Searching for Sequential Plans
Using Tabled Logic Programming

Roman Barták and Jindřich Vodrážka

Charles University in Prague, Faculty of Mathematics and Physics, Czech Republic

Abstract. Logic programming provides a declarative framework for modeling and solving many combinatorial problems. Until recently, it was not competitive with state of the art planning techniques partly due to search capabilities limited to backtracking. Recent development brought more advanced search techniques to logic programming such as tabling that simplifies implementation and exploitation of more sophisticated search algorithms. Together with rich modeling capabilities this progress brings tabled logic programing on a par with current best planners. The paper brings an initial experimental study comparing various approaches to search for sequential plans in the Picat planning module.

Keywords: planning; tabling; iterative deepening; branch-and-bound

1 Introduction

Automated planning was an important area for Prolog. PLANNER [5] was designed as a language for proving theorems and manipulating models in a robot, and it is perceived as the first logic programming (LP) language. Nevertheless, since the design of STRIPS planning model [6], planning approaches other than LP were more successful. SAT-based planning [9] is probably the closest approach to logic programming that is competitive with best automated planners.

For decades the so called domain-independent planning has been perceived as the major direction of AI research with the focus on "physics-only" planning domain models. This attitude is represented by International Planning Competitions (IPC) [8] that accelerated planning research by providing a set of standard benchmarks. On the other hand and despite the big progress of domain-independent planners in recent years, these planning approaches are still rarely used in practice. For example, it is hard to find any of these planners in areas such as robotics and computer games. This is partly due to low efficiency of the planners when applied to hard real-life problems and partly due to missing guidelines about how to describe planning problems in such a way that they are efficiently solvable.

IPC accelerated research in domain-independent planning by providing encodings (domain models) for many benchmark problems. On the other hand, as everyone is using IPC benchmark problems to evaluate the planners, there has not been almost any research about how to encode the planning problems efficiently. Also, though the role of domain knowledge is well known in planning [4],

24

the domain-dependent planners were banned from IPC which further decreased interest in alternative approaches to model and solve planning problems.

Recently, tabling has been successfully used to solve specific planning problems such as Sokoban [20], the Petrobras planning problem [2], and several planning problems used in ASP competitions [23]. This led to development of the `planner` module of the Picat programming language. This general planning system was applied to various domains in IPC and compared with best domain-independent optimal planners [24] as well as best domain-dependent planners [3]. In this paper we summarize the modeling and solving capabilities of Picat and we focus on their deeper experimental comparison.

2 Background on Planning

Classical AI planning deals with finding a sequence of actions that change the world from some initial state to a goal state. We can see AI planning as the task of finding a path in a directed graph, where nodes describe states of the world and arcs correspond to state transitions via actions. Let $\gamma(s, a)$ describe the state after applying action a to state s, if a is applicable to s (otherwise the function is undefined). Then the planning task is to find a sequence of actions $\langle a_1, a_2, \ldots, a_n \rangle$ called a *plan* such that, s_0 is the initial state, for each $i \in \{1, \ldots, n\}$, a_i is applicable to the state s_{i-1} and $s_i = \gamma(s_{i-1}, a_i)$, and, finally, s_n satisfies a given goal condition. For solving cost-optimization problems, each action has assigned a non-negative cost and the task is to find a plan with the smallest cost.

As the state space is usually huge, an implicit and compact representation of states and actions is necessary. Since the time of Shakey, the robot [15, 6], a *factored representation* of states is the most widely used. Typically, the state of the world is described as a set of predicates that hold in the state or by a set of values for multi-valued state variables. Actions are then describing changes of the states in the representation, for example, actions make some predicates true and other false or actions change values of certain states variables. The Planning Domain Definition Language (PDDL) [13] is the most widely used modeling language for describing planning domains using the factored representation of states. This is also the language of IPC competitions.

In Picat we will preserve the state-transition nature of classical AI planning, but instead of factored representation we will use a *structured representation* of states. Like in the PDDL, each action will have pre-conditions verifying whether the action is applicable to a given state. However, the precondition can be any Picat call. The action itself will specify how the state should be changed; we will give some examples later.

3 Background on Picat

Picat is a logic-based multi-paradigm programming language aimed for general-purpose applications. Picat is a rule-based language, in which predicates, functions, and actors are defined with pattern-matching rules. Picat incorporates

many declarative language features for better productivity of software development, including explicit non-determinism, explicit unification, functions, list comprehensions, constraints, and tabling.

