Tabling for Planning
Modeling and Solving Planning Problems With Picat

Neng-Fa Zhou
(CUNY Brooklyn College & GC)

With contributions by Roman Bartak, Agostino Dovier, Hakan Kjellerstrand, and Jonathan Fruhman

ECAI’14 Tutorial, Prague
8/18/2014
Outline of Tutorial

- An overview of Picat [20m]
- Tabling in Picat [10m]
- Planning with Picat [50m]
  - The planner module of Picat
  - Modeling techniques
  - Modeling examples
- Summary
An Overview of Picat

- Why the name “PICAT”?  
  - Pattern-matching, Imperative, Constraints, Actors, Tabling
- Core logic programming concepts (Prolog-like)  
  - Logic variables (arrays and maps are terms)  
  - Implicit pattern-matching and explicit unification  
  - Explicit non-determinism
- Language constructs for scripting  
  - Functions, loops, and list comprehension
- Modeling and solving  
  - CP, SAT, and MIP for constraint solving  
  - Tabling for dynamic programming and planning
Logic Variables

A = {{1, _3d4, _3d8}, {_3e0, _3e4, 5}}

Picat> M = new_map([alpha=1, beta=2]), M.get(alpha) = A
M = (map)[alpha = 1,beta = 2]
A = 1
Pre-created Maps
(Picat has no assert/retract)

- Map = get_heap_map()
  - Stored on the heap.
  - Like a normal map, updates are undone on backtracking.

- Map = get_global_map()
  - Stored in the global area.
  - Updates are not undone on backtracking.

- Map = get_table_map()
  - Stored in the table area; keys and values are hash-consed
  - Updates are not undone on backtracking.
Pattern Matching

Head, Cond  =>  Body.

- Pattern-matching rules
  - Pattern-matching is adequate for many applications
    - NLP, state-space search, …
  - Pattern-matching rules are fully indexed
    - Picat can be more scalable than Prolog
  - Explicit unification (X=Y) is supported
  - As-pattern

\[
is_{\text{sorted}}([X|L@[Y|\_]]) \Rightarrow X =< Y, \ is_{\text{sorted}}(L).
\]
Explicit Non-determinism

member(X, [Y|\_]) ?=> X=Y.
member(X, [\_|L]) => member(X,L).

- Explicit non-determinism
  - Although non-determinism is helpful for many computations, deterministic computations are the majority.
  - The cut operator is unneeded.
  - Use the once operator to remove choice points.
Functions


\[
\text{fib}(N) = 1, \ N = <1 \ => \ true.
\text{fib}(N) = \text{fib}(N-1) + \text{fib}(N-2).
\]

- Many logic languages support functions
  - Curry, Erlang, Mozart-OZ,…
- Functions are easier to use than relations
  - Function calls never fail (at least built-in functions)
  - Function calls can be nested
  - Directionality helps readability
- Special notation is needed for structures
  Picat> S = $student(mary,cs,3.8)
Loops

foreach(E_1 \text{ in } D_1, \text{Cond}_1, \ldots, E_n \text{ in } D_n, \text{Cond}_n)
\quad\text{Goal}
end

- Loops are convenient for scripting and modeling purposes
  - Lisp, Python, C#, Java, C++11, ...
- Loops are compiled to tail-recursion
Loops

- Scopes of variables in loops
  - Variables that occur within a loop but not before in its outer scope are local to each iteration

\[
p(A) \Rightarrow q(X),
\]
\[
\text{foreach}(I \text{ in } 1 \ldots A.length)
\]
\[
A[I] = (X, Y, Y, _)
\]
\[
\text{end.}
\]

\(x\) is global, and \(y\) is local.

