Model Checking as Planning and Service

B.Comp. Dissertation

Li Yi

Department of Computer Science
School of Computing
National University of Singapore

April 19, 2011

Outline

Outline

Two Problems

Model Checking

Given a system model $\mathcal{M}$, an initial state s_0, and a formula φ 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.

Two Problems

Model Checking

Given a system model $\mathcal{M}$, an initial state s_0, and a formula φ 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 $\mathbf{G}\neg\varphi$ that requires the formula φ never to hold, such that the model checker is able to search for a counterexample that leads to a state where φ holds.

Motivation

- Research shows the performance of model checkers are comparable to that of the state-of-the-art planners.

Motivation

- Research shows the 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.

Motivation

- Research shows the 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, which possibly implies better performance on real world problems compared with planners.

Motivation

- Research shows the 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, which possibly implies better performance on real world problems compared with planners.
- Model checking can be used as underlying planning service for upper layer applications.

Outline

Tools

PAT

- CSP# is an expressive model description language combining high-level compositional operators with program-like codes.
- Self-defined C# Libraries provide unlimited potentials on modelling complex data operations and data types.
- Flexible and modularized framework allows users to build customized model checking modules for specific domains.

Tools

PAT

- CSP# is an expressive model description language combining high-level compositional operators with program-like codes.
- Self-defined C# Libraries provide unlimited potentials on modelling complex data operations and data types.
- Flexible and modularized framework allows users to build customized model checking modules for specific domains.

NuSMV

- NuSMV is an extension of the symbolic model checker SMV.
- Models are described as transition relations between current and next state pairs: `next(identifier):=expression`.
- Specifications can be expressed in both CTL and LTL.
- Array indices in NuSMV must be statically evaluated to integer constants.

Tools cont'd

Spin

- Spin models are described in a modelling language called Promela that loosely follows CSP and hence models in CSP# can be converted with minimal efforts.
- The counterexamples produced by Spin are not guaranteed to be in the minimum size.

Tools cont'd

Spin

- Spin models are described in a modelling language called Promela that loosely follows CSP and hence models in CSP# can be converted with minimal efforts.
- The counterexamples produced by Spin are not guaranteed to be in the minimum size.

SatPlan

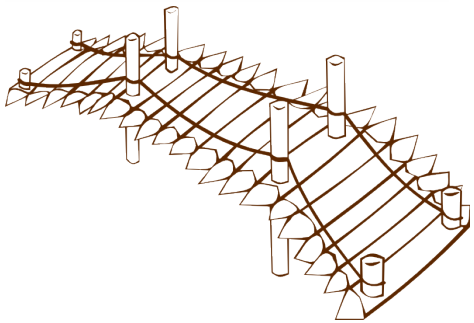
- SatPlan is an award winning planner for optimal deterministic planning created by Prof. Henry Kautz, Dr. Jörg Hoffmann and Shane Neph.
- SatPlan encodes the planning problem into a SAT formulation with length k and check the satisfiability using a SAT solver.
- The optimality of plan is restricted to the solution length or make-span.

Tools cont'd

Metric-FF

- Metric-FF is a domain independent planning system developed by Dr. Jörg Hoffmann.
- Numerical plan metrics and optimization criteria are allowed.
- Two parameters h and g can be customized to assign priorities to either speed or quality.
- Standard weighted A* search is used to speed up searching, thus the optimality is not guaranteed.

The bridge crossing problem



- The bridge has been damaged and can only carry two soldiers at a time.
- The soldiers only have a single torch which is needed when crossing the bridge.
- The time needed for each soldier are 5, 10, 20, 25 minutes respectively.
- The goal is to find a solution to get all the soldiers to cross the bridge to safety in 60 minutes or less.

The bridge crossing problem cont'd

The bridge crossing problem is a plan existence problem with a constraint on the total time.

The bridge crossing problem cont'd

The bridge crossing problem is a plan existence problem with a constraint on the total time.

