Master of Science in Informatics at Grenoble
Option PDES

Heuristics for HTN Planning

Auteur :
Ye XIA

Responsables :
M. Humbert Fiorino
M. Damien Pellier

Abstract

Hierarchical Task Network (HTN) is an important technique in the domain of automated planning. However, the heuristic strategies for HTN planning have not been much investigated.

In this report, we have proposed our Hierarchical Task Network (HTN) planning heuristic strategies, which help to accelerate the automated planning process, and also guarantee to find the best solution. The HTN planning algorithm on which the heuristics are based is also discussed.

Keywords: Automated Planning, HTN Planning, Heuristics, PDDL

Acknowledgement

This work would not have been possible without the kind support and help of many individuals. I would like to extend my sincere thanks to all of them.

First and foremost I express my gratitude to my supervisors M. Damien Pellier and M. Humbert Fiorino for their guide and kindness during my first research experience. I extend my thanks to the members of MAGMA team of Grenoble Informatics Laboratory, especially to Abdeldjalil Ramoul, Sandra Castellanos and Ankuj Arora, thanks for their patience when discuss with me.

On the personal side, I would like to dedicate this thesis to my parents for being always beside me, and to TAO Xinxiu for her love and support.

Last but not least, This work has been partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01), I earnestly thank for the financial support during my master's degree.

Contents

Chapter 1

Introduction

Automated planning, or simply planning, is a branch of Artificial Intelligence. Automated planning is a computational study of the deliberation process, it is the base of information processing tools which provide affordable and efficient planning resources. For over 30 years, many techniques have been developed to solve the planning problems.

Among the techniques, Hierarchical Task Network (HTN) planning is the most widely used one for practical applications. HTN planning has some great advantages: its domain-configurable feature makes HTN planners work efficiently, and allows the planners to solve complex real-world problems; HTN provides a convenient way to write problem-solving methods that correspond to how a human considers. Even though HTN technique has been widely used, we found that the HTN planners lack heuristics to guide their searches.

The goal of this research has been to propose the heuristics used in HTN planning systems, to guide the search to find the best solution of planning problems quickly.

The report is organized as follows:

chapter 2 introduces the principles of automated planning.

In chapter 3, we explain the principles of HTN planning; then an abstract HTN planning process is given, it is also the planning process for which we propose our heuristics; a group of HTN planners we have studied are also introduced in this chapter.

In chapter 4, we propose our heuristics.

In chapter 5, my implementation works have been discussed.

Finally, chapter 6 concludes and discusses the future works.

Chapter 2

Automated Planning

In our daily life, we plan the things we want to do before working them out. The things could be complex tasks like writing a scientific report, they could also be simpler ones like standing up from a seat. When we plan for something, we anticipate the outcomes of actions, then we know what to do (i.e. actions) step by step. Obviously, the planning process is a deliberation process in which we choose and organize actions. The product of the planning is a plan to be performed.

2.1 Introduction of Planning

Automated Planning is the reasoning side of acting. The aim of a planner is to generate a *solution plan* to achieve given *goals*. A planner uses knowledge of the world (e.g. initial world-state, possible actions) to decide what to do before actually performing it.

As in Figure 2.1, the input world-knowledge of a planning system includes possible *actions*, the *initial world-state* of the problem, the objective, etc. the output of the planning system is a *solution plan*. The simplest format of a plan is a sequence of actions (i.e. a totally-ordered plan).

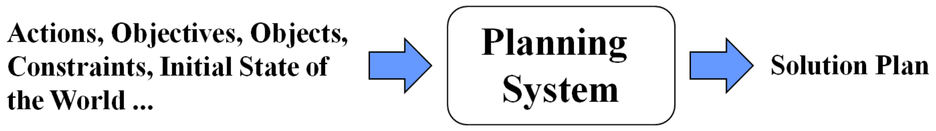
Actions, Objectives, Objects, Constraints, Initial State of the World ...
Planning System
Solution Plan

Figure 2.1: Planning System

- **Initial state**: The state of the world that we start in.

- **Actions/Operators**: Ways of changing the state of the world.
- **Objects**: Things in the world that interest us.
- **Predicates**: Properties of objects, they can be true or false.

Different from scheduling in which only certain constraints need to be considered (e.g. time), planning systems must reason about how the world changes to make decisions. Many planners need to keep track of world-state (i.e. the planners need the exact current world state in each step of the planning process). The world state also helps verifying the availability of an action. In planning, each action has a precondition and an effect (or postcondition), the precondition is verified according to the world state, and the effect indicates how the action changes the world state.

Different from reactive system which makes decisions based only on current input, a planning system takes into account the state in the "future" by anticipating the outcomes of actions.

The action sequences produced by the planning system will be executed by intelligent agents. The system plays an important role to ensure the agents' rational behavior. The automated planning is widely used in the domains like autonomous robots, space exploration, military logistics, computer games, etc.

2.2 Conceptual Model of Planning System

The conceptual planning system model could be represented as in Figure 2.2:

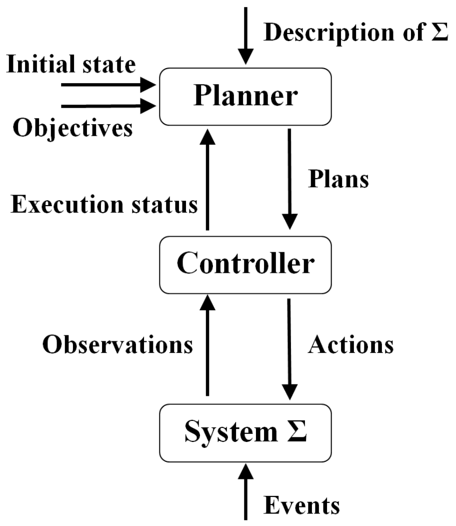

Figure 2.2: Conceptual Model

The model consists of the following three components:

- A **planner**, which produces plans and forwards them to the controller as actions to execute;
- A **controller**, which executes the plan according to observations perceived in the environment;
- A **state transition system** Σ, it models the environment in which the plan is to be executed. Σ is a 4-tuple, $\Sigma = (S, A, E, \gamma)$[1], where
    - $S = \{s_1, s_2, s_3...\}$ is a finite or recursively enumerable set of states;
    - $A = \{a_1, a_2, a_3...\}$ is a finite or recursively enumerable set of actions;
    - $E = \{e_1, e_2, e_3...\}$ is a finite or recursively enumerable set of events;
    - γ: $S \times (A \cup E) \to 2^S$ is a state translation function.

In Figure 2.2, the execution status is considered only when the planning is online (i.e. the environment is dynamic). Online planning is not discussed in this document. We will consider offline planning only.

2.3 Planning Model

Automated planning is inherently complex and causes large search space. Some planning problems are EXPSPACE-complete. Generally, some assumptions are made to lower the complexity, so that a practical planner can be designed and realized.

The assumptions of a restricted model[1] and the corresponding assumptions of the extended model[1] are as follows :

Assumption 1 (Finite Σ)**.** The system Σ has a finite set of states in which all the state variables need to be Boolean.

Relexing Assumption 1. In the system Σ, there may exist numerical state variables and actions that bring new objects, so infinite set of states must be supported.

Assumption 2 (Fully observable Σ)**.** The system Σ is fully observable, i.e. the planner has complete knowledge about the state of Σ.

Relexing Assumption 2. The system Σ could only be partially observable, i.e. not every aspect of Σ can be known.

Assumption 3 (Deterministic Σ)**.** The system Σ is deterministic, actions have strictly one possible outcome, i.e. for all s $\in S, u \in A \cup E : |\gamma(s, u)| \leq 1$.

Relexing Assumption 3. Each action or each event may have multiple alternative effects.

Assumption 4 (Static Σ)**.** The system Σ is static, changes in the environement are only caused by actions, there is no external events, i.e. E $= \emptyset$ or $\Sigma = (S, A, \emptyset, \gamma)$

Relexing Assumption 4. There may exist events in the system Σ. The events change the world state.

Assumption 5 (Restricted Goals)**.** The planner holds only restricted goals that are given as an explicit goal state s_g or a set of goal states S_g.

Relexing Assumption 5. More complex objectives may be required, the objective can be not only to reach a given state, but also to satisfy some constraints during the process to the goal state, e.g. some critical states to avoid, some states that must go through.

Assumption 6 (Sequential Plans)**.** A solution plan is a linearly finite sequence of actions.

Relexing Assumption 6. A plan can be partially-ordered. In this case, the plan is no longer a sequence of actions, the orderings between some actions may not be defined.

Assumption 7 (Implicit Time)**.** Actions and events have no duration in state translation systems.

Relexing Assumption 7. Action duration is taken into consideration, so that some temporally constrained goals can be expressed.

Assumption 8 (Offline Planning)**.** Planner is not concerned with changes of Σ while it is planning.

Relexing Assumption 8. Planner must consider the dynamic situations of the system, some objectives must be handled online.

The classical planning problem is under the assumptions of the restricted model. Our problem model is under the relaxing assumption 6, and other assumptions of the restricted model.

2.4 Domain-Independent Planning

There are many forms of planning: path and motion planning, perception planning, manipulation planning, communication planning, etc. They are all important domains, there exists some $domain - specific$ planners to solve specific types of problems (path planning, etc). The domain of such a planning system will be specified within the planner, so as in Figure 2.3, the input of a domain-specific planner only includes the description of a problem.

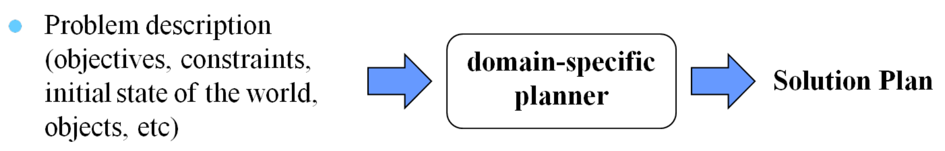
Problem description (objectives, constraints, initial state of the world, objects, etc)
domain-specific planner
Solution Plan

Figure 2.3: Domain-specific Planner

A domain-specific planner works efficiently, but the specification means less flexibility, their algorithms and the data representations are strictly specified. Each of them only solves a certain kind of problem, and when we need to modify the domain, the planner must be modified or even rewritten. On the contrary, a domain-independent planner is a generic planner that solves all kinds of problems. As in Figure 2.4, the input of a domain-independent planner consists of the descriptions of a problem and a domain.

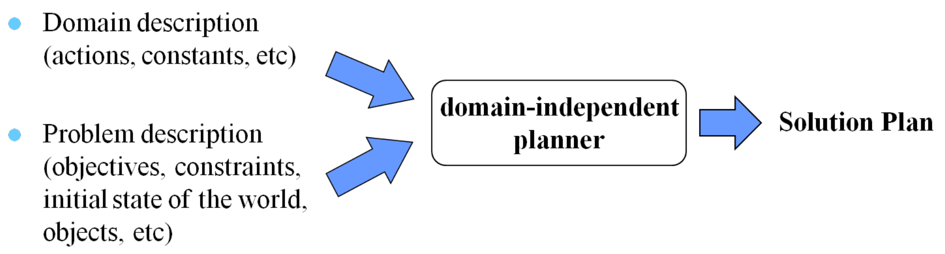
Domain description (actions, constants, etc)
Problem description (objectives, constraints, initial state of the world, objects, etc)
domain-independent planner
Solution Plan

Figure 2.4: Domain-Independent Planner

$Domain - independent$ planners gain flexibility, but their efficiency is more challenging and it requires a general high-level description language to specify the information. For example, Planning Domain Description Language(PDDL) is a standardization of planning domain and problem description languages. In this report, we focus on domain-independent planning.

