, \mathtt{F}, \mathtt{G}\}$, and $A_3 = \{\mathtt{H}, \mathtt{I}\}$. This defines three subproblems (for simplicity we do not distinguish operators from their labels):

- $P_1 = (A_1, O_1, i_1, G_1)$ with $O_1 = \{a, b, \alpha\}$ where $a = (\{\mathtt{A}\}, \{\mathtt{A}\}, \{\mathtt{B}, \mathtt{C}\})$, $b = (\{\mathtt{A}\}, \{\mathtt{A}\}, \{\mathtt{B}, \mathtt{D}\})$, and $\alpha = (\{\mathtt{B}\}, \{\mathtt{B}\}, \emptyset)$, $i_1 = \{\mathtt{A}\}$, and all states belong to G_1;
- $P_2 = (A_2, O_2, i_2, G_2)$ with $O_2 = \{\alpha, \beta\}$ where $\alpha = (\{\mathtt{E}\}, \{\mathtt{E}\}, \{\mathtt{F}\})$ and $\beta = (\{\mathtt{F}\}, \{\mathtt{F}\}, \{\mathtt{G}\})$, $i_2 = \{\mathtt{E}\}$, and $G_2 = \{\{\mathtt{G}\}, \{\mathtt{E}, \mathtt{G}\}, \{\mathtt{F}, \mathtt{G}\}, \{\mathtt{E}, \mathtt{F}, \mathtt{G}\}\}$;
- $P_3 = (A_3, O_3, i_3, G_3)$ with $O_3 = \{c, \beta\}$ where $c = (\{\mathtt{H}\}, \{\mathtt{H}\}, \{\mathtt{I}\})$ and $\beta = (\{\mathtt{I}\}, \emptyset, \emptyset)$, $i_3 = \{\mathtt{H}\}$, and all states belong to G_3.

To each of these subproblems we associate a cost function so that, for each operator, the sum of its local costs in all the subproblems that contain it gives its initial cost in the original problem: $c_1(a) = c(a)$, $c_1(b) = c(b)$, $c_1(\alpha) = 1$, $c_2(\alpha) = 1$, $c_2(\beta) = 1.5$, $c_3(c) = c(c)$, $c_3(\beta) = 1.5$.

Definition 2.2. A *weighted language* $\mathcal{L} = (W, \Sigma, w)$ is a set W of words over the finite alphabet Σ, equipped with a cost function $w : W \to \mathbb{R}_+$.

A planning problem with costs $P = (A, O, i, G, c)$ reduces immediately to this setting: Σ is exactly the set of operators O (or of operator names), a sequence of operators p is in W if and only if p is a plan in P, and the cost $w(p)$ of the word p in $\mathcal{L}$ is the cost $c(p)$ of the plan p in P.

Running example. The weighted language corresponding to P is then $\mathcal{L} = (W, \Sigma, w)$ with $\Sigma = O = \{a, b, c, \alpha, \beta\}$ and $W = \{a\alpha c\beta, ac\alpha\beta, ca\alpha\beta, b\alpha c\beta, bc\alpha\beta, cb\alpha\beta\}$ (the set of plans in P). For any plan $p \in W$ its associated weight is $w(p) = c(a).|p|_a + c(b).|p|_b + c(c).|p|_c + c(\alpha).|p|_\alpha + c(\beta).|p|_\beta$ where $|p|_\sigma$ is the number of occurrences of $\sigma\Sigma$ in p. For example, $w(a\alpha c\beta) = c(a) + c(\alpha) + c(c) + c(\beta) = 6$.

Each subproblem P_i being itself a planning problem, one can also represente it (in fact, its local plans) as a weighted language (where ε denotes an empty word):

- P_1 corresponds to $\mathcal{L}_1 = (W_1, \Sigma_1, w_1)$ with: $\Sigma_1 = \{a, b, \alpha\}$, $W_1 = \{\varepsilon, a, b, a\alpha, b\alpha\}$, $w_1(p) = c_1(a).|p|_a + c_1(b).|p|_b + c_1(\alpha).|p|_\alpha$;
- P_2 corresponds to $\mathcal{L}_2 = (W_2, \Sigma_2, w_2)$ with: $\Sigma_2 = \{\alpha, \beta\}$, $W_2 = \{\alpha\beta\}$, $w_2(\alpha\beta) = c_2(\alpha) + c_2(\beta) = 2.5$;
- P_3 corresponds to $\mathcal{L}_3 = (W_3, \Sigma_3, w_3)$ with: $\Sigma_3 = \{c, \beta\}$, $W_3 = \{\varepsilon, c, c\beta, c\beta\beta, \dots\}$, $w_3(p) = c_3(c).|p|_c + c_3(\beta).|p|_\beta$.

As we want to take Σ as our variable set and weighted languages as our systems, we need both a projection and a product on weighted languages. The projection is in fact the natural projection of languages over a subalphabet associated with a cost minimization.

Definition 2.3. For a word u over alphabet Σ, let us denote by $u_{|\Sigma'}$ the restriction of u to Σ' (obtained by removing all the letters not in Σ' from u). The *projection* $\mathcal{P}_{\Sigma'}(\mathcal{L})$ of a weighted language $\mathcal{L} = (W, \Sigma, w)$ over an alphabet Σ' is the language $\mathcal{L}' = (W', \Sigma', w')$ such that[1]: $W' = W_{|\Sigma'} = \{u_{|\Sigma'} \mid u \in W\}$ and $c'(u') = \min_{u_{|\Sigma'}=u',\ u\in W}\ c(u)$.

Running example. The projection of $\mathcal{L}$ over Σ_1 is the weighted language $\mathcal{P}_{\Sigma_1}(\mathcal{L}) = (W', \Sigma_1, w')$ such that: $W' = \{a\alpha c\beta_{|\Sigma_1}, ac\alpha\beta_{|\Sigma_1}, ca\alpha\beta_{|\Sigma_1}, b\alpha c\beta_{|\Sigma_1}, bc\alpha\beta_{|\Sigma_1}, cb\alpha\beta_{|\Sigma_1}\} = \{a\alpha, b\alpha\}$, $w'(a\alpha) = \min(w(a\alpha c\beta), w(ac\alpha\beta), w(ca\alpha\beta)) = 7$ and $w'(b\alpha) = \min(w(b\alpha c\beta), w(bc\alpha\beta), w(cb\alpha\beta)) = 8$.

Lemma 2.4. The projection of weighted languages satisfies axioms (1) and (2).

Proof:
For (1), let $\mathcal{L} = (W, \Sigma, w)$ and consider alphabets Σ_1 and Σ_2. One must prove $\mathcal{P}_{\Sigma_1}(\mathcal{P}_{\Sigma_2}(\mathcal{L})) = \mathcal{P}_{\Sigma_1\cap\Sigma_2}(\mathcal{L})$. For any word $u \in W$ of $\mathcal{L}$, one has $u' = (u_{|\Sigma_1})_{|\Sigma_2} = u_{|\Sigma_1\cap\Sigma_2}$, so the two projected languages above are clearly defined over the same alphabet $\Sigma' = \Sigma_1 \cap \Sigma_2 \cap \Sigma$, and contain the same words. Word u' also has the same weight in the two projected languages by observing

$$\min_{\substack{u\in W\\ u_{|\Sigma_1\cap\Sigma_2}=u'}} c(u) = \min_{\substack{v\in W_{|\Sigma_2}\\ v_{|\Sigma_1}=u'}} \left(\min_{\substack{u\in W\\ u_{|\Sigma_2}=v}} c(u) \right)$$

Axiom (2) is straightforward by taking $\Sigma' = \Sigma$. □

[1]Strictly speaking, the min should be an inf, as languages can be infinite. But as we deal later with weighted languages generated by weighted automata with positive transition costs, the infimum will be realized by some word.

To define the composition of weighted languages, we adapt the synchronous product of languages, taking costs into account by summing them.

Definition 2.5. Let $\mathcal{L}_i = (W_i, \Sigma_i, w_i)$, $i \in \{1,2\}$, be two weighted languages, their product $\mathcal{L} = \mathcal{L}_1 \times \mathcal{L}_2 = (W, \Sigma, w)$ is defined by $\Sigma = \Sigma_1 \cup \Sigma_2$, $W = \{\, u \in \Sigma^* : u_{|\Sigma_1} \in W_1,\ u_{|\Sigma_2} \in W_2 \,\}$, and $\forall u \in W,\ w(u) = w_1(u_{|\Sigma_1}) + w_2(u_{|\Sigma_2})$.

Running example. The product of $\mathcal{L}_1$ and $\mathcal{L}_2$ is the language $\mathcal{L}_1 \times \mathcal{L}_2 = (W_{1,2}, \Sigma_1 \cup \Sigma_2, w_{1,2})$ such that: $W_{1,2} = \{a\alpha\beta, b\alpha\beta\}$ where $a\alpha\beta$ is obtained from $a\alpha$ in W_1 and $\alpha\beta$ in W_2 and $b\alpha\beta$ is obtained from $b\alpha$ in W_1 and $\alpha\beta$ in W_2, notice that the words ε, a and b in W_1 have no representative in $W_{1,2}$ because any word in W_2 contains α and so the projection on $\Sigma_1 = \{a, b, \alpha\}$ of any word in $W_{1,2}$ is non-empty. Regarding weights, one has $w_{1,2}(a\alpha\beta) = w_1(a\alpha) + w_2(\alpha\beta) = 4.5$ and $w_{1,2}(b\alpha\beta) = w_1(b\alpha) + w_2(\alpha\beta) = 5.5$. One can also notice that $\mathcal{L}_1 \times \mathcal{L}_2 \times \mathcal{L}_3 = \mathcal{L}$, which justifies the use of $\{\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3\}$ as the factored representation of $\mathcal{L}$ in the following.

Lemma 2.6. Product and projection of weighted languages satisfy axiom 3.

Proof:
One has to prove that, for any languages $\mathcal{L}_1 = (W_1, \Sigma_1, w_1)$ and $\mathcal{L}_2 = (W_2, \Sigma_2, w_2)$, and any alphabet $\Sigma_3 \supseteq \Sigma_1 \cap \Sigma_2$, the following holds: $\mathcal{P}_{\Sigma_3}(\mathcal{L}_1 \times \mathcal{L}_2) = \mathcal{P}_{\Sigma_3}(\mathcal{L}_1) \times \mathcal{P}_{\Sigma_3}(\mathcal{L}_2)$. Note $\mathcal{P}_{\Sigma_3}(\mathcal{L}_1 \times \mathcal{L}_2) = (W, \Sigma, w)$ and $\mathcal{P}_{\Sigma_3}(\mathcal{L}_1) \times \mathcal{P}_{\Sigma_3}(\mathcal{L}_2) = (W', \Sigma', w')$. It is clear that $\Sigma = \Sigma' = \Sigma_3 \cap (\Sigma_1 \cup \Sigma_2)$, by definition of product and projection. Also note $\mathcal{P}_{\Sigma_3}(\mathcal{L}_1) = (W_1^3, \Sigma_1^3, w_1^3)$ and $\mathcal{P}_{\Sigma_3}(\mathcal{L}_2) = (W_2^3, \Sigma_2^3, w_2^3)$. And note $\mathcal{L}_1 \times \mathcal{L}_2 = (W_{1,2}, \Sigma_{1,2}, w_{1,2})$.

We first prove that $W = W'$. An intuition of this proof is given in Figure 4. This figure shows two words $u_1 \in W_1$ and $u_2 \in W_2$ and the part of W and W' they generate. Squares depict elements of $\Sigma_1 \cap \Sigma_2$. Notice that all these elements are in Σ_3. Dashed lines depict synchronizations between these elements. White circles depict the other elements of Σ_3 and black circles the elements of Σ_1 and Σ_2 which are not in Σ_3. The sign $\|$ between two parts of words means "all possible interleavings of these parts".

