The Australian National University
2600 ACT | Canberra | Australia

Australian National University

School of Computing

College of Engineering and Computer Science (CECS)

Transforming Partially Ordered Planning Problems into Totally Ordered Problems

— 24 pt Honours project (S1/S2 2022)

A thesis submitted for the degree
Bachelor of Advanced Computing

By:
Ying Xian Wu

Supervisors:
Dr. Pascal Bercher
Dr. Songtuan Lin

May 2022

Declaration of Authorship:

Except where otherwise indicated, this report is my own original work.

Saturday 7th May, 2022,

(Ying Xian Wu)

Acknowledgements

If you wish to do so, you can include some Acknowledgements here. If you don't want to, just comment out the line where this file is included.

There is absolutely no need to write an Acknowledgement section, so only do so when you want to – i.e., if there's somebody you really want to thank (for example if you received extraordinary supervision). The more important the work, the more likely that an Acknowledgement section doesn't look off. In a 24 pt. Honours thesis it would for example look more reasonable than for a 6 pt. project report.

Unrelated to the Acknowledgements, but important:

Note that there exist two different title page designs. You may choose the one that you find more appealing. Just use the configuration file to select this style. There, you also have to put in all the other information about this report like your name, the kind of report (Honours vs non-Honours) and so on.

Abstract

An abstract is a very short summary (around 15 lines) of your entire work (that doesn't use citations by convention). There are plenty of examples you can take a look at – simply take a look at some papers published at top-tier venues, e.g., by your supervisor.

Table of Contents

Chapter 1

Introduction

1.1 Introduction to HTN Planning

High-level introduction (without technical definitions) to research area that can be understood by anybody with some basic mathematical understanding.

1.2 Motivation

1.2.1 overview of potential pros and cons of total vs. partial order

- pros of partial order:
  - plan recognition: independent goals can be described in parallel (Daniel should be able to write something about that)
  - partial order is more expressive (both in terms of plan existence and in terms of computational complexity) meaning that more problems can be expressed
  - Domain model might be more intuitive: if a task is independent of some others it might be counter-intuitive to demand a certain position of it (if artificially made totally ordered)
- pros of total order:
  - computational complexity is lower (fewer worst-case solving time)
  - we can exploit specialised algorithms as well as heuristics. Note that heuristic design is comparably easy for total-order problems due to the missing interaction between tasks.

Further advantages from having a compilation of PO into TO plans

- We actually get another class of decidable partially ordered problems that's orthogonal to tail-recursive ones! (Because if the criterion 'matches', we know that the PO domain is equivalent to the resulting TO domain; the latter is decidable whereas the former is not.)
- We can use more efficient algorithms and heuristics.

1.2.2 POCL

1.3 Contributions

Though the algorithm existed and performed well in IPC contest, there does not exist empirical analysis of it's performance, or specification of it's formal properties. This paper provides both.

Chapter 2

Background

2.1 Classical planning

A classical planning problem is usually defined in the STRIPS formalism, as below:
Problem = (D, S_I, S_G)
D is the domain of the problem,
$S_I \in 2^F$ is the initial state and $S_G \in F$ is the goal.
Every fact $\in F$ included in S_G is true, and all other facts are false. This is the closed world assumption.

Domain D = (F, A)
F is a finite set of facts, or propositional state variables.
A is a finite set of actions.
Every $a \in A$ is of type $2^F \times 2^F \times 2^F$, and of the format $< pre, add, del >$. An action a is executable in a state s if its precondition *pre* holds in s, i.e. pre $\subseteq$ s.
If executable in s, its result is the successor state s' = (s \ del) $\cup$ add , i.e., delete effects get removed and add effects get added.

Solution: Solutions to a problem are action sequences executable in the initial state S_I that lead to a state s" that satisfies all goals, i.e., s" $\supseteq$ g. Any such state s" is called a goal state.

2.2 Partially Ordered Hierarchical Planning

Also known as Hierarchical Task Network Planning or HTN planning for short.

There are many formalizations for hierarchical planning. This one borrows heavily from "A Survey on Hierarchical Planning – One Abstract Idea, Many Concrete Realizations" by Bercher, Alford, Höller, except with the initial task network replaced by a initial compound task.

Problem $= (D, S_I, T_I)$ is over some domain D, has an initial state S_I, which is a total assignment to F, and has a initial compound task T_I.

D $= (F, T_P, T_C, \delta, M)$
F is the finite set of state variables, T_P is the finite set of primitive task names, T_C is the finite set of compound task names, and δ is a mapping from primitive task name to action. M is the finite set of decomposition methods. Each one maps a compound task name to a task network. T_P is the set of all possible primitive task names T_C is the set of all possible compound task names δ maps actions to it's (preconditions, add, deletes) $\in 2^F x \times 2^F \times 2^F$. This can alternatively be referred to as $prec(t), add(t), del(t)$ for $a \in F, t \in T_P$ M is a set of methods. If $m \in M$, then m maps a compound task to a Task Network.

Task Network $= (T, \prec, \alpha)$ T is a finite set of task ids, $\prec$ is a partial order over T, and α maps task ids to task names.

2.2.1 HTN Solution

The goal here is to refine this task network, such that a valid solution is obtained. A valid solution to a PO HTN problem a plan: - a sequence of primitive tasks/actions - The application of each action produces a corresponding state. The solution must be such that each action is applicable in the previous state. i.e. the corresponding state satisfies the precondition of the action (including the first action applicable in the initial state) - (Often, the final state produced will be checked against a 'goal state' encoded in a (artificially enforced) final primitive tasks' preconditions,?

2.3 Totally Ordered Hierarchical Planning

Totally ordered hierarchical planning is the same as partially ordered planning in all respects except the task network. Both define a **Task Network** $= (T, \prec, \alpha)$. The difference is that $\prec$ now specifies the order between task ids such that for every a, b, in T, there exists an ordering in $\prec$ that (a, b) or (b, a). In other words, the tasks are totally ordered.