In Picat, predicates and functions are defined with pattern-matching rules. Picat has two types of rules: a *non-backtrackable rule* (also called a *commitment rule*) $Head, Cond$ => $Body$, and a backtrackable rule $Head, Cond$?=> $Body$. In a predicate definition, the $Head$ takes the form $p(t_1, \ldots, t_n)$, where p is called the predicate name, and n is called the arity. The condition $Cond$, which is an optional goal, specifies a condition under which the rule is applicable. For a call C, if C matches $Head$ and $Cond$ succeeds, then the rule is said to be *applicable* to C. When applying a rule to call C, Picat rewrites C into $Body$. If the used rule is non-backtrackable, then the rewriting is a commitment, and the program can never backtrack to C. However, if the used rule is backtrackable, then the program will backtrack to C once $Body$ fails, meaning that $Body$ will be rewritten back to C, and the next applicable rule will be tried on C.

Briefly speaking, Picat programming is very similar to Prolog programming. By providing features like functions, list comprehensions etc., Picat programs are even more compact and declarative than equivalent Prolog programs. Moreover, the possibility of explicit non-determinism and unification gives the programmer better control of program execution to make the code even more efficient. More details about the Picat language can be found in the Picat documentation [16].

3.1 Tabling

The Picat system provides a built-in tabling mechanism [21] that simplifies coding of some search algorithms. Tabling is a technique to memorize answers to calls and re-using the answer when the same call appears later. Tabling implicitly prevents loops and brings properties of graph search (not exploring the same state more than once) to classical depth-first search used by Prolog-like languages. Both predicates and functions can be tabled; linear tabling [21] is used in Picat. In order to have all calls and answers of a predicate or a function tabled, users just need to add the keyword `table` before the first rule. For a predicate definition, the keyword `table` can be followed by a tuple of table modes [7], including + (input), - (output), `min`, `max`, and `nt` (not tabled). These modes specify how a particular attribute of the predicate should be handled. For a predicate with a table mode declaration that contains `min` or `max`, Picat tables one optimal answer for each tuple of the input arguments. The last mode can be `nt`, which indicates that the corresponding argument will not be tabled [22]. Ground structured terms are hash-consed [19] so that common ground terms are tabled only once. For example, for three terms `c(1,c(2,c(3)))`, `c(2,c(3))`, and `c(3)`, the shared sub-terms `c(2,c(3))` and `c(3)` are reused from `c(1,c(2,c(3)))`.

Mode-directed tabling has been successfully used to solve specific planning problems such as Sokoban [20], and the Petrobras planning problem [2]. A planning problem is modeled as a path-finding problem over an implicitly specified graph. The following code gives the framework used in all these solutions:

```
table (+,-,min)
path(S,Path,Cost), final(S) => Path = [], Cost = 0.

path(S,Path,Cost) =>
    action(S,S1,Action,ActionCost),
    path(S1,Path1,Cost1),
    Path = [Action|Path1],
    Cost = Cost1+ActionCost.
```

The call `path(S,Path,Cost)` binds `Path` to an optimal path from `S` to a final state. The predicate `final(S)` succeeds if `S` is a final state, and the predicate `action` encodes the set of actions in the problem.

3.2   Resource-Bounded Search

As mentioned in the previous section, the tabling mechanism supports solving optimization problems, such as looking for the shortest path, using the table modes `min` and `max`. When applied to the single-source shortest path problem, linear tabling is similar to Dijkstra's algorithm, except that linear tabling tables shortest paths from the encountered states to the goal state rather than shortest paths to the encountered states from the initial state. When looking for the shortest path from a single initial state to some goal state only, such as in planning, classical tabling may be too greedy as it visits the states that could be farther from the initial state than the length of the shortest path from start to goal. Resource-bounded search is a way to overcome this inefficiency.

Assume that we know the upper bound for the path length, let us call it a resource. Each time, we expand some state, we decrease available resource by the cost of the action used for expansion. Hence less quantity of resource will be available for expansion of the next state (if action costs are positive). The idea of resource-bounded search is to utilize tabled states and their resource limits to effectively decide when a state should be expanded and when a state should fail. Let S^R denote a state with an associated resource limit, R. If R is negative, then S^R immediately fails. If R is non-negative and S has never been encountered before, then S is expanded by using a selected action. Otherwise, if the same state S has failed before and R' was the resource limit when it failed, then S^R is only expanded if $R > R'$, i.e., if the current resource limit is larger than the resource limit was at the time of failure.

4   Planning in Picat

The Picat system has a built-in module `planner` for solving planning problems. The planning problem is described as an abstract state transition diagram and solved using techniques exploiting tabling. By abstraction we mean that states and actions are not grounded, but described in an abstract way similar to modeling operators in PDDL. In this section we briefly introduce the `planner` module, give an example of planning domain model in Picat, and describe available search techniques to solve the planning problems.

