Modeling and Solving Planning Problems With Picat

Neng-Fa Zhou
(CUNY Brooklyn College & GC)
Classical Planning

\[ P = (S, \Sigma, f, \delta, s_0, F) \]

- \( S \): A set of states (finite or countably infinite)
- \( \Sigma \): A set of actions
- \( f \): A transition function or relation \((S \times \Sigma \rightarrow S)\)
- \( \delta \): A cost function \((S \times \Sigma \rightarrow \mathbb{R})\)
- \( s_0 \): An initial state
- \( F \): A set of goal states

Planning with Picat, N.F. Zhou
Planning Formalisms

- Logic programming
  - PLANNER [Hewitt69], “a language for proving theorems and manipulating models in a robot”
  - Prolog for planning [Kowalski79,Warplan76]
  - ASP-based planners [Lifschitz02]

- STRIPS-based PDDL
  - The de facto language [McDermott98]
  - Many solvers (Arvand, LAMA, FD, SymBA*-2,…)
  - Extensions of PDDL (e.g., HTN)

- Planning as SAT and model checking
Planning With Picat

- A logic programming approach
  - Unlike PDDL and ASP, structured data can be used.
  - Domain-specific heuristics and control knowledge about determinism, dependency, and symmetry can be encoded.

- Tabled backtracking search
  - Every state generated during search is tabled.
    - Same idea as state-marking used in IDA* and other algorithms.
  - Term sharing: common ground terms are tabled only once.
    - Alleviate the state explosion problem.
  - Resource-bounded search
    - Unlike IDA*, results from previous rounds are reused.
Picat’s planner Module

- Resource-bounded search
  - `plan(State, Limit, Plan, PlanCost)`
  - `best_plan(State, Limit, Plan, PlanCost)`
    - Iterative deepening (unlike IDA*, results from early rounds are reused)

- Depth-unbounded search
  - `plan_unbounded(State, Limit, Plan, PlanCost)`
  - `best_plan_unbounded(State, Limit, Plan, PlanCost)`
    - Like Dijkstra’s algorithm
How to Use the Planner?

- Import the planner module
- Specify the goal states
  - `final(State)`
    - True if State is a goal state.
- Specify the actions
  - `action(State,NextState,Action,ActionCost)`
    - Encodes the state transition relation
    - States are tabled, and destructive updates of states (using :=) are banned.
- Define a heuristic function if necessary
  - `heuristic(State) = H => …`
- Call a built-in on an initial state to find a plan
Ex: The Farmer’s Problem

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).

...
Modeling Techniques

- Find a good representation for states
  - Keep the information minimal.
  - Use good data structures that facilitate
    - sharing
    - computation of heuristics
    - symmetry breaking

- Use heuristics and domain knowledge
  - A state should not be expanded if the travel from it to the final state costs more than the limit.
  - Identify deterministic actions and macro actions.
  - Use landmarks.
Modeling Examples
picat-lang.org/projects.html

15-puzzle  RushHour  Sokoban  Ricochet Robots

Logistics  Gilbreath’s card trick  Rubik’s Cube  Tower-of-Hanoi

Planning with Picat, N.F. Zhou
15-Puzzle

State representation

```picat
main =>
    Init = [(1,2),(2,2),(4,4),(1,3),(1,1),(3,2),(1,4),(2,4),
            (4,2),(3,1),(3,3),(2,3),(2,1),(4,1),(4,3),(3,4)],
    best_plan(Init,Plan).

final(S) => S = [(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)].
```

Planning with Picat, N.F. Zhou
15-Puzzle: Actions

\[
\text{action}([P0 @(R0,C0) | \text{Tiles}], \text{NextS}, \text{Action}, \text{Cost}) \Rightarrow \\
\quad \text{Cost} = 1, \\
\quad (R1 = R0 - 1, R1 >= 1, C1 = C0, \text{Action} = \text{up}; \\
\quad R1 = R0 + 1, R1 <= 4, C1 = C0, \text{Action} = \text{down}; \\
\quad R1 = R0, C1 = C0 - 1, C1 >= 1, \text{Action} = \text{left}; \\
\quad R1 = R0, C1 = C0 + 1, C1 <= 4, \text{Action} = \text{right}), \\
\quad P1 = (R1,C1), \\
\quad \text{slide}(P0,P1,\text{Tiles},\text{NTiles}), \\
\quad \text{NextS} = [P1|\text{NTiles}].
\]

