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

March 2022

Declaration of Authorship:

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

Monday 14th March, 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

1 Introduction **1**
- 1.1 Introduction to HTN Planning . . . 1
- 1.2 Motivation . . . 1
- 1.3 Contributions . . . 2

2 Background **3**
- 2.1 Classical planning . . . 3
- 2.2 Partially Ordered Hierarchical Planning . . . 4
  - 2.2.1 HTN Solution . . . 4
- 2.3 Totally Ordered Hierarchical Planning . . . 4
- 2.4 Further Definitions . . . 4

3 Related Work **7**

4 Algorithm, Formalisation, Benchmark **9**
- 4.1 Algorithm . . . 9
- 4.2 Formalisation . . . 10
  - 4.2.1 Formally define possibilities for linearizing a model . . . 10
  - 4.2.2 Formally define solution preserving properties . . . 10
  - 4.2.3 This algorithm won't always work . . . 11
  - 4.2.4 Theory: So long as it doesn't have to cycle-break, algorithm will LITERALLY always work . . . 12
  - 4.2.5 Even when it does have to cycle break it will sometimes work . . . 13
  - 4.2.6 For some tasks it doesn't matter where they are executed . . . 14
- 4.3 Empirical Evaluation . . . 14
- 4.4 Modification . . . 15

5 Evaluation **17**

6 Concluding Remarks **19**
- 6.1 Conclusion . . . 19
- 6.2 Future Work . . . 19

A Appendix: Explanation on Appendices **21**

B Appendix: Explanation on Page Borders **23**

Bibliography **25**

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

Provide an 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.

1.3 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_0, TN_G) is over some domain D, has an initial state S_0, which is a total assignment to F, and has a initial compound task TN_Gk.

D = (F, T_P, T_C, δ, 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.

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.

This definition is trivially the same as having a initial task network. One can obtain the initial task network by applying a method to the initial task.

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.

Tail recursion

Lifted Methods apply to a certain set of typed objects.

Grounded A transition is ground if the parameters list only involves 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

Possibly another chapter focusing on better preconditions collecting?

4.1 Algorithm

Dr. Gregor Behnke already implemented a technique for 'this' available in the PANDA-3 planner. It was used to create many IPC TO domains, though it was never published, i.e., neither described nor properties like preserving of solutions were investigated.

1. Consider each method m independently, no interaction is considered.
2. For each compound task c in the methods set of subtasks infer its (super-relaxed) preconditions and effects.
3. Construct a graph with possible dependencies:
   - For each add effect a of c:
     a) move all tasks with precondition a behind c
     b) move all tasks with a delete effect in front of it
   - For each delete effect d of c:
     a) move all tasks with precondition d before c
     b) move all tasks with an add effect behind it
4. If this graph does not have a circle, choose this linearization. [It needs to be proved that (or whether) this sacrifices solutions!]

5. If this graph does have a circle, break the circle in a random place and choose this linearization

4.2 Formalisation

4.2.1 Formally define possibilities for linearizing a model

Input: method with n linearizations
Possible outputs:

1. output: n new methods, one per linearization.(Though there are usually n=k! many linearizations if k is the number of plan steps in a method)
2. output: m¡n methods (i.e., we delete some linearizations; the interesting question is which!)

4.2.2 Formally define solution preserving properties

* all solutions remain (note that even if 1 method with n linearizations get transformed into all n methods, this question is still undecidable) * at least one solution remains (again: undecidable) * all optimal solutions remain * at least one optimal solution remains The report/work should answer which of these criteria is guaranteed for the translation that's investigated

For the beginning it might be easiest how a primitive plan can linearized to a subset of all linearizations without losing any solutions. (For some tasks it will simply not matter where they are executed – this should be formalised. That should essentially be the POCL or PO-all criterion [note that they are not equivalent: the PO criterion will find more linearizations; see Bercher & Olz, AAAI-2020 POPOCL] by selecting just any sequence. So the problem will be which of them select so that we don't lose anything upon upwards-propagation.)

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.

4.2.3 This algorithm won't always work

This algorithm linearises all the methods to be totally ordered. Since sub-tasks inherit the orderings of their parents, it's obviously impossible to preserve a solution that requires the interleaving of sub-tasks of parents that are already ordered with respect to each other.

