Planning as tabled logic programming

NENG-FA ZHOU

CUNY Brooklyn College and Graduate Center

ROMAN BARTÁK

Charles University

AGOSTINO DOVIER

Univ. di Udine

Abstract

This paper describes Picat's planner, its implementation, and planning models for several domains used in International Planning Competition (IPC) 2014. Picat's planner is implemented by use of tabling. During search, every state encountered is tabled, and tabled states are used to effectively perform resource-bounded search. In Picat, structured data can be used to avoid enumerating all possible permutations of objects, and term sharing is used to avoid duplication of common state data. This paper presents several modeling techniques through the example models, ranging from designing state representations to facilitate data sharing and symmetry breaking, encoding actions with operations for efficient precondition checking and state updating, to incorporating domain knowledge and heuristics. Broadly, this paper demonstrates the effectiveness of tabled logic programming for planning, and argues the importance of modeling despite recent significant progress in domain-independent PDDL planners.

1 Introduction

Planning and logic programming are two close areas of research. PLANNER (Hewitt 1969), which was designed as *a language for proving theorems and manipulating models in a robot*, is perceived as the first logic programming language. Planning has been an important problem domain for Prolog (Kowalski 1979; Warren 1974). Despite the amenability of Prolog to planning, Prolog is no longer a competitive tool for planning.

Tabling (Michie 1968; Tamaki and Sato 1986; Warren 1992) is a technique used in logic and functional programming systems, which caches the results of certain calculations in memory and reuses them in subsequent calculations through a quick table lookup. Like state marking used in search algorithms, tabling can prevent the same state from being expanded more than once during search. Tabling has been found useful in many search problems, including theorem proving (Nielson *et al.* 2004; Pientka 2003), program analysis (Dawson *et al.* 1996) and model checking (Ramakrishna *et al.* 1997). Recently, tabled logic programming has been successfully

employed to solve specific planning problems (Bartak and Zhou 2014; Zhou and Dovier 2013; Zhou 2014), and has been shown to be significantly faster than the state-of-the-art ASP (Answer Set Programming) planners on some problems (Zhou and Dovier 2013; Zhou 2014).

This paper describes Picat's planner and its implementation. This paper also presents planning models in Picat for several domains used in International Planning Competition 2014 (IPC'14) (Chrpa *et al.* 2014), and demonstrates broader applicability of tabled logic programming to planning. Picat is a logic-based multi-paradigm language that provides logic variables, pattern matching, nondeterminism through backtracking, loops, functions, constraints, and tabling as its core modeling and solving features. As a modeling language for planning, Picat differs from PDDL (Plan Domain Description Language) (McDermott 1998) and ASP (Brewka *et al.* 2011; Gebser *et al.* 2012; Lifschitz 2002) in several aspects: (1) Picat allows use of structures to represent states; (2) Picat supports explicit commitment and nondeterministic actions, which enables users to have better control over action applications; (3) Picat provides facilities for describing domain knowledge and heuristics for pruning search space.

As a solving system, Picat's planner implements several techniques for better performance. First, it tables every state encountered during search and avoids repeating the exploration of the same state. Second, it adopts the hash-consing technique (Zhou and Have 2012) to share common state data and to speed up the equality testing of states. Third, it utilizes tabled states to effectively perform *resource-bounded* search. For optimal planning, Picat offers built-ins to perform iterative search, but unlike IDA* (Korf 1985), Picat also reuses results tabled in early iterations (Zhou 2014).

This paper shows that the above-mentioned features of Picat make Picat a more appropriate language than PDDL for modeling and solving planning problems. To that end, this paper presents examples in Picat for several domains used in IPC'14. These examples illustrate several modeling techniques on how to design state representations to facilitate data sharing and symmetry breaking, on how to translate PDDL operators into Picat actions, and on how to incorporate domain knowledge and heuristics to reduce search spaces. This paper also gives the experimental results of the presented models and several other models encoded in the same way. The experimental results demonstrate the effectiveness of tabling and the importance of modeling.