We extend the original problem to versions with up to 9 soldiers:

Soldier	1	2	3	4	5	6	7	8	9
Time Cost	5	10	20	25	30	45	60	80	100

Table: Time cost of each soldier

Experimental Results

#	Time*	Metric-FF	PAT		NuSMV			Spin
			WITH	DFS	INVAR	CTL	LTL	
4	60	0.00	0.05	0.04	0.0	0.1	0.1	0.02
5	90	0.00	0.19	0.04	0.1	0.9	0.4	0.02
6	130	0.03	1.12	0.22	0.2	14.4	2.5	0.06
7	175	0.16	6.18	0.25	0.5	330.8	71.3	0.11
8	235	0.94	33.19	10.26	m	m	m	10.50
9	300	5.30	145.51	16.40	m	m	m	19.50

Table: Experimental results for *the bridge crossing problem*

Experimental Results cont'd

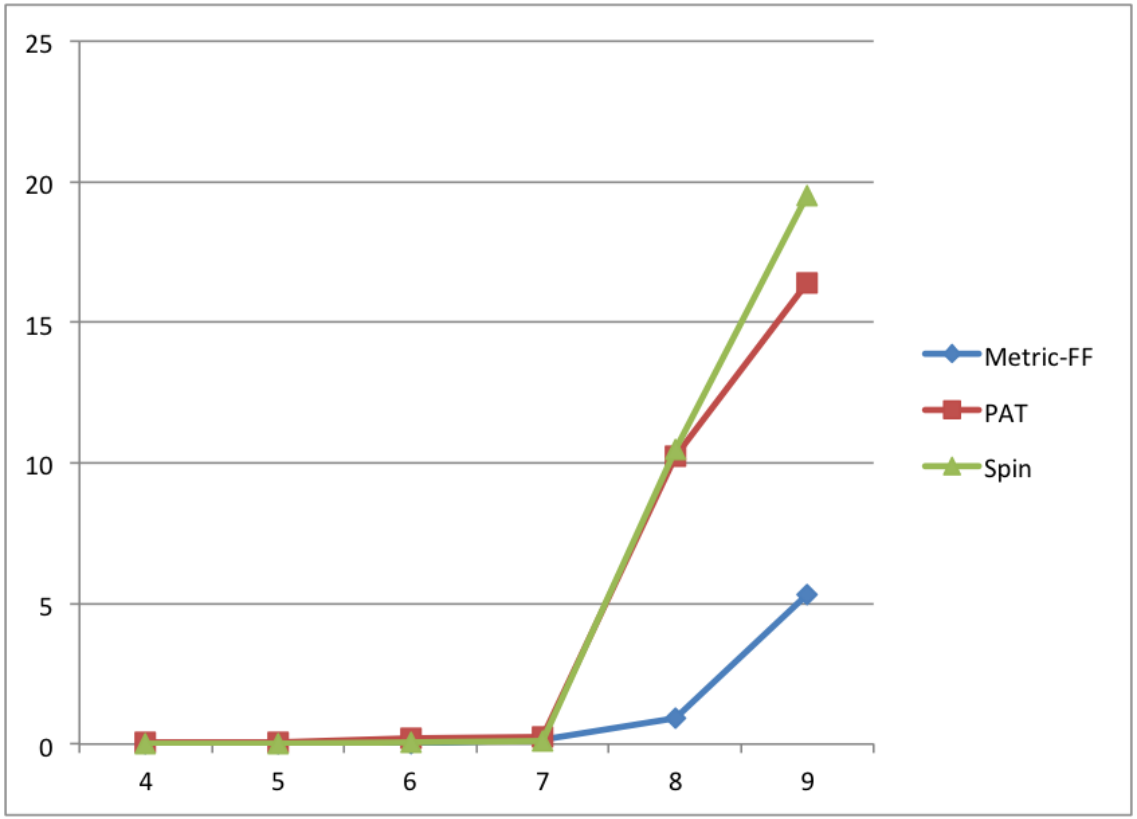
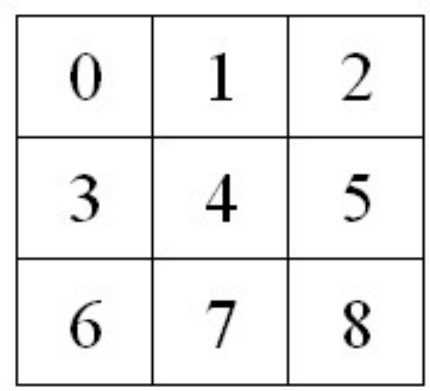


Figure: Execution time comparison of PAT, Spin and Metric-FF on *the bridge crossing problem*

