Scalable and Robust Multi-agent Planning with Approximated DCOP

Antonín Komenda[1], Robert N. Lass[2], Peter Novák[3], William C. Regli[2] and Michal Pěchouček[1]

[1] Agent Technology Center, Czech Technical University, Prague, Czech Republic
[2] Department of Computer Science, Drexel University, Philadelphia, PA, USA
[3] Dept. of Software and Computer Technology, Delft University of Technology, NL
antonin.komenda@agents.fel.cvut.cz, urlass@cs.drexel.edu,
P.Novak@tudelft.nl, regli@cs.drexel.edu,
michal.pechoucek@agents.fel.cvut.cz

Abstract. Distributed multi-agent planning presents several challenges as yet not addressed by existing techniques. First, the complexity of planning scales non-linearly as a function of the number of agents. Second, most multi-agent planning approaches assume perfect and reliable communications are available to coordinate actions across the distributed agents. Third, even given reliable communications, agents may not be able to converge on a reliable overall plan for the group without resorting to computationally expensive (and communication intensive) problem centralization. This paper develops a formalization of the multi-agent planning problem and a novel three-phase approach which generates plans, solves them using the approximate, robust Max-Sum DCOP algorithm and then uses plan repair to address these three points. We provide sketches of the proofs of these claims as well as empirical results, for a restricted version of the Crane-Robot domain, of better scalability than existing approaches, and robust performance with up to 90% message loss.

1 Introduction

Recent research has produced a general framework for distributed multi-agent planning [1], one in which the planning problem is represented as a constraint satisfaction problem (CSP). Once encoded as a CSP, distributed agents then solve the planning problem using optimal distributed constraint satisfaction algorithms. This approach, while quite general, has to date employed *complete* distributed constraint *satisfaction* algorithms (e.g., Asynchronous Forward Checking (AFC)). The result is that there remain challenges of scalability and robustness. Scalability because, as the number of agents and/or size of the plans grow, the search space for CSP-type algorithms grows exponentially. Robustness is a problem because most distributed coordination algorithms and constraint solvers assume perfect and reliable communications among distributed agents—a property that simply does not hold true in the real world. Even given reliable

communications, nearly all existing techniques require computationally expensive (and communication intensive) problem centralization in order to converge on a complete solution.

This paper develops a formalization of the multi-agent planning problem that builds on classical planning representations [2] and recent work on CSP-based planning [1]. Based on this formalization, we present an efficient method to generate valid action sequences—one that prunes those containing actions that are not part of feasible plans. We then show how to map these sequences, *generative action graphs*, that capture sets of feasible plans, into the domains for solving by an *approximate* Distributed Constraint *Optimization* (DCOP) algorithm. By using a DCOP, rather than a CSP, we offer a novel approach that is *robust* to message loss (the Max-Sum algorithm [3]). Using constraint *optimization* rather than *satisfaction* also allows us to compute "best effort" plans that maximize the number of goals agents reach, rather than only looking for plans that meet all of the goals. Hence the price of the robustness and scalability afforded by the DCOP is paid in optimality—the approximation technique may return infeasible plans. Therefore, the last step in our approach is to employ *plan repairing* techniques to transform any infeasible plans into feasible ones. We conclude by providing a sketch of some theoretical properties of this approach and an empirical analysis using a restricted Crane-Robot domain.

1.1 Example Domain: Box-Movers

To motivate this problems, we introduce an restricted instance of the Crane-Robot domain problem which we call Box-Movers. In Box-Movers, there are k mover agents (representing a robot and crane as one entity), which can each move in a $w \times h$ grid and b distinguishable boxes each of which has one target position. The boxes cannot be laid upon each other. Each grid point can be accessed by only one mover with the exception of s points, which can be accessed by two neighboring agents (hand-over points). Each mover can carry only one box at a time.

The initial state defines points where the movers start and where the boxes lie, or if a box is initially carried by a mover. The movers has three possible actions (i) move, (ii) load, and (iii) unload. A mover can *move* in its accessible grid by one cell horizontally, vertically, and diagonally (including the hand-over points) whether or not it is carrying a box. A mover can *load* a box, if it is at the same point as the box. A mover can *unload* a box if it is carrying it, and there is not another box in the unload position.

Each box has a target point. A solution of the problem is a plan of actions for each mover which relocates the boxes from their initial points to their target points.

An example instance of the domain is depicted in Figure 1, where $k = 2, w = 10, h = 5, b = 2$, and $s = 1$.

