Domain Recompilation-based Approach towards Hierarchical Task Network Planning with Constraints

Plaban Kr Bhowmick, Debnath Mukherjee, Prateep Misra[1] and Anupam Basu[2]

Abstract. In this paper, we present a domain recompilation-based approach towards specifying and handling state-trajectory constraints in Hierarchical Task Network (HTN) planning paradigm. The proposed constraint specification language is inspired by the PDDL3.0 constructs. The domain-recompilation technique is based on automata templates for the PDDL modal operators. To implement automata embedding we have introduced conditional effect construct in HTN. Introduction of dead automata state has helped in reducing amount of backtracking that was required originally. The constraint specification and handling strategy has been tested with a city tour and travel domain modelled using HTN.

1 INTRODUCTION

Design of intelligent environment rests upon three important activities: sense, analyze and respond. The sensors acquire relevant data from the environment, the analyze phase processes the sensor data to generate some contextual information and the respond phase needs to plan the actions for some assigned task given the context information. Thus planning is an important component of intelligent environment as far as the actuation part is concerned. Below we present an example scenario that points out the role of planners in the context of intelligent environment.

Example 1 ***City tour and travel***
A city tour and travel planner provides the travellers with smart plan based on the traveller specified intents and state of the environment. The state of the environment may be described with information related to the Points of Interests (PoIs), traffic updates, weather update and others. The state information can be obtained or updated by implanting related sensors (traffic sensors, rss feeds, twitter feeds etc.) and extracting contexts (blocked road segment, accident, inclement weather etc.) out of the sensor data.

To deploy planners in practical settings, they need to be scalable a feature that is lacking in most of the "first principle" planners [9]. Hierarchical Task Network (HTN) planning [4] paradigm handles the scalability issue by consuming knowledge regarding plan search process. This indicates that HTN paradigm is a natural fit to the domains where a planning task can be achieved through some standard process modules.

With the constraint specification standard set in PDDL3.0 [5], research in constraint-based planning has gained a considerable momentum. The classical planners that follow PDDL standard have made progress towards handling constraints imposed on goal states as well as entire set of states visited by a plan. The primary challenges in planning with constraints are

- processing of the constraint expressions for the consumption by the planning algorithm, and
- designing planning algorithm to handle constraints.

One of the popular ways to handle constraints is to model planning as a model checking problem [3]. The plan-space defines the model which needs to be validated against the stated constraints. One important decision here is the use of external model checkers which in turn may call for the adaptation of the planning algorithm that does not handle constraints. Another approach that avoids external model checkers and adaptation of planning algorithm is domain recompilation-based technique. In this approach, the effects of constraints are simulated by including them in the planning domain with no changes in the planning algorithm. The external model checker based approaches can handle complex and large number of constraints. However, model checking is expensive in terms of time and space. On the other hand, domain recompilation-based technique is efficient by limiting the scope in terms of complexity and number of constraints.

As compared to its classical counterpart, efforts towards constraint specification and handling in HTN planning is sparse. Thus the constraint or preference specification scheme, the constructs required to implement them and corresponding planning algorithm have not received the required level of attention.

In this work, we aim at incorporating constraint processing feature in HTN planner. This is achieved through extending a state-of-the-art open-source HTN planner, namely, JSHOP2[3] [7]. We have adopted the constraint specification syntax and semantics provided in PDDL3.0. A domain recompilation-based strategy has been adopted to transform state-trajectory constraints into goal constraints. The specific contributions of this paper are as follows:

- Constraint specification language
- Automata template-based approach for conversion of constraint expression to automata
- Constructs required to embed constraint automata into planing domain
- Reduction of backtracking effort

The paper is organized as follows: Section 2 provides background on HTN planning and related works in constraint-based HTN planning. Section 3 presents the constraint specification issues in HTN.

[1] Innovation Labs, Tata Consultancy Services, Kolkata, India, email: {plaban.bhowmick,debnath.mukherjee,prateep.misra}@tcs.com
[2] Department of Computer Science & Engoneering, Indian Institute of Technology Kharagpur, India, email: anupambas@gmail.com

[3] `http://www.cs.umd.edu/projects/shop/`

The constraint compilation technique for HTN is described in section 4. Section 5 describes the implementation details and some results with respect to a city tour and travel domain developed by us.