The sliding game problem

- The *sliding game 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

8	7	6
0	4	1
2	5	3

(a) Hard1

8	0	6
5	4	7
2	3	1

(b) Hard2

8	5	6
7	2	3
4	1	0

(c) Most1

8	5	4
7	6	3
2	1	0

(d) Most2

8	2	1
3	6	4
0	5	7

(e) Rand1

4	1	7
8	0	3
5	6	2

(f) Rand2

Figure: Initial configurations of *the sliding game problem* instances

Experimental Results

Problem	L*	H	SatPlan	PAT	NuSMV			Spin
				BFS	INVAR	CTL	LTL	suboptimal
Hard1	31	21	444.42	9.60	45.2	> 600	> 600	2.25
Hard2	31	21	438.34	10.05	41.6	> 600	> 600	2.06
Most1	30	20	152.76	9.84	42.8	> 600	> 600	1.99
Most2	30	20	152.24	10.01	42.0	> 600	> 600	2.47
Rand1	24	12	33.70	7.00	30.0	> 600	> 600	2.63
Rand2	20	16	2.89	3.54	16.8	505.6	> 600	2.13

Table: Experimental results for *the sliding game problem*

Experimental Results cont'd

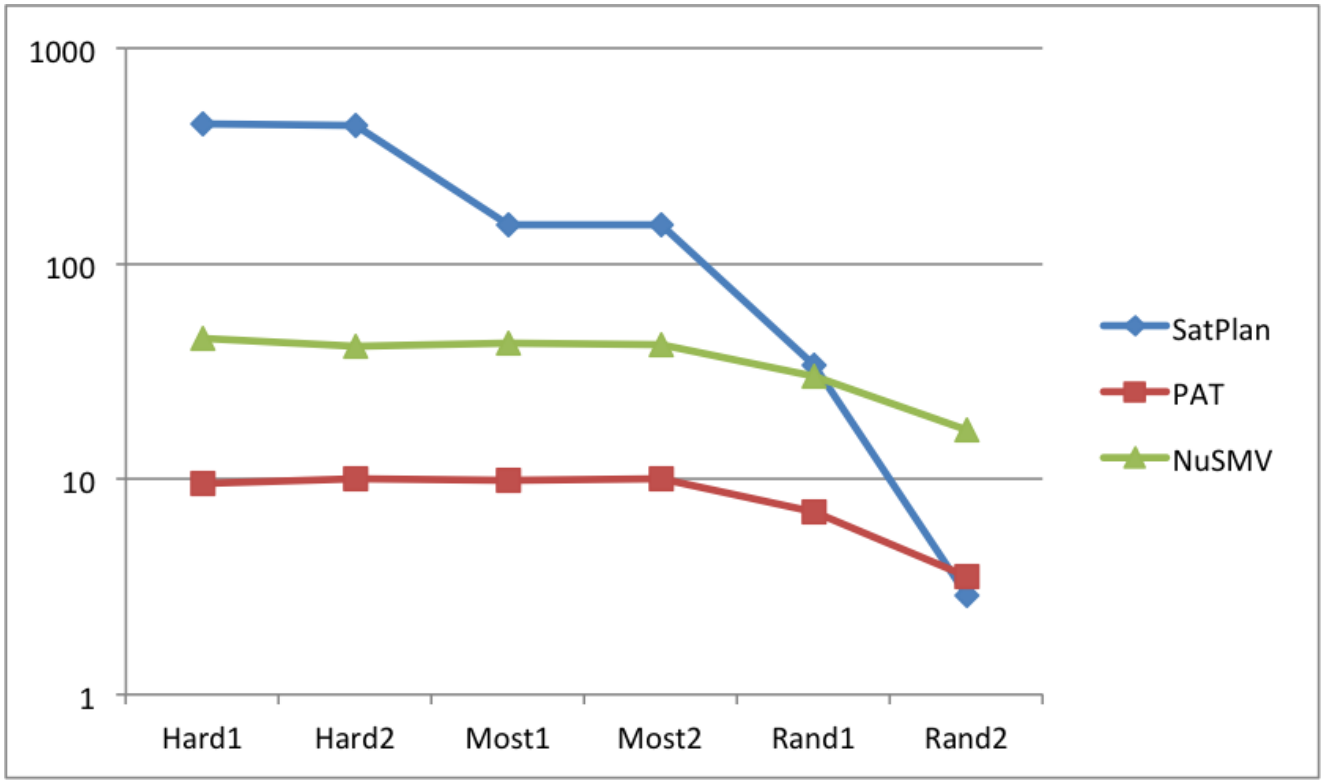


Figure: Execution time comparison of PAT, NuSMV and SatPlan on *the sliding game problem*, shown on a logarithm scale

Comparison of Tools

Tools	Numerical Metrics	bridge crossing		sliding game Optimality
		Existence	Optimality	
PAT	$\surd$	$\surd$	$\surd$	$\surd$
NuSMV	$\surd$	$\surd$	$\times$	$\surd$
Spin	$\surd$	$\surd$	$\times$	$\times$
SatPlan	$\times$	$\times$	$\times$	$\surd$
Metric-FF	$\surd$	$\surd$	$\times$	$\times$

- The counterexamples provided by Spin are not guaranteed the shortest.
- The plans found by Metric-FF are not guaranteed optimal.

