Improving Sample Strategies for Conformant Planning

Xiaodi Zhang
u6368740
COMP8755

Supervised by
Dr. Alban Grastien

May 30, 2019

A thesis submitted in part fulfillment of the degree of

Master of Computing
College of Engineering and Computer Science
The Australian National University
For Alban Grastien,
who designed the project, supervised Java
coding, discussed experiment results, guided
report writing

For Enrico Scala,
who developed CPCES and taught me how to
use CPCES

For Weifa Liang,
who supervised initial presentation and pro-
vided useful suggestions
Abstract

The planning problem in Artificial Intelligence is a task of searching a series of actions executed by intelligent agent. Planning techniques have been applied to various fields, such as robotics, automatic cars, and spacecraft mission control. Conformant planning, a branch of planning problem, makes decisions under uncertainty whose initial state, effects of action, or environment is unknown. To solve conformant planning problem, Grastien and Scala created a technique named CPCES, which searches a valid plan until no counter-example can be found. However, the efficiency of CPCES can be improved.

This report introduces the background of classical planning problem, search algorithms and conformant planning problem at the beginning. Then in the following pages, this report focuses on a current method created by Grastien and Scala which is named CPCES. Finally, I report my research project which extends CPCES by dividing predicates into several tags and ensuring counter-example has as many new tags as possible. According to the experiment results, my method does improve the efficiency for vertical instances.

Keywords: conformant planning, SAT, SMT, CPCES, tag
## Contents

1 Introduction .................................................. 1
   1.1 Motivations ........................................... 1
   1.2 Objectives ........................................... 2
   1.3 Report Structure ..................................... 2

2 Background .................................................. 3
   2.1 Classical Planning Problem ............................ 3
      2.1.1 Backgrounds of Classical Planning Problem ........ 3
      2.1.2 State Space Representation ......................... 3
      2.1.3 STRIPS Representation ............................. 5
      2.1.4 PDDL ............................................ 6
   2.2 Search Algorithms ..................................... 7
      2.2.1 BFS ............................................. 7
      2.2.2 DFS ............................................. 8
      2.2.3 A Star Search .................................... 10
   2.3 Classical Planning Problem Planner ................. 12
      2.3.1 State Space Planning ............................. 12
      2.3.2 Graphplan ....................................... 13
      2.3.3 SAT Planning .................................... 13
      2.3.4 Plan-space Planning ............................... 13
   2.4 Conformant Planning .................................. 14
      2.4.1 What is Conformant Planning Problem? ............ 14
      2.4.2 Mathematical Formulation of Conformant Planning Problem ........ 14
   2.5 CPCES ................................................... 15
      2.5.1 What is CPCES? ................................... 16
      2.5.2 How to computing a Counter-example ............. 18
      2.5.3 Why CPCES Works? ................................ 19

3 Improving CPCES by Using Superb Solver .............. 21
   3.1 Why CPCES can be improved? ......................... 21
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Example of Good Counter-Example</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>The Definition of Context, Tag and Superior Counter-example</td>
<td>23</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Context</td>
<td>23</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Tag</td>
<td>25</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Superior Counter-example</td>
<td>25</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Example of Superior Counter-Example</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>How to Improve a Counter-Example?</td>
<td>26</td>
</tr>
<tr>
<td>3.4.1</td>
<td>The General Improvement of Algorithm</td>
<td>26</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Compute a Better Counter-Example</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>My Coding Works</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>Experiments</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>Other Conformant Planning Techniques</td>
<td>30</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Conformant Planning via Symbolic Model Checking</td>
<td>30</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Conformant Planning via Heuristic Forward Search</td>
<td>33</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Conformant Planning via Heuristic Search and Symbolic Model Checking</td>
<td>34</td>
</tr>
<tr>
<td>4.1.4</td>
<td>T1 Planner</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Expectations</td>
<td>35</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Definition of Vertical and Horizontal Instances</td>
<td>35</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Expectations for Horizontal Instances</td>
<td>36</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Expectations for Vertical Instances</td>
<td>36</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Expectations for the Time Consuming Comparing SUPERB and T1</td>
<td>36</td>
</tr>
<tr>
<td>4.3</td>
<td>Experimental Setup</td>
<td>37</td>
</tr>
<tr>
<td>4.4</td>
<td>Results and Analysis</td>
<td>37</td>
</tr>
<tr>
<td>4.4.1</td>
<td>The Number of Iterations</td>
<td>37</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Plan Length</td>
<td>38</td>
</tr>
<tr>
<td>4.4.3</td>
<td>The Actual Elapsed Time</td>
<td>39</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Counter-Example Improving Time</td>
<td>41</td>
</tr>
<tr>
<td>5</td>
<td>Conclusion and Future Work</td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td>Conclusion</td>
<td>43</td>
</tr>
<tr>
<td>5.2</td>
<td>Future Work</td>
<td>44</td>
</tr>
</tbody>
</table>
# List of Figures

1. Example of PDDL language. ................................................. 6  
2. An example of BFS. .......................................................... 8  
3. An example of DFS. .......................................................... 9  
4. An example of A star search (from Youtube CS188 SP14 Lecture 3 Informed Search). .......................................................... 12  
5. An example of conformant planning problem [1]. ......................... 14  
6. The dependency networks of DISPOSE problem with two objects  
   to be disposed and a $2 \times 2$ grid map. ................................. 24  
7. The automation for BUTC domain ........................................... 31  
8. The automation for BUTC domain ........................................... 32  
9. Forward and backward heuristic search .................................. 34  
10. The number of iterations for horizontal, vertical and UTS instances. 38  
11. The plan length for horizontal, vertical and UTS instances. ........... 39  
12. The actual elapsed time for horizontal, vertical and UTS instances. 40  
13. The actual elapsed time between T1 planner and SUPERB. ............ 41  
14. Comparing counter-example improving time and actual elapsed time  
   for horizontal and vertical instances. .................................... 42
List of Tables

1 Bad and good executions of gCPCS: $L_i$ is the initial location of the $i$th item in the counter-example; each column describes a counter-example generated by gCPCS at a given iteration. . . . . . . . . . . 23
2 An example of conformant planning problem. There are three contexts: $c_1 = \{A, B, C\}$, $c_2 = \{D, E\}$, $c_3 = \{F, G\}$ . . . . . . . . . . . . 26
3 An example of Superior counter-example. There are three contexts: $c_1 = \{A, B, C, D\}$, $c_2 = \{E\}$, $c_3 = \{C, E, F, G\}$ . . . . . . . . . . . . . 28
Chapter 1

Introduction

This chapter provides a description of the motivation why I want to do this project. The objectives of this project is listed in the second section. Also, the structure of this report is given in the last section.

1.1 Motivations

Planning problem is a decision making process. The solution is a series of actions followed by which the agent can go to the goal state. Classical planning problem is a class of planning problems that the environment is completely observable, and the effects of actions are completely certain. In the last half century, several efficient classical planning algorithms have been developed, such as State Space Planning [2], Graphplan [3], SAT Planning [4], and Plan-space planning [5].

There is another kind of planning problem: conformant planning problem. In conformant planning problem, the initial state is unknown. The solution should satisfy all the possible initial states, which dramatically increases the complexity. So, finding an algorithm that search a valid plan in a short time is more difficult.

Grastien and Scala created CPCES to solve conformant planning problem [6][1]. CPCES repeatedly searches a counter-example, a subset of belief state, to negate the previous plan, until no counter-example can be found. CPCES has been proved to be sound and complete. CPCES has also been proved to be faster than T1 planner if solving a vertical instance. However, the approach to search counter-example in CPCES is random, which is not the most clever way. It is necessary for us to find a better approach to find a counter-example because it may increase the efficiency. Therefore, in this project, an improved counter-example strategy is expected.
1.2 Objectives

This project is aimed to improve CPCES. CPCES continuously searches counter-examples to excludes invalid plans. However, the counter-example is generated randomly, which is not efficient. My work is first studying what counter-examples can be regarded as good counter-examples. This can be achieved by defining contexts that connects all the relevant predicates in the belief state. Then, my work is to do some researches about how to use tags to improve counter-examples. The experiments consists of comparing the running time, the number of iterations, and the plan length under different conditions. Finally, I hope to give some explanations in terms of the experiment results.

1.3 Report Structure

The report structure is like follows. Chapter 2 contains a brief introduction of classical planning problem, search algorithms, approaches to solve classical planning problems, and conformant planning problem. This chapter ends with the algorithm of CPCES. In Chapter 3, I explain the basic idea of improving counter-example searching method, and introduce how do I write my code. Chapter 4 is to analyze the experiment results. I discuss the results in four points: the number of iterations, plan length, the actual elapsed time and counter-example improving time. The report ends with a conclusion and future work.
Chapter 2

Background

This chapter introduces classical planning problem and search algorithms first and
displaces how to use search algorithms to search a valid plan. Then, we introduced
conformant planning problem and a technique. In the final section, we focus on
CPECs which is an efficient technique to find a plan for conformant planning problem.