The anonymous variable is always local.
List Comprehension

\[[T : E_1 \text{ in } D_1, \text{Cond}_1, \ldots, E_n \text{ in } D_n, \text{Cond}_n]\]

- Convenient for constructing lists
  - Supported by more than 30 languages, according to Wiki.
- Compiled to a foreach loop
  - The assignment operator (=) is used to accumulate values
Picat = Python + Pattern Matching + Nondeterminism + ...

\[
\text{power\_set}([[]]) = [[]].
\]
\[
\text{power\_set}([H|T]) = P1++P2 \Rightarrow
\]
\[
P1 = \text{power\_set}(T),
\]
\[
P2 = [[H|S] : S \text{ in } P1].
\]

\[
\text{perm}([[]]) = [[]].
\]
\[
\text{perm}(\text{Lst}) = [[E|P] : E \text{ in } \text{Lst}, P \text{ in } \text{perm}(\text{Lst.delete}(E))].
\]

\[
\text{matrix\_multi}(A,B) = C \Rightarrow
\]
\[
C = \text{new\_array}(A.\text{length},B[1].\text{length}),
\]
\[
\text{foreach}(I \text{ in } 1..A.\text{length}, J \text{ in } 1..B[1].\text{length})
\]
\[
\quad C[I,J] = \text{sum}([A[I,K]*B[K,J] : K \text{ in } 1..A[1].\text{length}])
\]
\[
\text{end}.
\]
Outline of Tutorial

- An overview of Picat [20m]
- Tabling in Picat [10m]
- Planning with Picat [50m]
  - The planner module of Picat
  - Modeling techniques
  - Modeling examples
- Summary
Tabling for Dynamic Programming

- The idea [Michie68, Tamaki&Sato86, Warren92]
  - Tabling memorizes calls and their answers in order to prevent infinite loops and to limit redundancy

- Tabling is exciting
  - Computers have more and more memory
  - Advanced implementation techniques have improved the scalability

- Tabling approaches
  - Suspension-based tabling and linear tabling
Tabling-All

table
fib(0)=1.
fib(1)=1.
fib(N)=fib(N-1)+fib(N-2).

table
reach(X,Y) ?=> edge(X,Y).
reach(X,Y) => reach(X,Z),edge(Z,Y).
Mode-directed Tabling in Picat

table(+,+,-,min)
shortest_path(X,Y,Path,W) ?=>
  Path = [(X,Y)],
  edge(X,Y,W),
shortest_path(X,Y,Path,W) =>
  Path = [(X,Z)|PathR],
  edge(X,Z,W1),
  shortest_path(Z,Y,PathR,W2),
  W = W1+W2.

- Table modes
  - + (input), - (output), min, max, nt (not-tabled)

- Semantics
  - Table the best answer for each tuple of input arguments

- Linear tabling
  - Iteratively evaluate looping calls until the optimum is reached
The Omelet Problem
(Or The N-Eggs Problem)

Table (+,+,min)
omelet(_,0,NAttempts) => NAttempts=0.
omelet(_,1,NAttempts) => NAttempts=1.
omelet(1,H,NAttempts) => NAttempts=H.
omelet(N,H,NAttempts) =>
  between(1,H,L), % make a choice
  omelet(N-1,L-1,NAttempts1), % the egg breaks
  omelet(N,H-L,NAttempts2), % the egg survives
  NAttempts is max(NAttempts1,NAttempts2)+1.

http://www.datagenetics.com/blog/july2012/
Linear Tabling

A', a variant of A, fails. A needs to be re-evaluated in some cases.
Term Sharing
(Hash-consing)

Hashtable

[1, 2, 3]
[2, 3]
[3]

Hash codes are tabled, so it’s fast to test if two states are the same.
Tabled Planning

```prolog
table (+,-,min)
plan(S,Plan,Cost),final(S) => Plan=[],Cost=0.
plan(S,Plan,Cost) =>
    action(S,S1,Action,ActionCost),
    plan(S1,Plan1,Cost1),
    Plan = [Action|Plan1],
    Cost = Cost1+ActionCost.
```