2 A brief overview of picat

Picat is a dynamically-typed language. The basic types are taken from Prolog, except for *arrays* and *maps*. An array takes the form $\{t_1, \ldots, t_n\}$. The index of the first array element is 1, and the index notation X[I] can be used to access array elements. Picat also borrows the basic logical operators from Prolog, including *conjunction* (A, B), *negation* (not A), *disjunction* $(A; B)$, and *if-then-else* $(C\text{->}A; B)$.

Picat allows function calls in arguments. For this reason, it requires structures to be preceded with a dollar symbol $ in order for them to be treated as data, unless the structure is special, or it occurs in a head pattern.

For each type, Picat provides a set of built-in functions and predicates. Many built-in predicates are taken from Prolog, including `member/2`, `nth/3`, and `select/3`. The function `insert_ordered`(*List*, *Term*) inserts *Term* into the ordered list *List* such that the resulting list remains ordered.

In Picat, predicates and functions are defined with pattern-matching rules. Picat has two types of rules: the *non-backtrackable rule Head*, *Cond* => *Body*, and the backtrackable rule *Head*, *Cond* ?=> *Body*. In a predicate definition, the *Head* takes the form $p(t_1, \ldots, t_n)$, where p is a predicate name, and n is 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. In a function definition, the *Head* takes the form $f(t_1, \ldots, t_n) = Term$ where f is a function name and *Term* is a result to be returned. All of the rules in a function definition must be non-backtrackable.

A pattern can contain *as-patterns* of the form $V@Pattern$, where V is a new variable in the head, and *Pattern* is a non-variable term. The as-pattern $V@Pattern$ is the same as *Pattern* in pattern matching, but after pattern matching succeeds, V is made to reference the term that matched *Pattern*.

Picat supports loops and list comprehensions. For example, the loop

```
foreach(E in L) Goal end
```

is true if `Goal` is true for each `E` in `L`. Picat adopts the following simple scoping rule: *variables that occur only in a loop, but do not occur before the loop in the outer scope, are local to each iteration of the loop.* Loops are compiled into tail-recursive predicates, and list comprehensions are compiled into tail-recursive predicates through `foreach` loops.

Picat supports tabling for dynamic programming solutions. Other features of Picat include assignments, list comprehensions, global maps for storing permanent data, higher-order functions, action rules for defining event-driven actors, and modules for modeling and solving constraint satisfaction problems with CP, SAT, and MIP.

3 Tabling in picat

Both predicates and functions can be tabled. In order to have all calls and answers of a predicate or 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, including + (input), − (output), min, max, and `nt` (not tabled). For a predicate with a table mode declaration that contains `min` or `max`, Picat tables
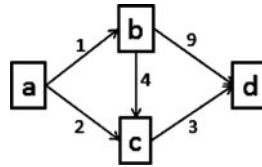
Fig. 1. A DAG.

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.

Linear tabling (Zhou *et al.* 2008) is used in Picat, and table modes are taken from (Guo and Gupta 2008), except for the `nt` mode, which was initially proposed by (Zhou *et al.* 2010). Ground structured terms are hash-consed (Zhou and Have 2012) so that common ground terms are tabled only once. For example, for the three lists `[1,2,3]`, `[2,3]`, and `[3]`, the shared sub-lists `[2,3]` and `[3]` are reused from `[1,2,3]`.

Mode-directed tabling has been successfully used to solve specific planning problems such as Sokoban (Zhou and Dovier 2013), and the Petrobras planning problem (Bartak and Zhou 2014). A planning problem is modeled as a path-finding problem over an implicitly specified graph. The following 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.

Consider using the `path/3` predicate to find a shortest path from node `a` to node `d` in the DAG shown in Figure 1. The final state is `d`. The call `path(a,Path,Cost)` initiates the search from node `a`. In order to resolve the call, Picat applies the transition a→b to node `a`, and generates a new call to `path` for finding a shortest path from `b` to `d`. After the answer b→c→d is found for the call, Picat tables the answer a→b→c→d, which has cost 8, for the inital call. After that, Picat backtracks, trying the alternative transition a→c to node `a`. After the shortest path c→d is found, Picat finds another path a→c→d, which has cost 5, for the initial call. Since this path is shorter than the tabled path, Picat replaces the tabled path with this new path. Since the graph has no cycles, Picat returns the new path as the final answer. In general, calls need to be re-evaluated until fixed points are reached if there are looping calls (Zhou *et al.* 2008).

