Bounded-Resource Reasoning as (Strong or Classical) Planning

A. Albore[1], N. Alechina[2], P. Bertoli[3], C. Ghidini[3], and B. Logan[2]

[1] Universitat Pompeu Fabra, pg.Circumval·lació 8 08003, Barcelona, Spain
[2] School of Computer Science, University of Nottingham, Nottingham, NG81BB, UK
[3] FBK-irst, via Sommarive 18, Povo, 38100, Trento, Italy
alexandre.albore@upf.edu,
{nza,bsl}@cs.nott.ac.uk, {bertoli,ghidini}@fbk.eu

Abstract. To appropriately configure agents so as to avoid resource exhaustion, it is necessary to determine the minimum resource (time & memory) requirements necessary to solve reasoning problems. In this paper we show how the problem of reasoning under bounded resources can be recast as a planning problem. Focusing on propositional reasoning, we propose different recasting styles, which are equally interesting, since they require solving different classes of planning problems, and allow representing different reasoner architectures. We implement our approach by automatically encoding problems for the MBP planner. Our experimental results demonstrate that even simple problems can give rise to non-trivial (and often counter intuitive) time and memory saving strategies.

1 Introduction

On devices with limited computational power, the reasoning process of an agent may fail because of lack of memory, or simply take too long to complete. Therefore, to appropriately configure agents and devices which rely on automated reasoning techniques so as to avoid resource exhaustion, we must address the problem of how much time and memory a reasoner embedded in an agent needs to achieve its goals. In this paper we show how the problem of determining the minimal time and memory bounds for propositional reasoning problems can be solved by applying planning techniques.

The key idea of our approach is to model a reasoning agent as a planning domain where fluents correspond to the set of formulas held in the agent's memory, and actions correspond to applications of the agent's inference rules. Naturally, the computational resources (time and memory) required to construct a proof depend on a number of factors. Firstly, they depend on the inference rules available to the agent. In this paper we focus on inference in propositional logic; the agent can reason by cases, by exploring different branches of a logical proof. The second key factor in determining resource requirements is the architecture of the agent. We consider a sequential (Von Neumann-style) architecture, since most existing reasoners perform sequential forms of reasoning, and/or implement different forms of reasoning on sequential hardware architectures. We show how this behaviour can be encoded as a planning problem in two different ways, which are both interesting in that they require different planning capabilities, and provide different degrees of adaptability to representing other architectures.

In the first approach, which we call 'architecture-oriented' since it mimics the sequential computation of a proof on a Von Neumann-style architecture, we explicitly model stack manipulation. This kind of modelling gives rise to an encoding of proof search as a classical planning problem. In the second approach, which we call 'proof-oriented' since it mimics the construction of logical proofs following their tree shape, the model explores different branches of the proof in parallel; still, the model represents faithfully the memory usage by the underlying Von Neumann reasoner, including stack usage. This model gives rise to an encoding of reasoning as a strong planning problem.

We implement both models for the MBP planner[3]. Our experimental results demonstrate that even for simple problems, determining bounds is not trivial, since time and memory saving strategies may not be intuitive. Also, they allow us to experimentally compare the different features of the models in terms of planning performance.

The paper is organized as follows. First, we discuss our reference reasoning agent (Section 2). Then, we discuss the analogy between an automated reasoning problem and a planning problem (Sections 3) and different planning encodings (Sections 4 and 5). This leads to a discussion of the implementation of such encodings on the MBP planner (Section 7). Finally, we show that our approach is general enough to be easily adaptable to different models for computational resources (Section 8) and we recap with a discussion on related and future work (Section 9).

2 Reasoning with Bounded Resources

The Reasoner Model. We study proof search for an agent which contains a large set of classical propositional inference rules, which is similar in spirit to natural deduction and analytic tableaux. We have focused on propositional rather than first order proof systems because they are easier to map into propositional planning languages, and at the same time they provide typical examples of challenging reasoning patterns such as reasoning by cases. However it is possible to use our approach to encode proof search for higher order logics as a planning problem in a similar way.

Our proof system, illustrated in Figure 1, contains, for every classical connective $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies), a set of rules. Following he approach of Natural Deduction, the rules are divided in introduction (I) and elimination (E) rules. Moreover they are formulated with respect to signed (true or false) formulas: $t : A$ means that A is true, $f : A$ means that A is false. Thus, also the rules themselves are signed: for instance I_t indicates a true Introduction rule while E_f indicates a false Elimination rule. The symbol | below the line means that two successor branches are created by the application f the rule. Thus, the rule

$$\frac{t{:}A \vee B}{t{:}A \mid t{:}B} \vee_{E_t}$$

indicates an inference rule which starts from $A \vee B$ being true, and allows to create two successor branches: one where A is true and another where B is true.

$$\frac{t{:}\neg A}{f{:}A}\ \neg_{E_t} \qquad \frac{f{:}\neg A}{t{:}A}\ \neg_{E_f} \qquad \frac{t{:}A}{f{:}\neg A}\ \neg_{I_f} \qquad \frac{f{:}A}{t{:}\neg A}\ \neg_{I_t}$$

$$\frac{t{:}A \wedge B}{t{:}A, t{:}B}\ \wedge_{E_t} \qquad \frac{t{:}A, t{:}B}{t{:}A \wedge B}\ \wedge_{I_t} \qquad \frac{f{:}A}{f{:}A \wedge B}\ \wedge_{I1_f} \qquad \frac{f{:}B}{f{:}A \wedge B}\ \wedge_{I2_f}$$

$$\frac{f{:}A \wedge B}{f{:}A \mid f{:}B}\ \wedge_{E_f} \qquad \frac{f{:}A \wedge B, t{:}A}{f{:}B}\ \wedge_{E1\,f} \qquad \frac{f{:}A \wedge B, t{:}B}{f{:}A}\ \wedge_{E2\,f}$$

$$\frac{t{:}A \vee B}{t{:}A \mid t{:}B}\ \vee_{Et} \qquad \frac{t{:}A}{t{:}A \vee B}\ \vee_{I1\,t} \qquad \frac{t{:}B}{t{:}A \vee B}\ \vee_{I2\,t}$$

$$\frac{f{:}A \vee B}{f{:}A, f{:}B}\ \vee_{E\,f} \qquad \frac{f{:}A, f{:}B}{f{:}A \vee B}\ \vee_{I\,f} \qquad \frac{f{:}A, t{:}A \vee B}{t{:}B}\ MT_1 \qquad \frac{f{:}B, t{:}A \vee B}{t{:}A}\ MT_2$$

$$\frac{t{:}A \to B}{f{:}A \mid t{:}B}\ \to_{E_t} \qquad \frac{f{:}A}{t{:}A \to B}\ \to_{I1\,t} \qquad \frac{t{:}B}{t{:}A \to B}\ \to_{I2\,t}$$

$$\frac{f{:}A \to B}{t{:}A, f{:}B}\ \to_{E\,f} \qquad \frac{t{:}A, t{:}A \to B}{t{:}B}\ MP \qquad \frac{f{:}B, t{:}A \to B}{f{:}A}\ MT$$

$$\frac{t{:}A, f{:}A}{t{:}B}\ ExC \qquad \frac{}{t{:}A \mid f{:}A}\ Split$$

Fig. 1. The proof system.

The system is complete for classical propositional logic, [4] and its subsets correspond to well-known deductive systems (such as analytic tableaux [11]). Proof search for such systems can be formulated as a special case of our proof system.