- With tabling, state-space tree search becomes state-space graph search
- Depth-unbounded search
Tabled Planning
Depth-bounded Search

\begin{align*}
table \ (+, -, \text{min}, +) \\
\text{plan}(S, Plan, Cost, Limit), \text{final}(S) \Rightarrow \text{Plan} = [], \text{Cost} = 0. \\
\text{plan}(S, Plan, Cost, Limit) \Rightarrow \\
\quad \text{action}(S, S1, Action, ActionCost), \\
\quad \text{Limit1} = \text{Limit} - \text{ActionCost}, \\
\quad \text{Limit1} \geq 0, \\
\quad \text{plan}(S1, Plan1, Cost1, Limit1), \\
\quad \text{Plan} = [\text{Action}|\text{Plan1}], \\
\quad \text{Cost} = \text{Cost1} + \text{ActionCost}.
\end{align*}

- Pass the resource limit as an input argument
- Calls with the same state and different resource limits are no longer variants
Tabled Planning
Depth-bounded Search

\[
\text{table } (+, -, \text{min}, nt) \\
\text{plan}(S, \text{Plan}, \text{Cost}, \text{Limit}), \text{final}(S) \Rightarrow \text{Plan}=[], \text{Cost}=0. \\
\text{plan}(S, \text{Plan}, \text{Cost}, \text{Limit}) \Rightarrow \\
\quad \text{action}(S, S1, \text{Action}, \text{ActionCost}), \\
\quad \text{Limit1} = \text{Limit} - \text{ActionCost}, \\
\quad \text{Limit1} \geq 0, \\
\quad \text{plan}(S1, \text{Plan1}, \text{Cost1}, \text{Limit1}), \\
\quad \text{Plan} = [\text{Action}|\text{Plan1}], \\
\quad \text{Cost} = \text{Cost1} + \text{ActionCost}.
\]

- Pass the resource limit as an \textit{nt} argument
  - Once a call is completed with a failure, it will fail forever, no matter how big the resource limit is.
  - Soundness and completeness are not guaranteed.
Resource-Bounded Search

- Special treatment of the resource limit argument
  - It is tabled but not used in variant checking

$S^R$ is the current node, where $S$ is the state and $R$ is the resource limit. $S^{R'}$ failed before. $S^R$ can be failed immediately if $R \leq R'$. 
Outline of Tutorial

- An overview of Picat [20m]
- Tabling in Picat [10m]
- Planning with Picat [50m]
  - The planner module of Picat
  - Modeling techniques
  - Modeling examples
- Summary
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 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)
  - `current_resource()`
  - `current_plan()`

- **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.
  - final(State,Plan,Cost)
    - True if a goal state can be reached from State by Plan with Cost.
- Specify the actions
  - action(State,NextState,Action,ActionCost)
    - Encodes the state transition relation
    - States are tabled, and destructive updates of states (using :=) are banned.
- 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
Setting for the Experiments

- Linux
- CPU 4-core AMD II X4 945
- 8GM RAM
- Timeout was set to 15 minutes per instance
15-Puzzle

State representation

State = \{SpaceLoc, CurTiles, GoalTiles\},
SpaceLoc = [1|3],
CurTiles = [[4|1],[3|4],...],
GoalTiles= [[1|2],[1|3],...]

Use a cons to represent a location.
A cons \([Row|Col]\) consumes 2 words,
and a pair \((Row,Col)\) consumes 3 words.

The goal state

\[\text{final}([1|1],\text{Tiles},\text{Tiles}) \Rightarrow \text{true.}\]
15-Puzzle

Actions

\[
\text{action}\{P0[R0|C0], Tiles, GTiles, NextS, Move, Cost\} \Rightarrow \\
R1 = R0 - 1, \\
R1 \geq 1, \\
Move = \text{up}, \\
Cost = 1, \\
P1 = [R1|C0], \\
update(Tiles, P0, P1, NTiles), \\
\text{manhattan\_distance}(NTiles, GTiles) < \text{current\_resource}(), \\
\text{NextS}\{P1, NTiles, GTiles\}.
\]

\ ...