4.1   The `planner` Module of Picat

The `planner` module is based on tabling but it abstracts away tabling from users. For a planning problem, users only need to define the predicates `final/1` and `action/4`, and call one of the search predicates in the module on an initial state in order to find a plan or an optimal plan.

- `final(`S`)`: This predicate succeeds if S is a final state.
- `action(`S`,`$NextS$`,`$Action$`,`$ACost$`)`: This predicate encodes the state transition diagram of a planning problem. The state S can be transformed to $NextS$ by performing $Action$. The cost of $Action$ is $ACost$, which must be non-negative. If the plan's length is the only interest, then $ACost = 1$.

These two predicates are called by the planner. The `action` predicate specifies the precondition, effect, and cost of each of the actions. This predicate is normally defined with nondeterministic pattern-matching rules. As in Prolog, the planner tries actions in the order they are specified. When a non-backtrackable rule is applied to a call, the remaining rules will be discarded for the call.

4.2   Modeling Example

To demonstrate how the planning domain is encoded in Picat, we will use the *Transport* domain from IPC'14. Given a weighted directed graph, a set of trucks each of which has a capacity for the number of packages it can carry, and a set of packages each of which has an initial location and a destination, the objective of the problem is to find an optimal plan to transport the packages from their initial locations to their destinations. This problem is more challenging than the *Nomystery* problem that was used in IPC'11, because of the existence of multiple trucks, and because an optimal plan normally requires trucks to cooperate. This problem degenerates into the shortest path problem if there is only one truck and only one package. We introduced the Picat model of this domain in [24], where other examples of domain models are given.

A state is represented by an array of the form {`Trucks`,`Packages`}, where `Trucks` is an ordered list of trucks, and `Packages` is an ordered list of waiting packages. A package in `Packages` is a pair of the form (`Loc`,`Dest`) where `Loc` is the source location and `Dest` is the destination of the package. A truck in `Trucks` is a list of the form [`Loc`,`Dests`,`Cap`], where `Loc` is the current location of the truck, `Dests` is an ordered list of destinations of the loaded packages on the truck, and `Cap` is the capacity of the truck. At any time, the number of loaded packages must not exceed the capacity.

Note that keeping `Cap` as the last element of the list facilitates sharing, since the suffix [`Cap`], which is common to all the trucks that have the same capacity, is tabled only once. Also note that the names of the trucks and the names of packages are not included in the representation. Two packages in the waiting list that have the same source and the same destination are indistinguishable, and as are two packages loaded on the same truck that have the same destination. This

representation breaks object symmetries – two configurations that only differ by a truck's name or a package's name are treated as the same state.

A state is final if all of the packages have been transported.

```
final({Trucks,[]}) =>
    foreach([_Loc,Dests|_] in Trucks)
        Dests == []
    end.
```

The PDDL rules for the actions are straightforwardly translated into Picat as follows.

```
action({Trucks,Packages},NextState,Action,ACost) ?=>
    Action = $load(Loc), ACost = 1,
    select([Loc,Dests,Cap],Trucks,TrucksR),
    length(Dests) < Cap,
    select((Loc,Dest),Packages,PackagesR),
    NewDests = insert_ordered(Dests,Dest),
    NewTrucks = insert_ordered(TrucksR,[Loc,NewDests,Cap]),
    NextState = {NewTrucks,PackagesR},
action({Trucks,Packages},NextState,Action,ACost) ?=>
    Action = $unload(Loc), ACost = 1,
    select([Loc,Dests,Cap],Trucks,TrucksR),
    select(Dest,Dests,DestsR),
    NewTrucks = insert_ordered(TrucksR,[Loc,DestsR,Cap]),
    NewPackages = insert_ordered(Packages,(Loc,Dest)),
    NextState = {NewTrucks,NewPackages}.
action({Trucks,Packages},NextState,Action,ACost) =>
    Action = $move(Loc,NextLoc),
    select([Loc|Tail],Trucks,TrucksR),
    road(Loc,NextLoc,ACost),
    NewTrucks = insert_ordered(TrucksR,[NextLoc|Tail]),
    NextState = {NewTrucks,Packages}.
```

For the *load* action, the rule nondeterministically selects a truck that still has room for another package, and nondeterministically selects a package that has the same location as the truck. After loading the package to the truck, the rule inserts the package's destination into the list of loaded packages of the truck. Note that the rule is nondeterministic. Even if a truck passes by a location that has a waiting package, the truck may not pick it. If this rule is made deterministic, then the optimality of plans is no longer guaranteed, unless there is only one truck and the truck's capacity is infinite.

