Turbo Planning

Loïg Jezequel, Eric Fabre

Turbo Planning

Loig Jezequel * **Eric Fabre** **

* *ENS Cachan Bretagne, IRISA, Rennes, France (e-mail: loig.jezequel@irisa.fr).*
** *INRIA Rennes Bretagne Atlantique, IRISA, Rennes, France (e-mail: eric.fabre@irisa.fr)*

Abstract: The complexity of planning problems comes from the size of the state graph of the systems, which suggests to consider factored (or distributed) solutions. We previously proposed a solution of this kind which revealed to be very efficient on problems where components have a sparse interaction. This work explores a step further in this direction. The idea is to extend the celebrated turbo algorithms, extremely successful to decode large-scale sparse error correcting codes. The paper proposes an adaptation of this technique to the setting of cost-optimal factored planning, and illustrates its behavior on large randomly generated systems.

Keywords: discrete event systems, planning, weighted automata, turbo algorithms, distributed algorithms

1. INTRODUCTION

Planning consists of driving a system from an initial state to a goal state by organizing a set of actions. It corresponds, in fact, to control when all actions are controllable. Traditionally, planning has been solved using "best first" search algorithms such as A^* (Hart et al. (1968)). However, recently, the intrinsic concurrency of planning problems has been exploited to represent plans (solutions) by partial orders of actions rather than sequences of actions (Blum and Furst (1995)). The interest of this approach is to significantly reduce the size of the space to explore for finding a solution. Finally, in the last few years, solutions have been proposed for finding these plans as partial orders of actions in a modular way (Amir and Engelhardt (2003)). This approach is called factored planning. It exploits the fact that in many planning problems some variables are almost independent. This allows one to solve these problems by parts. In the worst case, factored planning has the same complexity as planning. However, it can grant significant efficiency gain: the parts (or components) of a problem are in general exponentially smaller than the original problem.

In Fabre and Jezequel (2009) we proposed a new approach to factored planning. This approach was based on weighted automata calculus and on a message passing algorithm in networks of such automata. An important innovation of this approach with regards to previous results is that it allows one to perform *cost-optimal* factored planning. In other words it allows not only to find a plan but to find the best one. In Fabre et al. (2010) we proposed an implementation of our approach and tested it on classical planning benchmarks, showing that this approach is of interest in practice. However, this approach works only for problems where the graph defined by the interactions between components is a tree. This restriction, even if it is natural for propagating constraints, is quite strong. It significantly reduces the number of problems on which our method can be used.

There exists, though, a family of algorithms which give astonishingly good results when dealing with complex interaction graphs (containing cycles) with sparse interaction. These algorithms are mostly known for their use in coding theory (Berrou et al. (1993)). They are closely related (McEliece et al. (1998)) to Pearl's belief propagation algorithm (Pearl (1982)), which is well-known in the artificial intelligence community. Nowadays, turbo algorithms are used with great success in many areas, but few results exist explain their efficiency, in particular in our setting.

This paper is a study of turbo algorithms in the context of factored planning. Our goal is to show that these algorithms are of interest in the domains of planning and of cost-optimal planning). We recall, however, that planning has been proved to be a PSPACE-complete problem (Bylander (1991)). So there will necessarily exist cases where turbo algorithms will not be that efficient. We first formally describe the problems we address (Section 2). Then we consider the use of turbo algorithms for solving factored planning problems (Section 3) and their cost-optimal counterpart (Section 4). In each case experimental results are provided that demonstrate the interest of investigating the use of turbo algorithms in the domain of planning.

2. MESSAGE PASSING ALGORITHMS FOR COST-OPTIMAL FACTORED PLANNING

This section describes the problem we address in this paper: factored planning. It also presents a previously-explored way to solve this problem using an instance of a message passing algorithm.

2.1 Factored planning problems

As its name suggests, a factored planning problem is a planning problem represented by a set of subproblems or factors. If planning problems are represented by finite

automata, factored planning problems are represented by networks of automata (that are themselves a synchronous product of automata).