NOTE: Tabled data cannot be destructively updated using the := operator!
15-Puzzle

Experimental Results
- 15 instances from ASP’09 that require 30-45 steps were used.
- Picat solved all 15 instances in less than 1s per instance.
  - The Manhattan distance heuristic is important.
- Prolog using the same heuristic is over 100 times slower than Picat
  - Tabling is important.
- Potassco (Clasp) failed to solve 5 of the 15 instances.
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}.

...
Rush Hour Puzzle

Experimental Results

- Picat found a best plan of 81 steps in 2s (explored 30493 states)
- ProB, with breadth-first search, took about the same amount of time to find a best plan.
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

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

Tabling for Planning, N.F. Zhou
Sokoban

- Actions

% push a goal stone
action({SoLoc, GStLocs, NonGStLocs}, NextState, Action, Cost) ?=>
   NextState = {NewSoLoc, NewGStLocs, NonGStLocs},
   Action = \texttt{move\_push}(SoLoc, StLoc, StDest, Dir),
   Cost = 1,
   choose\_goal\_stone(Dir, SoLoc, NewSoLoc, GStLocs, StLoc, 
   StDest, GStLocs1, NonGStLocs),
   NewGStLocs = \texttt{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) =>
   ...

Tabling for Planning, 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

\[
\{\text{CurLoc}|\text{TargetLoc}, \text{ORobotLocs}\}
\]

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

- Goal states

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

- **Use heuristics**

  ```prolog
  action([From|To],ORobotLocs),NextState,Action,Cost) ?=>
  NextState = [[Stop|To],ORobotLocs],
  Action = [From|Stop], Cost = 1,
  choose_move_dest(From,ORobotLocs,Stop),
  current_resource() > heuristic_val(NextState).
  ```

  ```prolog
  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),
  current_resource() > heuristic_val(NextState).
  ```

- The current state is at least three steps away from the final state if the target robot is not in the same row or the same column, and the target position has no obstacle around it.
Ricochet Robots

- Experimental results
  - 30 instances of $16 \times 16$ used in ASP’13 were used.
  - Picat solved all 30 instances
    - On average 9s per instance when no heuristic was used.
    - On average 2s per instance when the heuristic was used.
  - Prolog struggles on $5 \times 5$ instances.
    - Tabling is important.
  - Potassco also solved all 30 instances
    - On average 49s per instance.
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

\text{final}({\_},[[],[]]) \Rightarrow \text{true.}
Nomystery

- **Actions**

  ```
  action({Loc, LCGs, WCGs}, NextState, Action, Cost),
  select(Loc, LCGs, LCGs1)
  =>
  Action = $unload(Loc),
  Cost = 0,
  NextState = {Loc, LCGs1, WCGs}.
  ``

  ```
  action({Loc, LCGs, WCGs}, NextState, Action, Cost),
  select([Loc|CargoDest], WCGs, WCGs1)
  =>
  Action = $load(Loc, CargoDest),
  Cost = 0,
  NextState = {Loc, LCGs1, WCGs1},
  LCGs1 = insert_ordered(LCGs, CargoDest).
  ``

  ```
  action({Loc, LCGs, WCGs}, NextState, Action, Cost) =>
  Action = $drive(Loc, Loc1),
  NextState = {Loc1, LCGs, WCGs},
  fuelcost(Cost, Loc, 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

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

\[
\text{final}(\text{shuffled}([C1,C2,C3,C4,C5,C6,C7,C8|_])) \Rightarrow
\]

\[
\text{Suites} = [c,d,h,s],
\]

\[
(\text{sort}([C1,C2,C3,C4]) \neq \text{Suites} ;
\]

\[
\text{sort}([C5,C6,C7,C8]) \neq \text{Suites}
\].
Gilbreath’s Card Trick

Actions