The size of the domain can be expressed as the number of possible states:

$$|S| \cong (wh)^k s(b-1)(2+s)^{b-2},$$

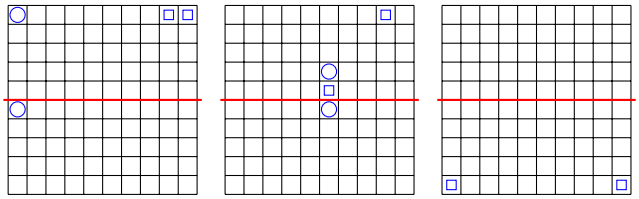

Fig. 1. The example domain, where circles represent movers and squares represent the boxes. (a) The initial configuration. (b) Box1 is being handed from Mover1 to Mover2. (c) The goal position.

For the example instance of the domain the size is ~ 20000.

The rest of this paper is organized as follows. First, we provide some background on the previous Planning as CSP framework. Next, we formally describe our solution. Then, we theoretically and empirically analyze our approach. We conclude with a summary and future work.

2 Proposed Solution

The solution is based on the Multi-agent Planner presented in [1]. The planner consists of two phases of planning. In the first phase, a multi-agent plan is formed (with help of the DCOP solver). This plan contains only public actions. A public action is an action which can affect other agents' actions. In the second phase, the plans are locally completed using the private actions, which ensure transition from one public action to the next one for each agent.

Using distributed constraint reasoning (DCR) to solve the planning problem shifts the complexity from the planner to the DCR solving algorithm, and these algorithms cannot scale to large problem sizes. The scalability of these algorithms is especially relevant in planning, where the domains grow exponentially with δ, the length of allowable action sequences.

We propose using an approximation algorithm for solving the constraint reasoning representation of the planning problem, and then using plan repair to remove infeasible parts of the plan. In our empirical results, we use the Max-Sum algorithm [**?**], which has the property that solution quality gracefully degrades with message are loss. Max-Sum also produces complete solutions for acyclic constraint graphs.

The outline of the approach can be summarized as:

- formulation and definition of the planning problem
- generation of candidate public action sequences (separately by each agent)
- selection of appropriate sequences (distributedly by the agents)
- reparation of invalid sequences (distributedly by the agents)

– private planning of local plans based on the public sequences (separately by each agent)

The following sections describe and analyze these phases.

2.1 Planning Problem Formalization

A *planning problem* is defined as a tuple $\Pi = \langle \varphi, S, s_{\text{init}}, S_{\text{goal}} \rangle$ where φ is a set of n *agents* (defined using their *actions* a, consisting of preconditions *pre*, *add* effects, and delete effects *del*):

$$\begin{aligned} \varphi &= (A_1, \ldots, A_n)\,, \\ A_i &= \{a | pre(a) \subseteq \mathcal{L}, add(a) \subseteq \mathcal{L}, del(a) \subseteq \mathcal{L}\} \cup \{\epsilon\}\,, \\ \epsilon &: pre(\epsilon) = \emptyset, add(\epsilon) = \emptyset, del(\epsilon) = \emptyset, \end{aligned}$$

where ϵ is an *empty action* and the language $\mathcal{L}$ is defined as a set of propositions. Each proposition in $\mathcal{L}$ represents a fact which can hold in a state. The set of all possible states S is defined using the language as $S = 2^{\mathcal{L}}$. The initial state and the goal state(s) are defined as $s_{\text{init}} \in S$ and $S_{\text{goal}} \subseteq \mathcal{L}$.

The definition of the actions respect classical STRIPS language as proposed in [2]. We base the agent definition and the relation between an agent and its actions on the MA-STRIPS language as defined in [1].

Private actions are those that an agent can take that will not affect other agents' variables. Public actions are those that will affect other agents' variables. Sets of public and private actions, respectively, are defined for agent A_i as:

$$\begin{aligned} A_i^{\text{pub}} &= \{a | vars(a) \cap \bigcup_{\forall a' : a' \in A, A \in \varphi \setminus A_i} vars(a') \neq \emptyset\}, \\ A_i^{\text{priv}} &= A_i \setminus A_i^{\text{pub}}, \end{aligned}$$

where $vars(a) = pre(a) \cup del(a) \cup add(a)$.

A *parameterized action* denoted as $a(p_1, \ldots, p_k)$ can be represented using the proposed definition of an action such that the set of actions is extended by the Cartesian product of all possible combinations of values from the domains of the action parameters (*i.e.*: grounded actions).