Formally, a planning problem can be modeled as an automaton $\mathcal{A} = (S, \Sigma, T, I, F)$ where S is a finite set of states, Σ is a finite set of labels (or actions), $T \subseteq S \times \Sigma \times S$ is a transition relation, $I \subseteq S$ is a set of initial states, and $F \subseteq S$ is a set of final states. The elements of T are denoted by $t = (t^-, \sigma, t^+)$. The goal in such a problem is to find an *accepted path* in $\mathcal{A}$. That is, a sequence of transitions $p = t_1 \dots t_n$ such that: $\forall 1 \leq i < n, t_i^+ = t_{i+1}^-$ (it is a path), $t_1^- \in I$, and $t_n^+ \in F$ (it is accepted). Such an (accepted) path is associated with an (accepted) word $\sigma(p) = \sigma_1 \dots \sigma_n$. Notice that in the case of a deterministic automaton, any accepted word corresponds to a single accepted path. In the following, the language $\mathcal{L}(\mathcal{A})$ of an automaton $\mathcal{A}$ is the set of its words.

It is possible to define a notion of composition of such automata, called the *synchronous product*. Given two automata $\mathcal{A}_1$ and $\mathcal{A}_2$, their synchronous product $\mathcal{A}_1 \times \mathcal{A}_2$ is the automaton $\mathcal{A}$ such that: $S = S_1 \times S_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $T = T_{\Sigma_1 \cap \Sigma_2} \cup T_{\Sigma_1 \setminus \Sigma_2} \cup T_{\Sigma_2 \setminus \Sigma_1}$, $I = I_1 \times I_2$, and $F = F_1 \times F_2$. Where $T_{\Sigma_1 \cap \Sigma_2} = \{((s_1, s_2), \sigma, (s_1', s_2')) \ : \ \sigma \in \Sigma_1 \cap \Sigma_2 \wedge (s_1, \sigma, s_1') \in T_1 \wedge (s_2, \sigma, s_2') \in T_2\}$ is a set of shared transitions, and $T_{\Sigma_1 \setminus \Sigma_2} = \{((s_1, s_2), \sigma, (s_1', s_2)) \ : \ \sigma \in \Sigma_1 \setminus \Sigma_2 \wedge (s_1, \sigma, s_1') \in T_1 \wedge s_2 \in S_2\}$ (resp. $T_{\Sigma_2 \setminus \Sigma_1}$ symmetrically defined) is a set of private transitions from $\mathcal{A}_1$ (resp. $\mathcal{A}_2$).

A factored planning problem is then an automaton $\mathcal{A} = \mathcal{A}_1 \times \dots \times \mathcal{A}_n$ represented by its factors (or components) $\mathcal{A}_1 \dots \mathcal{A}_n$. The goal is to find a tuple of paths $(p_1, \dots, p_n)$, one for each component, such that there exists a path p in $\mathcal{A}$ with the property that $\forall 1 \leq i \leq n, \sigma(p)_{|\Sigma_i} = \sigma_i(p_i)$ (where $w_{|\Sigma}$ represents w with all labels not in Σ removed). If possible, this has clearly to be done without computing $\mathcal{A}$ as it is in general exponentially larger than its components.

Any factored planning problem is associated with its *interaction graph*. This graph is defined as follows: its vertices are the automata that constitute the problem and its edges are such that there is an edge between $\mathcal{A}_i$ and $\mathcal{A}_j$ if and only if $j \neq i$ and $\Sigma_i \cap \Sigma_j \neq \emptyset$ (in this case $\mathcal{A}_i$ and $\mathcal{A}_j$ are said to be neighbors).

It is possible to consider planning problems with costs associated with transitions by a cost function $c : T \to \mathbb{R}^+$. This allows one to associate costs to paths: the cost of a path $p = t_1 \dots t_n$ would be $\sum_{1 \leq i \leq n} c(t_i)$. Synchronous product is also well-defined for automata with costs: it is sufficient to add costs of shared actions. The goal will be to find paths with minimal cost or tuples of compatible paths minimizing the sum of the costs of the elements of the tuples. In this case, one speaks about *cost-optimal planning* or *cost-optimal factored planning*.

2.2 Message passing algorithm

Previously (Fabre and Jezequel (2009)), we proposed a modular method for solving factored planning problems efficiently. This method, based on the use of message passing algorithms in networks of automata, along with its limitations, is briefly recalled in the following. The rest of the paper aims at trying to handle these limitations. For simplicity of presentation, from now on automata are considered to be minimal (this allows one to identify automata and languages). Notice that the message passing algorithm exists for weighted automata as well, allowing one to solve cost-optimal factored planning problems. Issues pertaining to minimization of weighted automata are discussed in Fabre et al. (2010).

