Planning as Model Checking Tasks

Yi Li \(^1\)  
Jing Sun \(^2\)  
Jin Song Dong \(^3\)  
Yang Liu \(^3\)  
Jun Sun \(^4\)

\(^1\)University of Toronto

\(^2\)The University of Auckland

\(^3\)National University of Singapore

\(^4\)Singapore University of Technology and Design

Oct 13, 2012
1 Introduction
   - Two Problems
   - Motivation

2 Experiments

3 PAT as Planning Service
   - Case Study: Transport4You
   - Route Planning Model Design

4 Future Work
Two Problems

Model Checking

Given a system model $\mathcal{M}$, an initial state $s_0$, and a formula $\varphi$ which specifies the property, Model Checking can be viewed as $\mathcal{M}, s_0 \models \varphi$.

Planning

Classical Planning is defined as a three-tuple $(S_0, G, A)$ where $S_0$ represents the initial state, $G$ represents the set of goal states and $A$ represents a finite set of deterministic actions.

Intuition: construct a safety property $G \neg \varphi$ that requires the formula $\varphi$ never to hold.
Performance of model checkers are comparable to that of the state-of-the-art planners.

Domain specific control knowledge can be exploited to improve the performance of model checkers on planning problems.

Model checkers are good at handling large state spaces.

Model checking can be used as underlying planning service for upper layer applications.
Tools

- PAT: Process Analysis Toolkit (demo)
- NuSMV: an extension of the symbolic model checker SMV
- Spin: established model checker, modeling language Promela similar to CSP# of PAT
- SatPlan: an award winning planner for optimal deterministic planning
- Metric-FF: domain independent planning system
The sliding game problem

- The *8-tiles problem* is the largest puzzle of its type that can be completely solved.
- The game is simple, and yet obeys a combinatorially large problem space of $9!/2$ states.
- The $N \times N$ extension of the problem is NP-hard.
The sliding game problem cont’d

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>(a)</td>
<td>Hard1</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>(b)</td>
<td>Hard2</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(c)</td>
<td>Most1</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(d)</td>
<td>Most2</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>(e)</td>
<td>Rand1</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>(f)</td>
<td>Rand2</td>
<td></td>
</tr>
</tbody>
</table>

Figure: Initial configurations of the sliding game problem instances
Experimental Results

Figure: Execution time comparison of PAT, NuSMV and SatPlan on the sliding game problem, shown on a logarithm scale.
Case Study: Transport4You

- Transport4You won the Formal Methods Award in SCORE contest out of 56 submissions.
- Presented at ICSE 2011 in Hawaii.
- Specifically designed municipal transportation management solution.
- Simplify the fare collection process and provide customized services to subscribers.
Route Planning Module

Figure: System architecture diagram of the “Transport4You” IPTM system
Figure: Simulator architecture diagram
Why Using PAT?

- The searching algorithms of PAT is highly efficient and ready to be used out-of-box.
- CSP# is a highly expressive language for modeling various kind of systems.
- PAT is constructed in a modularized fashion. Modules for specific purposes can be built to give better support for the domains that are considered.
A Route Planning task is defined by a 5-tuple \((S,B,t,c,L)\) with the following components:

- **S**: the set of bus stops
- **B**: the set of bus lines
- **\(t : S \rightarrow B_S\)**: a function mapping \(s\) to the set of available bus lines at stop \(s\)
- **\(c : S \rightarrow S\)**: the stop one can get to by crossing the road at stop \(s\).
- **\(L(s)\)** is true when the current location of user is at stop \(s\).
A Route Planning problem is mapped to a classical planning problem as follows:

**States:** Each state is represented as a literal $s \in S$, where $L(s)$ holds.

**Initial State:** $s_0$

**Goal States:** $s_g$

**Actions:**
1. $(\text{TakeBus}(b_i, s))$,  
   PRECOND: $b_i \in t(s)$,  
   EFFECT: $\neg L(s) \land L(b_i(s))$

2. $(\text{Cross}(s))$,  
   PRECOND: $s \in \text{dom}(c)$,  
   EFFECT: $\neg L(s) \land L(c(s))$
enum {TerminalA, Stop5, Stop7, Stop9 ... Stop26, Stop11, Stop35, Stop34};

var sLine1 = [TerminalA, Stop5, Stop7, Stop9, Stop58, Stop31, Stop33, Stop53, Stop57, TerminalC];
var <BusLine> Line1 = new BusLine(sLine1,1);
var sLine2 = [TerminalC, Stop56, Stop52, Stop32, Stop30, Stop59, Stop10, Stop8, Stop6, TerminalA];
var <BusLine> Line2 = new BusLine(sLine2,2);
...

