

# A New Approach to Tractable Planning

**Patrik Haslum**

NICTA & The Australian National University  
Patrik.Haslum@nicta.com.au

## Abstract

We describe a restricted class of planning problems and polynomial time membership and plan existence decision algorithms for this class. The definition of the problem class is based on a graph representation of planning problems, similar to Petri nets, and the use of a graph grammar to characterise a subset of such graphs. Thus, testing membership in the class is a graph parsing problem. The planning algorithm also exploits this connection, making use of the parse tree. We show that the new problem class is incomparable with, *i.e.*, neither a subset nor a superset of, previously known classes of tractable planning problems.

## Introduction

The computational hardness of even the classical, propositional planning problem has caused researchers to investigate restricted classes of planning problems that can be shown to be solvable in polynomial time. To claim a class of planning problems tractable, we need to show that both *membership*, *i.e.*, deciding if an arbitrary planning problem belongs to the class, and *plan existence* for problems in the class can be solved in polynomial time. Most research into tractable planning so far has focused on problem classes defined by syntactic restrictions on the representation formalism (Bylander 1994; Bäckström & Nebel 1995) and on certain types of structural restrictions, most defined via the causal graph (a compact summary of possible interactions in a planning problem). An advantage of studying restrictions of this form is that testing them (and hence solving the membership problem) in polynomial time is easy. However, the space of such restrictions is by now fairly well mapped out, with most classes assigned to either side of the line of tractability (Katz & Domshlak 2008; Giménez & Jonsson 2008).

Here, we take a different approach to defining restrictions on the structure of planning problems, using a formal grammar to define a language of graphs, which represent planning problems. This reduces the membership problem to that of graph parsing, which, due to certain restrictions on the grammar, can be solved in polynomial time. The resulting parse tree also plays an important role in the polynomial algorithm

for solving problems in the class. Plan existence is decided by bottom-up label-propagation over the parse tree, similar in spirit to the algorithm for tree-shaped CSPs.

Although it may be less transparent than restrictions on syntax or the causal graph, the use of a graph grammar has the important advantage of allowing us to explore novel classes of restrictions, that can not be formulated in those terms. To demonstrate this potential, we design a new tractable problem class, with the explicit aim of making it distinct from previously known tractable classes. The graph representation of planning problems we use is closely related to (and, indeed, strongly inspired by) the *Petri net* formalism, so our results are easily related also to known tractable classes of Petri nets. The class we define is novel also with respect to them.

## Graph Representation of Planning Problems

We consider classical, propositional STRIPS planning (*e.g.* Bylander 1994), and sequential plans.

A planning problem ( $P$ ) consists of a set of propositions, a set of actions each described by its preconditions ( $\text{pre}(a)$ ), positive ( $\text{add}(a)$ ) and negative ( $\text{del}(a)$ ) effects, an initial state and a goal. We say that action  $a$  *consumes* proposition  $p$  iff  $p \in (\text{pre}(a) \cap \text{del}(a))$ , and that  $a$  *produces*  $p$  when  $p \in \text{add}(a)$ . Positive effects take precedence over negative effects: if  $p \in \text{del}(a)$  and  $p \in \text{add}(a)$  for some proposition  $p$ , then  $p$  is true in the state resulting from execution of  $a$  (this is in accordance with PDDL semantics; Fox & Long 2003). In the context of sequential plans, this implies that  $p \in (\text{pre}(a) - \text{del}(a))$  is equivalent to  $p \in (\text{pre}(a) \cap \text{del}(a) \cap \text{add}(a))$ , *i.e.*, action  $a$  requiring  $p$  as a precondition but not consuming  $p$  is the same as  $a$  consuming and producing  $p$ . Thus, we can – and will – without loss of generality assume that every action consumes all its preconditions, *i.e.*, that  $\text{pre}(a) \subseteq \text{del}(a)$ .

Various graphs have been used to describe some aspect of a planning problem or to provide a condensed view of it – an example is the now well-known causal graph. In contrast, the graph defined below is a *complete representation* of a planning problem. However, it is limited in that it can not represent every planning problem. As mentioned, it is inspired by the Petri net formalism (*e.g.* Murata 1989): that connection is explored in the section on expressivity below.

**Definition 1** Let  $P$  be a planning problem. The graph representation of  $P$ , denoted  $G(P)$ , is a directed and attributed graph such that:

- $G(P)$  has a square node for each action  $a$  in  $P$  (denoted by  $G(a)$ ).
- $G(P)$  has a circular node for each proposition  $p$  in  $P$  (denoted by  $G(p)$ ).
  - The circle is shaded iff  $p$  is true in the initial state of  $P$ .
  - The circle has double borders iff  $p$  is goal in  $P$ .
- $G(P)$  has a solid edge from node  $G(p)$  to node  $G(a)$  iff action  $a$  consumes proposition  $p$ .
- $G(P)$  has a solid edge from node  $G(a)$  to node  $G(p)$  iff action  $a$  produces proposition  $p$ .
- $G(P)$  has no nodes or edges other than those prescribed by the above rules.

We say that  $P$  is graph representable iff all action effects are consumptions or productions (i.e.,  $del(a) \subseteq pre(a)$ ).