The message passing algorithm relies on the synchronous product and a second operation, called *projection*. This operation is defined as follows: $\mathcal{P}_{\Sigma'}(\mathcal{A}) = \text{MIN}(\text{RED}_{\Sigma \setminus \Sigma'}(\mathcal{A}))$. Here, MIN is a minimization operation. And $\text{RED}_{\Sigma \setminus \Sigma'}$ is a $\Sigma \setminus \Sigma'$-reduction operation.

The interesting fact relating projection and factored planning is that for $\mathcal{A} = \mathcal{A}_1 \times \dots \times \mathcal{A}_n$, the projection $\mathcal{A}_i' = \mathcal{P}_{\Sigma_i}(\mathcal{A})$ contains exactly the paths of $\mathcal{A}_i$ which are part of a solution to $\mathcal{A}$:

- any path p_i of $\mathcal{A}_i'$ is such that a path p exists in $\mathcal{A}$ verifying $\sigma(p)_{|\Sigma_i} = \sigma_i(p_i)$, and
- any path p of $\mathcal{A}$ is such that a path p_i exists in $\mathcal{A}_i'$ verifying $\sigma_i(p_i) = \sigma(p)_{|\Sigma_i}$.

Thus, computing $\mathcal{A}_i'$ is of interest for solving a factored planning problem.

Consider the following algorithm (Algorithm 1):

forall $\mathcal{A}_i$ **do**
  forall $j \in \mathcal{N}(i)$ **do**
    $\mathcal{M}_{i,j} \leftarrow \mathcal{P}_{\Sigma_j}(\mathcal{A}_i)$
until stabilization do
  select an $\mathcal{M}_{i,j}$
  $\mathcal{M}_{i,j} \leftarrow \mathcal{P}_{\Sigma_j}(\mathcal{A}_i \times \Pi_{k \in \mathcal{N}(i), k \neq j} \mathcal{M}_{k,i})$
forall $\mathcal{A}_i$ **do**
  $\mathcal{A}_i'' \leftarrow \mathcal{A}_i \times \Pi_{j \in \mathcal{N}(i)} \mathcal{M}_{j,i}$

It takes as input a factored planning problem $\mathcal{A}$ given as a set of automata $\{\mathcal{A}_1, \dots, \mathcal{A}_n\}$. When it converges, Algorithm 1 returns a collection of $\mathcal{A}_i''$ which are such that $\mathcal{L}(\mathcal{A}_i') \subseteq \mathcal{L}(\mathcal{A}_i'') \subseteq \mathcal{L}(\mathcal{A}_i)$. The principle of Algorithm 1 is to propagate some messages (the $\mathcal{M}_{i,j}$) between the different automata. The message $\mathcal{M}_{i,j}$ (sent from i to j) carries information about the constraints of the system on $\mathcal{A}_j$ via $\mathcal{A}_i$. The set $\mathcal{N}(i)$ is the set of neighbors of $\mathcal{A}_i$: $j \in \mathcal{N}(i)$ if and only if there is an edge between $\mathcal{A}_i$ and $\mathcal{A}_j$ in the interaction graph of the problem.

The convergence of this message passing algorithm is strongly related to the interaction graph of the problems considered. As soon as the interaction graph of a factored planning problem has a tree shape, the message passing algorithm can be proven to converge. Moreover, it can be shown that the $\mathcal{A}_i''$ computed by the algorithm are such that $\mathcal{A}_i'' = \mathcal{A}_i'$; however, when the interaction graph is not a tree, nothing is ensured.

2.3 Solution extraction

If Algorithm 1 converges, one can extract a solution to the factored planning from the $\mathcal{A}_i''$. When the interaction graph is a tree this is straightforward: it suffices to chose a path p_i in any $\mathcal{A}_i'' = \mathcal{A}_i'$. Then a path p_j compatible with p_i is chosen in each $\mathcal{A}_j''$ such that $j \in \mathcal{N}(i)$. This can be done by computing $p_i \times \mathcal{A}_j''$. Then for each j a

path p_k compatible with p_j is chosen in each $\mathcal{A}''_k$ such that $k \in \mathcal{N}(j)$ and $k \neq i$. This process continues until a path has been found in each component. Recall that for all i, $\mathcal{A}''_i = \mathcal{A}'_i$ and since the interaction graph is a tree, one can ensure that the tuple of paths found is indeed a solution of the factored planning problem under consideration.