Let $u \in W$, one has by definition of projection that $\exists u_{1,2} \in W_{1,2}$ such that $u_{1,2|\Sigma_3} = u$. Thus, by definition of the product, $\exists u_1 \in W_1$ and $\exists u_2 \in W_2$ such that $u_{1,2|\Sigma_1} = u_1$ and $u_{1,2|\Sigma_2} = u_2$. Then, by definition of the projection, $u_{1|\Sigma_3} \in W_1^3$, and $u_{2|\Sigma_3} \in W_2^3$. Then remark that $u_{1|\Sigma_3} = (u_{1,2|\Sigma_1})_{|\Sigma_3} = (u_{1,2|\Sigma_3})_{|\Sigma_1} = (u_{1,2|\Sigma_3})_{|\Sigma_1 \cap \Sigma_3} = u_{|\Sigma_1 \cap \Sigma_3}$ and similarly $u_{2|\Sigma_3} = u_{|\Sigma_2 \cap \Sigma_3}$. By definition of product this proves that $u \in W'$. It concludes the proof that $W \subseteq W'$.

Let $u \in W'$, one has, by definition of W', $u = u_1^1 u_2^1 \dots u_1^n u_2^n$ with $u_1^i \in (\Sigma_1 \cap \Sigma_3)^*$ and $u_2^i \in (\Sigma_2 \cap \Sigma_3)^*$ for all i. By definition of the product one has $u_{|\Sigma_1 \cap \Sigma_3} = u_1^1 u_1^2 \dots u_1^n$ is a word in W_1^3. Similarly $u_{|\Sigma_2 \cap \Sigma_3} = u_2^1 u_2^2 \dots u_2^n$ is a word in W_2^3. Thus, by definition of the projection, there exists a word $u_1 = u_1^{1'} u_1^{2'} \dots u_1^{n'}$ in W_1 such that $u_{1|\Sigma_3}^{i\,\prime} = u_1^i$ for all i. Similarly there exists a word $u_2 = u_2^{1'} u_2^{2'} \dots u_2^{n'}$ in $\mathcal{L}_2$ such that $u_{2|\Sigma_3}^{i\,\prime} = u_2^i$ for all i. As $\Sigma_3 \supseteq \Sigma_1 \cap \Sigma_2$, the word $u' = u_1^{1'} u_2^{1'} \dots u_1^{n'} u_2^{n'}$ is in $W_{1,2}$. Thus $u'_{|\Sigma_3}$ is a word in W. And $u'_{|\Sigma_3} = u_{1|\Sigma_3}^{1'} u_{2|\Sigma_3}^{1'} \dots u_{1|\Sigma_3}^{n'} u_{2|\Sigma_3}^{n'} = u_1^1 u_2^1 \dots u_1^n u_2^n = u$. This proves that $W' \subseteq W$.

Now prove that, for $u \in W$, $w(u) = w'(u)$. The first step is to show $w(u) \leq w'(u)$. By definition of product, it is known that $w'(u) = w_1^3(u_1^3) + w_2^3(u_2^3)$ for some $u_1^3 \in W_1^3$ and some $u_2^3 \in W_2^3$. By definition

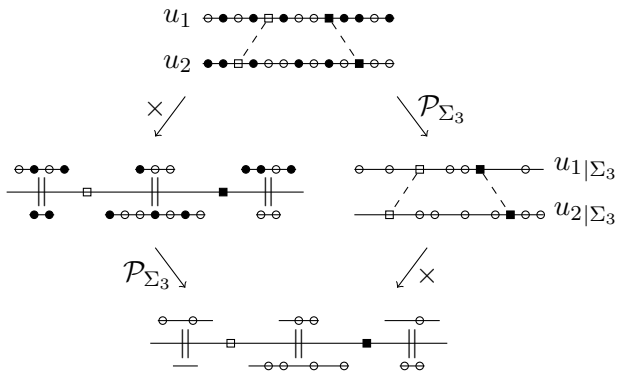

Figure 4. Product and projection of words. Top are two words. Middle left is a representation of their product (which is in fact a set of words). Middle right are the restrictions of these words to Σ_3. Down is a representation of the restriction to Σ_3 of their product, which is also the product of their restrictions to Σ_3.

of projection it is known that $w_1^3(u_1^3) = w_1(u_1)$ for some $u_1 \in W_1$ and $w_2^3(u_2^3) = w_2(u_2)$ for some $u_2 \in W_2$. Moreover, $u_{1|\Sigma_3} = u_1^3$ and $u_{2|\Sigma_3} = u_2^3$. As $\Sigma_3 \supseteq \Sigma_1 \cap \Sigma_2$, it is known that there is $u' \in W_{1,2}$ such that $u'_{|\Sigma_1} = u_1$ and $u'_{|\Sigma_2} = u_2$. Thus, by definition of projection, it is known that $w(u) \leq w_{1,2}(u')$. Moreover, by definition of product, $w_{1,2}(u') = w_1(u_1) + w_2(u_2) = w_1^3(u_1^3) + w_2^3(u_2^3) = w'(u)$. Finally, $w(u) \leq w'(u)$. The remaining is to prove that $w'(u) \leq w(u)$. It is known, by definition of product and projection, that there is $u_1 \in W_1$ and $u_2 \in W_2$ such that $w(u) = w_1(u_1) + w_2(u_2)$. Moreover, it is known that $u \in W'$, so, $u_{\Sigma_1} = u_1^3 \in W_1^3$ and $u_{\Sigma_2} = u_2^3 \in W_2^3$, and $w'(u) = w_1^3(u_1^3) + w_2^3(u_2^3)$. One has, in particular, that $u_{1|\Sigma_3} = u_1^3$ and $u_{2|\Sigma_3} = u_2^3$. By definition of projection, $w_1(u_1) \geq w_1^3(u_1^3)$ and $w_2(u_2) \geq w_2^3(u_2^3)$. Hence, $w'(u) = w_1^3(u_1^3) + w_2^3(u_2^3) \leq w_1(u_1) + w_2(u_2) = w(u)$. Finally $w'(u) \leq w(u)$. It has been proved that $w(u) \leq w'(u)$ and $w'(u) \leq w(u)$ for any $u \in W$. Hence, $w(u) = w'(u)$. □

Taking alphabets as variable sets and weighted languages as systems over these variables, we have proved that the axiomatic setting of Section 2.1 is satisfied. For completeness, axiom (4) holds for $1 = (\{\epsilon\}, \emptyset, 0)$ where ϵ denotes the empty word, and noticing that for every word u, $u_{|\emptyset} = \epsilon$.

Consider now the encoding of an optimal factored planning problem as the compound language $\mathcal{L} = \mathcal{L}_1 \times \ldots \times \mathcal{L}_n$, with $\mathcal{L}_i = (W_i, \Sigma_i, w_i)$. If $\mathcal{L}$ lives on a tree, then the message passing algorithm converges on a communication graph representing the interactions between the $\mathcal{L}_i$, and it yields the $\mathcal{L}'_i = \mathcal{P}_{\Sigma_i}(\mathcal{L})$ for $1 \leq i \leq n$. These reduced components help to solve the optimal factored planning problem thanks to the following property, that derives directly from the definition of projection (one can, for example, immediately check it on the running example):

Property 2.7. Let $\mathcal{L}'_i = (W'_i, \Sigma_i, w'_i) = \mathcal{P}_{\Sigma_i}(\mathcal{L})$ and $\mathcal{L} = (W, \Sigma, w)$. Let u be a word of minimal weight in $\mathcal{L}$, then its projection $u_i = u_{|\Sigma_i}$ has minimal weight in $\mathcal{L}'_i$. Moreover, these minimal weights are identical: $w'_i(u_i) = w(u)$. Conversely, any word u_i of minimal weight in $\mathcal{L}'_i$ is the projection $u_i = u_{|\Sigma_i}$ of a word u of minimal weight in $\mathcal{L}$, and one has $w'_i(u_i) = w(u)$.

In other words, once the MPA has converged, optimal *global* plans result from assembling optimal *local* plans. If each $\mathcal{L}'_i$ contains a unique optimal local plan, then these local plans can be assembled into the unique optimal global plan. In general, some $\mathcal{L}'_i$ will contain several local plans of identical minimal weight. To build an optimal global plan, one thus needs to select a tuple of *compatible* optimal local plans. This again can be solved by an MPA-like procedure. The idea is to select one optimal local plan u_i^* in some component $\mathcal{L}'_i$, that forms a seed $\mathcal{L}_i$" for this selection, initiated at node i. This weighted language $\mathcal{L}_i$" formed of a single word u_i^* initiates message updates towards the neighbours of node i. Let j be one of these neighbors, $\mathcal{L}'_j$ will then select a optimal local word u_j^* compatible with u_i^*, i.e. compatible with the projection of $\mathcal{L}_i$". And so on untill all nodes have been visited. This results in a tuple $(u_i^*)_{1\leq i\leq n}$ of compatible local plans, or equivalently to an optimal global plan encoded as a partial order of actions, which can further be reinterleaved into a proper optimal plan.

This approach to optimal factored planning is actually stronger than necessary. By construction, $\mathcal{L} = \mathcal{L}_1 \times \ldots \times \mathcal{L}_n$ represents *all* possible global plans, and consequently each word u_i in $\mathcal{L}'_i$, i.e. each local plan u_i, is the local view $u_i = u_{|\Sigma_i}$ of some global plan u. One may argue that each $\mathcal{L}_i$, and even the reduced systems $\mathcal{L}'_i$, can be infinite objects, which makes the MPA ineffective. We have actually proved above the soundness of this approach from an algebraic point of view. The next section will show how it can be made practical by replacing weighted languages by weighted automata, i.e finite objects. Before that, we illustrate the use of the language-based MPA on a simple example.

2.3. Using MPA for weighted languages on the running example

The interaction graph of $\mathcal{L}_1$, $\mathcal{L}_2$ and $\mathcal{L}_3$ is a tree, depicted in Figure 5 (left). It is also a communication graph as it contains no redundant edges (no edge can be redundant in a tree). So the MPA will converge on this graph.

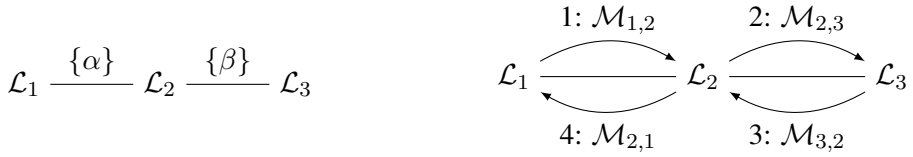

Figure 5. Interaction graph of $\mathcal{L}_1$, $\mathcal{L}_2$, and $\mathcal{L}_3$ (left) and a possible order for computing the messages (right).

According to Algorithm 2, a possible order for computing the messages for a run of the MPA on this communication graph is the one given in Figure 5 (right). This is the order we will use in the following.

One firsts computes $\mathcal{M}_{1,2}$ as $\mathcal{P}_{\Sigma_1\cap\Sigma_2}(\mathcal{L}_1) = (W_{1\to2}, \Sigma_{1\to2}, w_{1\to2})$ with $\Sigma_{1\to2} = \Sigma_1 \cap \Sigma_2 = \{\alpha\}$, $W_{1\to2} = \{\varepsilon_{|\{\alpha\}}, a_{|\{\alpha\}}, b_{|\{\alpha\}}, a\alpha_{|\{\alpha\}}\}, b\alpha_{|\{\alpha\}} = \{\varepsilon, \alpha\}$, $w_{1\to2}(\varepsilon) = \min(w_1(\varepsilon), w_1(a), w_1(b)) = 0$ and $w_{1\to2}(\alpha) = \min(w_1(a\alpha), w_2(b\alpha)) = 2$.

