Solving Several Planning Problems with Picat

Neng-Fa Zhou[1] and Hakan Kjellerstrand[2]

1. The City University of New York,
E-mail: zhou@sci.brooklyn.cuny.edu

2. Independent Researcher, hakank.org,
E-mail: hakank@gmail.com

Abstract: In this paper, we present programs in Picat for solving three planning puzzles, including 15-puzzle, Klotski, and the Rubik's cube. All these programs use the planner module of Picat. For a planning problem, we only need to specify the conditions on the final states and the set of actions, and call the planner on an initial state to find a plan or a best plan. The planner module uses tabling. It tables states encountered during search and performs *resource-bounded search* to fail states that cannot lead to a final state with the available resources. The Picat programs for the problems are straightforward. The programs for 15-puzzle and Klotski are very efficient. The Rubik's cube program has succeeded in solving instances that require 14 or fewer moves. As computers have more and more memories, we believe that the tabling approach to planning will become increasingly more effective and important.

Key Words: Planning, Logic Programming, Tabling, Picat, 15-Puzzle, Klotski, and Rubik's cube

1 INTRODUCTION

The classical planning problem has been a target problem for logic programming since its inception. The first logic programming language, PLANNER [3], was designed as "a language for proving theorems and manipulating models in a robot", and planning has been an important problem domain for Prolog [8, 11]. Despite the amenability of Prolog to planning, there has been little success in applying Prolog to planning due to the looping and the state explosion challenges. The planning problems that have been tackled by using Prolog are mostly toy problems, and Prolog is not recognized as a tool for planning.

The tabling approach to planning as provided by Picat[13] is more effective than Prolog, ASP [1, 9], and the satisfiability [4, 10] approaches to planning. Just like traditional STRIPS-based planners [2], a tabling-based planner treats a planning problem as a state-space search problem. During search, the planner tables all the states that have been encountered so that no state will be expanded more than once. The success of the tabling approach is attributed to several techniques, including term-sharing [15] and resource-bounded search [13]. The term-sharing technique alleviates the state explosion problem commonly seen in planning problems. The resource-bounded search technique determines if a state should be expanded based on the current resource amount and whether or not the state has failed before.

We have written programs in Picat to solve several planning puzzles. The programs are available at:

```
http://picat-lang.org/projects.html
```

In this paper, we present the programs for three problems, including the 15-puzzle, Klotski, and the Rubik's cube. For each problem, we describe the program and report on the experimental results.

2 THE PLANNER MODULE OF PICAT

Picat is a logic-based multi-paradigm programming language that provides pattern matching, deterministic and non-deterministic rules, loops, list comprehensions, functions, constraints, and tabling as its core modeling and solving features. This section briefly describes the planner module of Picat. The readers are referred to the user's guide [14] and Hakan Kjellerstrand's Picat page [5] for the details of Picat, and [13] for the details of the implementation.

The `planner` module of Picat provides predicates that can be used to solve planning problems. Given an initial state, a final state, and a set of possible actions, a planning problem is to find a plan that transforms the initial state to the final state. In order to use the `planner` module to solve a planning problem, users have to define `final/1` or `final/3` to specify the conditions on the final states and `action/4` to specify the state transition diagram.

- `final(`S`)`: This predicate succeeds if S is a final state.
- `final(`S`,`$Plan$`,`$Cost$`)`: A final state can be reached from S by the action sequence in $Plan$ with the cost $Cost$. If this predicate is not given, then the system assumes the following definition:

```
final(S,Plan,Cost) =>
    Plan=[], Cost=0, final(S).
```

- `action(`S`,`$NextS$`,`$Action$`,`$ActionCost$`)`: This predicate encodes the set of actions of the planning problem. The state S can be transformed to $NextS$ by performing $Action$. The cost of $Action$