The above model is very similar to the PDDL encoding available at IPC web pages [8]. The major difference is the model of states that is a structure consisting of two ordered lists. The ordering is used to obtain a unique representation of states. The encoding can be further extended by adding control knowledge, for example the predicate `action` can begin with a rule that deterministically unloads a package if the package's destination is the same as the truck's location. To exploit better the resource-bound search, one can also add heuristics to action definition. The heuristic can estimate the cost-to-goal and it can be added to actions through the following condition:

```
current_resource() - ACost >=  estimated_cost(NewState).
```

The `current_resource()` is a built-in function of the planner giving the maximal allowed cost-distance to the goal. Note that heuristic is a part of the domain model so it is domain dependent.

We discussed some domain modeling principles in [3]. Basically, the Picat `planner` module supports:

- *structured state representation* that is more compact than the factored representation and allows removing symmetry between objects by representing objects via their properties rather than via their names (see representation of trucks and packages in the *Transport* domain),
- *control knowledge* that guides the planner via ordering of actions in the model and using extra conditions to specify when actions are applicable (for example, always unload the package when the truck is at the package destination),
- *action symmetry breaking* by modeling possible action sequences via a non-deterministic finite state automaton (for example, load the truck and move it somewhere for further loading or unloading before assuming actions of another truck),
- *heuristics* that estimate the cost-to-goal and can be domain dependent (domain independent heuristics can be used as well).

4.3 Search Techniques

The planning-domain model is specified as a set of Picat rules that are explored by the Picat planner. This planner uses basically two search approaches to find optimal plans. Both of them are based on depth-first search with tabling and in some sense they correspond to classical forward planning. It means that they start in the initial state, select an action rule that is applicable to the current state, apply the rule to generate the next state, and continue until they find a state satisfying the goal condition (or the resource limit is exceeded).

The first approach starts with finding any plan using the depth first search. The initial limit for plan cost can (optionally) be imposed. Then the planner tries to find a plan with smaller cost so a stricter cost limit is imposed. This process is repeated until no plan is found so the last plan found is an optimal plan. This approach is very close to *branch-and-bound* technique [12]. Note that tabling is used there – the underlying solver remembers the best plans found for all visited states so when visiting the state next time, the plan from it can be reused rather than looked for again. This planning algorithm is evoked using the following call:

```
best_plan_bb(+InitState,+CostLimit,-Plan,-PlanCost)
```

This is where the user specifies the initial state and (optionally) the initial cost limit. The algorithm returns a cost-optimal plan and its cost. This approach can be also used to find the first plan using the call `plan(+S,+L,-P,-C)`.

Despite using tabling that prevents re-opening the same state, this approach still requires good control knowledge to find the initial plan (otherwise, it may be lost in a huge state space) or alternatively some good initial cost limit should be used to prevent exploring long plans.

The second approach exploits the idea of iteratively extending the plan length as proposed first for SAT-based planners [9]. It first tries to find a plan with cost zero. If no plan is found, then it increases the cost by 1. In this way, the first plan that is found is guaranteed to be optimal. Unlike the *IDA* search algorithm* [10], which starts a new round from scratch, Picat reuses the states that were tabled in the previous rounds. This planning algorithm is evoked using the call:

```
best_plan(+InitState,+CostLimit,-Plan,-PlanCost)
```

This approach is more robust with respect to weak or no control knowledge, but it has the disadvantage that it can only find the optimal plan, which could be more time consuming than finding any plan.

Note that the cost limit in the above calls is used to define the function `current_resource()` mentioned in the action rules. Briefly speaking the cost of the partial plan is subtracted from the cost limit to get the value of the function `current_resource()` that can be utilized to compare with the heuristic distance to the goal.

5   Experimental Comparison

The Picat planner uses a different approach to planning so it is important to show how this approach compares with current state-of-the-art planning techniques and to understand better the Picat search procedures. In [24] we compared the Picat planer with SymBA [18] – the domain-independent bidirectional A* planner which won the optimal sequential track of IPC'14. As the Picat planner can exploit domain-dependent information, in [3] we compared the Picat planner with leading domain-dependent planners based on control rules and hierarchical task networks (HTN). We will summarize these results first and then we will present a new experimental study comparing the search approaches in Picat.