A derivation of A from a set of premises Γ is a tree where nodes are sets of signed formulas, the root is an empty set, each child node is added to the tree in accordance with the expansion rules of the system, and each branch has a node containing $t{:}A$. In addition, we require that all the formulas occurring in the conclusion of any inference rule used in the derivation are subformulas of the formulas from $\Gamma \cup \{A\}$. Since a derivation with the subformula property always exists, the system is still complete for classical logic. As an example, here is a simple derivation of $B_1 \vee B_2$ from the set of facts $A_1 \vee A_2,\ A_1 \to B_1,\ A_2 \to B_2$:

[4] A proof sketch: suppose $\Gamma \models \phi$ where Γ is a finite set of propositional formulas and ϕ is a propositional formula, not necessarily in the same alphabet. We show how to construct a derivation of ϕ from Γ. We start with all formulas in Γ labelled with t and apply the Split rule on ϕ, that is create two branches, one of which contains $t : \Gamma$ and $t : \phi$, and the other $t : \Gamma$ and $f : \phi$. We need to construct a tree where every branch contains $t : \phi$. We do not need to do anything with the first branch since it already contains $t : \phi$. The second branch will be a usual refutation proof; since the elimination rules of our system include all the tableaux rules from [11], the same proof as in [11] can be used to show that every branch is guaranteed to end in contradiction, and then we can derive $t : \phi$ by ExC. Note that every formula in the derivation we just constructed is a subformula of Γ or ϕ.

$$\dfrac{\dfrac{t{:}A_1 \to B_1 \quad t{:}A_1}{t{:}B_1}\,MP}{t{:}B_1 \vee B_2}\,\vee_{I1t} \quad \Big| \quad \dfrac{\dfrac{t{:}A_1 \vee A_2}{}\,\vee_{Et} \quad \dfrac{t{:}A_2 \quad t{:}A_2 \to B_2}{t{:}B_2}\,MP}{t{:}B_1 \vee B_2}\,\vee_{I2t}$$

The Architecture. We consider agents modelled as Von Neumann reasoners, which act sequentially, attempting to derive a sentence (goal formula) ϕ from a set of facts stored in a knowledge base K. We assume that K is stored outside the agent's memory, and that instead, reasoning can only take place operating on formulas stored in memory. That is, facts must be first read into the memory, and then elaborated. The agent's memory consists of m memory cells. Our working assumption is that each cell holds a single formula (i.e., we abstract away from the size of the formulas). Applying an inference rule or loading new data from K may overwrite the contents of a memory cell.

As customary in a Von Neumann-style architecture, the memory is organized into a *call stack* and a *heap* sections. The stack is used to deal with the branching of computation threads, which in our case, is due to the application of branching reasoning rules such as e.g. $\vee_{Et}$. More in detail, it is used to save and restore memory contexts at the branching points, and is a Last-In First-Out structure organized in *frames*: a frame represents a memory context saved when branching, and push/pop operations act by creating or restoring entire stack frames. The *heap* is exploited by the reasoner by reading and overwriting memory cells, accessing cells directly via their address.

The Problem Statement. To motivate the question of resource bounds, consider the derivation of $B_1 \vee B_2$ given above. Figure 2 shows one possible way in which a reasoning agent with 4 memory cells can compute the derivation. In this example the agent only manipulates true formulae, and we therefore omit the prefix t :. In addition to the application of logical rules the reasoner needs to 'Read' formulae from K, can 'Overwrite' formulae, and can also 'Backtrack' if needed. In the table below, we omit the detailed description of the different overwrites as they are clear from the steps performed to produce the proof.

#	Applied rule	Memory content	Stack
1	`Read(`$A_1 \vee A_2$`)`	$\{A_1 \vee A_2\}$	$\emptyset$
2	$\vee_{Et}$, branch=left	$\{A_1\}$	$\{A_1 \vee A_2, A_2\}$
3	`Read(`$A_1 \to B_1$`)`	$\{A_1,\ A_1 \to B_1\}$	$\{A_1 \vee A_2, A_2\}$
4	`MP`	$\{A_1,\ B_1\}$	$\{A_1 \vee A_2, A_2\}$
5	$\vee_{I1t}$	$\{A_1,\ B_1 \vee B_2\}$	$\{A_1 \vee A_2, A_2\}$
6	`Backtrack`	$\{A_1 \vee A_2, A_2\}$	$\emptyset$
7	`Read(`$A_2 \to B_2$`)`	$\{A_2,\ A_2 \to B_2\}$	$\emptyset$
8	`MP`	$\{A_2,\ B_2\}$	$\emptyset$
9	$\vee_{I2t}$	$\{A_2,\ B_1 \vee B_2\}$	$\emptyset$

Fig. 2. A proof of $B_1 \vee B_2$ (left-first).

In step 1 the formula $A_1 \vee A_2$ is loaded into memory. At step 2 the $\vee_{Et}$ rule requires reasoning by cases; the agent decides to consider the left branch first, i.e. A_1, and to save the current state (i.e., $A_1 \vee A_2$) and other case A_2 on the stack. Steps 3–5 derive $B_1 \vee B_2$ by first reading $A_1 \rightarrow B_1$ from K and then using Modus Ponens and or introduction. Note that while the stack cannot be overwritten, the memory cells in the heap can, and the reasoner uses this feature to limit memory usage. Now backtracking takes place, and the stack content (A_2) is popped and put in memory; steps 7–9 derive $B_1 \vee B_2$ in a manner similar to the left branch.

While Figure 2 shows a derivation of $B_1 \vee B_2$ with 4 memory cells, are these the minimal memory requirements for a proof of this formula or can we achieve it using less memory resources? And if so, can we still perform the derivation in 9 steps – as we do in Figure 2 – or is there a different trade-off between time and memory usage? These are key questions in determining the computational resources (time and memory) required to construct a proof and in the rest of the paper we show how to answer them using planning.

We take the memory requirement of a derivation to be the maximal number of formulas in memory (heap and stack) on any branch of the derivation, and the time requirement to be the total number of inference steps on all the branches in the derivation. In a system with rules of conjunction introduction and elimination, as in our system, the 'number of formulas' measure would normally give a trivial constant space complexity (since any number of formulas can be connected by 'and' until they are needed). However, in our case, the restriction of $\wedge_{It}$ to satisfy the subformula property prevents this from happening. For this reason, and in the interest of simplicity, we use the number of formulas measure for most of the paper. It is straightforward to replace this measure with the number of symbols required to represent the formulas in memory, which we do in section 8.

Our approach to measuring the time and space requirements of proofs is very similar to that adopted in the proof complexity literature [7, 1]. Research in proof complexity considers non-branching proofs, and aims to establish, for various proof systems, the lower bounds on the length of proof and on the space required by the proof as a function of the size of the premises.