When the interaction graph is not a tree, one can not ensure that $\mathcal{A}''_i = \mathcal{A}'_i$. And, even if this is the case, one can not ensure that a solution will be found using the method described above. Finding a solution may require the use of *backtracking*. At some step of the extraction, one may be unable to choose a path p_j in an $\mathcal{A}''_j$, compatible with the paths $p_{i_1}, \ldots, p_{i_k}$ in its neighbors. This would require the modification of at least one of the p_{i_ℓ} (and potentially the paths that where used for computing p_{i_ℓ}).

In the remainder of the paper we (mostly experimentally) examine *turbo algorithms*, which basically consists of executing Algorithm 1 on problems where the interaction graph is not a tree.

3. TURBO ALGORITHMS FOR CONSTRAINT SOLVING

The idea when using turbo algorithms for solving planning problems is to abandon the quest for the exact $\mathcal{A}'_i = \mathcal{P}_{\Sigma_i}(\mathcal{A})$ by applying Algorithm 1 on problems with cycles of interaction. Rather, the objective is to compute a good refinement (in the sense that it contains fewer plans) $\mathcal{A}''_i$ of each $\mathcal{A}_i$. Recall that, in all cases, this refinement will be an over-approximation of $\mathcal{A}'_i$: it will always contain all local plans that are part of a solution of the problem considered. The interest of computing these refinements – rather than directly searching for a solution from the $\mathcal{A}_i$ – is to reduce the necessity for backtracking by removing paths that are clearly not part of distributed solutions and by ensuring local consistency. A path in $\mathcal{A}''_i$ will at least be compatible with a path in each of its neighbors.

This section recalls some known results about turbo algorithms for constraint solving, and shows how these results relate to the specific case of planning. Finally our experimental setting is presented and experimental results are given.

3.1 Conditions for convergence

In Fabre (2003) turbo algorithms are studied in detail for the case of "systems defined by local constraints". Factored planning problems belong to this class of systems. Hence, if a partial order $\sqsubseteq$ exists on automata that verify the following axioms:

$$\exists \mathbb{I}, \forall \mathcal{A}, \mathcal{A} \sqsubseteq \mathbb{I}, \quad (1)$$

$$\forall \mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_1 \sqsubseteq \mathcal{A}_2 \Rightarrow \mathcal{A}_1 \times \mathcal{A}_3 \sqsubseteq \mathcal{A}_2 \times \mathcal{A}_3, \quad (2)$$

$$\forall \mathcal{A}_1, \mathcal{A}_2, \forall \Sigma, \mathcal{A}_1 \sqsubseteq \mathcal{A}_2 \Rightarrow \mathcal{P}_\Sigma(\mathcal{A}_1) \sqsubseteq \mathcal{P}_\Sigma(\mathcal{A}_2), \quad (3)$$

then existence of a unique stabilization point for Algorithm 1 on any factored planning problem is ensured (Lemma 7 of Fabre (2003)). Axiom 1 ensures the existence of a least informative system, Axiom 2 ensures that synchronous product adds the same amount of information to all systems, and Axiom 3 ensures that projection does not add information. The idea behind these axioms is that applying Algorithm 1 to a factored planning problem will result into messages having a decreasing evolution with respect to $\sqsubseteq$.

There is an obvious partial order on automata verifying these axioms: $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ if and only if $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$.

If, in addition to the existence of a partial order verifying axioms 1, 2, and 3, the number of possible automata is bounded, convergence of Algorithm 1 is ensured within a finite number of steps on any factored planning problem. Messages decrease for $\sqsubseteq$ and there is a bounded number of possible messages (Theorem 3 of Fabre (2003)).

It is, however, not the case that the number of languages (and thus of automata) over a given alphabet is bounded. Thus, it is not possible to ensure convergence of Algorithm 1 on any factored planning problem. In fact, there are examples of such problems for which convergence is not achieved. Consider, for example, the problem in Figure 1. This problem has no solution: reaching the goal in $\mathcal{A}_1$ implies the firing of an α, which implies the firing of a γ in $\mathcal{A}_3$ and thus a β in $\mathcal{A}_2$, which enforces the firing of a second α in $\mathcal{A}_1$ and thus a second γ, and so on.