5.1   Comparison to Automated Planners

Optimal Domain Independent Planners. We have encoded in Picat most domains used in the deterministic sequential track of IPC'14. All of the encodings are available at: `picat-lang.org/ipc14/`. The Picat planner was using the iterative deepening `best_plan/4` planning algorithm. We have compared these Picat encodings with the IPC'14 PDDL encodings solved with SymBA. Table 1 shows the number of instances (#insts) in the domains used in IPC'14 and the number of (optimally) solved instances by each planner. The results were obtained on a Cygwin notebook computer with 2.4GHz Intel i5 and 4GB RAM. Both Picat and SymBA were compiled using g++ version 4.8.3. For SymBA, a

setting suggested by one of SymBA's developers was used. A time limit of 30 minutes was used for each instance as in IPC. For every instance solved by both planners, the plan quality is the same. The running times of the instances are not given, but the total runs for Picat were finished within 24 hours, while the total runs for SymBA took more than 72 hours.

Table 1. The number of problems solved optimally.

Domain	# insts	Picat	SymBA
Barman	14	**14**	6
Cave Diving	20	**20**	3
Childsnack	20	**20**	3
Citycar	20	**20**	17
Floortile	20	**20**	**20**
GED	20	**20**	19
Parking	20	**11**	1
Tetris	17	**13**	10
Transport	20	**10**	8
Total	171	**148**	87

Domain Dependent Planners. We took the following domains: *Depots, Zeno-travel, Driverlog, Satellite,* and *Rovers* from IPC'02. The Picat encodings are available at: `picat-lang.org/aips02/`. We compared Picat with TLPlan [1], the best hand-coded planner of IPC'02, TALPlanner [11] another good planner based on control rules, and SHOP2 [14], the distinguished hand-coded planner of IPC'02 using HTN. Each of these planners used its own encoding of planning domains developed by the authors of the planners.

All planners found (possibly sub-optimal) plans for all benchmark problems and the runtime to generate plans was negligible; every planner found a plan in a matter of milliseconds. Hence we focused on comparing the quality of obtained plans that is measured by a so called *quality score* introduced in IPC. Briefly speaking the score for solving one problem is 1, if the planner finds the best plan among all planners; otherwise the score goes down proportionally to the quality of the best plan found. The higher quality score means an overall better system.

For TLPlan, TALPlanner, and SHOP2 we took the best plans reported in the results of IPC'02. Taking in account the nature of planners and their runtimes, there is a little hope to get better plans when running on the current hardware. For the Picat planner we used the branch-and-bound `best_plan_bb/4` planning algorithm. Table 2 shows the quality scores when we gave five minutes to the Picat planner to improve the plan (running under MacOS X 10.10 on 1.7 GHz Intel Core i7 with 8 GB RAM).

The results show that the Picat planner is competitive with other domain-dependent planners and that it can even find better plans. In [3] we also demon-

Table 2. Comparison of quality scores for the best plan (5 minutes)

Domain	# insts	Picat	TLPlan	TALPlanner	SHOP2
Depots	22	**21.94**	19.93	20.52	18.63
Zenotravel	20	**19.86**	18.40	18.79	17.14
Driverlog	20	17.21	17.68	**17.87**	14.16
Satellite	20	**20.00**	18.33	16.58	17.16
Rovers	20	**20.00**	17.67	14.61	17.57
Total	102	**99.01**	92.00	88.37	84.65

strated that the Picat domain models are much smaller than domain models using control rules and are much closer in size to the PDDL models.

5.2 Comparison of Search Techniques

In the second experiment we focused on comparing two search approaches to find cost-optimal plans in Picat, namely branch-and-bound and iterative deepening. When looking for optimal plans, the hypothesis is that iterative deepening requires less memory and time because branch-and-bound explores longer plans and hence may visit more states. On the other hand, the advantage of branch-and-bound is that it can find some plan even if finding (and proving) optimal plan is hard (recall, that iterative deepening returns only optimal plans). So the second hypothesis is that when looking for any plan, branch-and-bound could be a better planning approach. Nevertheless, due to depth-first-search nature, branch-and-bound requires good control knowledge to find an initial plan. The final hypothesis is that if none or weak control knowledge is part of the domain model then iterative deepening is a more reliable planning approach.

