Master in Computer Science speciality Research in Computer Science

Distributed Optimal Planning in Large Distributed Systems

Loïg Jezequel

Thursday, the 4th of June 2009

Advisor: Eric Fabre (DistribCom)

Abstract: In this report we propose two approaches to factored planning (a relatively new method of problem decomposition exploiting the locality of actions). The first approach is based on a message passing algorithm and on weighted automata calculus. This approach suggests to always manipulate all the plans of a component (e.g. an element of the decomposition of a planning problem) instead of making assumptions on the coordination points between components (as previous approaches did). The second approach is based on the A^* algorithm and suggests a way to distribute it. The originality of these two approaches is that we are able to ensure to find optimal plans.

Introduction

Various domains of computer science are related to planning. In particular *automated planning* is an area of artificial intelligence (AI) which studies the planning process computationally. To be able to perform planning automatically and in a safe way is clearly primordial in domains where a lot of money – or lives – is involved. Indeed, some problems can involve so many variables that they can not be solved by a human being in reasonable time (as for example an efficient reorganization of a network when a part of it is down, the optimal management of several trucks transporting products between several places, or even the schedule of a social event involving many people). But, more than in AI, a lot of problems in computer science can be recasted as planning problems – in fact they can be recasted as the problem of finding a path (potentially optimal) in a discrete state space, which is basically planning. For example, some aspects of model checking, which is a formal verification technique, are based on the exploration of the state space of a system in order to ensure that some property is verified [GNT04]: it can be seen as planning. Another example is fault-diagnosis [SW05]: the problem is here to find paths in automata, which is clearly a planning problem.

So, automated planning is the domain where one has to design a procedure to choose and schedule actions in order to reach goal states from initial states. In fact automated planning is a vast domain and contains different kind of problems. These problems can be separated in three main difficulty levels:

1. *reachability problems* consist in determining whether or not there exists a plan, these problems are the simplest ones;

2. *planning problems* consist in finding a plan if there is one, these problems are a bit harder than reachability problems (in fact, to solve a planning problem allows to answer to the corresponding reachability problem, while solving a reachability problem does not necessary allow to exhibit a plan);

3. *optimal planning problems* consist in finding a plan if there is one and ensure that this plan is optimal (with respect to some costs associated to actions), this is the hardest problem (solving an optimal planning problem ensures to solve the the corresponding planning problem while solving a planning problem does not ensure to find an optimal plan).

Notice that that these three levels of difficulty do not handle every possible requirements: for example one could need good plans but not necessarily optimal ones, which is not really a planning problem nor an optimal planning problem.

An other important point to notice is that, even if, intuitively, a plan is a sequence of actions, this is not a necessity: sometimes actions can occur in different orders without any difference in the outcome. Hence, it is possible to express plans as partial orders of actions instead of sequences of actions [HRTW07]. Moreover, for a given problem, the representation of all the plans as partial orders of actions is more compact than the representation of all the plans as sequences of actions.

The problems studied in automated planning can have very large state-spaces – sometimes even too large to directly solve the problems. Thus, identify and exploit the structure of these problems is one of the main challenges of automated planning. To be able to decompose problems is an important part of this challenge: it allows to solve problems more easily or more quickly (or, for some of these problems, allows to solve them, which is ever significant). Several ways to decompose planning problems have inspired several efficient planning methods [Kno94] [BY94].

In this report we focus on factored planning, a relatively new method which exploits locality of actions in order to split a planning problem into several smaller planning problems (called components). Each component has to be as much independent as possible from the other components. The problem can then be solved by parts, with some consistency requirement between the local plans (solutions of the components). What is really interesting about this method is that the components can be exponentially more compact than the full problem (product of the components) they are extracted from, and, thus, the algorithms involved in planning can be exponentially more efficient when manipulating these components. However, the difficulty of this approach is to combinate local plans into global ones (e.g plans for the full problem) which are feasible.

Several approach to factored planning currently exist [BD06] [BD08] but none of them considers the costs of actions and, thus, none of them performs optimal planning.

In this report we give two new approaches to factored planning. Both performing optimal planning.

The first solution we propose consists in recasting the factored planning problem as an instance of a more general theory of distributed optimization [Fab07] [Dec03]. This theory, combined with weighted automata calculus [Moh09], leads to an algorithm for factored planning which generates optimal global plans (when action costs are taken in a semiring, in our case $(\mathbb{R}^+ \cup \{+\infty\}, \min, +, +\infty, 0)$). In fact, in factored planning, as suggested before, a component is the restriction of a planning problem to a subset of actions. There is interactions between components when some actions modify the variables of several components. In our approach, each component is represented as a weighted automaton (e.g. a transducer from action sequences – plans – to costs). The weighted automata operations allows to manipulate all the locals plans of a component, at the same time and in an efficient way. Moreover these operations ensure that the plans ultimately chosen are compatibles and part of optimal global plans. The fact that we deal with all possible plans at the same time – thanks to weighted automata – is the main difference between our approach to factored planning and the other ones (which make hypothesis on coordination points between components). As we explain in this report, this first approach is practical if: 1) components are small enough to make the necessary weighted automata operations tractable and 2) the interaction graph of the components is sparse enough. The complexity of our approach to factored planning is related to the tree width of this interaction graph, as it was in previous approaches.

The second solution we propose is some kind of distributed A^* algorithm. The A^* algorithm [HNR68] is an efficient algorithm for exploration of state-spaces, using heuristic functions to determinate in which order the states have to be considered. Our algorithm is very close to A^* but, instead of just exploring the state-space of a planning problem, we focus on couples (v, w) where v is a state and w a sequence of coordination points. For that we consider a new heuristic function, provided by the neighbors of a component, which depends of w. We proved this second approach correct when there is only two components and are still working on the general case.

This report is organized as follows. In the first chapter we start by introducing formally automated planning and giving two known approaches to it: A^* algorithm (which gives plans as sequences of actions) and a petri-nets based method (which gives plans as partial orders of actions). After that we introduce factored planning and give known methods to solve it: an approach based on landmarks (e.g. which try to guess the coordination points) and an approach which was developed for fault diagnosis.

In the second chapter we present our first approach for factored optimal planning. We first give an idea of the general theory we use, in particular the message passing algorithm on which is based our approach. Then we explain how the message passing algorithm can be implemented in terms of weighted automata operations and how it solves the factored optimal planning problem.

In the third chapter we present our second approach to factored optimal planning, first in the simple case where only two components are involved, and then in the general case.

Chapter 1

Planning

This chapter is a short introduction to automated planning and factored planning (the modular approach to automated planning). In the first section we give the formal definition of a planning problem, and present two solutions to it: a sequential one, based on the A^* algorithm [HNR68], and a partial order one, based on petri nets unfoldings [HRTW07]. In the second section we introduce the factored planning problem. We also present two solutions to this problem: the first one is based on constraint solving approaches [BD08], and the second one is based on synchronization between languages [SW05].

1.1 Automated Planning

This first section is focused on the traditional planning problem, also called automated planning problem. We first introduce a classical formalization of planning problems. After that we focus on two solutions of such a problem: 1) a solution based on the A^* algorithm, which provide plans as sequences of actions, 2) a solution based on petri-nets and their unfoldings, which provide plans as partial orders of actions.

1.1.1 Basic definitions

We call *planning problem* the problem of choosing and scheduling a set of actions in order to reach some final state from some initial state. To define such a problem formally we first have to introduce the notion of *planning operator*. In the following, for a set A of variables, we denote by $L = A \cup \{\neg a | a \in A\}$ the set of literals over A.

Definition 1. *A* planning operator *over a set A of variables is a pair $\langle p, e\rangle$ such that $p \cup e \subseteq L$, where p is called the set of preconditions and e the set of effect literals.*

In the following we define the *complement* $\bar{l}$ of a literal l by: $\overline{a} = \neg a$ and $\overline{\neg a} = a$, for a a variable. We extend this definition to the complement $\overline{L} = \{\bar{l} | l \in L\}$ of a set L of literals.

Definition 2. *A* planning problem *is a tuple $\mathcal{P} = (A, I, O, G)$ where: A is a finite set of variables, called* state variables*; $I : A \to \{0, 1\}$ is an initial state; G is a set of goal states; O is a set of planning operators over A.*

A solution – also called a plan – to the planning problem $\mathcal{P} = (A, I, O, G)$ is a sequence $s = s_0\langle p_0, e_0\rangle s_1\langle p_1, e_1\rangle \dots s_k$ of states (a state is a valuation of all the variables of A) and planning operators such that:

- $s_0 = I$;
- $s_k \in G$;
- $\forall 0 \leq i < k, \langle p_i, e_i\rangle \in O$;
- $\forall 0 \leq i < k$, the following holds: $p_i \subseteq s_i$ and $s_{i+1} = s_i \setminus \overline{e_i} \cup e_i$ (in this case we say that we can go from the state s_i to the state s_{i+1} using operator $\langle p_i, e_i\rangle$).

In fact we are interested in finding optimal solutions to a given planning problem: solutions which, if each operator (or action) has a cost, minimize the total cost of reaching the goal.

1.1.2 The A^* algorithm

A first and well known way to find optimal solutions to a planning problem is to use the A^* algorithm, introduced in [HNR68] and improved in [HNR72]. This algorithm finds a minimum cost path in a (directed) graph, using heuristics functions. Thus, we first explain how a planning problem can be represented as a graph, and then define the notion of heuristic function and present the A^* algorithm.

From planning problems to directed graphs We define as the path-finding problem $\mathcal{G} = (V, V_i, E, F)$, the problem of finding a path from $V_i \in V$ to any element of $F \subseteq V$ in the directed graph $g = (V, E)$, where the elements of V are called the vertices of g and the elements of $E \subseteq V \times V$ are called the (directed) edges of g.

It is easy to see that solving the planning problem $\mathcal{P} = (A, I, O, G)$ is equivalent to solving the path-finding problem $\mathcal{G} = (V, V_i, E, F)$ where: the set of vertices V matches exactly the set of all possible valuations over A (i.e. the states of $\mathcal{P}$); the initial vertex $V_i \in V$ corresponds to the state I; the set of final vertices F matches exactly the set of states G; there is an edge $(V_j, V_k) \in E$, from $V_j \in V$ to $V_k \in V$, if and only if it is possible to go from state s_j (corresponding to V_j) to state s_k (corresponding to V_k) in $\mathcal{P}$ using an operator from O.

If we assign to any edge e of E a weight corresponding to the cost associated to the operator from O represented by this edge then a minimal-cost solution of the path-finding problem $\mathcal{G} = (V, V_i, E, F)$ gives an optimal solution to the planning problem $\mathcal{P} = (A, I, O, G)$.

Heuristic functions and A^* algorithm The A^* algorithm gives a way to compute an optimal solution to a path-finding problem $\mathcal{G} = (V, V_i, E, F)$. It is based on an *evaluation function* $f : V \to \mathbb{N}$. The idea is to refine, for any $v \in V$ the value $f(v)$, which is an estimate of the optimal path reaching F from V_i, going through v. In fact f is the sum of two functions: g which gives the cost of the actual optimal path from V_i to v; and h which gives an estimation of the cost of an optimal path from v to F. The function h is called *heuristic function* and some conditions about it ensure that the A^* algorithm computes an optimal path.

Algorithm 1 is the A^* algorithm. It involves some data structures: $info$ is an array with an entry for each vertex, $info[v]$ is the entry corresponding to vertex v and contains a tuple (v_p, g, f) where v_p is the predecessor of v on the current shortest path from v_i to v; g is the cost of this path; $f = g + h(v)$ where h is the function described above. The stack *open* contains the next vertices to threat and the operation $add(v, open)$ add the vertex v to the stack *open* before v_1 and after v_2 such that $info[v_2].f \leq info[v].f \leq info[v_1].f$ (if v_1 does not exist then v is the last element of the stack and if v_2 does not exist then v is the first element of the stack).

Algorithm 1 A^* algorithm

```
1:  $info[s] \leftarrow (Nil, 0, h(s))$
2:  $add(s, open)$
3:  let $v = pop(open)$
4:  if $v \in F$ then
5:      return $v$
6:  else
7:      foreach $v'$ such that $(v, v') \in E$ do
8:          compute $f_v(v')$
9:          if $f_v(v') < info[v'].f$ then
10:             $info[v'] \leftarrow (v, g(v', v), f_v(v'))$
11:             $add(v', open)$
12:         endif
13:     done
14:     goto 3
15: endif
```

In [HNR68] is proved that, if $h(v)$ is smaller than the minimal cost to reach F from v, then the A^* algorithm is admissible: it will always terminate and find a path with minimal cost. Hence it is possible to use the A^* algorithm to give an optimal solution to a planning problem, by first translating this problem into a path-finding problem and then applying the A^* algorithm to this new problem.

The main challenge when using A^* algorithm is to find heuristics. In fact these heuristic functions can be automatically generated from a problem. However this generation has to be done in an efficient way: heuristic functions have to be easily computable (e.g. in polynomial time) and should provide tight lower bounds. A classical way to compute heuristic functions is to use *abstractions* (i.e. mappings reducing the state-space). The heuristic function computed from an abstraction $\mathcal{G}'$ of a graph $\mathcal{G}$ is just the exact distance in this abstraction (which can be much more simple to compute than the distance in $\mathcal{G}$ if $\mathcal{G}'$ is well-chosen). Some example of abstraction is given, for example, in [HHH08].

One can notice that the A^* algorithm only gives one optimal solution – i.e. a sequence of operators, or actions – to a given planning problem. In the following we discuss another way to solve planning problems, based on petri net unfoldings, which gives a partially ordered solution to a given planning problem.

1.1.3 Partial orders

Another way to solve planning problems was proposed in [HRTW07]. This solution is based on petri net unfoldings (and more precisely 1-safe place-transition nets, which can be unfolded by tools such as `MOLE`) which are an exact reachability analysis. In this section we first focus on the translation of a planning problem into a 1-safe place-transition net, and then give some intuitions about the link between petri net unfoldings and reachability analysis. For more information about petri nets and unfoldings we refer the reader to [McM93], [ERV96] and [BHHT08].

For the following we denote by $\mathcal{P} = (A, I, O, G)$ a planning problem.

Definition 3. *A planning operator* $\langle p, e\rangle \in O$ *is 1-safe if* $\overline{e} \subseteq p$.

The first step of the translation from planning problems to petri nets is to map any planning problem to an equivalent one such that all planning operators are 1-safe (it will ensure that the petri net built with our transformation is 1-safe). In fact each operator is replaced by $2^{|e\setminus\overline{p}|}$ operators. Given an operator $o = \langle p, e\rangle \in O$, for all possible $e' \subseteq e \setminus \overline{p}$ we define a new operator which works like o when o changes all the literals in e'. These new operators are $\langle p \cup \overline{e'} \cup (e \setminus \overline{p}) \setminus e', e' \cup e \cap \overline{p}\rangle$. In fact, with an example this is much more intuitive: if $o = \langle \{a\}, \{\neg a, b\}\rangle$ the new operators which replace o will be $o_1 = \langle \{a, b\}, \{\neg a\}\rangle$ (when $e' = \emptyset$) and $o_2 = \langle \{a, \neg b\}, \{\neg a, b\}\rangle$ (when $e' = \{b\}$).

Definition 4. *A planning operator* $\langle p, e\rangle \in O$ *has* positive preconditions *if none of the elements of* p *is of the form* $\neg a$ *for* $a \in A$.

The second step of the translation is to map our planning problem with only 1-safe operators to an equivalent problem where all planning operators are 1-safe and have positive preconditions, indeed negative preconditions does not exist in petri nets. The idea is to introduce a new set $\hat{A}$ of state variables defined as $\{\hat{a} | a \in A\}$. Then each negative precondition $\neg a$ is replaced by a corresponding positive precondition $\hat{a}$ and the variable $\hat{a}$ is forced to have the value opposite to a. Formally, $o = \langle p, e\rangle \in O$ is replaced by $o' = \langle p', e'\rangle$, where $p' = (p \cap A) \cup \{\hat{a} | \neg a \in p\}$ and $e' = e \cup \{\neg\hat{a} | a \in e \cap A\} \cup \{\hat{a} | \neg a \in e \cap A\}$. For example $o_2 = \langle \{a, \neg b\}, \{\neg a, b\}\rangle$ will become $o_2' = \langle \{a, \hat{b}\}, \{\neg a, b, \hat{a}, \neg\hat{b}\}\rangle$.

Definition 5. *A* place-transition net *is a tuple* (P, T, F, M_0) *where* P *and* T *are disjoint finite sets – of places and transitions respectively – the flow relation* $F : (P \times T) \cup (T \times P) \to \{0, 1\}$ *indicates the presence or absence of arcs, and* $M_0 : P \to \mathbb{N}$ *is the initial marking.*

A marking $M : P \to \mathbb{N}$ *enables a transition* t *if* $\forall p \in P, F(p, t) \leq M(p)$. *The* firing *of a transition* t *moves the net from the marking* M *to the new marking* M' *such that* $\forall p \in P$, $M'(p) = M(p) - F(p, t) + F(t, p)$.

A place-transition net is 1-safe if, after any number of firings, the number of tokens in each place does not exceed 1.

One can now map our planning problem $\mathcal{P} = (A, I, O, G)$ to the place-transition net (P, T, F, M_0) defined as follows, an example of such a mapping is given in Figure 1.1:

- $P = A \cup \hat{A}$;

- $T = \cup_{o \in O} S(o)$, where $S(o)$ is the set of 1-safe operators with positive precondition obtained from o;
- F is obtained from $t = \langle p, e \rangle \in T$ as $\{(a,t) | a \in p\} \cup \{(t,a) | a \in e \vee a \in p \wedge \neg a \notin e\}$;
- $\forall a \in A, M_0(a) = 1$ if and only if $I(a) = 1$ and $M_0(\hat{a}) = 1$ if and only if $I(a) = 0$, and $\forall a \in A \cup \hat{A}, M_0(a) = 0 \vee M_0(a) = 1$.

1-safe operators associated with o:

$$o_1 = \langle \{a, b\}, \{\neg a\} \rangle$$

$$o_2 = \langle \{a, \neg b\}, \{\neg a, b\} \rangle$$

1-safe operators with positive preconditions associated with o:

$$o'_1 = \langle \{a, b\}, \{\neg a, \hat{a}\} \rangle$$

$$o'_2 = \langle \{a, \hat{b}\}, \{\neg a, b, \hat{a}, \neg\hat{b}\} \rangle$$

Place-transition net associated with o:

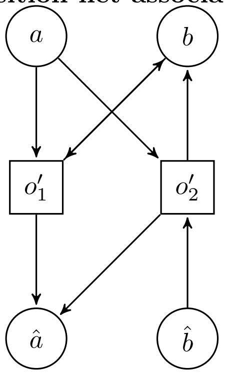

Figure 1.1: Mapping of the operator $o = \langle \{a\}, \{\neg a, b\} \rangle$ into a place-transition net (we do not care about marking because we only transform an operator and not a full planning problem)

Once this place-transition net is built it is possible to use some unfolding program, such as MOLE, to find a solution to the corresponding planning problem. Indeed the unfolding of petri nets is an exact reachability analysis method. It produces, from a petri net, an *occurrence net*; whose nodes are called conditions and events, and represent particular occurrences of places and transitions, respectively, in potential runs – i.e. sequences of firings – of the original petri net (in fact an unfolding is not always finite, that is why programs such as MOLE construct finite complete prefixes of the unfoldings, which give the same knowledge as the whole unfolding). The unfolding eliminates cycles and backward conflicts, which allows one to know exactly which transitions were fired to reach a given marking. Returning to our planning problems these unfoldings allows to know exactly which operators – or actions – were used to give a certain value to some variable at some point of the plan.

Moreover, unfoldings preserve concurrency information. The plans obtained from an unfolding will be partially ordered: if some operators can be used at the same time in a plan then they will not be ordered in it. This notion of concurrency is simpler to understand with an example. Figure 1.2 represents an occurrence net, the conditions are represented by circles and the events by squares. The events e_2 and e_3 are concurrent: one can not know which will occur first.

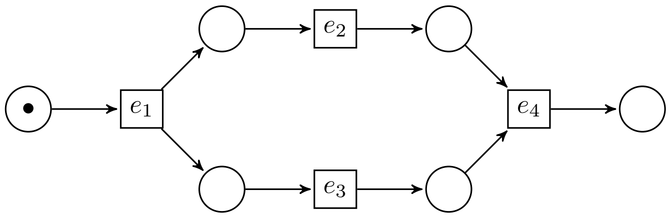

Figure 1.2: An occurrence net

In [HRTW07] and [BHHT08] some solutions are proposed to improve the efficiency of MOLE unfolding by heuristic approaches (taking inspiration from A^* algorithm). The idea is, when testing the reachability of a given state – or set of states – to use a function to estimate the best next step in the unfolding (e.g. the step which seems to be the more relevant for reaching the goal). When using these techniques it is important to ensure that the finite prefix obtained is complete.

1.2 Factored Planning

Sometimes planning problems involve a lot of variables, making such problems untractable if threated directly. A natural idea to solve these large problems is to separate them in several smaller subproblems (called factors), solve these subproblems independently, and merge the results. This way to solve planning problems, called *factored planning* or modular planning, was presented for example in [BD06]. Factored planning can be considered as two independent problems, both hard to solve: 1) separate a problem in factors, 2) solve each factor independently, ensuring that all the solutions are *compatible* (e.g. they can be combined into a solution for the original problem). In the following we focus on the second point which we call the *factored planning problem*.

We first describe several ways to define formally factored planning problems. Then we present two different solutions to these problems: one which was described in [BD08], and another based on a work about fault diagnosis [SW05].

1.2.1 From automated planning to factored planning

A first representation of a factored planning problem is developed in [BD08], and suggests to define subproblems by subsets of the operators of a classical planning problem. This representation is described more formally in the definition below.

Definition 6. *A* factored planning problem *is given by a tuple $(A, I, \{O_i\}_{i=1}^k, G)$ such that: A is a set of state variables, I is an initial state, and G is a set of goal states, as in a planning problem. The set O_i contains the operators $\langle p, e\rangle$ (as in a planning problem) that can be used to solve the i^{th} subproblem.*

One can note that a *factored planning problem* – or modular planning problem – is a collection of several planning problems with interactions between them: some state variables are shared and, thus, using an operator in one problem could change another problem variables values. Of course one could consider such a problem as a unique planning problem, with strong independence of variables, and solve it in one time. But this is not an acceptable solution as factored planning problems can have a huge number of variables. In fact it is more relevant to solve it in a modular way: each planning problem should be solved independently, in a way such that the global plan, reunion of all the plans obtained for the subproblems, is a valid plan and, if possible, an optimal plan, for the whole planning problem. It is clear that such a way to solve planning problems leads to distributed solving: each subproblem can be solved by a independent agent, while the other subproblems are solved by other agents. However, the main argument in favor of a modular solving of planning problems is that it can be impossible (or very difficult) to solve a given planning problem in one time, while it will be feasible in a modular way: the number of potential plans in the whole problem can be much higher than in each subproblem.

In other words a factored planning problem is a collection of smaller planning problems $(A_i, I_i, O_i, G_i)_{i=1}^k$, where: A_i is the set of variables from A involved in the operators of O_i; the local state I_i is the projection over A_i of I; and the set G_i is the projection over A_i of G. These problems have to be solved, as much independently as possible, in a way such that the global plan – reunion of all the local plans of the subproblems – is a solution to the planning problem $(A, I, O = \bigcup_i O_i, G)$. Instead of considering the subproblems (A_i, I_i, O_i, G_i) we could consider the subproblems $(A_i, I_i, O|_{A_i}, G_i)$, where $O|_{A_i} = \{o|_{A_i} | o \in O\}$, and $o|_{A_i}$ is the restriction of $o = \langle p, e\rangle$ to the variables of A_i: $o|_{A_i} = \langle p \cap L_i, e \cap L_i\rangle$, where L_i is the set of literals over A_i.