\%
\%
\% slide the tile at P1 to the empty square at P0
\%
\text{slide}(P0,P1,[P1|\text{Tiles}],\text{NTiles}) \Rightarrow \\
\quad \text{NTiles} = [P0|\text{Tiles}].
\%
\text{slide}(P0,P1,[\text{Tile}|\text{Tiles}],\text{NTiles}) \Rightarrow \\
\quad \text{NTiles}=[\text{Tile}|\text{NTilesR}], \\
\quad \text{slide}(P0,P1,\text{Tiles},\text{NTilesR}).
\]
15-Puzzle: Heuristics and Performance

heuristic([_|Tiles]) = Dist =>
  final([_|FTiles]),
  Dist = sum([abs(R-FR)+abs(C-FC) :
               {(R,C),(FR,FC)} in zip(Tiles,FTiles)]).
Rush Hour Puzzle

Move the red car to the exit (4,2).
Rush Hour Puzzle

- **State representation**

  \[ \{ \text{RedLoc}, L_{11}, L_{12}, L_{21}, L_{13}, L_{31} \} \]

  - L_{11} -- an ordered list of locations of the spaces.
  - L_{wh} -- an ordered list of locations of the \( w \times h \) cars.
  - Symmetries are removed.

- **Goal states**

  \[
  \text{final}([4|2], \_, \_, \_, \_, \_, \_) \rightarrow \text{true}.
  \]
Rush Hour Puzzle

Actions

% move the red car
  Cost=1,
  move_car(2,1,LocRed,NLocRed,L11,NL11,Action),
  NewS = {NLocRed,NL11,L12,L21,L13,L31}.

% move a 1*2 car
  Cost=1,
  select(Loc,L12,L12R),
  move_car(1,2,Loc,NLoc,L11,NL11,Action),
  NL12 = L12R.insert_ordered(NLoc),
  NewS = {LocRed,NL11,NL12,L21,L13,L31}.

...
Sokoban

In the ASP’13 version, there may be more stones than goal locations. This makes reversed solving difficult.

source: takaken
Sokoban

State representation

- \{SoLoc, GStLocs, NonGStLocs\}
  - SoLoc – the location of the man.
  - GStLocs – an ordered list of locations of the goal stones.
  - NonGStLocs – an ordered list of locations of the non-goal stones.

Goal states

final({_, GStLocs, _}) =>
  foreach(Loc in GStLocs)
    goal(Loc)
  end.
Sokoban

Actions

% push a goal stone
action({SoLoc, GStLocs, NonGStLocs}, NextState, Action, Cost) ?=>
  NextState = {NewSoLoc, NewGStLocs, NonGStLocs},
  Action = $move_push(SoLoc, StLoc, StDest, Dir),
  Cost = 1,
  choose_goal_stone(Dir, SoLoc, NewSoLoc, GStLocs, StLoc,
                     StDest, GStLocs1, NonGStLocs),
  NewGStLocs = insert_ordered(GStLocs1, StDest).
% push a non-goal stone
action({SoLoc, GStLocs, NonGStLocs}, NextState, Action, Cost) ?=>
  ...
% Sokoban moves alone
action({SoLoc, GStLocs, NonGStLocs}, NextState, Action, Cost) =>
  ...

Planning with Picat, N.F. Zhou
Sokoban

Experimental Results

- 30 instances from ASP’13 were used.
- Picat (using `plan_unbounded`) solved all the 30 instances (on average less than 1s per instance).
- Depth-unbounded search is faster than depth-bounded search.
- Potassco solved only 14 of the 30 instances.
- Not as competitive as Rolling Stone, a specialized Sokoban planner.
Ricochet Robots

source: Martin Gebser et al.
Ricochet Robots

State representation

\{[CurLoc|TargetLoc], ORobotLocs\}

Non-target robots are represented as an ordered list of locations. This representation breaks symmetries.

Goal states

\texttt{final([Loc|Loc],_)} => true.
Ricochet Robots

Actions

```prolog
action([From|To], ORobotLocs), NextState, Action, Cost) ?=>
    NextState = [[Stop|To], ORobotLocs],
    Action = [From|Stop], Cost = 1,
    choose_move_dest(From, ORobotLocs, Stop).
action([FromTo@[From|_], ORobotLocs], NextState, Action, Cost) =>
    NextState = [FromTo, ORobotLocs2],
    Action = [RFrom|RTo], Cost = 1,
    select(RFrom, ORobotLocs, ORobotLocs1),
    choose_move_dest(RFrom, [From|ORobotLocs1], RTo),
    ORobotLocs2 = insert_ordered(ORobotLocs1, RTo).
```