The input to the planner is a problem, Π, and the output of the planner is a multi-agent plan *MA-plan*. A MA-plan $\mathcal{P}$ is defined as a set of *personal plans* P_i:

$$\begin{aligned} &\mathcal{P} = (P_1, \ldots, P_n)\,, \text{where} \\ &P_i = a_1, \ldots, a_m, \text{ s.t. } a \in P_i \Rightarrow a \in A_i. \end{aligned}$$

All the personal plans have the same length m according to the longest one (the others are padded with ϵ). A personal plan consists of actions a for agent A.

The plan is considered *sound* if the global states, s, as the system evolves are induced by the MA-plan actions of all agents $\{P_1(k), \ldots, P_n(k)\}$ for each step $k \leq m$:

$$\exists s_0, \ldots, s_m : s_k \in S :$$
$$s_{k+1} = s_k \oplus \{P_1(k), \ldots, P_n(k)\},$$

where $\oplus$ is defined as removing the delete propositions of all actions in one step and adding the addition propositions into the next state, provided the preconditions are satisfied:

$$\begin{aligned} s' = s \oplus \{a_1, \ldots, a_k\} \equiv \\ (\forall j : 1 \leq j \leq k : pre(a_j) \subseteq s) \implies \\ s' = s \setminus \bigcup_{j=1}^{k} del(a_j) \cup \bigcup_{j=1}^{k} add(a_j). \end{aligned}$$

In each step k the actions of all agents must be *consistent*, *i.e.*: *add* and *del* sets have to be exclusive:

$$\begin{aligned} \forall k : 1 \leq k \leq m : \\ \bigcup_{j=1}^{n} add(P_j(k)) \cap \bigcup_{j=1}^{n} del(P_j(k)) = \emptyset \end{aligned}$$

The initial and goal states have to be taken into account:

$$s_0 = s_{\text{init}}, S_{\text{goal}} \subseteq s_m.$$

A *sequence of public actions* can be derived from a personal plan P_i by replacing of all private actions from A_i^{priv} by the empty action ϵ.

2.2 Public Action Sequence Generator

The plan for each agent is built upon a sequence of public actions. These sequences are generated and passed to the DCOP solver for selection of appropriate sequence tuples, which are then used for personal planning.

The complexity of the generation process is crucial, since the number of possible sequences grows exponentially with the number of variables parameterizing the actions. As the variables, we consider (i) what action a (ii) is used at what time t (a tuple (a, t) described in [1]), and (iii) with what parameters. All the values of the variables form a *variable value set* V. The number of possible combinations of values can be expressed as $\prod_{v \in V} |v_D|$, where v_D denotes the domain of variable v. Using information about a particular planning domain, we can prune the number of combinations. By pruning in this context, we mean generating

only sound sequences of public actions (*i.e.*: such which can be completed by the private actions resulting in sound multiagent plans).

Our approach uses a *generative graph representation* of the (public) planning domain such that all possible flows in the graph represent all possible public action sequences considering the planning domain constraints (*i.e.*: the pruned set of all possible sequences). In other words such a graph prescribes all possible public plans in a compact form. Obviously, this approach does not change the worst case complexity of generating all possible public plans, however it helps with a successive requests for valid public plans.

The representation uses a directed acyclic graph G, where:

$$G = (V', E, \eta),$$
$$V' = V \cup \{b, e\},$$

where the vertices V' represent the values of the action and parameter variables. The edges E represents possible sequencing of the actions and their parameters represented by their values. The function η defines the direction of the edges (if $\eta(e) = (u, v)$, then u is the tail and v is the head of the edge e). Between the beginning vertex b and ending vertex e, there are δ *layers* $L_1, ..., L_\delta$ of vertices representing timing of the actions within them (possibly with their parameter values):

$$L_i \subseteq V,$$
$$L_1 \cup \cdots \cup L_\delta = V,$$
$$\forall i, j : i \neq j : L_i \cap L_j = \emptyset.$$

The number of the layers reflects the required length of the resulting action sequence.

The edges can be separated into two groups: (i) *extra-layer edges* E^{e} and (ii) *intra-layer edges* E^{i}:

$$E^{\mathrm{e}} \subseteq E, E^{\mathrm{i}} \subset E,$$
$$E^{\mathrm{e}} \cup E^{\mathrm{i}} = E, E^{\mathrm{e}} \cap E^{\mathrm{i}} = \emptyset.$$

The extra-layer edges are oriented only from a layer of a lower index towards a layer of higher index, only between neighboring layers and represent an application of two consecutive actions:

$$\forall e : e \in E^{\mathrm{e}}, \eta(e) = (u, v), u \in L_i, v \in L_{i+1},$$
$$pre(v) \subseteq s_{\mathrm{init}} \oplus \cdots \oplus \bigcup_{\forall f : f \in E^{\mathrm{e}}, \eta(f) = (w,u)} \{w\} \oplus \{u\}.$$

Additionally, each vertex (excluding b and e) has to have at least one incoming and one outgoing edge. The intra-layer edges represent possible combinations of the parameter values if required by the related action.

