APPLICATIONS OF AI PLANNING IN GENOME REARRANGEMENT AND IN MULTI-ROBOT SYSTEMS

Tansel Uras

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of the requirements for the degree of
Master of Science

Sabancı University

August, 2011

APPLICATIONS OF AI PLANNING IN GENOME REARRANGEMENT AND IN MULTI-ROBOT SYSTEMS

Approved by:

Assist. Prof. Dr. Esra Erdem .....
(Thesis Co-Supervisor)

Assist. Prof. Dr. Volkan Patoğlu .....
(Thesis Co-Supervisor)

Assist. Prof. Dr. Mehmet Serkan Apaydın .....

Assist. Prof. Dr. Malte Helmert .....

Prof. Dr. Kemal İnan .....

Date of Approval: 27/07/2011

YAPAY ZEKA İLE PLANLAMANIN GENOM DÜZENLEME VE ÇOKLU ROBOT SİSTEMLERİ ÜZERİNE UYGULAMALARI

Tansel Uras

Bilgisayar Bilimi ve Mühendisliği, Yüksek Lisans Tezi, 2011

Tez Danışmanları: Esra Erdem ve Volkan Patoğlu

Özet

Yapay zeka ile planlamada amaç, verilen bir ilk durumu bir hedef duruma ulaştırmak için, bir etmenin hareketlerini planlamaktır. Bu tezde, yapay zeka ile planlama iki tane zorlayıcı problemi çözmek için kullanıldı: işlemsel biolojiden genom düzenleme problemi ve çoklu-robot sistemlerinden ayrışık planlama problemi.

Genom düzenleme problemi, motivasyonunu evrimsel ağaçlardan alır. Amacı, iki canlının genomlarını karşılaştırarak aralarındaki en az sayıdaki genom düzenleme olayını (genomda oluşan büyük çaplı mutasyonlar) bulmaktır. Bu problemin tek kromozomlu dairesel genomlarla olan ve genomda bazı genlerin birden fazla kopyasının olmasına izin verilenini çözmek için adı GENOMEPLAN olan yeni bir method geliştirdik. Genom düzenleme problemini bir yapay zeka ile planlama problemi olarak formüle ettik ve yapay zeka planlayıcısı TLPLAN'ı kullanarak planlar bulduk. İşlemsel verimi arttırmak için GENOMEPLAN'de hareket tanımları içine bir kaç çeşit buluşsal yöntem yerleştirdik. Gerçek veriler üzerinde daha kesin cevaplar alabilmek için GENOMEPLANda düzenleme olaylarının ağırlıklarının ve önceliklerinin tanımlanmasına izin verdik. GENOMEPLAN'in uygulanabilirliğini gerçek veriler üzerinde gösterdik.

Çoklu robot sistemlerinde farklı çalışma alanlarında bulunan ve birden fazla robottan oluşan robot takımlarının, birbirleriyle robot alış verişinde bulunarak kendi hedeflerine en kısa zamanda ulaşmaya çalıstıklari bir problemi ele aldık. Bu problemi çözmek için takimlar arasında uzlaşmayı sağlayıp en kısa planı bulan bir algoritma öneriyoruz. Bu algoritmada, bir taraftan takımlar sadece kendi çalışma alanlarına ait planlar bulurken, diğer taraftan her takım merkezi bir sistemle iletişim kurup toplamda en kısa planı buluyorlar. Algoritmamızın doğru sonuç verdiğini ve olan bir sonucu kaçırmadığını ispatladık ve işlemsel karmaşıklığını analiz ettik. Metodumuzu akıllı bir fabrika örneği üzerinde gösterdik ve fabrikadaki bir çalışma alanını hareket anlatma dili $\mathcal{C}+$ ile modelleyip nedensel akıl yürütücü CCALC'ın çalışma alanı hakkında akıl yürütmesine dair örnekler sunduk.

APPLICATIONS OF AI PLANNING IN GENOME REARRANGEMENT AND IN MULTI-ROBOT SYSTEMS

Tansel Uras

Computer Science and Engineering, Master's Thesis, 2011

Thesis Supervisors: Esra Erdem and Volkan Patoğlu

Abstract

In AI planning the aim is to plan the actions of an agent to achieve the given goals from a given initial state. We use AI planning to solve two challenging problems: the genome rearrangement problem in computational biology and the decoupled planning problem in multi-robot systems.