978-1-4799-3708-0/14/$31.00 ©2014 IEEE

```
import planner.

go =>
    S0=[s,s,s,s],
    best_plan(S0,Plan),
    writeln(Plan).

final([n,n,n,n]) => true.

action([F,F,G,C],S1,Action,ActionCost) ?=>
    Action=farmer_wolf,
    ActionCost=1,
    opposite(F,F1),
    S1=[F1,F1,G,C],
    not unsafe(S1).
action([F,W,F,C],S1,Action,ActionCost) ?=>
    Action=farmer_goat,
    ActionCost=1,
    opposite(F,F1),
    S1=[F1,W,F1,C],
    not unsafe(S1).
action([F,W,G,F],S1,Action,ActionCost) ?=>
    Action=farmer_cabbage,
    ActionCost=1,
    opposite(F,F1),
    S1=[F1,W,G,F1],
    not unsafe(S1).
action([F,W,G,C],S1,Action,ActionCost) =>
    Action=farmer_alone,
    ActionCost=1,
    opposite(F,F1),
    S1=[F1,W,G,C],
    not unsafe(S1).

opposite(n,Opp) => Op=s.
opposite(s,Opp) => Opp=n.

unsafe([F,W,G,_C]),W==G,F!==W => true.
unsafe([F,_W,G,C]),G==C,F!==G => true.
```

Figure 1: A program for the Farmer's problem.

is $ActionCost$, which must be a non-negative integer. In case the plan's length is the only interest, then $ActionCost$ should be 1.

The following predicates and functions are used in this paper.

- `plan(`S`,`$Limit$`,`$Plan$`,`$PlanCost$`)`: This predicate, if succeeds, binds $Plan$ to a plan that can transform state S to a final state that satisfies the condition given by `final/1` or `final/3`. $PlanCost$ is the cost of $Plan$, which cannot exceed $Limit$, a given non-negative integer.

  This predicate searches for a plan by performing *resource-bounded* search. In the search tree, each node represents a state and carries attribute values including the remaining resource amount that can be used by future actions to transform the state to a final state. In resource-bounded search, a node is expanded only if the state is new and the resource amount is non-negative, or the state has occurred before in an old node that had failed before due to a lack of resources but the current node carries more resources than the old one. The arguments $Limit$ and $PlanCost$ are optional. If $PlanCost$ is missing, then the cost of the plan is not returned. If $Limit$ is missing, then a large integer is used as the resource limit.

- `best_plan(`$InitS$`,`$Limit$`,`$Plan$`,`$PlanCost$`)`: This predicate uses `plan/4` to find an optimal plan. It first calls `plan/4` to find a plan of 0 cost. If no plan is found, then it increases the cost limit by 1. In this way, the first plan that is found is guaranteed to be optimal. If no plan is found after the cost exceeds $Limit$, then the predicate fails. The arguments $Limit$ and $PlanCost$ are optional. For example, `best_plan(`$InitS$`,`$Plan$`)` finds an optimal plan, imposing no limit on the cost.

The following built-ins can be used to retrieve the attribute values of the current node if the search is initiated by `plan` or `best_plan`.

- `current_resource()`: This function returns the resource amount of the current node. This function can be used to check a heuristic function. If the heuristic estimate of the cost to travel from the current state to a final state is greater than the available resource amount, then the current state can be failed.

- `current_plan()`: This function returns the plan in reversed order that has transformed the initial state to the current state. The current plan can be used to ban certain sequences of state transitions.

The program shown in Figure 1 solves the Farmer's problem by using the `planner` module. In this example, the length of the plan is the only interest, so the cost of each action is 1. Also, note that there is no heuristic function used. Pattern-matching requires all output unifications to be given in the bodies of rules. Pattern-matching facilitates full indexing of rules. For this example, the matching of the first argument of a call against the four-element list pattern is done only once for each call.