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.

The above framework performs depth-unbounded search. For many planning problems, branch & bound and IDA* (Korf 1985) are useful for finding optimal solutions. The `planner` module of Picat provides built-ins for planning with different types of search.

4 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.

The following predicates constitute the core of the `planner` module.

- `best_plan_unbounded(S, Limit, Plan, PlanCost)`: This predicate finds an optimal plan by performing *depth-unbounded* search. This predicate is implemented based on the path-finding framework shown above. The argument *Limit* is not utilized to limit the depth of search. It is compared with $PlanCost$ after an optimal plan has been found. During depth-unbounded search, once a state has failed, it will not be explored again.
- `plan(S, Limit, Plan, PlanCost)`: This predicate searches for a plan by performing *resource-bounded* search, in which a state is expanded only if it is new and its resource limit is non-negative, or if the state has previously failed but the current occurrence has a higher resource limit than before. This predicate is defined as a tabled predicate that tables the *Limit* argument but does not use it in variant checking (Zhou 2014). The implementation of this predicate is described in the next subsection.
- `best_plan(S, Limit, Plan, PlanCost)`: This predicate finds an optimal plan by performing resource-bounded *iterative-deepening* search. It calls the `plan/4` predicate to find a plan, using 0 as the initial cost limit and gradually relaxing the cost limit until a plan is found. Unlike IDA*, which starts a new round from scratch, Picat also reuses the states that were tabled in the previous rounds.
- `best_plan_bb(S, Limit, Plan, Cost)` This predicate finds an optimal plan using branch & bound. First, it calls `plan/4` to find a plan. Then, it tries to find a

```
plan(S,Limit,Plan,PlanCost) =>
    IPlan = {Limit,[],0},
    catch(plan_bounded_aux(S,IPlan), (Plan,PlanCost), true).

table (+,nt)
plan_bounded_aux(S,{Limit,IPlan,IPlanCost}),
    final(S)
=>
    throw((IPlan.reverse(), IPlanCost)).
plan_bounded_aux(S,{Limit,IPlan,IPlanCost}) =>
    action(S,NextS,Action,ACost),
    Limit1 = Limit-ACost,
    Limit1 >= 0,
    Inherited1 = {Limit1,
                  [Action|IPlan],
                  IPlanCost+ACost},
    plan_bounded_aux(NextS,Inherited1).
```

Fig. 2. The implementation of `plan/4`.

better plan by imposing a stricter limit. This step is repeated until no better plan can be found. It returns the last plan that was found.

- `current_resource()`: This function returns the resource limit argument of the latest call to `plan/4`. In order to retrieve the argument, the implementation has to traverse the call-stack until it reaches a call to `plan/4`. This function can be used to check against a heuristic value. If the heuristic estimate of the cost to travel from the current state to a final state is greater than the resource limit, then the current state should fail.

5 The implementation of resource-bounded search

Figure 2 sketches Picat's implementation of the predicate `plan/4`. The following array is passed from a state to the next state as an `nt` argument:

```
{Limit,IPlan,IPlanCost}
```

In order to perform resource-bounded search, Picat treats the second argument of the predicate `plan_bounded_aux` differently from other `nt` arguments. Picat stores the `Limit` argument of each failed call to `plan_bounded_aux`, and uses this information to decide whether the same state should fail when it recurs.

The first rule of `plan_bounded_aux` throws the inherited plan and its cost as an exception if `final(S)` succeeds. The exception will be caught by the `catch` call in the rule body of `plan/4`.

The second rule of `plan_bounded_aux` calls `action/4` to select an action, which produces a new state, `NextS`. Then, the rule computes the new resource limit, `Limit1`, by subtracting the cost of the selected action from `Limit`. If `Limit1 >= 0` succeeds, then the rule continues with the tabled search by recursively calling `plan_bounded_aux` on the new state `NextS`. Otherwise, if `Limit1 >= 0` fails, then Picat backtracks to select an alternative action.