2.5 An example : blocksWorld

The domain blocksWorld consists of a set of blocks, a table and a set of robot hands. The goal is to arrange the blocks into some given goal stacks. In the domain, we do not care about object positions on the table.

The constraints of blocksWorld are as follows:

- The blocks can be placed on top of another block or on a table;
- At most one block can be on top of another block, a block that has nothing on it is clear;
- Any number of blocks can be on the table;
- Robot hand can pick up blocks and stack them on other blocks;
- A robot hand can only hold one block or be empty.

We have simplified the blocks-world domain which consists of a set of blocks, a single table and a single robot hand, so that some of the constraints (e.g. constraint for multiple hands' synchronization) are not discussed in this report.

For the simplified blocksWorld domain, the constants of the planning problem are :

- A set of blocks {A, B, C, . . .}

x, y are two blocks, the predicates of the domain are :

▷ **ontable**(x) : the block x is on the table,
▷ **on**(x,y) : the block x is on top of the block y,
▷ **clear**(x) : there is no block on top of the block x,
▷ **handempty** : the robot hand is empty,
▷ **holding**(x) : the robot hand is holding the block x.

A precondition is a set of predicates. An effect is also a set of predicates.

The actions of the domain are :

▷ **pickup**(x) : pick up the block x which is currently on the table, x must be clear.
▷ **unstack**(x,y) : pick up the block x which is currently on top of the block y, x must be clear.
▷ **putdown**(x) : put the block x hold by the robot hand on the table.
▷ **stack**(x,y) : put the block x hold by the robot hand on top of the block y, y must be clear.

For example, the PDDL definition of the action *pickup* is as follows:

```
(:action pickup
    :parameters (?x - block)
    :precondition (and (clear ?x) (ontable ?x) (handempty))
    :effect
    (and (not (ontable ?x))
            (not (clear ?x))
            (not (handempty))
            (holding ?x)
        )
    )
```

In PDDL, a variable begins with the symbol "?". The parameter of *pickup* is a block x, its precondition is (clear x) ∧ (ontable x) ∧ (handempty), its effect is (not (ontable x)) ∧ (not (clear x)) ∧ (not (handempty)) ∧ (holding x). The other actions' PDDL definitions are in Appendix A.

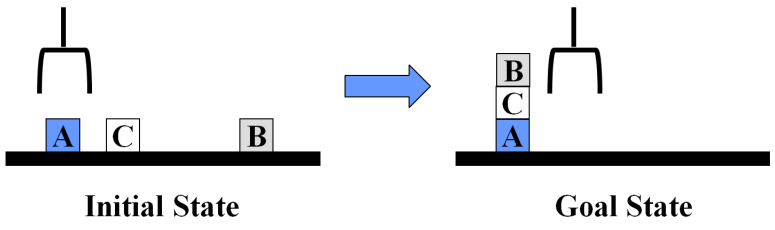

Figure 2.5: Planning Problem Example

Figure 2.5 is an example of the blocks-world planning problem, the initial state is : clear(a), clear(b), clear(c), ontable(a), ontable(b), ontable(c), handempty The goal state is : on(a,b), on(b,c), ontable(c)

A solution plan is : pickup(c) $\prec$ stack(c,a) $\prec$ pickup(b) $\prec$ stack(b,c), the symbol "$\prec$" indicates the ordering of a pair of actions, for example, $A_1 \prec A_2$ means that action A_1 must finish before A_2 starts.

Another solution plan is : pickup(c) $\prec$ putdown(c) $\prec$ pickup(c) $\prec$ stack(c,a) $\prec$ pickup(b) $\prec$ stack(b,c)

2.6 Planning Metrics

The two solutions of Figure 2.5 achieve the same goal. But obviously, the second one has some redundant actions, and the first one is better. The factors that make a solution plan better depends on the application, but some typical plan quality metrics are as follows:

- **Plan cost** : every action has a predefined cost, the plan cost is the sum of the cost of actions in the plan;
- **Plan length** : number of actions in the solution plan;
- **Makespan** : time to execute the plan.

The metrics of a planner are as follows:

- **Soundness**: if the planner returns a plan, then this is indeed a solution plan.
- **Completeness**: if there is a solution plan, then the planner will be able to find the solution plan.

If a planner is not complete, it is not guaranteed to find a solution when there exists one. But it is still interesting if an incomplete planner works with high performance. Generally, a planner which is not sound is senseless.

2.7 Planning Techniques

For over 30 years, many techniques have been developed to solve the problem of planning and to remove the assumptions described in section 2.3. Among these techniques, $classicalplanning$ which searches in state-spaces is the simplest.

Classical planning's search space can be represented as an oriented graph, in which, each node is a world state, and each arc is an action. State-space planning algorithms searches for a path to the goal state through the graph. Such a path is a solution plan. For example, in Figure 2.6, each node is a state, each solid arc is an action, the dot arcs indicate that some available actions and their following states are omitted. The node S_0 is the initial state, S_1 is the goal state. In the graph, there exists a path from S_0 to S_1 which is "stuck(C,B) $\prec$ unstuck(C,A)", this path is a solution plan.

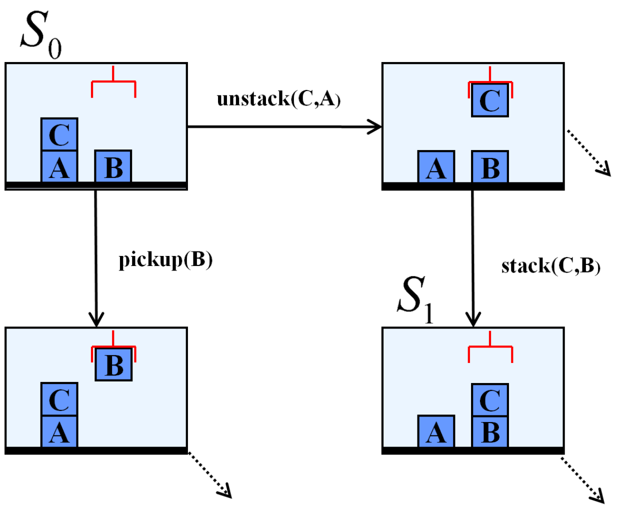

Figure 2.6: State Search Space

The planning algorithm generates part of the whole search space to find a path to goal state (i.e. to find a solution plan). In classical planning, there are several kinds of algorithms, such as forward search which is from the initial state towards the goal state. With the knowledge of the current state (at the beginning of the search, the current state is the initial state), forward search only tries available actions from the current node. Then the following states are obtained according to the effects of the action on the current state. The algorithm repeats this procedure until it reaches a goal state or the search has covered the whole graph.

There are many planning techniques which are different from classical planning. For example, some techniques search in $plan - spaces$ [25], they work by successively repairing a plan until all conflicts are removed.

The planning techniques using *graphs* [26] are based on two ideas: reachability analysis and disjunctive repairing technique, which solves one or more conflicts by using a disjunction of resolvers.

The SAT techniques (SATisfiability problem) encode a planning problem as a satisfiability problem and then search for a solution based on known SAT algorithms [27][28]. A planning problem can also be encoded as a CSP (Constraints Satisfaction Problem) problem [29][30]. The idea of these two techniques is to benefit directly from the advantages in the two areas.

With MDP (Markov Decision Process) techniques [31], [32], the planning problem is converted to develop an optimal policy, i.e. to associate a state with an action that maximizes the global reward. This technique is widely used to treat non-deterministic planning problems.

The techniques developed from $ModelChecking$ [33] take uncertainty, nondeterminism and partial observability of the environment into account.

Finally, HTN (Hierarchical Transition Network) planners [3][6] differ in the way to search for a plan, they decompose compound tasks recursively to primitive tasks. More details will be discussed in chapter 3.

Chapter 3

Hierarchical Task Network Planning

Hierarchical Task Network (HTN) planning has some similarities to classical planning (explained in section 2.7): each state of the world is represented by a set of predicates, and each action defines a deterministic state transition. However, HTN planning techniques differs from classical planning approaches in what they plan for and how they plan for it.

In this chapter, firstly, we explain the HTN technique; then we give the algorithm of an abstract HTN planning procedure; after that, we introduce a group of best-known HTN planners we studied; finally, we discuss and conclude the contents of this chapter.

3.1 Definitions and Principles

3.1.1 Task Network (TN)

Different from classical planning, HTN provides another approach to represent a plan. A task network is a directed acyclic graph. In this graph, each node is a task, the arcs represent precedence ordering between tasks.

In Figure 3.1, the nodes $A, B, C, \ldots$ are tasks, some orderings are: $A \prec B$, $B \prec C$, $B \prec D$, $E \prec C$. A task network can be partially ordered. There exists several possibilities when executing a partially ordered plan. For example, there is no ordering constraints between C and D, their execution order is either $C \prec D$, or $D \prec C$.

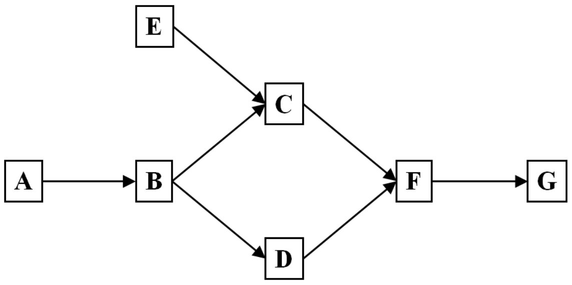

Figure 3.1: HTN Example

3.1.2 Hierarchical Structure

HTN has a hierarchical structure with two kinds of tasks: *primitive task* and *compound task*. In Figure 3.1, each node (e.g. A, B, C, ...) is either a primitive task or a compound task.

A **Primitive Task** is a task that can be achieved directly by executing a corresponding action.

A **Compound task** is a high-level action (HLA) which must be refined to a group of primitive tasks before execution.

In Figure 3.2, the compound task is to reverse a stack, this compound task needs several primitive tasks to be achieved.

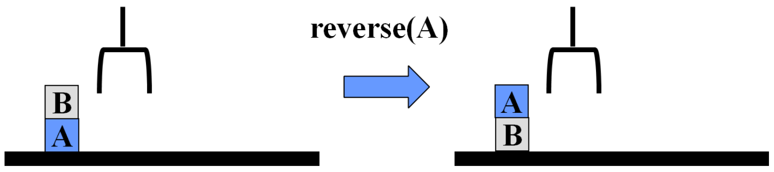

Figure 3.2: Compound Task Example

In Figure 3.3, the compound tasks Clear(x) is to achieve the predicate *clear(x)*, it has several possible refinements which lead to different states.