We used the following domains from the deterministic sequential track of IPC'14 [8]: *Barman*, *Cavediving*, *Childsnack*, *Citycar*, *Floortile*, *GED*, *Parking*, *Tetris*, and *Transport*. All of the encodings are available at: `picat-lang.org/ipc14/`. The experiment run on Intel Core i5 (Broadwell) 5300U(2.3/2.9GHz) with 4 GB RAM (DDR3 1600 MHz). For each problem, we used timeout of 30 minutes and memory limit 1 GB. We compared the following search procedures:

- `plan(`*InitState*`,`*CostLimit*`,`*Plan*`,`*PlanCost*`)`,
- `best_plan(`*InitState*`,`*CostLimit*`,`*Plan*`,`*PlanCost*`)`,
- `best_plan_bb(`*InitState*`,`*CostLimit*`,`*Plan*`,`*PlanCost*`)`,

using $99,999,999$ as the initial cost limit ($10,000$ for the *GED* domain).

We first report the number of solved problems with respect to time and memory consumed. Note that `best_plan/4` and `best_plan_bb/4` return cost-optimal plans while `plan/4` returns some (possibly sub-optimal) plan. Figure 1 shows the number of solved problems within a given time. Figure 2 shows the number of solved problems based on memory consumed.

The results confirm the first and second hypotheses, that is, iterative deepening requires less time and less memory than branch-and-bound when solving
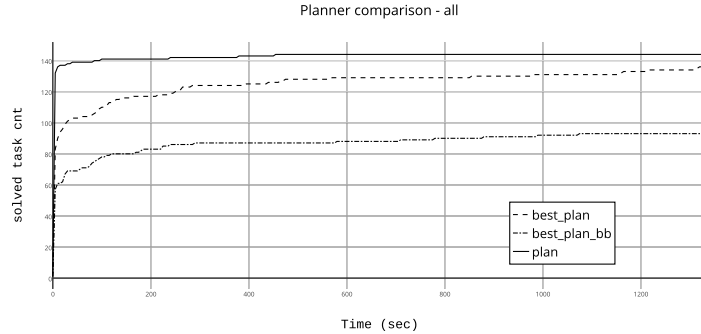
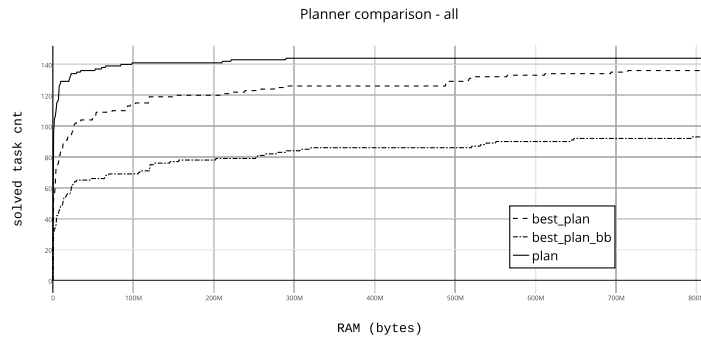**Fig. 1.** The number of solved problems within a given time.

Fig. 2. The number of solved problems dependent on memory consumption.

problems optimally, but branch-and-bound has the advantage of providing some (possibly sub-optimal) plan fast. If looking for any plan then branch-and-bound also requires less memory.

Describing dependence of planner efficiency on the model is more tricky as it is hard to measure model quality quantitatively. We annotated each involved domain model by information about using control knowledge and domain-dependent heuristics in the model. Table 3 shows the annotation of domain models based on these two criteria.

Based on Table 3 we can classify the Picat domain models into following groups:

– The Picat domain model for *Barman* is probably closest to the PDDL encoding; it only uses the structured representation of states, which alone seems to be advantage over PDDL as Table 1 shows. *GED* uses a bit specific model based on a PDDL model different from that one used in the IPC – this model uses some macro-actions – and hence it is not really tuned for Picat.

Table 3. The properties of domain models.

Domain	control knowledge	heuristics
Barman	no	no
Cave Diving	strong	no
Childsnack	strong	no
Citycar	no	yes
Floortile	strong	no
GED	macro	yes
Parking	weak	yes
Tetris	no	yes
Transport	weak	yes

– *Citycar* and *Tetris* are domains where useful admissible heuristics are used, but no control knowledge is implemented to guide the planner.
– The Picat domain models for *Parking* and *Transport* use some weak control knowledge in the form of making selection of some actions deterministic (see the example earlier in the paper). They also exploit admissible heuristics.
– *Cave Diving*, *Childsnack*, and *Floortile* are domains, where we use strong control knowledge and no heuristics. Control knowledge is used there to describe reasonable sequencing of actions either via finite state automata or macro-actions. The domain model for *Cave Diving* is described in detail in [3]; the domain model for *Childsnack* is almost deterministic as this problem does not require real planning; and the domain model for *Floortile* uses macro-actions to force reasonable action sequences, see [24] for details.