3 Bounded Resource Reasoning as Planning

To discuss the way bounded reasoning can be recast as planning, we start by a brief summary of the basic concepts and terminology in planning. A planning problem $P = (D, G)$ refers to a *domain* D and to a *goal* G. The domain $D = (F, O, A)$ represents the relevant field of discourse by means of a finite set F of *fluents* (for which initial values are defined), of a finite set O of *observations* on them, and of a finite set A of *actions* that can be performed; actions are defined in terms of *preconditions* which constrain their executability, and of *effects* which define the way they affect the values of fluents. The goal G describes a set of desirable domain configurations which ought to be attained. Then, the problem (D, G) consists in *building an entity π, called a* plan*, that interacts with D by reading its observations and commanding it with actions, and such that, for all possible behaviours of D when controlled by π, D satisfies G.*

Let us now consider the problem $Ex(R, K, \phi_G, M)$ of identifying the existence and minimal length of a deduction for a formula ϕ_G, starting from a set of facts K, for a reasoning agent whose memory has size M, and armed with a set of deduction rules R. In this setting, we can consider the agent as a planning domain. E.g., its memory elements can be mapped to fluents, which are fully observable to the agent; its knowledge, i.e. the values of the fluents, changes due to its usage of reasoning rules and facts - which therefore can be perceived as the actions at stake. The goal consists in building a plan (the proof) that leads the domain to finally achieve (for all possible outcomes of deduction rules) a situation where some memory cell contains ϕ_G. Under this view, the problem $Ex(R, K, \phi_G, M)$ is recast, using a function $BuildDom$, into that of finding the shortest plan to achieve ϕ_G on a planning domain $BuildDom(R, K, M)$ whose actions correspond to the reasoning rules R and to the facts K, and whose fluents represent the memory state. This makes evident a general analogy between a bounded reasoning problem and one of planning. Of course, within this framework, different recastings, i.e. different $BuildDom$ functions, are possible and useful, since they allow representing different kind of reasoning agents and provide different ways to establish resource bounds. In turn, the different models lead to different classes of planning problems, for which different techniques are required.

A first model we consider is designed to faithfully represent the behaviour of a Von Neumann propositional reasoning agent, so to allow a direct computation of time and memory bounds for it. To appropriately mimic the execution of reasoning on such architecture, such an "architecture-oriented" model must also represent the way branching is handled by saving memory contexts on the stack, and restoring them later on. That is, in this model, multiple outcomes of branching rules are "determined", faithfully representing the way the agent decides in which order to handle threads by making use of the stack. In this setting, a solution plan is a sequence of deterministic actions (including stack manipulation), which represent the reasoning agent's execution, and whose length directly measures the time spent by the agent. This means that the problem is one of *classical* planning, and that, in particular, *optimal* classical planning algorithms allow establishing the minimal time bound for a memory-bounded deduction.

A different modelling that can be considered is "proof-oriented", in that it aims at obtaining plans whose structure is the same of (tree-shaped) proofs. To achieve this, actions must have the same structure of reasoning rules; in particular, branching rules are mapped to *nondeterministic* actions with multiple outcomes. In this setting, due to the presence of nondeterminism, the planning problem becomes one of *strong* planning [6]: the goal ϕ_G needs to be finally achieved in spite of nondeterminism. Of course, similar to the "architecture-oriented" model, also in this model the stack evolution must be kept into account, on all proof branches, to faithfully represent the way the Von Neumann reasoning agent makes use of the memory. A stack-free version of such a model can be adopted, instead, to directly represent the behaviour of a *parallel* reasoning agent where multiple processing units, each owning a private memory storage, handle different branches of the proof. Again, in this case, if *optimal* strong planning is performed, the depth of the obtained plan identifies the minimal time for deducing ϕ_G within the given memory bound.

While both modelling styles are conceptually rather simple, *effective* representations of the bounded reasoning problem in terms of planning are not immediate, and are discussed in the next two sections.

4 Bounded Reasoning by Classical Planning

In general, modeling a portion of the real-world as a planning domain must obey two different and contrasting requirements. While the domain must *faithfully* represent all relevant aspects of the problem at stake, the state-space explosion issue implies that the domain must be as compact as possible, which is usually achieved by considering *abstractions* of the actual problem. This requires finding a careful balance for the problem being examined. In the following, we start by discussing the way we model the knowledge of our reference Von Neumann reasoning agent, which is the key decision point since, then, the modeling of actions is essentially implied.

Representing Knowledge For our problem, an immediate mapping of the agent's (heap and stack) memory cells as domain fluents, whose content represent the formulas stored in the cells, is easy to devise. However, such modeling is not effective, due to the fact that it fully represents the positional aspect of memory, i.e. *where* facts are stored. For our purposes, this is unneeded and harmful: to represent the agent's knowledge, it is irrelevant to know where a fact is stored on the heap, or where on the stack; all that is needed is to know *whether* a fact is stored on the heap, and *at which frames* it is stored on the stack. Thus we adopt a different modeling style, where, instead of having a fluent represent a cell, a fluent will represent a formula ϕ: ϕ will have an associated fluent $\mathbf{F}(\phi)$, which encodes (a) whether the formula is stored on the heap, and (b) for each stack frame, whether it is stored there.

Considering that, if memory has size M, the number of stack frames is bounded by $M - 1$ (since each frame occupies at least one cell), this means that each fluent is a string of M ternary values in $\{\top, \bot, \epsilon\}$:

$$\mathbf{F}(\phi_1) : \{\top, \bot, \epsilon\}^M \ \ldots \ \mathbf{F}(\phi_n) : \{\top, \bot, \epsilon\}^M$$

The digits of a fluent $\mathbf{F}(\phi_i)$ refer, right to left, to the heap, to the first stack frame, to the second stack frame, and so on up to the $(M-1)^{th}$ stack frame; namely, the value of the j-th digit $\mathbf{D}_j(\mathbf{F}(\phi_i))$ represents whether ϕ_i, at the specific area associated to the digit, is known to be true ($\top$), known to be false ($\bot$), or not known (ϵ).

This model must be enriched by describing the overall memory bound, stating that *"the memory usage U, i.e. the total number of* $\top$ *and* $\bot$ *symbols in* $\mathbf{F}(\phi_1), \ldots, \mathbf{F}(\phi_n)$*, may never exceed* M*"*:

$$U = |\{(\phi_i, j) : \mathbf{D}_j(\mathbf{F}(\phi_i)) \neq \epsilon\}| \leq M \tag{1}$$

This encoding assumes that:

– reasoning only takes place considering heap values, i.e. the least significant digit (LSD - i.e. $\mathbf{D}_0(\mathbf{F}(\phi_i))$) of the fluents;

– when backtracking occurs, popping the top stack frame means simply "shifting right" each string;
– when branching occurs, in essence, pushing a new stack frame means "shifting left, retaining the LSD" for all fluents. On top of this, each formula ϕ_i generated by a branching rule must be appropriately stored in the heap or in the top stack frame, affecting $\mathbf{D}_0(\mathbf{F}(\phi_i))$ and $\mathbf{D}_1(\mathbf{F}(\phi_i))$.

Representing Actions As mentioned, the domain actions represent the application of reasoning rules, and the usage of facts. In particular, rule schemata can be presented as planning action schemata, whose arguments represent the rule premises, and allow their instantiation onto specific formulas. E.g., the action schema $\vee_{I1t}(\psi_1, \psi_2)$, with $\psi_{1,2} \in \{\phi_1, \ldots, \phi_n\}$, indicates the application of $\vee_{I1t}$ on formula ψ_1, which is known to hold (i.e. $\mathbf{D}_0(\mathbf{F}(\psi_1)) = \top$), to derive that the formula $\psi_1 \vee \psi_2$ also holds (i.e. $\mathbf{D}_0(\mathbf{F}(\psi_1 \vee \psi_2)) = \top$). Notice that ψ_2 also appears as an argument in the schema, since it is necessary for the agent to choose explicitly which formula must be unioned to ψ_1, in order to fully instantiate the rule.