2.1 Classical Planning Problem

2.1.1 Backgrounds of Classical Planning Problem

Classical planning problem is about determining a sequence of actions performed
by an agent who tries to reach goal states. Classical planning problem is completely
deterministic: the initial state is certain; the effects of actions are fixed; the agent is
able to observe the environment. Classical planning problem is common in the real
world. For example, automatic car determines the direction and the speed under the
fully observation of the environment; robot decides a series of actions to complete
tasks; human makes daily schedule to achieve working goals in the company. The
techniques to solve classical planning problem provides us with efficient methods to
decide a plan and finish a task.

2.1.2 State Space Representation

Each classical planning problem must be represented by mathematical formulation. State space representation is one the most famous ways. A classical planning
problem $P$ consists of five elements $P = \langle S, A, \gamma, s_0, S_G \rangle$. I will provide detailed
explanations of these five elements.
$S$ is a set of states $s$ which represents the environment. The environment in the classical planning problem is of much importance, because environment continuously changes during the performance of the agent. To easily represent the environment and trace the changes, each state $s$ in the environment is represented by a set of Boolean variable $v$: if the environment is consisting of this state, we assign True to $v$; on the other hand, if this state does not hold, $v$ equals to False. For example, $s = \{\text{on\_table}(x) = \text{True}, \text{holding}(x) = \text{False}\}$ means a block is on the table and the hand of the agent is empty. Because the state continuously changes after applying actions, there are at most $2^{|v|}$ states in the problem, and we put all the possible states into a set $S$.

The agent in classical planning problem has some designated actions and we use a set $A$ to represent them. $A = \{\text{pickup}(x), \text{putdown}(x)\}$ means two actions can be done by agent: one is picking up the block from table, and another is putting the block on the table. It is worth noting that some actions can be applied only when current state satisfies specific requirements. For instance, $\text{pickup}(x)$ is applicable only when $\text{on\_table}(x) = \text{True}$ holds, and $\text{putdown}(x)$ can be applied when $\text{on\_table}(x) = \text{False}$ holds. Each action has the ability to change the states of current environment. For example, $\text{on\_table}(x) = \text{True}$ changes to $\text{on\_table}(x) = \text{False}$ after $\text{pickup}(x)$ being applied.

$\gamma$ is a transition function $\gamma : S \times A \Rightarrow S$. As is mentioned before, applying an action changes the states. For example, in BLOCKWORLD instance, $\text{on\_table}(x) = \text{True}$ changes to $\text{on\_table}(x) = \text{False}$ after applying action $\text{pickup}(x)$. So, we can write $\{\text{on\_table}(x) = \text{True}\} \times \{\text{pickup}(x)\} \Rightarrow \{\text{on\_table}(x) = \text{False}\}$. When solving the problem, we just search the transition function, looking for the applicable actions.

$s_0$ and $S_G$ represent the initial state and the goal state, respectively. Initial state describes the environment at very beginning of the problem. $S_G$ is a set of goal states. The reason why there may be more than one goal state is that we only need to satisfy some variables in the state. For example, two blocks $A$ and $B$ are on the table, and the goal is to hold block $A$. All the states with $\text{holding}(A) = \text{True}$ can be regarded as goal states. So, $S_G = \{\{\text{on\_table}(A) = \text{False}, \text{holding}(A) = \text{True}, \text{on\_table}(B) = \text{True}, \text{holding}(B) = \text{False}\}, \{\text{on\_table}(A) = \text{False}, \text{holding}(A) = \text{True}, \text{on\_table}(B) = \text{False}, \text{holding}(B) = \text{True}\}\}.$
The solution of a classical planning problem is a valid plan or returning no solution if no valid plan can be found. A plan $\pi$ is a sequence of actions $\pi = a_1...a_n$. A valid plan a kind of plan following by which the agent can reach the goal state. In specific, the problem starts from initial state $s_0$. When we apply $a_1$ to $s_0$, it transits to state $s_1$ according to transition function $\gamma$, and $s_1$ transits to $s_2$ when applying $a_2$, and so on. After applying all the actions in a valid plan $\pi$ step by step, the agent reaches one of goal states $S_G$, and the problem is solved.

2.1.3 STRIPS Representation

If there are $n$ variables in the state, there will have $2^n$ states. It will spend long time searching so many states. To reduce the number of states, STanford Research Institute Problem Solver (STRIPS) representation can be considered, because STRIPS is more compact [7]. STRIPS representation is an action-centric representation which specifies the precondition and the effect of an action. A classical planning problem can be represented by a tuple $P =< V, A, s_0, S_G >$.

Different from state space representation whose state is represented by several Boolean variables, the state $s$ in STRIPS is a set of variables that are True. We store all the variables in a set $V$. For instance, If two blocks $A$ and $B$ are on the table, the variable $v$ is $on\_table(x)$, the domain of variable is $D_v = \{A, B\}$, the set of variable is $V = \{on\_table(A), on\_table(B)\}$, and the state is $s = \{on\_table(A), on\_table(B)\}$.

$A$ is a set of actions. The action $a \in A$ is also different from state space representation. In STRIPS, we say an action is a pair of precondition and effect: $a = ( \text{pre}, \text{eff} )$. Precondition $\text{pre}$ is a set of positive state variables. Only when all the predicates in the precondition are also at the current state can the action $a$ be applicable. There will have some effects $\text{eff}$ after doing an action. $\text{eff}$ is a set of state variables that changes in the resulting state. If a predicate that is not in the current state enters the next state after an action is applied, we say this is a positive effect $\text{eff}^+$. On the other hand, if a predicate is in the current state but not in the next state after doing an action, we say this is a negative effect $\text{eff}^-$. The transition
function in state space representation can be represented by STRIPS in Formula 2.1.

$$\gamma(s, a) = \begin{cases} 
(s \setminus \text{eff}^-(a)) \cup \text{eff}^+(a), & \text{if } \text{pre}(a) \subseteq V \\
\text{undefined otherwise, } & \text{action not executable}
\end{cases} \quad (2.1)$$

$s_0$ is initial state and $S_G$ is a set of goal states. The problem starts from initial state and is expected to reach the goal state.

A plan $\pi = a_1 \ldots a_n$ is a sequence of actions. If this sequence of actions can be applied to state $s$, state $s$ can be transferred to $s[\pi] = s[a_1] \ldots [a_n]$.

### 2.1.4 PDDL

Inspired by STRIPS representation, the Planning Domain Definition Language (PDDL) is created by Drew McDermott and his colleagues in 1998. This language supports STRIPS and many extensions. It defines all the predicates (state variables), actions (with precondition and effects), initial state, and goal state. Combining with an instance (domain of state variable), we can search the plan.

```
(define (domain dispose)
  (:requirements :strips :typing)
  (:types pos obj)
   (trash_at ?x - pos) (disposed ?o - obj))
  (:action move
   :parameters (?i - pos ?j - pos )
   :precondition (and (adj ?i ?j) (located ?i))
   :effect (and (not (located ?i)) (located ?j)))
  (:action pickup
   :parameters (?o - obj ?i - pos)
   :precondition (located ?i)
   :effect (when (obj_at ?o ?i) (and (holding ?o) (not (obj_at ?o ?i)))))
  (:action drop
   :parameters (?o - obj ?i - pos )
   :precondition (and (located ?i) (trash_at ?i))
   :effect (when (holding ?o) (and (not (holding ?o)) (disposed ?o)))))
```

Figure 1: Example of PDDL language.
2.2 Search Algorithms

I have introduced what is classical planning problem and used mathematical formula to represent it. To solve a planning problem, we need to search a plan. In this chapter, several important and most useful search algorithms will be introduced, such as Breadth First Search (BFS), Depth First Search (DFS), A star search, and forward checking.

2.2.1 BFS

BFS, like its name, is a horizontal search for a tree data structure. It starts at the root node, exploring all the successors of at current depth before moving to the nodes in the next depth. To implement BFS, a First-In-First-Out data structure Queue should be used. The program pops out the first element from the Queue, checking whether this node is a goal state. If it is a goal state, the path from initial state to goal state will be returned. However, if it is not a goal state, we expand this node, get all the successors, and push successors into the Queue. We repeat this process until goal state is found (Algorithm 1). BFS is complete, because the algorithm can always find a goal state at a specific depth.

Algorithm 1 BFS

1: function BFS(problem)
2:   Let Q be a queue
3:   Let Explored be a set
4:   Q.push(problem.initial_state)
5: while Q is not empty do
6:   s = Q.pop()
7:   Explored.add(s)
8:   if s is in problem.goal_state then return s
9:   end if
10:  for all frontier in s.adjacent_nodes do
11:    if frontier is not in Explored then
12:      Q.push(frontier)
13:    end if
14:  end for
15: end while
16: return No Solution
17: end function
Figure 2 shows an example of BFS. The algorithm looks at root node “0” first, getting child nodes “1” and “2” and push them into the Queue. Then node “1” is popped from Queue, exploring its neighbours “3” and “4”, pushing them into the Queue. This process stops when a goal state is found.