From PDDL to CSP#

Assumptions

- The PDDL domain descriptions are written in the subset of PDDL 2.1 that includes STRIPS-like operators with literals having typed arguments and numerical plan metrics.
- The naming and structures of the original PDDL model should be preserved.

From PDDL to CSP#

Assumptions

- The PDDL domain descriptions are written in the subset of PDDL 2.1 that includes STRIPS-like operators with literals having typed arguments and numerical plan metrics.
- The naming and structures of the original PDDL model should be preserved.

The translation process from PDDL to CSP# can be divided into 5 steps:

1. Typing
2. Predicates
3. Initial State
4. Actions
5. Goal

Typing

PDDL

Domain File:

```
(:types place locatable - object
        soldier torch - locatable)
```

Problem File:

```
(:objects
    soldier0 soldier1 soldier2 soldier3 - soldier
    torch - locatable
    north south - place)
```

Typing

PDDL

Domain File:

```
(:types place locatable - object
        soldier torch - locatable)
```

Problem File:

```
(:objects
    soldier0 soldier1 soldier2 soldier3 - soldier
    torch - locatable
    north south - place)
```

CSP#

```
enum {north,south};
enum {soldier0,soldier1,soldier2,soldier3,torch};
```

Predicates

PDDL

```
(:predicates (at ?x - locatable ?y - place))
```

Predicates

PDDL

```
(:predicates (at ?x - locatable ?y - place))
```

CSP#

C# Library:

1. *void setPredicate(predicateName, x, y, value);*
2. *bool tryPredicate(predicateName, x, y);*
3. *int snapShot();*

CSP# File:

```
#import "Predicate";
var<Predicate> pre = new Predicate();

enum {At};
```

Initial State

PDDL

```
(:init (at soldier0 south) (at soldier1 south)
       (at soldier2 south) (at soldier3 south)
       (at torch south)
       (= (time soldier0) 5) (= (time soldier1) 10)
       (= (time soldier2) 20) (= (time soldier3) 25)
       (= (time-cost) 0))
```

Initial State

PDDL

```
(:init (at soldier0 south) (at soldier1 south)
       (at soldier2 south) (at soldier3 south)
       (at torch south)
       (= (time soldier0) 5) (= (time soldier1) 10)
       (= (time soldier2) 20) (= (time soldier3) 25)
       (= (time-cost) 0))
```

CSP#

```
var time[N] = {5,10,20,25};
var time_cost = 0;
ini() = initial{pre.setPredicate(At,soldier0,south,true);
                pre.setPredicate(At,soldier1,south,true);
                pre.setPredicate(At,soldier2,south,true);
                pre.setPredicate(At,soldier3,south,true);
                pre.setPredicate(At,torch,south,true)} -> Skip;
```

Actions

PDDL

```
(:action StoN
 :parameters (?x - soldier ?y - soldier)
 :precondition (and (at ?x south) (at ?y south) (at torch south))
 :effect (and
         (not (at ?x south)) (not (at ?y south))
             (not (at torch south))
         (at ?x north) (at ?y north) (at torch north)
         (when (>= (time ?x) (time ?y))
             (increase (time-cost) (time ?x)))
         (when (< (time ?x) (time ?y))
             (increase (time-cost) (time ?y)))))
```

Actions cont'd

CSP#

```
StoN(x,y) = [x!=y
            && pre.tryPredicate(At,x,south)
            && pre.tryPredicate(At,y,south)
            && pre.tryPredicate(At,torch,south)]
            s.x.y{pre.setPredicate(At,x,north,true);
                  pre.setPredicate(At,x,south,false);
                  pre.setPredicate(At,y,north,true);
                  pre.setPredicate(At,y,south,false);
                  pre.setPredicate(At,torch,north,true);
                  pre.setPredicate(At,torch,south,false);
                  if(time[x]>time[y])
                      {time_cost=time_cost+time[x];}
                  else
                      {time_cost=time_cost+time[y];}
            } -> Trans();
```

Actions cont'd

CSP#

```
Trans() = tau{snap = pre.snapshot()} ->
                  ([] z:{0..3}@([] y:{0..3}@StoN(z,y)))
                []([] x:{0..3}@NtoS(x));
```

Goal

PDDL

```
(:goal (and
          (at soldier0 north) (at soldier1 north)
          (at soldier2 north) (at soldier3 north)))
(:metric minimize (time-cost)))
```