Then, for computing $\mathcal{M}_{2,3}$. One first computes $\mathcal{M}_{1,2} \times \mathcal{L}_2$, a weighted language with alphabet $\Sigma_2 = \{\alpha, \beta\}$; with set of words $\{\alpha\beta\}$; and with weight function assigning weight $w_{1\to2}(\alpha) + w_2(\alpha\beta) = 4.5$ to the word $\alpha\beta$. And then one gets $\mathcal{M}_{2,3}$ as $\mathcal{P}_{\Sigma_2\cap\Sigma_3}(\mathcal{M}_{1,2} \times \mathcal{L}_2) = (W_{2\to3}, \Sigma_{2\to3}, w_{2\to3})$ with: $\Sigma_{2\to3} = \Sigma_2 \cap \Sigma_3 = \{\beta\}$, $W_{2\to3} = \{\alpha\beta_{|\{\beta\}}\} = \{\beta\}$, and $w_{2\to3}(\beta) = w_{1\to2}(\alpha) + w_2(\alpha\beta) = 4.5$.

Similarly, from right to left, one first computes $\mathcal{M}_{3,2}$ as $\mathcal{P}_{\Sigma_3\cap\Sigma_2}(\mathcal{L}_3) = (W_{3\to2}, \Sigma_{3\to2}, w_{3\to2})$ where: $\Sigma_{3\to2} = \Sigma_3 \cap \Sigma_2 = \{\beta\}$, $W_{3\to2} = \{\varepsilon_{|\{\beta\}}, c_{|\{\beta\}}, c\beta_{|\{\beta\}}, cc\beta_{|\{\beta\}}, \ldots\} = \{\varepsilon, \beta\}$, $w_{3\to2}(\varepsilon) =$

$\min(w_3(\varepsilon), w_3(c)) = 0$ and $w_{3\to2}(\beta) = \min(w_3(c\beta), w_3(cc\beta), \dots) = 2.5$.

Then, for computing $\mathcal{M}_{2,1}$. One first computes $\mathcal{M}_{3,2} \times \mathcal{L}_2$ whose alphabet is $\Sigma_2 = \{\alpha, \beta\}$; whose set of words is $\{\alpha\beta\}$; and whose weight function is such that the weight of $\alpha\beta$ is $w_{3\to2}(\beta) + w_2(\alpha\beta) = 5$. And then one gets $\mathcal{M}_{2,1}$ as $\mathcal{P}_{\Sigma_1\cap\Sigma_2}(\mathcal{M}_{3,2} \times \mathcal{L}_2) = (W_{2\to1}, \Sigma_{2\to1}, w_{2\to1})$ with: $\Sigma_{2\to1} = \Sigma_1 \cap \Sigma_2 = \{\alpha\}$, $W_{2\to1} = \{\alpha\beta_{|\{\alpha\}}\} = \{\alpha\}$, and $w_{2\to1}(\alpha) = w_{3\to2}(\beta) + w_2(\alpha\beta) = 5$.

From that, according to the last loop of Algorithm 2, on gets the following three reduced languages:

- $\mathcal{L}'_1 = \mathcal{L}_1 \times \mathcal{M}_{2,1} = (W'_1, \Sigma_1, w'_1)$ with: $W'_1 = \{a\alpha, b\alpha\}$, $w'_1(a\alpha) = w_1(a\alpha) + w_{2\to1}(\alpha) = 7$ and $w'_1(b\alpha) = w_1(b\alpha) + w_{2\to1}(\alpha) = 8$,
- $\mathcal{L}'_2 = \mathcal{L}_2 \times \mathcal{M}_{1,2} \times \mathcal{M}_{3,2} = (W'_2, \Sigma_2, w'_2)$ with: $W'_2 = \{\alpha\beta\}$, $w'_2(\alpha\beta) = w_2(\alpha\beta) + w_{1\to2}(\alpha) + w_{3\to2}(\beta) = 7$,
- $\mathcal{L}'_3 = \mathcal{L}_3 \times \mathcal{M}_{2,3} = (W'_3, \Sigma_3, w'_3)$ with: $W'_3 = \{c\beta\}$, $w'_3(c\beta) = w_3(c\beta) + w_{2\to3}(\beta) = 7$.

From the above computatinos, one can check that taking a word of minimal cost in each reduced language $\mathcal{L}'_1$, $\mathcal{L}'_2$, and $\mathcal{L}'_3$ leads to a word of minimal cost in $\mathcal{L}$ and thus to a cost-optimal solution to the original planning problem P. In fact, the only possibility is to take $a\alpha$ (cost 7) in $\mathcal{L}'_1$, $\alpha\beta$ (cost 7) in $\mathcal{L}'_2$, and $c\beta$ (cost 7) in $\mathcal{L}'_3$. These three words can be interleaved as either $ca\alpha\beta$, $ac\alpha\beta$, or $a\alpha c\beta$ which are three cost-optimal solutions to P, all of them with cost 7.

3. Weighted automata approach

As mentioned above, the weighted language approach to minimal cost factored planning is algebraically sound, but may not be practical, as the languages involved in the computations may be infinite. Fortunately, the weighted language describing an optimal planning problem is regular, i.e. can be encoded as a weighted automaton [9, 39]. And the operations of the previous section (composition and projection) preserve this regularity. Based on this remark, this section shows that all computations can actually be performed with automata, which are finite objects. We rely in particular on an algorithm from [35] for the projection.

Definition 3.1. A *weighted automaton* $\mathcal{A} = (S, T, \Sigma, s^0, F, c, f)$ is a standard automaton equipped with a cost-function on transitions and on marked states. Specifically, S is a finite state set and Σ a finite alphabet, $T \subseteq S \times \Sigma \times S$ is a transition set, $s^0 \in S$ an initial state and $F \subseteq S$ a set of final (or marked) states. The cost functions are $c : T \to \mathbb{R}_+$ on transitions, and $f : F \to \mathbb{R}_+$ on final states.

In such an automaton, a path $\pi = t_1 \dots t_n$ is a sequence of transitions such that there exist states $s_0, ..., s_n$ in S satisfying $t_i = (s_{i-1}, a_i, s_i)$, $1 \le i \le n$. The word corresponding to such a path is $\sigma(\pi) = a_1 \dots a_n$. Path π is said to be accepted by $\mathcal{A}$ if and only if $s_0 = s^0$ and $s_n \in F$. A word $u \in \Sigma^*$ is said to be accepted by $\mathcal{A}$ if and only if there exists a path π accepted by $\mathcal{A}$ such that $u = \sigma(\pi)$. The cost of an accepted path is $c(\pi) = c(t_1) + \dots + c(t_n) + f(s_n)$, and the cost of an accepted word u is $c(u) = min_{u=\sigma(\pi)} c(\pi)$. The language $\mathcal{L}(\mathcal{A}) = (W, \Sigma, w)$ of a weighted automaton $\mathcal{A}$ is such that W is the set of all words accepted by $\mathcal{A}$, and for $u \in W$, $w(u) = c(u)$.

As mentioned above, a planning problem with costs $P = (A, O, i, G, c)$ can be recast as a shortest path problem into a weighted directed graph. Similarly it can be translated into the problem of finding a

word of minimal cost in a weighted automaton $\mathcal{A} = (S, T, \Sigma, s, F, c', f)$, defined as: $S = \mathcal{P}(A) \triangleq 2^A$, $\Sigma = O$, $T = \{(s_1, o, s_2) \mid o \text{ is firable from } s_1 \text{ and leads to } s_2\}$, $s^0 = i$, $F = G$, $c'((s_1, o, s_2)) = c(o)$ for $o \in O$, and $f(s) = 0$ for all $s \in F$. The weights on final sets vanish here, but their necessity becomes obvious in the computations involved in factored planning (see below).

Running example. The weighted automaton $\mathcal{A}$ corresponding to P is represented in Figure 6. The states are represented as circles (each one with the corresponding set of atoms written inside), the transitions are represented by arrows whose labels give the corresponding actions and costs, the initial state is the one with an ingoing arrow outgoing from no state, and the final states are represented by double circles. Similarly, the weighted automata $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$ corresponding respectively to the subproblems P_1, P_2, and P_3 are represented in Figure 7. One can notice that $\mathcal{L}(\mathcal{A}) = \mathcal{L}$, $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}_1$, $\mathcal{L}(\mathcal{A}_2) = \mathcal{L}_2$, and $\mathcal{L}(\mathcal{A}_3) = \mathcal{L}_3$.

3.1. Product of weighted automata

Definition 3.2. The product of two weighted automata $\mathcal{A}_i = (S_i, T_i, \Sigma_i, s_i^0, F_i, c_i, f_i)$, $i \in \{1, 2\}$ is the weighted automaton $\mathcal{A}_1 \| \mathcal{A}_2 = (S, T, \Sigma, s^0, F, c, f)$, extending the synchronous product of automata as follows: $S = S_1 \times S_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $s^0 = (s_1^0, s_2^0)$, $F = F_1 \times F_2$, and for transitions

$$\begin{aligned} T &= \{((s_1, s_2), \alpha, (s_1', s_2')) \mid (s_1, \alpha, s_1') \in T_1 \ \wedge \ (s_2, \alpha, s_2') \in T_2\} \\ &\cup \{((s_1, s_2), \alpha, (s_1', s_2)) \mid (s_1, \alpha, s_1') \in T_1 \ \wedge \ \alpha \notin \Sigma_2\} \\ &\cup \{((s_1, s_2), \alpha, (s_1, s_2')) \mid (s_2, \alpha, s_2') \in T_2 \ \wedge \ \alpha \notin \Sigma_1\}, \end{aligned}$$

Regarding cost functions, for $t = ((s_1, s_2), \alpha, (s_1', s_2')) \in T$, if $\alpha \in \Sigma_1 \cap \Sigma_2$ then $c(t) = c_1((s_1, \alpha, s_1')) + c_2((s_2, \alpha, s_2'))$, if $\alpha \in \Sigma_1 \setminus \Sigma_2$ then $c(t) = c_1(s_1, \alpha, s_1')$, and symmetrically. Finally, for $(s_1, s_2) \in F$, $f((s_1, s_2)) = f_1(s_1) + f_2(s_2)$.

Running example. Figure 8 (left) represents the product of $\mathcal{A}_1$ and $\mathcal{A}_2$ (Figure 7). One can notice that the product of this automaton with $\mathcal{A}_3$ (Figure 7) is exactly $\mathcal{A}$ (Figure 6), that is the representation of the full planning problem.

Lemma 3.3. Let $\mathcal{A}_1, \mathcal{A}_2$ be two weighted automata, then $\mathcal{L}(\mathcal{A}_1 \| \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \times \mathcal{L}(\mathcal{A}_2)$.

In other words, the synchronous product of automata "implements" (or can be seen as a finite representation of) the product of languages.

Proof:
Note $\mathcal{L}(\mathcal{A}_1) \times \mathcal{L}(\mathcal{A}_2) = (W, \Sigma, w)$ and $\mathcal{L}(\mathcal{A}_1 \| \mathcal{A}_2) = (W', \Sigma', w')$. Moreover, note $\mathcal{L}_1 = (W_1, \Sigma_1, w_1) = \mathcal{L}(\mathcal{A}_1)$ and $\mathcal{L}_2 = (W_2, \Sigma_2, w_2) = \mathcal{L}(\mathcal{A}_2)$. It is clear that $\Sigma = \Sigma' = \Sigma_1 \cup \Sigma_2$. First show that $W = W'$.