It is also possible to represent a factored planning problem as a network of automata. The idea is that each automaton represents a subproblem and synchronizes with the automata which share variables with it. We give an idea of how to transform a factored planning problem as described in Definition 6 into a network of automata.

Definition 7. *An* automaton *is a tuple $\mathcal{A} = (S, T, f, s_0, \lambda, \Lambda, F)$ where: S is a set of states; s_0 is the initial state; T is a set of transitions; $f \subseteq (S \times T) \cup (T \times S)$ is a flow relation such that $\forall t \in T, \exists! s \in S, (s, t) \in f \wedge \exists! s \in S, (t, s) \in f$; $\lambda : T \to \Lambda$ affects labels to the transitions; and F is a set of final, or goal, states.*

From this definition comes an intuitive mapping of a subproblem $\mathcal{P}_i = (A_i, I_i, O|_{A_i}, G_i)$, as described above, into an automaton $\mathcal{A} = (S, T, f, s_0, \lambda, \Lambda, F)$:

- s_0 corresponds to I_i;
- S corresponds to all the possible states of $\mathcal{P}_i$;
- Λ corresponds to the operators of O;
- there is a transition T between $s_1 \in S$ and $s_2 \in S$ if and only if an operator $o|_{A_i} \in O|_{A_i}$ allows to move from the state corresponding to s_1 to the state corresponding to s_2;
- λ associates to each transition the corresponding operator o;
- F corresponds to G_i.

In such a subproblem $\mathcal{P}_i$ defined as an automaton $\mathcal{A} = (S, T, f, s_0, \lambda, \Lambda, F)$ a solution is a path from s_0 to F (e.g. a sequence $s_0, t_0, s_1, \ldots, t_{k-1}, s_k$ where $\forall 0 \leq i < k$, $(s_i, t_i) \in f$ and $(t_i, s_{i+1}) \in f$ and $s_k \in F$). One can clearly deduce from such a solution $\pi = s_0, t_0, s_1, \ldots, t_{k-1}, s_k$ a plan for $\mathcal{P}_i$ just by applying λ to π in the following way: $\lambda(\pi) = \lambda(t_0) \ldots \lambda(t_{k-1})$.

In fact, for a factored planning problem $(A, I, (O_i)_{i=1}^k, G)$, we map each local problem $(A_i, I_i, O|_{A_i}, G_i)$ to an automaton $\mathcal{A}_i = (S_i, T_i, f_i, s_{0i}, \lambda_i, \Lambda_i, F_i)$. The transitions t_i of $\mathcal{A}_i$ and t_j of $\mathcal{A}_j$ which corresponds to the same operator o – e.g t_i corresponds to $o|_{A_i}$ and t_j corresponds to $o|_{A_j}$ – are called *shared* transitions. Our automata have to synchronize: if $A_i \cap A_j \neq \emptyset$ then if the local plan in automaton $\mathcal{A}_i$ uses a transition shared with $\mathcal{A}_j$, then the local plan in automaton $\mathcal{A}_j$ also has to use this shared transition "at the same time". This notion of "same time" can be formalized by the *synchronous product* of automata.

Definition 8. *The* synchronous product $\mathcal{A}_1 \|_a \mathcal{A}_2$ *of the automata* $\mathcal{A}_1 = (S_1, T_1, f_1, s_{01}, \lambda_1, \Lambda_1, F_1)$ *and* $\mathcal{A}_2 = (S_2, T_2, f_2, s_{02}, \lambda_2, \Lambda_2, F_2)$ *is* $\mathcal{A} = (S, T, f, s_0, \lambda, \Lambda, F)$ *defined by:*

- $S = S_1 \times S_2$ *and* $s_0 = (s_{01}, s_{02})$;
- $T = \{(t_1, t_2) | t_1 \in T_1 \wedge t_2 \in T_2 \wedge \lambda_1(t_1) = \lambda_2(t_2)\} \cup \{(t_1, \star_2) | t_1 \in T_1 \wedge \lambda_1(t_1) \in \Lambda_1 \setminus \Lambda_2\} \cup \{(\star_1, t_2) | t_2 \in T_2 \wedge \lambda_2(t_2) \in \Lambda_2 \setminus \Lambda_1\}$;
- f *is given by:*
  - $((s_1, s_2), (t_1, t_2)) \in f$ *and* $((t_1, t_2), (s'_1, s'_2)) \in f$ *if and only if* $(s_i, t_i) \in f_i$ *and* $(t_i, s'_i) \in f_i$ *for* $i \in \{1, 2\}$,
  - $((s_1, s_2), (t_1, \star_2)) \in f$ *and* $((t_1, \star_2), (s'_1, s_2)) \in f$ *if and only if* $(s_1, t_1) \in f_1$ *and* $(t_1, s'_1) \in f_1$,
  - $((s_1, s_2), (\star_1, t_2)) \in f$ *and* $((\star_1, t_2), (s_1, s'_2)) \in f$ *if and only if* $(s_2, t_2) \in f_2$ *and* $(t_2, s'_2) \in f_2$;
- $\Lambda = \Lambda_1 \cup \Lambda_2$.
- $F = F_1 \times F_2$

One can notice that the automaton $\mathcal{A}$, synchronous product of all the automata of a network gives all the solutions to the corresponding planning problem.

Finally we can represent the factored planning problem in terms of languages: if in the formalization with automata networks, described above, we replace each automaton by its language, we can use operations about languages instead of operations about automata to solve the planning problem. This formalization was developed in [SW05] for the fault diagnosis problem.

In the next sections we examine these formalizations, given two modular solutions to factored planning problem: one in the context of Definition 6 (Section 1.2.2) and the other in the context of languages (Section 1.2.3).

1.2.2 A solution based on landmarks

A distributed way to solve factored planning problems was proposed in [BD08], using the representation of the problem proposed in Definition 6. The idea is to assume that each subproblem has a finite and determined maximum number δ of coordination points: in a solution to this subproblem no more than δ operators which uses variables shared with other subproblems will be used. Then the assignation of operators to these coordination points is considered as some constraint problem – as described in [Dec03] – with two kinds of constraints (coordination constraints to decide which operator should be used at each coordination point, and internal-planning constraints to find a plan with only private actions of a unique agent between the coordination points). The resolution of the whole problem involves coordination between subproblems, in order to solve this constraint problem. If it does not lead to a solution the maximum number of coordination points is incremented.

More formally, using the notations of Definition 6 (and assuming that each operator has positive preconditions), for the i^{th} subproblem $\mathcal{P}_i = (A, I, O_i, G)$, we denote by A_i the variables involved in operators of $\mathcal{P}_i$, defined as: $\bigcup_{\langle p,e\rangle \in O_i} p \cup e$. For the subproblem $\mathcal{P}_i$ we also distinguish two types of variables: internal variables and public variables (or shared variables), defined as $A_i^{int} = A_i \setminus \bigcup_{j\in\{1,\dots,k\}\setminus\{i\}} A_j$ and $A_i^{pub} = A_i \setminus A_i^{int}$. From this we define for each subproblem $\mathcal{P}_i$ a set of internal operators $O_i^{int} = \{o = \langle p, e\rangle | o \in O_i \wedge p \cup e \subseteq A_i^{int}\}$ and a set of public operators $O_i^{pub} = O_i \setminus O_i^{pub}$.

A constraint satisfaction problem, as presented in [Dec03], is defined by a finite set of variables $U = \{u1, \dots, u_k\}$, an associated set of domains $D = \{D_1, \dots, D_k\}$ which list the possible values for each variable: $D_i = \{v_1, \dots, v_n\}$, and a set of constraints $C = \{C_1, \dots, C_t\}$ over these variables. A constraint C_i is a relation defined on a subset of variables. The relation denotes the variables' simultaneous legal value assignments.

Here we associate a variable to each subproblem: $U = \{u_i\}_{i=1}^k$ where u_i is associated to the i^{th} subproblem $\mathcal{P}_i = (A, I, O_i, G)$. The variable u_i represents the choices of coordination points for the subproblem $\mathcal{P}_i$. It is a sequence $u_i = u_i^1, \dots, u_i^\delta$ which size is δ (the maximum number of coordination points) and where u_i^j is either a public operator from O_i^{pub} or an empty symbol. The value of u_i gives the public part of the plan s_i which will used to solve the i^{th} subproblem: it is a subsequence of s_i which contains all the occurrences of public operators in s_i.

As suggested above we have two types of constraints, here defined in a high level manner instead of in terms of relations:

Coordination constraints: an assignment $(\theta_1, \dots, \theta_k)$ to U satisfies this constraint if and only if, for $1 \le i \le k$, $\langle p_i, e_i\rangle$ is at position t in θ_i implies that for each $p \in p_i \cap A_i^{pub}$ the following holds:

- for some θ_j, $\langle p_j, e_j\rangle$ is at position t' in θ_j and $p \in e_j$ and $t' < t$ ("someone supplies p before t"); and
- for no θ_l, $\langle p_l, e_l\rangle$ is at position t'' in θ_l and $\neg p \in e_l$ and $t' \le t'' \le t$ ("nobody destroys p in $[t', t]$").

Internal-planning constraints: an assignment $(\theta_1, \dots, \theta_k)$ to U satisfies this constraint if and only if, for each $\theta_i = o_1 \dots o_\delta$, the problem $(A_i, I \cap A_i, O_i^{int}, \emptyset, (o_1|_{int}, \dots, o_\delta|_{int}))$, called problem with operator landmarks, has a solution. The goal of the problem with operator landmarks is to find a solution to the planning problem $(A_i, I \cap A_i, O_i^{int} \cup \{o_1|_{int}, \dots, o_\delta|_{int}\}, \emptyset)$ such that $(o_1|_{int}, \dots, o_\delta|_{int})$ is a subsequence of this solution, where $o_l|_{int}$ is the projection on A_i of the public operator o_l.

We denote by $\mathtt{CSP}_\delta$ the constraint satisfaction problem described above. One can define if such a problem has a solution, and, if it exists, find one of its solutions. This is the purpose of Algorithm 2, which solves in a modular way the factored planning problem.

In [BD08] two important properties are given, which prove the validity of Algorithm 2:

1. if an assignment $(\theta_1, \dots, \theta_k)$ is a satisfying assignment to $\mathtt{CSP}_\delta$, then it can be used to build a full solution for $(A, I, \{O_i\}_{i=1}^k, G)$;
2. for each problem $(A, I, \{O_i\}_{i=1}^k, G)$ there exist δ such that $\mathtt{CSP}_\delta$ is solvable.

Hence, Algorithm 2 provide a solution to a factored planning problem. The main part of this algorithm is to find a solution to $\mathtt{CSP}_\delta$. This resolution involves two phases: first the landmarks – i.e. the

Algorithm 2 solution to $(A, I, \{O_i\}_{i=1}^k, G)$

```
1: $\delta \leftarrow 1$
2: loop
3:     construct $\mathtt{CSP}_\delta$
4:     if $\mathtt{CSP}_\delta$ has a solution then
5:         build a solution $\rho$ to $(A, I, \{O_i\}_{i=1}^k, G)$ from a solution to $\mathtt{CSP}_\delta$
6:         return $\rho$
7:     else
8:         $\delta \leftarrow \delta + 1$
9:     endif
10: endloop
```

coordination points – have to be chosen (using the coordination constraints) and then it is necessary to fill the local plans, using these landmarks (it corresponds to internal-planning constraints). This way to solve the problem does not really use all the characteristics of planning problems. The choice of coordination points could be improved, involving methods such as backtracking: instead of trying to find coordination points for all the subproblems at once it could be possible to find potential coordination points for one subproblem and then try to extend them to another subproblem, and so on. If for one subproblem it is impossible to find coordination points consistent with the current coordination points for other problems then the coordination points for the previous subproblem have to be changed.

Moreover this solution does not necessary give an optimal plan and there is no result in [BD08] about a way to find an optimal one. The only fact is that the solution obtained is optimal with regard to δ: the number of coordination points used is minimal.

In the next part we discuss another solution to factored planning problem, based on languages, which gives all the possible plans for a given problem.

1.2.3 A solution based on languages

In this section we focus on two algorithms proposed in [SW05], that can potentially be applied to solve the factored planning problem, using the formalism of language theory, as it was suggested in Section 1.2.1. These algorithms were introduced for distributed fault diagnosis but are also relevant for planning. One important point about this language theory approach is that instead of searching for a single plan, as in Section 1.2.2, we search for all the plans for a given planning problem. This induces potential increasings of complexity but conducts to the solution of a more general problem.

We first give basic notions of language theory, in particular the notions of local and global consistencies, then define the factored planning problem in terms of languages, and finally give algorithms to achieve local and global consistencies and discuss about the links between these consistencies and the planning problem.

Definitions Let Σ be an alphabet. A word over Σ is a finite sequence of elements of Σ. We denote by Σ^+ the set of all possible words over Σ. Let ϵ be the empty word, we denote by Σ^* the set of words $\Sigma^+ \cup \{\epsilon\}$. A language L is a subset of Σ^*.

Let $\Sigma' \subseteq \Sigma$, we define the *natural projection* $P : \Sigma \to \Sigma'$ as:

- $P(\epsilon) = \epsilon$;
- $\forall \sigma \in \Sigma, P(\sigma) = \begin{cases} \sigma \text{ if } \sigma \in \Sigma' \\ \epsilon \text{ if } \sigma \notin \Sigma' \end{cases}$;
- $\forall \omega \in \Sigma^*, \forall \sigma \in \Sigma, P(\omega\sigma) = P(\omega)P(\sigma)$.

Let $A \subseteq \Sigma^*$, we denote by $P(A)$ the set $\{P(\omega) | \omega \in A\}$. The inverse image function of P is $P^{-1} : 2^{\Sigma'^*} \to 2^{\Sigma^*}$, defined by:

$$\forall U \subseteq \Sigma'^*, P^{-1}(U) = \{\omega \in \Sigma^* | P(\omega) \in U\}.$$

Let Σ_1 and Σ_2 be two alphabets, such that $\Sigma = \Sigma_1 \cup \Sigma_2$. We denote by $P_1 : \Sigma^* \to \Sigma_1$ and $P_2 : \Sigma^* \to \Sigma_2$ the natural projections over Σ_1 and Σ_2. Let $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ be two languages. The *synchronous product* of L_1 and L_2 is defined by $L_1 \| L_2 = P_1^{-1}(L_1) \cap P_2^{-1}(L_2)$. One can notice that $L_1 \| L_2 = \{\omega \in \Sigma^* | P_1(\omega) \in L_1 \wedge P_2(\omega) \in L_2\}$. The synchronous product is commutative and associative. Hence, for an index set I, a set of alphabets $\{\Sigma_i | i \in I\}$, and a corresponding set of languages $\{L_i \subseteq \Sigma_i^* | i \in I\}$, the synchronous product $\|_{i \in I} L_i$ is well defined.

Let I be an index set, $\{\Sigma_i | i \in I\}$ is a set of alphabets, and $\{L_i \subseteq \Sigma_i^* | i \in I\}$ is a corresponding set of languages. By $P_{J,K}$ we denote the natural projection from $J \subseteq I$ to $K \subseteq I$. We can define two notions of consistency, as in [SW05]:

Global consistency: A set $\mathcal{E} = \{E_i \subseteq L_i | i \in I\}$ is globally consistent with respect to I if, $\forall i \in I$, $E_i = P_{I,\{i\}}(\|_{j \in I} E_j)$.

Local consistency: A set $\mathcal{E} = \{E_i \subseteq L_i | i \in I\}$ is locally consistent with respect to I if, $\forall i, j \in I$, $P_{\{i\},\{j\}}(E_i) = P_{\{j\},\{i\}}(E_j)$.

To express the factored planning problem in terms of language theory one can use the automata version of this problem presented in 1.2.1. Each automaton is replaced by its language, L_i with the notations defined above, and the synchronous product between automata is replaced by the synchronous product between languages. In fact we are interested in computing $\mathcal{E}$ which are globally consistent with respect to I. Moreover we would like to obtain the maximal $\mathcal{E}$ which is globally consistent with I (where order between these sets is defined as: $\mathcal{E}' \leq \mathcal{E}$ if and only if $\forall i \in I, E'_i \subseteq E_i$). The planning problem with languages can be formally defined as follow.

Definition 9. *A set of languages $\{L_i \subseteq \Sigma_i^* | i \in I\}$ indexed by I is a planning problem, where the languages are the components. The solution to this planning problem is the maximal set $\{E_i \subseteq L_i | i \in I\}$ which is globally consistent with respect to I.*

The relation between global consistency and solutions to a planning problem is quite similar to the idea of coordination points presented in the previous section. The projection of elements from E_i, part of the set described in the Definition 9, over the shared elements of Σ_i^* gives all the landmarks – e.g. sequences of coordination points – which can be extended into a plan for the global problem.

In the following we first give the solution proposed in [SW05] to solve the planning problem with languages. Then, we focus on [Fab07] and explain why local consistency can also be useful.

Global consistency In [SW05] an algorithm is given to find the maximal set $\mathcal{E} = \{E_i \subseteq L_i | i \in I\}$ which achieves global consistency with respect to I. We could just compute E_i as $P_{I,\{i\}}(\|_{j \in I} L_j)$ for all $i \in I$, but, if there is a huge number of components, compute directly $\|_{j \in I} L_j$ could be infeasible. Algorithm 3 computes E_n, that is $P_{I,\{n\}}(\|_{j \in I} L_j)$. By Σ_J, for $J \subseteq I$, we denote $\bigcup_{j \in J} \Sigma_j$.

Algorithm 3 $E_n = P_{I,\{n\}}(\|_{j \in I} L_j)$ for $I = \{1, \dots, n\}$

```
1: $T_0 \leftarrow \Sigma_1$, $W_0 \leftarrow \Sigma_1^*$
2: for $k = 1$ to $n - 1$ do
3:     $J_k \leftarrow \{1, \dots, k\}$
4:     $T_k \leftarrow \Sigma_{J_k} \cap \Sigma_{I \setminus J_k}$
5:     $W_k \leftarrow P_{T_{k-1} \cup \Sigma_k, T_k}(W_{k-1} \| L_k)$
6: endfor
7: $E_n \leftarrow W_{n-1} \| L_n$
```

In fact, in the worst case, Algorithm 3 has the same complexity as for computing the whole synchronous product of the L_i. But, in practice, the complexity is frequently much lower than for the whole synchronous product. Moreover, it is possible, instead of counting from $k = 0$ to $k = n - 1$ to rely on heuristics for the enumeration of languages, in order to obtain smaller complexities. This is described in [SW05].

Local consistency One can notice that global consistency implies local consistency. Unfortunately the reverse is not generally true. For example, if $\Sigma_1 = \{\alpha, \beta\}$, $\Sigma_2 = \{\alpha, \gamma\}$, and $\Sigma_3 = \{\beta, \gamma\}$. One can notice that for $L_1 = \{\alpha\beta\}$, $L_2 = \{\gamma\alpha\}$, and $L_3 = \{\beta\gamma\}$, the following holds: $P_{\{1\},\{2\}}(L_1) = P_{\{2\},\{1\}}(L_2)$, $P_{\{1\},\{3\}}(L_1) = P_{\{3\},\{1\}}(L_3)$, and $P_{\{2\},\{3\}}(L_2) = P_{\{3\},\{2\}}(L_3)$, thus, local consistency is achieved; while $L_1 \| L_2 \| L_3 = \emptyset$, thus, global consistency is not achieved.

Thus one can ask why we are interested in local consistency. In fact, an important thing to notice is that the maximal set globally consistent with respect to an index set I is smaller than the maximal set locally consistent with respect to I: local consistency gives an over-approximation of global consistency. Moreover local consistency is generally easier to achieve than global consistency. Another point is that, as we explain below, local consistency is sometimes equivalent to global consistency.

We now define the concept of *communication graph* of a planning problem, which allows us to derive an algorithm achieving local consistency. After that we give a sufficient condition over the communication graph to ensure that the algorithm also achieves global consistency. This was described in [Fab07].

Definition 10. *The* connectivity graph *corresponding to a planning problem $\{L_i \subseteq \Sigma_i | i \in I\}$ is a graph where vertices are the elements of I and there is an (undirected) edge (i, j) between $i \in I$ and $j \in I$ if and only if $\Sigma_i \cap \Sigma_j \neq \emptyset$.*

In a connectivity graph we call *redundant edge* an edge (i, j) such that there exist a path $(i, k_1, \ldots, k_L, j)$ from i to j where $\Sigma_i \cap \Sigma_j \subseteq \Sigma_{k_l}$ and $k_l \notin \{i, j\}$ for $1 \leq l \leq L$.

Definition 11. *The* communication graph *of a planning problem is deduced from the connectivity graph of this problem by recursively removing redundant edges until minimality is reached (in fact there is several communication graphs for a same connectivity graph).*

Algorithm 4 is a specific utilization of a more general algorithm, called message passing algorithm. It takes a planning problem and a corresponding communication graph and, when it terminates, returns a collection $\{E_i \subseteq L_i | i \in I\}$ which achieves local consistency. Each component i of the system, i.e each vertex of the communication graph, maintains a message $\mathcal{M}_{i,j}$ for each of its neighbors j. By $\mathcal{N}(i)$ we denote the set of neighbors of i in the communication graph and by P_Σ we denote the natural projection over Σ.

Algorithm 4 Su's message passing algorithm

```
1: $\mathcal{M}_{i,j} = \bigcup_{i \in I} \Sigma_i$ for all $(i, j)$ edge of the communication graph
2: until stability of messages do
3:     select an edge $(i, j)$
4:     $\mathcal{M}_{i,j} \leftarrow P_{\Sigma_i \cap \Sigma_j}(L_i \| (\|_{k \in \mathcal{N}(i) \setminus \{j\}} \mathcal{M}_{k,i}))$
5: enduntil
6: $E_i = L_i \| (\|_{k \in \mathcal{N}(i)} \mathcal{M}_{k,i})$ for $1 \leq i \leq n$
```

An important fact about Algorithm 4 is given in [Fab07]:

Theorem 1. *If the planning problem studied lives on a tree – which means that all its communication graphs are trees – then Algorithm 4 terminates and achieves global consistency.*

An other thing to notice is that the choice of the edges to update the messages in Algorithm 4 is not specified. Hence, this algorithm is asynchronous and, thus, can be fully distributed.

We presented the notions of global and local consistencies. We then described the planning problem as a global consistency achievement. After that we gave an algorithm to compute global consistency, which, in the worst case, is not better than just computing a huge synchronous product but can frequently be really more efficient. Finally we discussed the links between local and global consistencies and remarked that a specific structure of the communication graphs of a planning problem can lead to an equivalence between these forms of consistencies.

In this chapter we presented the planning problem and the factored planning problem. We also gave some methods to solve these problems.

The two approaches we presented for the planning problem – one based on the A^* algorithm and the other on petri-nets unfolding – have strong differences. The A^* algorithm is a well known method which can be really efficient and give optimal plans. However it strongly depends on the heuristic used. Moreover it only provides solutions as sequences of actions. The petri-nets based approach has the strong advantage to provide solutions as partial orders of actions (the plan-space is smaller than for plans as sequences of actions). The main drawback is that the mapping from planning problems to petri-nets can be expensive.