Goal

PDDL

```
(:goal (and
          (at soldier0 north) (at soldier1 north)
          (at soldier2 north) (at soldier3 north)))
(:metric minimize (time-cost)))
```

CSP#

```
#define goal (pre.tryPredicate(At,soldier0,north)
           && pre.tryPredicate(At,soldier1,north)
           && pre.tryPredicate(At,soldier2,north)
           && pre.tryPredicate(At,soldier3,north));

#assert Plan reaches goal with min(time_cost);
```

Outline

Case Study: Transport4You

- "Transport4You" is a project submission for the 33^{rd} International Conference on Software Engineering (ICSE) - Student Contest on Software Engineering (SCORE).

Case Study: Transport4You

- "Transport4You" is a project submission for the 33^{rd} International Conference on Software Engineering (ICSE) - Student Contest on Software Engineering (SCORE).
- It is a specifically designed municipal transportation management solution which is able to simplify the fare collection process and provide customized services to each subscriber.

Case Study: Transport4You

- "Transport4You" is a project submission for the 33^{rd} International Conference on Software Engineering (ICSE) - Student Contest on Software Engineering (SCORE).
- It is a specifically designed municipal transportation management solution which is able to simplify the fare collection process and provide customized services to each subscriber.
- The project is selected as one of the finalists (5 out of 94 submissions) which are going to be presented for the final round of the competition at ICSE 2011 in Hawaii.

Route Planning Module

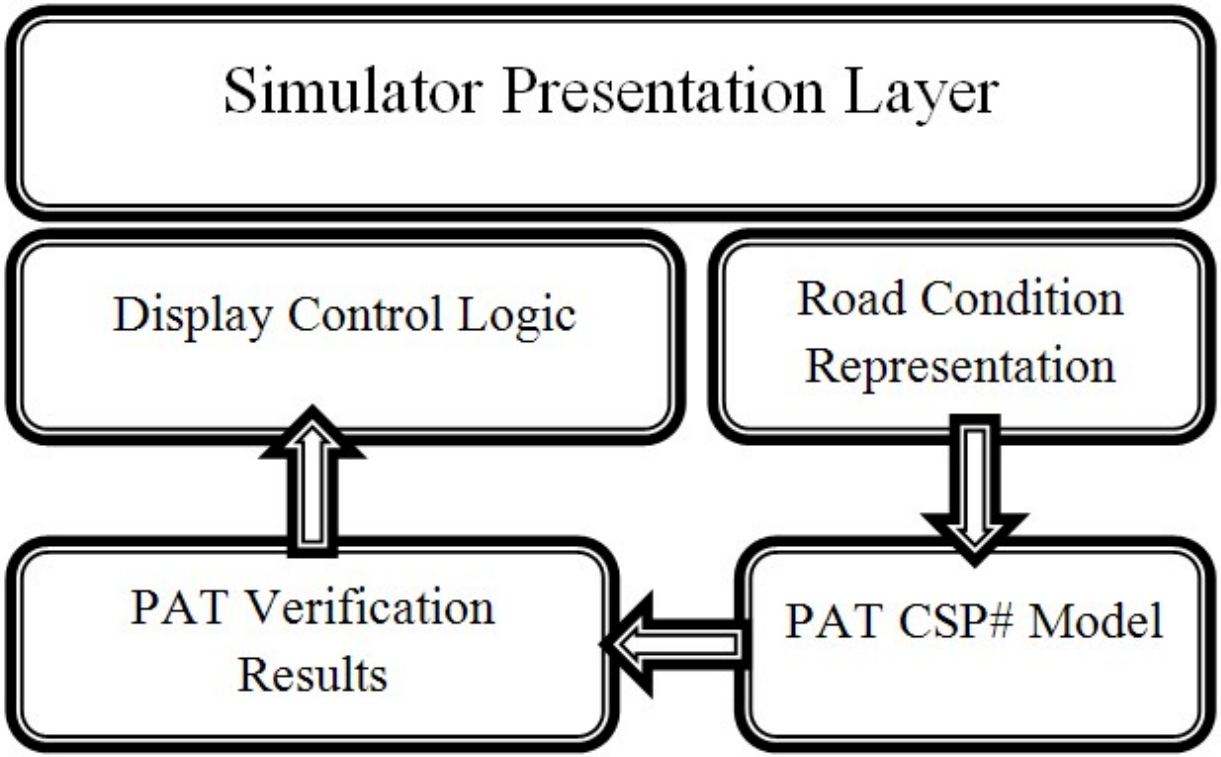


Figure: Simulator architecture diagram

Why Using PAT?