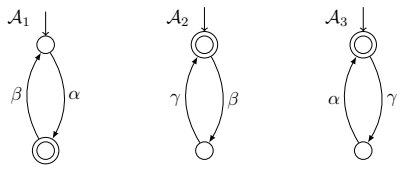

Fig. 1. A factored planning problem such that turbo algorithms do not converge.

3.2 Ensuring convergence in all cases

One needs however to decide convergence in all cases. This can be achieved by using a notion of distance d between languages and decide convergence as soon as the messages are stable up to some constant with respect to this distance. In other words, Algorithm 1 will stop as soon as the updating of each message $\mathcal{M}_{i,j}$ results in the new message $\mathcal{M}'_{i,j}$ such that $d(\mathcal{M}_{i,j}, \mathcal{M}'_{i,j}) \leq \epsilon$, for some small ϵ.

Denote by $\mathcal{L}_n(\mathcal{A}) = \{w \in \mathcal{L}(\mathcal{A}) \ : \ |w| = n\}$ the set of words of length n belonging to $\mathcal{L}(\mathcal{A})$. A distance which seems reasonable is then the following:

$$d(\mathcal{A}_1, \mathcal{A}_2) = \sum_{n=0}^{\infty} \frac{1}{2^n} 1_{\mathcal{L}_n(\mathcal{A}_1) \neq \mathcal{L}_n(\mathcal{A}_2)}$$

where $1_{\mathcal{L} \neq \mathcal{L}'} = 1$ as soon as $\mathcal{L} \neq \mathcal{L}'$ and 0 in other cases. Using this distance almost corresponds to looking only at words under a given length ℓ for checking convergence. As for any alphabet, the set of words of length smaller than ℓ is finite, and any update of messages in Algorithm 1 can only remove words from the considered message, convergence is granted using this distance. Moreover, in any planning problem, there exists a bound (difficult to compute in a modular way) such that if in a component there exists no plan with length smaller than this bound, the problem has no solution. This ensures that convergence with respect to this distance (using a correct bound) is sufficient for deciding the absence of a solution.

3.3 Experimental results

In this section we present experimental results obtained by running Algorithm 1 on randomly generated factored planning problems for which interaction graphs have cycles. Our goal is to estimate if the over-approximations of the $\mathcal{A}'_i$ computed by the message passing algorithm are of interest for planning. That is, if they allow us to find solutions with few backtrackings.

In all our experiments we randomly generated some factored planning problems. This is done by choosing a shape for the interaction graph of the problem, and then, for each node of this graph, randomly generating an automaton. These automata have up to 20 states and up to three time as much transitions. Each automaton shares 2 different labels with each of its neighbors in the interaction graph. Once a problem is generated, we check if it is not trivially solvable (that is no solution can be found without backtracking). If it is not trivially solvable we check if it is solvable by searching for a path in the full product $\mathcal{A}$ of the component automata. Only problems which are not trivially solvable but still have a solution are considered for our experiments (doing this we select the 5-10% "most complicated" problems among the ones we generate).

Experiment 1: automata on a circle. For our first experiment we chose a configuration of problems know to be favorable for turbo methods: a circle. Its setting is as follows: we consider a circle of n random automata $\mathcal{A}_0, \ldots, \mathcal{A}_{n-1}$ such that $\forall 0 \leq i < n$ one has $\Sigma_i \cap \Sigma_{(i+1)\%n} \neq \emptyset$ (here % stands for modulo) and $\forall j, |i-j| > 1$ it is the case that $\Sigma_i \cap \Sigma_j = \emptyset$. We call an *iteration* an update of all messages. Iterations are performed following the circle, as depicted in Figure 2 for five automata. Such iterations are repeated until stabilization of the messages.As soon as stabilization is reached an attempt to find a solution is initiated. It is done using the method presented in Section 2.3 without using backtracking. For this reason it is not always the case that a solution will be found. Our goal using this method is to get an idea of the quality of the computed $\mathcal{A}''_i$: if in many cases a solution is found, it is likely that in practice few backtracking steps will be needed for finding solutions from updated automata.