![Figure 2: An example of BFS.](image)

### 2.2.2 DFS

Different from BFS, DFS does not search horizontally but vertically. It explores a branch as far as possible before backtracking. To implement DFS, a First-In-Last-Out data structure Stack should be used. The algorithm removes the first element in the stack, checking whether it is a goal state. If so, the algorithm returns a path from initial state to goal state. If not, the algorithm explores successors, and adds successors into the Stack (Algorithm 2). If this is an infinite large tree structure in which the goal state is not at the most left branch, the algorithm will never find a goal state (suppose the algorithm always looks at the left branch first). Accordingly, DFS is an incomplete algorithm.
Algorithm 2 DFS

1: function DFS(problem)
2:     Let $S$ be a stack
3:     Let $Explored$ be a set
4:     $S$.add(problem.initial_state)
5:     while $S$ is not empty do
6:         $s = S$.remove()
7:         $Explored$.add($s$)
8:         if $s$ is in problem.goal_state then return $s$
9:         end if
10:         for all frontier in $s$.adjacent_nodes do
11:             if frontier is not in $Explored$ then
12:                 $S$.add(frontier)
13:             end if
14:         end for
15:     end while
16:     return No Solution
17: end function

Figure 3 shows an example of DFS. The algorithm first looks at the root node “0”, adding node “1” and “2” into the Stack. Then node “1” is removed from the Stack, exploring all the successors (node “3” and “4”), and adding them into the Stack. The algorithm may stop when a goal state is found, or may infinitely run.

![Figure 3: An example of DFS.](image)
2.2.3 A Star Search

BFS and DFS are uniformed search because we regard all the step costs are 1. However, in the real world, the step cost may be different. To solve this kind of problems, informed search algorithms are developed. A star search is one of the most powerful algorithms in informed search.

Compared with DFS and BFS which searches the nodes in sequence (horizontal or vertical), A star algorithm considers the costs so far and evaluates the possible costs in the future, and selects a path that minimizes:

\[ f(n) = g(n) + h(n) \]  \hspace{1cm} (2.2)

where \( g(n) \) is the real costs from initial state to current state, and \( h(n) \) is the heuristic function that evaluates the possible costs from current state to goal state \[8\]. It should be noticed that \( h(n) \) should not be greater than the real cost from current state to goal state. We call it as admissible heuristic function (Formula 2.3).

\[ \forall h(n) \leq h^*(n) \]  \hspace{1cm} (2.3)

where \( h^*(n) \) is the real cost from \( n \). If heuristic function is admissible, A star search can find the optimal solution.

Heuristic function can be stronger. We call a heuristic function is consistent if

\[ h(n) - h(n') \leq c(n, a, n') \]  \hspace{1cm} (2.4)

where \( c(n, a, n') \) represents the costs from node \( n \) to node \( n' \). It can be proved that if heuristic function is consistent, it must be admissible and \( f(n) \) must be non-decreasing along any path:

\[
\begin{align*}
  f(n') &= g(n') + h(n') \\
  &= g(n) + c(n, a, n') + h(n') \\
  &\geq g(n) + h(n) \\
  &= f(n)
\end{align*}
\]  \hspace{1cm} (2.5)

To implement A star algorithm, we need to use a priority Queue, which always pop out the elements with the smallest priority. If the heuristic function is admissible but inconsistent, because the \( f(n) \) is not a strictly non-decreasing function, the priority Queue must be reopened when a new state has smaller priority than the
same node in the Queue or in the explored nodes set. Reopen ensures to find the optimal solution. However, if heuristic function is consistent, because $f(n)$ is strictly non-decreasing function, priority Queue does not need to be reopened (Algorithm 3).

Algorithm 3 A Star Search

```plaintext
1: function A-star(problem)
2:     Let $Q$ be a priority queue.
3:     Let Explored be a dictionary.
4:     $Q$.push(problem.initial_state, problem.initial_state.priority)
5:     while $Q$ is not empty do
6:         $s = Q$.pop()
7:         Explored[$s$] = $s$.priority
8:         if $s$ is in problem.goal_state then return $s$
9:         end if
10:     for all frontier in $s$.adjacent_nodes do
11:         if frontier not in Explored and frontier not in $Q$ then
12:             $Q$.push(frontier.frontier.priority)
13:         end if
14:         if frontier in Explored and frontier.priority < Explored[frontiers]
15:             then
16:                 $Q$.push(frontier.frontier.priority)
17:                 Explored.delete(frontier)
18:             end if
19:             if frontier in $Q$ and frontier.priority < $Q$.getPriority(frontier)
20:                 then
21:                     $Q$.updatePriority(frontier.frontier.priority)
22:                 end if
23:         end for
24:     end while
25:     return No Solution
26: end function
```

Figure 4 is an example of A star search. The red value near the circle is the heuristic value of each node, while the black value on the arrow is the real cost from one node to a neighbor node. The algorithm starts from initial state $S$ where $f(n) = g(n) + h(n) = 0 + 2 = 2$, and push node $A$ (priority = $1+4=5$) and node $B$ (priority = $1+1=2$) into the priority Queue. Since $B$ is the node with smallest priority, we look at $B$ first, and push its neighbor node $C$ (priority = 4) into the
Queue. Then, we look at C which has the smallest priority in the Queue, and push G (priority = 6) into the Queue. After it, node A is popped out. Notice that this time C appears again, but the priority of new node C is smaller than the previous node C, we reopen the Queue and push C into the Queue again. Finally, we look at new node C and then node G, and get the goal state. As you can see, we reopen the priority Queue in the algorithm because heuristic function is not consistent, and we get the shortest path S → C → G. If we don’t reopen, we will get another solution (S → B → C → G) which is not optimal.

Figure 4: An example of A star search (from Youtube CS188 SP14 Lecture 3 Informed Search).

2.3 Classical Planning Problem Planner

2.3.1 State Space Planning

Planning researchers have developed several techniques to solve classical planning problems with high efficiency. One of the simplest classical planning problem algorithms is state space planning [2]. State space planning algorithm transfers a planning problem to a search problem: nodes are labelled by states of the world; actions decide successor states; a valid plan is from the root node (initial state) to a leaf node (goal state). Due to this smart idea, state space planning algorithm can use various search strategies, such as breadth first search algorithm, depth first search algorithm, and A star algorithm. Forward checking algorithm and backward tracing algorithm can also be used in state space planning. To design a good heuristic function in a forward or backward search, state space planning
sometimes chooses to delete the relaxation, ignoring negative effects of every action. In another word, once a predicate is true, it remains true forever. State space planning has been proved useful in solving classical planning problems.

2.3.2 Graphplan

Although state space planning works in various planning problems, it wastes time examining many different orderings of the same set of actions. Another short coming of state space is it does not exploit the structure of states when not using heuristic function. Graphplan algorithm, developed by Avrim Blum and Merrick Furst in 1995 [3], solves the disadvantages of space state planning. Graphplan algorithm first captures information of mutually exclusive actions and propositions to reduce the amount of search needed to find a plan, only considering the propositions that can be true and the actions that can be applicable in each step. Then, it extracts parallel plan by suing backward search. If no valid plan exists, Graphplan algorithm extends the planning graph and duplicates the previous steps. Graphplan is proved to be sound but may not be complete.

2.3.3 SAT Planning

The third planning algorithm is SAT planning. SAT planning converts instance of the planning problem into an instance of the Boolean satisfiability problem [4]. It uses logical formula constraints to limit the value of propositions. For example, all the propositions that are true in the initial state must hold at step 0, and all the goal propositions must be hold at step \( k \) (See formula 1). For each action at step \( k \), the preconditions must be true at step \( k \), the positive effects should be true at step \( k+1 \), and the negative effects should be false at step \( k+1 \) (See formula 2).

2.3.4 Plan-space Planning

Compared with state space planning which searches through graph of nodes, plan-space planning searches through graph of partial plans. In plan-space planning [5], nodes are partially specified plans and the successors are determined by plan refinement operations. It keeps searching until a partial plan is found which has no flaw. Plan-space planning algorithm is sound and complete.
2.4 Conformant Planning

2.4.1 What is Conformant Planning Problem?

Conformant planning is a kind of planning problem under uncertainty. In conformant planning problem, agent is unable to observe the environment, and the initial state is uncertain. The solution (valid plan) for conformant planning problem should satisfy all the possible initial states. Accordingly, it is hard to find a plan than classical planning problem due to the number of initial states is exponential in the number of state variables.

Figure 5 is an example of conformant planning. An agent is in the grid but where it is in the initial state is unknown. Four actions are defined: go_North, go_South, go_East, and go_West. If the agent touches the wall when applying an action, the agent will not move. No matter where the agent is at the beginning, we should find a plan performed by the agent so that it goes to the destination (2, 2). To solve this question, we can ask agent go_North four times and go_East four times so that the agent must be at (4, 4) after doing these 8 actions. This is because if the agent touches the wall, the agent cannot move over the wall and go forward. Then we apply go_South twice and go_West twice to the agent to reach the destination.