- The searching algorithms of PAT is highly efficient and ready to be used. It also saves the time of implementing a different planning algorithm for every new problem.

Why Using PAT?

- The searching algorithms of PAT is highly efficient and ready to be used. It also saves the time of implementing a different planning algorithm for every new problem.
- CSP# is a highly expressive language for modelling various kind of systems. PAT is ready to solve all kinds of planning problems.

Why Using PAT?

- The searching algorithms of PAT is highly efficient and ready to be used. It also saves the time of implementing a different planning algorithm for every new problem.
- CSP# is a highly expressive language for modelling various kind of systems. PAT is ready to solve all kinds of planning problems.
- PAT is constructed in a modularized fashion. Modules for specific purposes can be built to give better support for the domains that are considered.

Route Planning Demonstration

Definition

A Route Planning task is defined by a 5-tuple (S,B,t,c,L) with the following components:

Definition

A Route Planning task is defined by a 5-tuple (S,B,t,c,L) with the following components:

- S is a finite, non-empty set of bus stops. Terminal stops include start terminal $s_{start} \subseteq S$, and end terminal $s_{end} \subseteq S$, where $s_{start} \cap s_{end} = \emptyset$.

Definition

A Route Planning task is defined by a 5-tuple (S,B,t,c,L) with the following components:

- S is a finite, non-empty set of bus stops. Terminal stops include start terminal $s_{start} \subseteq S$, and end terminal $s_{end} \subseteq S$, where $s_{start} \cap s_{end} = \emptyset$.
- B is a finite set of bus lines, and for every bus line $b_i \in B$, $b_i : S \to S$ is a partial function. $b_i(s)$ is the next stop taking bus i from stop s.
  $\forall s \in s_{start} \forall b \in B, s \in dom(b) \longrightarrow b^{-1}(s) = \alpha$.
  $\forall s \in s_{end} \forall b \in B, s \in dom(b) \longrightarrow b(s) = \beta$.
  $\forall b \in B, b^{-1}(\alpha) = \alpha \wedge b(\beta) = \beta$.

Definition

A Route Planning task is defined by a 5-tuple (S,B,t,c,L) with the following components:

- S is a finite, non-empty set of bus stops. Terminal stops include start terminal $s_{start} \subseteq S$, and end terminal $s_{end} \subseteq S$, where $s_{start} \cap s_{end} = \emptyset$.
- B is a finite set of bus lines, and for every bus line $b_i \in B$, $b_i : S \to S$ is a partial function. $b_i(s)$ is the next stop taking bus i from stop s.
  $\forall s \in s_{start} \forall b \in B, s \in dom(b) \longrightarrow b^{-1}(s) = \alpha$.
  $\forall s \in s_{end} \forall b \in B, s \in dom(b) \longrightarrow b(s) = \beta$.
  $\forall b \in B, b^{-1}(\alpha) = \alpha \wedge b(\beta) = \beta$.
- $t : S \to B_S$ is a function where $B_S \subseteq B$. $t(s)$ is the set of available bus lines at stop s, i.e., $B_S = \{b_i \in B \mid s \in dom(b_i)\}$.

Definition

A Route Planning task is defined by a 5-tuple (S,B,t,c,L) with the following components:

- S is a finite, non-empty set of bus stops. Terminal stops include start terminal $s_{start} \subseteq S$, and end terminal $s_{end} \subseteq S$, where $s_{start} \cap s_{end} = \emptyset$.
- B is a finite set of bus lines, and for every bus line $b_i \in B$, $b_i : S \to S$ is a partial function. $b_i(s)$ is the next stop taking bus i from stop s.
  $\forall s \in s_{start} \forall b \in B, s \in dom(b) \longrightarrow b^{-1}(s) = \alpha$.
  $\forall s \in s_{end} \forall b \in B, s \in dom(b) \longrightarrow b(s) = \beta$.
  $\forall b \in B, b^{-1}(\alpha) = \alpha \wedge b(\beta) = \beta$.
- $t : S \to B_S$ is a function where $B_S \subseteq B$. $t(s)$ is the set of available bus lines at stop s, i.e., $B_S = \{b_i \in B \mid s \in dom(b_i)\}$.
- $c : S \to S$ is a partial function. $c(s)$ is the stop one can get to by crossing the road at stop s.