2 BACKGROUND & RELATED WORKS

Since our primary objective is to incorporate constraint specification in HTN planning paradigm, brief backgrounds on PDDL3.0 constraint specification and HTN planning paradigm are provided in this section followed by some previous efforts towards constraint and preference-based planning.

2.1 PDDL3.0 Constraint Specification

Among other features introduced in PDDL3.0, constraint specification language perhaps is the most important one. Through this language, both the soft and hard constraints can be specified over the goal state as well as entire state trajectory generated by a plan. To represent the *state-trajectory* constraints, some modal operators were introduced in PDDL3.0. The modal operators are of two types: *relative temporal* and *metric temporal*. The relative temporal operators are those that do not have any explicit mention of time unlike the metric temporal operators.

In this paper, we deal with constraints that can be specified with relative temporal modal operators. PDDL3.0 supported relative temporal modal operators are `at end`, `always`, `sometimes`, `at most once`, `sometime after` and `sometime before`. The semantics of the relative temporal modal operators are captured through Linear Temporal Logic (LTL). PDDL3.0 does not support nested modal operators as the nested construct in a constraint may contribute to the exponential growth of the corresponding automata required to process the constraint.

2.2 HTN Planning

The HTN planners differ from the classical notion of planning in various aspects. Firstly, apart from standard classical operators or *primitive tasks*, HTN makes use of *methods* for specifying domain specific plan search knowledge. Secondly, the goal in HTN is presented as a *complex task* whereas classical planning specify goal as a set of propositions.

A *task network* is defined by a set of task nodes (T) and set of precedence relations (P). Each of the task nodes contains a task and a precedence relation establishes a precedence constraint between two tasks. If all the tasks in the task network are primitive then the task network is called primitive.

An HTN domain ($D = (O, M)$) consists of primitive tasks or operators (O) and methods (M). The primitive tasks are *executable* and has precondition, add list and delete list. The complex tasks are decomposed by their corresponding methods. Depending upon different preconditions different branches of decomposition are followed. The decomposed tasks are more simpler than the original one and they again form a network depicting the precedence relations among them.

For a given task, there exists a plan *plan* $\pi = o_1 o_2 \ldots o_n$ if there is a primitive decomposition (T_p) of the initial task network T_0 of the planning problem $\mathcal{P} = (S_o, T_o, D)$ and π is an instance of T_p.

The planning process starts by solving the initial task network which in turn is decomposed into more basic task network until the whole task network becomes primitive (existence of plan) or no more decomposition is possible (non-existence of plan).

2.3 Constraints in HTN Planning

As compared to classical planning, efforts towards specifying constraints or preferences have been started recently. A constraint-based HTN planning is defined as follows

Definition 1 *A constraint-based planning problem is defined as* $\mathcal{P} = (S_0, T_0, D, C)$ *where* C *is the set of constraints. The plan* π *generated for* $\mathcal{P}$ *is valid if all constraints in* C *are satisfied by state trajectory generated by* π.

There have been few efforts towards incorporating PDDL constraint specification in HTN. HTNPLAN-P [11] is an HTN-based planner that can plan with soft constraints or preferences. A best-first search technique and different heuristics have been used in order to generate preferred plans. SCUP [8] is an algorithm to perform web service composition by modelling it as a preference-based HTN planning problem. Other notable non-HTN planning algorithm that can handle PDDL constraint constructs are HPlan-P [1] and SGPLAN [6].

3 PROPOSED CONSTRAINT SPECIFICATION FOR HTN

In order to keep the constraint specification compliant to PDDL standard, the proposed constraint specification borrows PDDL constructs with minor modifications. The constraints are specified in problem description in a separate block. The syntax for constraint specification is as follows

```
(:constraint constraint_name (:modal_operator
operands))
```

A constraint expression starts with a `:constraint` tag followed by the constraint name. The modal operator used to encode the temporal modality is specified followed by the list of arguments required by the modal operator. The modal operators can be anyone of the relative temporal modal operators proposed in PDDL.