To choose amongst applicable rules and facts is not the only choice the agent must take, and such choices need to be also explicitly represented as rule arguments:

– the agent must consider the possibility of *overwriting* formulas on the heap; this is essential to save memory, and is a choice that can be taken for every outcome of the fact/rule being applied. This means that a rule that produces n outcomes in the heap comes together with n arguments $\omega_1, \ldots, \omega_n$ ranging in $\{\epsilon, \phi_1, \ldots, \phi_n\}$, specifying whether the n-th outcome must overwrite a known formula, and if so, which one. E.g., $\vee_{I1t}$ will be of the form $\vee_{I1t}(\psi_1, \psi_2, \omega_1)$.
– in the case of branching rules, the agent must decide in which order to proceed, i.e. what to keep on the heap, and what to push on the stack. In particular, for rules with two branches, such as the ones we consider, this implies adding a "left/right" direction argument δ. Then, $\vee_{Et}$ has the form $\vee_{Et}(\psi_1, \omega_1, \delta)$; notice that since only one of its outcomes stays on the heap, only one ω argument is needed.

The following rule schema defines the available actions A for the agent, which essentially consist of those in Figure 1, plus the read and backtracking actions:

$$\neg_{E_t}(\psi_1, \omega_1) \qquad \neg_{E_f}(\psi_1, \omega_1) \qquad \neg_{I_f}(\psi_1, \omega_1) \qquad \neg_{I_t}(\psi_1, \omega_1)$$

$$\wedge_{E_t}(\psi_1, \psi_2, \omega_1, \omega_2) \qquad \wedge_{I_t}(\psi_1, \psi_2, \omega_1) \qquad \wedge_{I1_f}(\psi_1, \psi_2, \omega_1) \qquad \wedge_{I2_f}(\psi_1, \psi_2, \omega_1)$$

$$\wedge_{E_f}(\psi_1, \omega_1, \delta) \qquad \wedge_{E1f}(\psi_1, \omega_1) \qquad \wedge_{E2f}(\psi_1, \omega_1)$$

$$\vee_{Et}(\psi_1, \omega_1, \delta) \qquad \vee_{I1t}(\psi_1, \psi_2, \omega_1) \qquad \vee_{I2t}(\psi_1, \psi_2, \omega_1)$$

$$\vee_{Ef}(\psi_1, \omega_1, \omega_2) \qquad \vee_{If}(\psi_1, \psi_2, \omega_1) \qquad MT_1(\psi_1, \psi_2, \omega_1) \qquad MT_2(\psi_1, \psi_2, \omega_1)$$

$$\rightarrow_{Et}(\psi_1, \psi_2, \omega_1) \qquad \rightarrow_{I1t}(\psi_1, \omega_1) \qquad \rightarrow_{I2t}(\psi_1, \omega_1)$$

$$\rightarrow_{Ef}(\psi_1, \omega_1, \delta) \qquad MP(\psi_1, \psi_2, \omega_1) \qquad MT(\psi_1, \psi_2, \omega_1)$$

$$ExC(\psi_1, \psi_2, \omega_1) \qquad Split(\psi_1, \omega_1, \delta) \qquad Btk \qquad Read(\psi_1, \omega_1)$$

Note that it is possible to univoquely instantiate all rules using a single ϕ argument, which, depending on the rule, can be either the premise or the result of the rule. For instance, in the case of $\vee_{I2t}$, it is possible to use a single argument ψ corresponding to $\psi_1 \vee \psi_2$, which would implicitly define ψ_1 and ψ_2 as the rule premises. While this is actually used in the implementation, here, for the sake of clarity, we keep the semantics of arguments as being premises of rules.

Each rule schema comes with its specific executability preconditions, posing constraints on the current knowledge, and with effects specifying constraints on the next memory configuration. For instance, the precondition for $\vee_{I1t}(\psi_1, \psi_2, \omega_1)$ requires that ψ_1 is known to hold (i.e. $\mathbf{D}_0(\mathbf{F}(\psi_1)) = \top$). We omit the complete description of preconditions and effects for lack of space. We only note that they can be obtained almost directly from the semantics of the connectives related to the corresponding rule. On top of this, all actions have two kinds of general contraints. First, "overwrite" arguments must refer to facts actually on the heap, i.e. $\forall i : \mathbf{D}_0(\omega_i) \neq \epsilon$. Second, an action is not executable if its application would cause a memory overflow. For each action instance, this is easily decided a priori from its execution, taking into account the current memory usage U, the number of previously unknown formulas produced by the rule, and the number of overwritten formulas.
Naturally, also the result of the application of rules is specific to each rule. In particular:

- reading facts from K only affects the LSD of the fluent associated to the fact, and possibly the LSD of an overwritten formula. For instance, $Read(A1 \vee A2, \epsilon)$ only sets $\mathbf{D}_0(\mathbf{F}(A1 \vee A2)) = \top$.
- non-branching reasoning rules only affect the LSDs of some fluents: those related to the formula(s) produced by the rule application, and those related to the overwritten formulas. All other digits, and all other fluents, are unaffected. For instance, $\vee_{I1t}(B_1, B_2, B_2)$, which produces $B_1 \vee B_2$ overwriting it in place of B_2, affects only $\mathbf{D}_0(\mathbf{F}(B_1 \vee B_2))$, which becomes $\top$, and $\mathbf{D}_0(\mathbf{F}(B_2))$, which becomes ϵ.
- popping a stack frame, i.e., in terms of reasoning, backtracking, simply shifts right all fluent strings, introducing an ϵ at their most significant digit.
- branching rules operate on all fluents, by shifting them left and keeping their LSD; moreover, the two fluents correspondent to the rule outcomes are affected so that in one, the LSD is set to a truth value, and in the other, the second digit is set to a truth value - representing that one formula is dealt immediately on the heap, while the other is stored at the top stack frame. For instance, starting from $\mathbf{F}(A_1 \vee A_2) = \epsilon\top$ and $\mathbf{F}(A_1) = \mathbf{F}(A_2) = \epsilon\epsilon$, applying $\vee_{Et}(A_1 \vee A_2, A_1 \vee A_2, left)$ leads to $\mathbf{F}(A_1 \vee A_2) = \mathbf{F}(A_2) = \top\epsilon$ and $\mathbf{F}(A_1) = \epsilon\top$.

In Figure 3 we show an example of a deduction for our running example, where the facts in the knowledge base are $K = \{A_1 \vee A_2, A_1 \rightarrow B_1, A_2 \rightarrow B_2\}$ and the goal is $B_1 \vee B_2$. We consider an agent with 4 memory cells, and for the sake of simplicity, from the set of fluents associated to $\{A_1, A_2, B_1, B_2, A_1 \vee A_2, B_1 \vee B_2, A_1 \rightarrow \mathrm{B}_1, A_2 \rightarrow B_2\}$, we omit showing those whose formula is not stored anywhere in memory, that is fluents of the form $\mathbf{F}(\phi) = \epsilon\epsilon\epsilon\epsilon$. Also, for the sake of readability, we leave the $\mathbf{F}(\cdot)$ notation implicit. The proof shows that a deduction for $B_1 \vee B_2$ is possible in 4 cells, taking 9 steps.

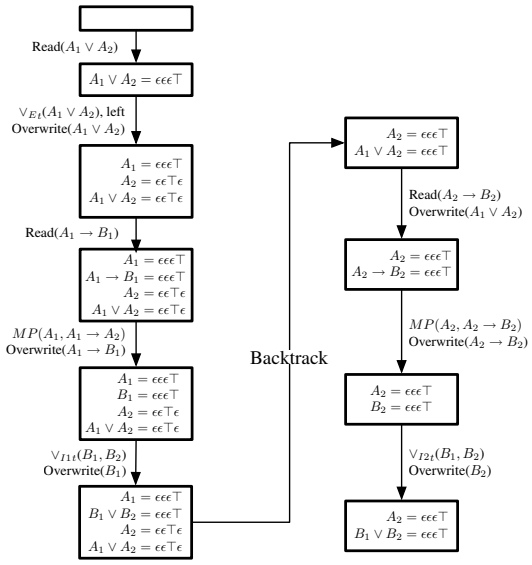

Fig. 3. A 'sequential' proof.