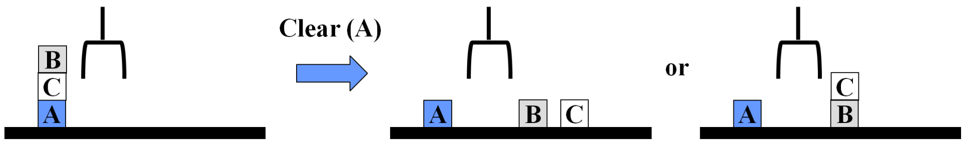

Figure 3.3: Compound Task Example 2

3.1.3 Constraints

In HTN planning, constraints are used to constrain the HTNs. There are two kinds of constraints[3], if n and n' are tasks' labels in a HTN,

- **Ordering Constraint**. Ordering constraints are of the form **n $\prec$ n'**, it indicates that n must finish before n' starts. n and n' both can be primitive task or compound task.

  In a more general case, instead of an individual node label like n or n', we use **first**$[n_i, n_j, \ldots]$ and **last**$[n_i, n_j, \ldots]$ to refer to the first task and the last one in execution respectively.

- **State Constraint**.p is a predicate, state constraints are as follows:

  - **Before constraint** is of the form **(n, p)**, which indicates that predicate p must be satisfied before the task n starts;
  - **After constraint** is of the form **(p, n)**, it indicates that p must be satisfied after the task n finishes.
  - **Between constraint** is of the form **(n, p, n')**. It indicates that p must be true in all states between n and n'.

A **causal link** reflects the interaction between two tasks. a_i and a_j are tasks in a HTN, a causal link is noted as : $a_i \xrightarrow{p} a_j$, p is a predicate which is part of a_j's precondition, the causal link indicates that p is satisfied by executing a_i (i.e. p is part of a_i's effect) and the order $a_i \prec a_j$.

There is a *flaw* if a state constraint is not satisfied. There are two kinds of flaws, threat and open-link. A threat breaks a causal link.

- A **threat** is noted as : $a_k, a_i \xrightarrow{p} a_j$, a_k is an action which threatens the causal link $a_i \xrightarrow{p} a_j$, i.e. ¬p is part of a_k's effect, and a_k is ordered between a_i and a_j.
- An **open link** is noted $a_s :\xrightarrow{p} a_i$, a_i's precondition is not satisfied, p is the corresponding predicate.

3.1.4 Hierarchical Task Network (HTN)

A HTN is a collection of *tasks* together with *constraints*, which is represented as: ($(n_1$: $t_1) \ldots (n_m$: $t_m)$, C), where
- each t_i is a task;
- C is a set of constraints;
- each n_i is a label for the task t_i.

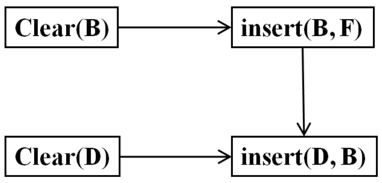

Figure 3.4: An example HTN: t_1

The HTN t_1 in Figure 3.4 is represented as:

((n_1: Clear(B)) (n_2: insert(B, F)) (n_3: Clear(D)) (n_4: insert(D, B)), ($n_1 \prec n_2$) $\wedge$ ($n_3 \prec n_4$) $\wedge$ ($n_2 \prec n_4$) $\wedge$ (n_1, clear(B)) $\wedge$ (n_3, clear(D)) $\wedge$ (n_2, on(B,F)) $\wedge$ (n_4, on(D,B)))

3.1.5 Method

Each method specifies a way to decompose a compound task into a set of subtasks through associating the compound task with a HTN (i.e. its refinement). A compound task can have several relevant methods (i.e. has several possible refinements). A refinement of a compound task can still contain compound tasks, a method can even be recursive.

We have extended PDDL to support HTN, an example method defined in PDDL is as the follows:

```
(:method reverse
    :parameters (?x - block)
    :precondition(and (handempty) (on ?y ?x) (ontable ?x))
    :expansion (
        (tag a (reverse(?y)))
        (tag b (pickup(?x)))
        (tag c (stack(?x, ?y)))
    )
    :constraints(
        (series a b c)
        (after (and (handempty) (ontable ?x) (clear ?x) (clear ?y)) a)
    )
)
```

The name of a method is the same to its corresponding compound task. In the example, the method contains:

relevant compound task : *reverse*(?*x* - *block*)
precondition : handempty $\wedge$ on(y, x) $\wedge$ (ontable(x)
refinement : ((a: reverse(y)) (b: pickup(x)) (c: stack(x, y)), (a $\prec$ b) $\wedge$ (b $\prec$ c) $\wedge$ (a, handempty) $\wedge$ (a, ontable(x)) $\wedge$ (a, clear(x)) $\wedge$ (a,

clear(y)))

A refinement consists of an *expansion* (i.e. the set of subtasks) and a group of constraints. In an expansion, "tag" is the key word to indicate the label of a task. A label is necessary, since the same task instance may repeat in a HTN, the labels help to identify the tasks in the constraints, a label can not repeat within a method. In the constraints, "series" is a key word to indicate the ordering constraints, in the example, the ordering of tasks is reverse(y) $\prec$ pickup(x) $\prec$ stack(x, y). "after" is to indicate an after state constraint. When task stack(x, y) finishes, (handempty) $\wedge$ (ontable x) $\wedge$ (clear x) must be satisfied. Other key words "before" and "between" indicate the corresponding constraints. Other method definitions are in Appendix C.

Methods provide additional knowledge to the planning process. A HTN planner do not need to search a complete state-space but to try several known possible decompositions. Thus, HTN's search space has been reduced compared with classical planning.

3.1.6 HTN Planning

Some differences between HTN planning and classical planning are as follows:

	Classical Planning	HTN Planning
Objective	Achieve a goal state	Perform a task
Planning procedure	Search for a sequence of actions that lead to the goal state	Incrementally refine the tasks until reaching an implementation

Table 3.1: Classical Planning vs HTN Planning

Different from state-space planning, HTN planning searches in plan-space, each node in the search space is a partially-specified plan, each arc is a refining procedure. **Refining** is the procedure of replacing a compound task with its refinement and updating the corresponding constraints. An **implementation** is a refinement of a HTN which contains only primitive tasks.

An HTN planning domain consists of a set of operators and a set of methods. Planning process proceeds by refining compound tasks into smaller and smaller tasks, until an implementation has been achieved which can be performed directly.

3.2 Abstract HTN Planning Procedure

Before introducing the abstract HTN planning procedure, we give some definitions firstly as follows:

A **HTN** is a pair $w = (U, C)$, where U is a set of tasks and C is a set of constraints.

A **method** is a 3-tuple $m = (task(m), expansion(m), constr(m))$ in which the elements are described as follows:

- $task(m)$ is the relevant compound task,
- $expansion(m)$ is a set of subtasks,
- $constr(m)$ is a a set of constraints,
- $(expansion(m), constr(m))$ is a HTN to refine to.

Suppose that $w = (U, C)$ is a HTN, $u \in U$ is a task, m is a method instance and $task(m) = u$. Then m refine u into $expansion(m)$, producing the HTN

$$\sigma(w, u, m) = ((U \; - \; \{u\}) \; \cup \; expansion(m), \; C' \; \cup \; constr(m))$$

where C' is the following modified version of C. p is a predicate, v is another task in w, the modification is shown as follows, the item before "$\rightarrow$" is the constraint to be replaced, the one after is its replacement.

- **After constraint** (u, p) $\rightarrow$ (last[$expansion(m)$], p)
- **Before constraint** (p, u) $\rightarrow$ (p, first[$expansion(m)$])
- **Between constraint**
  - $(v, p, u) \rightarrow$ (v, p, first[$expansion(m)$])
  - $(u, p, v) \rightarrow$ (last[$expansion(m)$], p, v)
- **Ordering constraint**
  - $(v \prec u) \rightarrow$ ($v \prec$ first[$expansion(m)$])
  - $(u \prec v) \rightarrow$ (last[$expansion(m)$] $\prec v$)

An HTN **planning problem** is a 4-tuple $\mathcal{P} = (s_0, w, O, M)$ where s_0 is the initial state, w is the initial HTN, O is a set of operators, and M is a set of methods.

A **linearization** of a partially-ordered plan is a totally-ordered plan which satisfies all the ordering constraints of the partially-ordered plan. For example, the plan (or HTN) in Figure 3.5 has two linearizations: $A \prec B \prec C$ and $A \prec C \prec B$.

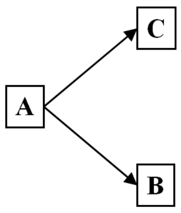

Figure 3.5: A partially-ordered plan

A **solution plan** is a partially-ordered plan, each of its linearizations satisfies all the state constraints of the HTN.

An abstract HTN planning procedure is as follows:

Algorithm 1: Abstract HTN Planning Procedure

```
Input: a planning problem $\mathcal{P} = (s_0, w, O, M)$
1  open ← $\{w\}$ ;
2  while open $\neq \emptyset$ do
3      nondeterministically choose a HTN $\nu \in$ open ;
4      remove $\nu$ from open ;
5      if $\nu = (U, C)$ is primitive then
6          if all constraints in $C$ are satisfied then return $\nu$;
7      else
8          nondeterministically choose a task $u \in \nu$ ;
9          active ← $\{m \in M \mid task(m)$ is relevant to $u\}$ ;
10         if active $\neq \emptyset$ then
11             for each method $m \in$ active do
12                 $\mu \leftarrow ((U - \{u\}) \cup expansion(m),\ C' \cup constr(m))$;
13                 add $\mu$ to open ;
14 return failure ;
```

In line 1, the algorithm begins with the input initial HTN, which is stored in *open*. *open* is a list of HTN, it is used to store all the generated HTNs during the search.

In line 3-4, a HTN ν is popped from *open*, it has been chosen non-deterministically, a deterministic choice will be discussed in chapter 4.

In line 5-6, if the selected HTN is primitive (i.e. an implementation), we check if all the constraints within the HTN has been satisfied, if so, we return the HTN which is a solution.

In line 7-9, if the selected HTN is not an implementation, which means there exists still compound tasks within the HTN. Then one of the compound tasks is chosen non-deterministically, and all its relevant methods will be stored in *active*. A deterministic choice of compound task will also be discussed in chapter 4.

In line 10-13, if *active* is not empty, each relevant method will be applied, we use m to refine the selected compound task, an updated HTN will be obtained and stored in μ, then it is added in *open* to be considered in following iterations.

In line 14, if *open* is empty, which means all possible refinements have been tried without finding a solution, there is no solution to the input planning problem, the algorithm returns failure.

algorithm 1 is sound and complete.

3.3 Related Work

The basic ideas of HTN planning were developed more than 25 years ago in works of Sacerdoti [1, p. 460] and in Tate's Nonlin planner [1, p. 503]. HTN planning has been more widely used in planning applications than any other classical planning techniques, e.g., in production line scheduling [1, p. 549], crisis management and logistics [1, p. 72] [1, p. 135], etc.

The first step toward a theoretical model of HTN planning is taken by Yang [1, p. 558] and Kambhampati [1, p. 301]. A complete model was developed by Erol [1, p. 174]. This model provided the basis for complexity analysis [1, p. 175] and the first provably correct HTN planning procedure (the planning procedure 1 is based on this work).

Some best-known domain-independent HTN planning systems are listed below.

UMCP [3] [4] [1, p. 174] is an implementation of the first provably sound and complete HTN planning algorithm. UMCP searches by iteratively refining a non primitive HTN to a primitive one. In each iteration, a non-primitive task is selected non-deterministically, one of its relevant methods will be chosen to refine it, then according to this refinement, a new HTN is created. Once a primitive HTN is created, it will be returned if all the constraints of the problem are satisfied, or the planner backtracks. To reduce the amount of backtracking, UMCP calls a function for detecting and resolving the flaws caused by interactions among tasks.