Consider the simple problem:

F = {a, b, c, g} $N_p = \{T_A, T_B, T_C, \text{G}\}$ $N_c = \{\text{AB }\}$ $\delta = \{(T_A, A), (T_B, B), (T_C, C)\}$
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. How do we make G executable? With the action B. How do we make B executable? With the action C. How do we make C executable? With the action A. How do we make A executable? It's executable in the initial state. Solution: 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. Thus PO -¿ TO not possible this way. You would need to

We can also see, if we apply Gregor's algorithm: AB = <pre a b, del a b, add c g> A = <pre a, del a, add c> C = <pre c, del c, add b> B = <pre b, del b, add g> G = <pre g, >

If we consider the method (Init $\implies$ {AB, C, G}) AB has precondition a b, and C has add effect b, so C is moved before AB. i.e. we add (C, AB) to $\prec$ AB has a delete effect del a b, which doesn't affect C or G, so nothing happens here. AB has add effect c g, and G has prec g so G is moved after AB. i.e. we add (AB, G) to $\prec$ C has a precondition c, which is in the add effect of AB. So AB is placed before C. i.e. we add (AB, C) to $\prec$

This produces a cycle.

The essence of this example is: some sub-task C would only NEED interleaving A, C, B from another parent before A and after B because of one or more of the following

1. its precondition needs the add effect of some sub-task A from another parent (that it can't get another way)
2. sub-task A has some precondition variable that is deleted by C (that can't be restored another way).

while simultaneously also needing one or more of the following:

1. some sub-task B from another parent has a precondition that C has in its add effect (that it can't get another way)
2. C has in its precondition a variable in the delete effect of some sub-task B from another parent (that can't be restored another way).

Due to the way Gregor's algorithm collects prec/effects for a compound task, interleaving requirements are a subset of the orderings found by:

- For each add effect a of c:
  1. move all tasks with precondition a behind c
  2. move all tasks with a delete effect in front of it
- For each delete effect d of c:
  1. move all tasks with precondition d before c
  2. move all tasks with an add effect behind it

So if the interleaving is necessary, gregors algorithm will find orderings (parent(A), C) and (C, parent(B)). Since parent(A) and parent(B) are the same compund task, a cycle is formed.

So requiring interleaving means that a cycle will form. A cycle does not always imply the solution is excluded though - due to the way the preconditons and effects are calculated, not all of the preconditions are always needed, and not all of the effects will actually occur.

4.2.4 Theory: So long as it doesn't have to cycle-break, algorithm will LITERALLY always work

A solution that requires interleaving under the current algorithm will be excluded from the set of solutions in the algorithm.

As we've said before, requiring interleaving means that a cycle will form. By simple contrapositive, if there are no cycles to break, that implies none of the the original solutions required interleaving sub-tasks. The methods are linearised in a way that is essentially equivalent to using POCl causal links. That means that if *every* method is linearized without using cycle-breaking, then the TO problem must preserve at least one solution if the PO problem has any?

(Furthermore if there are floating tasks after gregors algorithm is applied, you can produce multiple linearizations to preserve more solutions?)

1. if there is no cycle, then the subtasks of this method do not require their execution to be interleaved

2. If it can't be solved when no cycles were broken, then the original couldn't be solved either (needs proof, by induction (and contradiction)?)

1) Methods don't have any application preconditions except for what task it applies to. Why can't you just randomly decompose the initial task network into primitives and get a plan?

1. Cycles in decomposition process. This is clearly a problem that
2. **The proposed seq of primitives leads to a state where the next desired action is not executable.**

2) Assume that the original PO plan did have a solution. Let the solution take the form $a_0...a_n$. The preconditions, add, and delete effects are pre(a_i), add(a_i), del(a_i). This means that for every variable in a precondition of an action in the solution is in an add effect of another action OR present in the initial state.

Suppose we apply the methods that would lead to a solution in the PO version. Then we have one of the possible linearizations that could have resulted from the PO version. We now prove that this also leads to a solution?

Because of how the algorithm collects pre/eff for cmpd tasks, the subtasks each have a subset of the parents pre/eff. The union of all these prec/eff sets equal the compound tasks preconditions/effects.

Eventually, some of the sub-tasks decomposed to will be primitive tasks. These will still be a subset of the parents pre/eff, such that their union is equal to their parents pre/eff. Assuming that the method linearization for this method did not have cycles, that means that none of the pre/eff of these sub-tasks conflict. If they don't meet the pre/eff of the next action, neither could the PO version.