The first approach we gave for factored planning problems is based on constraints satisfaction. Its main drawback is that it can not deal with problems which have no solution: in this case the algorithm does not converge. Moreover this approach does not return optimal plans. However this is a factored planning approach: some planning problems which are untractable with classical approaches could be tractable with this one. The second approach we presented is based on languages. Its main drawback is the necessity for the studied problems to live on trees. However this approach can deal with problems which have no solutions (while the problems live on trees it always converge). Moreover the outcome this approach provide contains all the solutions of the problem studied, this is a first step to factored optimal planning.

In the next chapters we introduce the factored optimal planning problem, which is close to the factored planning problem but with a notion of optimization. We propose two solutions to this problem: the first one is based on the same idea as in [SW05] or [Fab07] (e.g. the message passing algorithm), the second solution is inspired from the A^* algorithm (in fact we could see it as a distributed A^* algorithm).

Chapter 2

Factored Optimal Planning Using Weighted Automata Calculus

In this chapter we present a new approach of factored planning, based on automata calculus [Moh09] and an algorithm, called message passing algorithm (MPA), presented in a general framework in [Fab07] but also in more specific papers like [Dec03] or [SW05].

Sometimes one can not just search for a plan but needs to have a (near to) optimal plan (e.g. a plan which is the less expensive as possible in terms of resource consumption, or anything else). In our study each action is associated with a cost and the cost of a plan is the sum of the costs of all its actions. An optimal plan is a plan which has minimal cost. To our knowledge the approach of factored planning we develop here is the first which can ensure to find optimal plans. Moreover, as we explain in this chapter, our approach, due to the MPA functioning, can easily be distributed: more than factored optimal planning we deal with distributed optimal planning.

Even if our approach can look a bit unpractical, because the MPA enforces problems to have a specific structure, it seems that a lot of planning problems can be handled. In fact, we can a priori handle exactly the same problems as the method we described in Section 1.2.3, but with a notion of optimization added.

We first describe the message passing algorithm (MPA) from [Fab07] and its application to factored planning (which matches the results presented in Section 1.2.3). After that we explain how the MPA can be implemented in terms of weighted automata calculus and prove that it allows to perform factored optimal planning. Finally we give an example of a planning problem, and show how the MPA is concretely applied to solve it.

2.1 Factored Planning Using Automata Calculus

The theory we present here was described, in the general case, in [Fab07], but also in more specific papers such as [Dec03] or [SW05]. This theory is based on a notion of system and on two operations – composition and projection – on these systems. It allows us to derive an efficient algorithm for factored planning: the MPA. Some axioms on the operations ensure the validity of this theory.

In this section we first define the systems, operations on them, and give the required axioms. We then describe the family of MPA. Finally we show how this algorithm can be used when the systems are languages or automata.

2.1.1 Message passing algorithm

Let $\mathcal{V}_{max}$ be a set of variables. We denote by $S_1, S_2, \ldots$ systems defined over subsets of these variables (this is specified later). Moreover, with the notion of system, two operations are provided: composition ($\wedge$) and projection (Π). The composition is associative and commutative and constructs from two systems S_1 defined over $\mathcal{V}_1 \subseteq \mathcal{V}_{max}$ and S_2 defined over $\mathcal{V}_2 \subseteq \mathcal{V}_{max}$ the new system $S = S_1 \wedge S_2$ (which, as we see later, is defined over $\mathcal{V}_1 \cup \mathcal{V}_2$). The projection takes the form of a family of operators $\Pi_{\mathcal{V}_i}$ indexed by sets of variables $\mathcal{V}_i \subseteq \mathcal{V}_{max}$. Intuitively, $\Pi_{\mathcal{V}_i}(S)$ projects S on $\mathcal{V}_i$, which results in a system over variables $\mathcal{V}_i$.

These operations are provided with four axioms. The first one defines the projection operation:

$$\forall \mathcal{V}_1, \mathcal{V}_2 \subseteq \mathcal{V}_{max}, \quad \Pi_{\mathcal{V}_1} \circ \Pi_{\mathcal{V}_2} = \Pi_{\mathcal{V}_1 \cap \mathcal{V}_2}. \tag{2.1}$$

The second axiom expresses that system S operates on a subset of variables from $\mathcal{V}_{max}$:

$$\forall S, \exists \mathcal{V} \subseteq \mathcal{V}_{max}, \quad \Pi_{\mathcal{V}}(S) = S. \tag{2.2}$$

The third axiom, which allows us to derive the MPA, expresses that the interaction between two systems S_1 and S_2 is fully captured by their shared variables $\mathcal{V}_1 \cap \mathcal{V}_2$:

$$\forall \mathcal{V}_3 \supseteq \mathcal{V}_1 \cap \mathcal{V}_2, \quad \Pi_{\mathcal{V}_3}(S_1 \wedge S_2) = \Pi_{\mathcal{V}_3}(S_1) \wedge \Pi_{\mathcal{V}_3}(S_2). \tag{2.3}$$

The last axiom denotes the existence of an identity element $\mathbb{I}$ for composition:

$$\exists \mathbb{I}, \forall S, \quad S \wedge \mathbb{I} = S. \tag{2.4}$$

An important problem about such systems is the *reduction problem*, which consists in computing $S'_i = \Pi_{\mathcal{V}_i}(S)$, e.g. what become the S_i once inserted in $S = S_1 \wedge \cdots \wedge S_n$, without computing the full compound system S. Indeed, a compound system can be very large, and sometimes intractable. In fact, it was shown in [Fab03] that these S'_i represent exactly the local solutions that can be extended into global ones. Moreover, under the strong assumption that composition is involutive (e.g. $\forall S, \forall \mathcal{V}, S \wedge \Pi_{\mathcal{V}}(S) = S$), the S'_i are the minimal systems such that $S = S'_1 \wedge \cdots \wedge S'_n$ (one could notice the analogy with the local consistency of Section 1.2.3).

As an example, consider the case where the systems are constraints satisfaction problems over variables. The $\mathcal{V}_i$ are sets of variables over a domain $\mathcal{D}$ and a system S_i is a set of assignments $\mathbf{x}_i : \mathcal{V}_i \to \mathcal{D}$ (representing all the assignments allowed by the constraints of system S_i). The composition of systems is the conjunction of constraints: $S_i \wedge S_j = \{\mathbf{x} : \mathcal{V}_i \cup \mathcal{V}_j \to D \mid \mathbf{x}_{|\mathcal{V}_i} \in S_i, \mathbf{x}_{|\mathcal{V}_j} \in S_j\}$. The projection is the restriction of variables: $\Pi_{\mathcal{V}'}(S) = \{\mathbf{x}_{|\mathcal{V}'} \mid \mathbf{x} \in S\}$. In this case, for a compound system $S = S_1 \wedge \cdots \wedge S_n$, any $\mathbf{x}_i \in \Pi_{\mathcal{V}_i}(S)$ is the local view in S_i of a solution to the global system S, and any solution $\mathbf{x}$ to the system S has a projection $\mathbf{x}_i = \mathbf{x}_{|\mathcal{V}_i}$ in each $S'_i = \Pi_{\mathcal{V}_i}(S)$.

Another illustration is a constrained optimization problem, where costs are now associated to the tuples $\mathbf{x}_i$. So a system S_i is now a function mapping all possible $|\mathcal{V}_i|$ tuples $\mathbf{x}_i$ to elements from $\mathbb{R}^+ \cup \{+\infty\}$. By convention $S_i(\mathbf{x}_i) = +\infty$ means that $\mathbf{x}_i$ is not allowed. One can notice that when 0 and $+\infty$ are the only allowed costs this case is similar to a constraint satisfaction problem. The composition of systems consists in summing costs of tuples. For example, for a tuple $\mathbf{x} : \mathcal{V}_i \cup \mathcal{V}_j \to \mathcal{D}$, we have: $(S_i \wedge S_j)(\mathbf{x}) = S_i(\mathbf{x}_{|\mathcal{V}_i}) + S_j(\mathbf{x}_{|\mathcal{V}_j})$. The projection is responsible for optimization, by minimizing the cost over discarded variables. For a tuple $\mathbf{x} = (\mathbf{x}_i, \overline{\mathbf{x}}_i)$ over $\mathcal{V} = \mathcal{V}_i \uplus (\mathcal{V} \setminus \mathcal{V}_i)$ we have: $(\Pi_{\mathcal{V}_i}(S))(\mathbf{x}_i) = \min_{\overline{\mathbf{x}}_i} S(\mathbf{x}_i, \overline{\mathbf{x}}_i)$. For a system $S = S_1 \wedge \cdots \wedge S_n$ it does not hold that $S = \Pi_{\mathcal{V}_1}(S) \wedge \cdots \wedge \Pi_{\mathcal{V}_n}(S)$, but the minimal cost element(s) of $\Pi_{\mathcal{V}_i}(S)$ (if exists) can be extended into minimal cost element(s) of S. Indeed, if $\mathbf{x}$ is an optimal tuple in S, then $\mathbf{x}_{|\mathcal{V}_i}$ is also an optimal tuple in $\Pi_{\mathcal{V}_i}(S)$, with the same cost: $S(x) = (\Pi_{\mathcal{V}_i}(S))(\mathbf{x}_{|\mathcal{V}_i})$. And an optimal tuple $\mathbf{x}_i$ in $\Pi_{\mathcal{V}_i}(S)$ is always such that $\mathbf{x}_i = \mathbf{x}_{|\mathcal{V}_i}$ with $\mathbf{x}$ an optimal tuple in S and we still have that: $(\Pi_{\mathcal{V}_i}(S))(\mathbf{x}_i) = S(x)$.

We now define the notion of *communication graph* which is a generalization of the notion presented in Section 1.2.3 and is necessary to introduce the MPA (which solves the reduction problem, under some conditions over the communication graph).

Definition 12. *The* connectivity graph *of a compound system $S = S_1 \wedge \cdots \wedge S_n$ is a graph where the n vertices are $\{1, \ldots, n\}$ and there is an edge (i, j) from i to j if and only if $\mathcal{V}_i \cap \mathcal{V}_j \neq \emptyset$.*

In a connectivity graph an edge (i, j) is said to be *redundant* if there is a path $(i, k_1, \ldots, k_L, j)$ from i to j such that $\mathcal{V}_i \cap \mathcal{V}_j \subseteq \mathcal{V}_{k_\ell}$ and $k_\ell \notin \{i, j\}$ for $1 \leq \ell \leq L$.

Definition 13. *A* communication graph *of a compound system is deduced from the connectivity graph of this system by recursively removing redundant edges until minimality is reached.*

Given n systems $S_1, \dots, S_n$ (called components) over n sets of variables $\mathcal{V}_1, \dots, \mathcal{V}_n$, such that $S = S_1 \wedge \cdots \wedge S_n$ lives on a tree (e.g. its communication graphs are trees), the MPA (Algorithm 5) allows one to compute the projections $S'_i = \Pi_{\mathcal{V}_i}(S)$ of the compound system S without computing S itself.

Algorithm 5 takes as input the S_i and a communication graph $\mathcal{G}_S$ of the compound system S and returns, when terminates, a collection of systems S'_i, one for each S_i. We denote by $\mathcal{N}(i)$ the neighbors of vertex i in the communication graph $\mathcal{G}_S$. Each vertex i maintains for each of its neighbors j a message $\mathcal{M}_{i,j}$, these messages have the same nature as the components and are initialized to $\mathbb{I}$ (the identity element for composition). Intuitively the message $\mathcal{M}_{i,j}$ gives to j the knowledge that i has about its side of $\mathcal{G}_S$ (composition), restricted to what is useful for j (projection).

Algorithm 5 The message passing algorithm

```
$\mathcal{M}_{i,j} \leftarrow \mathbb{I}$, $\forall (i,j) \in \mathcal{G}_S$
until stability of messages do
    select an edge $(i,j)$
    $\mathcal{M}_{i,j} \leftarrow \Pi_{\mathcal{V}_i \cap \mathcal{V}_j} \left( S_i \wedge \left( \wedge_{k \in \mathcal{N}(i) \setminus j} \mathcal{M}_{k,i} \right) \right)$
done
$S'_i \leftarrow S_i \wedge \left( \wedge_{k \in \mathcal{N}(i)} \mathcal{M}_{k,i} \right)$, $\forall i \in \{1, \dots, n\}$
```

The convergence of this algorithm is only ensured when $\mathcal{G}_S$ is a tree (in other words, S lives on a tree). In fact it is always possible to build a tree (called junction tree [Dec03]) by aggregating some components of a compound system. For example, in Figure 2.1, the communication graph of $S = S_1 \wedge \cdots \wedge S_4$ is not a tree but, by aggregating S_2 and S_3 into one component $S_{2,3} = S_2 \wedge S_3$, we obtain a new communication graph which is a tree. The drawback of this method is that merging components increases their complexity (and consequently the complexity of compositions and projections performed on them), generally exponentially in the number of aggregated components.

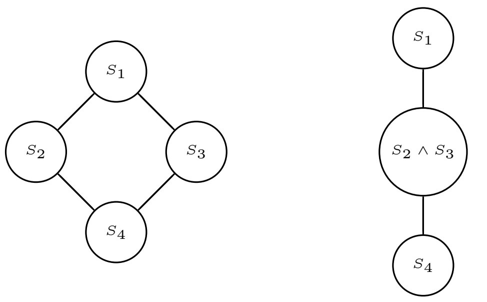

Figure 2.1: Communication graphs of $S_1 \wedge \cdots \wedge S_4$ (left) and $S_1 \wedge S_{2,3} \wedge S_4$ (right)

Notice that in Algorithm 5 the ordering of messages updates is not specified. Hence the MPA is asynchronous and can easily be distributed. Another important thing about the messages is that, when $\mathcal{G}_S$ is a tree (e.g. when the convergence is ensured), it is possible to define a scheduling of messages updates in order to ensure reaching of stability with at most one update per message. A way to do that is to update the messages in the following manner: update message $\mathcal{M}_{i,j}$ when all the $\mathcal{M}_{k,i}$ for $k \neq j$ have been updated. In Figure 2.2 is shown an example of such a scheduling of messages updates. In this figure each message is represented by a dot, near the state it comes from (i.e. $\mathcal{M}_{i,j}$ is on the edge (i,j), close to i). To ensure stability after one update per message the messages can be updated in the following order (as suggested in the figure): $\mathcal{M}_{4,2}, \mathcal{M}_{5,2}, \mathcal{M}_{2,1}, \mathcal{M}_{1,3}, \mathcal{M}_{3,6}, \mathcal{M}_{6,3}, \mathcal{M}_{3,1}, \mathcal{M}_{1,2}, \mathcal{M}_{2,4}, \mathcal{M}_{2,5}$.

2.1.2 Handling languages

We show here that the theory described above works well when systems are languages, in the sense of Section 1.2.3, associated to their alphabets, $S = (\mathcal{L}, \Sigma)$, and variables are elements of the alphabets, $\mathcal{V} = \Sigma$. The composition operation is the synchronous product of languages, and the projection is the natural projection. One can notice that, in this case, the MPA is exactly Algorithm 4. Hence,

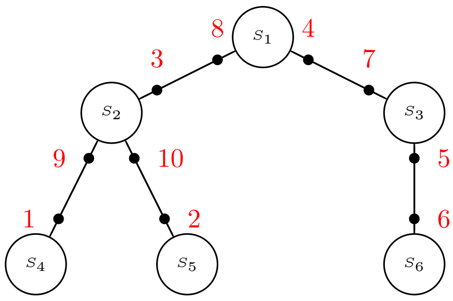

Figure 2.2: A communication graph which is a tree, the dots represent messages and the numbers a scheduling of updates which ensures stability in one update per message.

by proving that synchronous product and natural projection ensure the axioms of composition and projection described above, we prove that Algorithm 4 is correct.

Notice that a system is a language associated to a given alphabet, indeed, $\{\epsilon\}$ for example, has not the same properties for composition if it is a language over the empty alphabet or over another alphabet. We just give here the ideas of the proofs. Full proofs can be found in Appendix A.

Lemma 1 (axiom 2.1)**.** $\forall \Sigma_1, \Sigma_2,\ P_{\Sigma_1} \circ P_{\Sigma_2} = P_{\Sigma_1 \cap \Sigma_2}$.

Idea of the proof. Consider three alphabets Σ, Σ_1, Σ_2, and a word $w \in \Sigma^*$. Prove that $P_{\Sigma_1} \circ P_{\Sigma_2}(w) = P_{\Sigma_1 \cap \Sigma_2}(w)$, by considering the three possible cases for w ($w = \epsilon$, $w = \sigma \in \Sigma$, and $w = w'\sigma$ where $w' \in \Sigma^*$ and $\sigma \in \Sigma$) and applying the definition of the natural projection to them. □

Lemma 2 (axiom 2.2)**.** $\forall \mathcal{L},\ \exists \Sigma,\ P_\Sigma(\mathcal{L}) = \mathcal{L}$.

Idea of the proof. Consider a language $\mathcal{L}$ over an alphabet Σ, and a word w from $\mathcal{L}$. Show that $P_\Sigma(w) = w$ and thus $P_\Sigma(\mathcal{L}) = \mathcal{L}$, by considering the three possible cases for w and applying the definition of the natural projection to them. □

Lemma 3 (axiom 2.3)**.** $\forall \Sigma \supseteq \Sigma_1 \cap \Sigma_2,\ \forall \mathcal{L}_1 \subseteq \Sigma_1^*,\ \forall \mathcal{L}_2 \subseteq \Sigma_2^*,\ P_\Sigma(\mathcal{L}_1 \| \mathcal{L}_2) = P_\Sigma(\mathcal{L}_1) \| P_\Sigma(\mathcal{L}_2)$.

Idea of the proof. Consider two alphabets Σ_1 and Σ_2, and two languages over these alphabets $\mathcal{L}_1$ and $\mathcal{L}_2$. Consider another alphabet $\Sigma \supseteq \Sigma_1 \cap \Sigma_2$ and notice that:

$$P_\Sigma(\mathcal{L}_1 \| \mathcal{L}_2) = \{w \mid \exists u \in (\Sigma_1 \cup \Sigma_2)^*,\ w = P_\Sigma(u) \text{ and } P_{\Sigma_1}(u) \in \mathcal{L}_1 \text{ and } P_{\Sigma_2}(u) \in \mathcal{L}_2\};$$

and:

$$P_\Sigma(\mathcal{L}_1) \| P_\Sigma(\mathcal{L}_2) = \{w \in (\Sigma \cap (\Sigma_1 \cup \Sigma_2))^* \mid P_{\Sigma \cap \Sigma_1}(w) \in P_\Sigma(\mathcal{L}_1) \text{ and } P_{\Sigma \cap \Sigma_2}(w) \in P_\Sigma(\mathcal{L}_2)\}.$$

Finally prove that $P_\Sigma(\mathcal{L}_1 \| \mathcal{L}_2) \subseteq P_\Sigma(\mathcal{L}_1) \| P_\Sigma(\mathcal{L}_2)$ and $P_\Sigma(\mathcal{L}_1 \| \mathcal{L}_2) \supseteq P_\Sigma(\mathcal{L}_1) \| P_\Sigma(\mathcal{L}_2)$. □

Lemma 4 (axiom 2.4)**.** $\exists \mathbb{I},\ \forall \mathcal{L},\ \mathcal{L} \| \mathbb{I} = \mathcal{L}$.

Idea of the proof. Take as $\mathbb{I}$ the language $\{\epsilon\}$ over the empty alphabet. □

2.1.3 Message passing algorithm in terms of automata

In fact, languages can contain an infinite number of words. If we want our approach to be practical, we can not just work on (eventually infinite) sets of words. A good, and well known, finite representation of regular languages is given by (finite) automata. In this section we present a way to express composition and projection operations in terms of automata calculus, and prove that our operations are valid with respect to the theory.

We use the definitions of automata and synchronous product of automata given in Section 1.2.1. We also consider a special kind of transitions, called ϵ-transition, which can be fired without reading any symbol. An automaton is deterministic if it contains no ϵ-transition and no state such that two transitions with the same label start at this state. Two automata are said to be equivalent if they recognize the same language.

Given an automaton $\mathcal{A}$ and a set Λ of labels, we provide the following operations: $\mathrm{DET}(\mathcal{A})$ which returns the deterministic automaton equivalent to $\mathcal{A}$, $\mathrm{MIN}(\mathcal{A})$ which returns the smallest (in term of vertices) automaton equivalent to $\mathcal{A}$ (this operation only works on deterministic automata), and $\mathrm{EPS}(\mathcal{A}, \Lambda)$ which first replace in $\mathcal{A}$ all transitions with label $\alpha \in \Lambda$ by ϵ-transitions and then returns an equivalent automaton but without ϵ-transitions. There exists well know algorithms which realize these operations, see for example [CL99] or [Sak03].

Given $\mathcal{A}_1 = (S_1, T_1, f_1, s_1^0, \lambda_1, \Lambda_1, F_1)$ and $\mathcal{A}_2 = (S_2, T_2, f_2, s_2^0, \lambda_2, \Lambda_2, F_2)$ two automata and a set Λ of labels, we suggest the following operations for composition and projection:

$$\mathcal{A}_1 \wedge \mathcal{A}_2 = \mathrm{MIN}(\mathcal{A}_1 \|_a \mathcal{A}_2);$$

$$\Pi_\Lambda(\mathcal{A}_1) = \mathrm{MIN}(\mathrm{DET}(\mathrm{EPS}(\mathcal{A}_1, \Lambda_1 \setminus \Lambda))).$$

We now can prove that these operations satisfy the required axioms described above and, thus, that the MPA is usable when systems are automata. As for languages, we just give ideas of the proofs, the detailed ones are given in Appendix A. In the following the equality $\mathcal{A}_1 = \mathcal{A}_2$ means that $\mathcal{A}_1$ is isomorphic to $\mathcal{A}_2$, not only that they recognize the same language. The proofs are based on the following lemma:

Lemma 5. *The minimal deterministic automaton which recognizes a given language is unique (up to an isomorphism).*

The proof of this lemma can be found, for example, in [Sak03].

Lemma 6 (axiom 2.1)**.** $\forall \Lambda_1, \forall \Lambda_2, \Pi_{\Lambda_1} \circ \Pi_{\Lambda_2} = \Pi_{\Lambda_1 \cap \Lambda_2}$.

Idea of the proof. Given an automaton $\mathcal{A}$, prove that $\mathcal{L}(\Pi_{\Lambda_1} \circ \Pi_{\Lambda_2}(\mathcal{A})) = \mathcal{L}(\Pi_{\Lambda_1 \cap \Lambda_2}(\mathcal{A}))$ and conclude by Lemma 5. □

Lemma 7 (axiom 2.2)**.** $\forall \mathcal{A}$ *minimal deterministic*, $\exists \Lambda', \Pi_{\Lambda'}(\mathcal{A}) = \mathcal{A}$.

Idea of the proof. Consider a minimal deterministic automaton $\mathcal{A} = (S, T, f, s^0, \lambda, \Lambda, F)$. Take $\Lambda' = \Lambda$ and prove that $\mathcal{L}(\Pi_{\Lambda'}(\mathcal{A})) = \mathcal{L}(\mathcal{A})$. Conclude by Lemma 5. □

Lemma 8 (axiom 2.3)**.** $\forall \Lambda_3 \supseteq \Lambda_1 \cap \Lambda_2, \forall \mathcal{A}_1, \forall \mathcal{A}_2, \Pi_{\Lambda_3}(\mathcal{A}_1 \wedge \mathcal{A}_2) = \Pi_{\Lambda_3}(\mathcal{A}_1) \wedge \Pi_{\Lambda_3}(\mathcal{A}_2)$.