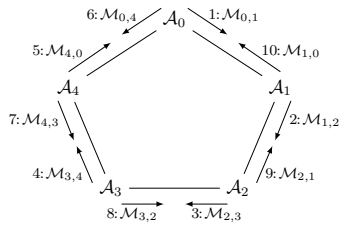

Fig. 2. Interaction graph and propagation of messages during one iteration for 5 automata. Numbers on the left of messages indicate the order of updates.

Figure 3 presents results obtained in this setting for circles of 3 to 15 automata. In each case, 50 cycles were considered. For each circle size the percentage of problems for which convergence occurred in 1, 2, or 3 iterations is represented, as well as the percentage of problems in which a solution has been found.

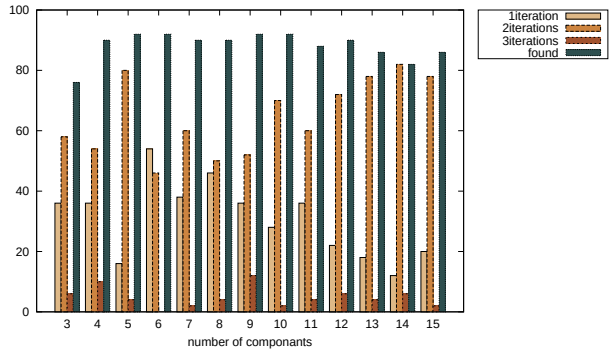

Fig. 3. Experimental results for circles of 3 to 15 automata.

We limited our experiments to cycles of 15 automata because of the time consuming method used for selecting problems. We however performed some experiments with larger circles, as Algorithm 1 is able to handle them. The results are presented in Figure 4. For each value of n, 50 cycles were generated but no selection was done among them. Due to the limitations presented above, we cannot give the percentage of cases where a solution was found.

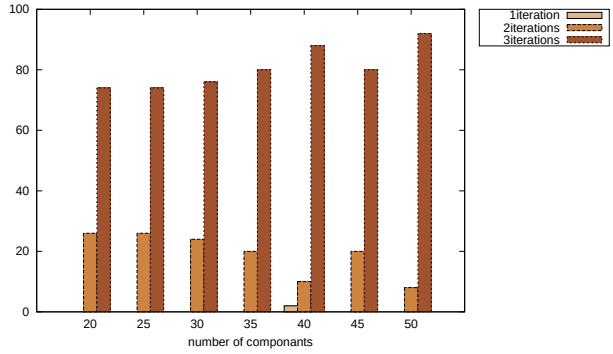

Fig. 4. Experimental results for large circles.

Experiment 2: automata on a tetrahedron. We considered problems with 4 automata (generated as described above) for which the interaction graph is a tetrahedron. An iteration is performed by updating each message exactly once. These updates being done along a path in the interaction graph which takes each edge only once in each direction. This is shown on Figure 5.

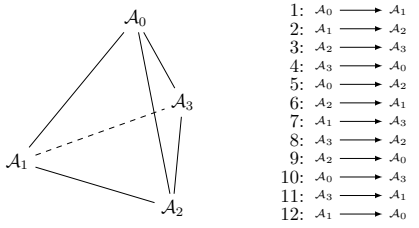

Fig. 5. Automata on a tetrahedron (left) and order of message updates during an iteration (right).

In this setting we obtained the following results for 50 "difficult" test cases (selected as described above): a quick convergence in all cases (2% within 1 iteration, 52% within 2 iterations, 42% within 3 iterations, and 4% within 4 iterations), and a solution found with no backtracking in 85% of the cases. The slightly slower convergence could

come from the way iterations are performed, which may be less efficient than in the circles of automata.

All these results lead us to believe that using the message passing algorithm directly on problems for which the interaction graph is not a tree may give good results as soon as the interactions are sufficiently sparse. In particular, searching for solutions in problems updated by Algorithm 1 using backtracking may allow us to find a solution with few backtracks: in our experiments a solution was frequently found with no backtracking at all.

4. TURBO ALGORITHMS FOR COST-OPTIMAL PLANNING

In this section we propose solutions for applying turbo algorithms to systems with quantitative aspects.The main issue to deal with is that cycles in the interaction graphs usually make cost being counted several times along an execution of Algorithm 1.

4.1 Necessity of a normalization

Consider the example in Figure 6. During the updates of messages, the cost of α will grow unbounded. This example shows that in any factored planning problem where costs of solutions are not null, it is hopeless to expect to see any stabilization of messages. This suggests the need for a normalization procedure after any synchronous product, in order to keep costs of a path within a reasonable scope.