![Figure 5: An example of conformant planning problem [1].](image)

2.4.2 Mathematical Formulation of Conformant Planning Problem

In conformant planning problem, because the initial state is uncertain, we call initial state as belief state $\beta$. $\beta$ is a set of state variables that evaluate to True at the beginning of the problem. Conformant planning problem can be represented by STRIPS $P = < V, A, S_I, S_G >$. 
• $V$ is a set of state variables. The same as classical planning problem, each variable $v \in V$ is a Boolean value.

• $A$ is a set of actions. Each action $a$ is pair $< prec, eff >$ where $prec$ is the precondition and the $eff$ is the effect. Different from the classical planning problem, the effect in conformant planning is conditional effect, because the state is uncertain before applying an action. Only if all the conditions are satisfied can the effects happen. For example, in the $BLOCKWORLD$ domain (see Figure 1), the conditional effect of action $pick\_up(x)$ is:

$$(\text{when } (\text{obj\_at } ?o \ ?i) \ (\text{and } \text{holding } ?o) \ (\text{not } (\text{obj\_at } ?o \ ?i))))$$

This conditional effect says only if the object and the agent is at the same position can the agent hold the object.

• $S_I$ is no long one initial state, but a set of belief states $\beta$. The initial state is uncertain, so there are $2^{|V|}$ belief states in total. Compared to classical planning problem which has only one initial state, conformant planning should consider all the $2^{|V|}$ belief states. This suggests that solving a conformant planning problem is more difficult than solving a classical planning problem.

• $S_G$ is a set of goal states which is the same as the $S_G$ in the classical planning problem.

A plan $\pi = a_1 \ldots a_n$ is a sequence of actions. If this sequence of actions can be applied to state $s$, $s$ will be transferred to $s[\pi]$: $s[\pi] = s[a_1] \ldots [a_n]$. A plan can be regarded as a valid plan for a state $s$ if this plan leads to a goal state: $s[\pi] \in S_G$. To solve a conformant planning problem, all the belief states should be satisfied by a valid plan: $\beta[\pi] = \{s[\pi] | s \in \beta\}$. Accordingly, we say a plan $\pi$ is valid for problem $P$ if it is valid for all initial states: $S_I[\pi] \in S_G$.

### 2.5 CPCES

It is impractical for us to use traditional search algorithms, such as BFS, DFS or A star search, to solve a conformant planning problem, because it takes much long time when considering various belief states. So, developing an efficient way to solve conformant planning problem is necessary. Grastien and Scala created a method named CPCES in 2017 [6] which searches for a valid plan efficiently.
2.5.1 What is CPCES?

CPCES replaces the belief state with a small number of initial states called the sample $B$. It always uses a sample to find a candidate plan $\pi$ that is valid for the sample $B[\pi] \in S_G$. If no such plan exists, it means the problem is unsolvable. A candidate is only valid for a specific sample rather than for all the belief states, so a counter-example $B'$ may be found where $B'[\pi] \notin S_G$. Then, CPCES searches for another candidate plan $\pi'$ that satisfies $B'[\pi'] \in S_G$. By implementing this procedures several times, when no counter-example can be found, the last candidate plan $\pi_n$ must be a valid plan for all belief state: $\beta[\pi_n] \in S_G$. This process is shown on Algorithm 4.

<table>
<thead>
<tr>
<th>Algorithm 4</th>
<th>The conformant planner CPCES.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: <strong>input</strong>: conformant planning problem $P$</td>
<td></td>
</tr>
<tr>
<td>2: <strong>output</strong>: a conformant plan, or <strong>no plan</strong></td>
<td></td>
</tr>
<tr>
<td>3: <strong>belief</strong>: := $\emptyset$</td>
<td></td>
</tr>
<tr>
<td>4: <strong>loop</strong></td>
<td></td>
</tr>
<tr>
<td>5: $\pi$ := <strong>produce-candidate-plan</strong>($P$, <strong>belief</strong>)</td>
<td></td>
</tr>
<tr>
<td>6: <strong>if</strong> there is no such $\pi$ <strong>then return</strong> No plan</td>
<td></td>
</tr>
<tr>
<td>7: <strong>end if</strong></td>
<td></td>
</tr>
<tr>
<td>8: $s$ := <strong>generate-counter-example</strong>($P$, $\pi$)</td>
<td></td>
</tr>
<tr>
<td>9: <strong>if</strong> there is no such $s$ <strong>then return</strong> $\pi$</td>
<td></td>
</tr>
<tr>
<td>10: <strong>end if</strong></td>
<td></td>
</tr>
<tr>
<td>11: <strong>belief</strong> := <strong>belief</strong> $\cup$ {$s$}</td>
<td></td>
</tr>
<tr>
<td>12: <strong>end loop</strong></td>
<td></td>
</tr>
</tbody>
</table>

2.5.1.1 How to Produce a Candidate Plan?

In CPCES, a conformant planning is reduced to $k$ classical planning problem and then merges them to one huge classical planning problem, in which the initial state, actions, preconditions, effects, and goal state are replaced by a notation with $q_i$. The candidate plan should valid for this huge classical planning problem. The detailed explanation is provided below.

Suppose there are $k$ different initial states in $B$, and we write $\{q_1, \ldots, q_k\} = \text{states}(B)$ as a set of $k$ initial states of $B$ where each $q_i$ is a formula. In order to be consistent with the notation from Palacios and Geffner, we use "fact" $f$ to represent a variable. We say $f/q_i$ is True if and only if $f$ is True in the initial state
$q_i$, and we say $F/q_i$ is a set of facts that is True in the initial state $q_i$.

A conformant planning problem $P = < F, A, B, S_G >$ now can be reduced to $k$ classical planning problem $red(P) = < F', A', B', S'_G >$ with the following definitions:

- $F' = F/q_1 \cup ... \cup F/q_k$. A conformant planning problem is transformed to $k$ classical planning problems running in parallel. The initial state of each classical planning problem is in fact one of belief states in conformant planning problem. The solver searches a plan that valids for all $k$ initial states. Accordingly, we put all the initial states into one set $F'$.

- $A' = \{ red(a) | a \in A \}$ where $red(< p, eff >) = < p', eff' >$. We define $p' = \bigwedge_{i \in \{1,...,k\}} p[F/F/q_i]$ and $eff(l/q_i) = eff(l)[F/F/q_i]$. This definition ensures that in each classical planning problem, the actions, preconditions, and effects are extended from conformant planning problem.

- $B' = \bigwedge_{i \in \{1,...,k\}} q_i[F/F/q_i]$. There are $k$ classical planning problem, so we put $k$ initial states into a set $B'$.

- $S'_G = \bigwedge_{i \in \{1,...,k\}} S_G[F/F/q_i]$. There are $k$ classical planning problem, so we put $k$ goal states into a set $S'_G$.

### 2.5.1.2 The Idea of Generating a Counter-example?

A SAT problem consists of a set of Boolean variables $V$ and a set of Boolean formula. The solution for a SAT problem is a valuation (True or False) for each variable $v$ that makes all Boolean formula be True.

For each action $a_i$ occurring at step $i$, the precondition of $a_i$ must be hold at state $q_i$ and its positive effects must be True at state $q_{i+1}$. The negative effects should be False at state $q_{i+1}$. We use $v@i$ to represent the variable at state $q_i$. So, we can write a constraint:

$$ a_i \rightarrow ( \bigwedge_{v \in \text{pre}(a)} v@i \land \bigwedge_{v \in \text{eff}^+(a)} v@i + 1 \land \bigwedge_{v \in \text{eff}^-(a)} \neg v@i + 1) \quad (2.6) $$

We also know that all predicates that are true at the initial state $s_0$ must be hold at step 0, and all predicates that are true at the goal state $s_g$ must be hold at step $k$. Therefore, another constraint is defined:
\[ \bigwedge_{v \in s_0} v \land \bigwedge_{v \notin s_0} \neg v \land \bigwedge_{v \in s_g} v \]  \hspace{1cm} (2.7)

Notice that it is easy for us to use SMT solver than to use SAT solver to solve a SAT problem.

CPCES generates a SAT problem when searching a counter-example. Suppose a plan \( \pi = a_1, ..., a_k \) is a sequence of actions, and it transfers the state from \( q_1 \) to \( q_k \). The main idea to find a counter-example is to break the correct constraint and generate a special constraint based on which the solution is not a valid plan: either the final state is not the goal state \( \Phi_G \), or action \( a_i \) is not applicable at state \( q_i \) (precondition \( \text{pre}_i \) of action \( a_i \) is unjustifiable):

\[ (\neg \text{pre}_1) \lor ... \lor (\neg \text{pre}_k) \lor (\neg \Phi_G [V/V_k]) \]  \hspace{1cm} (2.8)

Then, CPCES adds this special constraint, calling for SMT solver to find an initial state (counter-example) for which the candidate plan is invalid.

### 2.5.2 How to computing a Counter-example

The procedure of computing a counter-example is the same as what Grastien and Scala did in 2017. Adding enough constraints to SMT, SMT can find a satisfied counter-example.