var sLine14 = [TerminalC, Stop34, Stop32, Stop30, Stop16, TerminalB];
var <BusLine> Line14 = new BusLine(sLine14,14);
Initial State

```javascript
var currentStop = Stop5;
var B0 = [-2];
var <BusLine> currentBus = new BusLine(B0,-1);
```

Transition Functions

```javascript
takeBus() = case {
  currentStop == TerminalA: BusLine1[] BusLine3[] BusLine5[] BusLine7
  currentStop == Stop5: BusLine1[] BusLine5
  currentStop == Stop7: BusLine1[] BusLine5
  ...
  currentStop == Stop11: BusLine12
  currentStop == Stop35: BusLine13
  currentStop == Stop34: BusLine14
};
```
Transition Functions

\( \text{BusLine1} = \)
\( \text{TakeBus.1}\{\text{currentStop} = \text{Line1.NextStop}(\text{currentStop}); \}
\( \text{currentBus} = \text{Line1}; \} \rightarrow \text{plan}; \)
\( ... \)

\( \text{BusLine14} = \)
\( \text{TakeBus.14}\{\text{currentStop} = \text{Line14.NextStop}(\text{currentStop}); \}
\( \text{currentBus} = \text{Line14}; \} \rightarrow \text{plan}; \)

\( \text{crossRoad()} = \text{case}\{ \)
\( \text{currentStop} == \text{Stop5}: \text{crosscurrentStop} = \text{Stop6} \rightarrow \text{plan} \)
\( \text{currentStop} == \text{Stop7}: \text{crosscurrentStop} = \text{Stop8} \rightarrow \text{plan} \)
\( ... \)
\( \text{currentStop} == \text{Stop35}: \text{crosscurrentStop} = \text{Stop34} \rightarrow \text{plan} \)
\( \text{currentStop} == \text{Stop34}: \text{crosscurrentStop} = \text{Stop35} \rightarrow \text{plan} \}
\);
Transition Functions

\[
plan = \text{takeBus()}[] \text{crossRoad()};
\]

Goal States

\[
\text{#define goal currentStop==Stop53;}
\]
Modified Transition Functions

- \( \text{takeBus}()=\tau\{\text{cost} = \text{cost} + 10\}\rightarrow\text{case}\{\ldots\} \)
- \( \text{crossRoad}()=\tau\{\text{cost} = \text{cost} + 2\}\rightarrow\text{case}\{\ldots\} \)
- \( \text{BusLine1}=\tau\{\text{if}(!\text{currentBus isEqual}(\text{LineX}))\{\text{cost} = \text{cost} + 5\}\} \rightarrow \text{TakeBus.1}\ldots \)

- New assertion: \#assert plan reaches goal with min(\text{cost});
- \( \text{cost} = 10 \times \#\text{takeBus} + 5 \times \#\text{crossRoad} + 2 \times \#\text{busChange} \)
- Original problem can be solved by a simple breadth-first search.
- To find the goal state with minimum \text{cost}, the whole state space has to be searched?
Algorithm 1 newBFSVerification()

initialize queue: working;

\[ current \leftarrow \text{InitialStep}; \tau \leftarrow \infty; \]

repeat

value \leftarrow \text{EvaluateExpression}(current);

if \( current.\text{ImplyCondition}() \) then

if value < \( \tau \) then

\( \tau \leftarrow \text{value}; \)

end if

end if

if value > \( \tau \) then

continue;

end if

for all step \( \in current.\text{MakeOneMove}() \) do

working.\text{Enqe}(step);

end for

until working.\text{Count}() \leq 0
Search Space Pruning

Figure: An example bus line configuration

Figure: A solution produced by the basic model
Search Space Pruning cont’d

Given the current bus line is $b_k$, an action $\text{TakeBus}(b_i, s_j)$ is not redundant if one of the followings holds:

1. $b_i = b_k$
2. $b_i \in t(s_j) \land b_k \in t(s_j) \land b_i(s_j) \neq b_k(s_j) \land \exists m \in \mathbb{N}_1, b_i(s_j)^{-m} \neq b_k(s_j)^{-m}$
3. 1 and 2 do not hold and $b_i(s_j) \neq b_k(s_j) \land b_i^{-1}(s_j) \neq b_k^{-1}(s_j)$

**Figure:** Special pattern of two overlapping bus lines
Search Space Pruning cont’d

(a) Same Previous Stop

(b) Same Next Stop

**Figure:** Redundant bus changes
Future Work

- Extend the comparisons to a larger range of model checking as well as planning tools.
- By fine tuning the way of modeling or exploiting domain specific knowledge, some models can be further optimized.
- An automated translator for the translation from PDDL to CSP# can be implemented.
- The applications of PAT as planning service should be extended to a larger range on real problems in various fields.
The End