2.4 Further Definitions

Undecidable A decision problem is equivalent to, a function that accepts a infinite (set?) of inputs and returns a yes or no. The decision problem is undecidable if it can be proved that there exists no algorithm for the problem that always leads to a correct yes-or-no answer.

Task Decomposition Graph A directed Graph (V, E) that represents a given domain. The nodes in V represent tasks T in

Task Decomposition Tree A tree of all possible decompositions given an initial task network and a domain.

Lifted Each object that exists in the world has a type from a pre-determined set of types defined by the domain. Methods apply to the set of object(s) that belong to a certain type.

Grounded A transition is ground if the parameters list only involves specific objects. Problems are grounded when all methods are grounded.

So to ground a problem: Let ω be a set of typed objects. The groundings of a transition schema a over ω is denoted by σ(a, ω) and corresponds to the set of all ground transitions obtained by substituting σ with a list of compatible objects taken from ω , and then substituting each occurrence of the variables which were in σ with the newly introduced objects.

Chapter 3

Related Work

Connie's Paper?

This chapter reviews the work that is most related to the research questions investigated by you in this work. Please note that there are various options on *where* you include it.

- You could include it *here* (i.e., where you see it right now in the template). Since it's after the formal definitions (Chapter 2), you can explain what the other works have done on some level of detail, yet you need to keep in mind that you did not yet explain your own contributions (except abstractly in the abstract), which slightly limits the level of technical detail on which you can compare these approaches here.
- You could also make it a subsection of Chapter 2. This choice might also depend on the length of this chapter. Is it worth its own full chapter?
- Alternatively, you might include this chapter after the main part of your report, i.e., right before Chapter 6. When you do this, you can go into more technical detail since the readers will have read your entire work, so they know exactly what you've done and you can therefore discuss differences (like pros/cons etc.) in more detail.
- When you take a look at scientific papers (preferably at top-tier venues), you might notice that not every single paper has a related work section. This is because in principle related works might also be addressed/positioned in the introduction or in the main part of the work. But since this is not a "standardized scientific publication", it is very strongly advised that you devote its own section to related work as done in this template.

If you prefer any of the latter two options, discuss this with your supervisor(s).

Chapter 4

Algorithm, Formalisation, Benchmark

4.1 Algorithm 1

For the Domain (F, T_P, T_C, δ, M) and Problem = (T_I, S_0, TN_G)

1. Consider each $t_c \in T_C$, and infer its (super-relaxed) preconditions and effects.
   a) For every method m that can decompose t_c, consider each sub-task t_i of m:
      i. If $t_i \in T_P$, add $prec(t_i), add(t_i), del(t_i)$ effects to the set of $prec^*c(t_c), add^*(t_c), del^*(t_c)$
      ii. If $t_i \in T_C$, find the actions it can decompose to and repeat this algorithm for it
2. Consider each method m $\in$ M independently
   a) Let the set of non-required edges be NS_m
   b) $\forall a \in F$
      - Add $\{(t,t')|(a \in add^*(t) \wedge a \in prec^*(t') \wedge t, t' \in tasks(m))\}$ to NS_m
      - Add $\{(t',t)|(a \in add^*(t) \wedge a \in del^*(t') \wedge t, t' \in tasks(m))\}$ to NS_m
      - Add $\{(t',t)|(a \in del^*(t) \wedge a \in prec^*(t') \wedge t, t' \in tasks(m))\}$ to NS_m
      - Add $\{(t,t')|(a \in del^*(t) \wedge a \in add^*(t') \wedge t, t' \in tasks(m))\}$ to NS_m
   c) Construct a directed graph G
   d) Add the required orderings $\prec \in tn(m)$ to G with weight=0
   e) Add the edges in NS_m to G with weight=1
   f) Use DFS to find the back edges on this directed graph.

g) For each back-edge e

i. if e can be deleted because it is not required by the PO domain, delete it

ii. if e=(A,B) can't be deleted, because it is required by the PO domain, use Dijkstra to find path from B to A, i.e. the other components of the cycle.

iii. Randomly pick an non-required (weight=0) edge in the (B, A) path and delete it

h) Topological sort the resulting graph. This order is the total order for this method.

3. Create a new domain $(F, T_P, T_C, \delta, M')$, where M' is the modified methods produced in step 2.

4.2 Solution preserving properties of Algorithm 1

From Bercher & Olz, AAAI-2020 POPOCL

POCL criterion? A partial POCL plan P = (PS, $\prec$, CL) is called POCL plan (also POCL solution) to a planning problem if and only if every precondition is supported by a causal link and there are no causal threats.

PO criterion We refer to a partial POCL plan P = (PS , $\prec$, CL) without causal links, i.e., CL = $\emptyset$, as a partial partially ordered (PO) plan. Due to the absence of causal links, the solution criteria are here defined directly by the desired property of every linearization being executable.

i.e. A partial PO plan P = (PS , $\prec$) is called PO plan (also PO solution) to a planning problem if and only if every linearization is executable in the initial state and results into a goal state. i.e. it has every necessary ordering.

A linearization exists if and only if a POCL solution exists.

Proposition 0: *Removes linearizations*
Proof. This algorithm linearises all the methods to be totally ordered. Since sub-tasks inherit the orderings of their parents, it's impossible to preserve a solution that requires the interleaving of sub-tasks if their respective parents that are already ordered with respect to each other. This proves that the algorithm will always remove some linearizations, assuming the original domain was not already totally ordered.

Consider the simple problem:

F = {a, b, c, g}
N_p = {T_A, T_B, T_C, G}
N_c = {AB }
$\delta = \{(T_A, A), (T_B, B), (T_C, C)\}$

M = $(AB, \{\{4,5\},\{\},\{(4,A),(5,B)\}\})$
$TN_Init = \{\{0,1,2\},\{(0,2),(1,2)\},\{(0,AB),(1,C),(2,G)\}\}$

$S_I = < a >$ A = <pre a, del a, add c>
C = <pre c, del c, add b>
B = <pre b, del b, add g>
G = <pre g, ,>

The initial task network enforces that G is the last action. To make G executable it needs the variable g, which only B can add. To make B executable it needs the variable b, which only C can add. To make C executable it needs the variable c, which only A can add. A is executable in the initial state. Therefore, the only solution is A C B G. This is impossible to achieve by linearizing methods, since either AB before C or C before AB, both of which exclude the solution.