Idea of the proof. Consider two automata $\mathcal{A}_1 = (S_1, T_1, f_1, s_1^0, \lambda_1, \Lambda_1, F_1)$ and $\mathcal{A}_2 = (S_2, T_2, f_2, s_2^0, \lambda_2, \Lambda_2, F_2)$. Prove that $\mathcal{L}(\Pi_{\Lambda_3}(\mathcal{A}_1 \wedge \mathcal{A}_2)) = \mathcal{L}(\Pi_{\Lambda_3}(\mathcal{A}_1) \wedge \Pi_{\Lambda_3}(\mathcal{A}_2))$, then conclude by Lemma 5. □

Lemma 9. $\exists \mathbb{I}, \forall \mathcal{A}, \mathcal{A} \wedge \mathbb{I} = \mathcal{A}$.

Idea of the proof. Take $(\{s_0\}, \emptyset, f, s_0, \lambda, \emptyset, \{s_0\})$ as $\mathbb{I}$. □

Notice that these proofs are done with a very strong notion of equality between automata (same states and transitions, up to an isomorphism). In fact, we could change the notion of equality. The equality of languages, for example, is sufficient, thus one could use non-minimal automata, or even non-deterministic ones, and prove that when composition is the synchronous product of automata and projection consists only in replacing symbols by epsilon and removing ϵ-transitions, then the axioms are also preserved, thus MPA can be used and will work. We decided to use deterministic automata because synchronous product of deterministic automata can be much more simple than synchronous product of non-deterministic ones (even if determinising automata can potentially have exponential cost). In practice it is risky to determinise automata. It would be a good idea to test both deterministic automata and non-deterministic automata to see if, for our problems, it is generally more interesting to determinise or not. Once we decided to use deterministic automata it was logic to use minimal deterministic automata, to be able to handle the largest systems as possible.

In this section we presented the principles of a method for factored planning, based on an algorithm: the message passing algorithm. We shown that these principles were relevant to find plans in factored planning problems represented as languages, as suggested in Section 1.2.3, and can be implemented in terms of automata. In the following we explain how our method can be used to perform factored optimal planning, where systems are weighted languages, implemented in terms of weighted automata.

2.2 Factored Optimal Planning

In this section we show that the MPA can be used in the case of weighted languages and, thus, that this algorithm allows to perform factored optimal planning. In fact, as classical languages, weighted languages are infinite sets, so, to make our approach practical we propose (and prove) an implementation in terms of weighted automata calculus of the operations on weighted languages we give. This implementation is based on the work from Mohri [Moh09].

2.2.1 From factored planning to factored optimal planning

We first define the notion of weighted automaton:

Definition 14. *A* weighted automaton *(WA) is a tuple $\mathcal{A} = (S, T, f, s^0, \lambda, \Lambda, F, c, c^i, c^F)$ where $(S, T, f, s^0, \lambda, \Lambda, F)$ is an automaton, $c : T \to \mathbb{R}^+$ associates a cost to each transition, $c^i \in \mathbb{R}^+$ is an initial cost, and $c^F : F \to \mathbb{R}^+$ associates a cost to each final state.*

Notice that we assume that our WA have only one initial state. This is because in our applications we do not need to have several initial states (anyway, any WA with more than one initial state is equivalent – with respect to weighted languages – to a WA with only one initial state). However, all what we describe below works with WA which have several initial states.

A path $\pi = s_1, t_1, s_2, \dots, t_{k-1}, s_k$ is a sequence of states and transitions such that $\forall i \in \{1, \dots, k-1\}$ we have $(s_i, t_i) \in f$ and $(t_i, s_{i+1}) \in f$. We denote by $\lambda(\pi) = \lambda(t_1), \dots, \lambda(t_2)$ the word corresponding to π. The cost of this path is defined as $c(\pi) = \sum_{i=1}^{k} c(t_i)$. We also use the following notation: $c^F(\pi) = c^F(s_k)$ (assuming that $c^F(s) = 0$ for $s \notin F$). The path π is accepted by $\mathcal{A}$, noted $\pi \models \mathcal{A}$, if and only if $s_1 = s_0$ and $s_k \in F$.

The weighted language of a WA $\mathcal{A} = (S, T, f, s^0, \lambda, \Lambda, F, c, c^i, c^F)$ is defined by:

$$\mathcal{L}(\mathcal{A}) = \{(u, w) \in \Lambda^* \times \mathbb{R}^+ \mid \exists \pi \models \mathcal{A},\ u = \lambda(\pi),\ w = \min_{\substack{\pi \models \mathcal{A} \\ u = \lambda(\pi)}} c^i + c(\pi) + c^F(\pi)\}$$

In other words, the sequence of actions u is in the language if there is an accepted path which produces it and its weight is the minimal weight over all accepted paths that produce u.

As for classical automata, we define the synchronous product of weighted automata:

Definition 15. *The* synchronous product $\mathcal{A}_1 \times_a \mathcal{A}_2$ *of the weighted automata $\mathcal{A}_1 = (S_1, T_1, f_1, s_1^0, \lambda_1, \Lambda_1, F_1, c_1, c_1^i, c_1^F)$ and $\mathcal{A}_2 = (S_2, T_2, f_2, s_2^0, \lambda_2, \Lambda_2, F_2, c_2, c_2^i, c_2^F)$ is $\mathcal{A} = (S, T, f, s^0, \lambda, \Lambda, F, c, c^i, c^F)$ defined by:*

- *$(S, T, f, s^0, \lambda, \Lambda, F)$ is the synchronous product of $(S_1, T_1, f_1, s_1^0, \lambda_1, \Lambda_1, F_1)$ and $(S_2, T_2, f_2, s_2^0, \lambda_2, \Lambda_2, F_2)$ as defined in Section 1.2.1;*
- *$\forall t = (t_1, t_2) \in T, c(t) = c_1(t_1) + c_2(t_2)$, with the convention that $c_i(\star_i) = 0$;*
- *$c^i = c_1^i + c_2^i$;*
- *$\forall s = (s_1, s_2) \in F, c^F(s) = c_1^F(s_1) + c_2^F(s_2)$.*

Instead of considering a network of automata, as described in Section 1.2.1, we now consider a network of weighted automata $\mathcal{A} = \mathcal{A}_1 \times_a \dots \times_a \mathcal{A}_n$. As before each automata is a component of our problem. The only difference is that actions have now costs (initial and final costs are taken to be null). The *factored optimal planning problem* consists in finding an accepted path π_i in each $\mathcal{A}_i$ such that these paths are compatible (as defined for factored planning) and $\sum_{i=1}^{n} c(\pi_i)$ is minimal. Hence, it consists in finding the word(s) of minimal weight in $\mathcal{L}(\mathcal{A})$, without computing the full compound system $\mathcal{A}$ nor $\mathcal{L}(\mathcal{A})$.

2.2.2 Handling weighted languages

We presented the factored optimal planning problem. We now show how this problem can be solved, using the MPA with weighted languages (e.g. sets of couples (u, w) where u is a word and w is a weight) as systems. In fact, as for (non-weighted) languages we just have to check that the axioms required (axioms 2.1, 2.2, 2.3, and 2.4) are satisfied by such systems with the operations we propose. In this case, in the conditions of utilization of the MPA (when the communication graphs are trees), the results of the algorithm are all local plans (associated to costs) that are part of global plans, and only these plans. Moreover, the costs associated to local plans are the costs of the best global plans they are involved in [Fab03]. Thus, the MPA allows us to perform factored optimal planning.

As composition we take the synchronous product of weighted languages:

Definition 16. *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two weighted languages defined on Σ_1 and Σ_2 respectively. Their* synchronous product *is given by:*

$$\mathcal{L}_1 \times_\ell \mathcal{L}_2 = \{(u, w) \in (\Sigma_1 \cup \Sigma_2)^* \times \mathbb{R}^+ \mid \exists (u_{|\Sigma_1}, w_1) \in \mathcal{L}_1,\ \exists (u_{|\Sigma_2}, w_2) \in \mathcal{L}_2,\ w = w_1 + w_2\}.$$

Notice that this synchronous product is well defined: given u, there is at most one $(u_{|\Sigma_i}, w_i)$ in $\mathcal{L}_i$, and so at most one cost w for u. Moreover, as required this product is commutative and associative. It is also easy to show that the weighted language $\mathcal{L}(\mathcal{A})$ recognized by a product $\mathcal{A} = \mathcal{A}_1 \times_a \ldots \times_a \mathcal{A}_n$ of WA is the product of the languages recognized by the components: $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \times_\ell \ldots \times_\ell \mathcal{L}(\mathcal{A}_n)$.

As projection we take the natural projection of weighted languages:

Definition 17. *Let $\mathcal{L} \subseteq \Sigma^* \times \mathbb{R}^+$ be a weighted language and $\Sigma' \subseteq \Sigma$ be an alphabet. The* natural projection *of $\mathcal{L}$ over Σ' is given by:*

$$P_{\Sigma'}(\mathcal{L}) = \{(u', w') \in \Sigma'^* \times \mathbb{R}^+ \mid \exists (u, w) \in \mathcal{L},\ u' = u_{|\Sigma'}, w' = \min_{\substack{(u,w) \in \mathcal{L} \\ u_{|\Sigma'} = u'}} w\}.$$

Notice that a minimization is performed on eliminated letters: this is where is the optimization in the MPA, as in the example about optimization under constraints presented in Section 2.1.1.

We now show that the two operations described above preserve the axioms of Section 2.1.1. The proof of axiom 2.3 is huge, so we just give here idea of it. The full proof can be found in Appendix A.

Lemma 10 (axiom 2.1)**.** $\forall \Sigma_1, \Sigma_2,\ P_{\Sigma_1} \circ P_{\Sigma_2} = P_{\Sigma_1 \cap \Sigma_2}$.

Proof. Consider $\mathcal{L}$ a weighted language over the alphabet Σ. By Lemma 1 we know that for all $(u, w) \in P_{\Sigma_1} \circ P_{\Sigma_2}(\mathcal{L})$ there is w' such that $(u, w') \in P_{\Sigma_1 \cap \Sigma_2}(\mathcal{L})$, conversely for all $(u, w') \in P_{\Sigma_1 \cap \Sigma_2}(\mathcal{L})$ there is w such that $(u, w) \in P_{\Sigma_1} \circ P_{\Sigma_2}(\mathcal{L})$. Then we just prove that for all u we have $w = w'$ by remarking that:

$$\min_{\substack{(u', \bullet) \in P_{\Sigma_2}(\mathcal{L}) \\ u'_{|\Sigma_1} = u}} \left(\min_{\substack{(u'', w'') \in \mathcal{L} \\ u''_{|\Sigma_2} = u'}} w'' \right) = \min_{\substack{(u', w') \in \mathcal{L} \\ u'_{|\Sigma_1 \cap \Sigma_2} = u}} w'.$$

□

Lemma 11 (axiom 2.2)**.** $\forall \mathcal{L},\ \exists \Sigma,\ P_\Sigma(\mathcal{L}) = \mathcal{L}$.

Proof. The proof is similar that the one for non-weighted languages (Lemma 2): when Σ is the alphabet of $\mathcal{L}$ it holds that $\forall (u, w) \in \mathcal{L},\ u_{|\Sigma} = u$ and, thus, minimization is done over one element sets. Hence if $(u, w) \in L$ then $(u, w) \in P_\Sigma(\mathcal{L})$ and conversely. □

Lemma 12 (axiom 2.3)**.** $\forall \Sigma \supseteq \Sigma_1 \cap \Sigma_2,\ \forall \mathcal{L}_1 \subseteq \Sigma_1^*,\ \forall \mathcal{L}_2 \subseteq \Sigma_2^*,\ P_\Sigma(\mathcal{L}_1 \times_\ell \mathcal{L}_2) = P_\Sigma(\mathcal{L}_1) \times_\ell P_\Sigma(\mathcal{L}_2)$.

Idea of the proof. From Lemma 3 we know that $\forall (u, w) \in P_\Sigma(\mathcal{L}_1 \times_\ell \mathcal{L}_2)$ there is w' such that $(u, w') \in P_\Sigma(\mathcal{L}_1) \times_\ell P_\Sigma(\mathcal{L}_2)$, and $\forall (u, w') \in P_\Sigma(\mathcal{L}_1) \times_\ell P_\Sigma(\mathcal{L}_2)$ there is w such that $(u, w) \in P_\Sigma(\mathcal{L}_1 \times_\ell \mathcal{L}_2)$. We just have to prove that $w = w'$, which is done by remarking that the optimization made by projection can be done independently on words from $\mathcal{L}_1$ and $\mathcal{L}_2$. □

Lemma 13 (axiom 2.4)**.** $\exists \mathbb{I},\ \forall \mathcal{L},\ \mathcal{L} \| \mathbb{I} = \mathcal{L}$.

Proof. It is sufficient to notice that 0 is the identity element for the addition. Then taking the weighted language $\{(\epsilon, 0)\}$ over the empty alphabet as $\mathbb{I}$ leads to a proof similar than for non-weighted languages (Lemma 4). $\square$

We proved that our operations on weighted languages satisfy the four axioms required for the MPA. Hence, from [Fab03] we know that the weighted languages obtained by the MPA from a compound system contain exactly the local plans that can be extended into global plan. Moreover any optimal local plan can be extended into an optimal global plan. Using the MPA on weighted languages we perform factored optimal planning.

2.2.3 Message passing algorithm in terms of weighted automata calculus

In fact, as classical languages, weighted languages are potentially infinite sets. To make our approach of factored optimal planning practical we have to find a finite representation of these sets. The weighted languages which are relevant for us are fully captured by weighted automata.

In this section we present how the operations on weighted languages we proposed above can be represented in terms of weighted automata calculus. In particular, this section is based on the work from Mohri on weighted automata algorithms [Moh09].

It would be possible to just implement synchronous product of weighted languages (e.g. composition) as synchronous product of weighted automata and natural projection of weighted languages (e.g. projection) as ϵ-reduction of weighted automata (which is basically ϵ-reduction with weighted ϵ-transitions, this operation is described below). In this case the weighted automata involved will potentially be non-deterministic. But, deterministic weighted automata are very useful for us: their weighted languages are much more easy to compute that the ones of non-deterministic automata. Indeed, each word is recognized only once, thus, is associated to exactly one weight: the minimization in the definition of the weighted language of an automaton is performed by the determinisation operation.

The operations presented below assume that we work with deterministic weighted automata. It is easy to see that these operations implements exactly the operations wanted on weighted languages. Hence, we focus more on the description on these operation than on the proofs of them.

Composition As we suggested the composition operation can be just a synchronous product. But, assuming that our automata are deterministic and knowing that the synchronous product of deterministic automata gives a deterministic automaton, we would like to have the smallest automaton for the composition of two automata. That is why we take for composition of $\mathcal{A}_1$ and $\mathcal{A}_2$ the minimal deterministic automaton which recognize the same language as $\mathcal{A}_1 \times_a \mathcal{A}_2$.

The minimization of deterministic WA can be done in a canonical manner, as described in [Moh09] and proved in [Moh94]. One first apply a weight pushing algorithm, described formally in Appendix B, which pushes the weights of each path as much as possible toward the initial state. Then the classical minimization algorithm [CL99] is applied to the resulting automaton, considering each pair (label, weight) as a single label.

One can notice that the cost associated to initial state of weighted automata (c^i) is necessary, due to this minimization procedure. The complexity of minimization is $O(|T| \log |S|)$ for a weighted automaton $\mathcal{A} = (S, T, f, s^0, \lambda, \Lambda, F, c, c^i, c^F)$. The identity element (necessary for the MPA computations) of this composition is the automaton $\mathbb{I} = (\{s^0\}, \emptyset, f, s^0, \lambda, \emptyset, \{s^0\}, c, c^i, c^F)$ with $c^i = c^F(s^0) = 0$. The relations f, λ, and c do not need to be defined (in fact they are empty because defined on empty sets).

Projection As composition, the projection $\Pi_{\Lambda'}(\mathcal{A})$ of $\mathcal{A} = (S, T, f, s^0, \lambda, \Lambda, F, c, c^i, c^F)$ over an alphabet Λ' can be implemented easily in terms of WA, as an operation called *ϵ-reduction.* First each transition labeled by a symbol from $\Lambda \setminus \Lambda'$ is replaced by a transition labeled by ϵ, with the same weight as the old transition. After that an equivalent WA without ϵ-transitions is constructed. This can be done using an ϵ-removal algorithm described in [Moh09]. Given a WA $waa1$ with ϵ-transitions, for $s \in S_1$ we call invisible reach the subset $\mathcal{R}(s)$ of the states that can be reached from s in $\mathcal{A}_1$ using only ϵ-transitions. The automaton $\mathcal{A}_2 = (S_2, T_2, f_2, s_2^0, \lambda_2, \Lambda_2, F_2, c_2, c_2^i, c_2^F)$ resulting of ϵ-removal applied to $\mathcal{A}_1$ has the same states as $\mathcal{A}_1$ ($S_2 = S_1$), the same initial state ($s_2^0 = s_1^0$), and its alphabet is $\Lambda_2 = \Lambda_1$. The initial cost also remains the same: $c_2^i = c_1^i$. The transitions T_2, associated labels λ_2 and costs c_2, and the flow relation f_2, are defined in the following way: $t_2 \in T_2$ with label $\lambda_2(t_2) = \sigma$, cost $c_2(t_2) = c + c'$, and such that (s, t_2) and (t_2, s') are in the flow relation f_2 if and only if $\exists s'' \in \mathcal{R}(s)$ and $t_1 \in T_1$ such that c' is the smallest weight among the silent paths (paths with only ϵ-transitions) from s to s'', the flow relation f_1 contains (s'', t_1) and (t_1, s'), the label corresponding to t_1 is $\lambda_1(t_1) = \sigma$, and the corresponding cost is $c_1(t_1) = c$. The set of final states F_2 is the set of all $s \in S_1$ such that $\mathcal{R}(s) \cap F_1 \neq \emptyset$. The cost of a final state $s \in F_2$ is the smallest weight among the silent paths starting at s and finishing in $\mathcal{R}(s) \cap F_1$. In Appendix B we give this *epsilon*-removal algorithm. It has a complexity of $O(|S_1|^2 + |S_1||T_1|)$.

It is easy to see that ϵ-reduction does not keep the automata deterministic. Hence, to work with deterministic WA we have to determinise. The Projection operation thus becomes:

1. ϵ-reduction,
2. determinisation of the resulting automaton,
3. minimization of the result (as described above).

The determinisation of WA is responsible for the optimization part of our algorithm. Moreover, even if, in the worst case, the determinisation of $\mathcal{A}$ can return an automaton of size exponential in $|\mathcal{A}|$ – as for non-weighted automata – it is rarely the case. In fact the determinisation procedure we describe below often produces deterministic WA smaller than the input WA [Moh97]. Unfortunately there is also some drawbacks: not all WA can be determinised. In the following we describe a procedure to determinise WA and explain in which cases WA can not be determinised. After that we give a solution to avoid the problem of non-determinisable WA.

Determinisation of weighted automata For a WA $\mathcal{A}$ we say that we determinise $\mathcal{A}$ when we provide a new WA which is finite, deterministic, and recognizes exactly the same weighted language as $\mathcal{A}$. One can remark that not all WA are determinisable. See for example Figure 2.3. The accepted words in the automaton of this figure are $c\{a, b\}^*$. One either pay for the a or the b, depending on the path selected for the first c. The weight of an accepted word w is thus $\min(|w_{|\{a\}}|, |w_{|\{b\}}|)$. Intuitively, constructing a deterministic automaton which recognize exactly this weighted language corresponds to construct an automaton which counts the a and the b. This automaton can not be finite, hence the automaton of Figure 2.3 can not be determinised.

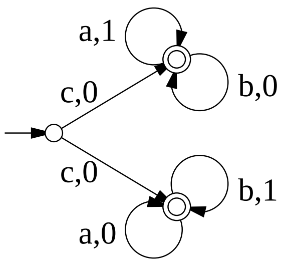

Figure 2.3: A WA that can not be determinised

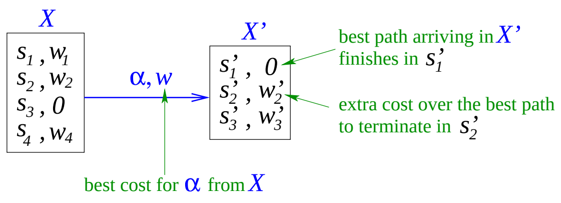

Figure 2.4: Construction of a transition in the determinisation procedure

A sufficient condition for determinisability of WA is the *twin property*. This condition stipulates that when states s and s' can be reached by the same label sequence, and moreover it is possible to loop around s and around s' with the same label sequence u, then these loops must have identical weights:

Definition 18. *In a WA $\mathcal{A} = (S, T, f, s^0, \lambda, \Lambda, F, c, c^i, c^F)$, two states $s, s' \in S$ are said to be* twins *if and only if, either $\nexists v \in \Lambda^*$ such that there is a path from s^0 to s and a path from s^0 to s', both labeled by v; or $\forall u \in \Lambda^*$ such that both s and s' are reachable from s^0 by a path labeled by u, one has:*

$$\min_{\substack{\pi,\ \lambda(\pi)=u \\ \pi \text{ loops on } s}} c(\pi) = \min_{\substack{\pi',\ \lambda(\pi')=u \\ \pi' \text{ loops on } s'}} c(\pi').$$

The WA $\mathcal{A}$ is said to be twin *if and only if all its states are twins.*

This twin property can be checked in polynomial time [Moh09]. There is an algorithm which determinise all the WA which have the twin property [Moh09]. We just give here an idea of this algorithm, which is fully described in Appendix B.

This algorithm, which determinise any twin WA $\mathcal{A} = (S, T, f, s^0, \lambda, \Lambda, F, c, c^i, c^F)$, is based on the classical idea of subset construction [CL99]. In fact, the states of the WA constructed by the algorithm are pairs (E, g) where $E \subseteq S$ and $g : E \to \mathbb{R}^+$. In the unique state $X = (E, g)$ reached by the word u the element E denotes the set of all states reachable from s^0 by paths π such that $\lambda(\pi) = u$. For $s \in E$, $g(s)$ is the difference between two costs: 1)the minimal cost over all π starting at s^0, reaching an element of E, and such that $\lambda(\pi) = u$ and 2)the minimal cost to reach s from s^0 by a path labeled by u. Hence, there is always $s \in E$ such that $g(s) = 0$. The successors of X can be built recursively. Consider a transition t from X to $X' = (E', g')$ with label $\sigma \in \Lambda$ and cost c_σ. The set E' contains all the states from S that can be reached from E by a transition labeled by σ. The weight c_σ is the difference of the minimal weight for word $u\sigma$ and the minimal weight for word u. In fact one has:

$$c_\sigma = \min_{s \in E,\ t \in T \text{ s.t. } \lambda(t) = \sigma,\ (s,t) \in f} g(s) + c(t),$$

and, for $s' \in E'$:

$$g'(s') = \min_{s \in E,\ t \in T \text{ s.t. } \lambda(t) = \sigma,\ (s,t) \in f,\ (t,s') \in f} g(s) + c(t) - c_\sigma.$$

The construction of such a transition is represented in Figure 2.4. In fact this procedure to determinise WA is only ensured to converge when $\mathcal{A}$ has the twin property. However, this property is not necessary for determinisability of general WA (in Appendix B we discuss other conditions for determinisability of WA), and some determinisable WA are not determinised by the procedure described above. See for example Figure 2.5. On the left is a WA $\mathcal{A}$ (its initial state is 1), on the center is an equivalent deterministic finite automaton, and on the right is the infinite result obtained by applying the procedure we described to $\mathcal{A}$. For matter of place a state $X = (\{e, e'\}, f)$ is denoted by $e_{f(e)}, e'_{f(e')}$.