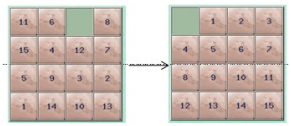

Figure 2: 15-Puzzle

3 THE 15-PUZZLE

In the 15-puzzle, there is a 4×4 board and there are fifteen tiles numbered from 1 to 15. In a configuration, each tile occupies a square on the board, and one square is empty. The goal of the puzzle is to arrange the tiles from their initial configuration to a goal configuration by making sliding moves that use the empty square. Figure 2 shows an instance.

3.1 Encoding

A state is represented by a list of sixteen elements. Each element is a cons `[`R_i`|`C_i`]` where R_i is a row number and C_i is a column number. The first element in the list gives the position of the empty square, and the remaining elements in the list give the positions of the numbered tiles from 1 to 15.
The `final/1` predicate is defined as follows:

```
final(State) =>
    State=[[1|1],[1|2],[1|3],[1|4],
           [2|1],[2|2],[2|3],[2|4],
           [3|1],[3|2],[3|3],[3|4],
           [4|1],[4|2],[4|3],[4|4]].
```

which represents the configuration where the empty square is located at the upper left corner, and the numbered tiles are ordered from left to right and top to down.
The actions are defined by four rules, each representing a changing move of the empty square in one of the four different directions: `up`, `down`, `left` and `right`. For example, the following rule defines the `up` move.

```
action([P0@[R0|C0]|Tiles],NextS,Move,Cost) ?=>
    R1 = R0-1,
    R1 >= 1,
    Move=up,
    Cost=1,
    NewP0 = [R1|C0],
    update(Tiles,P0,NewP0,NewTiles),
    manhattan_heuristic(NewTiles),
    NextS=[NewP0|NewTiles].
```

Let `P0=[R0|C0]` be the current position of the empty square. After it is moved up, its position is changed to `NewP0=[R0-1|C0]`. The `update` predicate changes the tile that is currently at `NewP0` to `P0`.

```
update([NewP0|Tiles],P0,NewP0,NewTiles) =>
    NewTiles = [P0|Tiles].
update([Tile|Tiles],P0,NewP0,NewTiles) =>
    NewTiles=[Tile|NewTiles1],
    update(Tiles,P0,NewP0,NewTiles1).
```

The predicate `manhattan_heuristic(NewTiles)` checks if the Manhattan distance from the new state to the final state is less than the current resource bound. The function `current_resource` is used to retrieve the current resource amount.

3.2 Experimental Results

We have run the Picat program on the instances used in the second ASP competition. Picat found optimal solutions to all of the 15 instances used in the competition. For most of the instances, it took less than 1 second, and it took only 4 seconds to solve the hardest instance. In comparison, the winner of the competition, Potassco, failed to find optimal solutions to 5 of the 15 instances within the time limit of 15 minutes per instance.

All the experimental results given in this section and afterwards were obtained on a Linux machine with 4-core AMD II X4 945 processor and 8GM RAM.

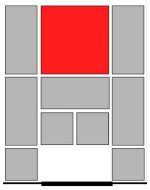

Figure 3: Klotski

Tabling and resource-bounded search are crucial for solving the problem. The Manhattan distance heuristic also contributes to the good performance of the program. When the heuristic is not used, the program runs out of memory on all of the instances.
Our Picat program is comparable in speed with the best IDA* algorithms that use heuristics and pattern databases [7]. Unlike the IDA* algorithms, our Picat program does not require pre-computation of pattern databases. Our program does not use the heuristics to select best nodes to expand, but uses the heuristics to fail nodes.

4 Klotski

Klotski is another sliding puzzle. There are one 2×2 piece, one 2×1 piece, four 1×2 pieces, and four 1×1 pieces. Initially the pieces are placed on a 4×5 board, as shown in Figure 3. The goal of the game is to slide the 2×2 piece to the exit. No pieces can be removed from the board and pieces can only be slid to the empty spaces horizontally or vertically.