## Graph Grammars and Parsing

A graph grammar is a formal rewriting system that operates on graphs. There are several types of graph grammars: we consider *node label controlled* (NLC) grammars, which are roughly analogous to context-free grammars for strings (e.g. Engelfriet & Rozenberg 1997). An NLC grammar operates on graphs whose nodes are labelled with symbols from an alphabet, divided into non-terminal and terminal symbols (we'll use these terms also for the nodes).

A production rule takes the form  $A \rightarrow (B, C)$ , where  $A$  is a non-terminal node label,  $B$  a graph (with nodes labelled by symbols of the alphabet) and  $C$  is the *embedding relation*. The rule is applicable to any node  $n$  in a graph that is labelled by  $A$ : the effect is that the node  $n$  is removed from the graph, a copy of the graph  $B$  inserted in its place, and edges between nodes that were previously neighbours of  $n$  and nodes of the newly inserted subgraph are established according to the embedding relation  $C$ . In general, the embedding relation can represent fairly powerful transformations, but in the grammar we construct it only redirects edges to one or more nodes in the replacement graph  $B$ .

A set of production rules  $R$  together with an *initial graph*  $G_0$  constitutes a graph grammar  $\mathbf{G}$ . Just like a standard CFG defines a set strings (a formal language),  $\mathbf{G}$  defines a set of graphs, which we denote  $L(\mathbf{G})$ : a terminal graph  $G$  (i.e., a graph consisting entirely of terminal nodes) belongs to  $L(\mathbf{G})$  iff  $G$  can be derived by from  $G_0$  by a finite series of rewrite steps according to the rules in  $R$ .

## The Parse Tree, and Graph Decomposition

By definition, any graph  $G \in L(\mathbf{G})$  has at least one associated *derivation tree* (more often called a *parse tree*). This is an ordered tree whose interior nodes correspond to the rules applied in the process of deriving  $G$  from the initial graph and whose leafs correspond to the nodes in  $G$ . Each interior node in the tree has one child corresponding to each node in the replacement graph (right-hand side) of the rule.

Each node  $N$  in the derivation tree corresponds to a unique subset of nodes in the final graph, viz. those corresponding to leaves beneath  $N$  in the tree. We say they

are the *nodes beneath*  $N$ , and that the subgraph of the final graph induced by the set of nodes beneath  $N$  is the *sub-graph beneath*  $N$  (denoted by  $G(N)$ ). We call the part of the graph that does not belong to  $G(N)$  the *context* of  $G(N)$ , and edges in the final graph between nodes in  $G(N)$  and its context *context edges*.

## Graph Parsing

The process of deciding if a given graph  $G$  belongs to  $L(\mathbf{G})$ , and reconstructing its derivation tree in case the answer is yes, is known as parsing. Algorithms for graph parsing can be constructed analogously to most string parsing algorithms. NLC grammars can be parsed by an adaption of the CYK algorithm (e.g. Aho & Ullman 1972), which works bottom-up, building a table of “partial matches” (pairs of parsable subgraphs and non-terminal symbols), until some partial match encompasses all the input or no more can be found. However, graph parsing is intractable in many cases where the corresponding string parsing problem is not, and this is the case also for general NLC grammars.

Brandenburg (1988) shows that under certain restrictions on the grammar  $\mathbf{G}$ , the number of partial matches (and therefore the running time of the CYK algorithm) is polynomial in the size of the input, i.e., the graph to be parsed. The restrictions are: first, that all graphs in  $L(\mathbf{G})$  are connected and have bounded degree; second, that  $\mathbf{G}$  has the finite Church-Rosser property, which holds if whenever two productions are applicable to different non-terminal nodes in a graph, applying them in either order yields the same result. The grammar that we define in the next section meets these requirements (the first are established by lemma 1; the second holds because rules preserve all edges incident on the reduced node, and redirect each such edge independently of the label of the node at the other end). Other polynomial time graph parsing algorithms place different restrictions on the grammar (e.g., Kaul 1982).

## The Problem Class GG1

We define a class of planning problems, called GG1, as those problems whose graph representation belongs to the set of graphs generated by a specific graph grammar. Figure 1 shows the rules of the grammar. The initial graph consists of a single node labelled by the non-terminal symbol A0.

Figure 2 shows the graph representation of a simple example problem (involving three “bits” and actions that “flip” them), which will be used as an illustration. This problem belongs to GG1. Figure 3 shows one possible parse tree. For ease of reference, nodes in the parse tree are identified by their path from the root: for example,  $N121$  is the first child of the second child of the first child of the root. The numbering of sibling nodes corresponds to the numbering of nodes in the replacement graph of their parent's rule. Figure 4 shows the intermediate graphs after each of the first three derivation steps.