From each class of domain models we selected one representative to demonstrate how different solving approaches behave (the other domains gave similar results). Figure 3 shows the number of solved problems for these representatives. If the Picat domain model is very close to the original PDDL model, then iterative deepening has a clear advantage when finding optimal plans, see the *Barman* domain. This corresponds to popularity of this solving approach in planners based on SAT techniques [9]. In case of *Barman* the branch-and-bound approach can still find some plans as the model itself guides the planner reasonably well (there are no extremely long plans). However, for the *GED* domain, only iterative deepening can find (optimal) plans while branch-and-bound was not able to find any plan due to being lost in generating extremely long plans not leading to the goal.

Adding admissible heuristics makes iterative deepening even more successful, see the *Tetris* domain. Finding optimal plans by iterative deepening is close to finding any plan by branch-and-bound. Also the gap between finding any plan and finding an optimal plan by branch-and-bound is narrower there. Obviously, this also depends on the quality of first plan found.

An interesting though not surprising observation is that adding even weak control knowledge makes finding any plan by branch-and-bound much more successful and decreases further the gap between iterative deepening and branch-
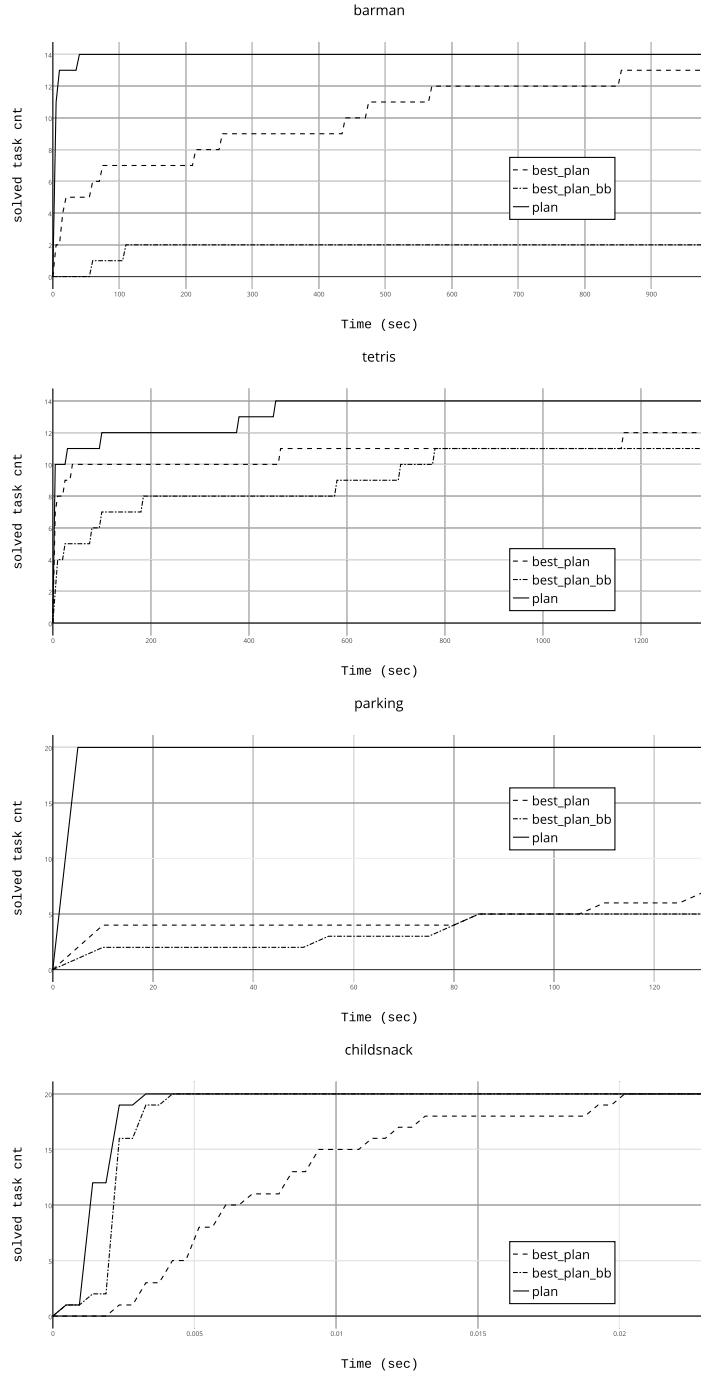
Fig. 3. The number of solved problems within a given time for specific domains.

and-bound when looking for optimal plans, see the *Parking* domain. The role of control knowledge is even more highlighted in the *Childsnack* domain, which shows that strong control knowledge has a big influence on efficiency of branch-and-bound. Longer runtimes of iterative deepening are caused by exploring short plans that cannot solve the problem before discovering the necessary length of the plan to reach the goal. Still control knowledge helps iterative deepening to find a larger number of optimal plans though it takes longer than for branch-and-bound.