This proves that the algorithm can remove all solutions, so Algorithm 1 is not complete.

Proposition 1: Soundness
Proof. We do not modify the sub-tasks a method produces, just the ordering between them, so the set of plans from the totally ordered method is just a subset of the plans possible from the partially ordered one. Any solution to the linearized problem is then obviously a solution to the original problem.

Proposition 2 *If Algorithm 1 didn't have to cycle-break, at least one solution is preserved*
Proof. Assume that there exists a solution in the PO domain. Using the same decomposition in the linearised domain, we can produce a linearization $(a_0, a_1, ..., a_n)$ of those actions in the PO solution. We then prove by induction over the sequence $(a_0, ..a_n)$ that it is executable.

If $(a_0, ...a_n)$ is not executable, that means there exists some action $a_k, 0 < k < n$ that is not executable in the corresponding state. However a_k must be executable in some linearization for $\{a_0, ..., a_n\}$, as we assumed it was a PO solution. So there must exist an action a_i, $0 < i < n$, that will adds A. Actions a_0 and a_k must have a shared parent p in TDG. Then p has subtasks t_0 and t_k that are parents of a_0 and a_k respectively.

The linearization of this method would have drawn an ordering (t_i, t_k) due to the way the algorithm defines $prec^*, add^*$ etc. We are assuming that all methods linearized without conflict, so (t_i, t_k) should not be required. This safely enforces (a_k, a_0) ordering in the final TO plan, meaning a_0 is not the first action in the resulting total order imposed by the algorithm. Therefore if a_k's precondition could be met by any action a_i, a_i would be ordered in front of it.

If a_i does not exist then a_k can never be executed for any linearization of $\{a_0, ...a_n\}$, contradicting the assumption that this was a PO solution. Since each action in the

solution is executable, the entire sequence is executable linearization of actions produced by decomposition of initial task, i.e. the solution.

2.1 *Even if Algorithm 1 didn't have to cycle break,* $\exists t.add^*(t)! = add(t) \vee del^*(t)! = del(t)$
Proof. Suppose there exists an compound task t whose method 1 decomposes to an action a, with $add(a) = \{A\}$. Assume there is another method which decomposes to action b, with $add(b) = \{B\}$ Therefore $prec^*(t) = \{A, B\}$ but both A and B, will not be applied in every execution of t.

2.2 *If Algorithm 1 didn't have to cycle break, there may be t such that* $prec^*(t)$
Proof. Assume some compound task t' decomposes into at least two sub-tasks, $\{t_1, t_2\}$. Suppose that $a \in F, a \in add(t_1)$ and $a \in F, a \in prec(t_2)$. By the algorithm rules

- Add $\{(t, t')|(a \in add^*(t) \wedge a \in prec^*(t') \wedge t, t' \in tasks(m))\}$ to NS_m
- Add $\{(t', t)|(a \in add^*(t) \wedge a \in del^*(t') \wedge t, t' \in tasks(m))\}$ to NS_m
- Add $\{(t', t)|(a \in del^*(t) \wedge a \in prec^*(t') \wedge t, t' \in tasks(m))\}$ to NS_m
- Add $\{(t, t')|(a \in del^*(t) \wedge a \in add^*(t') \wedge t, t' \in tasks(m))\}$ to NS_m

We know that t_1 is ordered before t_2. And by the previous proposition 2.1, we know that there does not exist a task t_3 such that $a \in del(t_3)$ Therefore, t_2 will always be executable regardless of what other tasks might do.

However, $a \in prec^*(t')$ because $a \in prec^*(t_2)$

Therefore it's possible for preconditions to be erroneous, even if Algorithm 1 did not need to cycle break.

Proposition 3: When it does have to cycle break, it may preserve solutions
Proof. As per proposition 2.1 and 2.2, not all of the preconditions and/or effects are always needed. Suppose the initial task network consisted of 2 sub-tasks, t_1, t_2. If t_2 can decompose into an action a_1 such that $add(a_1) = A$, and an action a_2 such that $del(a_2) = A$, and t_1 can decompose into an action a_3 with $prec(a_3) = A$ $t_2 = addA, delA$ and $t_1 = precA$. The rules $\{(addA, precA), (precA, delA))\}$ of Algorthm 1 require orderings $\{(t_2, t_2), (t_2, t_1), (t_1, t_2)\}$, i.e. Despite a cycle break being needed, this is obviously a solvable problem. Order t_2 before t_1 for this method. Then when solving decompose t_2 to a_1 and t_1 to a_3 - this creates a solution.

Proposition 4 *For some tasks it doesn't matter where they are executed*
Proof. Assume that some pair of tasks $(t_1), (t_2)$ has no ordering between them. Assume also that t_1t_2 is ok but $(t_2)(t_1)$ is not, i.e. it matters where they are executed.

The required execution sequence implies that t_2 deletes some variable A t_1 relies on. But from the algorithm, that would mean that an ordering (t_1, t_2) was created, which contradicts the assumption that there is no ordering between them. Therefore $(t_2)(t_1)$ must also be ok.

Thus if some task t has no ordering from the algorithm, it can never matter where it is in relation to any other task. The algorithm will not order it specifically when $(\forall t' \in m, \forall a \in prec(t).a \notin add(t') \wedge a \notin del(t')) \wedge (\forall t' \in m, \forall a \in add(t).a \notin prec(t') \wedge a \notin del(t')) \wedge (\forall t' \in m, \forall a \in del(t).a \notin add(t') \wedge a \notin prec(t'))$ In other words, the variables that affect t, do not affect other tasks in that method.