Let u be a word from W. Suppose that $u \notin W'$. Thus there is no path π in $\mathcal{A}_1 \| \mathcal{A}_2$ such that $\sigma(\pi) = u$. Hence, by definition of product of weighted automata, either there is no path π_1 in $\mathcal{A}_1$ such that $\sigma_1(\pi_1) = u_{|\Sigma_1}$ or there is no path π_2 in $\mathcal{A}_2$ such that $\sigma_2(\pi_2) = u_{|\Sigma_2}$. So, by definition of product of weighted languages, $u \notin W$. It proves that if $u \in W$ and $u \notin W'$, then $u \notin W$. Hence, $W \subseteq W'$.

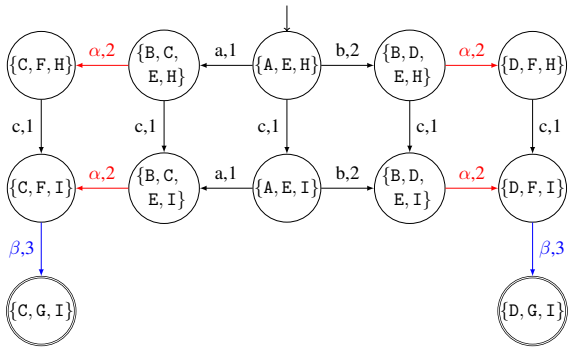

Figure 6. A weighted automaton representation of the planning problem P from the running example. Observe the numerous concurrency diamonds due to concurrent actions.

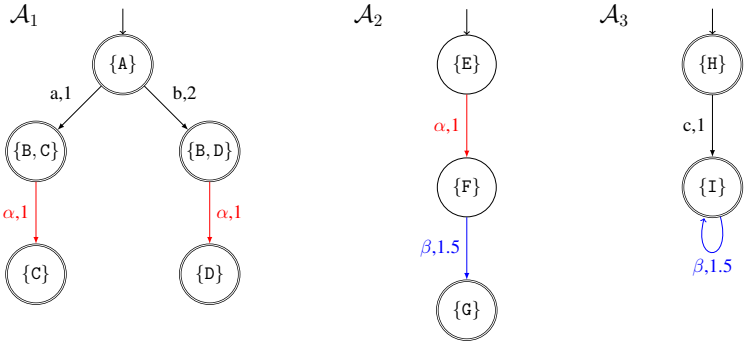

Figure 7. Weighted automata representations of the subproblems P_1, P_2, and P_3 from the running example.

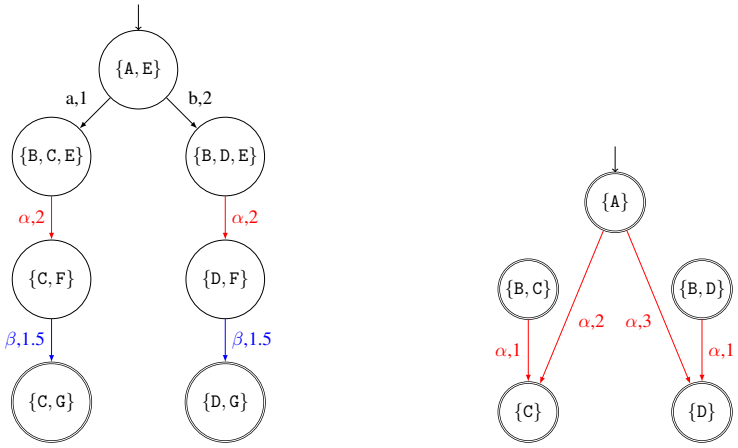

Figure 8. The product of $\mathcal{A}_1$ and $\mathcal{A}_2$ (left) and the projection of $\mathcal{A}_1$ on the alphabet $\{\alpha\}$ (right, unreachable states have been preserved).

Let u be a word from W'. Suppose that $u \notin W$. Hence, either $u_{|\Sigma_1} \notin W_1$ or $u_{|\Sigma_2} \notin W_2$. Thus, either there is no path π_1 in $\mathcal{A}_1$ such that $\sigma_1(\pi_1) = u_{|\Sigma_1}$ or there is no path π_2 in $\mathcal{A}_2$ such that $\sigma_2(\pi_2) = u_{|\Sigma_2}$. So, by definition of the product of weighted automata, there is either no path π in $\mathcal{A}_1 \| \mathcal{A}_2$ such that $\sigma(\pi)_{|\Sigma_1} = u_{|\Sigma_1}$ or no path π in $\mathcal{A}_1 \| \mathcal{A}_2$ such that $\sigma(\pi)_{|\Sigma_2} = u_{|\Sigma_2}$. Thus, no path π in $\mathcal{A}_1 \| \mathcal{A}_2$ such that $\sigma(\pi) = u$. Hence, $u \notin W'$. It proves that if $u \in W'$ and $u \notin W$, then $u \notin W'$. Hence, $W' \subseteq W$.

Finally, one has $W \subseteq W'$ and $W' \subseteq W$. Thus $W = W'$. The remaining is to prove that, for any $u \in W$, $w(u) = w'(u)$.

Let u be a word from W and suppose $w(u) < w'(u)$. It means that there is two paths, p_1 in $\mathcal{A}_1$ and p_2 in $\mathcal{A}_2$ such that $\sigma_1(p_1) = u_{|\Sigma_1}$, $\sigma_2(p_2) = u_{|\Sigma_2}$, and $c_1(p_1) + c_2(p_2) = w(u) < w'(u)$. In this case there is a path p in $\mathcal{A}_1 \| \mathcal{A}_2$ which is constructed from p_1 and p_2, thus $\sigma(p) = u$ and $c(p) \leq c_1(p_1) + c_2(p_2)$. By definition of $w'(u)$ one has $w'(u) \leq c(p)$. Thus $w'(u) \leq c_1(p_1) + c_2(p_2) < w'(u)$. It is impossible, and finally one has $w(u) \geq w'(u)$.

Let u be a word from W and suppose $w'(u) < w(u)$. It means that there is a path p in $\mathcal{A}_1 \| \mathcal{A}_2$ such that $\sigma(p) = u$ and $c(p) = w'(u) < w(u)$. Hence, there is a path p_1 in $\mathcal{A}_1$ and a path p_2 in $\mathcal{A}_2$, such that $\sigma_1(p_1) = u_{\Sigma_1}$, $\sigma_2(p_2) = u_{|\Sigma_2}$ and $c_1(p_1) + c_2(p_2) = c(p)$. Moreover, one has $c_1(p_1) + c_2(p_2) \geq w(u)$. Thus, $w(u) \leq c_1(p_1) + c_2(p_2) = c(p) < w(u)$. This is impossible, so one has $w'(u) \geq w(u)$.

To conclude: one has $w(u) \geq w'(u)$ and $w'(u) \geq w(u)$. Thus $w'(u) = w(u)$. □

In practice, one is only interested in the accessible part of the product, so the product operation may include some trimming. This makes it computable in a recursive manner, by breadth first search in both $\mathcal{A}_1$ and $\mathcal{A}_2$, starting from (s_1, s_2).

3.2. Projection of weighted automata

Just like the product, the projection of a (regular) weighted language can be directly performed on a weighted automaton that accepts this language. Let us introduce notation $s[\pi\rangle s'$ for a path π starting at state s and terminating at state s'. By convention, path π can be empty and then $s = s'$.

Definition 3.4. Let $\mathcal{A} = (S, T, \Sigma, s^0, F, c, f)$ be a weighted automaton, its projection $\mathcal{P}_{\Sigma'}(\mathcal{A})$ on alphabet Σ' is defined as $\mathcal{P}_{\Sigma'}(\mathcal{A}) = (S, T', \Sigma \cap \Sigma', s^0, F', c', f')$ with

$$\begin{aligned} T' &= \{(s, \alpha, s') \,:\, \exists \pi \in T^*, \exists t \in T, s[\pi t\rangle s', \sigma(t) = \alpha \in \Sigma',\, \sigma(\pi) \in (\Sigma \setminus \Sigma')^*\} \\ F' &= \{s \in S \,:\, \exists s' \in F,\, \exists \pi \in T^*,\, s[\pi\rangle s', \sigma(\pi) \in (\Sigma \setminus \Sigma')^*\} \end{aligned}$$

Regarding weights, for transition $t' = (s, \alpha, s') \in T'$ one has

$$c'(t') = \min_{\substack{\pi \in T^*, t \in T \,: \\ s[\pi t\rangle s',\, \sigma(t) = \alpha \\ \sigma(\pi) \in (\Sigma \setminus \Sigma')^*}} c(\pi t)$$

and for final state $s \in F'$

$$f'(s) = \min_{\substack{s' \in F, \pi \in T^* \,: \\ s[\pi\rangle s',\, \sigma(\pi) \in (\Sigma \setminus \Sigma')^*}} c(\pi) + f(s')$$

This definition actually encodes an epsilon-reduction (or epsilon-closure), where the epsilon (or silent) transitions are those labeled by $\Sigma \setminus \Sigma'$. Notice that the definition of projection does not include determinization nor minimization, as these operations are not always possible on weighted languages (see below).

Running example. Figure 8 (right) represents the projection of $\mathcal{A}_1$ (Figure 7) on the alphabet it shares with $\mathcal{A}_2$ (Figure 7), that is $\{\alpha\}$. Unreachable states have been preserved.

Lemma 3.5. Let $\mathcal{A}$ be a weighted automaton and let Σ' an alphabet. Then $\mathcal{L}(\mathcal{P}_{\Sigma'}(\mathcal{A})) = \mathcal{P}_{\Sigma'}(\mathcal{L}(\mathcal{A}))$

So the projection of weighted languages can be "implemented" as a projection on the automaton that accepts this language.

Proof:
Note $\mathcal{P}_{\Sigma'}(\mathcal{L}(\mathcal{A})) = (W, \Sigma \cap \Sigma', w)$ and $\mathcal{L}(\mathcal{P}_{\Sigma'}(\mathcal{A})) = (W', \Sigma \cap \Sigma', w')$. Note $\mathcal{L}(\mathcal{A}) = (W_a, \Sigma_a, w'_a)$. First show that $W = W'$.

Consider a word $u \in W$. Suppose $u \notin W'$. Then, there is no path p in $\mathcal{A}$ such that $\sigma(p) = u'$ and $u'_{|\Sigma'} = u$. Thus, there is no word $u' \in W_a$ such that $u'_{|\Sigma'} = u$. Hence, $u \notin W$. It proves that $u \in W$ and $u \notin W'$ implies $u \notin W$, which is impossible. Finally, one has $W \subseteq W'$.

Consider a word $u \in W'$. Suppose $u \notin W$. Then, there is no word $u' \in W_a$ such that $u'_{|\Sigma'} = u$. Thus, there is no path p in $\mathcal{A}$ such that $\sigma(p)_{|\Sigma'} = u$. Hence, $u \notin W'$. It proves that $u \in W'$ and $u \notin W$ implies $u \notin W'$, which is impossible. Finally one has $W' \subseteq W$.

It has been proved that $W \subseteq W'$ and $W' \subseteq W$, thus one has $W = W'$. The remaining is to prove that $w = w'$.

Consider a word $u \in W$. There is no u' in W_a such that $u'_{|\Sigma'} = u$ and $w_a(u) < w(u)$. Thus, there is no path p in $\mathcal{A}$ such that $\sigma(p) = u'$ with $u'_{\Sigma'} = u$ and c(p)¡w(u). Hence, there is no path p' in $\mathcal{P}_{\Sigma'}(\mathcal{A}) = (S', T', \Sigma', (s^0)', F', c', f')$ such that $\sigma(p') = u$ and $c'(p') < w(u)$. So, $w'(u) \geq w(u)$.

In the same way one can prove that $w(u) \geq w'(u)$, and finally one has $w(u) = w'(u)$. □

This projection operation can be effectively computed, using Mohri's epsilon-reduction algorithm for weighted automata [35].