## Properties of Problems in GG1

Nodes labelled by any of the non-terminal symbols A0, A1, A1I, A1S1, A1S2 or A2, and terminal action nodes, collec-

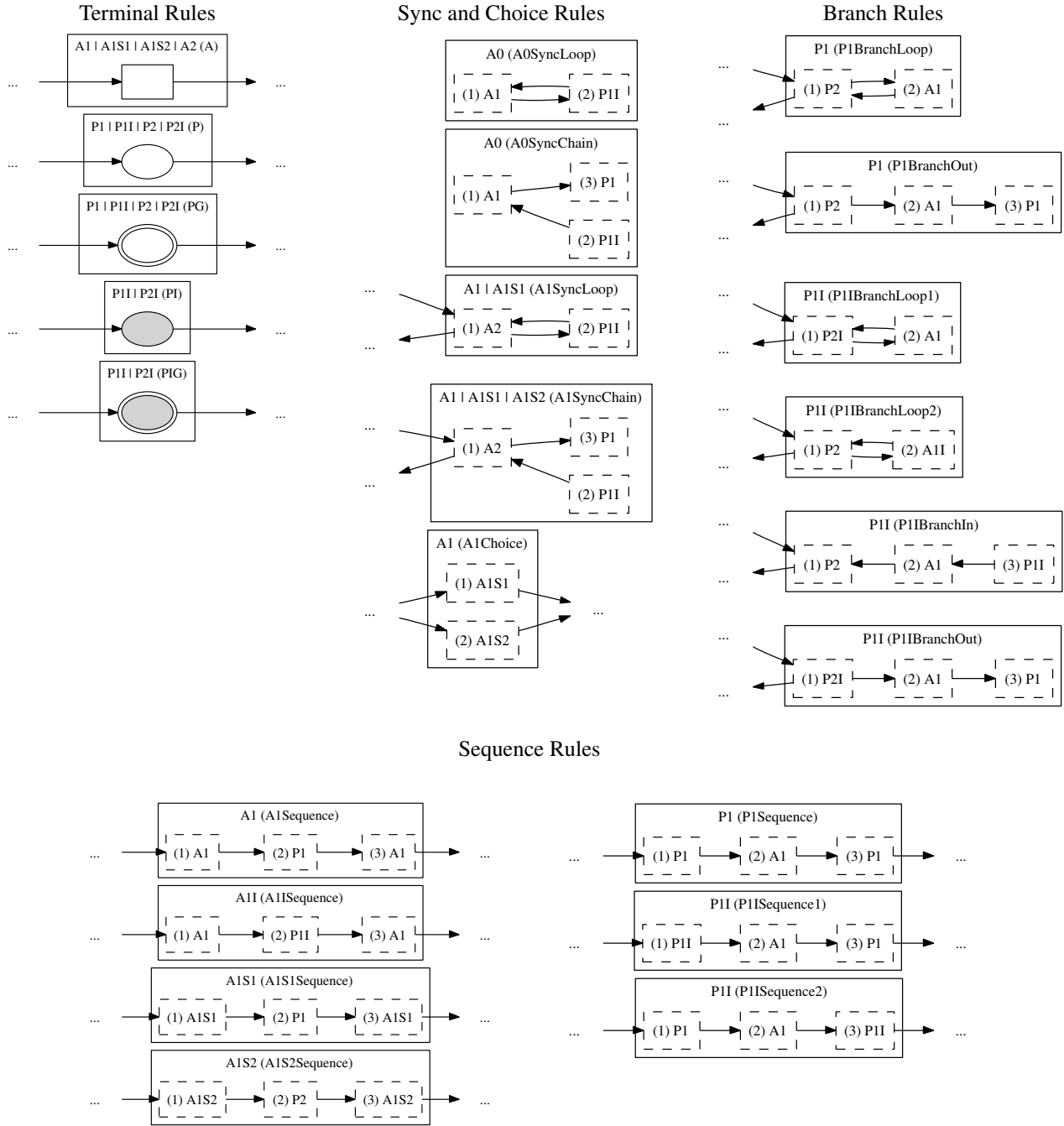


Figure 1: Rules of Graph Grammar GG1.

Each box depicts a rule  $A \rightarrow (B, C)$ , in the following way: the non-terminal symbol reduced by the rule,  $A$ , is written at the top of the box (followed by the name of the rule, in parenthesis), and the box contains the replacement graph,  $B$ . Non-terminal nodes in  $B$  are drawn with a dashed outline to distinguish them from terminal action nodes (solid boxes) and labelled by the non-terminal symbol. Child nodes in the replacement graph are numbered, except where there is only one. Terminal nodes are “labelled” by visual attributes (shape, shading, *etc.*), following the graph representation of planning problems (definition 1). The embedding,  $C$ , is indicated by a set of edges crossing the border of the box: different cases are distinguished by the direction of the edge. Where identical rules apply to more than one non-terminal, these are abbreviated into one and the non-terminals listed at the top of the box. The initial graph consists of a single node labelled  $A0$ , with no edges.

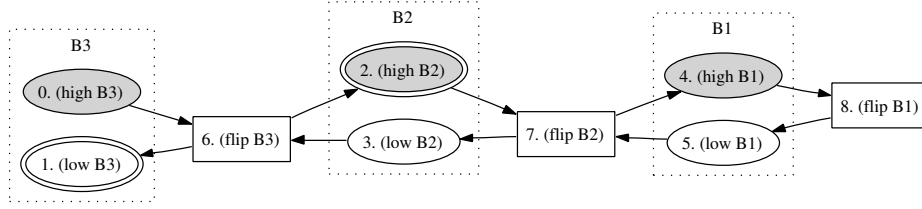


Figure 2: Graph representation  $G(P)$  of a problem  $P$  in GG1. Nodes are numbered and labelled by the name of the proposition/action, but this is only for ease of reference. The three sets of propositions indicated by dashed outlines and labelled B1–B3 are exclusive invariants (exactly one in each set is true in any reachable state), and may be seen as multi-valued state variables.