5 Bounded Reasoning by Strong Planning

While the previous model is "architecture-oriented", one can follow a "proof-oriented" modelling where the produced proofs are shaped as logical trees. As we shall see, one advantage of such a model is that it can further abstract from the specific memory configuration, while still representing the relevant features of our reference reasoning agent. Moreover, such a model can be easily adapted to represent reasoning agents based on a parallel architecture. Again, we will show this by first discussing the way the agent's knowledge is represented.

Representing Knowledge In a "proof-oriented" model, different than in the previous "architecture-oriented" model, separate proof branches correspond to deduction threads evolving in parallel, independently. This will be mapped into the planning process; hence, in such a model, proof search can be performed with no need to represent any context push/pop mechanism. Nevertheless, at each proof step, in order to evaluate memory consumption, we need to represent the actual memory usage of the underlying Von Neumann reasoning agent, which actually uses the stack. Then, to achieve a representation of knowledge as compact as possible, we build on top of the "position abstraction" taken for the previous model, and apply a further abstraction, so that:

1. concerning formulas, we only represent the *current* knowledge on them, i.e. their heap values;
2. concerning the stack, we only represent its current size.

This leads to a modelling where formula-representing fluents are 3-valued, and an additional numeric fluent S represents the stack size:

$$\begin{gathered}\mathbf{F}(\phi_1) \in \{\top, \bot, \epsilon\} \\ \cdots \\ \mathbf{F}(\phi_n) \in \{\top, \bot, \epsilon\} \\ S \in [0, M-1]\end{gathered}$$

Again, the bound on the memory usage U must be added; of course here, U is computed in a different way, although it produces the same result:

$$U = S + |\phi_i : \phi_i \neq \epsilon| \leq M \tag{2}$$

It is immediate to see that such a representation is significantly smaller, in terms of possible states, than the one considered in the previous model, since different stack configurations with equivalent memory occupation are grouped together. On the other side, this representation features, together with an additional state-level branching factor implied by the fact that rules correspond to nondeterministic actions, transform the problem from classical to strong planning.

It is important to notice that a stack-free version of this modelling provides a direct representation of a different, parallel reasoner architecture, where reasoning is actually performed in parallel by different processing units, each featuring a private memory bank of size M. In this case, the plan depth directly represents the overall time spent by the parallel architecture to complete the proof, and optimal planning can be used to establish a minimal bound in that respect.

Representing Actions In terms of available actions and arguments, the parallel modelling is substantially the same as the sequential one, apart from the absence of the 'backtracking' operator. Of course, what changes is the way actions are defined, in terms of preconditions and effects.

In terms of preconditions, the only change is in the description of the overflow-avoiding executability constraint, which makes use of the new computation for the memory usage U.

In terms of effects, the changes only regard the branching rules, and in that case they are considerable. Indeed, the parallel modelling is simpler in that, when two outcomes are produced, they correspond to two different assignments to the sets of fluents; therefore, for instance, starting from $\mathbf{F}(A_1 \vee A_2) = \top$, $\mathbf{F}(A_1) = \mathbf{F}(A_2) = \epsilon$, and $S = 0$, and applying $\vee_{Et}(A_1 \vee A_2, A_1 \vee A_2, left)$ produces two states: a state where $\mathbf{F}(A_1 \vee A_2) = \mathbf{F}(A_2) = \epsilon$, $\mathbf{F}(A_1) = \top$, $S = 2$, and a state where $\mathbf{F}(A_1 \vee A_2) = \mathbf{F}(A_1) = \top$, $\mathbf{F}(A_2) = \epsilon$, $S = 0$.

An example In Figure 4 we show the same example proof considered in the previous section model, but taking the parallel model as a reference. It is easy to see, by discarding the stack fluent S, that a parallel reasoning agent would only require 2 cells to complete such a proof in 5 steps.

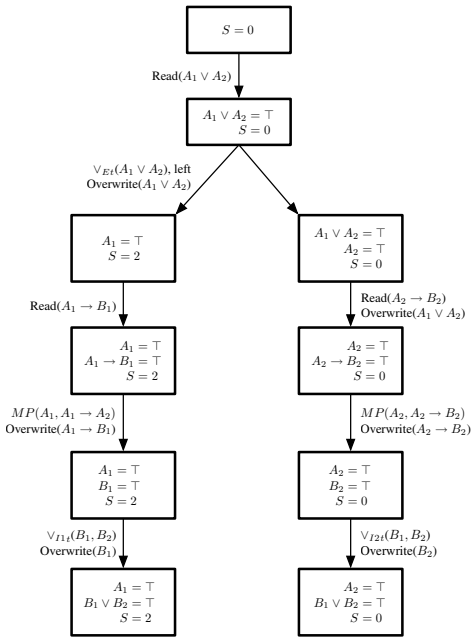

Fig. 4. A 'branching' proof.

6 Forgetful Reasoning Agents

The possibility of "overwriting" heap cells is not the only possibility that an agent must take into account in order to produce a proof within its memory bounds. Indeed, considering other options is essential to achieve even more considerable memory savings.

In particular, as a skilled reader may have spotted from our example proofs already, a key issue is related to the behaviour in the presence of branching rules. There, the standard behaviour implemented by a Von Neumann machine stores the whole current context on the stack. However, it may well be the case that some of the facts currently known will not be useful in the pending branch, and therefore, pushing them on the stack only wastes memory.

A "smart" agent would act by selecting which "relevant" facts must be stored on the stack, and storing only them. For instance, in our example, $A_1 \vee A_2$ is not used in the second branch of the proof; and even if it were, since it is an axiom, it could have been re-read, rather than saved in memory. So a "smart" agent would have been "forgetful" w.r.t. saving $A_1 \vee A_2$ on the stack, leading to a proof with the same schema of the one we presented, but where (a) in the left branch we would have $S = 1$, and (b) on top of the right branch, $A_1 \vee A_2 = \top$ would not appear. It is easy to see that such a proof only requires 3 cells, and still takes 9 steps.

Two remarks are in order. First, the issue of "forgetfulness" is indeed specific of stack treatment, since the ability to overwrite facts on the heap renders harmless the

presence of useless facts in that memory area. Second, while, during a specific proof search, forgetting some facts may lead to not derive a fact which would be otherwise derivable, it is clear that an agent which considers forgetfulness as an option, and tries out all the different choices (including "not forgetting anything") is still complete. Note also, that here we are not concerned on whether "forgetful" agents can be realised in efficient manners. Our main point here is that "forgetfulness" is a strategy that can help to build proofs which use a smaller number of memory cells.

Given this, the agent's forgetfulness choices are modelled as additional arguments to branching rules. Namely, branching rules are enriched with additional arguments $\rho_1, \ldots, \rho_M$ in $\epsilon, \phi_1, \ldots, \phi_n$, and formula ϕ_j must be "forgotten" on the pending branch iff $\exists i : \rho_i = \phi_j$. Of course, it must be possible only to forget facts which are currently in the heap. Moreover, to prune out equivalent symmetric specifications of the same forgetfulness, we impose the following ordering constraints on the ρ_i arguments:

1. if k formulas must be forgotten, they are specified within $\rho_1, \ldots, \rho_k$, i.e. $\rho_i = \epsilon \rightarrow \forall j > i : \rho_j = \epsilon$
2. the order of forgotten formulas is reflected in the order they are specified as ρ_i arguments: $\forall i, j : \rho_i = \phi_l \wedge \rho_j = \phi_m \wedge (i < j) \rightarrow (l < m)$.