3.3. Smaller automata

Factored planning aims at solving possibly large planning problems by a divide and conquer strategy, or a modular approach. So the computations attached to each module or component should remain tractable. In the setting of an MPA computing on weighted automata rather than on their languages, it is thus important to keep the size of these automata under control. A first strategy is of course to trim these automata, that is to remove states and transitions that are not accessible (not reachable from the initial state) or coaccessible (can not reach a final state). A trimming algorithm has been presented by Mohri [35], and works for any weighted automaton. Beyond trimming, one may also consider minimization. Again, Mohri shows [35] how a standard minimization procedure for ordinary deterministic automata (as presented by Sakarovitch in [39] for example) can be extended to the case of deterministic weighted automata. Of course, this minimization preserves the weighted language.

By construction, the weighted automata representing planning problems are deterministic – i.e. there is no state s such that $\exists \alpha, \exists s' \neq s'', (s, \alpha, s') \in T$ and $(s, \alpha, s'') \in T$. Clearly, the product of weighted automata preserves determinism. However, projections do not preserve determinism (see Figure 8 (right) for an example). As a consequence, if one wishes to work with minimal automata, determinization techniques are required. There exists a determinizationn algorithm for weighted automata [35], which derives from the classical subset construction for standard automata [39].

Let us denote by $Det(\mathcal{A})$ the determinized version of weighted automaton $\mathcal{A} = (S, T, \Sigma, s^0, F, c, f)$. The states of $Det(\mathcal{A})$ are of the form (A, λ), where $A \subseteq S$ and $\lambda : A \to \mathbb{R}_+$ associates a residual cost to each "inner" state s of A. The initial state of $Det(\mathcal{A})$ is $(\{s^0\}, \lambda^0)$, such that $\lambda^0(s^0) = 0$. The other states are recursively derived from this initial state. Let $q = (A, \lambda)$ be a state of $Det(\mathcal{A})$ and $\alpha \in \Sigma$ such that the new state $q' = (A', \lambda')$ obtained by firing $\alpha \in \Sigma$ is such that $A' = \{s \in S \mid \exists s' \in A, (s', \alpha, s) \in T\}$ and, for $s' \in A'$,

$$\lambda'(s') = \overline{\lambda'}(s') - \left(\min_{s'' \in A'} \overline{\lambda'}(s'') \right),$$

where $\overline{\lambda'}(s')$ is defined as:

$$\overline{\lambda'}(s') = \min_{s \in A, t=(s,\alpha,s') \in T} \lambda(s) + c(t).$$

The cost of the transition (q, α, q') in $Det(\mathcal{A})$ is then $min_{s' \in A'} \overline{\lambda'}(s')$. The cost of a final state (A, λ) is $\min_{s \in A \cap F} \lambda(s) + f(s)$. This definition of $Det(\mathcal{A})$ directly gives an algorithm for the determinization of weighted automata, which has been fully described by Mohri [35].

As an example, one can consider the deterministic automaton of Figure 9, which is the deterministic version of the automaton of Figure 8 (right). Notice that, after determinization, the link between the states of the automaton and the atoms of the planning problem is partially lost.

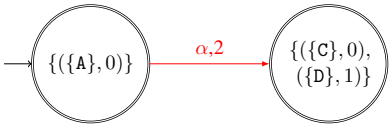

Figure 9. A deterministic weighted automaton with the same weighted language as the weighted automaton of Figure 8 (right).

Unfortunately, not all weighted automata have an equivalent finite deterministic weighted automaton. And, as a consequence, not all weighted automata are determinizable using the above algorithm (notice however that the fact that a weighted automaton is not determinizable by the above algorithm does not imply that this automaton admit no equivalent finite deterministic weighted automaton, this has been stated for example by Allauzen and Mohri [1]). See for example Figure 10. In this automaton, the cost of a word u is $\min(|u|_a, |u|_b)$. A deterministic weighted automaton recognizing the same language should be able to "count" the number of a and the number of b in a word, and, thus, could not be finite.

There exists a sufficient condition for the determinizability of weighted automata [35]. This condition is based on the *twin property*: two states s and s' are twins if and only if 1) they cannot be reached by the same label sequence from the initial state or 2) for all $u \in \Sigma^*$ such that u allows to loop on both s and s' the cost of this loop is the same for s and s'. An automaton has the twin property if and only if any s, s' in it are twins. Twin automata are determinizable. This condition is only sufficient in general,

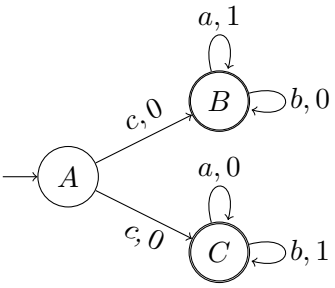

Figure 10. a weighted automaton that can not be determinized

but is also necessary for special classes of weighted automata [8, 31] which are well characterized [32]. For the general class of weighted automata there is – to our knowledge – currently no known necessary and sufficient condition to decide determinizability.

During our tests it appeared that the determinization and the minimization of weighted automata, when possible, often gave better performance for our planning system. This is due to the fact that taking the product of two deterministic automata is generally more efficient than taking the product of two non-deterministic automata. In particular, $\mathcal{A}\|\mathcal{A} = \mathcal{A}$ when $\mathcal{A}$ is deterministic, while $\mathcal{A}\|\mathcal{A}$ is a larger automaton than $\mathcal{A}$ when $\mathcal{A}$ is non-deterministic.

Moreover, it is possible to avoid the problem of the non-determinizability of weighted automata by performing a partial determinization: a determinization procedure that stops after some bound is reached and provides an automaton which is deterministic "at the beginning". Such a procedure allows one to recognize small words with the deterministic part of the automaton, and use the non-deterministic part for larger words only. The next section describes this procedure. Before that, an example of an execution of the message passing algorithm on weighted automata is given.

3.4. Using MPA for weighted automata on the running example

The communication graph of $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$ is, as in the case of languages, a tree, depicted in Figure 11.

$$\mathcal{A}_1 \overset{\{\alpha\}}{\text{———}} \mathcal{A}_2 \overset{\{\beta\}}{\text{———}} \mathcal{A}_3$$

Figure 11. Communication graph of $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$.

As the communication graphs are similar in both cases (which is normal as they correspond to the same planning problem, using the same decomposition) we will compute messages in the same order as in the case of languages.

One first computes $\mathcal{M}_{1,2}$ as $\mathcal{P}_{\Sigma_1\cap\Sigma_2}(\mathcal{A}_1)$ (this weighted automaton is represented in Figure 9).

Then, one computes $\mathcal{M}_{2,3}$. For that $\mathcal{M}_{1,2}\|\mathcal{A}_2$ is first computed (Figure 12, left). And then one gets $\mathcal{M}_{2,3}$ as $\mathcal{P}_{\Sigma_2\cap\Sigma_3}(\mathcal{M}_{1,2}\|\mathcal{A}_2)$ (Figure 12, right).

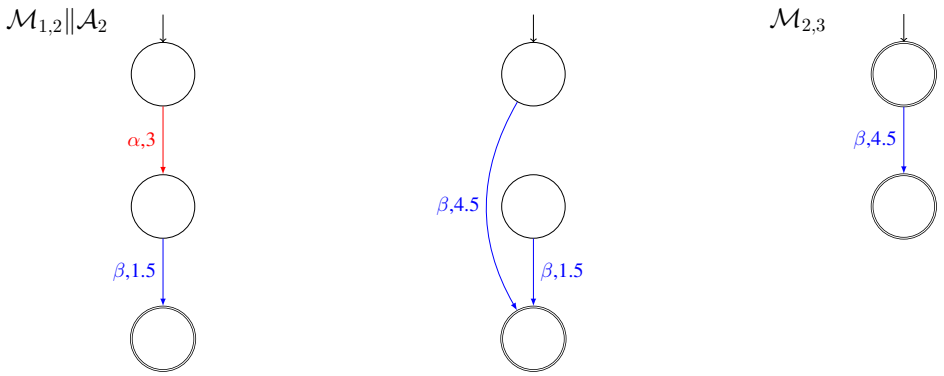

Figure 12. Steps for the computation of $\mathcal{M}_{2,3}$ in the running example: $\mathcal{M}_{1,2}\|\mathcal{A}_2$ (left), and $\mathcal{P}_{\Sigma_2\cap\Sigma_3}(\mathcal{M}_{1,2}\|\mathcal{A}_2)$ before (center) and after (right) minimization.

Similarly, from right to left, one first computes $\mathcal{M}_{3,2}$ as $\mathcal{P}_{\Sigma_3\cap\Sigma_2}(\mathcal{A}_3)$ (Figure 13, left). Then one computes $\mathcal{M}_{3,2}\|\mathcal{A}_2$ (Figure 13, center), which is then used to compute $\mathcal{M}_{2,1} = \mathcal{P}_{\Sigma_1\cap\Sigma_2}(\mathcal{M}_{3,2}\|\mathcal{A}_2)$ (Figure 13, right).

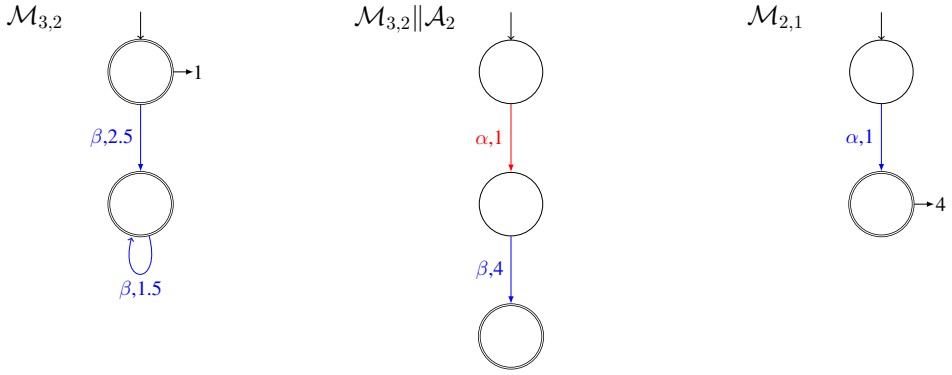

Figure 13. Steps for the computation of $\mathcal{M}_{2,1}$ in the running example: $\mathcal{M}_{3,2}$ (left), $\mathcal{M}_{3,2}\|\mathcal{A}_2$ (center), and $\mathcal{P}_{\Sigma_2\cap\Sigma_1}(\mathcal{M}_{3,2}\|\mathcal{A}_2)$ (right). Outgoing arrows at final states represent the cost of ending at these states.

One can then get the reduced automata $\mathcal{A}'_1 = \mathcal{A}_1\|\mathcal{M}_{2,1}$ (Figure 14, left), $\mathcal{A}'_2 = \mathcal{A}_2\|\mathcal{M}_{1,2}\|\mathcal{M}_{3,2}$ (Figure 14, center), and $\mathcal{A}'_3 = \mathcal{A}_3\|\mathcal{M}_{2,3}$ (Figure 14, right). Checking that the languages of theses automata correspond to the reduced languages computed in Section 2.3 (i.e $la(\mathcal{A}'_1) = \mathcal{L}'_1$, $la(\mathcal{A}'_2) = \mathcal{L}'_2$, and $la(\mathcal{A}'_3) = \mathcal{L}'_3$) concludes our running example.