4.1 Encoding

A state is represented as a list of positions of the pieces and the free spaces.

```
[FreeSpaces,Piece22,Piece21,Pieces12,Pieces11]
```

A position is represented as a pair of coordinates on the board. We use a coordinate system where the origin is at the top-left, the x-axis goes from left to right, and the y-axis goes from top to down. An element of the list is either a pair, such as `Piece22`, that represents the position of a piece, or an ordered list of pairs, such as `Pieces12`, that represents the positions of the pieces of the same type.
The condition for the final states is given by the `final` predicate.

```
final([_,(2,4)|_]) => true.
```

When the 2×2 piece is at the position `(2,4)`, the piece can be moved out the board in the next move.
The definition of the actions is straightforward. For example, in order to move the 2×2 piece left, the two free spaces must be vertically adjacent to the left edge of the piece. After the move, the x-coordinate of the piece becomes one unit smaller and the x-coordinates of the free spaces become one unit larger than the old values.

4.2 Experimental Results

Our Picat program finds in 15 seconds an optimal solution that consists of 116 moves. As sliding a piece to a free space and then sliding it to the next free space constitute

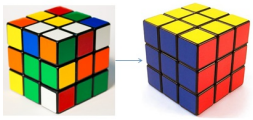

Figure 4: Rubik's Cube

two moves in the encoding, the solution is longer than the known minimum number of steps of 81.

Just like the 15-puzzle program, this program can be improved by using heuristics: after the current state is expanded to a new state, a heuristic function can be called to estimate the number of steps required to transform the state to a final state. If the estimated number of steps is greater than `current_resource()`, then the new state can be failed immediately.

We could not find programs for this problem in Prolog, ASP, or PDDL. This problem may be hard for ASP and any other SAT-based planners because grounding can be a big challenge. A breadth-first search program written Haskell solves the problem in 2 seconds [12].

5 Rubik's Cube

The Rubik's cube is a very popular and very challenging combinatorial puzzle. In the standard version, the cube is made up of $3 \times 3 \times 3 = 26$ cubies. Each of the 6 faces can be turned 90, 180, or 270 degrees relative to the rest of the cube. In the goal configuration, each of the 6 faces of the cube has a single unique color. Given a random configuration of the cube, the goal of the puzzle is to transform the configuration into the goal configuration through a sequence of turns. Figure 4 gives an example instance.

5.1 Encoding

The Rubik's cube has 26 cubies. We call a cubie a *piece* in this paper. A center piece has only one exposed square, an edge piece has two exposed squares, and a corner piece has three exposed squares. We assume that the center pieces stay still in the following way:

back (`b`)	yellow
down (`d`)	orange
front (`f`)	white
left (`l`)	blue
right (`r`)	green
up (`u`)	red

In this way, we can represent a face by the color of its center piece.

We use six letters (`b`, `d`, `f`, `l`, `r`, and `u`) to identify both the faces of the cube and the colors of the squares in the pieces. We use a two-letter ordered name to identify each edge piece and a three-letter ordered name to identify each corner piece. The letters in the name tell the colors of the squares of the piece. For example, the edge piece named `bd` has yellow and orange squares.

We also use the six letters to identify locations. The twelve edge locations are:

```
[bd,bl,br,bu,df,dl,dr,fl,fr,fu,lu,ru]
```

For example, the location `bd` intersects the back and down faces. The eight corner locations are:

```
[bdl,bdr,blu,bru,dfl,dfr,flu,fru]
```

We assume that the locations are ordered by name.

An edge piece has two possible orientations. For each piece and each of its orientations, we use a unique atom to identify the piece in the orientation. For example, the atom `fu2` represents the white-red piece and the orientation is 2. Let S_1 and S_2 be the two square colors of an edge piece ($S_1 < S_2$), and let the two faces that the edge piece intersects be F_1 and F_2 ($F_1 < F_2$). If the square S_1 is on F_1, then the orientation is 1; otherwise, if S_1 is on F_2, then the orientation is 2. For example, when `fu` is at the edge of back-down (`bd`), `fu1` means that the `f` (white) is on the back and `fu2` means that `f` is on the down face.