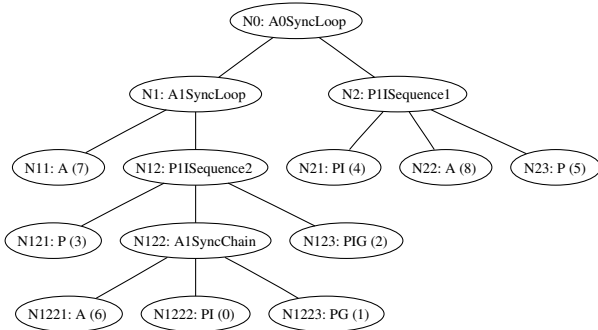


Figure 3: Parse tree for the graph in figure 2. Nodes are labelled with the name of the corresponding rule. For nodes labelled with terminal rules (*i.e.*, leaves), the number of the corresponding node in the graph in figure 2 is also indicated.

tively called “A-nodes”, represent (possibly abstract or composite) actions. Although a non-terminal A-node may be expanded into a subgraph containing both action and proposition nodes, the “interface nodes” of this subgraph (*i.e.*, nodes connected to the subgraphs context) will all be action nodes. As an example, the subgraph beneath  $N122$ , comprising nodes 0, 1 and 6 of the graph in figure 2, is derived from a node labelled A1: its context edges all connect to node 6, which is an action node. Nodes labelled with P1, P1I, P2 or P2I, and terminal proposition nodes, called “P-nodes”, similarly represent (abstract) propositions. This is formalised in the following lemma.

**Lemma 1** *Let  $G$  be any graph derivable in GG1:*

(ii)  $G$  is connected and bipartite, with A-nodes and P-nodes forming the two partitions.

(iii) The minimal and maximal in- and outdegree of each node  $G$  is determined by the node’s label, as follows: A0: 0/0; A1, AIS1, AIS2: 1/1; A2: 2/2; P1: 1–2/0–2; P1I: 0–2/1–2; and P2, P2I: 1–4/1–4. The maximal in- and outdegree of any terminal node is 4.

(iv) For any node  $N$  in the parse tree of  $G$ ,

(a) all incoming context edges of  $G(N)$  either originate in the same node (in the context of  $G(N)$ ) or terminate in the same node (in  $G(N)$ ); and

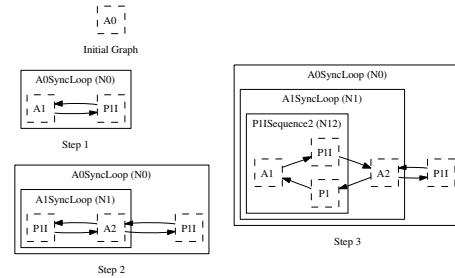


Figure 4: A few derivation steps for the graph in figure 2.

(b) all outgoing context edges of  $G(N)$  either originate in the same node or terminate in the same node. For short, we say  $G(N)$  has a simple context.

Although we use a propositional (STRIPS) representation, the set of propositions in any problem belonging to GG1 is covered by (not necessarily disjoint) subsets that are mutually exclusive, *i.e.*, such that in any reachable state at most one proposition in each such set is true. These sets may be viewed as multi-valued (or finite-domain) state variables, of the kind used in the SAS+ formalism. The problem depicted in figure 2 has three such state variables, named B1–B3 in the figure. Each action affects one or more state variables, by consuming and producing one proposition from each corresponding set (possibly the same proposition, in which case the action depends on, but does not change, the variable). This is key to proving the following lemma.

**Lemma 2** *Let  $P$  be a planning problem such that  $G(P)$ , the graph representation of  $P$ , is in  $L(GG1)$ : for every state  $s$  reachable from the initial state and action  $a$  executable in  $s$ , every proposition in  $\text{del}(a)$  is true in  $s$  and every proposition in  $\text{add}(a) - \text{del}(a)$  is false in  $s$ . For short, we say  $P$  is 1-safe.*

The synchronisation rules create a new multi-valued state variable, and make the (abstract) action corresponding to the reduced (A0 or A1) node have a precondition and/or effect on it. An example of this can be seen in step 2 in figure 4. The branch and sequences rules expand the domain transi-

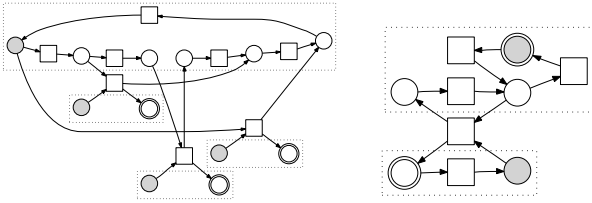


Figure 5: More examples of problems in GG1.

tion graph of a state variable, adding new values to it (as in step 3 in figure 4).

### Expressivity and Relation to Other Tractable Classes of Planning Problems

The problem class GG1 is designed with the aim of making it distinct from previous tractable planning problem classes, rather than to capture any specific application or benchmark problem. So, what kind of problems do we find in it?