An action sequence is encoded in a generative graph as a subset of vertices, which lie in one flow through the graph beginning at the node b and ending at the node e. Since the variables V represent both the action types and the parameters of the actions, a sequence $(v_a, v_{p_1}, \ldots, v_{p_{k_a}})$ represents an action $a(p_1, \ldots, p_{k_a})$, where k_a is number of the parameters of action a. A sequence of actions $(a^1(p_1^1, \ldots, p_{k_1}^1), \ldots, a^\delta(p_1^\delta, \ldots, p_{k_\delta}^\delta))$ of length δ is then a simple concatenation of the particular actions and their parameters.

To successively extract such flows from a generative graph, we firstly mark in the graph one of the outgoing edges from each vertex. We denote such edge as a *current edge* of the vertex. Note that there can be only one current edge going from each vertex. Following a path prescribed only by the current edges generates one particular flow through the graph.

To generate next flow, we have to change at least one current edge. By *changing a current edge*, another edge going out from the vertex gets marked. Provided that all possible combinations of the current edges are used consecutively, we consecutively generate all possible flows, therefore all possible public plans.

An algorithm which changes the current edges to a next marking combination and ensures generation of all possible combinations can be represented as a recursive function $next(G, v)$:

Require: $G = (V', E, \eta)$ is the generative graph,
$v \in V'$ is a vertex in that graph
Ensure: $True$ if the current edge of the vertex v was changed
Ensure: $False$ if the current edge of the vertex v was changed to its initial setting

```
1: if ∃u s.t. e = (v, u) ∈ E then
2:    u := curr(v)
3:    if next(G, u) then
4:       return True
5:    else
6:       if curr(v) = lastEdge(v) then
7:          curr*(v, firstEdge(v))
8:          return False
9:       else
10:         curr*(v, nextEdge(v, u))
11:         return True
12:      end if
13:   end if
14: else
15:   return False
16: end if
```

The function $curr(v)$ returns the current edge of a vertex v and the function $curr^*(v, e)$ changes the current edge of a vertex v to an edge e. The functions $lastEdge(v)$, $firstEdge(v)$, and $nextEdge(v, u)$ return respective outgoing edges from vertex v which are totally ordered.

To generate all the marking combinations, we can repetitively call $next(G, b)$, where b is the beginning vertex until it returns $False$ meaning that all the combinations were walked through. After each call, we extract the flow based on the marked current edges, effectively generating a next sequence of action/parameter variables.

Finally, we need a function which will transform a sequence of variables into an action sequence:

$$t : \mathcal{Q}(V) \rightarrow \mathcal{Q}(A),$$

where $\mathcal{Q}(\beta)$ denotes all possible sequences composed of elements of β. Since each action type coded by v_a has a known number k_a of parameters $v_{p_1}, \ldots, v_{p_{k_a}}$, the function t can transform a sequence of variables into a sequence of actions unambiguously. The output of the function t is later used by the DCOP part of the planner.

2.3 Mapping Target Domain to the Seq. Generator

An example of the public action sequence generative graph for the target domain is depicted in Figure 2. In the example domain, $A_i^{\text{pub}} = \{load, unload\}$, as only these actions can affect other agents by changing the position of the boxes. The private actions are $A_i^{\text{priv}} = \{move\}$, therefore they are not present in the graph. Following all the possible flows through the example graph, we can generate all the sequences of public actions of length 6, e.g. $(load, unload, \epsilon, \epsilon, \epsilon, \epsilon)$, $(load, unload, \epsilon, \epsilon, load, unload)$, and so on. However we cannot generate infeasible sequence, e.g. $(load, load, \epsilon, \epsilon, \epsilon, \epsilon)$, since there is no corresponding flow through the graph.

If the nodes of the graph are replaced by sub-graphs representing action parameters, the graph produces all possible sequences including the parameterized actions. For an instance, load nodes denoted in the figure as L can be replaced by a sub-graph with nodes L_{box1} and L_{box2}, so that the flows generate sequences such as $(load(box1), unload, load(box2), unload, \epsilon, \epsilon)$.