SHOP [6] [7] (Simple Hierarchical Ordered Planner) plans for tasks in the same order as their execution, thus it can always keep track of world-state during the planning process. With the knowledge of the current state, SHOP gains planning efficiency, it refines non-primitive tasks only with the available methods (i.e. a method whose precondition and also the precondition of the first task of the method's refinement are satisfied), so that the search space is reduced compared to UMCP; it also gains expressive power, since the knowledge of current world-state allows using *dynamic expressions* (e.g. numeric expressions, calling external programs). But these features force SHOP to do a forward search from the first task to the last one as the tasks' execution ordering, and the result plan is limited to be totally ordered. SHOP is sound and complete, but it suffers from backtracking. When there is no applicable method to decompose a non-primitive task, it backtracks to guarantee the completeness. There can be several possible ways to decompose a non-primitive task, backtracking happens after a non-primitive task has not been decomposed with the proper method. However, in SHOP, methods are chosen non-deterministically, and it is hard to decide which node in the search space to backtrack to.

M-SHOP [8] (Multi-task-list SHOP) generalizes SHOP by allowing the initial HTN to be partially ordered, thus the refinements of non-primitive tasks may be interleaved when executing the plan. The interleaving allows removing duplicated actions. M-SHOP does not guarantee to remove all the duplicated actions, and the solution plan is not guaranteed to be optimal (i.e. result plan with the smallest length). The interleaving may cause task-interaction issues. To deal with the issue, in M-SHOP, *protection request* and *protection cancellation* are defined in action effects. A protection request is to guarantee a predicate. The predicate guaranteed must be satisfied before the protection is canceled. M-SHOP uses a global list to store the protection list.

GoDel [9] (Goal Decomposition with Landmarks) is motivated by the fact that planners have methods which only solve some subproblems, but not the top-level problems. Instead of a HTN, GoDeL uses a goal network as input. A *goal network* is a partial order network which guides the algorithm to achieve the final goal step by step, each node in the network is a world state, the final node is the final goal state. The definition of method in GoDeL is also different, it does not have the HTN to refine to, instead, it has a sub goal network to indicate the steps to achieve the goal. In GoDeL, both methods and *subgoal inference* are used to decompose a task to subtasks (i.e. add subgoals into the goal network). Subgoal inference is based on landmarks. A *landmark* for a planning problem P is a fact that is true at some point in every plan that solves P, so it is considered as a subgoal that every solution to P must satisfy at some point. The algorithm performs a forward search from the initial state, it keeps track of the current state. By combining classical planning and HTN planning, GoDeL supports incomplete domains, no matter the domain knowledge is complete, it is always sound and complete.

Angelic The basic idea of Angelic [10] is to plan in a higher level. Angelic needs an additional goal description G (a set of literals) to resolve a planning problem and two sets of reachable states called Overstated (i.e. superset) and Understated (i.e. subset). These two sets can be considered as *MAY* and *MUST* reachable states respectively. If $MUST \subseteq$ G, ADD A VERB then arbitrary implementation of the HTN achieves the goal. If $MAY \cap G = \emptyset$, there is no need to continue to refine the plan which will never lead to the goal state. With deeper refining, a non-primitive task's reachable states will be exacter, so when goal state is in *MAY* but not in *MUST*, the non-primitive tasks in the plan need to be refined.

The algorithm does a top-down, forward search (i.e. from the top-level non-primitive task to a primitive plan, from the first task to the last one as the tasks' execution order), it is sound and complete, and the plan is totally-ordered. With the help of non-primitive task description, refining is performed only when it is necessary, so that the algorithm can avoid backtracking.

The extended version[11] of Angelic considers cost of actions, it generates provably optimal plans or generate nearly optimal plans with better performance. Its heuristics is inspired from A* algorithm, the heuristics use the data structure of *abstract lookahead tree* (i.e. lookahead tree adapted for non-primitive task). The algorithm's basic idea is the same to the original version, each node of the abstract lookahead tree has an *optimistic cost* and a *pessimistic cost*, the optimistic cost will be infinite for the nodes from which the goal states is unreachable. So the plan with the lowest optimistic cost will be refined prior to others. When an exact cost is obtained, the plans whose optimistic cost is greater than the obtained value will no longer be considered.

Yoyo The main idea of Yoyo[12] is to combine HTN with BDD (Binary Decision Diagram) for planning in non-deterministic domain. In non-deterministic domain, each action can have several alternative effects (i.e. the actual effect of an action is randomized), but the plan must work despite the non-determinism, so all possible effects (or all possible following states) must be considered, any state which leads to a *dead end* (i.e. no further refining can be done, but the goal state is not yet achieved) forces the algorithm to return a failure.

To guarantee that every execution path (multiple execution path is caused by different action effect) leads to the goal state, the returned plan is also different from other planners, it returns a list of situations (a situation is a pair of state and action), it indicates appropriate action according to the observed state. Yoyo does a forward search, in each iteration of the algorithm, it chooses a task without predecessor in the HTN, and it searches with the knowledge of the current state, so only applicable actions are considered in each step. Yoyo is both sound and complete.

BDD is used to represent a set of states in Yoyo. With the help of BDD, Yoyo realizes a set-based search, so that it avoids to search for each state separately and gains efficiency. To represent a set of states, the BDD does not need to list the propositions for which both arcs lead to the same terminal node, so an appropriate use of BDD also reduces the memory consumption.

HiPOP [13] (Hierarchical Partial-Order Planner) combines HTN and partial order planning (POP) [Add ref 50 Ghallab's book]. It supports optional user-defined non-primitive tasks and methods. HiPOP is both sound and complete.

If there is no non-primitive task defined, HiPOP works in the same way as a classical POP algorithm. The initial plan of classical POP consists of two dummy actions a_s and a_e, a_s is the first action of the result plan, and a_e is the last one. Precondition(a_s) = $\emptyset$, Effect(a_s) = I; Precondition(a_e) = G, Effect(a_e) = $\emptyset$. Obviously, if $I \notin G$, the plan is initialized with an open-link to repair. All generated plans for repairing a flaw are inserted into an open plan list. For each iteration, a plan which is not yet explored in the open list will be chosen and removed from the open list. According to the heuristics, the chosen plan should be the one which is most likely to be a solution. If the chosen plan does not contain any flaw, it will be returned. Otherwise, one of its flaws will be chosen and repaired. The algorithm stops and returns failure if the open list has been empty while no solution is found.

Extended from classical POP, HiPOP plans with non-primitive task. Each non-primitive task is considered as an *abstract flaw*, so a solution plan must be primitive. To take the advantage of non-primitive task, only non-primitive task is allowed to be directly added into the plan for repairing flaws, primitive actions can only be added through refining. In this way, through the additional knowledge provided by methods, the planning is guided as much as possible.

A* algorithm has been used as the plan heuristics. For flaw heuristics, the general order is: threat > open-link > abstract flaw (a > b means that a is prior to b), threats with the fewest resolvers will be solved firstly. Generally, open-links is solved earlier than abstract flaws, so that the algorithm deals with smaller plans during search process. Otherwise, after all the refining have been done, a huge plan with many flaws is likely to be generated.

IMPACTing SHOP [16] integrates SHOP and IMPACT [15] multiagent environment. In this work, although the environment is a Multiagent System, the planning is centralized, and it supports only a single planner while other agents are considered as information sources. Among IMPACT agents, there exists some special agents such as: statistics agent, monitoring agent, supplier agent which is for supporting calling external functions; math agent which supports numerical expressions.

To support the planner to interact with external agents (i.e. information sources), in the planning algorithm A-SHOP (agentized SHOP), the preconditions and the effects of actions have been replaced with code-calls, a code-call is a function call to other IMPACT agent. Direct execution of these code-calls cause difficulties when the algorithm needs to backtrack, since code-calls affect other agents' states. To solve this problem, a monitoring agent monitors the code-calls without executing them, the code-calls are actually applied only when the apply function is called.

A code-call is an arbitrary software function, thus the algorithm is sound and complete only when code-calls are *strongly safe*, which guarantees the finiteness of a function call.

CoRe Plannner [18] has a multiagent planning model which cooperates agents for achieving a common goal. The system has combined the advantages of POP and HTN, POP is adapted to concurrent planning in distributed environment, HTN has advantages in both efficiency and expressivity. Agents' partial knowledge and heterogeneous skills are also supported in the system.

The system plans for achieving a given goal state G, all the agents search within a global shared search space. The search space is represented with a Directed Acyclic Graph (DAG), whose nodes are partial plans. The nodes are allowed to contain flaws which are considered as promises, the flaws become new goals in the following planning. Each agent can refine, refute or repair a partial plan (i.e. a node in the DAG), and records other agents' propositions.

The initial plan is the same as in Classical POP (in explanation of HiPOP). Then in each iteration of the algorithm, one of the *non-terminal plans* (i.e. at least one refining or repair or refutation is applicable) which is not yet explored will be chosen. If the chosen plan does not contain any flaw, the agent proposes a "success". Otherwise, one of its flaws will be chosen, if the flaw is an open-link, it will be solved by adding a causal link or by the HTN-based refinement mechanism; if the flaw is a threat, the system will try to repair it. All the generated plans for solving a flaw are added in the DAG.

When an agent proposes a "success" (resp. failure) and wait for responses of other agents, the other agents verify whether the proposed plan is a solution plan (resp. whether this agent is not able to provide a possible solution) with their own knowledge. If the proposition is accepted by all the agents, the planning process ends. Otherwise, the system goes back to the planning phase.

The system is both sound and complete. The solution plan is partially-ordered. A* algorithm is applied as the plan heuristics. For flaw heuristics, a flaw with the fewest resolvers will be chosen.

3.4 Discussion and Conclusion

Compared with classical planners, the primary advantage of HTN planner is their additional knowledge representation and reasoning capabilities. With a good set of methods, HTN planners can solve classical planning problems orders of magnitude more quickly than classical planners.

All the Classical Planning problems can be translated to HTN planning problems, HTN planning is even more expressive than classical planning. HTN planners can represent and solve a variety of non-classical planning problems. For example, we need to reverse a stack and then reverse it back as in Figure 3.6. Obviously, the initial HTN is "reverse(A) $\prec$ reverse(B)", we can get a solution after HTN planning process. However, in classical planning, the problem will be as in Figure 3.7, we always get an empty plan since the goal state is right the same as the initial state.

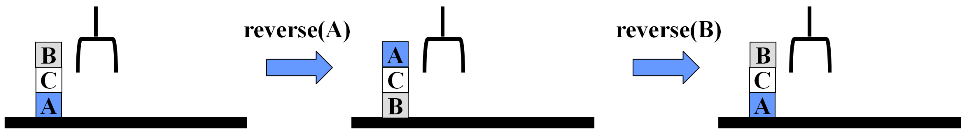

Figure 3.6: Expressivity Example: HTN

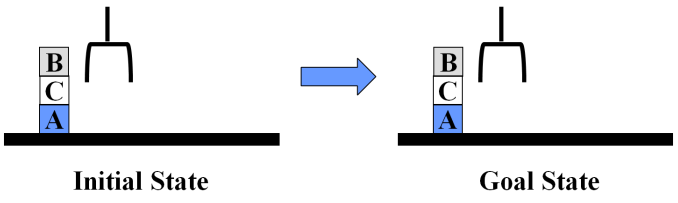

Figure 3.7: Expressivity Example: Classical Planning

The primary disadvantage of HTN planners is the need for the domain author to write not only a set of planning operators but also a set of methods.