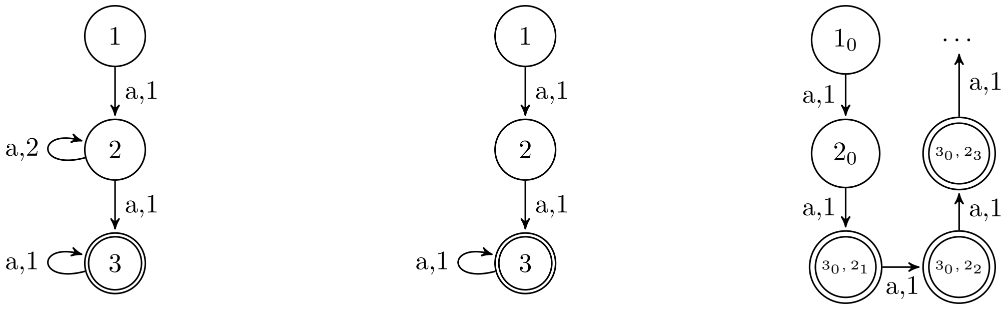

Figure 2.5: A determinisable WA (left) with which Mohri's algorithm does not terminate

As we explained above, to ensure the convergence of the determinisation procedure for WA described above we need them to have the twin property. Unfortunately, even if we assume that, for a system $\mathcal{A} = \mathcal{A}_1 \times_a \ldots \times_a \mathcal{A}_n$, all the subsystems $\mathcal{A}_i$ have the twin property, we need this property to be preserved by the operations we do. And if the twin property is clearly preserved by synchronous product, this is not the case with ϵ-reduction. See the example of Figure 2.3 where one of the $(c, 0)$ would be a $(d, 0)$. This automaton has the twin property. After ϵ-reduction on $\{a, b\}$ it is clear that the automaton has no longer the twin property.

To avoid these problems in determinisation one could perform *partial determinisation*. Indeed, for a WA $\mathcal{A} = (S, T, f, s_0, \lambda', \Lambda, F', c', c_i, c_F)$, any representant of the weighted language $\mathcal{L}(\mathcal{A})$ can be used in the computations of the MPA. Therefore, it is not necessary to complete the determinisation of $\mathcal{A}$: one can stop at any point, while it is terminated in such a way that the language $\mathcal{L}(\mathcal{A})$ is preserved. Let $\mathcal{A}' = (S', T', f', s'_0, \lambda', \Lambda, F', c', c'_i, c'_F)$ be a part of the determinisation of $\mathcal{A}$, e.g. the automaton produced by the determinisation procedure when we stop it before it terminates. We build $\mathcal{A}'' = (S'', T'', f'', s''_0, \lambda'', \Lambda, F'', c'', c''_i, c''_F)$ by connecting $\mathcal{A}$ (where all initial states become normal states) to $\mathcal{A}'$ in the following manner: for $X = (E, g) \in S'$, $s \in S$, and $\sigma \in \Lambda$ there is a transition $t'' \in T''$ such that $(X, t'') \in f''$, $(t'', s) \in f''$, and $\lambda''(t'') = \sigma$ if and only if there is a path labeled by σ from an element of E to s in $\mathcal{A}$. The cost associated to this transition is:

$$c''(t'') = \min_{s' \in E,\ t \in T \text{ s.t. } \lambda(t)=\sigma,\ (s',t)\in f,\ (t,s)\in f} g(s') + c(t).$$

The finite WA obtained satisfies $\mathcal{L}(\mathcal{A}'') = \mathcal{L}(\mathcal{A})$, so it can replace $\mathcal{A}$ in computations. The advantage is that words of $\mathcal{A}''$ start in the determinised part $\mathcal{A}'$. Hence, for short words, operations like product are easy to do (linear complexity).

This partial determinisation becomes really interesting if the determinisation procedure is driven in order to ensure that any word from $\mathcal{L}(\mathcal{A})$ with weight lower than some bound W is in $\mathcal{L}(\mathcal{A}')$. In this case, if one can ensure that the optimal words we are looking for have weight smaller than W it is possible to use only $\mathcal{A}'$ (the deterministic part of $\mathcal{A}''$) in the computations. Moreover it is clear that there exists, in any compound system $\mathcal{A} = \mathcal{A}_1 \times_a \ldots \times_a \mathcal{A}_n$, a bound on the weight of optimal global plans (notice for example that a global plan is a path in $\mathcal{A}$ and that an optimal path in $\mathcal{A}$ can not contain cycles, moreover, the maximal size of a path without cycle in $\mathcal{A}$ can be estimated in function of the number of shared symbols).

In this section we explained how the MPA presented above can be used to solve the factored optimal planning problem. The method we proposed is to consider each local subproblem of a large global problem as a WA. We gave operations on WA which verify the axioms necessary for the MPA and made some remarks on how it is possible to pass besides the drawbacks of these operations (in particular determinisation).

2.3 Example

This section gives an example of application of the MPA for factored optimal planning. This example is close to a real planning problem. It demonstrates how our algorithm could be used in real cases by first giving the modelisation of a concrete problem as a network of weighted automata and then showing how this problem can be solved using the MPA.

We consider a problem where a truck has to transport products between different sites: production sites and warehouses. The truck, each warehouse and each production site have maximum storage capacity. The truck has the possibility to move from site i to site j (M_{ij}), where the precise moves and their costs depend on the road network, the truck's load, or any other condition. The truck can also load a unit of product from production site i (L_i) and unload a unit to warehouse i (U_i), while the maximum storage capacities are respected. Production site i can also produce one unit of product (P_i), under the constraints of storage capacity. Production cost depends on the current number of units of product stored at the production site. The staff at a production site can also influence the cost of production and load by being ready for one task or another.

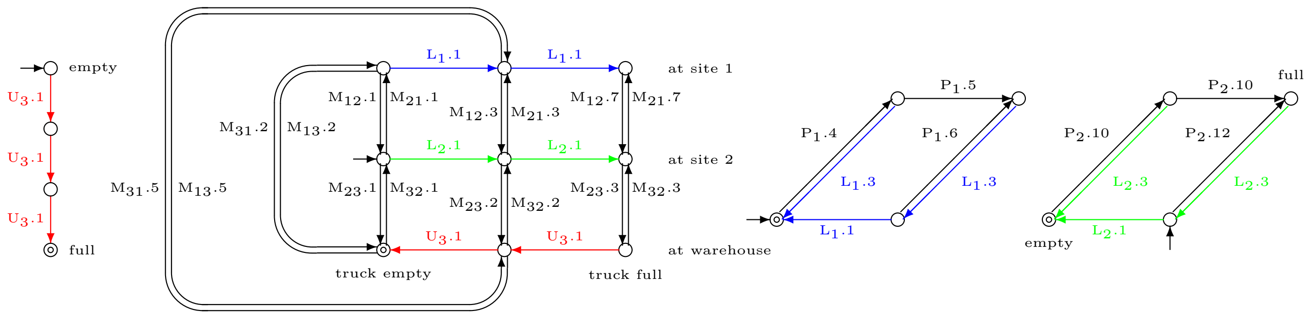

Figure 2.6: Warehouse (left), truck (middle), and two production sites (right).

Figure 2.6 shows the weighted automata corresponding to such a problem involving a truck (T), two production sites (P_1 and P_2) and a warehouse (W). The shared labels and the corresponding transitions are colored. Each production site has a storage capacity of 2, as the truck, and the warehouse has a storage capacity of 3. Initially the truck is at production site 2, which has one unit of product stored and ready to be loaded. The other sites are empty. The goal is to fill the warehouse. At the end production sites have to be empty and the truck has to be empty and at the warehouse. Notice that when the truck is full it can not go from production site 1 to warehouse directly.

The communication graph of our problem has a star shape, with the truck in the center and the production sites and the warehouse connected to it, Figure 2.7. So the MPA requires six message computations.

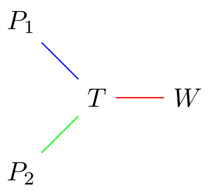

Figure 2.7: Communication graph

First, consider the computation of $\mathcal{M}_{1,T}$, the message from P_1 to T. $\mathcal{M}_{1,T}$ is the projection of P_1 on the load actions, as these are shared with the truck. The steps of this projection appear in Figure 2.8.

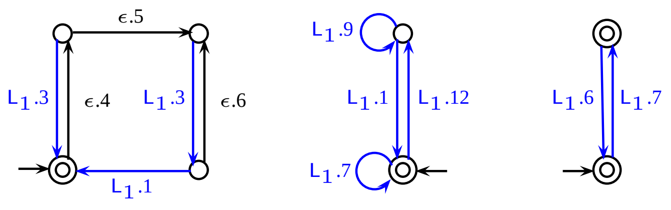

Figure 2.8: P_1 with private labels replaced by ϵ (left), after epsilon-removal (center), and finally after determinisation and minimisation (right).

One can obtain in a similar manner the two other messages to the truck, $\mathcal{M}_{2,T}$ and $\mathcal{M}_{W,T}$, from the second production site and from the warehouse, respectively (Figure 2.9). Notice that final states have termination costs (here 1 or 3).

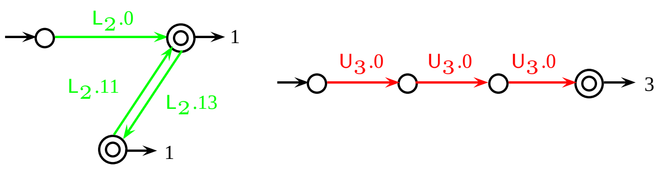

Figure 2.9: Messages from P_2 to T (left) and W to T (right).

Then the truck updates its own messages $\mathcal{M}_{T,1}$, $\mathcal{M}_{T,2}$, and $\mathcal{M}_{T,W}$ (Figure 2.10). Notice that these updates propagate the constraints imposed by P_1, P_2, and W. For example, only three U_3 are allowed by $\mathcal{M}_{W,T}$, hence the messages $\mathcal{M}_{T,1}$ and $\mathcal{M}_{T,2}$ allow at most three load actions.

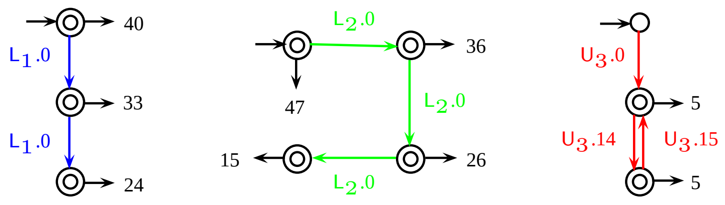

Figure 2.10: Messages from T to P_1 (left), P_2 (center), and W(right).

At this point messages are stable and the reduced components P'_1, P'_2, W' (Figure 2.11), and T' (not represented) can be derived. The following optimal local plans can easily be found: $\mathsf{P}_1\mathsf{P}_1\mathsf{L}_1\mathsf{L}_1$ in P'_1, L_2 in P'_2, and $\mathsf{U}_3\mathsf{U}_3\mathsf{U}_3$ in W'. Automaton T' is too large to be represented here, so we just give one of its optimal local plans: $\mathsf{M}_{21}\mathsf{L}_1\mathsf{M}_{12}\mathsf{L}_2\mathsf{M}_{23}\mathsf{U}_3\mathsf{U}_3\mathsf{M}_{31}\mathsf{L}_1\mathsf{M}_{12}\mathsf{M}_{23}\mathsf{U}_3$. It is easy to see that the four optimal local plans given above can be synchronized into a globally optimal plan of cost 37. Note that these four local plans can be synchronized in different ways: for example the optimal global plan could indifferently start with $\mathsf{P}_1\mathsf{P}_1\mathsf{M}_{21}$, with $\mathsf{P}_1\mathsf{M}_{21}\mathsf{P}_1$, or with $\mathsf{M}_{21}\mathsf{P}_1\mathsf{P}_1$.

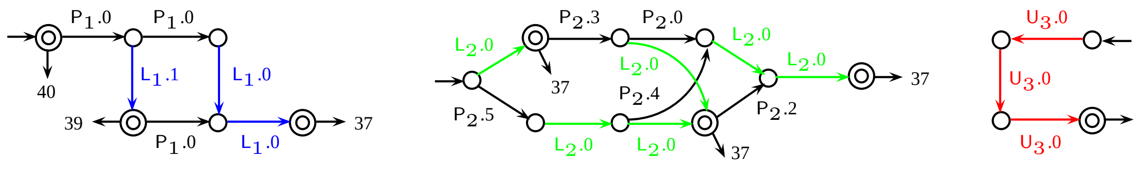

Figure 2.11: MPA output: P'_1 (left), P'_2 (center), and W'(right).

A possible corresponding optimal global plan is the following: producer 1 produces two units of product. Then the truck moves to producer 1, loads once, returns at producer 2, loads once, goes to the warehouse, unloads twice, returns to production site 1, loads once and then goes to the warehouse via production site 2, and unloads.

Using our implementation of the MPA on top of the AT&T FSM Library [MPR], the largest automaton generated at any point of the MPA is T', with 51 states and 124 transitions. The full system $S = P_1 \wedge P_2 \wedge T \wedge W$ has 576 states and 2176 transitions. It can be minimized to 123 states and 374 transitions, which remains more than twice as large as T'.

In this chapter we described the message passing algorithm from [Fab07] and explained how it can be used to perform factored optimal planning (and even distributed optimal planning, due to the principle of the algorithm). For that we introduced weighted automata calculus [Moh09] to implement the operations necessary for the MPA.

In the next chapter we present another way to perform factored optimal planning, based on the A^* algorithm.

Chapter 3

A Distributed A^* Algorithm

In this chapter we introduce a new algorithm to solve factored optimal planning problem. The idea is to represent a factored planning problem as a network of automata. On each automaton an agent has to find a path to the goal, in order to ensure that the paths finded by all the agents are compatible and that their combination is optimal. In fact, the algorithm we present here can be seen as a distributed A^* algorithm. Indeed, each agent uses an algorithm close to A^* to find its local path: as in A^* algorithm there is a notion of heuristic but here it is provided to an agent by its neighbors and can change along the time.

We first describe our algorithm in the simple case where the considered factored optimal planning problem involves only two automata (and thus two agents). After that we explain how it could be generalized to any factored optimal planning problem, while the communication graphs are trees.

3.1 A Simple Case Involving Two Automata

In this section we present our algorithm on a factored optimal planning problem involving two automata $\mathcal{A}_1 = (S_1, T_1, f_1, s_1^0, \lambda_1, \Lambda_1, F_1, c_1, c_1^i, c_1^F)$ and $\mathcal{A}_2 = (S_2, T_2, f_2, s_2^0, \lambda_2, \Lambda_2, F_2, c_2, c_2^i, c_2^F)$, and thus two agents φ_1 and φ_2. The communication graphs of such a problem are clearly trees, as represented in Figure 3.1.

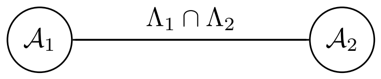

Figure 3.1: A communication graph

The principle of our algorithm is very close to A^*: each agent φ_i maintains a function $g_i : S_i \times (\Lambda_1 \cap \Lambda_2)^* \to \mathbb{R}^+ \cup \{+\infty\}$. This function is such that $g_i(v, w)$ gives the best cost known to go from s_i^0 to $v \in S_i$ by paths π such that $\lambda_i(\pi)_{|\Lambda_1 \cap \Lambda_2} = w$. For any (v, w), the value $g(v, w)$ is initialized to $+\infty$. As for the classical A^* algorithm, this function is updated along the algorithm. Agent φ_i also knows an heuristic function $h_i : S_i \to \mathbb{R}^+ \cup \{+\infty\}$ which is such that $h_i(v)$ is a lower bound on the cost of a path in $\mathcal{A}_i$ from $v \in S_i$ to F_i. Moreover we assume that φ_i has access to two functions, caracterising the other agent φ_j: a function $H_j : (\Lambda_1 \cap \Lambda_2)^* \to \mathbb{R}^+ \cup \{+\infty\}$ and a function $C_j : (\Lambda_1 \cap \Lambda_2)^* \to \mathbb{R}^+ \cup \{+\infty\}$:

- H_j is such that $H_j(w)$ is a lower bound on the cost of accepted paths π_j in $\mathcal{A}_j$ such that $\lambda_j(\pi_j)_{|\Lambda_1 \cap \Lambda_2} = w'$ with w a prefix of w' (e.g. a lower bound on the cost of accepted paths in $\mathcal{A}_j$ that could synchronize with an accepted word in $\mathcal{A}_i$ which has w as a prefix);
- C_j is such that there exists a time t such that after t the value $C_j(w)$ is stable and is the optimal cost of an accepted path π_j in $\mathcal{A}_j$ such that $\lambda_j(\pi_j)_{|\Lambda_1 \cap \Lambda_2} = w$, we assume that φ_i knows when $C_j(w)$ is stable.

Each agent φ_i also have access to a queue Q_i which contains couples (v, w) where $v \in S_i$ and $w \in (\Lambda_1 \cap \Lambda_2)^*$ and two sorts of special elements: $o(w)$ with $w \in (\Lambda_1 \cap \Lambda_2)^*$ and $\tilde{o}(w)$ with $w \in (\Lambda_1 \cap \Lambda_2)^*$. This queue has the following properties:

- $o(w)$ is in Q_i if and only if $C_i(w)$ and $C_j(w)$ are both stable, the ranking-cost associated to $o(w)$ is $C_i(w) + C_j(w)$;
- $\tilde{o}(w)$ is in Q_i if and only if an accepted word π_i such that $\lambda_i(\pi_i)_{|\Lambda_1 \cap \Lambda_2}$ has been found in $\mathcal{A}_i$ (which means that $g_i(v,w) \neq +\infty$ for some $v \in F_i$) and $o(w)$ is not in Q_i, the ranking-cost associated to $\tilde{o}(w)$ is $\min_{v \in F_i}(g_i(v,w)) + H_j(w)$ if $C_j(w)$ is not stable and $\min_{v \in F_i}(g_i(v,w)) + C_j(w)$ if $C_j(w)$ is stable;
- the ranking-cost of any (v,w) present in Q_i is $g_i(v,w) + h_i(v) + H_j(w)$;
- the elements of Q_i are ordered by increasing ranking-cost (Figure 3.2).

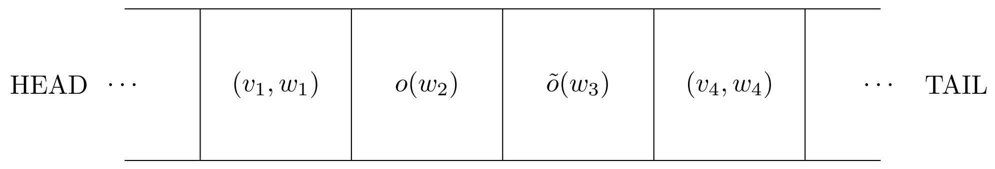

Figure 3.2: Ordering of Q_i, where the ranking costs are such that: $g_i(v_1,w_1) + h_i(w_1) + H_j(w_1) \leq C_i(w_2) + C_j(w_2) \leq \min_{v \in F_i}(g_i(v,w_3)) + H_j(w_3) \leq g_i(v_4,w_4) + h_i(w_4) + H_j(w_4)$

The queue Q_i is provided with two operations: $\mathrm{POP}(Q_i)$ which returns the head element of Q_i (and remove it from Q_i), and $\mathrm{ADD}(Q_i, e)$ which adds the element e in Q_i at the place corresponding to its ranking-cost. Algorithm 6 is the one that φ_i has to execute on automaton $\mathcal{A}_i$. Notice that this algorithm only adds elements of the type (v,w) to Q_i, in fact we assume that elements of the type $o(w)$ and $\tilde{o}(w)$ are in Q_i when necessary (this could be done by other procedures executed by φ_i but here we made the choice to only describe it by properties of Q_i).

Algorithm 6 Algorithm executed by φ_i on $\mathcal{A}_i$

```
$Q_i \leftarrow \emptyset$
$g_i(s_i^0, \epsilon) \leftarrow c_i^i$
ADD($Q_i$, $(s_i^0, \epsilon)$)
repeat
    $e \leftarrow$ POP($Q_i$)
    if $e = o(w)$ then return $w$ endif
    if $e = \tilde{o}(w)$ then
        ADD($Q_i$, $e$)
    else
        /* from now we have $e = (v, w)$ */
        foreach $v' \in S_i$, $t \in T_i$ such that $(v,t) \in f_i$ and $(t,v') \in f_i$ do
            if $\sigma = \lambda_i(t) \in \Lambda_1 \cap \Lambda_2$ then
                if $g_i(v', w\sigma) > g_i(v,w) + c_i(t)$ then
                    $g_i(v', w\sigma) \leftarrow g(v,w) + c_i(t)$
                    ADD($Q_i$, $(v', w\sigma)$)
                endif
            else
                if $g_i(v', w) > g_i(v,w) + c_i(t)$ then
                    $g_i(v', w) \leftarrow g(v,w) + c_i(t)$
                    ADD($Q_i$, $(v', w)$)
                endif
            endif
        done
    endif
endrepeat
```

We now prove that, when there is a solution to the factored optimal planning problem we consider, if φ_i executes Algorithm 6 on $\mathcal{A}_i$, it finds a word w such that there is an optimal global plan u in

$\mathcal{A} = \mathcal{A}_1 \times_a \mathcal{A}_2$ such that $u_{|\Lambda_1 \cap \Lambda_2} = w$. Which is sufficient to prove the algorithm we propose. This is done by proving that, when there is a solution, Algorithm 6 terminates (termination), and that when Algorithm 6 terminates it returns a local plan which is part of an optimal global plan (validity).

Termination: We first notice that there exists a time t after which $o(w)$ will be in Q_i, for some $w \in \Lambda_1 \cap \Lambda_2$. This is just due to the fact that there is a time where $C_i(w)$ and $C_j(w)$ will stabilize. Moreover there exists such a w such that $C_i(w) + C_j(w) \neq +\infty$, because there exists a solution.

Then we assume that $\forall w \in \Lambda_1 \cap \Lambda_2,\ H_j(w) = 0$ and $\forall v \in S_i,\ h_i(v) = 0$ (which is the case where the costs are underestimated the most). In this case, the elements (v, w) of Q_i are ordered by increasing $g_i(v, w)$. There is three possible cases for the head of the queue:

1. an element of type (v, w), this element is replaced by several elements (v', w') such that $g(v', w') > g(v, w)$ (assuming there is no transitions with cost 0 – in fact the proof can be done if we only assume no cycles with cost 0);
2. an element of type $\tilde{o}(w)$, if $C_j(w)$ is not stable it will be stable later (by definition), the case where $C_j(w)$ is stable is impossible: in this case $o(w)$ is in Q_i;
3. an element of type $o(w)$, the algorithm terminates.

From case 1 (and the first part of this demonstration, showing the existence of an element of type $o(w)$ in Q_i after time t) we deduce that there is a time $t' > t$ where an element of type $o(w)$ or $\tilde{o}(w)$ will be the head of Q_i. From case 2 we deduce that, later, an element of type $o(w)$ will be the head of Q_i, and thus, that Algorithm 6 terminates.