The problem in figure 2 can be scaled to any number  $n$  of “bits”, and has several interesting properties. First, that all the  $2^n$  states are reachable, which shows that the tractability of GG1 is not a trivial consequence of bounding the state space. Second, the action (`flip B1`) can be executed at most  $n$  times. A particular goal (`low ?B`), for each  $?B$  requires a plan of minimum length  $O(n^2)$ , in which this action must execute  $n$  times. We do not have an example requiring an exponential-length plan, but neither do we know any polynomial bound on plan length for the class. This is an open question.

Combining this example with the left graph in figure 5, scaling both appropriately, we obtain a problem with an  $m$ th order mutual exclusion, *i.e.*, a set of  $m$  propositions such that not all but any subset of  $m - 1$  can be made true.

Problems in GG1 can contain actions that change more than one multi-valued state variable. This distinguishes it from many known tractable classes of planning problems, which require (explicitly, or implicitly by requiring an acyclic causal graph) that actions are unary, *i.e.* change only one variable (Bäckström & Nebel 1995; Brafman & Domshlak 2003; Giménez & Jonsson 2008; Katz & Domshlak 2008). An exception is the SAS+-IAO class, which permits non-unary actions under certain restrictions (Jonsson & Bäckström 1998). However, the problem on the right in figure 5 falls outside this class as it is not *interference-safe* (the single non-unary action is not irreplaceable).

Factored planning algorithms work by decomposing planning problems into “loosely coupled parts”, solving these parts as independently as possible. (The algorithm described in the next section also exploits problem decomposition, but does so *recursively*, whereas the factoring in factored planning is “flat”.) Brafman & Domshlak (2006) show this can be done in polynomial time when the induced undirected causal graph over the parts (“interaction graph”) is a tree, and the required plan length in each part (“local depth”) is bounded by a constant. Amir & Engelhardt (2003) show a similar result by bounding the number of inter-part ex-

changes. While the causal graph of problems in GG1 may contain cycles, the induced undirected causal graph is indeed a tree (assuming cycles of length 2 are collapsed into a single undirected edge), but the required local plan length, and number of interactions, can be unbounded. Again, the example in figure 2 (appropriately scaled) serves as an example. This holds also if any finite number of variables are combined into “parts”. The example can also be modified to have an exponential number of alternative local plans for each variable.

Bylander (1994) shows that plan existence is tractable under some syntactic restrictions on actions in the STRIPS formalism. These include: a single postcondition; no precondition; and a single precondition if the problem has a bounded number of goals. It is easy to see that GG1 permits the construction of problems disobeying these restrictions.

### Relation to Petri Nets

Petri nets (*e.g.* Murata 1989) is a graphical formalism for discrete dynamical systems. It is similar to (and the source of inspiration for) our graph representation, but does not represent initial and goal states graphically. Sets of derivation rules have been used to define special classes of Petri nets (*e.g.* Esparza 1990), though not for the design of polynomial time algorithms. The Petri net formalism is more expressive than propositional STRIPS planning, allowing state variables (known as “places”) that are general counters rather than binary propositions. The special case of *1-safe* nets, in which no place can reach a value greater than 1, is equivalent to propositional STRIPS planning. Deciding if there is a reachable state in which a given set of places are all 1 (known as the “coverability” problem) is PSPACE-complete, like planning. The problem is known to be tractable for a few restricted classes of 1-safe nets: *conflict-free* nets and *live free-choice* nets (Esparza 1998). However, the right graph in figure 5, viewed as a 1-safe Petri net, is neither conflict-free nor free-choice (though it is live).

### A Polynomial Algorithm for GG1

The polynomial time planning algorithm for problems in GG1 relies on the recursive problem decomposition induced by the parse tree. As mentioned, deciding if a graph belongs to  $L(\text{GG1})$ , and constructing the parse tree in case it does, can be done in polynomial time. The planning algorithm works in two phases: The first traverses the parse tree bottom-up, labelling nodes with information about possible plan fragments for the part of the problem represented by the subgraph beneath the node. This phase decides plan existence. The plan generation phase traverses the tree top-down, using the label information to avoid any need for backtracking. This phase is polynomial in the plan length.

Before describing the label-propagating phase of the algorithm, we need to introduce some terminology.

**Definition 2** Let  $P$  be a planning problem whose graph representation  $G(P)$  is in  $L(\text{GG1})$ , and let  $N$  be a node in the parse tree for  $G(P)$  and  $G(N)$  the subgraph beneath  $N$ .

- We say (the execution of) action  $a$  in  $P$  is a move into  $G(N)$  iff either

- $G(a)$  belongs to  $G(N)$  and  $a$  consumes some proposition  $p$  such that  $G(p)$  is not in  $G(N)$ ; or
- $G(a)$  does not belong to  $G(N)$  and  $a$  produces some proposition  $p$  such that  $G(p)$  is in  $G(N)$ .
- We say (the execution of) action  $a$  in  $P$  is a move out of  $G(N)$  iff either
  - $G(a)$  belongs to  $G(N)$  and  $a$  produces some proposition  $p$  such that  $G(p)$  is not in  $G(N)$ ; or
  - $G(a)$  does not belong to  $G(N)$  and  $a$  consumes some proposition  $p$  such that  $G(p)$  is in  $G(N)$ .