??? Therefore it must be a solution??

4.2.5 Even when it does have to cycle break it will sometimes work

(Again) A cycle does not always imply the solution is excluded though - due to the way the preconditons and effects are calculated, not all of the preconditions are always needed, and not all of the effects will actually occur. Thus the orderings imposed as a result of these preconditions and effects will sometimes not be necessary for a solution.

If you randomly break the cycle by removing an ordering like that (one from a unneeded precondition or incorrect effect) it will still work.

For another, the 'precondition needed by a compound task' may sometimes be possible to satisfy internally. The specific variable that is required may be added

by a sub-task of the parent. In that case, it may not need the ordering between itself and another compound task.

4.2.6 For some tasks it doesn't matter where they are executed

From content-notes: That should essentially be the POCL or PO-all criterion?? If the task has no causal links, it won't matter where they are executed.

A task can execute anywhere in any possible plan if:

- 1) The variables in its precondition are present in the initial state, and there are no actions that can delete it.
- 2) It only adds variables already present in the initial state, and there are no actions that can delete those variables
- 2b) It only adds variables that other actions do not rely on
- 3) It only deletes variables that don't hold in the initial state, and there are no actions that can add those variables
- 3b) It only deletes variables that other actions do not rely on.

Additionally, it can execute anywhere in a specific plan if:

- 1a) the variables in its precondition are present in the initial state, and no actions in the plan delete it.
- 2c) it adds variables that another action already adds before it does, and is not deleted again.

R.g. if gregor's algorithm deems it a floating task (no orderings imposed on it), it can definitely be executed anywhere.

4.3 Empirical Evaluation

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

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

Results, graphs, etc here

4.4 Modification

(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.

2. Find a more accurate set of preconditions and effects for a given task. E.g. if every possible decomposition of a given compound task that has <add a> in action will also have <del a> in a action after it, then there is no need to include <add a> as a possible effect of the compound task. E.g. another may be a precondition that is already met by an action before it. This precondition can be excluded from the compound tasks'??? Can it?

   Both will be difficult to find and remove in practice. But if we can remove some of these extraneous preconditions and/or effects, than extraneous orderings also disappear. This could reduce the number of cycles.

   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.

3. Because floating tasks can be executed where-ever, whenever there is such a floating task, we should produce

Chapter 5

Evaluation

Clearly not every project has empirical components (in particular in mathematics or theoretical topics) – though many do. So in case you were coding anything and conducted an empirical evaluation, this is where you should report the results.

As always, make adequate use of sections/subsections to structure this chapter. 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).

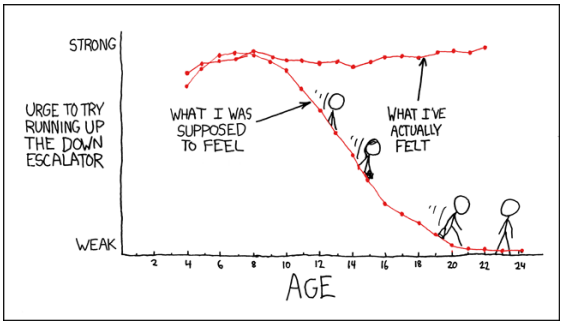

Figure 5.1: A graph illustrating the importance of axis and graph labeling. (Graphic taken from https://xkcd.com/252/.)

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.

B Appendix: Explanation on Page Borders

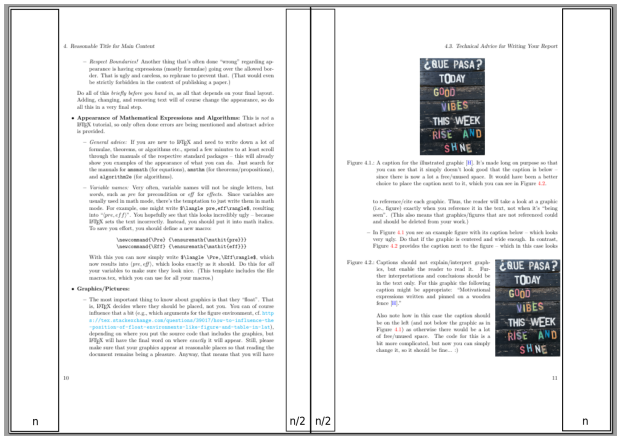

Figure B.1: Illustration showing why page borders flip.

Bibliography