4.3 Algorithm 2

For the Domain (F, T_P, T_C, δ, M) and Problem = (T_I, S_0, TN_G)

1. Consider each $t_c \in T_C$, and infer its (super-relaxed) preconditions and effects.
   a) For every method m that can decompose t_c, consider each sub-task t_i of m:
      i. If $t_i \in T_P$, add $prec(t_i), add(t_i), del(t_i)$ effects to the set of $prec^*c(t_c), add^*(t_c), del^*(t_c)$
      ii. If $t_i \in T_C$, find the actions it can decompose to and repeat this algorithm for it
2. Consider each method m ∈ M independently
   a) Let the set of non-required edges be NS_m
   b) $\forall a \in F$
      - Add $\{(t,t')|(a \in add^*(t) \wedge a \in prec^*(t') \wedge t, t' \in tasks(m))\}$ to NS_m
      - Add $\{(t',t)|(a \in add^*(t) \wedge a \in del^*(t') \wedge t, t' \in tasks(m))\}$ to NS_m
      - Add $\{(t',t)|(a \in del^*(t) \wedge a \in prec^*(t') \wedge t, t' \in tasks(m))\}$ to NS_m
      - Add $\{(t,t')|(a \in del^*(t) \wedge a \in add^*(t') \wedge t, t' \in tasks(m))\}$ to NS_m
   c) Construct a directed graph G
   d) Add the required orderings $\prec \in tn(m)$ to G with weight=0
   e) Add the edges in NS_m to G with weight=1
   f) Use DFS to find the back edges on this directed graph.
   g) If there exist back edges, do not modify the method.
   h) Else topological sort the resulting graph. This order is the new order for this method.
3. Create a new domain $(F, T_P, T_C, \delta, M')$, where M' is the modified methods produced in step 2.

Proposition 5 *Algorithm 2 is sound*
Proof. Same as Proposition 1

Proposition 6 *Algorithm 2 is complete*
Proof. Assume that the original problem is solvable. This means that for some PO sequence of actions that we got from decomposition of an initial task, there exists a linearization of this that is executable. Assume this solution is $(a_0, a_1, ..., a_n)$. A linearization produces Assuming that a method is linearizable, Not even it's textitpossible preconditions and effects conflict in this sequence, so it must be executable.

If a method does require interleaving, it's executable sub-units can remain an unit. Why would we need to split m? If m's subtasks adds a then deletes a, and another action needs a. But then there would be conflict in m by algorithm definiton, (add) You must 1) add 1) 2) meet prec of those with a) and 3) delete a.

Actions only: Methods prioritise add variable (del then add ordering enforced) Given a compound task with only complete methods in it. If the effect of a compound task claims it can add a (even if it claims to delete it too), there exists a method that decomposes it to a linearised set of tasks that DEFINITELY ADD A.

If another task B needs a precondition b, there exists a linearization of A that adds it, so there is no need to interleave them. If it needs several, either 1 method does them all, OR the adds are split among different methods for A, in this case PO wouldn't help either. (a0 ... a1) (B) There will not exist a method that adds a, deletes a, adds b.

It will ONLY delete a precondition you were relying on if it was literally impossible for them to add it in the first place. If you need something that was true before they deleted it, just place yourself in front of them. We are not linearizing via Gregor's algorithm, due to this method having this conflict, so you can do that. Henceforth the task that this decomposes will have no guarantee that it will delete then add. Also note that you can add things between the other task B and linearised task A, in case you want to have task B effects but not task A effects.

Note that for a fully linearised method, if it's children could not all be fully linearised, then you cannot rely on it adding something in it's effects list. However, you can always just interleave yourself between its sub-tasks, as it will have partially ordered sub-tasks, or sub-sub-tasks etc.

4.4 Algorithm 3

The definitions in Conny's paper assumes that there are no negative preconditions. Strict guaranteed:

$$eff_*^+ := (\bigcap_{s \in E(c)} \bigcap_{s' \in R_s(c)} s') \backslash (\bigcap_{s \in E(c)} s)$$

$$eff_*^- := \bigcap_{s \in E(c)} (F \backslash \bigcup_{s' \in R_s(c)} s')$$

Strict possible:

$$poss - eff_*^+ := \bigcup_{s \in E(c)} (\bigcup_{s' \in R_s(c)} s' \backslash s)$$

$$poss - eff_*^- := \bigcup_{s \in E(c)} ((\bigcup_{s' \in R_s(c)} (F \backslash s')) \cap s)$$

Relaxation: Define a new domain such that A' = $\{\emptyset, add, del\} | \{\emptyset, add, del\} \in A$ Define the relaxed guaranteed and possible as before, under this new domain.

Since these effects will *definitely* happen, even if some methods cannot be linearised,

4.5 Algorithm 1

For the Domain (F, T_P, T_C, δ, M) and Problem = (T_I, S_0, TN_G)

1. Consider each $t_c \in T_C$, and infer its (super-relaxed) preconditions and effects.
   a) For every method m that can decompose t_c, consider each sub-task t_i of m:
      i. If $t_i \in T_P$, add $prec(t_i), add(t_i), del(t_i)$ effects to the set of $prec^*c(t_c), add^*(t_c), del^*(t_c)$
      ii. If $t_i \in T_C$, find the actions it can decompose to and repeat this algorithm for it
2. Consider each $t_c \in T_C$, and infer its (precondition-relaxed) effects as $add^*_\emptyset(t_c), del^*_\emptyset(t_c)$.
3. Consider each method m ∈ M independently
   a) Let the set of non-required edges be NS_m
   b) $\forall a \in F$
      - Add $\{(t,t') | (a \in add^*(t) \wedge a \in prec^*(t') \wedge t, t' \in tasks(m))\}$ to NS_m
      - Add $\{(t',t) | (a \in add^*(t) \wedge a \in del^*(t') \wedge t, t' \in tasks(m))\}$ to NS_m
      - Add $\{(t',t) | (a \in del^*(t) \wedge a \in prec^*(t') \wedge t, t' \in tasks(m))\}$ to NS_m
      - Add $\{(t,t') | (a \in del^*(t) \wedge a \in add^*(t') \wedge t, t' \in tasks(m))\}$ to NS_m
   c) $\forall a \in F$
      - Add $\{(t,t') | (a \in add^*_\emptyset(t) \wedge a \in prec^*(t') \wedge t, t' \in tasks(m))\}$ to NS_m
      - Add $\{(t',t) | (a \in add^*_\emptyset(t) \wedge a \in del^*_\emptyset(t') \wedge t, t' \in tasks(m))\}$ to NS_m
      - Add $\{(t',t) | (a \in del^*_\emptyset(t) \wedge a \in prec^*(t') \wedge t, t' \in tasks(m))\}$ to NS_m
      - Add $\{(t,t') | (a \in del^*_\emptyset(t) \wedge a \in add^*_\emptyset(t') \wedge t, t' \in tasks(m))\}$ to NS_m
   d) Construct a directed graph G