7 Implementation in MBP and Experiments

We implemented our approach using the MBP planner, devising an automated modeller which converts a bounded-resource reasoning problem (given as a set of facts, a goal formula ϕ_G and a memory bound) into a planning problem description, following one amongst our proposed models. Then, we use MBP to identify (a) whether a deduction exists for ϕ_G within the given memory bound, and (b) if so, which is the minimal number of computation steps it takes. Of course, as a side-effect, if a deduction exists, we also obtain an optimal strategy, which can be outputted or simulated.

Our choice of MBP has several reasons. First, MBP is a flexible planning tool, which combines different planning algorithms, amongst which the optimal classical and strong planning we need. Second, it is an effective tool; due to its internal data representations which are based on Binary Decision Diagrams, it has proved to be state of the art, especially for what concerns optimal planning in the presence of nondeterminism. Third, its input language, SMV, is very flexible and apt to express the kind of models at hand. In particular, the ability of SMV to express the behaviour of a domain by modularly specifying the dynamics of each of its fluents is very useful, and can be combined with the usage of constraints spanning across fluents to achieve very compact encodings – even when modelling imply nondeterministic dynamics. In particular, in our models, the dynamics of each fluent is defined independently, by a case analysis, using SMV's `assign` and `case` constructs. Each case identifies one action that affects the fluent, and the "default" case is used to specify the inertial behaviour. This allows for a very compact encoding: in no case, the SMV model exceeds 52 Kbyte of size.

Based on our automatically generated SMV encodings, we ran tests with MBP, considering 8 different deduction schemata, and 3 reference architectures: a standard Von Neumann reasoning agent, a Von Neumann reasoning agent considering forgetfulness, and a parallel reasoning agent. All tests have been performed with a cut-off time at

1800 seconds on a Linux machine running at 2.33GHz with 1.8Gb of RAM; results are reported in Figures 5 and 6.[5]

	Von Neumann Std		**Von Neumann Fgt**		**Parallel**	
	min. M	min. t	min. M	min. t	min. M	min. t
P_1	4 ($t \geq 10$)	7 ($M \geq 5$)	4 ($t \geq 7$)	7 ($M \geq 4$)	3 ($t \geq 5$)	5 ($M \geq 3$)
P_2	3 ($t \geq 11$)	10 ($M \geq 4$)	3 ($t \geq 11$)	10 ($M \geq 4$)	2 ($t \geq 9$)	6 ($M \geq 3$)
P_3	4 ($t \geq 24$)	24 ($M \geq 4$)	4 ($t \geq 24$)	24 ($M \geq 4$)	3 ($t \geq 9$)	9 ($M \geq 3$)
P_4	3 ($t \geq 13$)	13 ($M \geq 3$)	3 ($t \geq 13$)	13 ($M \geq 3$)	3 ($t \geq 6$)	6 ($M \geq 3$)
P_5	3 ($t \geq 9$)	9 ($M \geq 3$)	3 ($t \geq 9$)	9 ($M \geq 3$)	3 ($t \geq 5$)	5 ($M \geq 3$)
P_6	3 ($t \geq 16$)	13 ($M \geq 4$)	3 ($t \geq 15$)	13 ($M \geq 4$)	2 ($t \geq 8$)	7 $M \geq 3$)
P_7	3 ($t \geq 8$)	8 ($M \geq 3$)	3 ($t \geq 8$)	8 ($M \geq 3$)	2 ($t \geq 4$)	4 ($M \geq 2$)
P_8	2 ($t \geq 8$)	8 ($M \geq 2$)	2 ($t \geq 8$)	8 ($M \geq 2$)	2 ($t \geq 5$)	5 ($M \geq 2$)

Fig. 5. Memory and time bounds for the three models.

	Von Neumann Std	
	standard	proof
P_1	1.20	0.27
P_2	3.28	0.61
P_3	33.43	1.82
P_4	26.25	3.46
P_5	1.83	0.33
P_6	30.14	2.83
P_7	0.79	0.24
P_8	0.24	0.10

Fig. 6. Computation times (s).

Problems vary in size and nature, with up to 14 fluents and 30 actions. Problem P_8 is our reference example; as one can see, in that case, one can prove $B_1 \vee B_2$ using just 2 cells, which seems surprising. The proof is reported in Figure 7, and makes use of two main ideas: (a) branching immediately, so that context saving is not required, and (b) carefully selecting the order in which branches are handled, so that "the first branches are the easier ones to solve" (since they have some memory occupied by the stack). As a result, the proof is very different from the intuitive ones shown in the previous sections. The fact that bounds are not trivial, and neither are proof-saving strategies, is also true for most of the other examples.

More in general, Figure 5 shows an evident trade-off between space and time: when minimal memory is used (denoted by 'min. M' in the results), more time is required, and faster proofs require more memory (we denote it by 'min. t', giving in parenthesis the

[5] The benchmark instances and executables of the planner and the modelers are available at `http://boundedreasoning.i8.com/`.

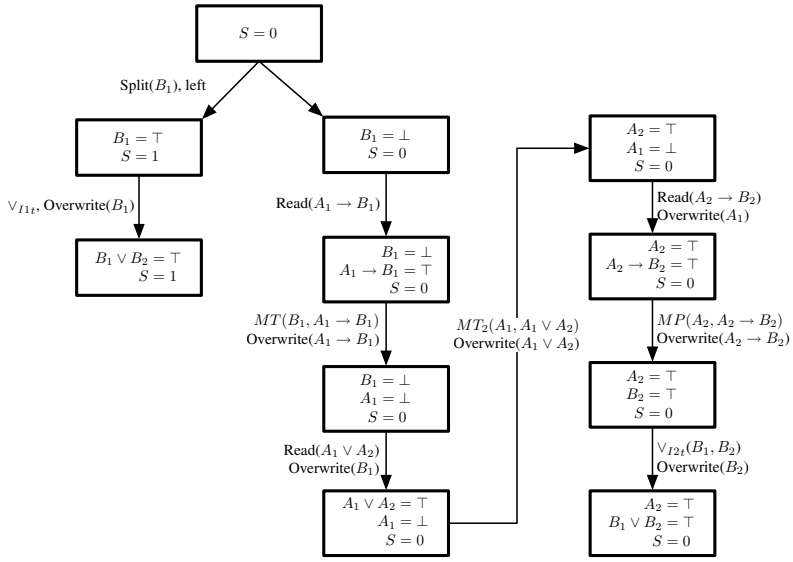

Fig. 7. A proof with 2 cells.

memory used for that minimum proof, within the memory and time cutoff used). Also, the experimental results show clearly the importance of forgetfulness to save memory: in problems P_1 and P_4, the minimal bounds of memory for a forgetful Von Neumann reasoning agent improve those of a standard reasoning agent (while not requiring more time).

In terms of MBP performance, it is interesting to compare the relative performances of the "classical" and "strong" models of the problem for Von Neumann reasoning agents. Results for the forgetful Von Neumann reasoning agent are presented in Figure 6, and show that there is a clear advantage of the "proof-oriented" model. This indicates that the effective way MBP handles nondeterminism pays off, and hints at using the "strong" first to identify memory bounds, leaving the "classical" model to check time limits, once the memory bound is fixed.