Some examples of constrains are as follows:

Example 2 *There should always 3000 amount of cash available to the traveller.*

```
(:constraint c1 (:always ((call > avail-cash
3000))))
```

Example 3 *Sometime in the plan poi1 has to be visited*

```
(:constraint c2 (:eventually (visited poi1)))
```

Example 4 *Payment by credit card will be done at most once*

```
(:constraint c3 (:at-most-once (pay-via
creditCard)))
```

Example 5 *poi7 has to be visited sometime before poi2*

```
(:constraint c4 (:s-before(visited
poi2)(visited poi7))
```

Example 6 *poi7 has to be visited sometime after poi2*

```
(:constraint c5 (:s-after(visited
poi2)(visited poi7))
```

4 COMPILATION OF CONSTRAINTS

After specification, the next step is constraint processing. In this work, we adopted a domain recompilation based technique for processing constraints. A fully automated domain recompilation technique has the following steps.

- Conversion of the constraint expressions into LTL formulae.
- Generating Büchi automata for LTL formulae.
- Embedding the automata into planning problem by changing the problem and domain description.

As modal operator set in PDDL is a closed one and PDDL does not support nesting of operators, the generation of automata from a constraint expression can be simplified by defining automata templates for the modal operators. Thus the conversion process is adapted into the following steps.

- Defining automata templates for modal operators.
- Instantiating transition labels of the automata by operands of the modal operators in the specified constraints.

4.1 Automata Templates

The automata templates are generic representation of the modal operators. An automata template consists of start state, set of transitions and set of accepting states. A transition is described by a source-destination state pair and a transition label. The automata templates are stored as xml specification. An example automata for `at-most-once` is shown in Figure 1.

On encountering a constraint expression, the corresponding automata template is retrieved and the labels of the transitions are instantiated with the logical expressions formed out of the operands in constraint expression.

4.2 Conditional Effects (CE)

After instantiation of automata, the next task is to embed it into planning problem. Thus the transitions has to be translated into a form which the original planner can process. *Conditional effect* is a construct that supports switch-case like syntax in operator definition. Apart from the standard effects, the application of an operator may impose one of multiple effects depending on the branch of precondition that is being satisfied.

Conditional effects are like operators as they have similar components like precondition, delete list and add list. The automata states are represented by state predicates like `(state-C S0)`, `(state-C S1) ... (state-C Sn)`. The plan state and the automata states are synchronized by adding automata state predicate to the current plan state. Thus for a constraint `C`, a plan state will contain no more than one of the automata state predicates. A transition from one state `Sm` to another state `Sn` with label ϕ can be modelled by a CE having

- Precondition: logical conjunction of `(state-C Sm)` and ϕ
- Delete list: current automata state i.e., `(state-C Sm)`, `(accepting-C)` if `Sn` is not an accepting state.
- Add list: next automata state i.e., `(state-C Sn)`, `(accepting-C)` if `Sn` is an accepting state.

The conditional effect form for automata representing the constraint `always (call > avail-cash 3000)` (available cash should be greater than 3000) is presented below.

```
(:when (!cond_c3_0)
     ((state-c3 S0)(avail-cash ?VARC0)(call >
      ?VARC0 3000.0 ))
     ((state-c3 S0))
     ((state-c3 S1)(accepting-c3))
) (:when (!cond_c3_1)
     ((state-c3 S1)(avail-cash ?VARC0)(call >
      ?VARC0 3000.0 ))
     ((state-c3 S1))
     ((state-c3 S1)(accepting-c3))
)
(:when (!cond_c3_2)
     ((state-c3 S1)(avail-cash ?VARC0)(not
      (call > ?VARC0 3000.0 )))
     ((state-c3 S1)(accepting-c3))
     ((state-c3 S2)(dead-c3))
)
(:when (!cond_c3_3)
     ((state-c3 S0)(avail-cash ?VARC0)(not
      (call > ?VARC0 3000.0 )))
     ((state-c3 S0)(accepting-c3))
     ((state-c3 S2)(dead-c3))
)
```

4.3 Automata Simulation

After representing the automata into required construct, one needs to simulate the automata correctly to test the validity of the constraints. This is achieved through the following changes in the original planning problem.