e) Add the required orderings $\prec \in tn(m)$ to G with weight=0

f) Add edges between definite adds and definite deletes, weight =1

g) Add edges between definite effects and possible preconditions, weight=2

h) Add edges between possible effects and possible preconditions, with weight=3

i) Use DFS to find the back edges on this directed graph.

j) For each back-edge e

i. if e weight == 4, delete it

ii. else use Dijkstra to find path from B to A, i.e. the other components of the cycle.

iii. Randomly pick an weight==3 edge in the (B, A) path and delete it. If it doesn't exist, search for weight=weight-1 to delete, etc,

k) Topological sort the resulting graph. This order is the total order for this method.

4. Create a new domain $(F, T_P, T_C, \delta, M')$, where M' is the modified methods produced in step 2.

SOUND, NOT COMPLETE PARTIAL LINEARIZATION: SOUND, COMPLETE

(To reduce the preconditions and effects requires cutting depth wise (e.g. a sequence of tasks s.t. add a, del a occurs in every possible sub branch) If ALL methods are possible, you cant exclude the possibility of their prec/eff. Otherwise it would imply knowing which method to take, which makes no sense?) The obvious solution to it's current deficiencies is:

1. When tasks need to be interleaved as part of it's solution, introduce new (totally ordered) methods that will allow this interleaving. If for example, The initial task network contains (AB, C') in that order. Then AB $\implies$ A, B. and C' $\implies$ C. And the only solution is A, C, B. We can detect a cycle (as in the example above) Then when we detect the cycle between AB and C', we need a new method such that their parent (the initial task) $\implies$ A,C,B in that order. This one seems more impractical to find, since in practice some solutions might need interleaving of tasks that are very, very far apart in a TDG. Also, if enough interleaving is required, this is equivalent to solving the problem.

   Depending on the problem, even this will not be able to remove all cycles. Some compound tasks will have *different* preconditions and effects depending on the method(s) applied to decompose it to a sequence of primitive tasks. If any number of these is conflicting with another tasks, we will still have cycles.

2. Because floating tasks can be executed where-ever, whenever there is such a floating task, we could produce multiple linearizations of a given method, such that every possible placement of the floating task is possible.

Chapter 5

Evaluation

These are some suggestions on what you may report on:

- Benchmark Selection: Which data set did you choose to evaluate your approach on? Why did you make that specific selection? Would there have been alternatives?
- Hardware setup: What hardware was used (processor, RAM, etc.), and which operating systems and software were used?
- Results: Report the plain data (plots, tables, etc.) and their *interpretation*, i.e., did the approach work well or not? Do we know on which subset it worked well (or badly) and why was it like this? Do the results raise further questions and thus directions for future research/investigations?

When reporting your results using graphs and plots, make sure to provide all information necessary to interpret the data, e.g., axis and graph labeling (cf. Figure **??**).

5.1 Empirical Evaluation

To prove that our technique is beneficial, we conduct a standard empirical evaluation on PO domains and compare the runtime of PO planners vs. transformation + TO planners

We should also do an evaluation that shows how often the criterion works, i.e., in how many instances (per domain) we can say 'yes: translate!'

5.1.1 Pre-processing performance

Rover					
Tasks	avg***	Methods	Linearizable Methods % *	Grounding**	Preprocessing**
396	2.24823	424	35.2727	0.049	0.004175
2258	2.18431	2589	25.1229	0.1288	0.037174
1484	2.20161	1866	39.6364	0.1101	0.023642
460	2.32727	441	25.3776	0.0283	0.006185
65	2.34694	50	35.1351	0.0119	0.001148
59	2.35556	46	40.625	0.0112	0.001594
1202	2.22602	1377	33.114	0.0767	0.029776
382	2.37709	359	30.303	0.0235	0.006008
561	2.28114	595	32.2115	0.0363	0.009243
626	2.30711	662	33.6323	0.0422	0.008775
89	2.47143	71	44.0	0.0132	0.001044
595	2.19498	678	34.4262	0.0364	0.010964
494	2.30469	513	32.7684	0.0292	0.013627
245	2.22321	225	24.1379	0.0177	0.02508
152	2.26016	124	21.5686	0.0141	0.004343
414	2.3222	420	33.5593	0.0255	0.006752
707	2.26154	716	22.7941	0.0367	0.010028
243	2.42857	218	33.5443	0.0187	0.002684
81	2.46667	61	40.0	0.0124	0.00088