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
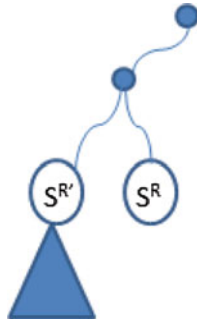
Fig. 3. Resource-bounded search.

fail. Let S^R denote a state with an associated resource limit, R, as depicted in Figure 3. 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.

6 Modeling examples

This section describes the Picat models for four of the problem domains used in the sequential optimal track of IPC'14: *Transport*, *Tetris*, *Floortile*, and *Parking*. Each of the following models will show the state representation, the encoding of the actions, the search predicate that is used (best_plan or best_plan_unbounded), and the domain knowledge and heuristics that are employed. In these models, states are typically represented by lists, and preconditions and state updates are handled by standard list operations. Sometimes, arrays are used when no list suffixes can be shared. The Picat encodings of actions are mostly straightforward translations from the PDDL encodings. These models do not use sophisticated domain knowledge or heuristics that would hurt the readability or compromise the optimality of answers. These four domains, as well as five other domains, are described in Appendix A.

6.1 Example-1: Transport

This problem is a variant of the popular logistics domain in planning. 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. The PDDL and Picat encodings of the problem are given in Appendix B.

Basic encoding

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 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.

Domain knowledge and heuristics

Domain knowledge can be used to reduce the nondeterminism and avoid unnecessary applications of actions. In the complete Picat encoding given in Appendix B, the predicate `action` begins with a rule that deterministically unloads a package if the package's destination is the same as the truck's location.

Resource-bounded search is used to find an optimal plan. After each new state is generated, the following condition is checked to ensure that the current path is viable.

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

Let $P_1, \ldots, P_n$ be the remaining packages, and C_i be the minimum cost of moving package P_i to its destination by using any truck. The moving cost of a state can be safely estimated as $\max(\{C_1, \ldots, C_j\})$. The estimated total cost is the estimated moving cost plus the loading and unloading costs of all of the remaining packages. This heuristic is admissible.

6.2 Example-2: Tetris

The problem is a simplified version of the well-known Tetris. There are three kinds of pieces: 1-cell *boxes*, 2-cell *rectangles*, and 3-cell *L-shaped* pieces. Initially, all the pieces are distributed on a grid board. The pieces can freely move on the board to any cells as long as the cells are not occupied by other pieces. There is a region on the board that is designated as the target region. The goal of the game is to move all the pieces to the target region. In the IPC setting, rectangles can rotate but L-shaped pieces cannot.

Basic encoding

Pieces can be represented as follows: A square is represented by the cell it occupies; a rectangle by a term of the form `rect(C1,C2)` where C1 and C2 are cell locations; and an L-shaped piece by a term of the form `ell(C1,C2,C3)`. In the IPC setting, the ordering of the cells that are occupied by a piece is important. So `rect(C1,C2)` and `rect(C2,C1)` represent two different rectangle pieces. A state is represented by an array of the form {Squares,Rects,Ls}, where each argument gives a sorted list of a single kind of pieces.

The PDDL rules for actions can be translated into Picat in a straightforward manner. For example, the following rule selects a rectangle piece to move.

```
action(State@{Squares,Rects,Ls},NewState,Action,ACost) ?=>
    Action = $move(Rect,NewRect), ACost = 1,
    select(Rect,Rects,RectsR),
    Rect = $rect(C1,C2),
    connected(C2,C3),
    not_occupied(C3,State),          %%  C3 is free
    NewRect = $rect(C2,C3),
    NewState = {Squares,insert_ordered(Rects,NewRect),Ls}.
```

The predicate `not_occupied(C3,State)` is true if C3 is not occupied by any of the pieces in `State`.

Heuristics

Resource-bounded search is used to find an optimal plan for this problem. The goal of the problem is to move all the pieces to the target region. For each piece, assume that it's the only piece on the board, disregarding all other pieces. The estimated cost of moving the piece to the target region is its shortest distance to the nearest cell in the target region times the cost of each move. The estimated cost of transforming a state into a final state is the sum of the estimated costs of all the pieces. This heuristic function is admissible and easy to compute, but very conservative because a cell in the target region can serve as a target for multiple pieces.