4. Partial determinization

As explained before, the determinization is not always possible for weighted automata (and it is not even known how to decide the determinizability of weighted automata in the general case). That is why, in this section, is suggested a method to partially determinize weighted automata. The principle is to determinize up to some depth and complete the determinized automaton obtained with a non-deterministic

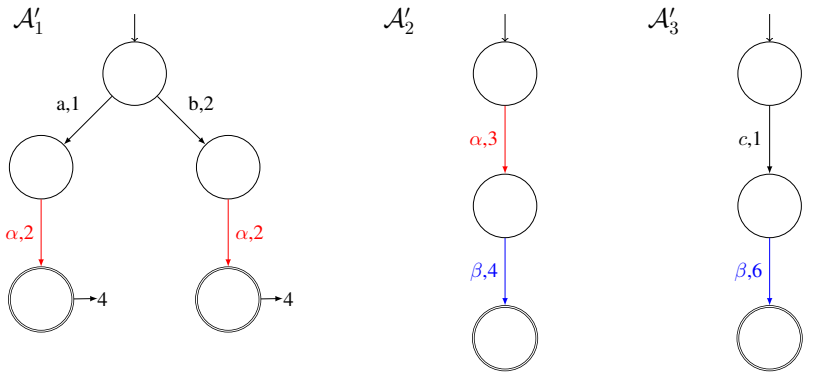

Figure 14. Reduced automata for the running example. Outgoing arrows at final states represent the cost of ending at these states.

part. In fact, a related idea has been mentioned by Mohri in [34], named *local determinization*, where determinization is done only at some particular states of an automaton.

4.1. Basics of the method

In the determinized version of a weighted automaton it is possible to associate a depth to each state. The initial state has depth 0. Each successor of the initial state has depth 1. Their successors have depth 2, and so on. Equivalently, the depth of a state is the minimal depth among its predecessors, plus 1. For example, Figure 15(a) represents the (beginning of the) determinization of the automaton of Figure 10. The depths of the states are represented by dashed lines.

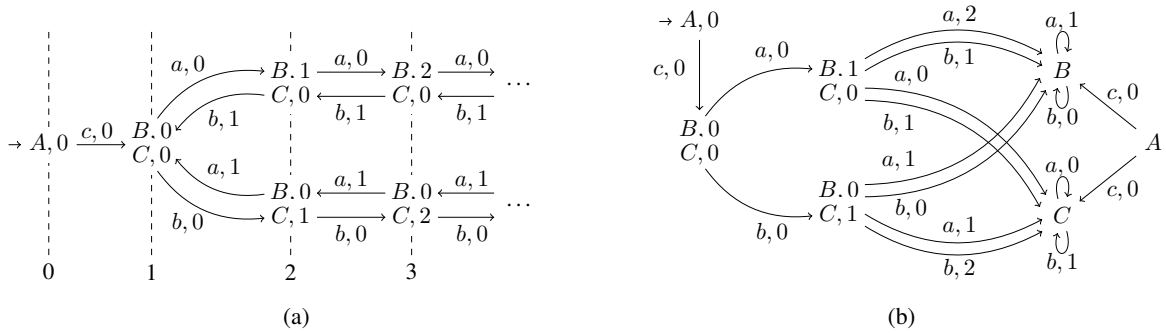

Figure 15. (a) first steps of the (infinite) determinization of Figure 10. Notice that all states – excepted (A,0) – are final states ; (b) partial determinization at depth 2 of Figure 10. Notice that all states – excepted (A,0) and A – are final states. State A may be trimmed.

The partial determinization consists in stopping the determinizastion at some depth and branching the states at this depth to the original (non deterministic) automaton, without initial states. More formally, in the partial determinization at depth n of a weighted automaton $\mathcal{A}$, for each state (A, λ) at depth n, all the transitions (A, α, s) such that $\exists s' \in A, (s', \alpha, s) \in T$ are added. The costs of these transitions are such that $c'((A, \alpha, s)) = \min_{s' \in A, (s', \alpha, s) \in T} \lambda(s') + c((s', \alpha, s))$. This clearly preserves the languages.

For example, for the automaton of Figure 10, the partial determinization at depth 2 is represented in Figure 15(b).

One may have noticed that, even if some states can be removed (like A in the example of Figure 15(b)), the partial determinization of a non-determinizable weighted automaton is usually bigger than the original automaton. However, using deterministic automata reduces the complexity of taking the synchronous product, so the partial determinization can reduce the complexity of an execution of our algorithm for planning.

4.2. Smaller representation

In the same spirit as in the previous section one wants to have data structures as small as possible. To do that we can take inspiration from [41]. In this paper, a method for the determinization of probabilistic automata was suggested. For the determinization of a probabilistic automaton $\mathcal{A}$, this method consists in a standard determinization of the non-probabilistic automaton $\mathcal{A}'$ support of $\mathcal{A}$, on which some new costs are attached afterwards. However, these new costs are not probabilities but matrices, changing the power of the model.

Applying some of the principles of this method to the framework of this paper, it is possible to complete a partially determinized automaton by transitions with matrices of weights instead of adding transitions leading to the original automaton. Formally, the principle is, as before, to stop the determinization procedure at a given depth, and, from that, to add some transitions to the remaining states. If (A, λ) is a state at the stopping depth, one as to add a transition $((A, \lambda), \alpha, (A', \lambda'))$ for each α such that there is $a \in A$ such that $\exists (a, \alpha, a') \in T$. The set A' contains all states a' such that there is $(a, \alpha, a') \in T$ and $a \in A$. For all $a' \in A'$, $\lambda'(a') = 0$. The "cost" M of the new transition is a matrix with $|A|$ rows and $|A'|$ columns. One has $M_{i,j} = \lambda(a_i) + c(a_i, \alpha, a'_j)$ where a_i is the i^{th} element of A and a'_j is the j^{th} element of A'. In fact, when there already exists a state (A', λ'') in the determinized automaton it is possible to plug the new transition directly to this state (or one of these states). In this case one has $M_{i,j} = \lambda(a_i) + c(a_i, \alpha, a'_j) - \lambda''(a'_j)$. Notice that, if some states (A', λ'') exist, there exists always one such that, $\forall i, j, M_{i,j} \geq 0$.

In such an automaton, the cost of a word is not obtained directly: if some transitions have a cost matrix some computations are needed to find the best possible cost for this word. For example, in the determinized automaton presented in Figure 16, the words of length greater than 2 involve computations.

5. Structured states

In factored planning problems, the sets of states – and in particular the initial and the final states – are products of the sets of states of the sub-problems. However, it is possible to extend the algorithms presented in this paper in order to solve more general problems, not expressible in the planning formalism. In this section are presented such problems, were the set of final states is not exactly the product of all the sets of final states but a subset of it, defined by a relation on the final states. The results of this section may be extended to the initial states (or even to any type of states). For matter of clarity only the case of final states is presented here.

Given two automata, we want to be able to express that it is not sufficient that s_1 and s_2 are final for a state (s_1, s_2) to be final. It is also needed that s_1 and s_2 are *in relation*. To express this notion of

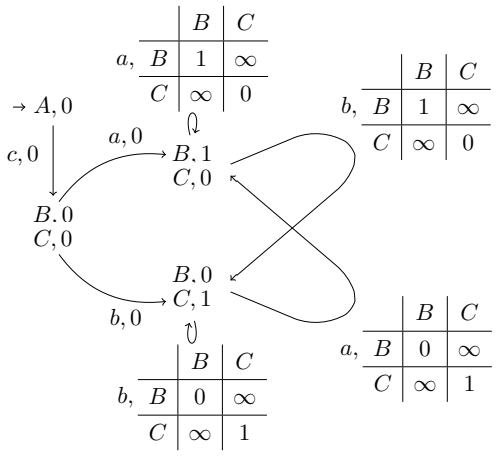

Figure 16. partial determinization at depth 2 of Figure 10: matrices representation.

being in relation one can add to each final state of an automaton a set of labels. Two final states are then in relation if they share some of their labels. Formally, a weighted automaton is, in this case, a tuple $\mathcal{A} = (S, T, \Sigma, s^0, F, c, f, \Lambda, \ell)$, where $(S, T, \Sigma, s^0, F, c, f)$ is a weighted automaton as defined above, Λ is a set of labels, and $\ell : F \to 2^\Lambda$ associates a set of these labels to each final state. The remaining is to define the product of weighted automata in this framework.

Definition 5.1. The product $\mathcal{A}_1 \| \mathcal{A}_2 = (S, T, \Sigma, s^0, F, c, f, \Lambda, \ell)$ of two weighted automata with labels on the final states $\mathcal{A}_1 = (S_1, T_1, \Sigma_1, s_1^0, F_1, c_1, f_1, \Lambda_1, \ell_1)$ and $\mathcal{A}_2 = (S_2, T_2, \Sigma_2, s_2^0, F_2, c_2, f_2, \Lambda_2, \ell_2)$ is such that: $(S, T, \Sigma, s^0, F', c, f)$ is the product of the weighted automata $(S_1, T_1, \Sigma_1, s_1^0, F_1, c_1, f_1)$ and $(S_2, T_2, \Sigma_2, s_2^0, F_2, c_2, f_2)$ without labels on the final states. Moreover, $F = \{(f_1, f_2) \in F' \mid \ell_1(f_1) \cap \ell_2(f_2) \neq \emptyset\}$, $\Lambda = \Lambda_1 \cup \Lambda_2$, and $\ell((f_1, f_2)) = \ell_1(f_1) \cap \ell_2(f_2) \cup (\ell_1(f_1) \setminus \Lambda_2) \cup (\ell_2(f_2) \setminus \Lambda_1)$.

It is a bit tricky to define the projection of these weighted automata with labels on the final states. However, there exists a simple translation from weighted automata with labels on the final states to weighted automata. The idea is, from an automaton $\mathcal{A} = (S, T, \Sigma, s^0, F, c, f, \Lambda, \ell)$ with label on the final states, to construct a standard weighted automaton $\mathcal{A}' = (S', T', \Sigma', s^{0'}, F', c', f')$ such that: $S' = S \cup \{s_F\}$, $T' = T \cup \{(s, \alpha, s_F) \mid s \in F, \alpha \in \ell(s)\}$, $\Sigma' = \Sigma \dot\cup \Lambda$, $s^{0'} = s^0$, $F' = \{s_F\}$, $\forall t \in T, c'(t) = c(t)$, $\forall t = (s, \alpha, s_F) \in T' \setminus T, c'(t) = f(s)$, $f'(s_F) = 0$. In fact, this corresponds to an encoding of the labels of the final states as shared transitions.

Lemma 5.2. A path $\pi = (s_1, \alpha_1, s_2) \dots (s_{n-1}, \alpha_{n-1}, s_n)$ is accepted in a weighted automaton $\mathcal{A}$ with labels on the final states if and only if $\exists \alpha \in \ell(s_n)$ $(s_1, \alpha_1, s_2) \dots (s_{n-1}, \alpha_{n-1}, s_n)(s_n, \alpha, s_F)$ is accepted in the corresponding weighted automaton $\mathcal{A}'$ (with the convention that $\ell(s) = \emptyset$ if $s \notin F$).

Proof:
This lemma comes directly from the construction of $\mathcal{A}'$. □

From this lemma and the definition of the product of weighted automata with labels on the final states, one gets – for any two weighted automata $\mathcal{A}_1$ and $\mathcal{A}_2$ with labels on the final states – that $(\mathcal{A}_1 \| \mathcal{A}_2)' = \mathcal{A}'_1 \| \mathcal{A}'_2$. Thus, a system represented as a network of weighted automata with labels on the final states (i.e. a systems such that the final states are not expressible as products of the final states of its components) can be represented as a network of standard weighted automata. This allows one to solve such problems using the exact algorithms presented above.

6. Experimental results

As for any planning algorithm the theoretical complexity of the message passing algorithm for planning is huge. However, we wanted to see if this algorithm is efficient in practice. Thus, we implemented it and tested this implementation on some standard benchmarks from planning community.