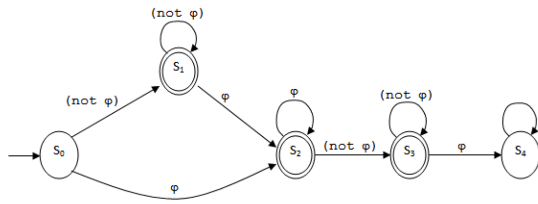
- Start state initialization: The start states of the automata have to be initialized and the corresponding state predicates have to be inserted in the plan state prior to start solving the actual task network. This is achieved through the inclusion of a new operators `!start` in the compiled domain description. The `!start` operator for two constraints is shown below.

  ```
  (:operator (!start)
       ()
       ()
       ((state-c1 S0)(state-c2 S0))
  )
  ```

  It is to be noted that the operator has empty precondition and delete list. The add list consists of the predicates stating that the automata are in start state.
- Probing plan validity: A constraint is satisfied if the corresponding automata is in an accepting state at the end of the plan. For a constraint `C`, the `!finish` operator performs this task by checking whether `(accepting-C)` is there in the final plan state. If so the plan satisfies `C`; otherwise it is violated. An example finish operator is shown below.

  ```
  (:operator (!finish)
       (accepting-c1)(accepting-c2)
       ()
       ()
  )
  ```

- Changes in operators: After applying an operator, some state predicates will be changed which in turn ask for changes in the current states of some automata. As discussed earlier, the transitions can be modelled with CEs. The union of the set of CEs corresponding to the automata for all the constraints will decide the next automata state after applying one action. Thus the union of the CEs




```
<automata>
    <name value="At-Most-Once"></name>
    <start state="S0"></start>
    <transition from="S0" to="S1" label="not X"></transition>
    <transition from="S1" to="S1" label="not X"></transition>
    <transition from="S1" to="S2" label="X"></transition>
    <transition to="S2" from="S2" label="X"></transition>
    <transition from="S0" to="S2" label="X"></transition>
    <transition from="S2" to="S3" label="not X"></transition>
    <transition from="S3" to="S3" label="not X"></transition>
    <transition from="S3" to="S4" label="X"></transition>
    <accept state="S1"></accept>
    <accept state="S2"></accept>
    <accept state="S3"></accept>
    <dead state="S4"></dead></automata>
<automata>
```

Figure 1. Automata template for at-most-once modal operator

are added to the domain operators in compiled domain description.

- Modified initial task network: Apart from the tasks in the original initial task network, two additional tasks `!start` and `!finish` have to be performed. The modified initial task network in the compiled problem description is `(!start)(original initial task network)(!finish)`.
- Removal of constraint block: As the constraints have already been taken care of by the automata, they are removed in the compiled problem description.

5 IMPLEMENTATION AND RESULTS

In this section, we describe the implementing details of constraint processing in existing HTN framework. We study different aspects of constraint-based HTN planning in the purview of a domain called city tour and travel.

5.1 Constraints in JSHOP2

JSHOP2 is a java implementation of Simple Hierarchical Ordered Planner (SHOP2) [10]. It is an open source tool for generating problem specific planner. Current implementation does not have the facility to specify and process constraints. In this work, we have extended JSHOP2 to incorporate PDDL3.0 supported relative temporal operators. Here we describe the implementation details of the said extension. The modifications are as follows:

- Constraint block: A constraint block has been added in the problem description to specify constraints. The constraints are expressed by using modal operators. The original ANTLR[4] JSHOP2 grammar has been modified to add rules for parsing constraint expressions.
- Conditional Effect: JSHOP2 does not support conditional effect construct. Looking at the similarity of the constructs, the CE extension has been implemented by modelling CEs as operators.
- Translation of CEs: The CEs are automatically translated into JSHOP2 operator format and appended into the original operators.
- Inserting `!start` and `!finish`: Depending on the expressed constraints, the `!start` and `!finish` operators are constructed and added to the operator list in domain description.