Motivated by the reconstruction of phylogenies, the genome rearrangement problem seeks to find the minimum number of rearrangement events (i.e., genome-wide mutations) between two given genomes. We introduce a novel method (called GENOMEPLAN) to solve this problem for single chromosome circular genomes with unequal gene content and/or duplicate genes, by formulating the pairwise comparison of entire genomes as an AI planning problem and using the AI planner TLPlan to compute solutions. The idea is to plan genome rearrangement events to transform one genome to the other. To improve computational efficiency, GENOMEPLAN embeds several heuristics in the descriptions of these events. To better understand the evolutionary history of species and to find more plausible solutions, GENOMEPLAN allows assigning costs and priorities to rearrangement events. The applicability of GENOMEPLAN is shown by some experiments on real data sets as well as randomly generated instances.

In multi-robot systems, multiple teams of heterogeneous robots work in separate workspaces towards different goals. The teams are allowed to lend robots to one another. The goal is to find an overall plan of minimum length where each team completes its assigned task. We introduce an intelligent algorithm to solve this problem. The idea is, on the one hand, to allow each team to autonomously find its own plan and, on the other hand, to allow a central agent to communicate with the representatives of the teams to find an optimal decoupled plan. We prove the soundness and completeness of our decoupled planning algorithm, and analyze its computational complexity. We show the applicability of our approach on an intelligent factory scenario, using the action description language $\mathcal{C}+$ for representing the domain and the causal reasoner CCALC for reasoning about the domain.

Acknowledgements

I wish to express my gratitude to

- Esra Erdem and Volkan Patoğlu for their invaluable supervision,
- my thesis committee for their reviews and suggestions,
- all my friends from Sabancı University for their motivation and endless friendship,
- last, but not the least, my family for their unconditional love, support and persistent confidence in me.

Parts of this thesis are supported by Sabancı University Internal Research Fund.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis, we present two applications of AI Planning in two different areas, one in computational biology and the other in cognitive robotics. In an AI planning problem, we are given an initial state, a set of goals, a nonnegative integer k, and descriptions of actions; and the aim is to find a sequence of acions that lead the initial state to a goal state in at most k steps.

The first problem we consider is the genome rearrangement problem, a well studied problem in computational biology. In biology, phylogenies can be reconstructed by comparing the genomes of species. One metric of evolutionary distance for a comparison of two genomes is the number of rearrangement events (i.e., genome-wide mutations that change the order, orientations, presence of genes in a genome) that convert one genome to the other. Finding the minumum number of rearrangement events between two genomes is called the genome rearrangement problem.

Our work on this problem has started as an extension of the work of Erdem and Tillier [24]. We view the genome rearrangement problem as an AI planning problem as in [24], and use the AI planner TLPLAN [1] to solve it. Our contributions can be summarized as follows:

- Extending the earlier work of Erdem and Tillier [24], we introduce a computational method to solve the genome rearrangement problem for single chromosome circular genomes with duplicate genes and unequal gene content. The rearrangement events we consider are transpositions, inversions, transversion, insertions and deletions. Although the genomes of many species have unequal gene content and duplicate genes, most of the existing genome rearrangement software (e.g., GRIMM [57], GRAPPA [46],DERANGE 2 [7], MGR [9]) cannot handle unequal gene content and duplicate genes directly.

- To improve the computational efficiency, we embed three heuristics in the action descriptions: We ensure the number of breakpoints (pairs of genes that are adjacent

in the first genome but not in the second) decreases at each step of the plan. If a gene segment occurs in both genomes then the second heuristic identifies this gene segment as a "good segment" and maintains it through the search. The third heuristic identifies some genes as "irrelevant" if they form good segments with their two neighbor genes and the irrelevant genes are discarded. The first two heuristics combined, reduce the branching factor from $O(n^3)$ to $O(b^2)$, where n is the length of the genome and $b \leq n$ is the remaining number of breakpoints in the genome. The third heuristic does not effect the branching factor, but reduces the number of candidate rearrangement events and speeds up the search.