6.1. Implementation

We implemented the message passing algorithm for weighted automata in a planner called Distoplan. Our implementation is based on the openFst library[2] for most of the operations on weighted automata (ε-reduction, trimming, minimization, determinization). Our planner accepts as input planning problems given directly in terms of weighted automata (using the format specified in openFst) or as PDDL (Planning Domain Definition Language) files [15] (which are a standard representation of planning problems in the planning community). The parsing of the PDDL files is done using the parser from HSP* planner [17].

Notice that our implementation only deals with factored planning problem. It cannot automatically find a decomposition of a given planning problem. In other words, it is unable to find a decomposition of the set of atoms that ensures that the communication graphs are trees. This is due to the fact that, currently, it is not known what is an efficient decomposition. One could however remark that a tree decomposition of a problem so that, when the problem grows, the number of components grows but the size of these components remain small will, in general, certainly be better suited to our approach than a decomposition so that the size of the components grows with the size of the problem (this is due to the fact that compositions of large components – and search in them – is the main source of complexity for our approach). This has been our main guideline for selecting benchmarks: they had to have a good decomposition in the sense of this remark (notice that a problem where such a good decomposition exists but our approach is still not scalling well has been analysed in details in a previous paper [14]). Methods to decompose problems exists (such as in [10] or similarly in [2]) but they only provide a communication graph which is a tree, not necessarily the best one for factored planning, or even a good one (as, again, it is not known what exactly is a good decomposition).

6.2. Rooms and robot

The first test we performed was on a problem presented in [2], called rooms and robot. We wanted to try this problem because [2] presents the only implemented factored planner to our knowledge and uses rooms and robot as a benchmark for testing it. Thus, it was of interest to note if the performances of our

[2] http://www.openfst.org/

planner were comparable to the performances of this one (notice that our planner performs cost-optimal planning, which is not the case of the one from [2]).

The rooms and robot problem is the following: a robot has to move in different rooms in order to close and lock one window per room. More precisely, the rooms are organized into a circle and the robot can only move in one step from the room where it is to an adjacent room on the circle. Inside a room the robot can close the window, and lock it if it is closed. The goal is to lock all windows (in our case with a minimal number of actions).

A natural decomposition of this problem considers each room as a component. However, this decomposition is such that the interaction graph contains a cycle and has no redundant edges. In order to obtain a tree shaped communication graph, and thus to permit the use of the message passing algorithms on this problem, we had to propose another decomposition. For this purpose, we added an atom for each room, giving the position of the robot. We accordingly modified the moving actions by adding these new atoms to the preconditions and the effects. The window closing and window locking actions remained as before. We consider the same components as in the natural decomposition of the problem, with the addition of a new one containing the new atoms. The corresponding interaction graph still has cycles. However, some edges are redundant and the only communication graph is a tree.

The results obtained by Distoplan on this problem are given in Figure 17. The leftmost plot presents the execution time of three versions of Distoplan on small instances of rooms and robot (5 to 11 rooms). The upper curve is obtained using no size reduction technique. The middle curve is obtained when the automata are trimmed after each product and each projection. The lower curve is obtained when the automata are minimized after each each product and projection (they are thus also determinized after each projection). The rightmost plot presents the execution time of Distoplan on larger instances of rooms and robot (up to 50 rooms). It has been obtained with the version of Distoplan where automata are minimized after each iteration.

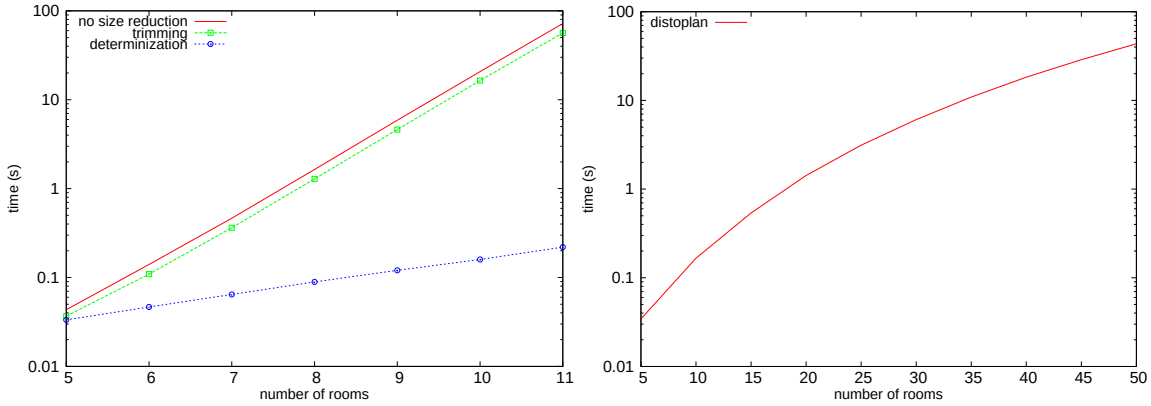

Figure 17. results obtained by distoplan on rooms and robots, the time is in logscale.

The first conclusion we can draw from these results is that, on this particular problem, using minimal deterministic automata strongly improves the time efficiency of Distoplan. This significant efficiency gap shows that the use of deterministic automata - when possible - can really be of interest and may allow a better scalability of our planner.

A second conclusion is that, as expected, the execution time of Distoplan scales well with the size of this problem. We obtain, in fact, a problem solving time subexponential in the number of rooms. This is comparable with what is presented in [2]. It is possible, thought, that a better efficiency can be achieved using another decomposition of the problem. Indeed our decomposition has the drawback that the size of one of the components grows with the number of rooms. Finding a decomposition which avoids this phenomenon may result in better performances. Such a decomposition is proposed in [2], but we were not able to adapt it to our setting (as our decompositions are defined by subsets of the atoms while in [2] the decompositions are defined by subsets of the actions).

6.3. IPC Benchmarks

We then tested Distoplan on problems from international planning competitions (IPC). Among the problems we considered we found two that we were able to decompose well. That is, for which we found a decomposition so that only the number of components is increased (their size remaining the same) with growing instances of the problem. These two problems come from the Promela domains of the fourth international planning competition [24]. These domains regroup problems translated to PDDL from the Promela language. The first problem corresponds to the classical "dinning philosophers" problem, while the second problem is based on a communication protocol for a network of optical telegraphs. In each problem the objective is to find a deadlock. We also considered other versions of these problems where no deadlock exists. In this case one has to detect the absence of deadlock.

For each problem we proposed a decomposition ensuring the communication graphs to be trees. Using these decompositions we ran Distoplan on instances of growing size for each problem and compared the run times obtained with the performances of other up-to-date planners. For comparison we used a planner based on Fast-Downward [20] (i.e. an A* based search) with the landmark cut heuristic [21] and the IPC-5 version of SATPLAN [30]. Notice that SATPLAN would not take part in the same track as Distoplan in a planning competition. Indeed it does not ensure cost-optimality of the plans found.

Dinning philosophers Some philosophers want to eat. They sit all around a table, with one fork between any two philosophers. To eat a philosopher needs two forks: not all philosophers can eat at the same time. He has to take a first fork, then a second one. When he has finished eating a philosopher releases the forks he used. In this setting one has to find deadlocks: situations were no philosopher can eat and no fork is free.

More precisely, philosophers and forks form an alternating cycle. Each philosopher can perform the following actions: 1) take left fork if free, 2) take right fork if free, 3) release right fork if taken, and 4) release left fork if taken. These actions can be performed only in this order. Hence, a simple deadlock occurs when each philosopher has taken his left fork: no fork can be released and no more fork can be taken.

When looking at this problem as a factored problem, an intuitive approach is to consider the set of atoms p_i defining each philosopher as a component and the set of atoms f_i defining each fork as a component. However, in this case the communication graph obtained is not a tree: it is a cycle, as represented in Figure 18 (left) for four philosophers. It is possible to come up with a tree shaped communication graph by defining components as the union of the atoms defining each philosopher with the atoms defining the opposite fork on the circle. Figure 18 (right) represents the interaction graph obtained for four philosophers. It is a line, and thus a tree.

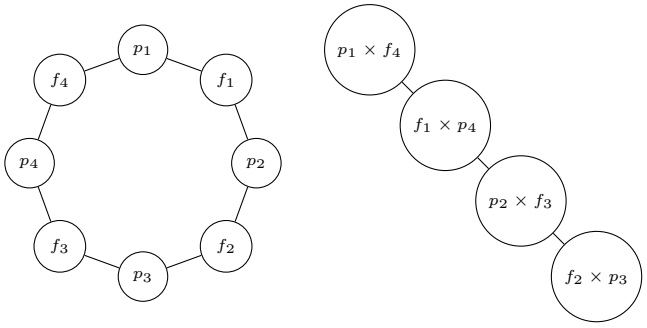

Figure 18. philosophers problem: philosophers – p_i – and forks – f_i – as components (left), merging component to get communication graph which is a tree (right)

Results obtained with Distoplan on instances of the dinning philosophers problem of growing size are presented in Figure 19. The same figure also gives results obtained with Fast Downward and SATPLAN. The left plot presents results obtained in the exact case presented above, when a deadlock exists. The right plot presents results obtained with slightly modified problems where there is no deadlock (this is achieved by allowing one of the philosophers to take right fork first).

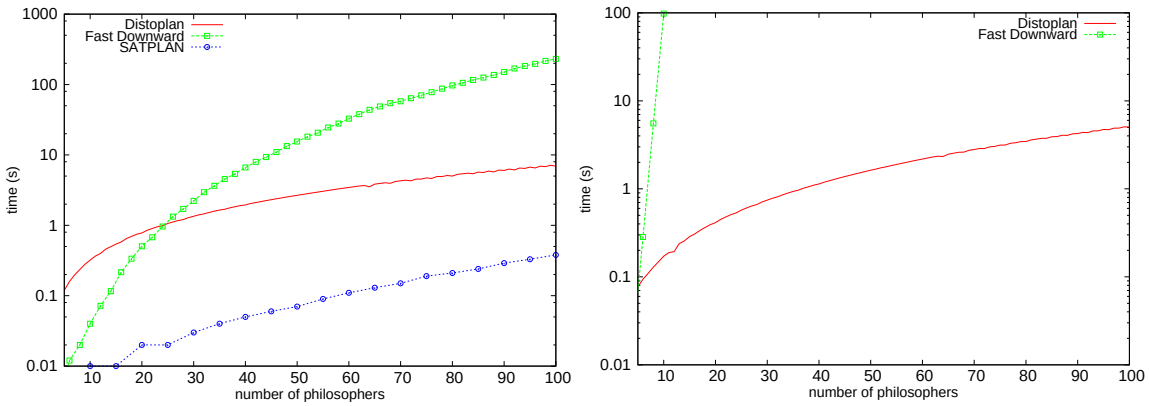

Figure 19. Performances of Distoplan, Fast Downward, and SATPLAN on philosophers problems with (letf) and without (right) deadlocks. Time is logscale.

One can notice that Distoplan runs in sub-exponential time in the number of philosophers (that is in the number of components) in both cases. In fact, Distoplan is not affected by the presence or the absence of solution. We used the version of Distoplan which minimizes automata after each operations (as for rooms and robot, the version with no size reducing technique and the version using trimming were much less efficient). With regards to other planners we remark that SATPLAN is – as expected – very efficient when there is solutions. Remark however that it does not guarantee cost-optimality of solutions. When there is no solution SATPLAN is not able to detect it, that's why it is not plotted in the results. Fast Downward works well in presence of deadlocks, but scales less efficiently than Distoplan. When there is

no solutions it has to explore the full state space of the problem to detect it and thus runs in exponential time in the number of philosophers.

Optical telegraph This second benchmark is the following: some telegraph stations – organized as a circle – have to communicate, following a precise protocol. As for philosophers, the goal is to find potential deadlocks in the communication protocol. In fact, this problem can be seen as philosophers with more private actions. The decomposition we proposed for this problem is based on the same principles as for philosophers. Each natural component of the circle (here telegraph stations and their communication channels) is merged with its opposite on the circle.