We define \( \pi = a_1, ..., a_n \) is a candidate plan to transfer the state from \( q_0 \) to \( q_n \) where \( q_0 \) is a belief state and \( q_n \) is a goal state. To find a counter-example is to find a belief state which makes \( \pi \) becomes invalid: at least one specific action \( a_i \) is not applicable at a state \( q_i \) or \( q_n \) does not satisfy the requirement of goal state. We assume a SAT variable \( v_i^l \) is true if the value of state variable \( v \) in state \( q_i \) is \( l \), where \( v \in V, l \in D_v \), and \( I \) represent the time step \( i \in \{1, ..., n\} \). Three constrains must be considered to find a counter-example:

- \( \Phi_I \) constraints the initial state \( q_0 \);
- \( \Phi_A \) constraint the states before and after an action. The state \( q_i \) is the result of applying action \( a_i \) to state \( q_{i-1} \);
- \( \Phi_{\text{notapp}} \) constraints either one of actions is not applicable or the goal is unreachable.
If there is a counter-example to plan $\pi$, $\Phi_\pi = \Phi_I \land \Phi_A \land \Phi_{val}$ can be true for a variable $v_i$ where $i=0$.

To explain it with more details, here is an example: Suppose there are four state variables $V = \{W, X, Y, Z\}$, and a candidate plan is $\pi = a_1, a_2$. Suppose the initial states must satisfy $S_B = W \land (X \lor Y)$, and the goal states is $S_G = Z$. Suppose the precondition of action $a_1$ and $a_2$ is $W$, and the effect of action $a_1$ and $a_2$ is $Y$. The constraints should be constructed as:

- To limit the initial states, $\Phi_I = W^0 \land (X^0 \lor Y^0)$;
- We know variable $Z$ can be true after doing action $A$ under the condition of $W : W \rightarrow Z$. Accordingly, $W \rightarrow Z$ and $Z \rightarrow Z \lor W$ must be satisfied for all the timestamp:
  $$\Phi_A = (\neg W^0 \lor Y^1) \land (\neg Y^1 \lor Y^0 \lor W^0) \land (\neg W^1 \lor Y^2) \land (\neg Y^2 \lor Y^1 \lor W^1)$$
- To find a counter-example, we must find a counter-example that prevent the candidate plan $\pi$ from reaching the goal states or a counter-example that can one of actions cannot be applicable: $\Phi_{val} = \neg W^0 \lor \neg W^1 \lor Z^2$;
- $\Phi_\pi = \Phi_I \land \Phi_A \land \Phi_{val}$ and we put these constraints into the SMT solver.

### 2.5.3 Why CPCES Works?

We say a sample $B$ dominates another sample $B'$ if its valid plan set is a subset of those $B_1 \subseteq B_2$. We say a sample $B$ is dominant-preserving if the counter-example in each iteration is dominant all the previous counter-examples. We write $\Pi(P)$ is a set of valid plan for the problem $P$, and we write $\Pi(B)$ is a set of valid plan for sample $B$ where $B \in S_I$. CPCES works because it always relies on a property: $B_1 \subset B_2 \Rightarrow \Pi(B_2) \subset \Pi(B_1)$. In other words, if a sample provides more constraints, the candidate plan satisfies this sample is more dominant. During the implementation of CPCES, the sample $B$ becomes more and more restricted, and eventually $\Pi(B) = \Pi(P)$. So, CPCES is dominant-preserving, and it can find a valid plan.
Grastien and Scala have proved that CPCES is sound and complete and performs better than other standard algorithms for problems which have a non-trivial width.
Chapter 3

Improving CPCES by Using Superb Solver

This chapter first illustrate the reason why CPCES can be improved, and what a good counter-example should be. Then I give the definition of context, tag, and superior counter-example (Superb). Next, I introduce how to compute Superb and illustrate the algorithm of generating Superb. Finally, I list all my coding works.

3.1 Why CPCES can be improved?

CPCES can be improved because the counter-example generated in each round are randomly selected. As is mentioned in 5.1.2, when there are few samples, large number of counter-examples can be found. SMT picks up one of them randomly. Some counter-examples are only able to exclude one invalid plan, while others may be able to exclude several invalid plans. To increase the efficiency of CPCES, it is necessary to find a counter-example in each round that can remove as many invalid plans as possible.

3.2 Example of Good Counter-Example

Here I use DISPOSE problem (see Figure 1) to explain what is a good counter-example. In DISPOSE problem, the agent is expected to pick up the object, and drop it to the trash. There are three actions:

- move: The agent can move inside the grid.
• **pickup**: If the location of the object and the agent is the same, then the object will be held by the agent after applying this action.

• **drop**: If the location of the trash and the agent is the same, then the object will be disposed after applying this action.

Suppose there are 3 objects and 4 locations (A to D) in DISPOSE problem. Table 1.1 shows a bad sequence of counter-examples. In the first counter-example, all three objects are located at A. When searching a candidate plan, planner does not consider the possibility of location B, C, and D, and the candidate plan generated by planner only just visits location A to pick up three items. In second counter-example, item 1 is located at B and other items are at location A. This counter-example helps a little to improve the candidate plan, because even though the planner will consider both location A and B, the candidate plan only picks up one item at location B. Agent will not try to pick up three items at location B. Then, we continuously search counter-examples, and in each counter-example only one object changes the location. These bad counter-examples improve the candidate plan slowly, since only after 10 counter-examples can the planner considers three objects may be distributed on four different locations.

On the other hand, when we look at Table 1.2, which displaces good counter-examples, we will find these counter-examples improve candidates quickly and efficiently. Just after four counter-examples being generated, planner can know three objects may be distributed at all four locations. The reason why it is efficient is that the location for one object in a new counter-example is different from all the locations in previous counter-examples. For example, planner has considered object 1 may be at location A and B in the first two counter-example, so in counter-example 3, the location of object 1 is C, which is different from previous.

This example inspired me a possible way to generate good counter-examples: we can divide the state variables into several groups, and we hope we can update as many groups as possible in the new counter-example.
<table>
<thead>
<tr>
<th>#</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1$</td>
<td>$A$</td>
<td>$B$</td>
<td>$A$</td>
<td>$A$</td>
<td>$C$</td>
<td>$A$</td>
<td>$A$</td>
<td>$D$</td>
<td>$A$</td>
<td>$A$</td>
</tr>
<tr>
<td>$L_2$</td>
<td>$A$</td>
<td>$A$</td>
<td>$B$</td>
<td>$A$</td>
<td>$A$</td>
<td>$C$</td>
<td>$A$</td>
<td>$A$</td>
<td>$D$</td>
<td>$A$</td>
</tr>
<tr>
<td>$L_3$</td>
<td>$A$</td>
<td>$A$</td>
<td>$A$</td>
<td>$B$</td>
<td>$A$</td>
<td>$A$</td>
<td>$C$</td>
<td>$A$</td>
<td>$A$</td>
<td>$D$</td>
</tr>
</tbody>
</table>

Table 1: Bad and good executions of gCPCES: $L_i$ is the initial location of the $i$th item in the counter-example; each column describes a counter-example generated by gCPCES at a given iteration.

3.3 The Definition of Context, Tag and Superior Counter-example

3.3.1 Context

We define a subgoal $\phi$ is a conjunct of preconditions of an action or the goal. A subgoal is a logical formula that must be evaluated to True before an action is done. For example, in DISPOSE problem with one object $o_1$, suppose the trash is at (1,1). One subgoal can be a conjunction of preconditions of action drop:

$$located_{o_1,11} \land trash_{at,11}$$  \hspace{1cm} (3.1)

We use $(v, l)$ to represent an assigned variable, where $v$ is the conditional variable and $l$ is the value of $v$. We use $\text{eff}(v, l)$ to represent the conditional effect of an action. We say $(v, l)$ depends on $(v', l')$ if the conditional effect $\text{eff}(v, l)$ mentions $(v', l')$. Recall the DISPOSE problem with only one object $o_1$, action drop has conditional effects:

$$(\text{when } (\text{holding } ?o) \text{ (and } (\text{not } (\text{holding } ?o)) \text{ (disposed } ?o)))$$

This can be written as:

$$\text{eff}(\text{holding}, o_1) \Rightarrow (\text{disposed}, o_1)$$

which means $(\text{disposed}, o_1)$ depends on $(\text{holding}, o_1)$.\n
The dependency has transitivity. If \((v, l)\) depends on \((v', l')\) and \((v', l')\) depends on \((v'', l'')\), then we say \((v, l)\) also depends on \((v'', l'')\). Based on transitivity, we can find a dependency network of a subgoal.

A context of subgoal \(ctx(\phi)\) is a set of predicates that contains all the predicates of a dependency network of a subgoal. When computing the contexts of an instance, if we find a context is a subset of another context, the smaller one will be regarded as a redundant one which should be removed. After remove all the redundant contexts, we use a set \(C\) to store all the necessary contexts.

For example, if we want to compute the context of an instance from domain \(DISPOSE\) with two objects to be disposed and a \(2 \times 2\) grid map, the dependency networks are shown on Figure 6.

Figure 6: The dependency networks of \(DISPOSE\) problem with two objects to be disposed and a \(2 \times 2\) grid map.