Next we provide a brief description of HTN domain developed for city tour and travel application and this domain is used to test the proposed extension of JSHOP2.

[4] `http://www.antlr.org/`

5.2 City Tour and Travel Domain

City tour and travel is a service provided by the city authority to aid the prospective tourists with smart travel plans. The travellers may specify their travel intents in terms of points of interest to be visited, other activities. The application in response generates valid plans with respect to the traveller's intents and constraints imposed by him/her. Here, we briefly describe the domain the detail of which is given in [2].

The operators and the methods in the domain description are given in Table 1 and Table 2 respectively.

Table 1. Operators in city tour and travel domain

Operators	Synopsis
`!load-pref`	This is used in loading a traveller intent
`!set-cost`	This is used to set the tour cost based on available amount and estimated cost
`!mark-stay-hotel`	Marking one hotel of traveller's choice and updates time of day
`!lunch`	This is used to represent the fact that the user has taken lunch
`!dinner`	Mark the fact that the traveller has taken dinner
`!set-loc`	The traveller has visited a particular location
`!end-day`	Marks the end of day and reset time of day for the next day

Table 2. Methods in city tour and travel domain

Methods	Synopsis
`ini-pref`	a generic method that loads all the traveller intents
`end-day`	To probe the end of day condition
`find-hotel-go`	To find the hotel that matches traveller specified features
`travel`	To visit the traveller provided points of interest
`find-rest`	To find the restaurant that matches traveller specified features

The state information consists of facts related to hotels and their services, restaurants and information about points of interest. The initial task network consists of one task that specifies choices regarding hotels, restaurants and points of interest as arguments.

5.3 Experimental Setup & Results

The results presented in [2] were based on original JSHOP2. Here, we modify the experimental setup in order to test the implemented extension over JSHOP2. The experiments have been performed on a 2.99 GHz core 2 duo Intel processor with 2GB RAM machine. The problem description consists of 6 hotels having 7 different services (total 15 facts), information about 17 restaurants (17 facts) and information about 28 locations (28 facts). In experimental study, we have considered the following test scenarios.

- Experiment I: Order of PoIs and constraint
- Experiment II: The effect of initial available cash with constraint on PoIs.
- Experiment III: The effect of constraint over traveller's wallet.

Experiment I: In this experiment, we test the performance of the planner based on the setup where the traveller has specified the POIs in some order and placed constraint on a particular PoI that needs to be visited. All the PoIs are placed in order in the initial task network description. Figure 2 depicts the effect of selecting different PoIs and number of days to be planned. The x-axis plots the values of the position of the constrained PoI in the PoI list specified in the initial task network and y-axis plots the number of plan steps[5] for each constrained PoI. This experiment also presents the comparison of planner performance in case of one-day and two-day travel planning.

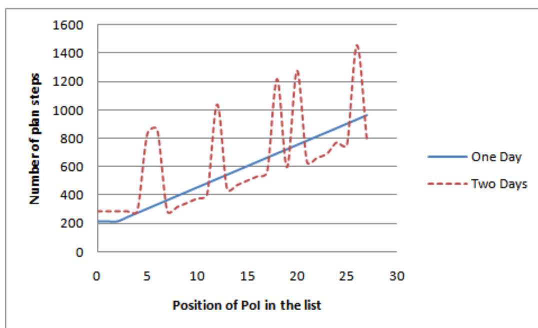


Figure 2. Effect of position of constrained PoI

From the specified list, the number of PoIs selected by the planner is limited by the number of days to be visited and available cash. It is to be noted that in both one-day and two-day plan the number of plan steps increases linearly with the relative position of the constrained PoI with notable spikes in two-day plan. The explanations behind the observations are as follows.

- Linearity: To include the constrained PoI in the plan, the planner needs to backtrack, remove the PoI that was selected last and include the constrained PoI. This takes same amount of steps to be performed for each PoI preceding the constrained PoI. This attributes to the linearity of the plan step size growth.
- Spikes: A spike is observed when a PoI with high cost is selected as constrained PoI. To satisfy the budget, the planner needs to remove more than one PoIs or select some other PoIs to make room for the constrained PoI. Hence the amount of backtracking increases non-linearly.