The proof of validity is based on a lemma involved in the proof of A^* by Hart et al. [HNR68].

Lemma 14. *In $\mathcal{A}_i$, when Algorithm 6 is applied, at any moment the following holds: for all π, path from s^0 to F such that $\lambda_i(\pi)_{|\Lambda_1 \cap \Lambda_2} = w$ and π optimal according to w, either $o(w)$ is in O_i or there exists (v', w') with v' element of π and $\lambda_i(\pi')_{|\Lambda_1 \cap \Lambda_2} = w'$ (where π' is a prefix of π ending by v') such that: 1) (v', w') is in Q_i and 2) $g_i(v', w') + h_i(v') \leq c_i(\pi)$.*

The proof of lemma 14 is exactly the proof of the corollary of lemma 1 in [HNR68] (considering that we apply A^* on the infinite graph $\mathcal{G} = (V, E)$ where V contains all couples (v, w) such that $v \in S_i$ and $w \in (\Lambda_1 \cap \Lambda_2)^*$, and there is an edge in E from (v, w) to $(v', w\sigma)$ if and only if there is a transition from v to v' labeled by σ in $\mathcal{A}_i$). The first step is to prove that (v', w') is such that $g_i(v', w')$ is the optimal cost to reach v' from s^0 according to w' (which is clear from the manner used to explore $\mathcal{G}$). Then it is sufficient to notice that $h_i(v')$ is a lower bound on the cost of an optimal path from v' to F_i.

Validity: The algorithm terminates when an element of type $o(w)$ is at the head of Q_i. The ranking-cost of $o(w)$ is $C_i(w)+C_j(w)$, which are both stable, which means that it is impossible to find a better cost with w as synchronization word. Suppose that there exists $w' \in \Lambda_1 \cap \Lambda_2$ which is a better synchronization word than w. Given an optimal path π in $\mathcal{A}_i$ such that $\lambda_i(\pi)_{|\Lambda_1 \cap \Lambda_2} = w'$. We know from Lemma 14 that there is (v'', w'') in Q_i such that v'' is an element of π and $\lambda_i(\pi')_{|\Lambda_1 \cap \Lambda_2} = w''$ (where π' is a prefix of π ending by v'') and such that $g_i(v'', w'') + h_i(v'') \leq c_i(\pi)$. Hence $g_i(v'', w'') + h_i(v'') + H_j(w'') \leq c_i(\pi) + H_j(w'')$, and, by the properties required for $H_j(w'')$, we have that $H_j(w'') \leq c_j(\pi'')$ with π'' such that $\lambda_j(\pi'')_{|\Lambda_1 \cap \Lambda_2} = w'$, and thus, $g_i(v'', w'') + h_i(v'') + H_j(w'') \leq c_i(\pi) + c_j(\pi'')$. Moreover $c_i(\pi) + c_j(\pi'')$ is the cost of an optimal global plan with synchronization word w'. However (v'', w'') is after $o(w)$ in Q_i, thus, w' can not be a better synchronization word than w. Finally the result w is an optimal synchronization word. Our algorithm is valid.

We proved termination and validity of Algorithm 6. Hence, our algorithm is correct. We now propose a way to maintain the functions H_j and C_j in order to ensure the properties we require.

Intuitively, for $C_j(w)$ we take an estimate of the cost of an optimal path π from v_j^0 to F_j, such that $\lambda_j(\pi)_{|\Lambda_1 \cap \Lambda_2} = w$. This $C_j(w)$ is computed by φ_j while Algorithm 6 is executed on $\mathcal{A}_j$. Formally it is defined in the following way:

- $C_j(w)$ is said to be optimal when:

  - φ_j has taken (v, w) in Q_j, with $v \in F_j$,
  - and $\nexists (v', w')$ in Q_j such that $g_j(v', w') + h(v') \leq \min\limits_{v \in F_j}(g_j(v, w))$ and w' is a prefix of w,

  in this case we have $C_j(w) = \min\limits_{v \in F_j}(g_j(v, w))$;

- if $C_j(w)$ is not optimal we have:
  - if $\exists v$ such that (v, w) is in Q_i, then $C_j(w) = \min\limits_{(v,w) \text{ in } Q_j \text{ or } v \in F_j}(g_j(v, w) + h_j(v))$,
  - else if $\exists v \in F_j$ such that $g_j(v, w) \neq +\infty$, then $C_j(w) = \min\limits_{v \in F_j}(g_j(v, w))$,
  - else $C_j(w) = +\infty$.

From this C_j one can construct H_j in the following way:

$$H_j(w) = \min_{\tilde{w} \text{ prefix of } w \text{ or } w \text{ prefix of } \tilde{w}} C_j(\tilde{w})$$

By construction of C_j it is clear that this function satisfies the properties required (e.g. that there exists a time where $C_j(w)$ will be optimal for any w). Checking that $H_j(w)$ is correct is a bit harder but can be done using Lemma 14. In fact the intuition behind the construction of C_j and H_j is the following: we take $C_j(w)$ as the best estimation one can obtain of the minimal cost of an accepted path π in $\mathcal{A}_j$ such that $\lambda_j(\pi)_{|\Lambda_i \cap \Lambda_j} = w$. From that we compute $H_j(w)$ as the minimal cost possible for a path $\tilde{\pi}$ in $\mathcal{A}_j$ which could synchronize with a path π in $\mathcal{A}_j$ such that $\lambda_j(\pi)_{|\Lambda_i \cap \Lambda_j} = w$. This means that $\lambda_j(\tilde{\pi})_{|\Lambda_i \cap \Lambda_j} = \tilde{w}$ with $\tilde{w}$ prefix of w – extending $\tilde{\pi}$ could allow it to synchronize with π – or with w prefix of $\tilde{w}$ – extending π could allow it to be synchronized with $\tilde{\pi}$.

In this section we described an algorithm close to A^* algorithm and proved that if φ_i executes this algorithm on $\mathcal{A}_i$ while φ_j executes it on $\mathcal{A}_j$ then the result obtained by φ_i is a word $w \in \Lambda_1 \cap \Lambda_2$ such that there is an optimal plan P in $\mathcal{A} = \mathcal{A}_1 \times_a \mathcal{A}_2$ with $P_{|\Lambda_1 \cap \Lambda_2} = w$. Hence, this algorithm allows us to perform factored optimal planning. In the next section we explain how we could generalize our algorithm to the case where any number of automata are involved (with the assumption that communication graphs are trees).

3.2 General Case

We now consider a compound system $\mathcal{A} = \mathcal{A}_1 \times_a \ldots \times_a \mathcal{A}_n$, constituted of n weighted automata. We assume that the communication graphs of $\mathcal{A}$ are trees and we consider one of them: $\mathcal{G}$. By $\mathcal{N}(i)$ we denote the indices of the neighbors of $\mathcal{A}_i$ in $\mathcal{G}$. As for the previous case we would like to have an agent φ_i for each $\mathcal{A}_i$, which executes Algorithm 6 on $\mathcal{A}_i$. In this section we explain how Q_i could be ordered to ensure the correctness of Algorithm 6. In fact the concepts involved are very close to the ones of previous section, excepted that each agent φ_i has several neighbors in $\mathcal{G}$ and must give to each of these neighbors φ_j some knowledge about $\mathcal{A}_i$ but also knowledge about the $\mathcal{A}_k$ for $k \neq j$ that φ_i knows (as in the MPA).

Exactly as in the previous case, each agent φ_i has its own functions $g_i : S_i \times (\bigcup_{j \in \mathcal{N}(i)} \Lambda_j \cap \Lambda_i)^* \to \mathbb{R}^+ \cup \{+\infty\}$ and $h_i : S_i \to \mathbb{R}^+ \cup \{+\infty\}$. These functions have exactly the same meaning as in the previous case.

Each agent φ_i can now access to a collection of functions $H^i_j : (\Lambda_j \cap \Lambda_i)^* \to \mathbb{R}^+ \cup \{+\infty\}$ (one for each of its neighbors φ_j in $\mathcal{G}$), a collection of functions $C^i_j : (\Lambda_j \cap \Lambda_i)^* \to \mathbb{R}^+ \cup \{+\infty\}$ (also one for each neighbor of φ_i in $\mathcal{G}$) and a function $C_i : (\bigcup_{j \in \mathcal{N}(i)} \Lambda_j \cap \Lambda_i)^* \to \mathbb{R}^+ \cup \{+\infty\}$. These functions have the following properties:

- $H^i_j(w)$ is a lower bound on the cost of accepted paths π_j in $\mathcal{A}_j \times_a \mathcal{A}'$ (where $\mathcal{A}'$ is the part of the communication graph $\mathcal{G}$ which is separated from A_i by $\mathcal{A}_j$, see Figure 3.3) which are such that $\lambda'(\pi_j)_{|\Lambda_j \cap \Lambda_i} = w'$ (where λ' associates a symbol to each transition of $\mathcal{A}_j \times_a \mathcal{A}'$) with w a prefix of w';

- $C_j^i(w)$ is such that there exists a time t after which the value $C_j(w)$ is stable and is the optimal cost of an accepted path π_j in $\mathcal{A}_j \times_a \mathcal{A}'$ such that $\lambda'(\pi_j)_{|\Lambda_j \cap \Lambda_i} = w$. Moreover we assume that φ_i knows when $C_j^i(w)$ is stable;
- $C_i(w)$ is such that there exists a time t after which the value $C_i(w)$ is stable and is the optimal cost of an accepted path π_i in $\mathcal{A}_i$ such that $\lambda_i(\pi_i)_{|\bigcup_{j\in\mathcal{N}(i)} \Lambda_j \cap \Lambda_i} = w$. We assume that φ_i knows when $C_i(w)$ is stable.

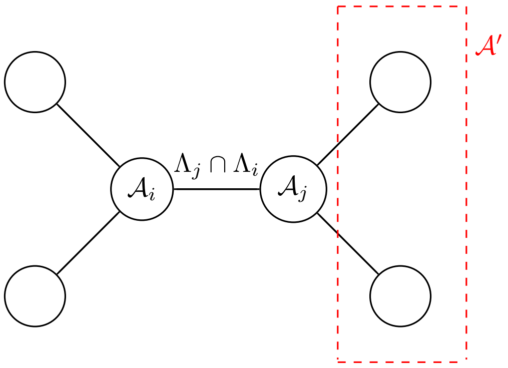

Figure 3.3: A communication graph, the part $\mathcal{A}'$ of the graph, separated from $\mathcal{A}_i$ by A_j, is squared

As in previous case, each agent φ_i have access to a queue Q_i which contains couples (v, w) where $v \in S_i$ and $w \in (\bigcup_{j\in\mathcal{N}(i)} \Lambda_j \cap \Lambda_i)^*$ and two sorts of special elements: $o(w)$ with $w \in (\bigcup_{j\in\mathcal{N}(i)} \Lambda_j \cap \Lambda_i)^*$ and $\tilde{o}(w)$ with $w \in (\bigcup_{j\in\mathcal{N}(i)} \Lambda_j \cap \Lambda_i)^*$. This queue has the following properties:

- $o(w)$ is in Q_i if and only if $C_i(w)$ is stable and $\forall j \in \mathcal{N}(i), C_j^i(w_{|\Lambda_j\cap\Lambda_i})$ is stable, the ranking-cost associated to $o(w)$ is $C_i(w) + \sum_{j\in\mathcal{N}(i)} C_j(w)$;
- $\tilde{o}(w)$ is in Q_i if and only if an accepted word π_i such that $\lambda_i(\pi_i)_{|\bigcup_{j\in\mathcal{N}(i)} \Lambda_j\cap\Lambda_i}$ has been found in $\mathcal{A}_i$ (which means that $g_i(v, w) \neq +\infty$ for some $v \in F_i$) and $o(w)$ is not in Q_i, the ranking-cost associated to $\tilde{o}(w)$ is $\min_{v\in F_i}(g_i(v, w)) + \sum_{j\in\mathcal{N}(i)} \mathcal{J}(w_{|\Lambda_j\cap\Lambda_i})$, where $\mathcal{J}(w_{|\Lambda_j\cap\Lambda_i})$ is $C_j^i(w_{|\Lambda_j\cap\Lambda_i})$ if it is stable and $H_j^i(w_{|\Lambda_j\cap\Lambda_i})$ else;
- the ranking-cost of any (v, w) present in Q_i is $g_i(v, w) + h_i(v) + \sum_{j\in\mathcal{N}(i)} \mathcal{J}(w_{|\Lambda_j\cap\Lambda_i})$, where $\mathcal{J}$ is defined as above;
- the elements of Q_i are ordered by increasing ranking-cost.

One can prove that, with this definition of Q_i, if algorithm 6 is executed by all agents φ_i on their own automata $\mathcal{A}_i$, it will converge (assuming that there is a path from initial to final states in $\mathcal{A} = \mathcal{A}_1 \times_a \dots \times_a \mathcal{A}_n$) and return a word $w \in (\bigcup_{j\in\mathcal{N}(i)} \Lambda_j \cap \Lambda_i)$ such that 1) there is a plan P_i in $\mathcal{A}_i$ which verifies $(P_i)_{|\bigcup_{j\in\mathcal{N}(i)} \Lambda_j\cap\Lambda_i} = w$ and 2) this plan P_i is part of an optimal global plan in $\mathcal{A}$. The proof of this statement is quite similar than in the simple case presented in previous section, hence we do not give it here.

In fact, as in the previous case, it is possible to construct, for φ_i, the functions H_j^i, C_j^i, and C_i. One can notice that this is where the necessity of living on a tree appears. Indeed, nothing in Algorithm 6 enforce this restriction. But, when constructing the functions H_j^i, C_j^i, and C_i it becomes necessary to propagates some information (in particular on optimality of C_j^i), and thus, if communication graphs contain cycles some troubles appear. The functions H_j^i, C_j^i, and C_i can be constructed as follows:

- C_i is constructed exactly as in the previous case;
- C_j^i is constructed by φ_j from C_j and all the C_k^j for $k \in \mathcal{N}(j) \setminus \{i\}$ and, for $w \in \Lambda_j \cap \Lambda_i$, has the value $C_j^i(w) = \min\limits_{w'_{|\Lambda_j\cap\Lambda_i}=w} C_j(w') + \sum_{k\in\mathcal{N}(j)\setminus\{i\}} C_k^j(w'_{|\Lambda_k\cap\Lambda_j})$;

- H^i_j is constructed by φ_j fromC^i_j and, for $w \in \Lambda_j \cap \Lambda_i$, has the value $H^i_j(w) = \min\limits_{\tilde{w} \text{ prefix of } w \text{ or } w \text{ prefix of } \tilde{w}} C^i_j(\tilde{w})$.

To decide if $C^i_j(w)$ is optimal is probably the hardest part of our problem. Indeed, to compute $C^i_j(w)$ we make a reverse projection of w into w'. Generally this generates a huge number of w', and even, potentially, an infinite number of w' (notice that this does not create problems to compute $C^i_j(w)$ as, at a given moment, only a finite number of these w' can be such that $C_j(w') \neq +\infty$ and, thus, all the others are just avoided). To ensure that $C^i_j(w)$ is optimal we need to check that, for $w'' = \operatorname*{argmin}\limits_{w'_{|\Lambda_j \cap \Lambda_i} = w} C_j(w') + \sum_{k \in \mathcal{N}(j)\setminus\{i\}} C^j_k(w'_{|\Lambda_k \cap \Lambda_j})$ we have:

1. $C_j(w'')$ is optimal (which can be done exactly as in the previous simple case);
2. $C^j_k(w''_{|\Lambda_k \cap \Lambda_j})$ is optimal for all $k \in \mathcal{N}(j) \setminus \{i\}$ (which is provided by φ_k), this clearly implies that problems live on trees, as shown in Figure 3.4;
3. and there is no w''' such that $C_j(w''') + \sum_{k \in \mathcal{N}(j)\setminus\{i\}} C^j_k(w'''_{|\Lambda_k \cap \Lambda_j})$ could become smaller than $C_j(w'') + \sum_{k \in \mathcal{N}(j)\setminus\{i\}} C^j_k(w''_{|\Lambda_k \cap \Lambda_j})$ latter.

The third point is the difficult one. For the moment we do not have a clever way to ensure the optimality of $C^i_j(w)$. This problem is probably close to the one of finding a bound to stop the partial determinisation suggested in the previous chapter.

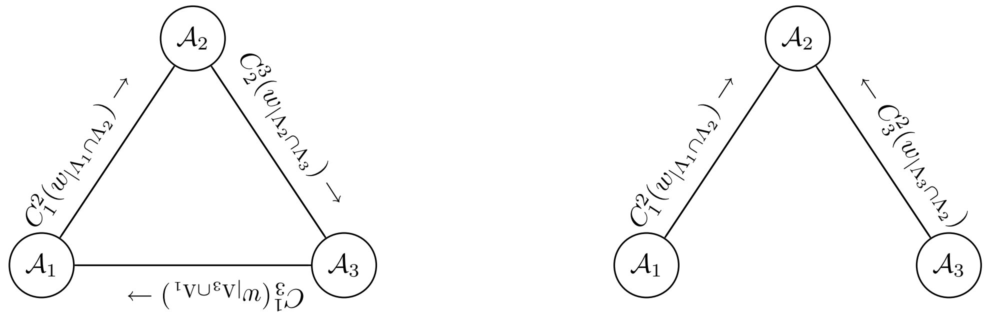
$\mathcal{A}_2$
$\mathcal{A}_1$
$\mathcal{A}_3$
$C^2_1(w_{|\Lambda_1 \cap \Lambda_2}) \rightarrow$
$C^3_2(w_{|\Lambda_2 \cap \Lambda_3}) \rightarrow$
$C^1_3(w_{|\Lambda_3 \cap \Lambda_1}) \rightarrow$
$\mathcal{A}_2$
$\mathcal{A}_1$
$\mathcal{A}_3$
$C^2_1(w_{|\Lambda_1 \cap \Lambda_2}) \rightarrow$
$\leftarrow C^2_3(w_{|\Lambda_3 \cap \Lambda_2})$

Figure 3.4: Left: communication graph contains a cycle, to decide optimality of $C^2_1(w_{|\Lambda_1 \cap \Lambda_2})$ agent φ_1 needs that φ_3 decides optimality of $C^1_3(w_{|\Lambda_3 \cap \Lambda_1})$, which implies that φ_2 has decided optimality of $C^3_2(w_{|\Lambda_2 \cap \Lambda_3})$ and thus that φ_1 has decided optimality of $C^2_1(w_{|\Lambda_1 \cap \Lambda_2})\ldots$ who decides first? Right: communication graph is a tree, φ_1 and φ_3 can decide optimality of $C^2_3(w_{|\Lambda_3 \cap \Lambda_2})$ and $C^2_1(w_{|\Lambda_1 \cap \Lambda_2})$ alone.

In this chapter we introduced a new algorithm to solve factored optimal planning problems. This algorithm is based on the well know A^* algorithm [HNR68].

We first proved that, in a simple case involving only two automata, our algorithm is correct: it solves the problem. However, in the general case, involving any number of automata, even if we proved that the algorithm is correct we are not, for the moment, able to ensure all the properties needed by the objects manipulated.

Conclusion

In this report we proposed what are, to our knowledge, the two first approaches to factored planning which ensure to find optimal plans.

Our first approach is based on a message passing algorithm and weighted automata calculus. It consists in manipulating all the local plans of each component (thanks to weighted automata) and to combine these plans in order to compute all the valid optimal global plans (with message passing algorithm). The only difficulty is in determinisation of weighted automata (which is not always feasible). Two solutions are possible: avoid determinisation (which increases the difficulty of necessary operations such as synchronous product) or perform partial determinisation (which seems to be the better solution).

Our second approach is an adaptation of the classical A^* algorithm to a distributed context. The principle is to execute an A^* like algorithm on each component, taking into account the information which comes from other components along the execution of the algorithm. This approach has been proved correct in the simple case involving only two components. In general case there is still some work to do on technical aspects, but the principle of the algorithm has also been proved correct.

Further Work

We are currently working on the technical stuff needed for our distributed A^* algorithm and on the concept of partial determinisation which seem to be closely related. Indeed, if we find an efficient way to compute a bound W such that optimal words can not have a cost greater than W, then it is possible to stop the determinisation of an automaton $\mathcal{A}$ when any word of cost smaller than W is in the determinised automaton $\mathcal{A}'$ – as described in Section 2.2.3 – but it is also possible to restrict the number of synchronization words to check in our distributed A^* algorithm to something finite. Conversely, if we are able to ensure that a word w is such that $C^i_j(w_{|\Lambda_i \cap \Lambda_j})$ is minimal then we can probably make an adaptation of it to stop the determinisation procedure in way such that the determinised automaton obtained is sufficient to perform our computations.

We also would like to improve our approach by weighted automata calculus in several ways. A thing could be to avoid the restriction on communication graphs (that currently have to live on trees) by using *turbo-algorithms* [Fab03]. These algorithms, inspired from turbo-codes, can deal with cyclic communication graphs and yield approximations of the solutions. When the communication graph is sufficiently sparse these approximate solutions can be close to optimal.

Another point would be to include in our framework the notion of *read arcs*. This means that actions which only read some variables (e.g. have given values of these variables as precondition and does not modify these variables) should not be in conflict. This idea implies existence of asymmetric conflict, when one action reads variables and another action modifies them.

We also would like to have more *partial ordering* in our plans. Indeed, currently our global plans are partially ordered (private actions from local plans can be ordered in several ways) but this is not the case of our local plans. Hence, we would like to find local plans as partial order. This could be done using (weighted) petri-nets for components instead of (weighted) automata. However, if product (or composition) of petri-nets can be defined easily, this is not the case for projection.

Finally it could be a good thing to look into *game theory*. The idea is that each component of a planning problem could be owned by an agent. In this case each agent would like to minimize its personal cost (e.g. find a local plan as close to optimal as possible) but has to deal with the other agents, and, thus, make some concession. In this case, we would like to know how the message passing algorithm should be adapted in order to ensure each agent to obtain the best cost it can expect.

Acknowledgments

I want to thank Eric Fabre for having given me the opportunity to work on this wonderful topic and for his support and disponibility all along this internship. I also thank all the members from DistribCom team. Finally I would like to thank Sylvie Thiébaux for having allowed us to have some experimental confirmation of the usefulness of the message passing algorithm for planning.

Bibliography

[BD06] Ronen I. Brafman and Carmel Domshlak. Factored planning: How, when, and when not. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-2006)*, pages 809–814, 2006.

[BD08] Ronen I. Brafman and Carmel Domshlak. From one to many: Planning for loosely coupled multi-agent systems. In *ICAPS*, pages 28–35, 2008.

[BHHT08] Blai Bonet, Patrik Haslum, Sarah Hickmott, and Sylvie Thiébaux. Directed unfolding of petri nets. *Transactions on Petri Nets and Other Models of Concurrency I*, pages 172–198, 2008.

[BY94] Fahiem Bacchus and Qiang Yang. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, 71:43–100, 1994.

[CL99] Christos G. Cassandras and Stphane Lafortune. *Introduction to Discrete Event Systems.* Kluwer Academic Publishers, sep 1999.

[Dec03] Rina Dechter. *Constraint Processing.* Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, 2003.

[ERV96] Javier Esparza, Stefan Romer, and Walter Vogler. An improvement of mcmillan's unfolding algorithm. In *Formal Methods in System Design*, pages 87–106, 1996.

[Fab03] Eric Fabre. Convergence of the turbo algorithm for systems defined by local constraints. Research report PI 4860, INRIA, May 2003.