Planning with Picat, N.F. Zhou
Logistics

- IPC domains
  - Nomystery
  - Airport pickup
  - Drivelog
  - Elevator planning
  - Petrobrass planning
  - ...
Nomystery

- There is only one truck involved.
- The truck has a fuel level.
- A number of packages need to be transported between nodes in a graph.
- The graph is weighted and the weight of an edge is the fuel cost.
Nomystery

- **State representation**
  - \{\text{TruckLoc}, \text{LCGs}, \text{WCGs}\}
    - \text{LCGs} – an ordered list of destinations of loaded cargoes
    - \text{WCGs} – an ordered list of source-destination pairs of waiting cargoes

- **Goal states**
  
  ```plaintext
  final({_, [], []}) => true.
  ```
Nomystery

**Actions**

\[
\text{action}(\{\text{Loc}, \text{LCGs}, \text{WCGs}\}, \text{NextState}, \text{Action}, \text{Cost}), \quad \text{select}(\text{Loc}, \text{LCGs}, \text{LCGs1})
\]

\[
= \Rightarrow \\
\text{Action} = \text{unload}(\text{Loc}), \\
\text{Cost} = 0, \\
\text{NextState} = \{\text{Loc}, \text{LCGs1}, \text{WCGs}\}.
\]

\[
\text{action}(\{\text{Loc}, \text{LCGs}, \text{WCGs}\}, \text{NextState}, \text{Action}, \text{Cost}), \quad \text{select}(\{\text{Loc}|\text{CargoDest}\}, \text{WCGs}, \text{WCGs1})
\]

\[
= \Rightarrow \\
\text{Action} = \text{load}(\text{Loc}, \text{CargoDest}), \\
\text{Cost} = 0, \\
\text{NextState} = \{\text{Loc}, \text{LCGs1}, \text{WCGs1}\}, \\
\text{LCGs1} = \text{insert}_{\text{ordered}}(\text{LCGs}, \text{CargoDest}).
\]

\[
\text{action}(\{\text{Loc}, \text{LCGs}, \text{WCGs}\}, \text{NextState}, \text{Action}, \text{Cost}) \Rightarrow \\
\text{Action} = \text{drive}(\text{Loc}, \text{Loc1}), \\
\text{NextState} = \{\text{Loc1}, \text{LCGs}, \text{WCGs}\}, \\
\text{fuelcost}(\text{Cost}, \text{Loc}, \text{Loc1}).
\]

**Domain knowledge**

- If the truck is at the destination of a loaded cargo, then unload it **deterministically**.
- If the truck is at a location where there is a cargo that needs to be delivered, then load it deterministically.
Nomystery

Experimental results

- 30 instances from ASP’13 were used.
- Picat solved all the 30 instances.
  - On average less than 0.1s per instance.
- Potassco solved only 17 of the 30 instances.
- Picat solved all the instances used in IPC’11, including the hardest instance that was not solved by any of the participating solvers.
Gilbreath’s Card Trick

\[(5\clubsuit), (3\heartsuit), (Q\diamondsuit), (8\diamondsuit),\]
\[(K\spadesuit), (2\heartsuit), (7\clubsuit), (4\diamondsuit),\]
\[(8\spadesuit), (J\heartsuit), (9\diamondsuit), (A\diamondsuit)\]

\[\text{split} \quad \rightarrow \quad \]
\[(5\clubsuit), (3\heartsuit), (Q\diamondsuit), (8\diamondsuit), (K\spadesuit),\]
\[(2\heartsuit), (7\clubsuit), (4\diamondsuit), (8\spadesuit), (J\heartsuit), (9\diamondsuit), (A\diamondsuit)\]

\[\text{reverse deck-1} \quad \downarrow \quad \]
\[(2\heartsuit), (7\clubsuit), (4\diamondsuit), (8\spadesuit), (J\heartsuit), (9\diamondsuit), (A\diamondsuit), (K\spadesuit), (8\diamondsuit), (Q\diamondsuit), (3\heartsuit), (5\clubsuit), \]
\[(K\spadesuit), (8\diamondsuit), (Q\diamondsuit), (3\heartsuit), (5\clubsuit),\]
\[(2\heartsuit), (7\clubsuit), (4\diamondsuit), (8\spadesuit), (J\heartsuit), (9\diamondsuit), (A\diamondsuit)\]

\[\text{riffle-shuffle} \quad \leftarrow \quad \]
\[(2\heartsuit), (7\clubsuit), (4\diamondsuit), (8\spadesuit), (J\heartsuit), (9\diamondsuit), (A\diamondsuit)\]