8 Different Resource Models

While our work took definite assumptions over the underlying model for computational resources, established in Section 3, our approach is general enough to be easily adaptable to different views. In particular, different data storage means and methodologies can be mapped into different, but equally interesting, memory bound models. For instance, taking the assumption that every formula has size one is very apt to represent situations where the actual representation of formulae is not stored in the working memory, but on an external mass storage, and the data on the heap and stack are simply "pointers" to such representations. Vice versa, the memory usage of a reasoning system

where formulae are stored in the main memory cannot be faithfully captured by such a model, since in that case, different formulae occupy different memory quotas. In this case, we can take the reasonable assumption that a formula is represented as a parse tree, where logical connectives occupy single memory cells, as well as atomic facts. Thus, for instance, the formula $(A \wedge B) \vee C$ occupies 5 memory cells.

We now show that shifting from the previous, "unitary formula size" model to this new "structural formula size" model is very simple to achieve. In fact, all we need to affect is the way the memory bound is described, by introducing a function $\mathbf{S}(\phi_i)$ that associates a fact ϕ_i to a positive integer representing its memory occupation. Considering the modeling described in Section 4 to solve the problem by classical planning, this means that the overall memory bound described by Equation (1) must become

$$U = \sum_i \sum_j (\mathbf{S}(\phi_i) \times \mathbf{A}(\phi_i, j)) \leq M$$

where $\mathbf{A}(\phi_i, j)$ attains value 1 if $\mathbf{D}_j(\mathbf{F}(\phi_i)) \neq \epsilon$, and attains value 0 otherwise. Similarly, in the modeling described in Section 5 to solve the problem by strong planning, the constraint described by Equation (2) must be replaced by

$$U = S + \sum_i (\mathbf{S}(\phi_i) \times \mathbf{A}(\phi_i)) \leq M$$

where $\mathbf{A}(\phi_i)$, this time, has value 1 if $\phi_i \neq \epsilon$, and value 0 otherwise. Note that, in both cases, replacing the **S** function with the constant function 1 causes the modelings to fall back in the unitary formula size case.

Expressiveness considerations apart, the relevance of being able to apply different models of resource utilization stands in the fact that the optimality of derivation strategies is only relative to a specific model. We show this considering a simple scenario where the knowledge base K contains the five formulae

$$(A \wedge B) \wedge (C \wedge D) \qquad ((B \wedge A) \wedge (D \wedge C)) \rightarrow G \qquad (B \wedge A) \rightarrow G_1$$
$$(D \wedge C) \rightarrow G_2 \qquad (G_1 \wedge G_2) \rightarrow G$$

and the goal is G.

If we consider the unitary formula size model, we can run MBP over the classical model representation of the problem and we obtain that at least 3 memory cells are needed to derive G, and the corresponding shorter proof P_9 takes 9 steps. The proof is shown in Figure 8 and makes use only of the first two facts listed in the knowledge base above. No shorter proof can be found, even when more than 3 memory cells are available.

However, the situation is quite different when using the structural formula size model. In this case, again running MBP over the classical planning model, we obtain that the most memory-saving proof is a proof P_{15} that, making use of all the facts in K apart from $((B \wedge A) \wedge (D \wedge C)) \rightarrow G$, takes 15 steps, using 9 memory cells (see Figure 9).

Intuitively, in this model, applying the modus ponens rule over the facts $(B \wedge A) \wedge (D \wedge C)$ and $((B \wedge A) \wedge (D \wedge C)) \rightarrow G$, like P_9 does, is too memory-expensive, as

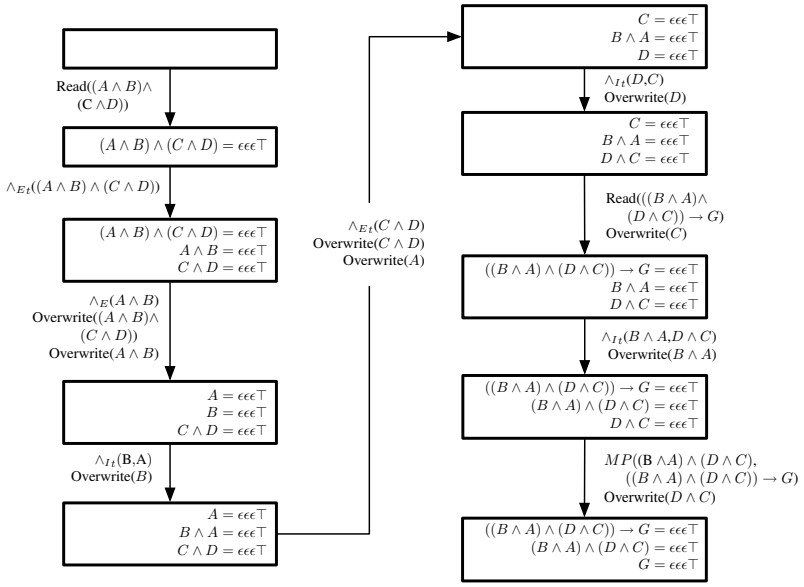

Fig. 8. P_9: a derivation of G in 9 steps.

these two formulas, together, occupy 16 memory cells. Rather, it becomes convenient, in terms of memory usage, to go through the derivation of G_1 and G_2, which exploits "smaller" facts (but takes additional derivation steps). This consideration of course did not apply to the unitary size model, where all formulae are regarded as having the same size. We also observe that in P_{15}, again to save memory, the fact $(A \wedge B) \wedge (C \wedge D)$ is read twice from K (since it is needed twice, but it is overwritten). Indeed, if some more memory is available (between 10 and 15 cells), the proof P_{13} is generated, which does not perform such re-reading, therefore saving two reasoning steps (see Figure 10).

Finally, if at least 16 memory cells are available, the proof P_9 is identified as the faster way to obtain G (in 9 steps). These results indicate that, in the structural formula size model, P_{15}, P_{13}, P_9 are all interesting, as they express different memory-space trade-offs: each of them takes longer, but is more memory-efficient, than the next. Vice versa, in the unitary formula size model, P_{15} and P_{13} are not more memory-effective than P_9 - actually, P_{13} is even more memory consuming. As such, in the unitary formula size model there is no reason for using P_{15} or P_{13}, under any possible resource configuration. Of course, these considerations about the dependency of optimality trade-offs on the resource models are general, and different memory models can be very easily accommodated by appropriately devising the **S** function.

9 Conclusions and future work

The study of the behaviour of systems in the presence of computational resource bounds is receiving considerable attention, due to its practical impacts. This has led to develop-