A corner piece has three possible orientations. Again, for each piece and each of its orientations, we use a unique atom to identify the piece in the orientation. Let S_1, S_2, and S_3 be the three square colors of a corner piece ($S_1 < S_2 < S_3$), and let the three faces that the piece intersects be F_1, F_2, and F_3 ($F_1 < F_2 < F_3$). The orientation is 1 if the square S_1 is on F_1, 2 if S_1 is on F_2, and 3 if S_1 is on F_3. For example, consider the corner piece `fru` (white-green-red). When this piece is located at the corner that intersects the back, left, and up faces (i.e., the corner location `bfl`), if the white square (f) is on left, then the representation is `fru2` because `l` is the second letter in the location name.

We represent a state as a structure `pieces(Edges,Corners)`, where `Edges` is a list of edge pieces in the ordered edge locations and `Corners` is a list of corner pieces in the ordered corner locations. We use two separate lists rather than one larger list because edges and corner do not share names. Also we use lists rather than arrays or structures because list suffixes can be shared among states [15].

The final state is specified by the following predicate:

```
final(pieces(Es,Cs)) =>
  Es=[bd1,bl1,br1,bu1,df1,dl1,
      dr1,fl1,fr1,fu1,lu1,ru1],
  Cs=[bdl1,bdr1,blu1,bru1,dfl1,dfr1,flu1,fru1].
```

A state is final if all of its pieces have correct positions and orientations.

It is very straightforward to define the actions. For example, the action that turns the front face 90 degree clockwise can be defined as follows:

```
action(pieces(Es,Cs),NewS,Action,Cost) ?=>
  Es=[BD,BL,BR,BU,DF,DL,DR,FL,FR,FU,LU,RU],
  Cs=[BDL,BDR,BLU,BRU,DFL,DFR,FLU,FRU]),
  Action=f,
  Cost=1,
  turn_edge(FR,FR1),
  turn_edge(DF,DF1),
  turn_corner(f,dfr_dfl,DFR,DFR1),
  turn_corner(f,dfl_flu,DFL,DFL1),
```

The list patterns of `Es` and `Cs` are moved to the body for the sake of formatting. They should be placed in the head so that all the rules can share the patterns and pattern-matching can be done only once for a call.

```
turn_corner(f,flu_fru,FLU,FLU1),
turn_corner(f,fru_dfr,FRU,FRU1),
NEs=[BD,BL,BR,BU,FR1,DL,DR,DF1,FU,FL,LU,RU],
NCs=[BDL,BDR,BLU,BRU,DFR1,FRU1,DFL1,FLU1],
NewS = $pieces(NEs,NCs).
```

The call `turn_edge(X,X1)` changes the edge piece `X` to `X1`. If `X`'s orientation is 1, then `X1`'s orientation becomes 2; otherwise, if `X`'s orientation is 2, then `X1`'s orientation becomes 1. The call `turn_corner(Face,C_C1,X,X1)` changes the corner piece `X` to `X1` after a turn of `Face` and the piece moves from the location `C` to `C1`.

5.2 Improvements and Experimental Results

The basic model is not efficient. In order to find a plan of length n, it generates all the nodes of the search tree that have depth n or less. It can be used to solve configurations that require 7 or fewer steps. For more difficult configurations, the program will run out of memory since it will require a large table to store all the states. For $n = 8$, there are already 1,373,243,544 nodes [6]! Even with sharing, these nodes would require a huge amount of memory to table.