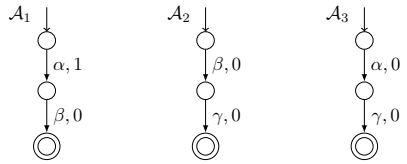

Fig. 6. An example where stabilization is not possible.

In our setting, one can imagine two ways for normalizing an automaton: either (1) adding a constant to the cost of each path, or (2) dividing the cost of each path by a constant. The main default of (2) is that it changes the difference of costs between paths. In our case where costs are additive, it may result in choosing the wrong path. For example, see Figure 7: originally the best solution consisted of firing an α in each automaton, but after dividing the costs of paths in $\mathcal{A}_1$ by $c = 3$, the best solution is a firing of β in each automaton. By contrast (1) gives small costs to local paths which already have the smallest costs. It concentrates the smallest costs on the paths that are potentially part of a cost-optimal solution. In this way, constraint solving will be helped by costs (paths with very high costs do not have to be considered) and solution extraction should also be improved (the smallest cost path in each automaton will likely be part of a solution, while the random path considered in the previous case has no particular reason of being part of a solution). For these reasons we will use (1) rather than (2).

In the next part we describe concretely how to add a cost to each path of a weighted automaton. We also suggest a possible constant to use for normalization.

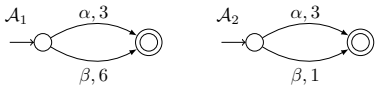

Fig. 7. Dividing costs is not acceptable.

4.2 Normalization procedure

The idea of our normalization procedure is to add a constant to the cost of each path of the automaton that is being normalized. A simple way to do this is to add the constant as an cost to each initial state of the considered automaton (a cost which will be added to any path starting from this state). However, this may lead to automata where there is a huge negative initial cost and a huge cost for each path. Thus, we suggest that once the cost has been added to the initial cost it is propagated throughout the automaton, so that costs of transitions do not grow too much. This can be done by pushing this cost towards final states, using algorithms similar to the weight pushing algorithm from Mohri (2009).

A possible normalization constant is the cost of a shortest path minus one. Using this constant ensures that, after each normalization, the cost of the shortest path(s) is one. This has the interest of guaranteeing that a normalized automaton with no negative costs on transitions always exists.

Using this normalization it is very likely that no stabilization will be reached when applying Algorithm 1 to systems with costs. The fact is that normalization (as expected) only stabilizes paths with smallest costs. To decide when the execution of Algorithm 1 should stop on a given problem, a distance, similar to the one proposed in Section 3.2, can be used.

4.3 Experimental results

As in the case of factored planning, we generated cost-optimal factored planning problems for our experiments. The automata generated have the same characteristics as before and random costs on transitions. We still consider the "difficult" problems only by eliminating the ones with no solutions or trivial solutions.

In all our experiments, the condition for stopping the algorithm is now stability of the cost-optimal solution in each component.In order to select a solution we proceed as follows. First choose a locally cost-optimal solution p_0 in $\mathcal{A}_0$, and send it to its neighbors as before. Then a locally cost-optimal solution compatible with p_0 is chosen in each neighbor, and propagated as well. This is done until a solution is found in each component, or no compatible solution can be found in a component. We still do not perform backtracking. If a solution is found, one searches for a cost-optimal solution using a centralized approach and compares the costs of the two solutions, in order to estimate the quality of that found with the turbo approach.

Experiment 1: automata on a circle. We look at circles of 3 to 7 components, trying 20 different circles in each case. We do not test as many problems as for the no-costs case because generating difficult problems requires much

more time in practice when there is costs (minimization of weighted automata may not terminate, and not minimizing increases a lot the computation time in many cases). We thus focus on the percentage of solution found, and in the cases where a solution is found, on the quality of this solution. Results for this experiment are presented in Figure 8. Among solutions found with turbo methods are represented the amount of solutions which are optimal or of a given quality (x-y% means that the cost of the solution found is between x% (exclusive) and y% greater than the cost of a cost-optimal solution).