6.3 Example-3: Floortile

A set of robots use two different colors (black and white) to paint patterns in floor tiles. The robots can move around the floor tiles in four directions (up, down, left and right). Robots paint with one color at a time, but can change their spray guns to the other color. However, robots can only paint the tile that is in front (up) and behind (down) them, and once a tile has been painted no robot can stand on it.

Basic encoding

A state is represented by a list of the form `[Robots,WTiles,BTiles]`, where `Robots` is an ordered list of robots and `WTitles` (`BTiles`) is an ordered list of locations of the painted white (black) tiles. A robot in `Robots` is a pair `(Color,Loc)` where `Color` is the color of the paint that the robot is holding and `Loc` is the robot's location.

A state is final if all the tiles that are required to be painted are all painted.

```
final({_,WTiles,BTiles}) =>
    painted_w_tiles_in_goal(WTiles),
    painted_b_tiles_in_goal(BTiles).
```

The actions are encoded according to the state representation. For example, once a tile is painted, the tile's location is added into `WTiles` or `BTiles` depending on the color of the paint; once a robot `(Color,Loc)` moves from `Loc` to `Loc1`, the pair is changed to `(Color,Loc1)`. The current graph is determined by the initial graph,

the set of painted tiles, and the set of robots. A robot can move from its current location `Loc` to a new location `NextLoc` if `NextLoc` is connected to `Loc` in the graph, `NextLoc` is not painted, and `NextLoc` is not occupied by another robot.

Macro actions and domain knowledge

The color-changing action, if needed, can be forced to take place right before a painting action. This macro action helps reduce the search space.

For IPC'14, the used instances were generated in the fashion that *robots should only paint tiles in front of them*. This special condition can be exploited to reduce the nondeterminism of the painting rules: if a robot is at location `Loc`, trying to paint the up location `ULoc`, and the up location of `ULoc` has already been painted, then `ULoc` can be painted deterministically; similarly, if a robot is at location `Loc`, trying to paint the down location `DLoc`, and the down location of `DLoc` has already been painted, then `DLoc` can be painted deterministically.

Depth-unbounded search is used for this problem. During depth-unbounded search, dead ends are tabled and are not re-explored when they are encountered again. For this problem, a state becomes a dead end if there is an unpainted tile that is not reachable by any robot. No extra code is needed to detect dead ends. Because depth-unbounded search is used, no heuristics are needed.

6.4 Example-4: Parking

This domain involves parking cars on a street with N curb locations, and where cars can be double-parked but not triple-parked. The goal is to find a plan to move from one configuration of parked cars to another configuration, by driving cars from one curb location to another. For each curb location, there are two parking spots: the curb side and the road side. For a car to move from a spot at a curb into a spot at a different curb, the destination spot must be clear; and if the spot is on the curb side, then the road-side spot must also be clear. In some ways, this problem is similar to the *Blocks World* and the *Tower of Hanoi*.

Basic encoding

A configuration is represented by an array of curbs $\{B_1, B_2, \ldots, B_n\}$, where each B_i is a list of up to two cars. When a curb has two cars $[C_1, C_2]$, it is assumed that C_1 is the road-side car and C_2 is the curb-side car. This representation allows the road-side car C_1 to be removed without touching the curb-side car C_2.

A state is represented by a pair of the form (`CurrConfig,GoalConfig`), where a `CurrConfig` is the current configuration, and `GoalConfig` is the goal configuration. As shown below, the inclusion of the goal configuration in the state representation facilitates the encoding of domain knowledge.