The results we obtained are presented in Figure 20. The left plot has been obtained for problems with deadlocks. The right plot has been obtained for problems without deadlocks (telegraph stations are organized as a line rather than as a circle).

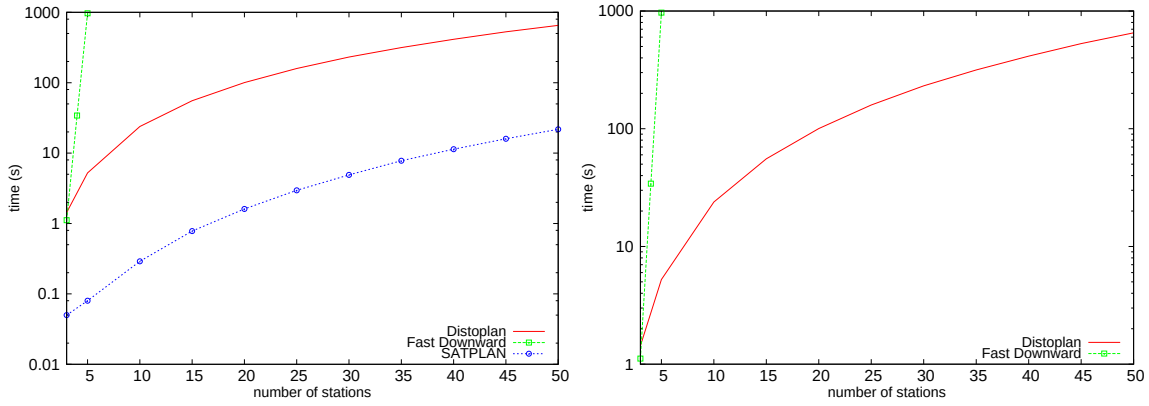

Figure 20. Performances of Distoplan, Fast Downward, and SATPLAN on optical telegraph problems with (left) and without (right) deadlocks. Time is logscale.

For Distoplan and SATPLAN these problems are not that different from the dinning philosophers problems. The results obtained are thus similar to the previous ones: a computation time sub-exponential in the size of the problems. The addition of many private actions in each component (compared to the dinning philosophers case) makes the state space of this problem much larger. This explains the lower efficiency of Fast Downward. Notice that, on these problems, most of the computation time of Distoplan (90%) is spent building the initial automata from the PDDL representation of the problems.

7. Conclusions

The solution to cost optimal factored planning presented here is based on two main ingredients: 1/ distributed constraint satisfaction algorithms, encoded as message passing algorithms and extended to perform as well cost optimization, and 2/ weighted automata calculus. An essential difference with previous approaches is that we handle all plans solving the problem, under a suitable factorized (thus compact) form. This is crucial to guarantee the optimality of the best plan(s) found, a feature that was missing to previous factored planning approaches. Of course, as for all factored planning methods, the theoretical

worst case complexity is huge. However, the "practical complexity" seems much smaller as evidenced by our experiments on classical planning problems.

Several extensions to this work have already been explored and partially published. A first technical extension concerns the fact that some resources may be necessary to enable some action, but may not be consumed by this action. This must not be encoded as a read/write operation on this non-modified resource since it enforces the ordering of accesses to this resource, whereas one would rather like to preserve the possibility of concurrent (i.e. simultaneous) accesses. A solution to this was proposed under the form of weighted automata with read arcs, which mimic the read arcs proposed for Petri nets [25]. As a general rule, one should take advantage of the concurrency of actions as much as possible in order to reduce the set of possible plans to consider. Another step in this direction was proposed in [28], where components are handled as (weighted) Petri nets, instead of (weighted) automata, which allows to exploit as well the internal concurrency of events within each individual component. As a third extension of our approach, we have examined approximate planning methods in order to deal with large networks of components that do not have a tree structure. Other communities (e.g. error correcting codes) have evidences that message passing algorithms could still perform well even in the presence of cycles. This gave birth to turbo-codes, so by analogy we have experimented turbo-planning [27]. Surprisingly, and despite any theoretical proof of convergence, MPA still perform well on large and cyclic graphs of components, drawn at random. Most of the time a feasible plan is found, and very often it is optimal or close to optimal (when the comparison can be performed, that is when the problem is still manageable by classical optimal planning methods). Finally, the reader may be bothered by our top-down approach, that progressively reduces the sets of possible local plans, and he may be more confident in bottom-up approaches that *build* one plan (possibly the optimal one). So let us mention another line of work inspired by the present paper, which recasts the celebrated A* algorithm into a fully distributed message passing version [26].

Acknowledgment

We would like to thank Patrik Haslum for lending us his PDDL parser, and for his help in selecting meaningful benchmarks for our experiments.

References

[1] Allauzen, C., Mohri, M.: Efficient Algorithms for Testing the Twins Property, *Journal of Automata, Languages and Combinatorics*, **8**(2), 2003, 117–144.

[2] Amir, E., Engelhardt, B.: Factored Planning, *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, 2003.

[3] Blum, A., Furst, M.: Fast Planning Through Planning Graph Analysis, *Artificial Intelligence*, **90**(1-2), 1995, 281–300.

[4] Bonet, B., Geffner, H.: Planning as Heuristic Search, *Artificial Intelligence*, **129**(1-2), 2001, 5–33.

[5] Bonet, B., Haslum, P., Hickmott, S., Thiébaux, S.: Directed Unfolding of Petri Nets, *Transactions on Petri Nets and other Models of Concurrency*, **1**(1), 2008, 172–198.

[6] Brafman, R., Domshlak, C.: Factored Planning: How, When, and When Not, *Proceedings of the 21st AAAI Conference on Artificial Intelligence*, 2006.

[7] Brafman, R., Domshlak, C.: From One to Many: Planning for Loosely Coupled Multi-Agent Systems, *Proceedings of the 18th International Conference on Automated Planning and Scheduling*, 2008.

[8] Buchsbaum, A., Giancarlo, R., Westbrook, J.: On the Determinization of Weighted Finite Automata, *SIAM Journal on Computing*, **30**(5), 2000, 1502–1531.

[9] Cassandras, C., Lafortune, S.: *Introduction to Discrete Event Systems*, Kluwer Academic, 1999.

[10] Dechter, R.: *Constraint Processing*, Morgan Kaufmann, 2003.

[11] Edelkamp, S.: Planning with Pattern Databases, *Proceedings of the 12th International Conference on Automated Planning and Scheduling*, 2001.

[12] Fabre, E.: *Bayesian Networks of Dynamic Systems*, Habilitation à diriger des recherches, Université de Rennes1, 2007.

[13] Fabre, E., Jezequel, L.: Distributed Optimal Planning: an Approach by Weighted Automata Calculus, *Proceedings of the 48th IEEE Conference on Decision and Control*, 2009.

[14] Fabre, E., Jezequel, L., Haslum, P., Thiébaux, S.: Cost-Optimal Factored Planning: Promises and Pitfalls, *Proceedings of the 20th International Conference on Automated Planning and Scheduling*, 2010.

[15] Ghallab, M., Isi, C., Penberthy, S., Smith, D., Sun, Y., Weld, D.: *PDDL - The Planning Domain Definition Language*, Technical report, Yale Center for Computational Vision and Control, 1998.

[16] Hart, P., Nilsson, N., Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths, *IEEE Transactions on Systems Science and Cybernetics*, **4**(2), 1968, 100–107.

[17] Haslum, P.: Tp4'04 and HSP*-a, *4th International Planning Competition Booklet*, 2004.

[18] Haslum, P., Geffner, H.: Admissible Heuristics for Optimal Planning, *Proceedings of the 5th International Conference on Automated Planning and Scheduling*, 2000.

[19] Haslum, P., Helmert, M., Hoffmann, J.: Explicit-State Abstraction: A New Method for Generating Heuristic Functions, *proceedings of the 23rd AAAI Conference on Artificial Intelligence*, 2008.

[20] Helmert, M.: The Fast Downward Planning System, *Journal of Artificial Intelligence Research*, **26**(1), 2006, 191–246.

[21] Helmert, M., Domshlak, C.: Landmarks, Critical Paths and Abstractions: What's the Difference Anyway?, *Proceedings of the 19th International Conference on Automated Planning and Scheduling*, 2009.

[22] Helmert, M., Haslum, P., Hoffmann, J.: Fexible Abstraction Heuristics for Optimal Sequential Planning, *Proceedings of the 17th International Conference on Automated Planning and Scheduling*, 2007.

[23] Hickmott, S., Rintanen, J., Thiébaux, S., White, L.: Planning via Petri Net Unfolding, *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, 2007.

[24] Hoffmann, J., Edelkamp, S., Thiébaux, S., Englert, R., dos Santos Liorace, F., Trüg, S.: Engineering Benchmarks for Planning: the Domains Used in the Deterministic Part of IPC-4, *Journal of Artificial Intelligence Research*, **26**(1), 2006, 453–541.

[25] Jezequel, L., Fabre, E.: Networks of Automata with Read Arcs: A Tool for Distributed Planning, *Proceedings of the 18th IFAC World Congress*, 2011.

[26] Jezequel, L., Fabre, E.: A#: A Distributed Version of A* for Factored Planning, *Proceedings of the 51th IEEE Conference on Decision and Control*, 2012.

[27] Jezequel, L., Fabre, E.: Turbo Planning, *Proceedings of the 11th International Workshop on Discrete Event Systems*, 2012.

[28] Jezequel, L., Fabre, E., Khomenko, V.: Factored Planning: From Automata to Petri Nets, *Proceedings of the 13th International Conference on Application of Concurrency to System Design*, 2013.

[29] Karpas, E., Domshlak, C.: Cost-Optimal Planning With Landmarks, *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, 2009.

[30] Kautz, H., Selman, B., Hoffmann, J.: SATPLAN: Planning as Satisfiability, *5th International Planning Competition Booklet*, 2006.

[31] Kirsten, D., Maürer, I.: On the Determinization of Weighted Automata, *Journal of Automata, Languages and combinatorics*, **10**(2), 2005, 287–312.

[32] Klimann, I., Lombardy, S., Mairesse, J., Prieur, C.: Deciding Unambiguity and Sequentiality From a Finitely Ambiguous Max-Plus Automaton, *Theoretical Computer Science*, **327**(3), 2004, 349–373.

[33] Knoblock, C.: Generating Parallel Execution Plans With a Partial-Order Planner, *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems*, 1994.

[34] Mohri, M.: Finite-State Transducers in Language and Speech Processing, *Computational Linguistics*, **23**(2), 1997, 269–311.

[35] Mohri, M.: *Handbook of Weighted Automata*, chapter 6, Springer, 2009.

[36] Nissim, R., Brafman, R. I.: Multi-agent A* for parallel and distributed systems, *Proceedings of the 11th International Conference on Autonomous Agents on Multi-Agent Systems*, 2012.

[37] Nissim, R., Brafman, R. I.: Cost-Optimal Planning by Self-Interested Agents, *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[38] Pearl, J.: Reverend Bayes on Inference Engines: A Distributed Hierarchical Approach, *Proceedings of the 2nd National Conference on Artificial Intelligence*, 1982.

[39] Sakarovitch, J.: *Éléments de théorie des automates*, Vuibert, 2003.

[40] Su, R., Wonham, W. M.: Global and Local Consistencies in Distributed Fault Diagnosis for Discrete-Event Systems, *IEEE Transactions on Automatic Control*, **50**(12), 2005, 1923–1935.

[41] Thorsley, D., Teneketzis, D.: Diagnosability of Stochastic Discrete-Event systems, *IEEE Transactions on Automatic Control*, **50**(4), 2005, 476–492.

[42] Zielonka, W.: *The Book of Traces*, chapter 7, World Scientific, 1995.