Definition

A Route Planning task is defined by a 5-tuple (S,B,t,c,L) with the following components:

- S is a finite, non-empty set of bus stops. Terminal stops include start terminal $s_{start} \subseteq S$, and end terminal $s_{end} \subseteq S$, where $s_{start} \cap s_{end} = \emptyset$.
- B is a finite set of bus lines, and for every bus line $b_i \in B$, $b_i : S \to S$ is a partial function. $b_i(s)$ is the next stop taking bus i from stop s.
  $\forall s \in s_{start} \forall b \in B, s \in dom(b) \longrightarrow b^{-1}(s) = \alpha$.
  $\forall s \in s_{end} \forall b \in B, s \in dom(b) \longrightarrow b(s) = \beta$.
  $\forall b \in B, b^{-1}(\alpha) = \alpha \wedge b(\beta) = \beta$.
- $t : S \to B_S$ is a function where $B_S \subseteq B$. $t(s)$ is the set of available bus lines at stop s, i.e., $B_S = \{b_i \in B \mid s \in dom(b_i)\}$.
- $c : S \to S$ is a partial function. $c(s)$ is the stop one can get to by crossing the road at stop s.
- L is a unary predicate on S. L(s) is true when the current location of user is at stop s.

Definition cont'd

Given initial location s_0 and destination s_g, a Route Planning domain maps a Route Planning task to a classical planning problem with close-world assumption 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. $(TakeBus(b_i, s)$,
PRECOND: $b_i \in t(s)$,
EFFECT: $\neg L(s) \wedge L(b_i(s)))$
2. $(Cross(s)$,
PRECOND: $s \in dom(c)$,
EFFECT: $\neg L(s) \wedge L(c(s)))$

Basic Model

Environment Variables

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);

Basic Model cont'd

Initial State

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

Basic Model cont'd

Initial State

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

Transition Functions

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
};

Basic Model cont'd

Transition Functions

BusLine1=
TakeBus.1{currentStop=Line1.NextStop(currentStop);
currentBus=Line1;} ->plan;
...
BusLine14=
TakeBus.14{currentStop=Line14.NextStop(currentStop);
currentBus=Line14;} ->plan;

crossRoad()=case{
currentStop==Stop5: crosscurrentStop=Stop6 ->plan
currentStop==Stop7: crosscurrentStop=Stop8 ->plan
...
currentStop==Stop35: crosscurrentStop=Stop34 ->plan
currentStop==Stop34: crosscurrentStop=Stop35 ->plan
};

Basic Model cont'd

Transition Functions

plan=takeBus()[]crossRoad();

Basic Model cont'd

Transition Functions

plan=takeBus()[]crossRoad();

Goal States

#define goal currentStop==Stop53;

Cost Function Approach

Modified Transition Functions

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

Cost Function Approach

Modified Transition Functions

- *takeBus()*=*tau*{$cost = cost + 10$}->*case*{...
- *crossRoad()*=*tau*{$cost = cost + 2$}->*case*{...
- *BusLine1*=*tau*{*if*(!*currentBus.isEqual*(*LineX*)){$cost = cost + 5$}} ->*TakeBus.1*...

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

Cost Function Approach cont'd

Algorithm 1 newBFSVerification()

initialize queue: *working*;
current $\leftarrow$ *InitialStep*; $\tau \leftarrow \infty$;
repeat
  value $\leftarrow$ *EvaluateExpression*(*current*);
  if *current*.*ImplyCondition*() **then**
    if *value* $< \tau$ **then**
      $\tau \leftarrow$ *value*;
    end if
  end if
  if *value* $> \tau$ **then**
    continue;
  end if
  for all *step* $\in$ *current*.*MakeOneMove*() **do**
    working.*Enque*(*step*);
  end for
until *working*.*Count*() $\leqslant 0$