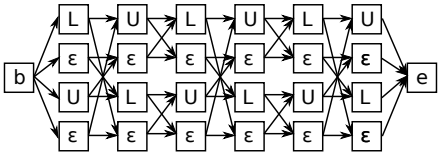

Fig. 2. Example of the generative graph for the example instance and for $\delta = 6$. Each vertex represents a sub-graph with parameters of the actions load (L) and unload (U).

2.4 Action Sequences to Plans via DCOP

The solver takes the set of action sequences generated in the previous section as input, and outputs a set of actions for each agent. As the solver is actually trying to minimize the number of unsatisfied constraints rather than finding only a satisfying assignment, we represent the problem as a Distributed Constraint Optimization problem (DCOP). We will formally define a DCOP here, describe the specific algorithm used, and then describe our mapping of the example domain (Section 1.1) to a DCOP.

A DCOP is represented as tuple $\langle A, V, \mathcal{D}, f, \alpha, \sigma \rangle$, where:

- φ is a set of agents (as in the planning definition), $\{A_1, A_2, \ldots, A_{|\varphi|}\}$;
- $\mathcal{V}$ is a set of variables, $\{\nu_1, \nu_2, \ldots, \nu_{|\mathcal{V}|}\}$;
- $\mathcal{D}$ is a set of domains, $\{D_1, D_2, \ldots, D_{|\mathcal{V}|}\}$, where each $D_i \in \mathcal{D}$ is a finite set containing the values to which its associated variable may be assigned, $\{d_{i,1}, d_{i,2}, \ldots, d_{i,|D_i|}\}$;
- f is a function
$$f : \bigcup_{S \in \wp(\mathcal{V})} \prod_{\nu_i \in S} (\{\nu_i\} \times D_i) \to \mathbb{N} \cup \{\infty\}$$
mapping every possible variable assignment to a cost. This function defines constraints between variables;
- α is a function $\alpha : \mathcal{V} \to A$ mapping variables to their associated agent. $\alpha(\nu_i) \mapsto A_j$ implies that it is agent A_j's responsibility to assign the value of variable ν_i. Note that it is not necessarily true that α is either an injection or surjection;
- λ is a function, $\lambda : \mathcal{V} \to \{\nu_i\}$, mapping a variable to its neighboring variables in the constraint graph (variables it shares a constraint with).

The goal of a DCOP algorithm is generally to find the value assignment for all variables that minimizes f[4].

The algorithm used in this work is the Max-Sum algorithm [4]. This algorithm is only complete on acyclic graphs. It provides bounded approximation on all graphs, although actual bounds are specific to a problem instance. Empirically, the algorithm performs well on many cyclic graphs. Another advantage of using Max-Sum is that it is robust to message loss, one of the few algorithms in this category.

A brief description of the algorithm follows, please see the original paper for more details. The problem is represented as a factor graph, where each agent creates one variable node for the variable it controls, and one function node that is linked to all of the constraint graph neighbors of the variable. There are two kinds of messages: Q-messages which variable nodes send to function nodes, and R-messages which function nodes send to variable nodes.

A Q-message sent from the variable node of agent i to the function node of agent j will aggregate R-messages that agent i has received so far for domain

[4] Or maximizes f, if it represents utility rather than cost.

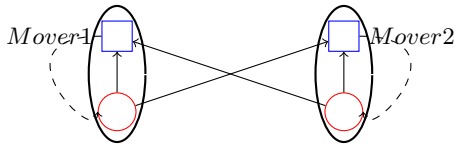

Fig. 3. Factor graph for our example domain, with boxes representing function nodes, and circles representing variable nodes. R-messages travel along the dotted lines and q-messages travel along the solid lines.

value $d_{i,k}$. Formally, it is defined as:

$$Q_{i \to j}(d_{i,k}) = \alpha_{ij} + \sum_{j' \in \lambda(i) \setminus j} R_{j' \to i}(d_{i,k})$$

where α_{ij} is a normalizing scalar, chosen such that:

$$\sum_{d_i} Q_{i \to j}(d_{i,k}) = 0$$

An R-message sent from the function node of agent j to the variable node of agent i informs agent i of the utility of the best configuration that the group of agents aggregated by the function node of agent j can achieve if variable i chooses the domain value $d_{i,k}$. This includes both the utility for the chosen configuration (given by U_j below) and the Q-message values for that configuration. Formally:

$$R_{j \to i}(d_{i,k}) = \max_{\boldsymbol{d_{j,k}} \setminus i} \left(U_j(\boldsymbol{d_{j,k}}) + \sum_{n' \in N(m) \setminus n} Q_{n' \to m}(x_{n'}) \right)$$

Each agent, a_n, locally calculates the utility for each value in its domain as:

$$Z_i(d_{i,k}) = \sum_{j \in \lambda(i)} R_{j \to i}(d_{i,k})$$

and selects the value with the highest utility.