A state is final if the current configuration is the same as the goal configuration.

```
final((Config,Config)) => true.
```

All possible moves are specified with one rule as follows:

```
action((Config,GConfig),NextS,Action,ACost) =>
    Action = $move(I,J), ACost = 1,
    nth(I,Config,ICars),ICars = [ClearCar|NewICars],
    nth(J,Config,JCars), J !== I, JCars != [_,_],
    N = length(Config),
    NewConfig = new_array(N),
    foreach (K in 1..N)
        (K == I -> NewConfig[K] = NewICars
        ;K == J -> NewConfig[K] = [ClearCar|JCars]
        ; NewConfig[K] = Config[K])
    end,
    NextS = (NextConfig,GConfig).
```

The built-in predicate `nth(I,Array,Arg)` is true if the `I`-th argument of `Array` is `Arg`. When `I` is a variable, this predicate nondeterministically searches for an argument that unifies with `Arg`. The rule finds a non-empty curb `I` and a different curb `J` that has less than two cars, and moves the clear car, `ClearCar`, from curb `I` to curb `J`.

Domain knowledge and heuristics

Two knowledge rules can be incorporated into the basic encoding in order to speed up the search. First, when a car is moved into a spot that is its final spot in the goal configuration, such a move should be made deterministically. Note that if a car is moved to a road-side final spot then the curb-side spot must be occupied by a correct car. Second, a car that has been placed in its final spot should not be moved away. This kind of domain knowledge has been used in solving the Tower of Hanoi (Alford *et al.* 2009) and the Blocks World (Bacchus and Kabanza 2000).

Resource-bounded search is used to find an optimal plan. The cost of transforming a state to a final state can be simply estimated as the number of incorrectly-positioned cars. This estimation is admissible, since at least one action is needed to move each incorrectly-positioned car. The following improved heuristic function, which takes special curb configurations into account, is used by the model: For each curb, if the curb has two cars `[A,B]` in the current state, but is required to have `[B,A]` in the final state, then its cost is 4; if the curb has two cars `[A,B]`, but is required to have `[C,A]` or `[B,C]` ($A \neq C$, $B \neq C$), then its cost is 3; otherwise, the cost is the number of incorrectly-positioned cars.

7 Experimental results

In addition to the four domains presented in this paper, we have encoded in Picat several other domains used in the deterministic sequential track of IPC'14. The domains and their Picat encodings are given in Appendix A. The Picat encodings are simple, compact, and comparable in size with the PDDL encodings used in IPC'14. Most of the encodings can be further improved by incorporating sophisticated domain knowledge and heuristics. In order to evaluate the effectiveness of the use of tabling and the use of heuristics, we have built two separate sets of encodings,

Table 1. *The number of instances solved optimally*

Domain	# insts	Picat	Picat-nt	Picat-nh	Symba
Barman	14	**14**	0	**14**	6
Cave	20	**20**	0	**20**	3
Childsnack	20	**20**	20	**20**	3
Citycar	20	**20**	17	18	17
Floortile	20	**20**	0	**20**	**20**
GED	20	**20**	19	13	19
Parking	20	**11**	4	0	1
Tetris	17	**13**	13	9	10
Transport	20	**9**	0	4	8

namely, Picat-nt and Picat-nh, which use the same state representation as the original Picat encodings but have some component removed. Picat-nt performs Prolog-style non-tabled iterative-deepening search. Since Picat-nt does not table any state, it may explore the same state multiple times during search. The Picat-nh encodings do not use any heuristics. We have compared these Picat encodings with the IPC'14 PDDL encodings solved with Symba (Torralba *et al.* 2014), a domain-independent bidirectional A* planner which won the optimal sequential track of IPC'14. This comparison offers a glimpse of how well Picat compares with the best domain-independent planner. A comparison of Picat's planner and several domain-dependent planners also shows the promise of tabled planning (Barták *et al.* 2015).

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 that was solved by both Symba and Picat, the plan quality is the same.

A comparison of Picat and Picat-nt shows the effectiveness of the use of tabling. For every domain, except for *Childsnack* and *Tetris*, Picat solved more instances than Picat-nt. For *Barman*, *Cave*, *Floortile* and *Transport*, Picat-nt could not solve any of the instances. The Picat encodings for five of the domains (*Citycar*, *GED*, *Parking*, *Tetris*, and *Transport*) use heuristics. The use of heuristics is helpful for these domains, especially for *Parking*, for which Picat-nh did not solve any of the instances.