Each quartet contains a card from each suit

Take from “Unraveling a Card Trick”, by Tony Hoare & Natarajan Shankar
Gilbreath’s Card Trick

- State representation

\[
\text{init}([s,h,c,d,s,h,c,d,s,h,c,d])
\]
\[
\text{splitted}(\text{Deck1}, \text{Deck2})
\]
\[
\text{shuffled}(\text{Cards})
\]

- Goal states

\[
\text{final}(\text{shuffled}(\text{Cards})) \Rightarrow
\]
\[
\text{test\_quartet}(\text{Cards}, [c,d,h,s]).
\]
\[
\text{test\_quartet}([C1,C2,C3,C4|_\text{Cards}], \text{Suits}),
\]
\[
\text{sort}([C1,C2,C3,C4]) \neq \text{Suits}
\]
\[
\Rightarrow \text{true}.
\]
\[
\text{test\_quartet}([_,_ ,_ ,_ |\text{Cards}], \text{Suits}) \Rightarrow
\]
\[
\text{test\_quartet}(\text{Cards}, \text{Suits}).
\]
Gilbreath’s Card Trick

Actions

\[
\text{action}(\text{init}(\text{Cards}), \text{NewS}, \text{Action}, \text{ActionCost}) \rightarrow \\
\text{NewS} = \$\text{splitted}(\text{Deck1}, \text{RDeck2}), \\
\text{Action} = \text{split}, \\
\text{ActionCost} = 1, \\
\text{append}(\text{Deck1}, \text{Deck2}, \text{Cards}), \\
\text{Deck1} \neq [], \\
\text{Deck2} \neq [], \\
\text{RDeck2} = \text{Deck2}.\text{reverse}().
\]

\[
\text{action}(\text{splitted}(\text{Deck1}, \text{Deck2}), \text{NewS}, \text{Action}, \text{ActionCost}) \rightarrow \\
\text{NewS} = \$\text{shuffled}(\text{Cards}), \\
\text{Action} = \text{shuffle}, \\
\text{ActionCost} = 1, \\
\text{shuffle}(\text{Deck1}, \text{Deck2}, \text{Cards}).
\]
Rubik’s Cube

$$12! \times 2^{12} \times 8! \times 3^8 = 43,252,003,274,489,856,000$$

43 quintillion possible states!

$$8! \times 3^7 = 88,179,840$$
Rubik’s Cube

- **State representation**
  
  \[
  \text{pieces}(\text{Es}, \text{Cs})
  \]
  
  \(\text{Es} : \) A list of positions of edge pieces.
  
  Edge positions: \([\text{bd}, \text{db}, ..., \text{ru}, \text{ur}]\).
  
  \(\text{Cs} : \) A list of positions of corner pieces.
  
  Corner positions: \([\text{bdl}, \text{bld}, ..., \text{ufr}, \text{urf}]\).

- **The goal state**
  
  \[
  \text{final}(\text{pieces}(\text{Es}, \text{Cs})) \Rightarrow
  \]
  
  \(\text{Es} = [\text{bd}, \text{bl}, \text{br}, \text{bu}, \text{df}, \text{dl}, \text{dr}, \text{fl}, \text{fr}, \text{fu}, \text{lu}, \text{ru}],\)
  
  \(\text{Cs} = [\text{bdl}, \text{bdr}, \text{blu}, \text{bru}, \text{dfl}, \text{dfr}, \text{flu}, \text{fru}].\)
Rubik’s Cube

- Expand the goal state into a goal region

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>243</td>
</tr>
<tr>
<td>3</td>
<td>3,240</td>
</tr>
<tr>
<td>4</td>
<td>43,254</td>
</tr>
<tr>
<td>5</td>
<td>577,368</td>
</tr>
<tr>
<td>6</td>
<td>7,706,988</td>
</tr>
<tr>
<td>7</td>
<td>102,876,480</td>
</tr>
<tr>
<td>8</td>
<td>1,373,243,544</td>
</tr>
<tr>
<td>9</td>
<td>18,330,699,168</td>
</tr>
<tr>
<td>10</td>
<td>244,686,773,808</td>
</tr>
<tr>
<td>11</td>
<td>3,266,193,870,720</td>
</tr>
<tr>
<td>12</td>
<td>43,598,688,377,184</td>
</tr>
<tr>
<td>13</td>
<td>581,975,750,199,168</td>
</tr>
<tr>
<td>14</td>
<td>7,768,485,393,179,328</td>
</tr>
<tr>
<td>15</td>
<td>103,697,388,221,736,960</td>
</tr>
<tr>
<td>16</td>
<td>1,384,201,395,738,071,424</td>
</tr>
<tr>
<td>17</td>
<td>18,476,969,736,848,122,368</td>
</tr>
<tr>
<td>18</td>
<td>246,639,261,965,462,754,048</td>
</tr>
</tbody>
</table>