Recursive: False

Woodworking					
Tasks	avg***	Methods	Linearizable Methods % *	Grounding**	Preprocessing**
1005	1.16356	1450	0.0	0.045	0.004655
29804	1.02155	129218	0.0	1.4686	0.562532
2748	1.11924	5620	0.0	0.0975	0.016805
896	1.13949	1528	0.0	0.0444	0.00505
93209	1.01596	485051	0.0	5.5487	4.02608
12	1.0	9	0.0	0.013	5.5e-05
75238	1.01763	373234	0.0	4.0672	2.19396
56590	1.02022	258945	0.0	2.8876	1.29956
21	1.0	15	0.0	0.0179	0.000101
82285	1.01898	390298	0.0	4.5634	2.7429
16	1.54545	12	0.0	0.0137	8.8e-05
964	1.1419	1798	0.0	0.0547	0.006413
23568	1.03312	84962	0.0	1.1152	0.331937
1629	1.13057	2988	0.0	0.0644	0.00987
15266	1.02848	57717	0.0	0.7028	0.210608
32	1.12121	34	0.0	0.0143	0.000138
4386	1.11759	9594	0.0	0.152	0.035009
6167	1.06607	14167	0.0	0.2109	0.04804
66431	1.01987	306499	0.0	3.4698	1.87959
146	1.11111	199	0.0	0.019	0.000637
721	1.24538	1191	0.0	0.0378	0.004715
102683	1.01772	512099	0.0	5.9302	4.55552
36	1.10811	38	0.0	0.0149	0.000148
110366	1.01926	521261	0.0	7.7864	5.41724
1267	1.22291	2087	0.0	0.057	0.008655
120819	1.01804	592235	0.0	6.9148	6.97687
43515	1.02161	196969	0.0	2.4068	0.891157
254	1.11599	320	0.0	0.0233	0.001087
49650	1.02036	230394	0.0	2.477	1.07154

Recursive: False

Satellite					
Tasks	avg***	Methods	Linearizable Methods % *	Grounding**	Preprocessing**
29	2.16667	37	53.125	0.01	0.000293
128	2.27027	260	37.5	0.0081	0.002181
10	2.33333	10	57.1429	0.0045	9.8e-05
17	2.55	21	50.0	0.0044	0.000223
32	2.18919	38	56.25	0.0053	0.000301
88	2.22388	135	46.9565	0.0068	0.001072
22	2.60714	29	48.0	0.0048	0.000327
58	2.1875	81	54.9296	0.0061	0.000598
55	2.24176	92	41.5584	0.0056	0.000872
66	2.64286	99	50.5747	0.0064	0.001052
76	2.7395	120	49.5413	0.0068	0.001529
55	2.17808	74	54.6875	0.006	0.000559
139	2.39801	202	53.2967	0.0091	0.001877
98	2.27273	155	42.4	0.0087	0.001523
40	2.55172	59	50.0	0.0052	0.000595
44	2.63492	64	50.0	0.0058	0.000685
24	2.14286	29	56.0	0.0047	0.000207
17	2.55	21	50.0	0.0048	0.000222
36	2.0	45	50.0	0.005	0.000288
63	2.19512	83	54.9296	0.0064	0.000621
66	2.20225	90	55.1282	0.0066	0.000683
34	2.2	46	52.5	0.0051	0.000346
131	2.75943	213	50.5102	0.0094	0.002685
31	2.55	41	51.4286	0.0059	0.000431

Recursive: False

Barman-BDI					
Tasks	avg***	Methods	Linearizable Methods % *	Grounding**	Preprocessing**
1419	2.46201	1146	27.7451	0.0561	0.019318
32612	2.52212	26972	22.9102	1.2605	3.24646
18408	2.51788	15184	23.4252	0.6573	1.16557
1125	2.45215	910	27.599	0.0446	0.017246
165	2.32283	128	36.6337	0.0182	0.001367
300	2.38723	236	34.3434	0.021	0.002747
10434	2.51241	8586	23.929	0.3766	0.422801
649	2.42747	518	30.4444	0.0346	0.006654
2597	2.4759	2117	25.7727	0.0814	0.042462
3060	2.49415	2480	26.899	0.0924	0.057508
393	2.40909	309	33.9695	0.0239	0.003794
3250	2.49112	2648	25.8966	0.0994	0.059478
1484	2.475	1201	27.5349	0.0505	0.019722
580	2.43573	460	31.9095	0.0298	0.006146
559	2.42568	445	31.1688	0.0287	0.005748
782	2.4464	626	29.8725	0.0364	0.008496
1871	2.48379	1512	27.6549	0.0641	0.027674
628	2.41833	503	29.7483	0.0333	0.006294
444	2.39601	352	32.0	0.0256	0.004284

Recursive: False

UM-Translog					
Tasks	avg***	Methods	Linearizable Methods % *	Grounding**	Preprocessing**
8	0.0	1	0.0	0.0188	0.000111
12	0.0	1	0.0	0.016	0.000217
11	7.5	3	0.0	0.016	0.000185
12	0.0	1	0.0	0.0161	0.00024
14	0.0	1	0.0	0.0162	0.000583
12	3.5	3	0.0	0.0221	0.00026
20	14.5	3	0.0	0.0221	0.001484
10	7.5	3	0.0	0.0161	0.000186
8	0.0	1	0.0	0.0165	0.000198
22	0.0	1	0.0	0.0183	0.001156
8	0.0	1	0.0	0.0174	0.000162
8	0.0	1	0.0	0.017	0.000167
14	0.0	1	0.0	0.0167	0.000537
10	0.0	1	0.0	0.0168	0.000198
14	7.5	3	0.0	0.0168	0.000551
24	17.5	3	0.0	0.0191	0.001745
10	0.0	1	0.0	0.0212	0.00023
10	0.0	1	0.0	0.0173	0.000195
8	0.0	1	0.0	0.0168	0.00017
21	0.0	1	0.0	0.0174	0.000967
8	0.0	1	0.0	0.0171	0.0002

Recursive: False

Monroe-Partially-Observable					
Tasks	avg***	Methods	Linearizable Methods % *	Grounding**	Preprocessing**
67418	2.21429	55016	68.1127	2.0101	1.9434
68890	2.21431	56051	68.0574	2.225	2.07487
67422	2.2142	55020	68.1127	2.039	2.01091
67433	2.2142	55029	67.8145	2.1308	2.00445
73827	2.19955	60547	68.0642	2.1147	1.8394
67430	2.21402	55028	68.1127	2.1293	2.03887
67429	2.21408	55027	68.1134	2.7032	2.05474
67812	2.21714	55315	67.682	2.1563	2.18095
67811	2.21707	55314	67.9555	2.107	2.13088
67428	2.21407	55026	68.1127	2.0435	1.89057
68884	2.21444	56045	68.0574	2.1099	1.95732
67425	2.21417	55023	68.1134	2.0865	1.90319
67418	2.21429	55016	68.1127	2.0806	1.98215
67801	2.21729	55304	67.9758	2.0694	2.11927
73821	2.19967	60541	68.0642	2.2321	2.20587
67425	2.21438	55021	68.1086	2.0549	1.84788
67428	2.21407	55026	68.1127	2.0925	1.85908
67422	2.2142	55020	68.1127	2.016	1.90154
67433	2.21407	55031	67.8186	2.0786	1.87328
67425	2.21438	55021	68.1086	2.0699	1.97273
67801	2.21729	55304	67.9758	2.1719	2.01427
67429	2.21415	55027	68.1127	2.1738	2.08523
67443	2.21377	55041	68.1134	2.3989	1.95702