So there are two contexts:

\[
\text{disposed}_o1 \\
\downarrow \\
\text{holding}_o1 \\
\downarrow \\
\text{object}_{at\_o1\_11} \quad \text{object}_{at\_o1\_12} \quad \text{object}_{at\_o1\_21} \quad \text{object}_{at\_o1\_22} \\
\downarrow \\
\text{disposed}_o2 \\
\downarrow \\
\text{holding}_o2 \\
\downarrow \\
\text{object}_{at\_o2\_11} \quad \text{object}_{at\_o2\_12} \quad \text{object}_{at\_o2\_21} \quad \text{object}_{at\_o2\_22}
\]
• Context 1 = \{\textit{dispose}_o1, \textit{holding}_o1, \textit{located}_o1_{11}, \textit{located}_o1_{12}, \textit{located}_o1_{21}, \textit{located}_o1_{22}\}

• Context 2 = \{\textit{dispose}_o2, \textit{holding}_o2, \textit{located}_o2_{11}, \textit{located}_o2_{12}, \textit{located}_o2_{21}, \textit{located}_o2_{22}\}

3.3.2 Tag

We use \(c\) to denote a context, and use \(s\) to denote a state, the tag of \(q\) for \(c\) is \(\text{tag}_c(q) = s \cap c, \) For instance, if a state is \{\(A,B,C,D\)\} and a context is \{\(A,E\)\}, the tag is the intersection of two sets \{\(A\)\}. We use \(T_c(B) = \{\text{tag}_c(q) | q \in B\}\) to represents all the possible tags for context \(c\), where \(B\) is a belief state. Because there may be more than one context in a conformant planning problem, we use \(T(B) = \bigcup_{c \in C} T_c(B)\) to represent all the tags in the problem. Each tag \(t\) can be associated with a set of \(\Pi(t)\) so that for all belief state \(B\), the following formula holds:

\[
\Pi(B) = \bigcap_{t \in T(B)} \Pi(t) \tag{3.2}
\]

This is because if a plan \(\Pi\) is invalid, one of subgoals must be failed, which suggests one of tags in the context of this subgoal is not does not satisfy this plan. In other words, if a plan \(\Pi\) is valid, all the tags should satisfy this plan.

3.3.3 Superior Counter-example

We say a counter example \(q'\) is better than another counter-example \(q\) if more tags are new to history tags:

\[
T(B \cup \{q\}) \subseteq T(B \cup \{q'\}) \tag{3.3}
\]

Based on formula 3.2, the more tags we had used in searching counter-examples, the more invalid plans have been excluded. So, we hope when searching a counter-example, we can use as many new tags as possible. According to this reason, we say Formula 3.3 holds.

We use the notation \(q' \geq Bq\), and we say counter-example \(q'\) is superior to counter-example \(q\). Based on the following two lemmas, it can be proved that superior counter-example is more easily to find a valid plan and increase the efficiency.
Lemma 1 If \( q' \) is superior to \( q \) for belief state \( B \), \( \Pi(B \cup \{q\}) \subseteq \Pi(B \cup \{q'\}) \).

If a plan satisfies a superior counter-example \( q' \) which is improved from another counter-example \( q \), this plan must satisfy \( q \), because \( q' \) provides more limitations. In another word, \( B \cup \{q\} \) dominates \( B \cup \{q'\} \).

Lemma 2 If \( q' \) is superior to \( q \) for belief state \( B \), then for all set of states \( B' \), \( q' \) is superior to \( q \) for belief state \( B \cup B' \).

This lemma can be proved by mathematical derivation: \( T(B \cup \{q\} \cup B') = T(B \cup \{q\}) \cup T(B') \subseteq T(B \cup \{q'\}) \cup T(B') = T(B \cup \{q'\} \cup B') \).

3.3.4 Example of Superior Counter-Example

As is shown on Table 2, an instance has three context: \( c_1 = \{A, B, C\}, c_2 = \{D, E\}, c_3 = \{F, G\} \). One counter-example \( CE_1 \) has been created when searching the first candidate plan: \( CE_1 = \{A, \neg B, \neg C, D, E, F, \neg G\} \). The tags are: \( T(CE_1) = \{t_1, t_2, t_3\} \) where \( t_1 = \{A\}, t_2 = \{D, E\}, t_3 = \{F\} \). Then, we find \( CE_2 \) invalidates the previous candidate plan: \( CE_2 = \{A, B, \neg C, D, \neg E, F, \neg G\} \). However, the new \( t'_3 = \{F\} \) is the same as \( t_3 \). So, \( CE_2 \) is not a superior counter example. If the second counter-example \( CE'_2 = \{A, B, \neg C, D, \neg E, F, G\} \), all three new tags are different from previous tags. So, \( CE'_2 \) is a superior counter-example.

<table>
<thead>
<tr>
<th>Counter-example</th>
<th>tag 1</th>
<th>tag 2</th>
<th>tag 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( CE_1 )</td>
<td>{A}</td>
<td>{D, E}</td>
<td>{F}</td>
</tr>
<tr>
<td>non-optimal ( CE_2 )</td>
<td>{A, B}</td>
<td>{D}</td>
<td>{F}</td>
</tr>
<tr>
<td>optimal ( CE_2 )</td>
<td>{A, B}</td>
<td>{D}</td>
<td>{F, G}</td>
</tr>
</tbody>
</table>

Table 2: An example of conformant planning problem. There are three contexts: \( c_1 = \{A, B, C\}, c_2 = \{D, E\}, c_3 = \{F, G\} \)

3.4 How to Improve a Counter-Example?

3.4.1 The General Improvement of Algorithm

The algorithm to generate superior counter-example is shown on Algorithm 5. The algorithm searches a counter-example first. If no such counter-example exists,
the plan is valid. Otherwise, the algorithm uses a loop function to improve the counter-example gradually. When no better counter-example can be found, this counter-example will be regarded as a superior counter-example.

**Algorithm 5** Computing an optimal counter-example for $\pi$ ($\perp$ indicates a failure to find a solution).

1: **input**: candidate plan $\pi$, current sample belief
2: $q := \text{compute-counter-example}(\pi)$
3: if $q = \perp$ then return $\pi$ is valid
4: end if
5: loop
6: $q' := \text{improve-counter-example}(\text{belief}, q)$
7: if $q' = \perp$ then return $q$
8: end if
9: $q := q'$
10: end loop

The algorithm is guaranteed to terminate. We know that state $q'$ is strictly superior than $q$, and we know that $q'$ contains more new tags than $q$: $T(q) \setminus T(B) \subseteq T(q') \setminus T(B)$. The maximum number of tags of a state is limited by the number of subgoal. In other words, the number of loops can only up to the number of subgoals of a state. Therefore, algorithm 5 is guaranteed to terminate.

### 3.4.2 Compute a Better Counter-Example

We have known that the more new-tags are, the better the context is. To find a superior counter-example, some more constraints must be added into SAT solver ($q' \geq Bq$). We use $\Phi \geq B$ to specify that the new tags of $q$ will be kept:

$$\bigwedge_{c \in C_{\text{new}}(B,q)} \text{tag}_c(q)[V/V_0]$$

and use $\Phi \leq B$ specify there are at least one new tag:

$$\bigvee_{c \in C \setminus C_{\text{new}}(B,q)} \neg \bigvee_{t \in T_c(B)} t[V/V_0]$$

Where $t[V/V_0]$ is the conjunction of predicate $v^0_t$ of a tag $t$. 
We define $C_{\text{new}}(B, q)$ is a set of context in which the tag of $q$ is not in a tag of $B$:

$$C_{\text{new}}(B, q) = \{ c \in C | \text{tag}_c(q) \notin T_c(B) \}$$  \hspace{1cm} (3.6)

If context $q'$ is superior to context $q$, new tags must come from $C_{\text{new}}(B, q)$. In a context $c$, the new tag $t' = \text{tag}_c(q')$ in $q'$ is not in $T_c(B)$. So, the new tag in $q'$ if $\neg \forall t \in T_c(B) \exists t'[V/V_0]$.

**Lemma 3** There exits a state $q'$ that is strictly superior to $q$ for $B$ iff $\Phi_{B,q}$ is specifiable, and one such state is represented by the variable $v_i^1$ where $i = 0$.

There are two reasons why Lemma 3 holds. First, if context $q'$ is superior to context $q$, then $q'$ must contain new tags $T(q') \setminus T(B)$ comparing to $q$. This is proved by $\Phi \leq B$. Second, if there exists a context $q'$ that is strictly superior to $q$, then $q'$ must contain at least one more tag. This is proved by $\Phi \not\leq B$.