- We allow the users to express domain specific information to get more plausible results from the point of view of biology, by allowing the specification of costs and priorities of the actions that model the rearrangement events. A user may express "transpositions occur more often in these species" by assigning a lower cost and/or higher priority to transpositions.

- We develop the genome rearrangement software GENOMEPLAN, that incorporates the features described above, and conduct an extensive experimental evaluation using it:

  - We find phylogenies for species from three sets of real data.
  - We compare GENOMEPLAN with DERANGE 2 on randomly generated data.
  - We compare GENOMEPLAN with TD-ESTIMATOR (a rearrangement software by Lin et al. [41], that estimates the distance between two genomes with unequal gene content and duplicate genes), on randomly generated data.

The second problem we study is from cognitive robotics. Consider a domain of multiple teams of robots where each team has separate tasks and they work in separate workspaces. Each team consists of several different kinds of robots with different capabilities, and some robots may swap their end effectors to perform different actions. The teams are allowed to help each other by lending robots to each other, and the goal is to complete all the teams' tasks in minimum time. One straightforward way to solve this problem might seem to formalize the whole domain, and pose the problem above as a single planning problem; however, as the number of teams and robots grow, the domain description and the search space gets large too quickly.

We consider a restricted version of this problem, where robots are not allowed to lend or borrow more than one robot, and solve it with a decoupled planning algorithm as follows: we create representatives for each team, that are able to find minimum length plans for the teams they represent (plans that may involve lending or borrowing a robot). We also have a central agent that communicates with the representatives to find a minimum

length decoupled plan (possibly with robot exchanges) where each team is able to complete its task. Our work on this problem can be summarized as follows:

- We propose a novel algorithm for finding optimal decoupled plans for the types of problems described above. It operates by asking individual teams certain types of queries (such as "can you lend a robot before step k' and still be able to complete your task by step k?") until it can make robot lend/borrow arrangements between the teams and guarantee that the decoupled plan found is of minimum length. We provide the termination, soundness, completeness and complexity analysis of the proposed algorithm.

- We devise a domain, a Cognitive Painting Factory, where boxes are painted, waxed and stamped. Each team works in a separate workspace (painting the boxes different colors) and can lend robots to other teams. We model a workspace in the action description language $\mathcal{C}+$ [33] and use the causal reasoner CCALC [43] to reason about the model.

- We embed our optimal decoupled plan algorithm in an execution and monitoring framework and show its applicability on the Cognitive Painting Factory domain

We use two different planners, for the two different problems we study. We use TLPLAN for the genome rearrangement problem because it allows us to specify domain specific heuristics and gives us extensive control over the search. For the multi-robot systems, we use CCALC, because it handles concurrency effectively and allows us to ask expressive queries.

The rest of this thesis is organized as follows: In Chapter 2, we give preliminaries on AI planning. Our work on the genome rearrangement problem is summarized in Chapter 3 and our work on the decoupled planning algorithm is summarized in Chapter 4. We conclude by providing an overlook of contributions and future work on Chapter 5.

Chapter 2

AI Planning

In this chapter we give a brief background on AI planning. We give the definition of a planning problem (Section 2.1), describe how planning problems are represented (Section 2.2), and show how they are solved (Section 2.3). We conclude by giving two examples of modeling a planning problem in the input languages of TLPLANand CCALC(Section 2.4).

2.1 Planning Problem

In a planning problem, we are given an initial state, a set of goals, a nonnegative integer k, and descriptions of actions that an agent can execute. The aim is to find a plan—a sequence of actions that leads the initial state to a goal state—whose length is at most k. For instance, consider a number of blocks on a table such that each block is labeled by a letter. There is a robot who can move a block from one location to another location on the table. Suppose that initially the configuration of the blocks is as in Fig. 2.1(initial state). The goal is to obtain the configuration in Fig. 2.1(goal state) in at most 3 steps. The robot can achieve this goal by first moving Block r onto Block i, next moving Block a onto Block r, and then moving Block p onto Block a. This is a plan of length 3.

Classical planning is NP-hard for plans of polynomially-bounded length [11].