We can use a heuristic function to improve the efficiency. After each node is generated, we estimate the cost for transforming the state to the final state. We can use the built-in function `current_resource()` of the `planner` module to retrieve the current resource limit. If the estimated cost is greater than the limit, the node should be failed. In order to guarantee the soundness and completeness of the program, the heuristic function must be admissible. Unfortunately, there are no good admissible heuristic functions available for Rubik's cube. The total Manhattan distance of the pieces is not an admissible heuristic, even when it is divided by 8. We can also use the number of moves required to put the corner pieces into correct positions as a heuristic, as is used in Korf's program [6]. However, it's space consuming to store the pattern databases, unless the pattern databases are compressed. Therefore, we need different kinds of improvements.

The first improvement, called *target enlargement*, is inspired by bi-directional search. The number of nodes to be generated from depth n to $n + 1$ is exponential in n. Therefore, it takes a lot of time and memory space to go one level deeper when $n \geq 8$. Nevertheless, we can bring the target configuration closer by expanding it. When all the states that can be reached in 5 steps are treated as final, the program can solve instances that require 12 moves.

The second improvement, called *forward checking*, combines tabled search with Prolog style depth-first search. After the remaining resource limit becomes 8, the program switches from tabled search to Prolog style non-tabled search, checking if a final state is reachable from the current state. If no final state can be reached, then the current state is failed. With this technique, the program can solve instances that require 14 moves in five hours.

The third improvement, called *state compression*, compresses the two lists of pieces into two big integers. This improvement makes it possible to generate states at deeper levels, both forward from the initial configuration and backward from the target configuration.

Further refinements are made by banning certain sequences of moves. First, a face is not allowed to move consecutively. Although tabling is able to deal with looping, it is a waste to generate the same state again. Second, a symmetry-breaking rule is enforced that imposes an order on the consecutive moves of opposite faces. These refinements improve the speed by about 10%.

6 Conclusion

In this paper, we present three example solutions using the `planner` module of Picat. Since users of the module only need to specify conditions on the final states and the set of actions, and call one of the predicates of the module on an initial state to find a plan, the `planner` module simplifies modeling of planning problems. The experimental results show that the tabling approach to planning is promising. The programs for 15-puzzle and Klotski are very efficient. The Rubik's cube program has succeeded in solving instances that require 14 or fewer moves. We believe that the program can be used to solve hard instances on a computer with a large amount of memory.

Acknowledgements

Neng-Fa Zhou was supported in part by the NSF under grant number CCF1018006.

REFERENCES

[1] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in nonmonotonic logic programs. In *ECP*, pages 169–181, 1997.

[2] R. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971.

[3] C. Hewitt. Planner: A language for proving theorems in robots. In *IJCAI*, pages 295–302, 1969.

[4] H.A. Kautz and B. Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

[5] H. Kjellerstrand. Hakan's Picat Page, 2013. http://hakank.org/picat/.

[6] R.E. Korf. Finding optimal solutions to rubik's cube using pattern databases. In *AAAI/IAAI*, pages 700–705, 1997.

[7] R.E. Korf. Research challenges in combinatorial search. In *AAAI*, 2012.

[8] R. Kowalski. *Logic for Problem Solving*. North Holland, Elsevier, 1979.

[9] V. Lifschitz. Answer set programming and plan generation. *Artif. Intell.*, 138(1-2):39–54, 2002.

[10] J. Rintanen. Planning as satisfiability: Heuristics. *Artif. Intell.*, 193:45–86, 2012.

[11] D.H.D. Warren. WARPLAN: A system for generating plans. Technical Report DCL Memo 76, University of Edinburgh, 1974.

[12] K. Wiberg. Solving klotski. `http://www.treskal.com/kalle/klotski.pdf`, 2009.

[13] N.F. Zhou, R. Bartak, A. Dovier, and H. Kjellerstrand. Tabling for Planning. Technical report, 2013.

[14] N.F. Zhou and J. Fruhman. A User's Guide to Picat, 2013.

[15] N.F. Zhou and C.T. Have. Efficient tabling of structured data