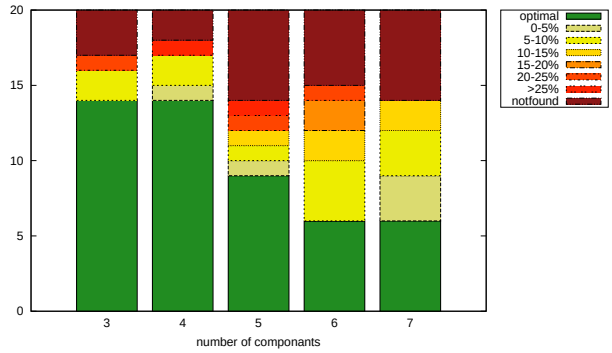

Fig. 8. Results for circles of 3 to 7 weighted automata.

Experiment 2: automata on a tetrahedron. We also generated problems where automata are one a tetrahedron, as in the case with no costs. The results obtained on 50 such problems are the following: quick convergence speed in almost all cases (except in one experiment, stabilization was always reached after 2 or 3 iterations), and a solution found without backtracking in 68% of our experiments. Among the 34 solutions found, 17 where optimal.

The experimental results obtained here by applying turbo algorithms to factored cost-optimal planning problems for which the interaction graph is not a tree led us to believe that these algorithms are of interest. In particular, we frequently obtained optimal or close-to-optimal solutions in "difficult" problems. Moreover, these solutions were found within times comparable to what we obtained in Fabre et al. (2010) for factored cost-optimal planning problems when the interaction graph was a tree.

5. CONCLUSION

In this paper we presented very encouraging experimental results on the use of turbo algorithms for factored planning and cost-optimal factored planning. In the case of constraint solving, the algorithms converged in very few iterations on many test cases. Moreover, in almost all cases, solutions have been obtained quickly on networks of up to 50 small automata, which are hardly manageable using a centralized approach. This corresponds to finding a path in an automaton with up to 10^{50} states. In the case of cost-optimal planning the algorithms converged in few iterations as well. The solutions obtained were frequently optimal, or close-to-optimal. This can be explained by the fact that the costs of solutions which are not likely to be optimal diverge along execution of the algorithm. One should, however, consider these results carefully as they were obtained using a random problem generator. It may be the case that on real problems, turbo algorithms converge slowly, or do not filter sufficiently many wrong plans to ensure a quick isolation of a solution after convergence. However, we believe that the experimental results presented in this paper show that turbo algorithms may render accessible problems that are otherwise intractable with standard centralized approaches. This should, at least, be a reason for considering these algorithms in the context of planning.

Among possible future work, we would like to understand more precisely the cases in which turbo algorithms are efficient. In particular, it would be useful, given a factored planning problem, to be able to decide whether or not turbo algorithms may reasonably be applied. Interaction graphs where the only cycles are very large are a known example. However, in the case of planning other parameters than the shape of interaction have to be taken into account. A way to measure the quantity of information in an automaton, and, in particular, the quantity of information added to an automaton after having taken its synchronous product with another automaton, would be useful.

REFERENCES

Amir, E. and Engelhardt, B. (2003). Factored planning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, 929–935.

Berrou, C., Glavieux, A., and Thitimajshima, P. (1993). Near shannon limit error-correcting coding and decoding: Turbo-codes. In *Proceedings of the IEEE International Conference on Communications*, 1064–1070.

Blum, A. and Furst, M. (1995). Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2), 281–300.

Bylander, T. (1991). Complexity results for planning. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, 274–279.

Fabre, E. (2003). Convergence of the turbo algorithm for systems defined by local constraints. Technical Report RR-4860, INRIA.

Fabre, E. and Jezequel, L. (2009). Distributed optimal planning: an approach by weighted automata calculus. In *Proceedings of the 48th IEEE Conference on Decision and Control*, 211–216.

Fabre, E., Jezequel, L., Haslum, P., and Thiébaux, S. (2010). Cost-optimal factored planning: Promises and pitfalls. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling*, 65–72.

Hart, P., Nilsson, N., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100–107.

McEliece, R., MacKay, D., and Cheng, J.F. (1998). Turbo decoding as an instance of pearl's belief propagation algorithm. *IEEE Journal on Selected Areas in Communications*, 16(2), 140–152.

Mohri, M. (2009). *Handbook of Weighted Automata*, chapter 6. Springer.

Pearl, J. (1982). Reverend bayes on inference engines: A distributed hierarchical approach. In *Proceedings of the 2nd National Conference on Artificial Intelligence*, 133–136.