One important optimization when calculating R-messages is that once a solution that achieves the maximum utility possible for the local problem over the function node is found, it stops searching. This allows us to reduce the search space by taking advantage of variable ordering in many cases. We plan to exploit of this in future work.

2.5 Mapping Target Domain to a DCOP

In general, any planning problem can mapped to a DCOP as described in [**?**]. In our example, the sequences generated by the public action sequence generator are converted to a DisCSP as follows:

– One agent, a_i, is generated for each mover in the problem.
– Each agent is assigned one variable, v_i, representing their plan.
– Each domain, D_i is assigned to be the set of action sequences defined as t above.
– The function, f, is used as a utility function, which the algorithm maximizes. The following are illegal actions which incur a penalty (negative value):
  • A box is picked up from someplace other than where it is.
  • A box is unloaded on top of another box.
  • A mover loads or unloads a box outside of its movement area.
  • A box is not in its goal state after all of the actions are taken.

  and a utility (positive value) is incurred when a box is placed in the goal position.
– α is a trivial function that maps each agent to its variable.
– The neighbor function, λ, maps a variable, v_i to the set of variables whose movement area is adjacent to it.

2.6 Repairing Resulting Plans

To ensure soundness of the overall multi-agent planning process, the plan repair phase solves potential infeasible plans returned by the approximate DCOP algorithm.

A plan repairing problem Σ is defined using a planning problem Π, resulting in a MA-plan $\mathcal{P}$, a failed (unanticipated) state s_{fail}, and a step k in which the failure occurred. The input of the repairing algorithm is Σ. The output is a repaired sound MA-plan.

Since an inconsistency in the plan can, in general, happen with any action, more precisely preconditions of such action would not be satisfied due to the failure, the plan repairing approach has to be general enough to handle failures by means of both private and public actions. We use a plan repairing algorithm based on the Back-on-Track principle proposed in [5] which (i) ensures soundness of the resulting plan, (ii) it is complete, therefore it can repair any inconsistency in the plan generated by approximated DCOP and (iii) it preserves as largest suffix of the original plan as possible. Note that the completeness is ensured by the fact that the repairing approach uses replanning from scratch as a last resort technique, which is sound and complete as the planner is sound and complete.

The key idea of the Back-on-Track repair is to utilize a multiagent planner, in this case, using DisCSP to ensure soundness of the repair. The planner is used to generate a repairing plan from the failed state s_{fail} to one of states which could be passed using the original plan without a failure. The failed state is the first possible inconsistency in the plan caused by the approximated DCOP.

The definition of the algorithm follows:

1: $s_{\text{curr}} := simulate(\mathcal{P}, s_{\text{init}}, k)$
2: **while** $S_{\text{goal}} \not\subseteq s_{\text{curr}}$ **do**
3: $\quad \mathcal{P}' := plan((A_1, \ldots, A_n), S, s_{\text{fail}}, s_{\text{curr}})$
4: $\quad$ **if** $\mathcal{P}' \neq \emptyset$ **then**
5: $\quad\quad$ **return** $headplan(\mathcal{P}, k-1) \cdot \mathcal{P}' \cdot tailplan(\mathcal{P}, k)$

6: **else**
7: $k := k + 1$
8: $s_{\text{curr}} := simulate(\mathcal{P}, s_{\text{init}}, k)$
9: **end if**
10: **end while**
11: **return** $Fail$

The function $simulate(\mathcal{P}, s, k)$ returns a new state induced by the k-th action of plan $\mathcal{P}$ from state s. The function $plan(\varphi, S, s_{init}, S_{goal})$ runs the planner and the functions $headplan(\mathcal{P}, k)$ and $tailplan(\mathcal{P}, k)$ cuts the beginning and rest of the plan $\mathcal{P}$ at the k-th action respectively.

The algorithm iteratively shortens the preserved part of the original plan (lines 7 and 8). If a repairing plan $\mathcal{P}'$ can be found, it returns the fixed plan reusing the prefix and suffix of the original plan (line 5). The algorithm iterates over all inconsistencies until they are all repaired. The principle of the algorithm is to add newly generated repairing plan parts $\mathcal{P}'$ into all discontinuities in the plan resulting from the approximate planning process.