Recursive: False

Transport					
Tasks	avg***	Methods	Linearizable Methods % *	Grounding**	Preprocessing**
8062	1.38812	9073	0.0	0.2104	1.34733
6386	1.37436	6826	0.0	0.1462	0.637496
3673	1.38752	4135	0.0	0.0918	0.100144
12969	1.40114	15377	0.0	0.3408	3.99726
13421	1.36337	13761	0.0	0.3683	9.91777
511	1.43529	511	0.0	0.016	0.004958
129	1.43077	131	0.0	0.0077	0.000732
402	1.36	401	0.0	0.0142	0.003512
381	1.375	401	0.0	0.0143	0.003101
116	1.20792	102	0.0	0.0093	0.032754
19	1.53333	16	0.0	0.0055	0.000315
45	1.3	41	0.0	0.0058	0.000963
4763	1.38814	5365	0.0	0.1162	0.285897
340	1.37288	355	0.0	0.0125	0.002433
14939	1.40585	18057	0.0	0.4223	6.58352
104	1.2551	99	0.0	0.0072	0.010046
2286	1.36952	2396	0.0	0.052	0.050062
175	1.42857	169	0.0	0.0084	0.001073
18581	1.36651	19291	0.0	0.5784	14.5498
397	1.43704	406	0.0	0.0128	0.00289
235	1.42982	229	0.0	0.0115	0.001958
1473	1.37047	1545	0.0	0.0348	0.023913
44	1.31707	42	0.0	0.0052	0.000912
1416	1.4306	1406	0.0	0.0323	0.027404
605	1.36967	634	0.0	0.0176	0.004384
1161	1.43952	1241	0.0	0.0273	0.011946
2649	1.3913	3037	0.0	0.0639	0.037584
204	1.42857	197	0.0	0.0085	0.001312
20001	1.36381	20561	0.0	0.5632	19.3826
363	1.38824	341	0.0	0.0132	0.008704
9038	1.38179	9920	0.0	0.2186	2.12052
115	1.42593	109	0.0	0.0068	0.000621
71	1.30769	66	0.0	0.0055	0.005033
81	1.23288	74	0.0	0.0067	0.005131
103	1.21111	91	0.0	0.0074	0.018213
144	1.42647	137	0.0	0.0074	0.000798
79	1.28986	70	0.0	0.0066	0.010016
68	1.24194	63	0.0	0.0068	0.002338
309	1.39205	353	0.0	0.0118	0.001922

Recursive: False

PCP					
Tasks	avg***	Methods	Linearizable Methods % *	Grounding**	Preprocessing**
13	3.66667	13	0.0	0.0051	0.000401
15	3.875	17	0.0	0.0051	0.00056
13	3.66667	13	0.0	0.0049	0.000446
15	3.16667	13	0.0	0.0052	0.00028
15	4.5	17	0.0	0.0057	0.000935
13	3.83333	13	0.0	0.0048	0.000576
13	3.83333	13	0.0	0.005	0.000466
13	3.66667	13	0.0	0.0049	0.000404
17	3.125	17	0.0	0.0052	0.000377
11	3.0	9	0.0	0.0044	0.000177
11	3.5	9	0.0	0.0044	0.000277
11	4.0	9	0.0	0.0046	0.000415
11	4.5	9	0.0	0.0101	0.000928
11	5.0	9	0.0	0.0045	0.00096
11	3.5	9	0.0	0.0045	0.000274
13	3.16667	13	0.0	0.0052	0.000282

Recursive: False

Monroe-Fully-Observable					
Tasks	avg***	Methods	Linearizable Methods % *	Grounding**	Preprocessing**
67418	2.21429	55016	68.1127	1.9754	1.88009
68890	2.21431	56051	68.0574	2.1289	1.98078
67422	2.2142	55020	68.1127	2.0137	2.16721
67435	2.21416	55031	67.8145	2.144	2.076
73827	2.19955	60547	68.0642	2.1169	1.96254
67436	2.21389	55034	67.8186	2.0457	1.70454
67433	2.21399	55031	68.1134	2.145	1.75014
67818	2.21701	55321	67.682	2.3581	2.08889
67811	2.21707	55314	67.9555	2.143	2.06021
67428	2.21407	55026	68.1127	2.1133	2.05435
68884	2.21444	56045	68.0574	2.1945	2.07282
67425	2.21417	55023	68.1134	2.1373	1.98499
67418	2.21429	55016	68.1127	2.5008	2.08575
67801	2.21729	55304	67.9758	2.4429	2.13274
73821	2.19967	60541	68.0642	3.5701	2.43004
67425	2.21438	55021	68.1086	2.252	2.13456
67430	2.21402	55028	67.8186	2.1134	1.9436
67422	2.2142	55020	68.1127	2.0775	1.89634
67439	2.21393	55037	67.8186	2.1591	2.0417
67425	2.21438	55021	68.1086	2.1023	1.99735
67801	2.21729	55304	67.9758	2.0668	1.9264
67429	2.21415	55027	68.1127	2.1173	1.91552
67451	2.2136	55049	68.1134	2.297	2.05399
67427	2.2142	55025	68.1127	2.2347	2.17898

Recursive: False

Zenotravel					
Tasks	avg***	Methods	Linearizable Methods % *	Grounding**	Preprocessing**
485	2.01786	449	48.8998	0.0262	0.002971
123	2.02913	104	44.2105	0.011	0.000698
97	2.05263	77	46.3768	0.0138	0.0006
281	2.09244	239	46.9484	0.017	0.00185