<table>
<thead>
<tr>
<th>Counter-example</th>
<th>tag one</th>
<th>tag two</th>
<th>tag three</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CE1$</td>
<td>${A, C}$</td>
<td>${E}$</td>
<td>${C, E, G}$</td>
</tr>
<tr>
<td>$CE2$</td>
<td>${B, C}$</td>
<td>${E}$</td>
<td>${C, E}$</td>
</tr>
<tr>
<td>$CE3$</td>
<td>${A, B, C}$</td>
<td>${E}$</td>
<td>${C, E, G}$</td>
</tr>
<tr>
<td>$CE4$</td>
<td>${A, B, C}$</td>
<td>${E}$</td>
<td>${C, E, F}$</td>
</tr>
</tbody>
</table>

Table 3: An example of Superior counter-example. There are three contexts: $c_1 = \{A, B, C, D\}$, $c_2 = \{E\}$, $c_3 = \{C, E, F, G\}$

Suppose $V = \{A, B, C, D, E, F, G\}$, and there are three contexts: $\{A, B, C, D\}$, $\{E\}$, $\{C, E, F, G\}$ as is shown on Table 2. Two counter-examples $CE1$ and $CE2$ have been generated, and we are searching the third counter-example. We assume that $CE3$ has a new $tag1 = \{A, B, C\}$, but $tag2$ and $tag3$ in $CE3$ is the same in $CE1$.

Because we hope there are as many new-tags as possible in $CE3$, a new constraint must be added into SAT solver:

$$\Phi_I \land (\Phi \geq B) \land (\Phi \not\leq B)$$  \hspace{1cm} (3.7)

where

$$\Phi \geq B = tag11$$  \hspace{1cm} (3.8)
and

\[ \Phi \not\models B = \neg(tag21 \lor tag22) \lor \neg(tag31 \lor tag32) \] (3.9)

The result of it is \( CE3' \) where only \( tag2 \) is not new tag, and a better counter-example is expected. We repeat the same procedure, but not better counter-example can be found, because \( context2 \) only has one variable which will never be improved. So, \( CE3' \) is a superior counter-example.

### 3.5 My Coding Works

My coding work is based on the work of Grastien and Scala who wrote CPCES program (Java language). I write four classes (Context, Tag, ConstraintByCounterExample, and ComputeContext), and modified two classes (SampleGenerator and main).

- To improve the counter example, we are required to compute contexts and tags, so two more classes named Context and Tag are created by me. In these two classes, I use a set to represent a context/tag in which all the relevant predicates are stored;

- Because a context is computed by the dependencies of state variables, and redundant contexts should be removed, I write a class named ComputeContext to compute the dependencies, contexts and tags;

- Grastien and Scala have written an abstract class named SampleGenerator that generates a counter-example and searches candidate plan. I write ContextSampleGenerator class which extends SampleGenerator class to inherit all the functions. To generate superior counter-examples, I modify computeSingleCPCESCCounterExample and callSMTSolver based on AlgorithmX;

- Since several constraints should be added into SAT solver to compute a superior counter-example, I write a class named ConstraintByCounterExample to generate constraints based on history; tags;

- I modify the main class to compute context, tags, and call ContextSampleGenerator class to solve the problem by using superior counter-examples.
Chapter 4

Experiments

This chapter first define what is vertical instance and what is horizontal instances. Then some expectations are listed on this chapter. This chapter ends with the experiment results and analysis.

4.1 Other Conformant Planning Techniques

4.1.1 Conformant Planning via Symbolic Model Checking

Conformant Planning via Symbolic Model Checking is created by Cimatti and Roveri in 2000 [9]. The basic idea is to use Binary Decision Diagram to represent the conformant planning problem and use BFS and backward search algorithm to find a plan.

Here we use the same example used in their paper. In BTUC domain, there are two packages, one of which contains an armed bomb. The goal is to dunk the package in the toilet and defuse the bomb. There are three state variables: $In_i$ means the bomb is in package $i$; $Defused$ means the bomb has been defused; $clogged$ means the toilet is clogged. Two actions are in this domain. $Dunk_i$ is to dunk package $i$ with uncertain effect: either the toilet is clogged and the bomb is defused, or the toilet is normal and the bomb is defused. $Flush$ is to flush the toilet. It has a conditional effect: when the toilet is clogged, $Flush$ can smooth the toilet. $BTUC$ domain can be represented in a Finite State Automata (Figure 7).
The algorithm uses BFS and backward search algorithm. It first builds Belief state-Plan (BsP) tables (Figure 8). BsP tables start from the goal states (here goal states are state 5 and state 7 which are put in one table) and set it as level 0. Then the algorithm tries to find which actions can lead to level 0, and find the possible states in level 1. By repeatedly doing backward search, the algorithm can finally find a table contains all the belief states at level $x$, and the actions from level $x$ to level 0 is a valid plan for this problem.
The algorithm uses propositional formula to represent each belief state in the BsP table. However, it is not a good way to enumerate all the belief states, so it uses Binary Decision Diagram (BDD) to represent belief states, which is more compacted and increases the efficiency of backward search dramatically.

This algorithm has several advantages that other conformant planners do not have. First of all, this algorithm guarantees to return a plan with minimal length if there exists a solution. This is because the algorithm uses BFS to search the plan, and BFS ensures to find a best solution. accordingly, this algorithm has high efficiency. It transfers belief states to BDD, which reduces redundant branches
to get a compact diagram. Since redundant belief states have been removed, it saves amount of time. Thirdly, this algorithm is very general. It just uses BFS and backward search, combining with BDD to compact belief states. However, there are still some disadvantages. For example, the time complexity and the space complexity of BFS is exponential. So, when solving a complex question, this algorithm may be failed.

4.1.2 Conformant Planning via Heuristic Forward Search

Neither backwards search nor forward search is efficient to extract a valid plan. Based on the idea introduced in Chapter 2, a heuristic function can estimate the cost and avoid searching nodes that are unlikely to be in the solution. However, designing a good admissible heuristic function is a trade-off process: a good heuristic function needs more information which makes the algorithm be slow, while a bad heuristic function ignores many information but the algorithm is fast.

Hoffmann and Brafman created a conformant planner by using heuristic forward search [10]. They considered the trade-off in designing heuristic function, relaxing the problem, deciding to sacrifice the admissibility. In other words, they do not want an optimal solution, but hope to have a great runtime performance. They relax the problem by ignoring the negative effects as well as all but random one of the predicates. Suppose an action $a$ has a conditional effect with condition $c_1 \land ... \land c_i$, and the effects are $p_1 \land ... \land p_i \land n_1 \land ... \land n_i$, where $p_x$ is positive effect and $n_x$ is negative effect. After relaxing the problem by using their algorithm, the conditional effect has only one random condition $c_x$ and the effect is $p_1 \land ... \land p_i$. After relaxing the problem, the algorithm uses forward search (graph search, see Section 2.3.2) to find a plan. However, we have known that in graph search, one state can be reached via different search path. To further improve the runtime performance, the algorithm chooses to use a hash table technique combined with a belief state equivalence test to avoid repeated states.

As you can see, the advantages of this algorithm are excellent runtime performance. It simplifies the problem by ignoring all the negative effects and ignore all but one condition in the conditional effect. So, the heuristic function is easy to be computed. It further improves the graph search algorithm by avoiding repeated states, which speeds up the algorithm. However, due to the heuristic function is
not admissible, the plan is not the optimal plan (with minimal plan length). This is a trade-off. They sacrifice the optimal plan but get a fast speed to find a valid plan.

4.1.3 Conformant Planning via Heuristic Search and Symbolic Model Checking

I have introduced conformant planning via Symbolic Model Checking or Heuristic search. Bertoli and his colleague tried to combine these two approaches and they did it [11].

A heuristic search can be forward or backward, as is shown on Figure 9. A forward search starts from initial state to find a path to goal state. On the other hand, backward search starts from goal state, searching back to find a path satisfying all the belief states. This algorithm chooses to use the cardinality of belief states as the heuristic value, because they believe as more promising a belief state with a low degree of knowledge. This heuristic function is not admissible, but the experiment result shows this function is great enough. By using heuristic function in search method, the number of nodes expanded will be small. To further increase the efficiency, this algorithm prunes the node that has been visited. For example, in Figure 9, the node 6 in forward search is visited twice. We can prune one of branches.

![Figure 9: Forward and backward heuristic search](image)

To further compact the search space, this algorithm chooses to use BDD to represent each belief state. As is introduced in section 4.1.1, using symbolic model checking can increase the efficiency obviously.
This algorithm combines the advantages of both heuristic search and symbolic model checking, so it has a very high efficiency. The only thing that is missing is that the heuristic function is not admissible. So we may improve this algorithm by finding a heuristic function that is admissible.

4.1.4 T1 Planner

T1 planner is created by Albore and his colleagues [12]. It solves some conformant problems very quickly, especially when the width of the problem is one.

T1 planner is based on translation approach. However, unlike the other planners, this translation approach does not translate a conformant problem to a classical planning problem, but solves the problem by using heuristics and compute beliefs inside a standard belief-space planner. T1 planner first computes the subgoals (preconditions of actions and goals), and computes the context, the same as what SUPERB does. Then, it picks up $i$ predicates from the context, putting $i$ propositions into a set $t$, and $t$ is one of initial states. T1 planner computes all the sets of $t$, and finally uses almost the same procedures to find a valid plan as CPCES. Albore found when $i = 1$, T1 planner has a very high efficiency and can solve the problem in a very short time. On the other hand, when $i$ is greater than 1, the performance is bad. This experiment result suggests that T1 planner is powerful to solve the problem with width equals to 1 (we can also call it as vertical instances which is defined in the next section).