The generative graph of the planner can be preserved from the planning phase for the plan repairing phase, which implies, the process reuses a data structure from the planning phase and exploits it for more efficient plan repairing similarly to [6]. Since the planning problem in the plan repairing phase is simpler, we can use optimal DisCSP to ensure soundness of the repaired plan.

2.7 Mapping Target Domain to the Plan Repairing

An example of a non-valid plan returned by the DCOP solver can require a pickup of a `box1` at a hand-over point at step 3, but the box is not located there as the mover has to simultaneously move `box1` and `box2` in steps 1 and 2 (which is not possible).

The plan repairing problem contains s_{fail} before the infeasible load action of `box1` and with initial s_{curr}, where both boxes are at the hand-over points. Invoking of the optimal planner with such an input, results in a repairing plan relocating the `box1` from its initial position to the anticipated hand-over point. After this repair the plan continues by moving of both boxes to the target points.

3 Analysis

3.1 Theoretical

There are three main differences in performance between the previous approach using optimal DisCSP algorithms and our approach. They involve scalability, robustness and optimality. We will address each of them in turn.

Max-Sum scales better than a complete algorithm. As constraint reasoning is NP-Complete, solving it with an optimal algorithm is intractable for all but very small problems. The most expensive operation in our domain is computing r-messages. Assuming the agents all have the same size domain, he asymptotic runtime of Max-Sum is $O(|n(a_i)|^{|D|})$ where a_i is the agent with the most

neighbors. If they have different domain sizes, the runtime should be asymptotically bounded by the product of the size of all of the neighboring domains of a_i,*i.e.*: $\prod_{k \in n(a_i)} |D_k|$. The size of the messages grows linearly with the size of the domains, and the number of messages grows linearly with the number of agents.

Completeness and soundness proof sketch To give an outline of the *soundness* and *completeness* of our approach, we must consider all four phases. The generation of candidate action sequences enumerates all valid action sequences, because the generative graph edges E^e are build upon a recursive chaining of all possible combinations of public actions. As the approximate DCOP does not ensure the soundness of the plans, we have to ensure it in the last step – the plan repairing. Since the plan repairing could, in the worst case, plan the entire problem with optimal DisCSP, the approach is sound and complete, as proved in [1].

3.2 Empirical

Most DCR algorithms are not robust to failures of agents, or message loss [7]. One of the advantages of using Max-Sum is that the algorithm degrades gracefully as message loss occurs. The empirical results from [3] show that even with 80% message loss, the Max-Sum performs nearly as well as with no loss. Our results were similar, with Max-Sum creating plans that achieve the same number of goals with up to 90% message loss as with 0% message loss on problems with four agents.

These advantages are being traded off for a loss of optimality. Max-Sum is always able to return *a* plan, because at any iteration the best plan can be determined simply by each agent locally choosing the plan with the maximum Z_i value. However, it does not guarantee an optimal solution in cyclic constraint graphs. Max-Sum does provide a *bounded* approximation for cyclic constraint graphs, but the bound is a function of the graph's structure. Empirically, this has been shown to be about a 1.23-approximation algorithm [3].

For experimental purposes, a prototype of the plan repairing algorithm was implemented. As a planner, the state-of-the-art technique [1] was used. The particular planning problem implemented in the prototype is the example domain from the Introduction.

3.3 Results

Once δ is big enough to reach any of the goals, a plan is produced which does so. We wish to emphasize, that this is one of the key differences between our approach and a constraint satisfaction algorithm. For example, the results of our framework in the example domain with two agents for $\delta = 1 \dots 3$ was a set of ϵ for each agent. For $\delta = 4 \dots 5$ it produces a plan that moves one of the boxes to the goal. For $\delta \geq 6$ it produces a plan that meet both goals.

When the number of agents was increased to four, which resulted in cyclic constraint graph, Max-Sum failed to construct an optimal plan. At this point, plan repair was used to construct the optimal plan. The search space (*i.e.*: the