[Fab07] Eric Fabre. *Bayesian Networks of Dynamic Systems.* Habilitation à diriger des recherches, Université de Rennes1, 2007.

[GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory & Practice.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[HHH08] Patrik Haslum, Malte Helmert, and Jörg Hoffmann. Explicit-state abstraction: A new method for generating heuristic functions. *Proc. AAAI Conference on Artificial Intelligence*, 2008.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

[HNR72] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Bull.*, (37):28–29, 1972.

[HRTW07] Sarah Hickmott, Jussi Rintanen, Sylvie Thiébaux, and Lang White. Planning via petri net unfolding. *Proc. International Joint Conference on Artificial Intelligence (IJCAI-07)*, 2007.

[KLMP04] Ines Klimann, Sylvain Lombardy, Jean Mairesse, and Christophe Prieur. Deciding unambiguity and sequentiality from a finitely ambiguous max-plus automaton. *Theor. Comput. Sci.*, 327(3):349–373, 2004.

[KM05] Daniel Kirsten and Ina Mäurer. On the determinization of weighted automata. *Journal of Automata, Languages and Combinatorics*, 10(2/3):287–312, 2005.

[Kno94] Craig Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68:243–302, 1994.

[LM06] Sylvain Lombardy and Jean Mairesse. Series which are both max-plus and min-plus rational are unambiguous. *RAIRO-THEORETICAL INFORMATICS AND APPLICATIONS*, 40:1, 2006.

[McM93] Kenneth L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *CAV '92: Proceedings of the Fourth International Workshop on Computer Aided Verification*, pages 164–177, London, UK, 1993. Springer-Verlag.

[Moh94] Mehryar Mohri. Minimization of sequential transducers. In *Lecture Notes in Computer Science*, pages 151–163. Springer-Verlag, 1994.

[Moh97] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23:269–311, 1997.

[Moh09] Mehryar Mohri. *Weighted automata algorithms.* Springer, 2009.

[MPR] Mehryar Mohri, Fernando C. N. Pereira, and Michael D. Riley. AT&T FSM library^{TM}, finite-state machine library. `http://www.research.att.com/~fsmtools/fsm/`.

[Sak03] Jacques Sakarovitch. *Éléments de theorie des automates.* Éditions Vuibert, 2003.

[SW05] Rong Su and W. Murray Wonham. Global and local consistencies in distributed fault diagnosis for discrete-event systems. *Automatic Control, IEEE Transactions on*, 50(12):1923–1935, Dec. 2005.

Appendix A

Proofs

A.1 Languages

Proof of Lemma 1. Let $w \in \Sigma^*$ be a word. Let Σ_1 and Σ_2 be two alphabets. We prove that $P_{\Sigma_1} \circ P_{\Sigma_2}(w) = P_{\Sigma_1 \cap \Sigma_2}(w)$.

Three cases are possible: $w = \epsilon$, $w = \sigma \in \Sigma$, or $w = w'\sigma$ where $w' \in \Sigma^+$ and $\sigma \in \Sigma$.

If $w = \epsilon$, then:

$$\begin{aligned}
P_{\Sigma_1} \circ P_{\Sigma_2}(w) &= P_{\Sigma_1} \circ P_{\Sigma_2}(\epsilon) \\
&= \epsilon \\
&= P_{\Sigma_1 \cap \Sigma_2}(\epsilon) \\
&= P_{\Sigma_1 \cap \Sigma_2}(w).
\end{aligned}$$

If $w = \sigma$, then:

$$\begin{aligned}
P_{\Sigma_1} \circ P_{\Sigma_2}(w) &= P_{\Sigma_1} \circ P_{\Sigma_2}(\sigma) \\
&= \begin{cases} P_{\Sigma_1}(\sigma) & \text{if } \sigma \in \Sigma_2 \\ P_{\Sigma_1}(\epsilon) & \text{if} \sigma \notin \Sigma_2 \end{cases} \\
&= \begin{cases} \sigma & \text{if } \sigma \in \Sigma_2 \text{ and } \sigma \in \Sigma_1 \\ \epsilon & \text{if } \sigma \in \Sigma_2 \text{ and } \sigma \notin \Sigma_1 \\ \epsilon & \text{if } \sigma \notin \Sigma_2 \end{cases} \\
&= \begin{cases} \sigma & \text{if } \sigma \in \Sigma_1 \cap \Sigma_2 \\ \epsilon & \text{if } \sigma \notin \Sigma_1 \cap \Sigma_2 \end{cases} \\
&= P_{\Sigma_1 \cap \Sigma_2}(\sigma) \\
&= P_{\Sigma_1 \cap \Sigma_2}(w).
\end{aligned}$$

If $w = w'\sigma$ then:

$$\begin{aligned}
P_{\Sigma_1} \circ P_{\Sigma_2}(w) &= P_{\Sigma_1} \circ P_{\Sigma_2}(w'\sigma) \\
&= P_{\Sigma_1}(P_{\Sigma_2}(w')P_{\Sigma_2}(\sigma)) \\
&= P_{\Sigma_1} \circ P_{\Sigma_2}(w')P_{\Sigma_1} \circ P_{\Sigma_2}(\sigma) \\
&= P_{\Sigma_1} \circ P_{\Sigma_2}(w')P_{\Sigma_1 \cap \Sigma_2}(\sigma).
\end{aligned}$$

By induction over the length of w we conclude that, in this case, $P_{\Sigma_1} \circ P_{\Sigma_2}(w) = P_{\Sigma_1 \cap \Sigma_2}(w)$.

We proved that in all possible cases the lemma is verified. □

Proof of Lemma 2. Let $\mathcal{L}$ be a language over an alphabet Σ. Let w be a word from $\mathcal{L}$. We show that $P_\Sigma(w) = w$, and thus $P_\Sigma(w) = w$, which prove the lemma.

Three cases are possible: $w = \epsilon$, $w = \sigma \in \Sigma$, or $w = w'\sigma$ where $w' \in \Sigma^+$ and $\sigma \in \Sigma$.

If $w = \epsilon$, then:

$$\begin{aligned} P_\Sigma(w) &= P_\Sigma(\epsilon) \\ &= \epsilon \\ &= w. \end{aligned}$$

If $w = \sigma$, with $\sigma \in \Sigma$, then:

$$\begin{aligned} P_\Sigma(w) &= P_\Sigma(\sigma) \\ &= \begin{cases} \sigma & \text{if } \sigma \in \Sigma \\ \epsilon & \text{if} \sigma \notin \Sigma \end{cases} \\ &= \sigma \\ &= w. \end{aligned}$$

If $w = w'\sigma$, with $\sigma \in \Sigma$ then:

$$\begin{aligned} P_\Sigma(w) &= P_\Sigma(w'\sigma) \\ &= P_\Sigma(w')P_\Sigma(\sigma) \\ &= P_\Sigma(w')\sigma. \end{aligned}$$

By induction over the length of w we conclude that, in this case, $P_\Sigma(w) = w$.

We proved that $P_\Sigma(w) = w$, thus $P_\Sigma(\mathcal{L}) = \mathcal{L}$. This ends the proof of the lemma.

□

Proof of Lemma 3. Let $\mathcal{L}_1$ be a language over Σ_1. Let $\mathcal{L}_2$ be a language over Σ_2. Let Σ be an alphabet such that $\Sigma \supseteq \Sigma_1 \cap \Sigma_2$.

One can notice that:

$$\begin{aligned} P_\Sigma(\mathcal{L}_1 \| \mathcal{L}_2) &= P_\Sigma(P_{\Sigma_1}^{-1}(\mathcal{L}_1) \cap P_{\Sigma_2}^{-1}(\mathcal{L}_2)) \\ &= P_\Sigma(\{w \in (\Sigma_1 \cup \Sigma_2)^* \mid P_{\Sigma_1}(w) \in \mathcal{L}_1 \text{ and } P_{\Sigma_2}(w) \in \mathcal{L}_2\}) \\ &= \{w \mid \exists u \in (\Sigma_1 \cup \Sigma_2)^*, w = P_\Sigma(u) \text{ and } P_{\Sigma_1}(u) \in \mathcal{L}_1 \text{ and } P_{\Sigma_2}(u) \in \mathcal{L}_2\}. \end{aligned}$$

One can also notice that:

$$\begin{aligned} P_\Sigma(\mathcal{L}_1) \| P_\Sigma(\mathcal{L}_2) &= P_{\Sigma\cap\Sigma_1}^{-1}(P_\Sigma(\mathcal{L}_1)) \cap P_{\Sigma\cap\Sigma_2}^{-1}(P_\Sigma(\mathcal{L}_2)) \\ &= \{w \in (\Sigma \cap \Sigma_1 \cup \Sigma \cap \Sigma_2)^* \mid P_{\Sigma\cap\Sigma_1}(w) \in P_\Sigma(\mathcal{L}_1) \text{ and } P_{\Sigma\cap\Sigma_2}(w) \in P_\Sigma(\mathcal{L}_2)\}. \end{aligned}$$

We first prove that: $P_\Sigma(\mathcal{L}_1 \| \mathcal{L}_2) \subseteq P_\Sigma(\mathcal{L}_1) \| P_\Sigma(\mathcal{L}_2)$. Let w be an element of $P_\Sigma(\mathcal{L}_1 \| \mathcal{L}_2)$. We have: $\exists u \in (\Sigma_1 \cup \Sigma_2)^*$ such that $w = P_\Sigma(u)$, $P_{\Sigma_1}(u) \in \mathcal{L}_1$, and $P_{\Sigma_2}(u) \in \mathcal{L}_2$.

We know that $w \in (\Sigma_1 \cup \Sigma_2)^*$ and $w \in \Sigma^*$, thus, $w \in (\Sigma \cap (\Sigma_1 \cup \Sigma_2))^* = (\Sigma \cap \Sigma_1 \cup \Sigma \cap \Sigma_2)^*$. Moreover, for $i \in \{1, 2\}$, we have:

$$\begin{aligned} P_{\Sigma\cap\Sigma_i}(w) &= P_{\Sigma_i\cap\Sigma}(P_\Sigma(u)) \\ &= P_{\Sigma_i\cap\Sigma}(u) \\ &= P_{\Sigma\cap\Sigma_i}(u) \\ &= P_\Sigma(P_{\Sigma_i}(u)). \end{aligned}$$

Now, $P_{\Sigma_i}(u) \in \mathcal{L}_i$, and thus $P_\Sigma(P_{\Sigma_i}(u)) \in P_\Sigma(\mathcal{L}_i)$, hence, $P_{\Sigma\cap\Sigma_i}(w) \in P_\Sigma(\mathcal{L}_i)$. This proves that $P_\Sigma(\mathcal{L}_1\|\mathcal{L}_2) \subseteq P_\Sigma(\mathcal{L}_1)\|P_\Sigma(\mathcal{L}_2)$.

We now prove that: $P_\Sigma(\mathcal{L}_1\|\mathcal{L}_2) \supseteq P_\Sigma(\mathcal{L}_1)\|P_\Sigma(\mathcal{L}_2)$. Let w be an element of $P_\Sigma(\mathcal{L}_1)\|P_\Sigma(\mathcal{L}_2)$. We know that $w \in (\Sigma \cap (\Sigma_1 \cup \Sigma_2))^*$.

As $w' = P_{\Sigma\cap\Sigma_1}(w) \in P_\Sigma(\mathcal{L}_1)$, there exists $u_1 \in \mathcal{L}_1$ such that $w' = P_\Sigma(u_1)$. As $w'' = P_{\Sigma\cap\Sigma_2}(w) \in P_\Sigma(\mathcal{L}_2)$, there exists $u_2 \in \mathcal{L}_2$ such that $w'' = P_\Sigma(u_2)$. Moreover $P_\Sigma(w) = w$, thus, $w' = P_{\Sigma_1}(w)$ and $w'' = P_{\Sigma_2}(w)$. Hence, all shared symbols are in the same order in u_1 and u_2. Thus one can easily construct u from u_1 and u_2 such that $P_\Sigma(u) = w$, $P_{\Sigma_1}(u) = u_1$, and $P_{\Sigma_2}(u) = u_2$. Figure A.1 shows the relation between u, w, u_1, and u_2; and an example of construction of $u = a_1a_2\dots a_n$ (obtained by following the line on the figure) from w,u_1, and u_2, which have the properties described above.

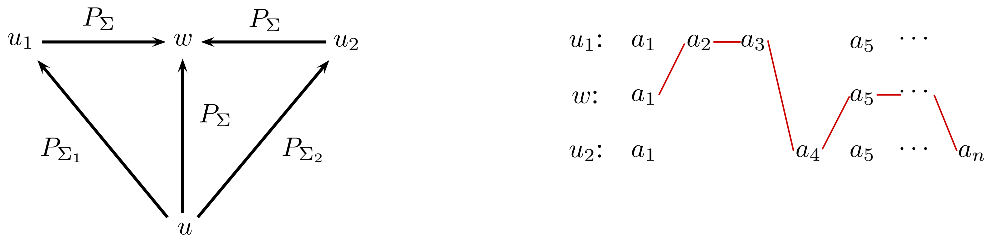
u_1
P_Σ
w
P_Σ
u_2
P_Σ
P_{Σ_1}
P_{Σ_2}
u
u_1: a_1 a_2 a_3 a_5 $\cdots$
w: a_1 a_5 $\cdots$
u_2: a_1 a_4 a_5 $\cdots$ a_n

Figure A.1: Relation between w, u, u_1, and u_2 (left); Construction of u (right).

This is equivalent to $P_\Sigma(u) = w$, $P_{\Sigma_1}(u) \in \mathcal{L}_1$, and $P_{\Sigma_2}(u) \in \mathcal{L}_2$. This proves that $P_\Sigma(\mathcal{L}_1\|\mathcal{L}_2) \supseteq P_\Sigma(\mathcal{L}_1)\|P_\Sigma(\mathcal{L}_2)$ and ends the proof of the lemma.

□

Proof of Lemma 4. We show that $\mathbb{I} = \{\epsilon\}$, language over the alphabet $\Sigma_\mathbb{I} = \emptyset$, is acceptable. Let $\mathcal{L}$ be a language over the alphabet Σ.

$$\begin{aligned}
\mathcal{L}\|\mathbb{I} &= P_\Sigma^{-1}(\mathcal{L}) \cap P_{\Sigma_\mathbb{I}}^{-1}(\mathbb{I}) \\
&= P_\Sigma^{-1}(\mathcal{L}) \cap \{s \in \Sigma^* \mid P_\emptyset(s) = \epsilon\} \\
&= P_\Sigma^{-1}(\mathcal{L}) \cap \Sigma^* \\
&= \{s \in \Sigma^* \mid P_\Sigma(s) \in \mathcal{L}\} \\
&= \mathcal{L}
\end{aligned}$$

This proves the lemma. □

A.2 Automata

Proof of Lemma 6. Let $\mathcal{A} = (S, T, f, s^0, \lambda, \Lambda, F)$ be an automaton. We first prove that:

$$\mathcal{L}(\Pi_{\Lambda_1} \circ \Pi_{\Lambda_2}(\mathcal{A})) = \mathcal{L}(\Pi_{\Lambda_1\cap\Lambda_2}(\mathcal{A})). \tag{A.1}$$

We know that:

$$\mathcal{L}(\Pi_{\Lambda_1} \circ \Pi_{\Lambda_2}(\mathcal{A})) = P_{\Lambda_1} \circ P_{\Lambda_2}(\mathcal{L}(\mathcal{A})),$$

and that:

$$_{L}(\Pi_{\Lambda_1\cap\Lambda_2}(\mathcal{A})) = P_{\Lambda_1\cap\Lambda_2}(_{L}(\mathcal{A})).$$

Moreover, for any language $\mathcal{L}$, and alphabets Σ_1 and Σ_2, we have:

$$P_{\Sigma_1} \circ P_{\Sigma_2}(\mathcal{L}) = P_{\Sigma_1\cap\Sigma_2}(\mathcal{L}).$$

Thus Formula A.1 is proved.

By Lemma 5, Formula A.1 is sufficient to prove the lemma. □

Proof of Lemma 7. Let $\mathcal{A} = (S, T, f, s^0, \lambda, \Lambda, F)$ be an automaton. Let $\Lambda' = \Lambda$. We first prove that:

$$\mathcal{L}(\Pi_\Lambda(\mathcal{A})) = \mathcal{L}(\mathcal{A}). \tag{A.2}$$

We know that:

$$\mathcal{L}(\Pi_\Lambda(\mathcal{A})) = P_\Lambda(_L(\mathcal{A})),$$

and that:

$$P_\Lambda(\mathcal{L}(\mathcal{A})) = \mathcal{L}(\mathcal{A}).$$

Thus Formula A.2 is proved.

By Lemma 5, Formula A.2 is sufficient to prove the lemma. □

Proof of Lemma 8. We first prove that:

$$\mathcal{L}(\Pi_{\Lambda_3}(\mathcal{A}_1 \wedge \mathcal{A}_2)) = \mathcal{L}(\Pi_{\Lambda_3}(\mathcal{A}_1) \wedge \Pi_{\Lambda_3}(\mathcal{A}_2)). \tag{A.3}$$

We know that:

$$\begin{aligned}\mathcal{L}(\Pi_{\Lambda_3}(\mathcal{A}_1 \wedge \mathcal{A}_2)) &= P_{\Lambda_3}(\mathcal{L}(\mathcal{A}_1 \wedge \mathcal{A}_2)) \\ &= P_{\Lambda_3}(\mathcal{L}(\mathcal{A}_1) \|_\ell \mathcal{L}(\mathcal{A}_2)),\end{aligned}$$

and that:

$$\begin{aligned}\mathcal{L}(\Pi_{\Lambda_3}(\mathcal{A}_1) \wedge \Pi_{\Lambda_3}(\mathcal{A}_2)) &= \mathcal{L}(\Pi_{\Lambda_3}(\mathcal{A}_1)) \|_\ell \mathcal{L}(\Pi_{\Lambda_3}(\mathcal{A}_2)) \\ &= P_{\Lambda_3}(\mathcal{L}(\mathcal{A}_1)) \|_\ell P_{\Lambda_3}(\mathcal{A}_2).\end{aligned}$$

Moreover, for any languages $\mathcal{L}_1$ over Σ_1 and $\mathcal{L}_2$ over Σ_2, and any alphabet $\Sigma_3 \supseteq \Sigma_1 \cap \Sigma_2$ we have:

$$P_{\Sigma_3}(\mathcal{L}_1 \|_\ell \mathcal{L}_2) = P_{\Sigma_3}(\mathcal{L}_1) \|_\ell P_{\Sigma_3}(\mathcal{L}_2).$$

Thus Formula A.3 is proved.

By Lemma 5, Formula A.3 is sufficient to prove the lemma. □

Proof of Lemma 9. Take $(\{s_0\}, \emptyset, \rightarrow, s_0, \lambda, \emptyset, \{s_0\})$ as $\mathbb{I}$. We first prove that:

$$\mathcal{L}(\mathcal{A} \wedge \mathbb{I}) = \mathcal{L}(\mathcal{A}). \tag{A.4}$$

We know that:

$$\mathcal{L}(\mathcal{A} \wedge \mathbb{I}) = \mathcal{L}(\mathcal{A}) \|_\ell \mathcal{L}(\mathbb{I}).$$

Moreover, it is easy to see that:

$$\mathcal{L}(\mathbb{I}) = \{\epsilon\}$$

Thus, we have:

$$\mathcal{L}(\mathcal{A}) \|_\ell \mathcal{L}(\mathbb{I}) = \mathcal{L}(\mathcal{A}),$$

and Formula A.4 is proved.

By Lemma 5, Formula A.4 is sufficient to prove the lemma. □

A.3 Weighted Languages

Proof of Lemma 12. From Lemma 3 we know that $\forall (u, w) \in P_\Sigma(\mathcal{L}_1 \times_\ell \mathcal{L}_2)$ there is w' such that $(u, w') \in P_\Sigma(\mathcal{L}_1) \times_\ell P_\Sigma(\mathcal{L}_2)$, and $\forall (u, w') \in P_\Sigma(\mathcal{L}_1) \times_\ell P_\Sigma(\mathcal{L}_2)$ there is w such that $(u, w) \in P_\Sigma(\mathcal{L}_1 \times_\ell \mathcal{L}_2)$. We just have to prove that $w = w'$.

Let (u, w') be an element of $P_\Sigma(\mathcal{L}_1) \times_\ell P_\Sigma(\mathcal{L}_2)$. Let (u, w) be the corresponding element in $P_\Sigma(\mathcal{L}_1 \times_\ell \mathcal{L}_2)$.

We know that $w \leq w'$, indeed $w' = w_1 + w_2$ is obtained from $(u_1, w_1) \in \mathcal{L}_1$ and $(u_2, w_2) \in \mathcal{L}_2$ such that $u_{1|\Sigma}$ can synchronize with $u_{2|\Sigma}$. Because $\Sigma \supseteq \Sigma_1 \cap \Sigma_2$ we have that u_1 and u_2 are also synchronizable (all the synchronization symbols are ever in $u_{1|\Sigma}$ and $u_{2|\Sigma}$), the corresponding element in $\mathcal{L}_1 \times_\ell \mathcal{L}_2$ is $(u', w_1 + w_2) = (u', w')$. It is clear that $u'_{|\Sigma} = u$. Hence, by definition of projection, we have that $w \leq w'$.

We also know that $w' \leq w$. Indeed, $w = w_1 + w_2$ is obtained from $u_1, w_1 \in \mathcal{L}_1$ and $(u_2, w_2) \in \mathcal{L}_2$ such that u_1 and u_2 are synchronizable. Hence, because $\Sigma \supseteq \Sigma_1 \cap \Sigma_2$, we have that $u_{1|\Sigma}$ and $u_{2|\Sigma}$ are also synchronizable. The word u' obtained from synchronization of $u_{1|\Sigma}$ and $u_{2|\Sigma}$ is such that (u', w'') is in $P_\Sigma(\mathcal{L}_1) \times_\ell P_\Sigma(\mathcal{L}_2)$, and, by definition of projection, we have that $w'' \leq w_1 + w_2 = w$. Moreover, one can notice that $u' = u$. Thus, (u, w'') is in $P_\Sigma(\mathcal{L}_1) \times_\ell P_\Sigma(\mathcal{L}_2)$. As we know that (u, w') is in $P_\Sigma(\mathcal{L}_1) \times_\ell P_\Sigma(\mathcal{L}_2)$ we have $w' = w''$, and thus, $w' \leq w$.

We proved that $w \leq w'$ and $w' \leq w$. Thus, $w = w'$. This concludes the proof of the lemma.

$\square$

Appendix B

Weighted Automata algorithms

This part of our work is based on Mohri's work about speech processing [Moh09].

For a WA $\mathcal{A} = (S, T, f, s^0, \lambda, \Lambda, F, c, c^i, c^F)$ we consider that c^i and c^F are functions defined on S: $c^i(s^0)$ is the initial cost and $c^i(s) = 0$ for any $s \neq s^0$; and $c^F(s)$ is the final cost of s if $s \in F$ and has value 0 if $s \notin F$. Moreover, if $(s, t) \in f$ we write $s \to t$.

For convenience we sometimes write $\delta(s, \alpha) = S'$, where $S' = \{s' \mid s \to t \to s' \wedge \lambda(t) = \alpha\}$. We extend δ from $(S \times \Lambda) \to 2^S$ to $(2^S \times \Lambda^*) \to 2^S$.

For deterministic automata we denote by $c(s, w)$ the cost of the path starting at s and labeled by $w \in \Lambda^*$. For deterministic automata we denote by $c^F(w)$ the final cost of the path starting at s_0 and labeled by $w \in \Lambda^*$ (e.g. the cost of its last state).