The experimental results justify the role of control knowledge for solving planning problems and confirm the last hypothesis that control knowledge is important for the branch-and-bound approach especially if the dead-ends can be discovered only in very long plans.

6 Summary

This paper puts in contrast two approaches for searching for sequential plans, iterative deepening used in [24] and branch-and-bound used in [3]. We demonstrated that the modeling framework proposed for the Picat `planner` module is competitive with state-of-the-art planning approaches and we showed some relations between the modeling techniques and used search algorithms. In particular, we demonstrated the role of control knowledge in planning and we showed that control knowledge is more important for branch-and-bound though it also contributes to efficiency of iterative deepening. The role of heuristics is known in planning as for a long time heuristic-based forward planners are the leading academic planners. Note however that Picat is using heuristics in a different way. Rather than guiding the planner to promising areas of the search space, the heuristics are used to cut-off sub-optimal plans earlier. Hence the role of heuristics is stronger for iterative deepening than for branch-and-bound.

This paper showed some preliminary results on the relations between various modeling and solving techniques for planning problems. The next step is a deeper study of influence of various modeling techniques on efficiency of planning.

References

1. Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.
2. Roman Barták and Neng-Fa Zhou. Using tabled logic programming to solve the Petrobras planning problem. *Theory and Practice of Logic Programming*, 14(4-5):697–710, 2014.
3. Roman Barták, Agostino Dovier, Neng-Fa Zhou. On modeling planning problems in tabled logic programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming – PPDP'15*, 32–42, 2015.

4. Patrik Haslum and Ulrich Scholz. Domain knowledge in planning: Representation and use. In *ICAPS Workshop on PDDL*, 2003.
5. Carl Hewitt. Planner: A language for proving theorems in robots. In *Proceedings of IJCAI*, 295–302, 1969.
6. Richard E. Fikes and Nils J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2 (3-4):189–208, 1971
7. Hai-Feng Guo and Gopal Gupta. Simplifying dynamic programming via mode-directed tabling. *Software: Practice and Experience*, 38(1):75–94, 2008.
8. International Planning Competitions web site, `http://ipc.icaps-conference.org/`, Accessed April 5, 2015.
9. Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of ECAI*, 359–363, 1992.
10. Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
11. Jonas Kvarnström and Martin Magnusson. Talplanner in the third international planning competition: Extensions and control rules. *J. Artificial Intelligence Research* (JAIR), 20:343–377, 2003.
12. A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica* 28(3):497–520, 1960.
13. Drew McDermott. The planning domain definition language manual. CVC Report 98-003, Yale Computer Science Report 1165, 1998.
14. Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. SHOP2: an HTN planning system. *J. Artificial Intelligence Research* (JAIR), 20:379–404, 2003.
15. Nils J. Nilsson. Shakey The Robot, Technical Note 323. AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Apr 1984.
16. Picat web site, `http://picat-lang.org/`, Accessed July 3, 2015.
17. TLPlan web site, `http://www.cs.toronto.edu/tlplan/`, Accessed April 5, 2015.
18. Alvaro Torralba, Vidal Alcazar, and Daniel Borrajo. Symba: A symbolic bidirectional a planner. In *The 2014 International Planning Competition*,105–109, 2014.
19. Neng-Fa Zhou and Christian Theil Have. Efficient tabling of structured data with enhanced hash-consing. *Theory and Practice of Logic Programming*, 12(4-5):547–563, 2012.
20. Neng-Fa Zhou and Agostino Dovier. A tabled Prolog program for solving Sokoban. *Fundamenta Informaticae*, 124(4):561–575, 2013.
21. Neng-Fa Zhou, T. Sato, and Y.-D. Shen. Linear tabling strategies and optimizations. *Theory and Practice of Logic Programming*, 8(1):81–109, 2008.
22. Neng-Fa Zhou, Y. Kameya, and T. Sato. Mode-directed tabling for dynamic programming, machine learning, and constraint solving. In *Proceedings of 22nd IEEE International Conference on Tools with Artificial Intelligence* (ICTAI), 213–218, 2010.
23. Neng-Fa Zhou. Combinatorial Search With Picat. *http://arxiv.org/abs/1405.2538*, 2014.
24. Neng-Fa Zhou, Roman Barták, Agostino Dovier. Planning as Tabled Logic Programming. To appear in *Theory and Practice of Logic Programming*, 2015.