```
action(init(Cards),NewS,Action,ActionCost) =>
  NewS = $splitted(Deck1,RDeck2),
  Action = split,
  ActionCost = 1,
  append(Deck1,Deck2,Cards),
  Deck1 !== [],
  Deck2 !== [],
  RDeck2 = Deck2.reverse().
```

```
action(splitted(Deck1,Deck2),NewS,Action,ActionCost) =>
  NewS = $shuffled(Cards),
  Action = shuffle,
  ActionCost = 1,
  shuffle(Deck1,Deck2,Cards).
```
Gilbreath’s Card Trick

- Experimental results

<table>
<thead>
<tr>
<th>#Cards</th>
<th>#States</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>13,840</td>
<td>0.08s</td>
</tr>
<tr>
<td>20</td>
<td>165,908</td>
<td>1.43s</td>
</tr>
<tr>
<td>24</td>
<td>1,990,680</td>
<td>34.22s</td>
</tr>
</tbody>
</table>
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**
  
  \[
  \text{pieces}(E_s, C_s) \]
  
  \[
  \begin{align*}
  E_s & : \text{A list of positions of edge pieces.} \\
  & \text{Edge positions: } [bd, db, \ldots, ru, ur]. \\
  C_s & : \text{A list of positions of corner pieces.} \\
  & \text{Corner positions: } [bdl, bld, \ldots, ufr, urf]
  \end{align*}
  \]

- **The goal state**
  
  \[
  \text{final}(\text{pieces}(E_s, C_s)) \Rightarrow \\
  \begin{align*}
  E_s & = [bd, bl, br, bu, df, dl, dr, fl, fr, fu, lu, ru], \\
  C_s & = [bdl, bdr, blu, bru, dfl, dfr, flu, fru].
  \end{align*}
  \]
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>7,768,485,393,179,528</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

\[
\text{final}(S, \text{Plan}, \text{Cost}) \Rightarrow \\
m = \text{get_table_map}(), \\
m.get(S, []) = (\text{Plan}, \text{Cost}).
\]
Rubik’s Cube

- **Actions**

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

- **Some domain knowledge**
  - Do not turn one face consecutively.
  - Do not turn opposite faces consecutively.
Rubik’s Cube

Experimental results

- 2x2x2
  - 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.

- 3x3x3
  - 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
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

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

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

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

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

```prolog
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,C State,G State},Plan,Cost) =>
    nth(From,C State,[_]),
    nth(To,G State,[_]),
    Plan = [move(From,To)],
    Cost = 1.
% divide the problem into sub-problems
hanoi4({N,C State,G State},Plan,Cost) =>
    partition_disks(N,C State,G State,It State,M,Peg),   % set up a landmark
    remove_larger_disks(C State,M) = C State1,
    hanoi4({M,C State1,It State},Plan1,Cost1),   % sub-problem1
    remove_smaller_or_equal_disks(C State,M) = C State2,
    remove_smaller_or_equal_disks(G State,M) = G State2,
    N1 is N-M,
    hanoi3({N1,C State2,G State2,Peg},Plan2,Cost2),   % sub-problem2, 3-peg version
    remove_larger_disks(G State,M) = G State1,
    hanoi4({M,It State,G State1},Plan3,Cost3),   % sub-problem3
    Plan = Plan1 ++ Plan2 ++ Plan3,
    Cost = Cost1 + Cost2 + Cost3.
```
Hanoi Tower (4 Pegs)

- 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 is 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)
Summary
Picat Vs. PDDL & ASP

- **Picat**
  - Structures can be used to represent states
  - Tabled state-space search (based on IDA* but different)
  - Rely on programmers to encode domain knowledge
    - Heuristics, control, determinism, dependency, and symmetry

- **PDDL & ASP**
  - States are represented as propositional facts
  - Various kinds of algorithms for PDDL and SAT for ASP
  - Rely on the solver to learn domain knowledge
References

- R. Barták and N.F. Zhou, Using Tabled Logic Programming to Solve the Petrobras Planning Problem, TPLP 2014.