Picat solved more instances than Symba for every domain except for *Floortile*, for which both systems solved all of the instances. 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.

8 Conclusion and future work

This paper has presented Picat's planner, its implementation, and example models for several domains from IPC'14. The example models illustrate several modeling

techniques in Picat. One key task of modeling is finding an efficient state representation. While classical planning frameworks such as PDDL are based on a factored representation of states, Picat uses a structured representation. A structured state representation can leave out unnecessary information that is not needed for planning and can break symmetries by avoiding enumerating all possible permutations of objects. Another key task of modeling is utilizing domain knowledge to reduce search spaces. In the past, a lot of work has been done on the use of domain knowledge in planning (Bacchus and Kabanza 2000; Haslum and Scholz 2003; Kautz and Selman 1998), but recently, this part of modeling has been put aside, because of the advancement of domain-independent PDDL planners. This paper has shown that, even with simple domain knowledge, the declarative encodings of Picat significantly outperform Symba, a state-of-the-art domain-independent planner.

This paper has demonstrated for the first time that tabled logic programming is competitive with the cutting-edge PDDL planners. The key to the success is tabling. Tabling avoids repeating the exploration of the same state and facilitates performing resource-bounded search. The Picat planner does not do prior grounding that is typical for most current state-space search planners, and hence Picat has no problem with exploded memory consumption due to grounding. Nevertheless, memory consumption can be demanding during search since every encountered state is tabled. This is why careful modeling that removes symmetries and uses domain control knowledge to prune useless state transitions is important.

A good state representation should also exploit the underlying term-sharing technique that is used in the tabling system. In the examples that are presented in this paper, ordered lists are used to represent collections of objects. It takes linear time to perform the basic operations. One direction for future work is to design data structures for state representations which are compact, efficient, and good for sharing.

A plethora of action languages have been designed for modeling and solving planning problems (e.g., *A* and its successors (Gelfond and Lifschitz 1998), Golog (Levesque *et al.* 1997), and *K* (Eiter *et al.* 2004)). The focus of these languages has been on the modeling power rather than efficiency and scalability. These languages have been implemented by translation into SAT, ASP, CP, or PDDL (Baier *et al.* 2011; Dovier *et al.* 2011), but no implementation has been shown to be competitive with the cutting-edge PDDL planners. Picat can be used as an implementation language for these action languages. Like in PDDL, a state is represented as a set of flat facts in all of these action languages. Another direction for future work is to devise an efficient translation from these action languages into Picat that automatically exploits structural representation, symmetries, domain control knowledge, and heuristics.

References

ALFORD, R., KUTER, U. AND NAU, D. S. 2009. Translating HTNs to PDDL: A small amount of domain knowledge can go a long way. In *IJCAI*. 1629–1634.

BACCHUS, F. AND KABANZA, F. 2000. Using temporal logics to express search control knowledge for planning. *Artif. Intell. 116*, 1-2, 123–191.

BAIER, J. A., FRITZ, C. AND MCILRAITH, S. A. 2011. Golog-style search control for planning. In *Knowing, Reasoning, and Acting: Essays in Honour of Hector J. Levesque*, G. Lakemeyer and S. A. McIlraith, Eds. College Publications.

BARTÁK, R., DOVIER, A. AND ZHOU, N.-F. 2015. On modeling planning problems in tabled logic programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. PPDP '15. 31–42.

BARTAK, R. AND ZHOU, N. 2014. Using tabled logic programming to solve the Petrobras planning problem. *Theory and Practice of Logic Programming 14,* 4-5, 697–710.

BREWKA, G., EITER, T. AND TRUSZCZYŃSKI, M. 2011. Answer set programming at a glance. *Commun. ACM 54,* 12, 92–103.

CHRPA, L., VALLATI, M. AND MCCLUSKEY, L. 2014. International planning competition.

DAWSON, S., RAMAKRISHNAN, C. R. AND WARREN, D. S. 1996. Practical program analysis using general purpose logic programming systems — A case study. *ACM SIGPLAN Notices 31,* 5, 117–126.