We denote by $P(I, w, F)$ the set of paths from any element of $I \subseteq S$ to any element of $F \subseteq S$, labeled by w.

To each word $w \in L(\mathcal{A})$ we associate a cost $\mathcal{C}(w) = \min_{p \in P(s_0, w, F)} c(p)$, where $c(s_0, t_1, s_1, t_2, \ldots, s_n) = \Sigma_{i=1}^{n} c(t_i) + c^i(s_0) + c^F(s_n)$. For convenience we sometimes write $\mathcal{C}(q, w, q')$ for $\min_{p \in P(q, w, q')}(c(p) - c^i(q) - c^F(q'))$.

B.1 Weight-pushing

Given $\mathcal{A} = (S, T, f, s^0, \lambda, \Lambda, F, c, c^i, c^F)$ a WA, we denote by $d[q]$ the minimal weight of a path from q to F, including final weight. The weight pushing algorithm consists in computing each $d[q]$ and of rewriting the transition weights, initial weight, and final weights in the following way:

$$\forall q \to t \to q',\ q, q' \in S,\ t \in T,\ d[q] \neq +\infty, \qquad c(t) \leftarrow c(t) - d[q] + d[q'] \tag{B.1}$$

$$c^i(s^0) \leftarrow c^i(s^0) + d[s^0] \tag{B.2}$$

$$\forall q \in F,\ d[q] \neq +\infty, \qquad c^F(q) \leftarrow c^F(q) - d[q] \tag{B.3}$$

B.2 ϵ-removal

Given a WA $\mathcal{A} = (S, T, f, s_0, \lambda, \Lambda, F, c, c_i, c_F)$, with ϵ-transitions, given $s \in S$, we denote by $Cl(s)$ the *weighted ϵ-closure* of s. We have:

$$Cl(s) = \{(q, w) \mid P(s, \epsilon, q) \neq \emptyset \text{ and } w = d_\epsilon[s, q]\},$$

where $d_\epsilon[s, q]$ is the minimal weight of a path from s to q, using only ϵ-transitions. Algorithm 7 is Mohri's ϵ-removal algorithm for weighted automata. Notice that λ and c are relations, so they can be considered as sets: $(t, \sigma) \in \lambda$ means $\lambda(t) = \sigma$ and $(t, c_t) \in c$ means $c(t) = c_t$.

B.3 Determinisation

B.3.1 Algorithm

If $q' = \{(q_0, c_0), (q_1, c_1), \ldots, (q_m, c_m)\}$, we denote by $S(q')$ the set $\{q_0, q_1, \ldots, q_m\}$.

Algorithm 7 Mohri's weighted automata ϵ-removal algorithm

```
**Input:** $\mathcal{A} = (S, T, f, s_0, \lambda, \Lambda, F, c, c_i, c_F)$
**Output:** $\mathcal{A}' = (S, T', f', s_0, \lambda', \Lambda, F', c', c_i, c'_F)$
**For** each $p \in S$ **do**
    compute $Cl(p)$
**Endfor**
$T' \leftarrow \{t \in T \mid p \rightarrow t \rightarrow p',\ \lambda(t) \neq \epsilon\}$
$f' \leftarrow \{(x, y) \mid y \in T' \text{ or } x \in T'\}$
$\lambda' \leftarrow \{(t, \lambda(t)) \mid t \in T'\}$
$c' \leftarrow \{(t, c(t) \mid t \in T'\}$
$F' \leftarrow F$
$c'_F \leftarrow c_F$
**For** each $p \in S$ **do**
    **For** each $(q, w) \in Cl(p)$ **do**
        $E \leftarrow \{(t_{pr}, \sigma_{pr}, c_{pr}) \mid \exists t \in T, \exists r \in S,\ \text{such that } (q, t) \in f, (t, r) \in f, \lambda(t) = \sigma_{pr},\ \text{and } c(t) = c_{pr}\}$
        $T' \leftarrow T' \cup \{t_{pr} \mid (t_{pr}, \sigma_{pr}, c_{pr}) \in E\}$
        $f' \leftarrow f' \cup \{(p, t_{pr}) \mid (t_{pr}, \sigma_{pr}, c_{pr}) \in E\} \cup \{(t_{pr}, r) \mid (t_{pr}, \sigma_{pr}, c_{pr}) \in E\}$
        $\lambda' \leftarrow \lambda' \cup \{(t_{pr}, \sigma_{pr}) \mid (t_{pr}, \sigma_{pr}, c_{pr}) \in E\}$
        $c' \leftarrow c' \cup \{(t_{pr}, c_{pr} + w) \mid (t_{pr}, \sigma_{pr}, c_{pr}) \in E\}$
        **if** $q \in F$ **then**
            **if** $p \notin F$ **then**
                $F' \leftarrow F \cup \{p\}$
                $c'_F(p) \leftarrow +\infty$
            **Endif**
            $c'_F(p) \leftarrow \min(c'_F(p), w + c_F(q))$
        **Endif**
    **Endfor**
**Endfor**
```

Algorithm 8 Mohri's weighted automata determinisation algorithm

```
**Input:** $\mathcal{A} = (S, T, f, s_0, \lambda, \Lambda, F, c, c_i, c_F)$
**Output:** $\mathcal{A}' = (S', T', f', s'_0, \lambda', \Lambda, F', c', c'_i, c'_F)$
$s'_0 \leftarrow \{(s_0, 0)\}$
$c'_i(s'_0) \leftarrow c_i(s_0)$
$Q \leftarrow \{s'_0\}$
**While** $Q \neq \emptyset$ **do**
    $p' \leftarrow pop(Q)$
    **Foreach** $\sigma$ such that $\delta(p', \sigma) \neq \emptyset$ **do**
        $w \leftarrow \min\{v + c(t) \mid (p, v) \in p' \wedge p \rightarrow t \wedge \lambda(t) = \sigma\}$
        $q' \leftarrow \{(q, \min\{v + c(t) - w \mid (p, v) \in p' \wedge p \rightarrow t \rightarrow q \wedge \lambda(t) = \sigma\}) \mid q \in \delta(p', \sigma)\}$
        $T' \leftarrow T' \cup \{t_{p'q'}\}$
        $f' \leftarrow f' \cup \{(p', t_{p'q'}), (t_{p'q'}, q')\}$
        $\lambda'(t_{p'q'}) \leftarrow \sigma$
        $c'(t_{p'q'}) \leftarrow w$
        **If** $q' \notin S'$ **then**
            $S' \leftarrow S' \cup \{q'\}$
            **If** $S(q') \cap F \neq \emptyset$ **then**
                $F' \leftarrow F' \cup \{q'\}$
                $c'_F(q') \leftarrow \min\{v + c_F(q) \mid (q, v) \in q' \wedge q \in F\}$
            **Endif**
            $push(Q, q')$
        **Endif**
    **Endfor**
**Endwhile**
```

For $q' = \{(q_0, c_0), (q_1, c_1), \ldots, (q_m, c_m)\}$ and a label σ we write: $\delta(q', \sigma) = \{q'' \in S | \exists (q, c) \in q', \exists t \in T, q \to t \to q'', \lambda(t) = \sigma\} = \delta(S(q'), \sigma)$.

Algorithm 8 is Mohri's determinisation algorithm for weighted automata.

Theorem 2. *When Algorithm 8 terminates, the result is a deterministic automaton $\mathcal{A}'$ which is equivalent to $\mathcal{A}$.*

Proof. It is obvious that, when Algorithm 8 terminates, the result is a deterministic automaton $\mathcal{A}'$. We now prove that this automaton is equivalent to $\mathcal{A}$: we first show that $\mathcal{A}$ and $\mathcal{A}'$ accept the same words, and then we show that for a given word w, accepted by $\mathcal{A}$ and $\mathcal{A}'$, the cost corresponding to w in $\mathcal{A}'$ is the minimal cost corresponding to w in $\mathcal{A}$ (e.g. $\mathcal{C}(w)$).

One can notice that, by construction, the following holds in $\mathcal{A}'$ for any $w \in \Lambda^*$:

$$\begin{aligned}
\delta'(s'_0, w) &= \bigcup_{q \in \delta(s_0, w)} \{(q, c_{q,w})\} \\
c_{q,w} &= \mathcal{C}(s_o, w, q) - \min_{q' \in \delta(s_0, w)} (\mathcal{C}(s_0, w, q')).
\end{aligned}$$

A word w is accepted in $\mathcal{A}$ if and only if $\delta(s_0, w) \cap F \neq \emptyset$. From the above definition of δ' one can notice that w is accepted if and only if $\delta'(s'_0, w)$ contains an element of the form $(q, c_{q,w})$, with $q \in F$. This is the definition of F'. Thus $\mathcal{A}$ and $\mathcal{A}'$ accept exactly the same sets of words.

Let w be a word accepted by $\mathcal{A}$ (and thus $\mathcal{A}'$). From the algorithm one can notice that:

$$c'_F(\delta'(s'_0, w)) = \min_{q \in \delta(s_0, w) \cap F} (c_i(s_0) + \mathcal{C}(s_0, w, q) + c_F(q)) - \min_{q' \in \delta(s_0, w)} (\mathcal{C}(s_0, w, q')) - c'_i(s'_0).$$

And thus:

$$c'_F(\delta'(s'_0, w)) = \mathcal{C}(w) - \min_{q' \in \delta(s_0, w)} (\mathcal{C}(s_0, w, q')) - c'_i(s'_0).$$

Finally:

$$c'_i(s'_0) + \min_{q' \in \delta(s_0, w)} (\mathcal{C}(s_0, w, q')) + c'_F(\delta'(s'_0, w)) = \mathcal{C}(w).$$

□

B.3.2 Remarks on determinisability

We introduce a notion of distance between words: $\forall u, v \in \Lambda^*$, $d(u, v) = |u| + |v| + 2|pre(u, v)|$, where $pre(u, v)$ is the longest common prefix of u and v.

Definition 19. *We say that $\mathcal{C}$ has* bounded variation *if and only if there exists K such that for all $u, v \in L(\mathcal{A})$ we have:*

$$\frac{|\mathcal{C}(u) - \mathcal{C}(v)|}{d(u, v)} \leq K.$$

Lemma 15. *If $\mathcal{A} = (S, T, f, s_0, \lambda, \Lambda, F, c, c_i, c_F)$ is determinisable, then $\mathcal{C}$ has bounded variation.*

Proof. Let $\mathcal{A} = (S, T, f, s_0, \lambda, \Lambda, F, c, c_i, c_F)$ be a determinisable automaton. By definition there exists a deterministic automaton $\mathcal{A}' = (S', T', \to', s'_0, \lambda', \Lambda, F', c', c'_i, c'_F)$ such that $\mathcal{C}' : L(\mathcal{A}') \to \mathbb{R}^+$ is equal to $\mathcal{C} : L(\mathcal{A}) \to \mathbb{R}^+$.

We define L and R as follows:

$$L = \max_{t \in T'} c'(t),$$

$$R = \max_{q, q' \in F'} (c'_F(q) - c'_F(q')).$$

Let u_1 and u_2 be two elements of $L(\mathcal{A}')$, by definition of d (the distance between words) we have:

$$\exists u \in \Lambda^* \text{ such that } u_1 = uv_1,\ u_2 = uv_2, \text{ and } |v_1| + |v_1| = d(u_1, u_2).$$

Moreover, recalling that $\mathcal{A}'$ is deterministic:

$$\begin{aligned}\mathcal{C}'(u_1) &= c'_i(s'_0) + c'(s'_0, u) + c'(\delta(s'_0, u), v_1) + c'_F(u_1)\\ \mathcal{C}'(u_2) &= c'_i(s'_0) + c'(s'_0, u) + c'(\delta(s'_0, u), v_2) + c'_F(u_2).\end{aligned}$$

Thus:

$$\begin{aligned}|\mathcal{C}'(u_1) - \mathcal{C}'(u_2)| &= |(c'_i(s'_0) + c'(s'_0, u) + c'(\delta(s'_0, u), v_1) + c'_F(u_1))\\ &\quad -(c'_i(s'_0) + c'(s'_0, u) + c'(\delta(s'_0, u), v_2) + c'_F(u_2))|\\ &= |c'(\delta(s'_0, u), v_1) - c'(\delta(s'_0, u), v_2) + c'_F(u_1) - c'_F(u_2)|\\ &\leq L \times (|v_1| + |v_2|) + R\\ &= L \times d(u_1, u_2) + R.\end{aligned}$$

Hence, for $u_1 \neq u_2$ (in this case $d(u_1, u_2) \geq 1$): $|\mathcal{C}'(u_1) - \mathcal{C}'(u_2)| \leq (L + R) \times d(u_1, u_2)$.
The fact that $\mathcal{C}' = \mathcal{C}$ ends the proof:

$$\frac{|\mathcal{C}(u_1) - \mathcal{C}(u_2)|}{d(u_1, u_2)} \leq (L + R).$$

□

Definition 20. *An automaton $\mathcal{A} = (S, T, \to, s_0, \lambda, \Lambda, F)$ is said to be* trim *if and only if any element of S is on a path from s_0 to F.*

Definition 21. *An automaton $\mathcal{A} = (S, T, \to, s_0, \lambda, \Lambda, F)$ is said to be* unambiguous *if and only if for any word $w \in \Lambda^*$ there exists at most one path labeled by w from s_0 to F.*

Lemma 16. *A trim, unambiguous, weighted automaton $\mathcal{A}$ such that $\mathcal{C}$ has bounded variation has the twin property.*

Proof. Let $\mathcal{A} = (S, T, f, s^0, \lambda, \Lambda, F, c, c^i, c^F)$ be a trim and unambiguous weighted automaton, such that $\mathcal{C}$ has bounded variation.

Let q and q' be two elements of S; and u and v be two words from Λ^* such that:

$$\{q, q'\} \subseteq \delta(s_0, u),\ q \in \delta(q, v), \text{ and } q' \in \delta(q', v).$$

One can notice that if q, q', u, and v can not be found, then $\mathcal{A}$ is twin.

As $\mathcal{A}$ is trim, one can find two words w and w' in Λ^* such that:

$$\delta(q, w) \cap F \neq \emptyset \text{ and } \delta(q', w') \cap F \neq \emptyset.$$

Moreover, as $\mathcal{C}$ has bounded variation, we have:

$$\exists K \geq 0 \text{ such that } \forall k \geq 0,\ |\mathcal{C}(uv^kw) - \mathcal{C}(uv^kw')| \leq K,$$

because, for all k, $d(uv^kw, uv^kw') = d(w, w')$.

As $\mathcal{A}$ is unambiguous, there is a unique path from s_0 to F labeled by uv^kw: the path reaching q (this is also true for uv^kw' and q'). Thus:

$$\begin{aligned}\mathcal{C}(uv^kw) &= \mathcal{C}(uw) + k\mathcal{C}(q, v, q)\\ \mathcal{C}(uv^kw') &= \mathcal{C}(uw') + k\mathcal{C}(q', v, q').\end{aligned}$$

This leads to the following:

$$\exists K \geq 0, \text{ such that } \forall k \geq 0,\ |(\mathcal{C}(uw) + k\mathcal{C}(q, v, q)) - (\mathcal{C}(uw') + k\mathcal{C}(q', v, q'))| \leq K.$$

Hence:

$$k(\mathcal{C}(q,v,q) - \mathcal{C}(q',v,q')) = 0.$$

Finally:

$$\mathcal{C}(q,v,q) = \mathcal{C}(q',v,q')$$

The automaton $\mathcal{A}$ is twin. $\square$

Lemma 17 (Pumping lemma)**.** *Let $\mathcal{A} = (S,T,f,s_0,\lambda,\Lambda,F,c,c_i,c_F)$ be a WA. Let w be a word over Λ^*, such that: there exists a path π from $p \in S$ to $q \in S$, labeled by w and a path π' from $p' \in S$ to $q' \in S$, also labeled by w. If $|w| > |S|^2 - 1$, then there exists u_1, u_2, u_3 three words over Λ^*, and p_1, p'_1 two states of S such that: $|u_2| > 0$, $u_1u_2u_3 = w$, and π and π' are factorisable as follows:*

$$\begin{array}{rcl} \pi & = & p \overset{u_1}{\rightsquigarrow} p_1 \overset{u_2}{\rightsquigarrow} p_1 \overset{u_3}{\rightsquigarrow} q \\ \pi' & = & p' \overset{u_1}{\rightsquigarrow} p'_1 \overset{u_2}{\rightsquigarrow} p'_1 \overset{u_3}{\rightsquigarrow} q'. \end{array}$$

Proof. The proof is based on the products of $\mathcal{A}$ by himself, defined as: $\mathcal{A}\times\mathcal{A} = (S\times S, T', f', (s_0,s_0), \lambda', \Lambda, F\times F, c', c'_i, c'_F)$, where:

- $t = (t_1,t_2) \in T'$ if and only if $t_1 \in T$, $t_2 \in T$, and $\lambda(t_1) = \lambda(t_2)$;
- $\lambda'(t) = \lambda(t_1)$;
- $(x_1,x_2) \rightarrow (t_1,t_2) \rightarrow (x'_1,x'_2)$ if and only if $x_i \rightarrow t_i \rightarrow x'_i$;
- c', c'_i and c'_F are not necessary for the proof.

Let π and π' be two paths in $\mathcal{A}$, with same length (in terms of transitions) strictly greater than $|S|^2 - 1$.

$$\begin{array}{rcl} \pi & = & q_0 \rightarrow t_0 \rightarrow q_1 \rightarrow \cdots \rightarrow t_{m-1} \rightarrow q_m \\ \pi' & = & q'_0 \rightarrow t'_0 \rightarrow q'_1 \rightarrow \cdots \rightarrow t'_{m-1} \rightarrow q'_m \end{array}$$

If π and π' are labeled by the same word w, the following path exists in $\mathcal{A}\times\mathcal{A}$:

$$\Pi = (q_0,q'_0) \rightarrow' (t_0,t'_0) \rightarrow' (q_1,q'_1) \rightarrow' \cdots \rightarrow' (t_{m-1},t'_{m-1}) \rightarrow' (q_m,q'_m).$$

Moreover $\mathcal{A}\times\mathcal{A}$ has exactly $|S|^2$ states. Hence, Π has at least one cycle. This proves the lemma. $\square$

Lemma 18. *If $\mathcal{A}$ is twin then Algorithm 8 terminates when applied to $\mathcal{A}$.*

Proof. Let $\mathcal{A} = (S,T,f,s_0,\lambda,\Lambda,F,c,c_i,c_F)$ be a twin weighted automaton.

If Algorithm 8 does not terminate when applied to $\mathcal{A}$ it means that there exists at least a set $Q = \{q_0,\ldots,q_m\}$ of elements from S such that the algorithm produces an infinite number of distinct sets: $\{(q_0,c_0),\ldots,(q_m,c_m)\}$. Moreover there is at least two elements in Q (because, by construction, there always exists j such that $c_j = 0$). We denote by $A \subseteq \Lambda^*$ the set of words w such that there is a path in $\mathcal{A}'$ from s'_0 to q', where the states of S present in q' are exactly the states of Q. For any $w \in A$ we write $q' = \{(q_0,c_0(w)),\ldots,(q_m,c_m(w))\}$. As A is infinite (by definition of Q) and for any $w \in A$ there exists j such that $c_j(w) = 0$: there exists j_0 such that $c_{j_0}(w) = 0$ for an infinite number of elements w of A. Without loss of generality, let $j_0 = 0$. We denote by $B \subseteq A$ the infinite set of words $w \in A$ such that $c_0(w) = 0$. As B is infinite, there exists j such that $c_j(w)$ takes an infinite number of different values for $w \in B$. Without loss of generality we assume that $j = 1$. We denote by $C \subseteq B$ an infinite set of words w such that the $c_1(w)$ are all different (and $c_0(w) = 0$).

Let $R(q_0,q_1) = \{c(\pi_1) - c(\pi_0) \mid \pi_i$ is a path from s_0 to q_i labeled by $w \wedge |w| \leq |S|^2 - 1\}$. This set is finite (because the number of words $w \in \Lambda^*$ of size smaller than $|S|^2 - 1$ is finite).

The idea of the proof is to show that $\{c_1(w) \mid w \in C\} \subseteq R(q_0,q_1)\}$. Which is in contradiction with the fact that C is infinite, and thus proves the termination of Algorithm 8.

Let w be a word from C. We denote by π_0 a minimum cost path from s_0 to q_0 labeled by w. We denote by π_1 a minimum cost path from s_0 to q_1 labeled by w. By definition of C we have: $c_1(w) = c(\pi_1) - c(\pi_0)$.

If $|w| \leq |S|^2 - 1$ then $c_1 \in R(q_0, q_1)$.

Else, $|w| > |S|^2 - 1$. Thus Lemma 17 allows to factorize π_0 and π_1:

$$\begin{aligned} \pi_0 &= s_0 \overset{u_1}{\rightsquigarrow} p_0 \overset{u_2}{\rightsquigarrow} p_0 \overset{u_3}{\rightsquigarrow} q_0 \\ \pi_1 &= s_0 \overset{u_1}{\rightsquigarrow} p_1 \overset{u_2}{\rightsquigarrow} p_1 \overset{u_3}{\rightsquigarrow} q_1. \end{aligned}$$

As p_0 and p_1 are twins (because $\mathcal{A}$ is twin) we have: $\mathcal{C}(p_0, u_2, p_0) = \mathcal{C}(p_1, u_2, p_1)$.

We denote by π'_0 and π'_1 the following paths:

$$\begin{aligned} \pi'_0 &= s_0 \overset{u_1}{\rightsquigarrow} p_0 \overset{u_3}{\rightsquigarrow} q_0 \\ \pi'_1 &= s_0 \overset{u_1}{\rightsquigarrow} p_1 \overset{u_3}{\rightsquigarrow} q_1. \end{aligned}$$

As π_0 and p_1 are minimal cost paths we have:

$$\begin{aligned} c(\pi_0) &= c(\pi'_0) + \mathcal{C}(p_0, u_2, p_0) \\ c(\pi_1) &= c(\pi'_1) + \mathcal{C}(p_1, u_2, p_1). \end{aligned}$$

And thus:

$$c_1(w) = c(\pi'_1) - c(\pi'_0).$$

Using induction one can find $\tilde{\pi}_0$ and $\tilde{\pi}_1$, corresponding to words of length smaller or equal to $|S|^2 - 1$, and such that:

$$c_1(w) = c(\tilde{\pi}_1) - c(\tilde{\pi}_0),$$

which is in $R(q_0, q_1)$, by definition of this set.

Thus $c_1(w) \in R(q_0, q_1)$, which is in contradiction with infinity of C. Hence Algorithm 8 terminates. □

Theorem 3. *A trim, unambiguous, weighted automaton is determinisable if and only if it is determinisable by Algorithm 8.*

Proof. Let $\mathcal{A}$ be a trim, unambiguous, determinisable, weighted automaton. By Lemma 15, $\mathcal{C}$ has bounded variation, and thus, by Lemma 16, $\mathcal{A}$ is twin. Hence, by Lemma 18 this ensure that Algorithm 8 terminates. This proves that $\mathcal{A}$ is determinisable by Algorithm 8.

The reverse is obvious (see Theorem 2). □

If it is possible to find for any WA an equivalent trim WA, it is unfortunately not always possible to find an equivalent unambiguous WA. In [KLMP04], [KM05], and [LM06] are given some more results about this unambiguity hypothesis, how it can be relaxed, and necessary conditions for determinisability of WA.