An action may be both a move into and a move out of a subgraph  $G(N)$ : in that case, we say that the move into  $G(N)$  takes place before the move out of  $G(N)$ . By a move through  $G(N)$  we mean a move into  $G(N)$  followed by a move out (not necessarily by a single action).

This terminology too is inspired by usage in Petri nets, where values are referred to as “tokens”, and execution of actions (“transitions”) is said to move tokens along edges of the graph.

**Lemma 3** *Let  $P$  be a planning problem whose graph representation  $G(P)$  is in  $L(GG1)$ , and let  $N$  be a node in the parse tree for  $G(P)$  and  $G(N)$  the subgraph beneath  $N$ . In any valid plan for  $P$ , moves into and out of  $G(N)$  strictly alternate.*

The proof of lemma 3 relies on 1-safeness (lemma 2) and simple context property (lemma 1(iv)). By lemma 3, the interaction between the subgraph  $G(N)$  beneath a parse tree node  $N$  and its context is a sequence of alternating moves into and out of  $G(N)$ : due to the simple context property, this sequence can be summarised by:

$N.out$ : The number of moves out of  $G(N)$  before any move in (0 or 1). In other words, if the sequence of alternating moves begins with a move out or not.

$N.min$ : The minimum number of moves through  $G(N)$  required to achieve any goals within (the last move through is not necessarily completed, *i.e.*, it may be a move in without a subsequent move out).

$N.max$ : The maximum number of moves through  $G(N)$  that is possible.

$N.in$ : The number moves into  $G(N)$  after the last move out (or through) required to achieve goals within (0 or 1). In other words, if the sequence must end with a move in.

We call these quadruples *cases*, and write them as  $(N.out, N.min, N.max, N.in)$ . The labels assigned to each node in the parse tree by the first phase of the algorithm are sets of them, called the *case set* of the node. The meaning of a case belonging to the case set of parse tree node  $N$  is that there exists a subplan, having precisely the context interactions described by the case, that achieves all goals within the subproblem  $G(N)$ . Thus, an empty set means the subproblem has no solution. There is a dominance relation between cases: if  $(N.out_1, N.min_1, N.max_1, N.in_1)$  and  $(N.out_2, N.min_2, N.max_2, N.in_2)$  are two cases such that  $N.out_1 \geq N.out_2$ ,  $N.min_1 \leq N.min_2$ ,  $N.max_1 \geq N.max_2$  and  $N.in_1 \leq N.in_2$ , then the first case dominates

the second in the sense that it places fewer requirements on the context.

Table 1 contains the rules for propagating case sets up the parse tree. For leaf nodes, case sets are given directly. For interior nodes (non-terminal rules), rules specify a set of conditional cases: the intended meaning is that the case set of the node contains those cases for which some combination of cases for the child nodes can be found that satisfies the condition. Note that conditions are not necessarily mutually exclusive. The final case set of a node is obtained by removing dominated cases.

**Lemma 4** *Let  $P$  be a planning problem whose graph representation  $G(P)$  is in  $L(GG1)$ : the case set assigned to each node  $N$  of the parse tree for  $G(P)$  by the rules in table 1 is correct and complete.*

In the example parse tree in figure 3, for the deepest leaf nodes on the left we have  $N1221 : \{(0, 0, \infty, 0)\}$ ,  $N1222 : \{(1, 0, \infty, 0)\}$  and  $N1223 : \{(0, 0, \infty, 1)\}$ . For their parent node, labelled  $A1SyncChain$ , there is only one rule, and only one combination of child node cases to examine. As the conditions of the rule are met (*e.g.*,  $N1222.min + N1222.in = 0$ ,  $N1222.out = 1 \geq 1 = \max(N1221.min + N1221.in, N1223.min + N1223.in)$ , etc), the node gets the case set  $N122 : \{(0, 1, 1, 0)\}$ . For its sibling leaf nodes we have  $N121 : \{(0, 0, \infty, 0)\}$  and  $N123 : \{(1, 0, \infty, 1), (0, 0, 0, 0)\}$ . Node  $N12$ , labelled  $P1ISequence2$ , has several rules and also two combinations of child node cases to examine (since  $N123$  has two cases in its set). For the first case of  $N123$ , the first rule gives the case  $(1, 1, 1, 1)$ , and the second rule gives  $(1, 0, 1, 1)$ ; both rule’s conditions are met. For the second case of  $N123$ , the first rule does not apply (its condition  $N12.min \leq N12.max$  is not met) but the second rule gives the case  $(0, 0, 0, 1)$ . The third rule can not be applied, since its condition  $N121.out = 0 \geq 1 = N122.min + N122.in$  does not hold. However,  $(1, 0, 1, 1)$  dominates both  $(1, 1, 1, 1)$  and  $(0, 0, 0, 1)$ , so the final case set is  $N12 : \{(1, 0, 1, 1)\}$ .

The second phase of the algorithm, plan extraction, proceeds top-down, picking a case for the root node and, recursively, the combination of cases for each child node that produced it. The plan is built from fragments that perform the moves required by the selected cases. The rules for constructing and composing these fragments follow straightforwardly from the proof of lemma 4.