DOVIER, A., FORMISANO, A. AND PONTELLI, E. 2011. Perspectives on logic-based approaches for reasoning about actions and change. In *LNCS*. Vol. 6565. 259–279.

EITER, T., FABER, W., LEONE, N., PFEIFER, G. AND POLLERES, A. 2004. A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Trans. Comput. Log. 5,* 2, 206–263.

GEBSER, M., KAMINSKI, R., KAUFMANN, B. AND SCHAUB, T. 2012. *Answer Set Solving in Practice*. Morgan and Claypool Publishers.

GELFOND, M. AND LIFSCHITZ, V. 1998. Action languages. *Electron. Trans. Artif. Intell. 2,* 193–210.

GUO, H.-F. AND GUPTA, G. 2008. Simplifying dynamic programming via mode-directed tabling. *Softw., Pract. Exper. 38,* 1, 75–94.

HASLUM, P. AND SCHOLZ, U. 2003. Domain knowledge in planning: Representation and use. In *ICAPS Workshop on PDDL*.

HEWITT, C. 1969. Planner: A language for proving theorems in robots. In *IJCAI*. 295–302.

KAUTZ, H. AND SELMAN, B. 1998. The role of domain-specific knowledge in the planning as satisfiability framework. In *AIPS98*. 181–189.

KORF, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell. 27,* 1, 97–109.

KOWALSKI, R. 1979. *Logic for Problem Solving*. North Holland, Elsevier.

LEVESQUE, H. J., REITER, R., LESPÉRANCE, Y., LIN, F. AND SCHERL, R. B. 1997. GOLOG: A logic programming language for dynamic domains. *J. Log. Program. 31,* 1-3, 59–83.

LIFSCHITZ, V. 2002. Answer set programming and plan generation. *Artif. Intell. 138,* 1-2, 39–54.

MCDERMOTT, D. 1998. The planning domain definition language manual. CVC Report 98-003, Yale Computer Science Report 1165.

MICHIE, D. 1968. "memo" functions and machine learning. *Nature*, 19–22.

NIELSON, F., NIELSON, H. R., SUN, H., BUCHHOLTZ, M., HANSEN, R. R., PILEGAARD, H. AND SEIDL, H. 2004. The succinct solver suite. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference (TACAS), LNCS 2988*. 251–265.

PIENTKA, B. December 2003. Tabled higher-order logic programming. Ph.D. thesis, Technical Report CMU-CS-03-185.

RAMAKRISHNA, Y. S., RAMAKRISHNAN, C. R., RAMAKRISHNAN, I. V., SMOLKA, S. A., SWIFT, T. AND WARREN, D. S. 1997. Efficient model checking using tabled resolution. In *Computer Aided Verification*. 143–154.

Tamaki, H. and Sato, T. 1986. OLD resolution with tabulation. In *ICLP*. 84–98.

Torralba, A., Alcazar, V. and Borrajo, D. 2014. Symba: A symbolic bidirectional a planner. In *The 2014 International Planning Competition*. 105–109.

Warren, D. H. D. 1974. WARPLAN: A system for generating plans. Tech. Rep. DCL Memo 76, University of Edinburgh.

Warren, D. S. 1992. Memoing for logic programs. *Comm. of the ACM, Special Section on Logic Programming 35*, 93–111.

Zhou, N.-F. 2014. Combinatorial search with Picat. *ICLP, invited talk, http://arxiv.org/abs/1405.2538*.

Zhou, N.-F. and Dovier, A. 2013. A tabled Prolog program for solving Sokoban. *Fundam. Inform. 124,* 4, 561–575.

Zhou, N.-F. and Have, C. T. 2012. Efficient tabling of structured data with enhanced hash-consing. *Theory and Practice of Logic Programming 12,* 4-5, 547–563.

Zhou, N.-F., Kameya, Y. and Sato, T. 2010. Mode-directed tabling for dynamic programming, machine learning, and constraint solving. In *ICTAI*. 213–218.

Zhou, N.-F., Sato, T. and Shen, Y.-D. 2008. Linear tabling strategies and optimizations. *Theory and Practice of Logic Programming 8,* 1, 81–109.