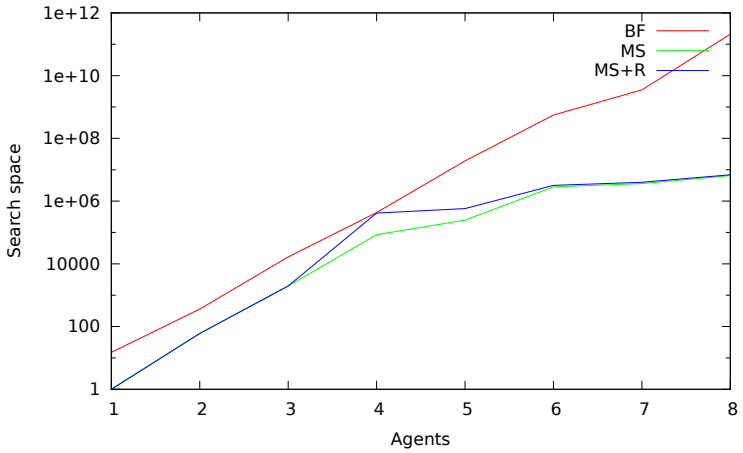

Fig. 4. Growth of the search space for $\delta = 6$ as the number of agents increases for brute-force (BF), Max-Sum (MS), and Max-Sum with plan repairing (MS+R).

number of public plan combinations need to be evaluated) is shown in Figure 4. The complexity of plan repair was the same for any number of agents, because δ was fixed at 6.

4 Background

In 2010, the first paper appeared proposing a General, Fully Distributed Multi-Agent Planning Algorithm [1]. This approach integrates planning with DisCSP to coordinate plans between agents. The paper presents heuristics that take advantage of the structure of planning problems to construct an advantageous variable ordering in the DisCSP problem and to potentially reduce the size of the variables' domains.

One of the first works on plan repairing was published in 1995 [8] and gave a theoretical analysis formally investigating the complexity of planners reusing parts of the old plans. The results of the paper states *it is not possible to achieve a provable efficiency gain of [plan] reuse over [plan] generation* and that, assuming conservative plan modification, plan reuse can be strictly more complex than plan generation from scratch. An alternative approach by [6] uses a principle called *derivational Al analogy*. Derivational analogy takes into an account not only the goal of the process, but also the history of problem solution(s). The important result of the paper is that plan adaptation (repair) by derivational analogy is more efficient then planning from scratch (in special cases, plan repairing by analogy has logarithmic complexity and planning from scratch is PSPACE-complete). This result is consistent with the conclusions previous paper, because they were restricted to conservative plan changes.

5 Conclusions and Future Work

This work proposes an extension of a state-of-the-art multi-agent planning technique with a focus on improvements in scalability, robustness and "best effort" planning. The theoretical background is built on a proposed extension of STRIPS and MA-STRIPS planning languages. The improvement in scalability is due to the replacement of an optimal DisCSP solver by an approximated DCOP solver and from a newly designed public plan generating algorithm. The potential suboptimality is resolved by a plan repairing technique. The improvement in robustness is provided by a Max-Sum DCOP solver. This approach was analyzed theoretically, and empirically on an instance of a classical multi-agent planning problem. The main areas for future work include (i) heuristic extension of the action generator, (ii) tighter integration of the DCOP solver and the action generator, (iii) usage of plan repairing technique tailored for the problem, and (iv) exploring mappings of planning problems to optimization problems (this being less straightforward than mapping to satisfaction problems).

References

1. Nissim, R., Brafman, R.I., Domshlak, C.: A general, fully distributed multi-agent planning algorithm. In: Autonomous Agents and Multiagent Systems. (2010) 1323–1330
2. Fikes, R.E., Nilsson, N.J.: Strips: a new approach to the application of theorem proving to problem solving. In: International Joint Conference on Artificial intelligence. (1971) 608–620
3. Farinelli, A., Rogers, A., Jennings, N.R.: Bounded Approximate Decentralised Coordination using the Max-Sum Algorithm. In: Distributed Constraint Reasoning 2009
4. Farinelli, A., Rogers, A., Petcu, A., Jennings, N.: Decentralised coordination of low-power embedded devices using the max-sum algorithm. In: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 2, International Foundation for Autonomous Agents and Multiagent Systems (2008) 639–646
5. Komenda, A., Novák, P., Pěchouček, M.: Domain-independent multi-agent plan repair. Journal of Network and Computer Applications (2013) DOI: 10.1016/j.jnca.2012.12.011.
6. Au, T.C., Muñoz-Avila, H., Nau, D.S.: On the complexity of plan adaptation by derivational analogy in a universal classical planning framework. Advances in Case-Based Reasoning (2002) 199–206
7. Lass, R.N., Sultanik, E.A., Greenstadt, R., Regli, W.C.: Robust Distributed Constraint Reasoning. In: Distributed Constraint Reasoning. (2009) 75
8. Nebel, B., Koehler, J.: Plan reuse versus plan generation: a theoretical and empirical analysis. Artificial Intelligence **76**(1-2) (1995) 427–454