The basic ideas of the planners we have studied are as follows: UMCP refines compound tasks in an arbitrary order. SHOP simplifies the planning process to be in the same order as execution. M-SHOP allows un-ordered tasks in initial HTN. GoDeL combines HTN with classical planning to support incomplete domain. Angelic avoids backexpansioning through HLA descriptions. Yoyo combines HTN with BDD to support non-deterministic domains. IMPACTing SHOP combines SHOP with IMPACT to support multiple information sources. Both the CoRe and HiPOP combine POP with HTN.

Among these planners, SHOP, M-SHOP, GoDeL, Angelic, Yoyo and IMPACTing SHOP keep track of the world-state, they do a forward search step by step from the initial state. They loose flexibility, but they gain efficiency, since only available methods will be applied. UMCP chooses arbitrary compound task in a HTN to refine, but it is not as efficient as the planners like SHOP. CoRe and HiPOP both based on POP and HTN, they repair flaws during the planning process, which do not require exact world states, but they are rather POP planners, POP is the base of their planning process, HTN is only a strategy to increase the efficiency. HTN works differently in the two systems, HiPOP uses HTN to support abstract actions, in the CoRe, HTN is used in flaw repairing. The current version of HiPOP only supports single agent planning, while the unified framework is a multiagent planning system.

HTN planning is efficient, expressive, and widely used. However, among the above HTN planners, there is no heuristics except the extended Angelic, whose heuristics is based on HLA description. The heuristics play an important role in automated planning for improving system's performance, we are proposing our heuristics without the information of HLA description in chapter 4.

Chapter 4

HTN Planning Heuristics

In automated planning, finding a solution is an exponential problem. Therefore, heuristics are necessary to speed up the searches. Heuristic strategies have been very successful for state-space searches, and state-space planners are widely used by AI community [20, 21, 22]. On the contrary, to our knowledge, HTN heuristics have not been very much investigated. Performances of HTN planners are based on knowledge abstraction and refinement of the search space.

However, in HTN algorithms (see algorithm 1), there are several non deterministic choices that are opportunities for heuristic choices in deterministic implementations of these algorithms: we need to choose a HTN in the *openList* structure (see section 4.2); then in the chosen HTN, we select a compound task to refine (see section 4.3).

In this section, firstly, we remind the principle of *A* algorithm*; then we explain our heuristics for choosing a HTN network from the openList; after that, we introduce the algorithm to calculate the minimum distance between two states; finally, our heuristic for choosing a non-primitive task in a HTN will be discussed.

4.1 Reminder on A* Algorithm

A* [23] is a search algorithm which aims at finding a least-cost path between two nodes in a graph. A* search begins from the initial node, it searches within current reachable nodes in each step. For example, in Figure 4.1, the graph is a grid, the blue node is the initial node. In the first step of the search, the green nodes are reachable, then in the second step, the yellow nodes are reachable. The white node are non-reachable nodes in the first two steps.

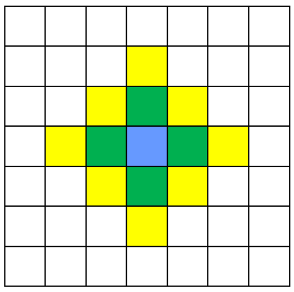

Figure 4.1: Reachable nodes example

Different from Dijkstra's algorithm, A* does not search all the reachable nodes, it does a best-first search, in each step, only the nodes with the lowest cost are searched. The evaluation of nodes is through a knowledge-plus-heuristic function, the function is as the follows:

$f(n) = g(n) + h(n)$, where

- n is a node in the search space;
- $f(n)$ is the cost function;
- $g(n)$ is the past cost function, which is the known distance from the initial node to n;
- $h(n)$ is the future cost function (or heuristic function), which is the estimated cost from n to the goal.

Through the cost function, A* avoids expanding paths that are already expensive. The higher $h(n)$ is, the fewer nodes A* expands and the faster A* is. But if $h(n)$ is greater than the true cost of moving from n to the goal, A* does not guarantee to find the shortest path.

On the contrary, if $h(n)$ is always lower than (or equal to) the true cost, A* is guaranteed to find one of the shortest paths, while A* will be slower. An extreme example is $h(n) = 0$, in this case, the algorithm works in the same as Dijkstra's algorithm. A heuristic function $h(n)$ is admissible if the distance to the goal is never overestimated (i.e. $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal from n), an admissible $h(n)$ guarantees to find the shortest path.

The best case is that $h(n)$ always equals exactly to the true cost, then A* gets the shortest path without expanding meaningless nodes, but it is difficult to make this happen in all the cases.

4.2 Choice of HTNs

The goal of our heuristics is to guide the planning algorithm to find the best solution as quick as possible. There are several metrics to assess plan's quality (as explained in section 2.6). In our work, we use plan's length as the metric. So the best solution is the solution whose length is the lowest among all the solutions of a planning problem.

HTN's hierarchical structure makes it difficult to get the length of a future solution. Since a non-primitive task may have several possible refinements, we cannot know exactly the length of the final solution before having refined all non-primitive tasks in to primitive ones.

We propose to see the abstract HTN procedure (see algorithm 1) as the A* algorithm: the cost function of our heuristics is $f(\nu) = g(\nu) + h(\nu)$, where ν is a HTN. We consider the length of primitive tasks in a HTN as the past cost $g(\nu)$, the length caused by non-primitive tasks is considered as an estimation of the remaining cost $h(\nu)$ to build a primitive HTN.

4.2.1 h_1: A Naive Approach

A naive idea is that a small HTN is more likely to lead to a short solution. The heuristics $f(\nu) = g(\nu) + h(\nu)$ is as follows:

- $g(\nu)$ is the number of primitive tasks in ν;
- $h_1(\nu)$ is the number of non-primitive tasks in ν.

This heuristic is quite simple. But, is this heuristic admissible? To answer, consider the method *do_nothing()* below:

```
(:method do_nothing
    :parameters ()
    :precondition(...)
    :expansion ()
    :constraints()
)
```

The non-primitive task *do_nothing()* contains no primitive task, but the heuristic function $h_1(\nu)$ will return 1. In other words, it is overestimated in the cost function, which may lead to a non-best solution. So this heuristic is not admissible.

In this approach, the weights of primitive tasks and non-primitive tasks are the same. However, generally, a non-primitive task contains more than one primitive tasks, which should weight more than a primitive task.

4.2.2 h_2: A Preciser Approach

To consider the different weights of tasks, we precise the non-primitive tasks through their refinements. A preciser approach is as follows:

- $g(\nu)$ is the number of primitive tasks in ν
- $h_2(\nu) = \sum_{t \,\in\, NPT(\nu)} min\{length(r) \mid r \in refinements(t)\}$

In $h_2(\nu)$, the function $NPT(\nu)$ returns the set of non-primitive tasks in ν, $refinements(t)$ is the set of all possible refinements of t (each relevant method of t corresponds to a refinement). Among all the possible refinements, only the one with the minimum length is counted, since the shortest refinement is more likely to lead to the best solution values.

This approach is still not admissible, since the refinement of a non-primitive task may still contain non-primitive tasks. For example:

```
(:method try_something
    :parameters ()
    :precondition(...)
    :expansion(
        (tag a (do_nothing))
    )
    :constraints()
)
```

The refinement of *try_ something* contains only a non-primitive task *do_ nothing*, *try_ something* is estimated as 1, but its true cost is 0. As $h_2(\nu)$ can be over-estimated, this approach is not admissible.

4.2.3 h_3: An Admissible Search

An admissible search is as follows:

- $g(\nu)$ is the number of primitive tasks in ν
- $h_3(\nu) = \sum_{t \in NPT(\nu)} min\{Length(r) \mid r \in Implementations(t)\}$

$Implementations(t)$ is the set of all possible implementations of t. In this approach, to get the weight of a non-primitive task, the task must be refined to primitive but without taking into account the constraints. The weight is much more informative, and therefore increase the planning performance. It can be relatively expensive to calculate $h_3(\nu)$ dynamically. Thus, we recommend to compute statically $h_3(\nu)$ for each non-primitive task of the planning problem during the preprocessing step, instead of in a dynamic way. We analyze for each non-primitive task its minimum implementation length only with the information within the planning domain. The algorithm to compute $h_3(\nu)$ is given by algorithm 2.

Algorithm 2: Non-primitive Task Length Analysis

```
   Input: a method set $METHODS$
   Output: a dictionary $d$ (key: non-primitive tasks, value: minimum
           length)
1  TASKS ← ∅;
2  for each $m \in METHODS$ do
3      TASKS ← TASKS $\bigcup$ $NonPrimitiveTasks(m)$;
4  d ← ∅;
5  for each $t \in TASKS$ do
6      d[t] ← 0;
7  isFinish ← FALSE;
8  while $\neg isFinish$ do
9      isFinish ← TRUE;
10     for each $t \in TASKS$ do
11         length ← ∞;
12         for each $m \in RelevantMethods(t)$ do
13             l ← 0;
14             r ← GetRefinement(m);
15             for each $task \in GetTasks(r)$ do
16                 if $IsPrimitive(task)$ then
17                     l ← l + 1;
18                 else
19                     l ← l + d[task];
20             if $l < length$ then
21                 length ← l;
22         if $length > d[t]$ then
23             d[t] ← length;
24             isFinish ← FALSE;
25 return d;
```

In line 1-3, the methods defined in the input domain are traversed, the function $NonPrimitiveTasks(m)$ returns the associated compound task of the method m. The variable $TASKS$ is the set of all non-instantiated compound tasks in the input domain.

In line 4-6, the dictionary d is initialized. The key set is non-instantiated compound tasks. All the values are initialized as 0.

In line 7-9, the variable $isFinish$ controls the algorithm's iteration. The loop stops only when d is no longer updated, which means d contains the same values as the previous iteration.

In line 10, the algorithm traverses all compound tasks, t is the currently considered task.

In line 12-14, the function $RelevantMethods(t)$ returns all t's relevant methods (i.e. the methods used to decompose t), each relevant method will be tried. To make sure the variable *length* stores t's minimum implementation length, *length* is initialized as ∞. $GetRefinement(m)$ returns the refinements of m.

In line 15-19, the function $GetTasks(r)$ returns the set of tasks within the HTN r. The algorithm checks whether each task is primitive or not. If the task is primitive, the task counts 1 as in line 17; if not, it counts $d[task]$ which is the current minimum implementation length of the compound task $task$.

In line 20-21, l stores the minimum implementation length of method m's refinement, if l is lower than the variable *length*, *length* will be updated.

In line 22-24, *length* stores t's minimum implementation length through all relevant methods; if it is higher than the previous minimum length $d[t]$, we update the value $d[t]$, and set $isFinish$ to be false so that the loop continues to see if this update leads to other updates.