4.2 Expectations

Starting from this section, we call the improved CPCES as SUPERB.

4.2.1 Definition of Vertical and Horizontal Instances

We know that a context of subgoal $ctx(\Phi)$ is a set of predicates that contains all the dependencies in a subgoal. You may notice that in some contexts there exists uncertain initial values, but in other contexts all initial values are certain. For instance, in DISPOSE domain with two objects to be disposed and a 2 grid map (Figure 6), context1 and context2 are uncertain, because the location of object1 and object2 are unknown. However, no uncertain context exists in context3.
We define an instance is a vertical instance if all contexts have only one initial unknown variable. In vertical instances, at least two tags can be updated when searching superior counter-example. On the other hand, an instance is regarded as a horizontal instance if it features a context that spans over all state variables whose initial value is uncertain. In horizontal instances, only one tag can be updated when searching superior counter-example. For example, the \textit{DISPOSE} instance with two objects is vertical instance, but the \textit{DISPOSE} instance with one object is horizontal instance.

### 4.2.2 Expectations for Horizontal Instances

Since no counter-example can be improved in the horizontal instance, the superior counter-example is in fact the same as normal counter-example. Accordingly, the number of iterations should be the same as CPCES. However, when searching superior counter-example, some more steps are taken (line 5-9 in Algorithm 5). So, we expect that for superior counter-example, the time consuming is slightly longer than CPCES. Because the superior counter-example generated in horizontal instances should be the same as normal counter-example, the plan length should be the same.

### 4.2.3 Expectations for Vertical Instances

There are at least two uncertain contexts in vertical instance, so the counter-example can be improved quickly when applying our improve-counter-example function in Algorithm X. As is discussed in Chapter 2.6.2, even though considering a smaller number of counter-examples than CPCES, since we update as many tags as possible in each iteration, it is possible to find a valid plan in a smaller number of iterations. In addition, due to the high efficiency of superior, the time spending by superior should be less than CPCES. Different from the expectation for horizontal instances, because of the uncertainty of counter-example generated in each iteration, we cannot predict the length of the plan.

### 4.2.4 Expectations for the Time Consuming Comparing SU-PERB and T1

T1 planner is one of the fastest conformant planner, especially when the problem is a small vertical instance. Grastien and Scala has proved that T1 is much faster
than CPCES for small vertical instances. Here, we hope that SUPERB performs faster than T1 for some horizontal instances.

4.3 Experimental Setup

My code is developed from CPCES written by Grastien and Scala as is mentioned in 6.7. To compare the performance of superb and CPCES, I measure the actual elapsed time, the number of iterations, the plan length, and the counter-example improving time for vertical instances.

I use the same benchmarks as Grastien and Scala used. I also compare superb and T1 in terms of time consuming.

All experiments are run on Ubuntu. We allot 16 GB memory on a 3.6 GHz CPU.

4.4 Results and Analysis

The horizontal instances include all the instances of BLOCKWORLD, LOOKANDGRAB, RAOSKEY, WALLGRID, EMPTYGRID, and some instances in DISPOSE domain (one object). The vertical instances include all instances of SINK, WALL, BOMB, COINS, ONEWPOSE. We regard UTS instances separately, because even though UTS is vertical, but the performance is very poor, which is different from all the other vertical instances. Accordingly, 85 horizontal instances, 25 vertical instances, and 15 UTS instances are tested in the experiment.

4.4.1 The Number of Iterations

Figure 10 compares superb with CPCES in terms of the number of iterations. 25 vertical instances perform much better if we search superior counter-examples. Some instances are even 300\% times better than CPCES. Superior uses the same number of iterations to search a valid plan as CPCES does when they are solving horizontal instances. UTS instances, which should be regarded as vertical instances, perform the same as horizontal instances. This is a very strange phenomenon, and we are unable to provide any explanation. Accordingly, except for UTS, all the
instances are the same as our expectations.

![Graph](image.png)

Figure 10: The number of iterations for horizontal, vertical and UTS instances.

4.4.2 Plan Length

Figure 11 compares superb and CPCES in terms of plan length. We totally tested 125 instances, but there are only 8 instances get different plan length. The differences between them are tiny (normally has a difference of 1-2 steps). As what we expected, almost all the different instances are performed on vertical instances (7 in vertical instance and 1 in UTS instance). We are excited that 6 of 7 vertical instances find a shorter plan in superb. This phenomenon is out of our expectation, and it shows the superiority of superb in plan length to some extent.
4.4.3 The Actual Elapsed Time

Figure 12 compares superb and CPCES in terms of time consuming. For 25 vertical instances, superb consistently solve the problem in less time. Time cost savings in vertical instances ranges from 0% to 93%, which is obviously a significant improvement. When it comes to horizontal instances, it takes few more time by using superior than using CPCES. This difference can almost be ignored. However, UTS is very wired. The actual elapsed time spending by superb in UTS is much longer than CPCES. This is because it takes much longer time to improve the counter-example, which will be discussed in the next section. When it comes to the difference between T1 planner and SUPERB, we find for some horizontal instances,
SUPERB spends much less time than T1, but for all vertical instances (except for BOMB100 − 100), SUPERB performs much poor. This follows our expectations.

Figure 12: The actual elapsed time for horizontal, vertical and UTS instances.
4.4.4 Counter-Example Improving Time

Counter-example improving time only happens when using superior. For 85 horizontal instances, as is mentioned in 7.4.3, the total time cost by superior is a little longer than CPCES. The extra time is from counter-example improving time. As you can see in Figure 14, the time to improve counter-examples in horizontal instances occupies very low ratios in total time. This illustrates that the efficiency of the function “improve-counter-example” is high enough. For 25 vertical instances, even though superb spends more time in improving counter-examples, it does not influence the performance of high efficiency in solving vertical instances.
Figure 14: Comparing counter-example improving time and actual elapsed time for horizontal and vertical instances.
Chapter 5

Conclusion and Future Work

This chapter concludes the whole report by summarizing the background, my improving algorithm, and the experiment results. This chapter also introduces the future works.

5.1 Conclusion

Solving planning problem is a decision making process that finds a sequence of actions followed which the agent can go the goal state. If the environment is fully observable, and the effects of actions are known, it is called classical planning problem. A classical planning problem can be solved by using various search algorithms. On the other hand, if the environment is uncertain, initial state is unknown or the effects of actions are uncertain, this kind of problem is called conformant planning problem. A valid plan of conformant planning problem should satisfy all the possible initial states. Normally, to solve a conformant planning problem, we have to transform it to many classical planning problem, the number of which is based on the number of initial states. Therefore, it is much more complex to solve a conformant planning problem than solve a classical planning problem.

Grastien and Scala developed an approach named CPCES to solve conformant planning problem. CPCES continuously searches counter-example (a belief state) which does not satisfy the previous plan, until no counter-example can be found. This method has been proved completed and sound, and it shows an excellent performance in the experiments. However, the approach to find counter-examples is not clever, because it picks up counter-example randomly. This inspires me to improve CPCES.

I divided variables into different tags. The belief state is the conjunction of all
the tags, in which the variables are assigned Boolean values. My idea to improve CPCES is when CPCES is looking for a counter-example, the new counter-example should update as many tags as possible. I call this method as SUPERB.

I separated all the instances into three parts: horizontal instances, vertical instances, and UTS. I defined an instance is horizontal instance if there is only one tag contains uncertain variable, and an instance is vertical instance if two or more tags contain uncertain variables. According to the experiment results, I find for all vertical instances SUPERB takes smaller number of iterations than CPCES, and SUPERB spends less than that CPCES. I also find the plan length is almost the same no matter an instance is used by CPCES or SUPERB. When comparing the performance of SUPERB and T1 planner, I find SUPERB has a better performance for vertical instances, but T1 planner has better performance for horizontal instances. UTS, though they should be vertical instances, are very strange, because neither the number of iterations or the time consuming is worse than CPCES. The reason of this wired phenomenon is unknown.

5.2 Future Work

We have discussed in the previous section that UTS performs strange: though it is vertical, the number of iteration and the time are worse when using SUPERB. So, the first task in the future is to find the reason why UTS is wired.

Secondly, I find the plan length of SUPERB does not have huge change. This phenomenon does not mean that SUPERB does not have any advantages in improving plan length. In fact, if the shortest plan length for an instance is 100, CPCES finds a plan with 102 actions and SUPERB finds another plan with 101 actions, we can say the plan found by SUPERB improves 50%, which is a significant improvement. So, in the future, I can compute the shortest plan length for each instance, and then I am able to see whether the plan found by SUPERB has a huge improvement.

Finally, SUPERB now is only designed to solve conformant planning problem. However, if we add more features, it can be applied to broader areas. Probabilistic planning problem may be of one the candidate planning problem that can be solved by SUPERB. I will focus on probabilistic planning problem in the future, and try to use SUPERB to solve this kind of problems.
Bibliography