[5] The total number of steps including backtracking steps required by the planner

Experiment II: In this experiment, we study the effect of initial cash on the number of plan steps. The planner was asked to generate plan for two-day trip with constraint on one particular PoI to be visited. Figure 3 presents the variation of number of plan steps with initial cash available to the traveller.

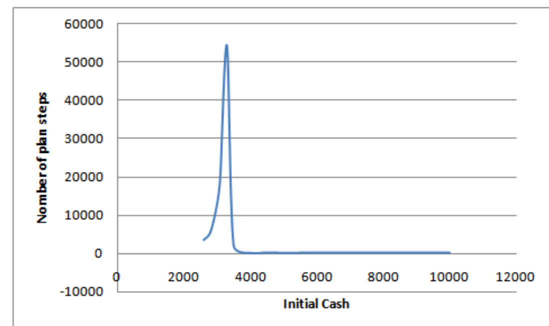


Figure 3. Effect of initial cash and constrained PoI

Similar experiment has been performed on the planner without any constraints. The variation was similar as in Figure 3 with minor decrease in number of plan steps.

In both the cases, no plan was getting generated below a certain limit on initial cash. After that the number of plan steps was increasing in rapid rate with increasing initial cash upto a certain limit and was decreasing again with increasing cash.

Experiment III: In this experiment, we put constraint like 'the traveller should always have greater than 3000 amount of cash available'. With this constraint, the variation in the number of plan steps with initial cash has been studied and presented in Figure 4.

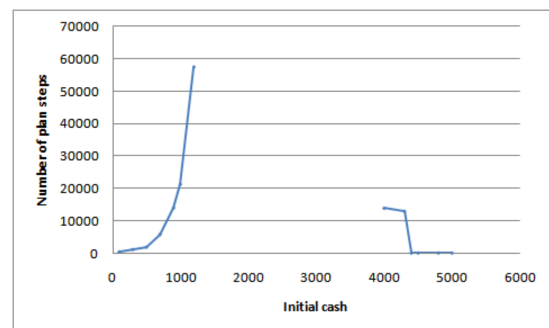


Figure 4. Effect of constraint over traveller's wallet with 'always' modal operator

It is noticed that for smaller and larger amount of cash the planner has been able to generate plan within reasonable time. However, for moderate amount of cash (900 – 4000) planner failed to generate plan with the given computing power. This is due to huge amount of backtracking in case moderate initial cash.

5.4 Avoiding some Pitfalls

In experiment III, the planner should indicate that the constraint is violated for initial cash less than 3000. Instead, the planner back-

tracks enormously with different available options. A close look at the automata for the modal operators reveals an interesting state of the automata which we call as *dead state*. A dead state is defined as the run phase of the automata in which the automata is in a state that does not have any outgoing transition. For example, S_2 is a dead state of the automata for *always* operator (Figure 5).

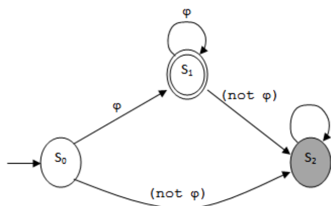


Figure 5. Dead state of *always* automata

For any value less than 3000 the automata will be in a dead state and will never reach the accepting state. Consequently, the acceptance check operator `!finish` will fail and the planner will backtrack with other instantiations of variables, operators and methods.

This huge amount of backtracking can be avoided by making the planner aware of the dead state. Whenever an automata is in a dead state `(dead-C)` predicate is added to the aggregated state information. The precondition of the `!finish` operator for a constraint `C` is changed to `(or (accepting-C)(dead-C))`.

The improvement in performance of the planner after the inclusion of dead state is shown in Figure 6.