For example, suppose that in a planning domain, there are 4 non-instantiated compound tasks, t_1, t_2, t_3, t_4. t_1 has 3 relevant methods, they contain 5 primitive tasks, 3 primitive tasks and t_2, 4 primitive tasks and t_4 respectively. As in Table 4.1, d[t_1] = min(5, 3+d[t_2], 4+d[t_4]). The other tasks' possible refinements are represented similarly. In each iteration of algorithm 2, d[t] = min(...) will be done through line 12-21. In algorithm 2, the procedure of d's update is given Table. 4.1.

iteration of algorithm	1st	2nd	3rd	4th	5th
d[t_1] = min(5, 3+d[t_2], 4+d[t_4])	3	4	4	5	5
d[t_2] = min(1+d[t_3], 1+d[t_4])	1	1	2	2	2
d[t_3] = min(0+d[t_4])	0	1	1	1	1
d[t_4] = min(1)	1	1	1	1	1

Table 4.1: Compound Task Length Analyse Procedure Example

In this table, the dictionary d does not change from the 4th iteration to the 5th, so the algorithm stops and returns d at the 5th iteration.

With the knowledge of non-primitive tasks' minimum implementation length, h_3(p) will never overestimate a HTN; this approach is admissible. In this way, only the HTNs that can be shorter than the current best solution are searched, and the search-space is reduced.

4.3 Choice of Tasks

4.3.1 Heuristic Principle

Once a HTN is chosen, to choose a non-primitive task (as in algorithm 1), in our heuristics, we propose to always choose the "easiest" one. As in Figure 4.2, an easy task is a non-primitive task whose refining is not time-consuming.

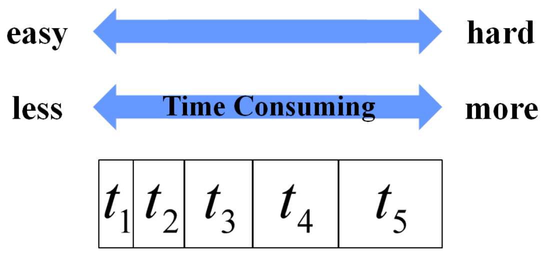

Figure 4.2: Easy and Hard Tasks

Suppose that a HTN consists of five non-primitive tasks (see Figure 4.2), t_1, t_2, t_3, t_4 and t_5, whose complexities are in ascending order, and that the only flaw occurs when refining t_3. Obviously, we will spend less time to detect this HTN does not lead to a solution with the following refining ordering: $t_1 < t_2 < t_3 < t_4 < t_5$ than with $t_5 < t_4 < t_3 < t_2 < t_1$ ($t_i < t_k$ means that t_i is refined before t_k).

Our heuristic strategy is to refine "easy" tasks before "complex" ones, so that the planning process will be less time-consuming, since the planning from a HTN which cannot lead to a solution ends earlier.

Our hypothesis is that, generally, an easy task does not require much preparation. For example, standing up from a seat is much easier than writing a scientific report which requires months of preparation, reading, doing experiments, etc. Whereas standing up just needs a mind. So we use this idea to assess whether a task is easy or not.

We define the preparation of a task as the actions satisfying the task's *before* constraints. Easier to satisfy is the *before* constraints of a task, easier is the task. Thus, in order to choose a non-primitive task, firstly, we extract the *before* constraints of each task; then, for each non-primitive task t, we get the *minimum distance* (explained in section subsection 4.3.2) from the input initial state I of the planning problem to the set of *before* constraints of t. Finally, among the set of the non-primitive tasks, we choose the task whose *minimum distance* is the lowest: the lower the minimum distance is, the easier the non-primitive task is to refine.

4.3.2 Estimating the distance between states

To get the distance, *method* cannot be used, since the search is between two arbitrary states. The procedure to get minimum distance is similar to classical planning, which searches in state-spaces.

But classical planning is complex and time-consuming, an intuitive relaxation idea is to neglect $Effects^{-}$(a), but consider only $Effects^{+}$(a). $Effects^{-}$(a) is the set of negative effects of action, $Effects^{+}$(a) is the set of positive effects. This simplifies $\gamma(s,a)$ involves only a monotonic increase in the number of propositions from s to $\gamma(s,a)$ (As explained in section 2.2 γ: $S \times (A \cup E) \rightarrow 2^{S}$ is a state translation function). Hence, it is easier to compute distances to goals with such a simplified γ. The following heuristic functions are based on this relaxation idea.

A First Approach

A first approach is as follows, p is a predicate, g is a state which contains a set of predicates, $\Delta_0(s,p)$ is the estimated distance from s to p, $\Delta_0(s,g)$ is from s to g.

$\Delta_0(s,p) = 0$ if p $\in s$
$\Delta_0(s,p) = \infty$ if $\forall\, a \in A,\ p \notin Effects^{+}(a)\ and\ p \notin s$
$\Delta_0(s,g) = 0$ if g $\subseteq s$

Otherwise,

$$\Delta_0(s,p) = min_a\{1+\Delta_0(s, Precond(a)) \mid \text{p} \in Effects^{+}(\text{a})\}$$

$$\Delta_0(s,g) = \sum_{p\, \in\, g} \Delta_0(s,p)$$

If $g \subseteq s$ or $p \in s$, which means the goal state is a subset of the goal state, there is no need to do anything, since the goal state has already been obtained, so $\Delta_0 = 0$.

If for a predicate p, there is no relevant operator (i.e. there is no action whose effects include p), there is no way to satisfy p, so the distance is ∞.

Otherwise, when calculating $\Delta_0(s,p)$, action a is performed to satisfy p, then a's precondition must be satisfied, so this is a backward analysis from a final predicate p to the initial state, only the path with the fewest actions counts finally. $\Delta_0(s,g)$ is the sum of $\Delta_0(s,p) \mid p \in g$.

However, this approach does not consider the fact that more than one predicates in g can be satisfied with a single action, instead it uses the sum of $\Delta_0(s,p)$. So the value of $\Delta_0(s,g)$ can be overestimated, it is not admissible.

An Admissible Approach

We consider then, the algorithm to compute the real distance between two states. The basic idea is the same as the backward search of classical planning. There are three cases:

$$\Delta^*(s,g) = \begin{cases} 0, & if\ g \subseteq s; \\ \infty, \quad if\ \forall\ a \in A,\ a\ is\ not\ relevant & for\ p \in g; \\ min_a\{1 + \Delta^*(s,\ \gamma^{-1}(g,a)) \mid a\ relevant\ for\ g\}, & otherwise. \end{cases}$$

If $g \subseteq s$, there is no need to do anything, Δ^*(s,g) = 0.

If for a predicate $p \in g$, there is no relevant operator (i.e. there is no action whose effects include p), there is no way to achieve g, so the distance is ∞.

Otherwise, we search for the shortest path from s to g. We use its inverse function of γ, γ^{-1}: $g \times a \rightarrow \mathrm{s}_a$ where g is the result state of action a, s_a is the state in which a is performed to achieve g, $s_a = \{g$ - $Effects^+(a)\} \bigcup$ Precond(a). An action a is relevant for g means that there exists at least one predicate p_i, $p_i \in g$ and $p_i \in Effects^+(a)$.

Through this approach, when a satisfies multiple predicates in g, all the predicates are considered through γ^{-1}. So we always get the real minimum distance between two states, the distance is never overestimated, and the approach is admissible.

A Complexity-Bounded Approach

The complexity of the admissible approach above is high when g contains a big amount of predicates. To limit the complexity, we propose the following algorithm:

$$\Delta_k(s,g) = \begin{cases} 0, & if\ g \subseteq s; \\ \infty, \quad if\ \forall\ a \in A,\ a\ is\ not\ relevant & for\ p \in g; \\ min_a\{1 + \Delta^*(s,\ \gamma^{-1}(g,a)) \mid a\ relevant\ for\ g\}, & if\ |g| \leq k; \\ max_{g'}\{\Delta_k(s,\ g') \mid g' \subseteq g\ and\ |g'| = k\}, & otherwise. \end{cases}$$

k is a predefined value, this algorithm is the same as the original version when |g| $\leq k$. If |g| $> k \wedge \Delta_k(s,g) \neq 0 \wedge \Delta_k(s,g) \neq \infty$, we analyze the minimum distance from s to g' , $g' \subseteq g$ and |g'| $= k$, all possible g' will be considered, the final result of the algorithm will be the maximal value of all $\Delta_k(s,\ g')$. This algorithm returns an approximate value, but it still contains much information, and we avoid searching in huge state-space but to search in several smaller state-spaces, the complexity of the algorithm is bounded.

The algorithm of Δ_k is used in our HTN planning system to analyse the distance.

4.3.3 Conclusion

Learning from A* algorithm, we have proposed the heuristics to choose a HTN, which help to find a good solution (with low length); through the algorithm for estimating distance between states, we get an approach to access a task is easy or not, so that an easy compound task can be chosen to be refined, which help to find a solution more quickly. Both the choices in algorithm 1 can be deterministic.

However, there are still a lot of work to do. There exists other possible heuristics, for example, to choose the most constrained compound task firstly, so that a flaw can be found as soon as possible, and a pruning can be done earlier. We may work on such ideas in the following work. And our heuristics are to be evaluated, whereas we are still working on the implementation of the HTN algorithm, the base planning system of the heuristics has not been finished, we are currently not able to evaluate the heuristics. In the following chapter, my work on implementation will be introduced.

Chapter 5

Implementation

The implementation of the heuristics must be based on the realisation of the HTN algorithm. In our project, the HTN algorithm is a Phd student's work, however, we have faced up to a lot of difficulties in this part. So we change the plan for my implementation work, I have participated in the HTN algorithm realisation part and have prepared a HTN planning domain (as in Appendix C).

5.1 PDDL4J

Our project has been based on the project PDDL4J[24]. PDDL4J is a planning system whose purpose is to facilitate the development of JAVA tools for Automated Planning based on PDDL language.

PDDL4J's work flow is as follows:

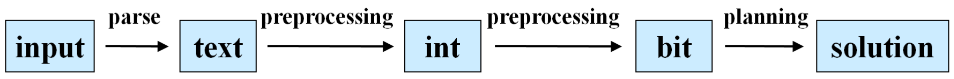

Figure 5.1: PDDL4J Flow Chart

The input of PDDL4J consists of a planning domain file and a planning problem file, both the files are in PDDL. These files are parsed through a parser within PDDL4J. The parsed information is represented in text format and stored in strings. The strings are memory-consuming, and they cause difficulties for following planning. So the preprocessing has been performed to convert the information to integer format and then to bit format as in Figure 5.1.

To represent the information in bit format (i.e. to store the information with bitset), we need to enumerate all the objects, all possible predicate instances and task instances. For example, there are three objects A, B and C in a blocksWorld planning problem, the possible instances of task $insert(x, y)$ includes: insert(A, B), insert(A, C), insert(B, A), insert(B, C), insert(C, A), insert(C, B). Each of the task instances corresponds to a bit of the bitset for storing tasks. The convertion from text to bit goes through integer format for storing the correspondence between the string representations and bit representations. So that when we finally get a solution in bit format, we can still convert and output the solution in text format.

We have extended PDDL4J to support HTN planning. The work includes:

- extending PDDL language to support HTN planning,
- extending the parser to support HTN semantics in input files,
- extending the data structure in the program to support HTN planning, e.g. methods, initial HTN, compound tasks, constraints, etc.

5.2 Algorithm Implementation

We have been implementing a HTN planning algorithm. The abstract HTN planning procedure algorithm 1 is a simplified version of our implementation. algorithm 1 has already been explained, in this section, we explain only the different parts.