From Richard E. Korf’97

final(S, Plan, Cost) =>
  M = get_table_map(),
  M.get(S,[]) = (Plan, Cost).
Rubik’s Cube

- **Actions**

  \[
  \text{action}(S,\text{New}S,\text{Action},\text{Cost}) \Rightarrow \\
  \text{current\_resource\_plan\_cost}(\text{Limit},\text{CurPlan},_\text{CurPlanLen}), \\
  \text{actions}(\text{Actions}), \\
  \text{Cost} = 1, \\
  \text{member}(\text{Action},\text{Actions}), \\
  \text{not\ nogood\ action}(\text{CurPlan},\text{Action}), \\
  \text{transform}(\text{Action},S,\text{New}S).
  \]

- **Some domain knowledge**

  - Do not turn one face consecutively.
  - Do not turn opposite faces consecutively.
Rubik’s Cube

- **Experimental results**
  - 2×2×2
    - Out-of-memory for table area if no goal region is used.
    - When the goal is expanded backward by 5 steps, Picat solves most instances in seconds.
  - 3×3×3
    - Picat can solve only easy instances that require up to 14 steps.
    - Hard instances normally require 18 steps (in theory, no more than 20 steps).
    - Korf’s pattern database is too big to store in the table area.
Hanoi Tower (4 Pegs)

Two snapshots from the sequence of the *Frame-Stewart* algorithm

Planning with Picat, N.F. Zhou
Hanoi Tower (4 Pegs)

- Remove correctly-positioned largest disks
Hanoi Tower (4 Pegs)

Set up a landmark

Sub-prob-1

Sub-prob-2
Hanoi Tower (4 Pegs)

- State representation

\{N, \text{CurTower}, \text{GoalTower}\}

\text{CurTower} = [\text{CPeg1}, \text{CPeg2}, \text{CPeg3}, \text{CPeg4}]

\text{GoalTower} = [\text{GPeg1}, \text{GPeg2}, \text{GPeg3}, \text{GPeg4}]

\text{Pegi} = [D_1, D_2, \ldots, D_k], \quad D_1 > D_2 > \ldots > D_k
Hanoi Tower (4 Pegs)

table (+,-,min)

hanoi4({0,_,_},Plan,Cost) => Plan=[],Cost=0.
% reduce the problem if the largest disk already is on the right peg
  NewS = {N-1,[C Peg1|C Pegs],[G Peg1|G Pegs]},
  hanoi4(NewS,Plan,Cost).

...hanoi4({1,CT,GT},Plan,Cost) =>
  nth(From,CT,[_]),
  nth(To,GT,[_]),
  Plan = [move(From,To)],
  Cost = 1.
% divide the problem into sub-problems
hanoi4({N,CState,GState},Plan,Cost) =>
  partition_disks(N,CState,GState,ItState,M,Peg),    % set up a landmark
  remove_larger_disks(CState,M) = CState1,
  hanoi4({M,CState1,ItState},Plan1,Cost1),    % sub-problem1
  remove_smaller_or_equal_disks(CState,M) = CState2,
  remove_smaller_or_equal_disks(GState,M) = GState2,
  N1 is N-M,
  hanoi3({N1,CState2,GState2,Peg},Plan2,Cost2),    % sub-problem2, 3-peg version
  remove_larger_disks(GState,M) = GState1,
  hanoi4({M,ItState,GState1},Plan3,Cost3),    % sub-problem3
  Plan = Plan1 ++ Plan2 ++ Plan3,
  Cost = Cost1 + Cost2 + Cost3.
Experimental results

- 15 instances from ASP’11 were used
- Picat solved all
  - In less than 0.1s when no partition heuristic was used.
  - Is even faster if a partition heuristic was used.
- Clasp also solved all 15 instances
  - On average 20s per instance
Summary
Modeling Techniques

- Use an ordered list to represent positions
  - Rush Hour, Sokoban, Ricochet Robots, and Nomystery.
  - Breaks symmetry and facilitates sharing
- Use heuristics (15-puzzle and Ricochet)
- Identify deterministic actions (Nomystery)
- Goal expansion (Rubik’s cube)
- Use landmarks (4-peg Hanoi Tower)