In the example, the selected cases for the children of the root node are  $N1 : (0, 1, 1, 0)$  and  $N2 : (1, 0, \infty, 0)$  (there is only this choice). Thus, the plan begins with a move out of  $G(N2)$ , that is also a move through  $G(N1)$ , and the rest is internal to  $G(N1)$ . From the selected cases for nodes  $N121$ ,  $N122$  and  $N123$  and the rule labelling  $N12$ , we get that the first step in performing a move out of  $G(N2)$  is action (`flip B1`). The move is then done by the action that interfaces  $G(N1)$  to  $G(N2)$ , *i.e.*, (`flip B2`). These are the first two actions in the plan.

**Theorem 5** *Let  $P$  be a planning problem whose graph representation  $G(P)$  is in  $L(GG1)$ : (i) Plan existence for  $P$  can*

A (terminal action node): $\{(0, 0, \infty, 0)\}$
P (terminal proposition node): $\{(0, 0, \infty, 0)\}$
PI (terminal proposition node, initially true): $\{(1, 0, \infty, 0)\}$
PG (terminal proposition node, a goal): $\{(0, 0, \infty, 1)\}$
PIG (terminal proposition node, initially true and a goal): $\{(1, 0, \infty, 1), (0, 0, 0, 0)\}$
A0SyncLoop: $(0, 0, 0, 0)$ if $N1.min + N1.in = 0, N2.min + N2.in = 0$ . $(0, 0, 0, 0)$ if $N2.out > 0, N2.out + N2.max \geq N1.min + N1.in, N1.max \geq N2.min + N2.in, N1.in + N2.in \leq 1$ .
A0SyncChain: $(0, 0, 0, 0)$ if $N2.min + N2.in = 0, N2.out \geq \max(N1.min + N1.in, N3.min + N3.in),$ $N1.max \geq N3.min + N3.in, N1.in + N3.in \leq 1$ .
A1SyncLoop: $(0, 0, 0, 0)$ if $N1.min + N1.in = N2.min + N2.in = 0$ . $(0, \max(N1.min, N2.min + N2.in), \min(N1.max, N2.out + N2.max), N1.in)$ if $N2.out > 0, N2.out + N2.max \geq N1.min + N1.in, N1.max \geq N2.min + N2.in, N1.in + N2.in \leq 1$ .
A1SyncChain: $(0, \max(N1.min, N3.min + N3.in), \min(N1.max, N2.out), N1.in)$ if $N2.min + N2.in = 0, N2.out \geq \max(N1.min + N1.in, N3.min + N3.in),$ $N1.max \geq N3.min + N3.in, N1.in + N3.in \leq 1$ .
A1Choice: $(0, N1.min + N2.min, N1.max + N2.max, N1.in + N2.in)$ if $N1.in + N2.in \leq 1$ .
P1BranchLoop, P1IBranchLoop1, P1IBranchLoop2: $(N1.out + N2.out, 0, N1.max, N1.in)$ if $N2.min + N2.in = 0$ . $(N1.out + N2.out, 1 - (N1.out + N2.out), \infty, 0)$ if $N2.min > 0, N1.in = N2.in = 0$ . $(N1.out + N2.out, 0, \infty, 1)$ if $N1.in + N2.in = 1$ . $(0, 0, 0, 0)$ if $N1.out + N2.out = 1, N1.in + N2.in \leq 1$ .
P1BranchOut, P1IBranchOut: $(N1.out, 0, N1.max, N1.in)$ if $N2.min = N3.min + N3.in = 0$ . $(0, 0, \infty, 0)$ if $N1.out \geq \max(N2.min, N3.min + N3.in) > 0, N2.max \geq N3.min + N3.in, N1.in = 0$ . $(N1.out, 0, \infty, 1)$ if $N1.in + \max(N2.min, N3.min + N3.in) = 1, N2.max \geq N3.min + N3.in$ .
P1IBranchIn: $(0, 0, \infty, N1.in)$ if $N2.min + N2.in = 0, N3.min + N3.in = 0$ . $(0, 0, \infty, 0)$ if $N3.out \geq N2.min + N2.in, \min(N3.out, N2.max) \geq N1.in, N3.min + N3.in = 0$ . $(1, 0, \infty, N1.in)$ if $N3.out \geq N2.min, \min(N3.out, N2.max) = 1, N3.min + N3.in = 0$ .
All Sequence Rules: $(N.out, N.min, N.max, N.in)$ , where $N.out = \min(N1.out, N2.max, N3.max) + \min(N2.out, N3.max) + N3.out,$ $N.min = \max(N1.min, N2.min - N1.out, N3.min - (N1.out + N2.out)),$ $N.max = \min(N1.max, N2.max, N3.max),$ $N.in = N1.in + N2.in + N3.in$ if $N.min \leq N.max, N1.out + N1.max \geq \max(N2.min + N2.in, N3.min + N3.in - N2.out),$ $N2.out + N2.max \geq N3.min + N3.in, N1.in + N2.in + N3.in \leq 1$ . $(N.out, N.min, N.max, N.in)$ , where $N.out = \min(N1.out, N2.max, N3.max) + \min(N2.out, N3.max) + N3.out,$ $N.min = \max(N1.min, N2.min - N1.out, N3.min - (N1.out + N2.out)) - 1,$ $N.max = \min(N1.max, N2.max, N3.max),$ $N.in = 1$ if $N.min \leq N.max, N1.out + N1.max \geq \max(N2.min + N2.in, N3.min + N3.in - N2.out),$ $N2.out + N2.max \geq N3.min + N3.in, N1.in + N2.in + N3.in \leq 1$ . $(0, 0, 0, 0)$ if $N1.out \geq N2.min + N2.in, \min(N1.out, N2.max) + N2.out \geq N3.min + N3.in, N1.min + N1.in = 0$ .