5.2.1 HTN Planning Algorithm

The HTN planning algorithm we have been implementing is given by algorithm 3.

In line 1-2, we check if the input plan contains any ordering circle. A *circle* in the ordering is illegal. As in Figure 5.2, in the HTN, we have ordering constraints: $B \prec A$, $A \prec C$, $C \prec B$. These orderings form a circle which cause that we can never find an executable plan. If a circle exists, there is no need to search, we return failure directly.

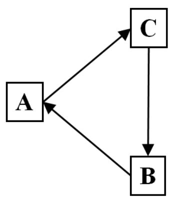

Figure 5.2: A HTN with a ordering circle

```
Algorithm 3: HTN Planning Algorithm
  Input: a planning problem $\mathcal{P} = (s_0, w, O, M)$
  Output: solution plan
1 if CheckOrderCircle(w) then
2     return failure;
3 openList ← w;
4 while openList ≠ ∅ do
5     p ← PopBestHTN(openList);
6     t ← SelectBestTask(p);
7     if t = NULL ∧ IsConstrSatisfied(p) = TRUE then
8         return p;
9     if t ≠ NULL then
10        for each m ∈ GetRelevantMethods(t,D) do
11            newPlan ← Refine(t, m, p);
12            add newPlan to openList;
13 return failure;
```

In line 5, the function *PopBestHTN* implements the heuristics to pop a HTN p from *openList*, the heuristics has been has been explained in section 4.2. If the admissible heuristic h_3 has been applied, a returned solution is guaranteed to be the best solution.

In line 6, the function *SelectBestTask* implements the heuristics explained in section 4.3, it returns a compound task t to refine. If the popped plan p is already primitive (i.e. p contains no compound task), the function *SelectBestTask* returns NULL.

in line 7-8, if t is NULL, the function *IsConstrSatisfied* checks if all the state constraints of p have been satisfied. If p is indeed a solution, p will be returned.

In line 10, the function *GetRelevantMethods* returns all the methods whose relevant compound task is t, then each of these methods will be applied.

In line 11, the function *Refine* is called, it applies method m for refining task t, then a new HTN is obtained, the variable *newPlan* stores the newly obtained HTN. The details of function *Refine* will be discussed in subsection 5.2.2.

5.2.2 Refining Algorithm

Our algorithm has been based on UMCP (explained in section 3.3), which plans without exact world states, and without caring about tasks' refining ordering. The same as UMCP, our algorithm needs to verify HTNs' constraints with uncertain world states caused by un-refined compound tasks. During the planning process, some constraints are considered as promises (i.e. promissory constraints), since there is not yet enough information to verify it, a promissory constraint can only be verified until some necessary compound tasks are refined.

The algorithm of the function *Refine* is given by algorithm 4.

Algorithm 4: Refining Algorithm

```
Input: Task: t, Method: m, Plan: p
Output: newPlan
1  $newPlan \leftarrow p$;
2  $refinement \leftarrow$ GetRefinement($m$, $t$);
3  $precond \leftarrow$ GetPrecond($m$, $t$);
4  AddConstr($newPlan$, $precond$, $t$);
5  if IsConstrUnsatisfiable(precond, t, newPlan) then
6      return $\emptyset$;
7  MergeExpansion($newPlan$, $refinement$);
8  TransferConstr($newPlan$, $t$);
9  MergeConstr($newPlan$, $refinement$);
10 RemoveTask($newPlan$, $t$);
11 VerifyAllConstr($newPlan$);
12 if IsConstrUnsatisfiable(newPlan) then
13     return $\emptyset$;
14 return $newPlan$;
```

The input of the Refining Algorithm includes:

- **t**: the task to refine;
- **m**: the method to apply;
- **p**: the original HTN.

In line 1, the variable *newPlan* is initialized as a copy of the input original HTN *p*.

In line 2, the function *GetRefinement(m, t)* returns the compound task *t*'s refinement (i.e. an instance of method *m*'s associated refinement).

In line 3, The function *GetPrecond(m, t)* returns the precondition instance (a precondition is a group of predicates) of method *m*.

In line 4, the method's precondition *precond* is converted to a before constraint (*precond*,*t*), which is added to HTN *newPlan*.

In line 5-6, we check if the before constraint (*precond*,*t*) is unsatisfiable(i.e. the constraint can never get satisfied), if so, the method *m* is not applicable, the algorithm returns an empty set $\emptyset$.

Line 7-10 is the process of updating the original HTN as explained in algorithm 1.

$$\sigma(newPlan, t, m) = ((U \ - \ \{t\}) \ \cup \ expansion(m), \ C' \ \cup \ constr(m))$$

Firstly, in line 7, we merge $expansion(m)$ into $newPlan$; then, in line 8, we traverse and update $newPlan$'s constraints C to C' according to the labels of merged $expansion(m)$, the modifications have been explained in Page. 18; after the traversal of constraints, in line 9, we merge $constr(m)$ into $newPlan$; lastly, in line 10, the task t is removed from the HTN *newPlan*, all the constraints which contain t will also be removed.

In line 11, all the constraints of *newPlan* are verified. In *newPlan*, the sets of *verified constraints*, *promissory constraints* and *unsatisfiable constraints* are updated. A constraint is verified only if it is satisfied in every linearization of the plan. On the contrary, a constraint is unsatisfiable if there exists any linearization in which the constraint is unsatisfiable; it's the same for promissory constraint, even if a constraint is satisfied in all the linearization except one in which it is promissory, then it is a promissory constraint.

In line 12-14, if the performed refining causes a *unsatisfiable constraint* (i.e. there exists a *unsatisfiable constraint* in *newPlan*), the algorithm returns $\emptyset$. Otherwise, it returns *newPlan*.

5.2.3 Discussion

A difference between our implementation and the abstract HTN planning procedure algorithm 1 is that in the implementation, we verify the constraints each time after having refined a compound task; however, in algorithm 1, the constraints are verified until a primitive HTN is obtained. We verify the constraints early, so that we can prune the HTNs which can not lead to a solution earlier.

SHOP's (explained in section 3.3) performance is better than UMCP. However we do not choose SHOP as the base of our algorithm, since SHOP needs exact world state in each step to verify constraints and methods' preconditions. In another word, the planning process of SHOP is inherently a sequential process. On the contrary UMCP's planning process introduces an interest to be distributed.

Being based on UMCP, our algorithm still has some differences from UMCP:

- Our algorithm use *openList* to store all the HTNs which possibly lead to a solution. This allows us to implement the heuristics to improve the search efficiency. In UMCP, the search is non-deterministic.
- In our algorithm, all relevant methods of a compound task are applied when refining it. In another word, all possibilities are considered, so there will not be any backtrack. Differently, in UMCP, a backtrack is necessary when the search reaches a dead end. The backtrack causes many difficulties, e.g. it is hard to decide which task to backtrack to.
- UMCP calls a function *critics* [5] for detecting and resolving the flaws caused by interactions among tasks. *critics* repairs the flaws, or it refuses the HTN if a flaw can not be repaired. UMCP considers only a single HTN during the search, which makes the correctness of the current HTN important. However, we do not apply *critics*. In our algorithm, a group of HTNs are considered, which makes applying *critics* meaningless.

Chapter 6

Conclusion and Future Work

In this work, we have proposed a sound and complete abstract HTN planning procedure, and based on this procedure, we have proposed the heuristics which help to find the best solution quickly. The heuristics work on two choices of a HTN planning algorithm, they are based on A* algorithm and minimum distance estimation algorithm respectively. For the implementation, we have discussed our work on HTN planning algorithm, which introduces the interest to be distributed to support multiagents planning.

For the future work, we will finish the implementation of the HTN algorithm, and evaluate the heuristics we proposed. Other heuristics which suit for HTN planning are possible, we may make comparison with the heuristics in this report. If possible, we will distribute the planning process of our HTN planning algorithm, and extend the system to be a multiagents HTN planning system.

Bibliography

[1] M. Ghallab, D. Nau, P. Traverso, Automated Planning: Theory and Practice, Morgan Kaufmann, 2004.

[2] M. Ghallab, C. Knoblock, D. Wilkins et al., PDDL–the planning domain definition language, the AIPS'98 Planning Competition Committee, 1998.

[3] K. Erol, Hierarchical task network planning: Formalization, Analysis and Implementation, Ph.D thesis, Dept. of computer science, Univ. of Maryland, College Park, 1995.

[4] K. Erol, J. Hendler, and D.S. Nau, UMCP: A Sound and Complete Procedure for Hierarchical Task Network Planning, Proceedings of the Second International Conference, K. J. Hammond, editor, pp. 249–254, Los Altos, CA, Morgan Kaufmann Publishers, Inc, 1994.

[5] K. Erol, J. Hendler, D.S. Nau, and R. Tsuneto, A Critical Look at Critics in HTN Planning, Proceedings of the 1995 International Joint Conference on Artificial Intelligence (IJCAI-95), pp. 1592-1598, 1995.

[6] D. S. Nau, Y. Cao, A. Lotem, and H. Munoz-Avila, SHOP: Simple Hierarchical Ordered Planner, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-99), pp.968—973, 1999.

[7] D.S. Nau, T.-C. Au, O. Ilghami, U. Kuter, D. Wu, Fusun Yaman, H. Munoz-Avila, and W. Murdock. Applications of SHOP and SHOP2. IEEE Intelligent Systems, 20(2):34–41, 2005. Earlier version as Tech. Rep. CS-TR-4604, UMIACS-TR-2004-46.

[8] D.S. Nau, Y. Cao, A. Lotem, and H. Munoz-Avila, SHOP and M-SHOP: Planning with Ordered Task Decomposition. Tech. Report CS TR 4157, University of Maryland, College Park, MD, 2000.

[9] V. Shivashankar, R. Alford, U. Kuter, and D. Nau, The GoDeL Planning System: A More Perfect Union of Domain-Independent and Hierarchical Planning. Proceedings of the Twenty-Third international joint conference on Articial Intelligence (pp. 2380–2386). 2013.

[10] B. Marthi, S. Russell, and J. Wolfe, Angelic Semantics for High-Level Actions, in Intl Conf on Automated Planning and Scheduling, Providence, RI, 2007.

[11] B. Marthi, S. J. Russell, and J. Wolfe, Angelic Hierarchical Planning: Optimal and Online Algorithms. ICAPS, 222–231. AAAI Press, 2008.

[12] U. Kuter, D.S. Nau, M. Pistore, P. Traverso, Task Decomposition on Abstract States, for Planning under Nondeterminism, Articial Intelligence 173, 2009.

[13] P. Bechon, M. Barbier, G. Infantes, C. Lesire, V. Vidal, HiPOP: Hierarchical Partial-Order Planning, 2014.

[14] M. M. de Weerdt and B. J. Clement. Introduction to planning in multiagent systems. Multiagent and Grid Systems An International Journal, 5(4), 2009.

[15] K. Arisha, F. Ozcan, R. Ross, V. Subrahmanian, T. Eiter, and S. Kraus. IMPACT: A Platform for Collaborating Agents. IEEE Intelligent Systems, 14:64–72, March/April 1999.

[16] J. Dix, H. Muoz-Avila, D. S. Nau, and L.Zhang, IMPACTing SHOP: Putting an AI Planner into a Multi-Agent Environment. Annals of Mathematics and Articial Intelligence, 37 (4), 381–407, 2003.