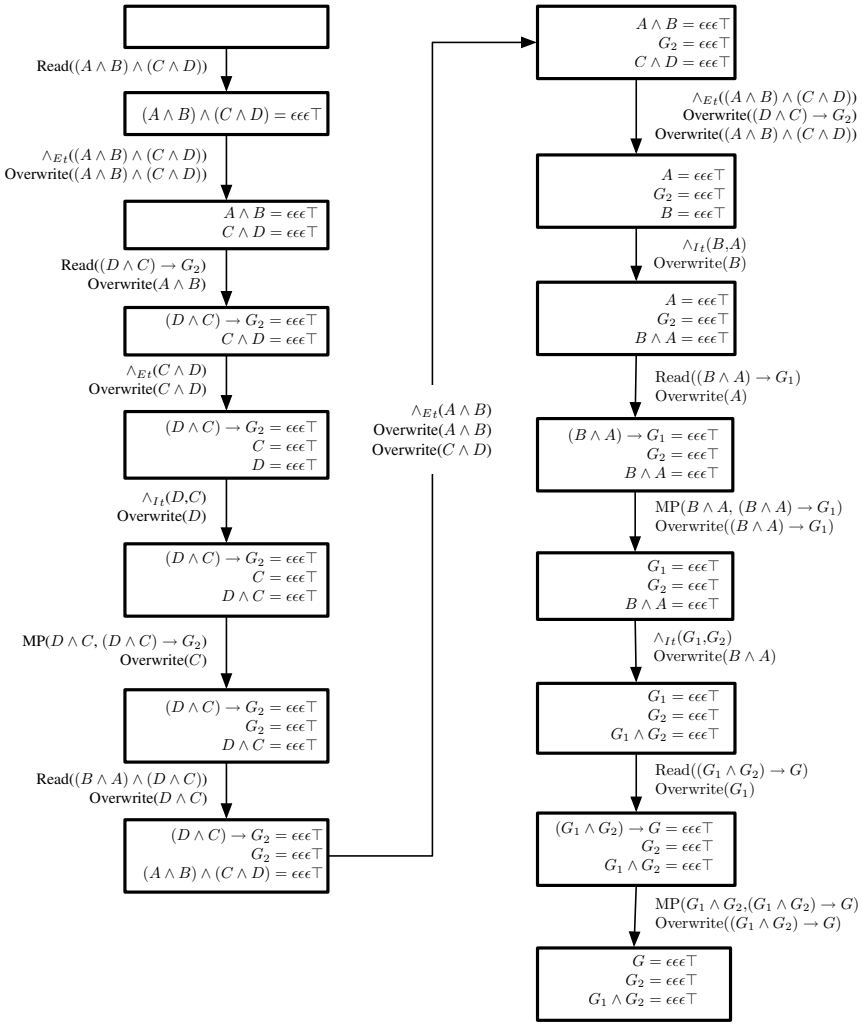

Fig. 9. P_{15}: a derivation of G in 15 steps.

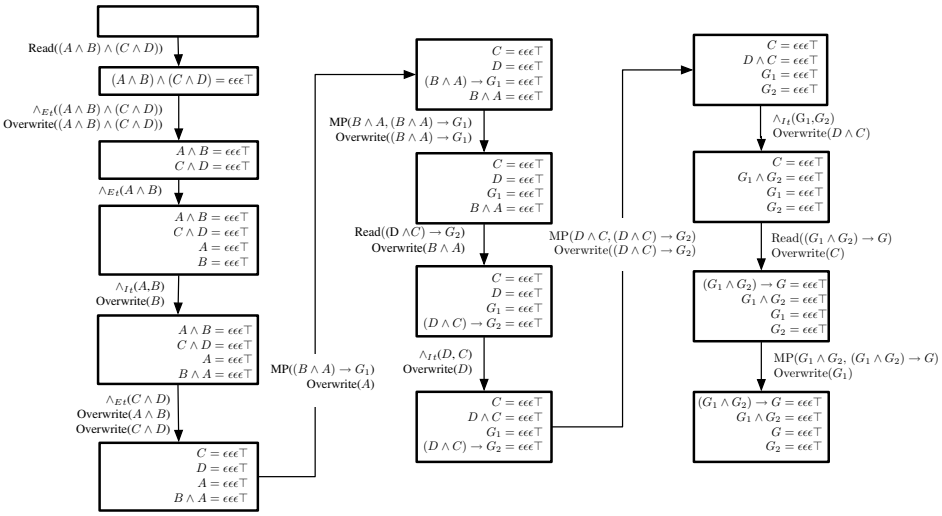

Fig. 10. P_{13}: a derivation of G in 13 steps.

ing models and approaches to deal with bounded rationality, with a particular emphasis on the design of anytime algorithms to work within strict time limits (see e.g. [10]).

In this work, we link reasoning to planning, and provide an approach to evaluate the minimal time and memory bounds for solving reasoning problems, a crucial step to appropriately dimension the computational power of devices that support reasoning functionalities. While the connection between deduction and planning has already been established for a variety of logics, e.g., temporal, linear and propositional logics, see [8, 4, 9], existing work has essentially focused on using theorem provers to build plans. Here, we go the other way around, showing that reasoning - and taking into account resource bounds - can be recast in terms of planning, and that different re-castings are possible, interesting and useful. Our experiments show that, already considering simple scenarios, interesting memory-space trade-offs appear; minimal resource bounds are often surprising, and so are the reasoning strategies enforcing them.

While the focus of this work is on propositional logics, it has to be remarked that the general approach is open to extensions to different logics and logic fragments, which can be pre-compiled into planning domains following the conceptual scheme presented here. Indeed, a further direction of investigation, which we intend to pursue, concerns the relation between the power of the reasoning system, and the minimal resource bounds for performing deductions. Studying this can help identifying the proper 'kind' of reasoning agents, given the resource bounds and the kind of proofs that need be carried out. Also, our approach can be very easily adapted to follow different memory models, e.g. one where the occupancy of formulas depends on their internal structure. Such different models may imply different bounds, and different time- and memory-saving strategies: for instance, in our model, conjunction introduction can be used to compress formulas to save space (trading off time for space); in different models, this may not be the case.

A further relevant concern stands in improving the scalability of our approach, to cover complex scenarios where reasoning taks place over large sets of facts. This is crucial to lift the approach to practical settings, using deduction to model the behavior of actual bounded-reasoning agents. At the current stage, however, scalability is limited, since we perform an exhaustive blind plan search, in order to never rule out any resource-optimal deduction. This makes the complexity of identifying optimal resource bounds extremely high, and in such a setting even the most effective technologies for representing and searching proof trees can only help to a limited extent. To solve this issue, we need to identify heuristics or strategies that significantly reduce the complexity of the search, adapting planning techniques which embed resource optimization, such as those of [5], or techniques which allow describing control rules, such as those of [2]. Of course, the critical issue in this respect stands in guaranteeing that such heuristics and strategies preserve resource optimality. This is far from trivial, since, as witnessed in our examples, resource-optimal strategies can be far from intuitive. Such a challenging problem is the key long term goal of our research line.

References

1. Michael Alekhnovich, Eli Ben-Sasson, Alexander A. Razborov, and Avi Wigderson. Space complexity in propositional calculus. *SIAM J. Comput.*, 31(4):1184–1211, 2002.
2. Fahiem Bacchus and Froduald Kabanza. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.
3. P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. MBP: a model based planner. In *Proceedings of the IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information*, Seattle, August 2001.
4. W. Bibel. A Deductive Solution for Plan Generation. *New Generation Computing*, 4:115–132, 1986.
5. S. Chien, G. Rabideau, R. Knight, R. Sherwood, B. Engelhardt, D. Mutz, T. Estlin, B. Smith, F. Fisher, T. Barrett, G. Stebbins, and D. Tran. ASPEN - Automated Planning and Scheduling for Space Mission Operations. In *Proc. of SpaceOps*, 2000.
6. A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artificial Intelligence*, 147(1/2):35–84, 2003.
7. A. Haken. The intractability of resolution. *Journal of Theoretical Computer Science*, 39(2-3):297–308, 1985.
8. E. Jacopin. Classical AI planning as theorem proving: The case of a fragment of linear logic. In *AAAI Fall Symposium on Automated Deduction in Nonstandard Logics*, pages 62–66, Palo Alto, California, 1993. AAAI Press.
9. Henry Kautz and Bart Selman. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1194–1201. AAAI Press, 1996.
10. Marek Petrik and Shlomo Zilberstein. Anytime Coordination Using Separable Bilinear Programs. In *Proceedings of the Twenty-Second National Conference on Artificial Intelligence (AAAI-07)*, 2007.
11. Raymond M. Smullyan. *First-Order Logic*. Dover Publications, 1995.