Table 1: Rules for assigning case sets to parse tree nodes.

be decided in polynomial time. (ii) If a plan for  $P$  exists, the first action in the plan can be generated in polynomial time, and the entire plan in time polynomial in the plan length. (iii) If  $G(P)$  is derivable not using the A1Choice rule, the generated plan is optimal w.r.t. the sum of (constant, independent and non-negative) action costs.

The reason for the qualified statement of theorem 5(ii) is that we do not know if plans for problems in GG1 have polynomially bounded length (or another compact representation). The keys to establishing tractability are, first, 1-safeness and the simple context property (lemma 1(iv) and 2), which permit the compact summary of interactions between a subgraph and its context in a plan (a case), and second, bounding the size of case sets of interior nodes in the parse tree to at most linear in those of the child nodes (as the depth of the tree is at most linear in the size of the graph). The second requirement is tricky for the A1Choice rule, and is the reason why one of its child nodes is labelled with the A1S2 non-terminal, which has more restricted options for expansion.

So, is GG1 the maximal grammar with these properties? Clearly not. For example, in GG1, each action changes at most two multi-valued state variables, but we could easily increase this to three, by adding synchronisation rules for the A2 non-terminal (and a corresponding A3 symbol). In fact, for any fixed and finite number  $k$ , we could design a grammar permitting problems with actions changing  $k$  variables. The set of branching rules could be similarly extended. However, one condition for the graph parsing algorithm to run in polynomial time is that the maximum degree of every graph in the language is bounded, so we can not capture, e.g., problems with domain transition graphs that are cliques (as is common in simple planning benchmarks). There are other graph parsing algorithms with different restrictions (e.g. Kaul 1982).

## Conclusion

Tractability of the problem class GG1, although novel, is potentially not the most interesting contribution of this paper. The *method* of defining restricted classes of planning problems via a graph grammar and using the structure imposed by the parse tree to solve such problems, is very flexible: it may allow the design of tractable problem classes “tailored” to include particular problems of interest. The GG1 class was designed in this manner, with the specific aim of making it distinct from previously known classes of tractable planning problems. Next, we plan to target other criteria, such as the inclusion of certain tractable benchmark domains.

Finally, we note that for inputs not in  $L(\text{GG1})$ , the graph parsing algorithm identifies some maximal parsable fragments, which may correspond to abstractions of the input problem. This in combination with theorem 5(iii) points to the possibility of constructing admissible structural pattern heuristics (Katz & Domshlak 2008).

**Acknowledgement** NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the

Australian Research Council through the ICT Centre of Excellence program.

## References

- Aho, A., and Ullman, J. 1972. *The Theory of Parsing, Translation and Compiling*. Prentice-Hall. ISBN: 0-13-914556-7.
- Amir, E., and Engelhardt, B. 2003. Factored planning. In *Proc. 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, 929–935.
- Bäckström, C., and Nebel, B. 1995. Complexity results for SAS+ planning. *Computational Intelligence* 11:625–656.
- Brafman, R., and Domshlak, C. 2003. Structure and complexity of planning with unary operators. *Journal of AI Research* 18:315–349.
- Brafman, R., and Domshlak, C. 2006. Factored planning: How, when and when not. In *Proc. 21st National Conference on Artificial Intelligence (AAAI'06)*.
- Brandenburg, F. 1988. On polynomial time graph grammars. In *Proc. 5th Annual Symposium on Theoretical Aspects of Computer Science (STACS'88)*, volume 294 of *LNCS*, 227–236. Springer.
- Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 69(1–2):165–204.
- Engelfriet, J., and Rozenberg, G. 1997. Node replacement graph grammars. In Rozenberg, G., ed., *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific. chapter 1, 1–94.
- Esparza, J. 1990. Synthesis rules for Petri nets, and how they lead to new results. In *Proc. CONCUR'90, Theories of Concurrency: Unification and Extension*, volume 458 of *LNCS*, 182–198. Springer.
- Esparza, J. 1998. Decidability and complexity of petri net problems – an introduction. In Rozenberg, G., and Reisig, W., eds., *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*. Springer. 374–428.
- Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of AI Research* 20:61–124.
- Giménez, O., and Jonsson, A. 2008. The complexity of planning problems with simple causal graphs. *Journal of AI Research* 31:319–351.
- Jonsson, P., and Bäckström, C. 1998. State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence* 100(1–2):125–176.
- Katz, M., and Domshlak, C. 2008. New islands of tractability of cost-optimal planning. *Journal of AI Research* 32.
- Kaul, M. 1982. Parsing of graphs in linear time. In *Proc. 2nd International Workshop on Graph Grammars and their Application in Computer Science*, volume 153 of *LNCS*, 206–218. Springer.
- Murata, T. 1989. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* 77(4):541–580.