[17] J.S. Cox, E.H. Durfee, An Efficient Algorithm for Multiagent Plan Coordination, Proceedings of AAMAS, pp. 828–835. ACM, 2005.

[18] D. Pellier and H. Fiorino. A Unified Framework based on HTN and POP Approaches for Multi-Agent Planning. In International Conference on Intelligence Agent Technology (IAT), California, USA, 2–5 November 2007.

[19] D. Pellier, Modèle dialectique pour la synthèse de plans, PhD thesis, UJF - Grenoble, France, 2005.

[20] Xia, Y., Etchevers, X., Letondeur, L., Coupaye, T. and Desprez, F., 2018, April. Combining hardware nodes and software components ordering-based heuristics for optimizing the placement of distributed IoT applications in the fog. In Proceedings of the 33rd annual ACM symposium on applied computing (pp. 751-760).

[21] Xia, Y., Etchevers, X., Letondeur, L., Lebre, A., Coupaye, T. and Desprez, F., 2018, December. Combining heuristics to optimize and scale the placement of iot applications in the fog. In 2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC) (pp. 153-163). IEEE.

[22] Xia, Y., Etchevers, X., Letondeur, L., Coupaye, T. and Desprez, F., 2024, April. Optimizing Cloud Application Scheduling: A Dual-Stage Heuristic Approach. In 2024 9th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA) (pp. 134-140). IEEE.

[23] P. E. Hart, N. J. Nilsson, B.Raphael, A Formal Basis for the Heuristic Determination of Minimum Cost Paths, IEEE Transactions on Systems

[24] Science and Cybernetics SSC4 4 (2): 100–107, 1968. D. Pellier, PDDL4J, [Online] http://sourceforge.net/projects/pdd4j/, 2011.

[25] J. Penberthy and D. Weld. UCPO : A Sound, Complete, Partial Order Planner for ADL. In C. Rich B. Nebel and W. Swartout, editors, Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning, pages 103–114. Morgan Kaufmann Publishers, 1992.

[26] A. Blum and M. Furst. Fast Planning Through Planning Graph Analysis. Artificial Intelligence, 90(1-2) :281–300, 1997.

[27] K. Ray and M. Ginsberg. The complexity of optimal planning and a more efficient method for finding solutions. In Proceedings of the International Conference on Automated Planning and Scheduling, pages 280–287, 2008.

[28] H. Kautz and B. Selman. The Role of Domain-Specific Knowledge in the Planning as Satisfiability Framework. In Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling, pages 181–189, 1998.

[29] S. Kambhampati. Planning graph as a (dynamic) CSP : Exploiting EBL, DDB and other CSP search techniques in graphplan. Journal of Artificial Intelligence Research, 12(1) :1–34, 2000.

[30] C. Pralet and G. Verfaillie. Using constraint networks on timelines to model and solve planning and scheduling problems. In Proceedings of the International Conference on Automated Planning and Scheduling, pages 272–279, 2008.

[31] J. Marecki and M. Tambe. Towards faster planning with continuous resources in stochastic domains. In Proceedings of the Association for the Advancement of Artificial Intelligence, pages 1049–1055, 2008.

[32] M. Sridharan, J. Wyatt, and R. Dearden. Hippo : Hierarchical pomdps for planning information processing and sensing actions on a robot. In Proceedings of the International Conference on Automated Planning and Scheduling, pages 346–354, 2008.

[33] M. Pistore and P. Traverso. Planning as Model Checking for Extended Goals in Non-Deterministic Domains. In Proceedings of the International Joint Conference on Artificial Intelligence, pages 479–484, 2001.

Appendix A

A PDDL BlocksWorld Domain

```
(define (domain blocksworld)
    (:requirements :strips :typing)
    (:types block)
    (:predicates (on ?x - block ?y - block)
        (ontable ?x - block)
        (clear ?x - block)
        (handempty)
        (holding ?x - block)
    )

   (:action pickup
        :parameters (?x - block)
        :precondition (and (clear ?x) (ontable ?x) (handempty))
        :effect
        (and (not (ontable ?x))
            (not (clear ?x))
            (not (handempty))
            (holding ?x)
      )
   )

    (:action putdown
        :parameters (?x - block)
        :precondition (holding ?x)
        :effect
        (and (not (holding ?x))
            (clear ?x)
            (handempty)
            (ontable ?x)
        )
    )

     (:action stack
         :parameters (?x - block ?y - block)
         :precondition (and (holding ?x) (clear ?y))
         :effect
         (and (not (holding ?x))
             (not (clear ?y))
             (clear ?x)
             (handempty)
             (on ?x ?y)
         )
    )

     (:action unstack
         :parameters (?x - block ?y - block)
         :precondition (and (on ?x ?y) (clear ?x) (handempty))
         :effect
         (and (holding ?x)
             (clear ?y)
             (not (clear ?x))
             (not (handempty))
             (not (on ?x ?y))
        )
    )
)
```

Appendix B

A PDDL BlocksWorld Problem

```
    (define (problem BLOCKS-4-0)
    (:domain BLOCKS)
    (:objects D B A C - block)
    (:INIT (CLEAR C) (CLEAR A) (CLEAR B) (CLEAR D) (ONTABLE
C)
    (ONTABLE A) (ONTABLE B) (ONTABLE D) (HANDEMPTY))
    (:goal (AND (ON D C) (ON C B) (ON B A)))
    )
```

Appendix C

A HTN BlocksWorld Domain

```
(define (domain blocksworld)
    (:requirements :strips :typing)
    (:types block)
    (:predicates (on ?x - block ?y - block)
        (ontable ?x - block)
        (clear ?x - block)
        (handempty)
        (holding ?x - block)
    )

   (:action pickup
        :parameters (?x - block)
        :precondition (and (clear ?x) (ontable ?x) (handempty))
        :effect
        (and (not (ontable ?x))
            (not (clear ?x))
            (not (handempty))
            (holding ?x)
       )
   )

    (:action putdown
        :parameters (?x - block)
        :precondition (holding ?x)
        :effect
        (and (not (holding ?x))
            (clear ?x)
            (handempty)
            (ontable ?x)
        )
    )

     (:action stack
        :parameters (?x - block ?y - block)
        :precondition (and (holding ?x) (clear ?y))
        :effect
        (and (not (holding ?x))
            (not (clear ?y))
            (clear ?x)
            (handempty)
            (on ?x ?y)
        )
    )

     (:action unstack
        :parameters (?x - block ?y - block)
        :precondition (and (on ?x ?y) (clear ?x) (handempty))
        :effect
        (and (holding ?x)
            (clear ?y)
            (not (clear ?x))
            (not (handempty))
            (not (on ?x ?y))
        )
    )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    (:method reverse
        :parameters (?x - block)
        :precondition(and (handempty) (on ?y ?x) (ontable ?x))
        :expansion (
            (tag a (reverse(?y)))
            (tag b (pickup(?x)))
            (tag c (stack(?x, ?y)))
        )
        :constraints(
            (series a b c)
            (after (and (handempty) (ontable ?x) (clear ?x) (clear ?y))
a)
        )
    )
```

```
(:method reverse
    :parameters (?x - block)
    :precondition(and (handempty) (on ?y ?x) (on ?x ?z))
    :expansion (
        (tag a (reverse(?y)))
        (tag b (unstack(?x, ?z)))
        (tag c (stack(?x, ?y)))
    )
    :constraints(
        (series a b c)
        (after (and (handempty) (on ?x ?z)(clear ?x) (clear ?y)) a)
    )
)

(:method reverse
    :parameters (?x - block)
    :precondition(and (handempty) (clear ?x) (on ?x ?y))
    :expansion (
        (tag b (unstack(?x, ?y)))
        (tag c (putdown(?x)))
    )
    :constraints(
        (series b c)
    )
)

(:method reverse
    :parameters (?x - block)
    :precondition(and (handempty) (clear ?x) (ontable ?x))
    :expansion (
    )
    :constraints(
    )
)

(:method reverse
    :parameters (?x - block, ?y - block)
    :precondition(and (handempty) (on ?z, ?x) (clear ?y) (ontable
?x))
    :expansion (
        (tag a (reverse(?z, ?y)))
        (tag b (pickup(?x)))
        (tag c (stack(?x, ?z)))
    )
```

```
        :constraints(
            (series a b c)
            (after (and (handempty) (ontable ?x) (clear ?x) (clear ?z)
(not (clear ?y))) a)
        )
    )

    (:method reverse
        :parameters (?x - block, ?y - block)
        :precondition(and (handempty) (on ?z, ?x) (clear ?y) (on ?x ?j))
        :expansion (
            (tag a (reverse(?z, ?y)))
            (tag b (unstack(?x, ?j)))
            (tag c (stack(?x, ?z)))
        )
        :constraints(
            (series a b c)
            (after (and (handempty) (on ?x ?j) (clear ?x) (clear ?z) (not
(clear ?y))) a)
        )
    )

    (:method reverse
        :parameters (?x - block, ?y - block)
        :precondition(and (handempty) (clear ?x) (clear ?y) (on ?x ?z))
        :expansion (
            (tag b (unstack(?x, ?z)))
            (tag c (stack(?x, ?y)))
        )
        :constraints(
            (series b c)
        )
    )

    (:method reverse
        :parameters (?x - block, ?y - block)
        :precondition(and (handempty) (clear ?x) (clear ?y) (ontable
?x))
        :expansion (
            (tag b (pickup(?x)))
            (tag c (stack(?x, ?y)))
        )
        :constraints(
            (series b c)
        )
    )

    (:method insert
        :parameters (?x - block, ?y - block)
        :precondition(and (clear ?x) (handempty) (ontable ?x) (on(?z,
?y)))
        :expansion (
            (tag a (reverse(?z)))
            (tag b (pickup(?x)))
            (tag c (stack(?x, ?y)))
            (tag d (reverse(?z, ?x)))
        )
        :constraints(
            (series a b c d)
            (after (and (clear ?y) (ontable ?x) (handempty)) a)
            (after (and (on ?x ?y) (on ?z ?x)) d)
        )
    )

    (:method insert
        :parameters (?x - block, ?y - block)
        :precondition(and (clear ?x) (handempty) (on ?j ?x) (on(?z,
?y)))
        :expansion (
            (tag a (reverse(?z)))
            (tag b (unstack(?x, ?j)))
            (tag c (stack(?x, ?y)))
            (tag d (reverse(?z, ?x)))
        )
        :constraints(
            (series a b c d)
            (after (and (clear ?y) (on ?j ?x) (handempty)) a)
            (after (and (on ?x ?y) (on ?z ?x)) d)
        )
    )

    (:method Clear
        :parameters (?x - block)
        :precondition(and (not (clear ?x)) (handempty))
        :expansion (
            (tag b (reverse(?x)))
        )
        :constraints(
            (after (clear ?x) b)
        )
    )

    (:method Clear
        :parameters (?x - block)
        :precondition(and (handempty) (on ?y ?x) (on ?x ?z))
        :expansion (
            (tag a (Clear(?y)))
            (tag b (unstack(?x, ?z)))
            (tag c (putdown(?x)))
        )
        :constraints(
            (series a b c)
            (after (and (handempty) (clear ?x) (on ?x ?z)) a)
        )
    )
)
```