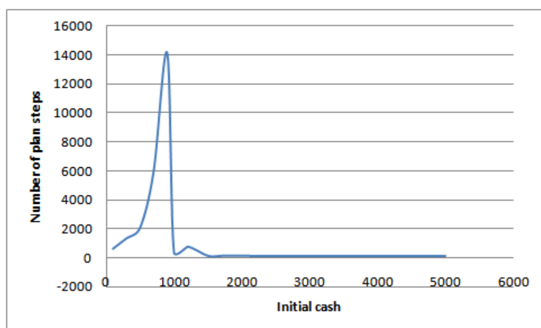


Figure 6. Performance improvement due to inclusion of dead state

6 Discussion

In this paper, we have adopted a domain recompilation-based technique towards extending HTN planner for handling temporal constraints. The constraints are processed by representing them into automata corresponding to the modal operators used to express them. The resulting automata have been embedded into the original planning domain thus generating new domain and problem descriptions that the original planner can consume without the knowledge of the existence of any constraints. We have tested different aspects of the constraint-based HTN planner with a city tour and travel domain. Different issues and limitations of the proposed extension are discussed below.

- The automata for the modal operators have been specified through generic templates. This limits the scope of expressive power of the constraint specification. The inclusion of expressions involving nested modal constructs calls for automatic translation of constraint expressions into automata.
- The present extension deals with only untimed (LTL) category of the modal operators where the automata template-based constraint processing is feasible. However, the constraint expression involving the timed modal operators cannot be captured through the current scheme.

We will take up the above mentioned issues as our future research challenges.

REFERENCES

[1] Jorge A. Baier, Fahiem Bacchus, and Sheila A. McIlraith, 'A heuristic search approach to planning with temporally extended preferences', *Artificial Intelligence*, **173**(5-6), 593–618, (April 2009).

[2] Plaban Kumar Bhowmick, Debnath Mukherjee, Prateep Misra, and Arunasish Sen, 'A study on applicability of hierarchical task network planner in smart city scenario: A smart city tour and travel planning use case', in *27th International Conference on Computers and their Applications (CATA-2012)*, pp. 252–257, (2012).

[3] Stefan Edelkamp, Shahid Jabbar, and Mohmmed Nazih, 'Cost-Optimal Planning with Constraints and Preferences in Large State Spaces', in *In International Conference on Automated Planning and Scheduling (ICAPS) Workshop on Preferences and Soft Constraints in Planning*, pp. 38–45, (2006).

[4] Kutluhan Erol, James Hendler, and Dana S. Nau, 'UMCP: A sound and complete procedure for hierarchical task-network planning', in *International Conference on AI Planning Systems (AIPS)*, pp. 249–254, (June 1994).

[5] Alfonso E. Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos, 'Deterministic planning in the fifth international planning competition: Pddl3 and experimental evaluation of the planners', *Artificial Intelligence*, **173**(5-6), 619–668, (April 2009).

[6] Chih-Wei Hsu, Benjamin W. Wah, Ruoyun Huang, and Yixin Chen, 'Constraint partitioning for solving planning problems with trajectory constraints and goal preferences', in *Proceedings of the 20th international joint conference on Artifical intelligence*, IJCAI'07, pp. 1924–1929, San Francisco, CA, USA, (2007). Morgan Kaufmann Publishers Inc.

[7] Okhtay Ilghami and Dana S. Nau, 'A General Approach to Synthesize Problem-Specific Planners', Technical Report CS-TR-4597, UMIACS-TR-2004-40, University of Maryland, (2003).

[8] Naiwen Lin, Ugur Kuter, and Evren Sirin, 'Web service composition with user preferences', in *Proceedings of the 5th European semantic web conference on The semantic web: research and applications*, ESWC'08, pp. 629–643, Berlin, Heidelberg, (2008). Springer-Verlag.

[9] Alexander Nareyek, Eugene C. Freuder, Robert Fourer, Enrico Giunchiglia, Robert P. Goldman, Henry Kautz, Jussi Rintanen, and Austin Tate, 'Constraints and ai planning', *IEEE Intelligent Systems*, **20**(2), 62–72, (March 2005).

[10] Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman, 'Shop2: an htn planning system', *J. Artif. Int. Res.*, **20**(1), 379–404, (December 2003).

[11] Shirin Sohrabi, Jorge A. Baier, and Sheila A. McIlraith, 'Htn planning with preferences', in *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pp. 1790–1797, San Francisco, CA, USA, (2009). Morgan Kaufmann Publishers Inc.