Recursive: False

SmartPhone					
Tasks	avg***	Methods	Linearizable Methods % *	Grounding**	Preprocessing**
10	1.5	3	0.0	0.0279	0.000249
74	1.18421	77	40.0	0.0279	0.003691
26	2.27273	12	0.0	0.0247	0.000326
281	1.05831	344	62.5	0.034	0.03811
22	1.125	25	50.0	0.0243	0.000192
23	1.375	9	0.0	0.025	0.001222

Recursive: False

Monroe					
Tasks	avg***	Methods	Linearizable Methods % *	Grounding**	Preprocessing**
10303	1.80762	13973	75.0408	0.9821	0.197555
13609	1.87757	14132	65.5856	0.8972	0.218261
13609	1.87757	14132	65.5856	0.9332	0.26452
17739	2.08939	27207	68.4257	1.2638	0.488081
14373	1.97155	23727	78.7107	1.0089	0.357765
12092	1.84136	12684	68.3509	0.9201	0.184313

Recursive: False

* Percentage of number of linearised methods / number of methods with 2 or more subtasks

** All time in seconds

*** Average number of tasks per method, excluding the initial task network

5.1.2 Solving Performance

Rover

Tasks	time to solve PO	time to solve TO	PO/TO time	Problem Name
396	203.617	0.168	121032.0	Rover/pfile12
460	187.619	4.077	4601.0	Rover/pfile10
65	0.236	0.057	410.0	Rover/pfile01
59	0.111	0.055	200.0	Rover/pfile02
1202	380.662	5.17	7363.0	Rover/pfile17
382	155.551	0.6	25923.0	Rover/pfile08
561	148.941	0.191	77783.0	Rover/pfile14
626	204.075	4.275	4773.0	Rover/pfile16
89	0.134	0.058	228.0	Rover/pfile03
595	202.559	0.403	50291.0	Rover/pfile15
494	179.874	24.747	726.0	Rover/pfile11
245	125.468	0.257	48861.0	Rover/pfile06
152	134.515	0.152	88698.0	Rover/pfile05
414	201.005	0.204	98302.0	Rover/pfile09
707	261.94	47.815	547.0	Rover/pfile13

Chapter 6

Concluding Remarks

If you wish, you may also name that section *"Conclusion and Future Work"*, though it might not be a perfect choice to have a section named "A & B" if it has subsections "A" and "B". Also note that you don't necessarily have to use these subsections; that also depends on how much content you have in each. (E.g., having a section header might be odd if it contains just three lines.)

6.1 Conclusion

This chapter usually summarizes the entire paper including the conclusions drawn, i.e., did the developed techniques work? Maybe add why or why not. Also note that every single scientific paper has such a section, so you can check out many examples, preferably at top-tier venues, e.g., by your supervisor(s).

6.2 Future Work

On top of that, you could discuss future work (and make clear why that is future work, i.e., by which observations did they get justified?).

Note that future work in scientific papers is often not mentioned at all or just in a very few sentences within the conclusion. That should not stop you from putting some effort in. This will (also) show the examiner(s)/supervisor(s) how well you understood the topic or how engaged you are.

Appendix A

Appendix: Explanation on Appendices

You may use appendices to provide additional information that is in principle relevant to your work, though you don't want *every reader* to look at the entire material, but only those interested.

There are many cases where an appendix may make sense. For example:

- You developed various variants of some algorithm, but you only describe one of them in the main body, since the different variants are not that different.
- You may have conducted an extensive empirical analysis, yet you don't want to provide *all* results. So you focus on the most relevant results in the main body of your work to get the message across. Yet you present the remaining and complete results here for the more interested reader.
- You developed a model of some sort. In your work, you explained an excerpt of the model. You also used mathematical syntax for this. Here, you can (if you wish) provide the actual model as you provided it in probably some textfile. Note that you don't have to do this, as artifacts can be submitted separately. Consult your supervisor in such a case.
- You could also provide a list of figures and/or list of tables in here (via the commands `\listoffigures` and `\listoftables`, respectively). Do this only if you think that this is beneficial for your work. If you want to include it, you can of course also provide it right after the table of contents. You might want to make this dependent on how many people you think are interested in this.

Appendix B

Appendix: Explanation on Page Borders

What you find here is an explanation of why the border width keeps flipping from left to right – which you might have spotted and wondered why that's the case.

Firstly, that is *intended* and thus correct, so there is no reason to worry about this. The reason is that this document is configured as a two-sided book, which means:

- We assume the document will be printed out,
- that this will be done in a two-sided mode (i.e., the document will be printed on both sides of each page), and
- that the bookbinding will be in the middle, just like in every book.

When you open the book, there are three borders of equal size n. This however requires that even pages have a border of n on their left and $\frac{n}{2}$ on their right, and odd pages have a border of $\frac{n}{2}$ on their left and n on their right. This is illustrated in Figure B.1.

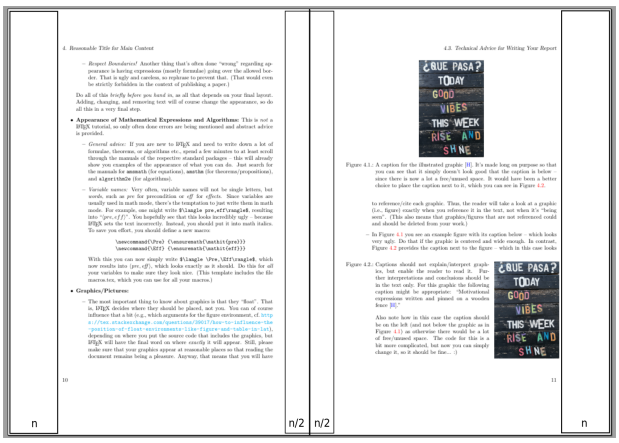

Figure B.1: Illustration showing why page borders flip.

Bibliography