Search Space Pruning

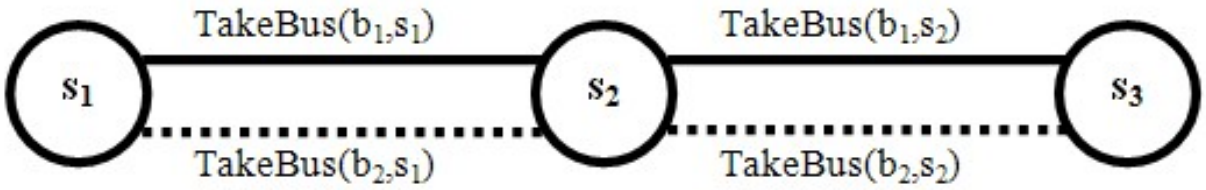


Figure: An example bus line configuration

Figure: A solution produced by the basic model

Search Space Pruning cont'd

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

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

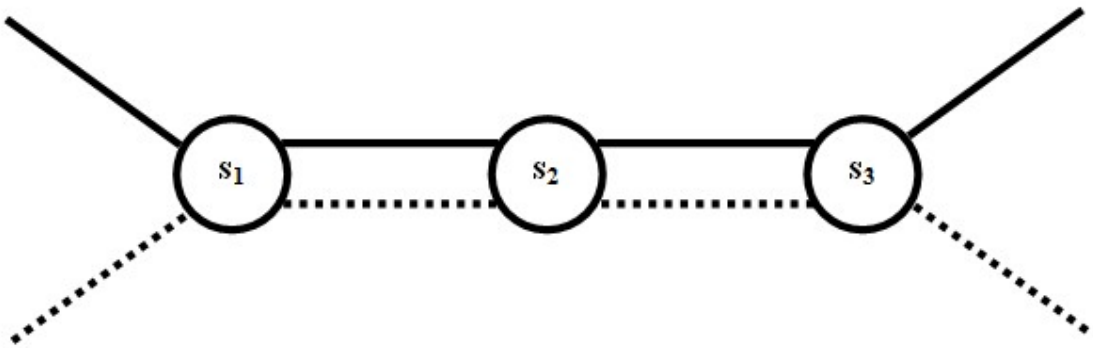
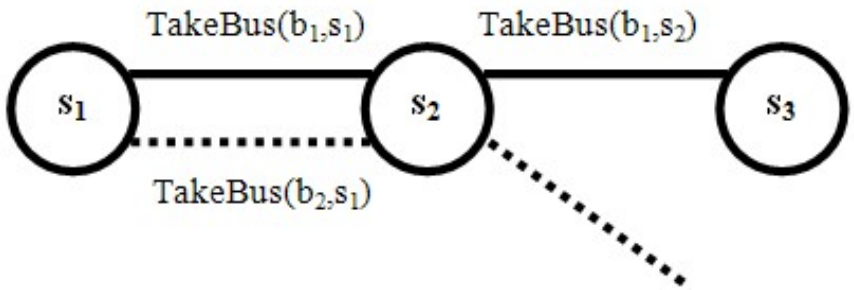
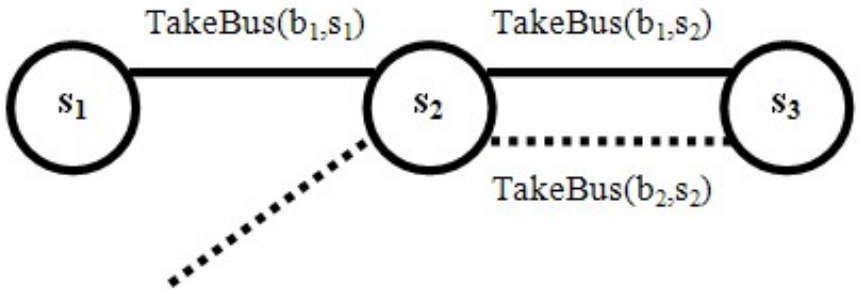


Figure: Special pattern of two overlapping bus lines

Search Space Pruning cont'd



(a) Same Previous Stop



(b) Same Next Stop

Figure: Redundant bus changes

Performance Comparison

	State	Transition	Time	Memory	Cost	Cost'	Length	Length'
Basic	1029.46	1070.93	0.0448	11119.91	58.23	1254	5.51	0
Cost	1125.31	1169.82	0.0483	11281.58	56.02	0	5.59	247
Prune	158.48	185.77	0.0179	9197.95	56.79	379	5.51	0

Table: Comparison results of three route planning models

- Values are average among the 3660 (61×60) test cases.
- The length of the shortest solution was get by solving the shortest path problem using Dijkstra algorithm.
- The *search space pruning* model performs the best in terms of execution time and memory space.
- The *cost function* model guarantees the lowest total cost.

Future Work

- Extend the comparisons to a larger range of model checking as well as planning tools to get a more general view of the subject.

Future Work

- Extend the comparisons to a larger range of model checking as well as planning tools to get a more general view of the subject.
- By fine tuning the way of modelling or exploiting domain specific knowledge, some models can be further optimized.

Future Work

- Extend the comparisons to a larger range of model checking as well as planning tools to get a more general view of the subject.
- By fine tuning the way of modelling or exploiting domain specific knowledge, some models can be further optimized.
- An automated translator for the translation from PDDL to CSP# can be implemented.

Future Work

- Extend the comparisons to a larger range of model checking as well as planning tools to get a more general view of the subject.
- By fine tuning the way of modelling 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