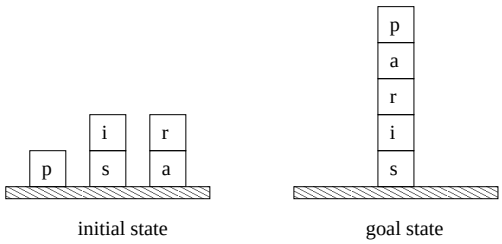

Figure 2.1: A planning problem

2.2 Action Languages

Action languages allow us to describe actions taht agents can perform in a dynamic world, and thus specify planning problems to planners.

- **Preconditions:** For an action to be applicable at a state, its preconditions must be met: each fluent in the preconditions has to be in the current state as well.
- **Add effects:** Executing this action at a state (where its preconditions are met), modifies the state by adding this set of fluents to the world.
- **Delete effects:** Executing this action at a state (where its preconditions are met), modifies the state by deleting this set of fluents from the world.

Most of the action languages have evolved from STRIPS: ADL [48] is another famous action language, which is an extension of STRIPS; ADL is discussed in detail, in Subsection 2.2.1. There are also more expressive languages, such as the Planning Domain Definition Language (PDDL) [44] which was first developed to make the 1998/2000 International Planning Competitions possible (IPC). It is evolving with each IPC and is currently at version 3.1. $\mathcal{C}+$is another action language based on causal reasoning; it is discussed in detail, in Subsection 2.2.2. The expressivity of all these languages are compared in table 2.1.

There are various action languages, such as STRIPS [28], ADL [48], the family of PDDL [44], and the family of action description languages [30], including $\mathcal{C}+$ [33]. A comparison of some of these languages is given in Table 2.1. In the following, we describe ADLand $\mathcal{C}+$, since these are the languages used in this thesis.

2.2.1 Action Description Language ADL

2.2.1.1 Describing Actions in ADL

We start with the two sets of atoms: *fluents* and *actions*. Fluents denote properties of the world that change over time so that a state of the world is described by a set of fluents. For instance, in the blocks world, a predicate of the form $on(block, loc)$ can be introduced to describe that block $block$ is on location loc; this predicate is a fluent since the locations of blocks may change over time. For instance, $on(B, Table)$ expresses that $Table$ is a location and Block B is right on top of it. Here $Table$ is a constant denoting the table and Block B is a constant denoting a specific block in the world. (From now on, we adapt the following notation: object constants start with an uppercase letter whereas object variables start with a lowercase letter.) Similarly another predicate of the form $clear(loc)$ can be introduced as a fluent to describe that the location loc is clear. With these fluents,

Table 2.1: Comparison of planning languages

	STRIPS	ADL	PDDL 1.0	PDDL 2.1	PDDL 2.2	PDDL 3.0	PDDL 3.1	$\mathcal{C}+$
strips style operators	✓	✓	✓	✓	✓	✓	✓	✓
type names in variable declarations	×	✓	✓	✓	✓	✓	✓	✓
negative preconditions	×	✓	✓	✓	✓	✓	✓	✓
disjunctive preconditions	×	✓	✓	✓	✓	✓	✓	✓
equality checks	×	✓	✓	✓	✓	✓	✓	✓
existential-preconditions	×	✓	✓	✓	✓	✓	✓	✓
universal-preconditions	×	✓	✓	✓	✓	✓	✓	✓
conditional effects	×	✓	✓	✓	✓	✓	✓	✓
axioms	×	×	✓	✓	✓	✓	✓	✓
numeric fluents	×	×	×	✓	✓	✓	✓	✓
plan quality measures: metrics	×	×	×	✓	✓	✓	✓	✓
durative actions	×	×	×	✓	✓	✓	✓	✓[a]
temporal preconditions and effects	×	×	×	✓	✓	✓	✓	✓
timed intitial literals	×	×	×	×	✓	✓	✓	✓
derived predicates	×	×	×	×	✓	✓	✓	✓
state trajectory constraints	×	×	×	×	×	✓	✓	✓
preferences	×	×	×	×	×	✓	✓	×
object fluents	×	×	×	×	×	×	✓	✓
(interleaved) concurrency	×	×	×	✓	✓	✓	✓	✓
ramifications	×	×	×	×	×	×	×	✓
qualifications	×	×	×	×	×	×	×	✓
defaults	×	×	×	×	×	×	×	✓
attributes of actions	×	×	×	×	×	×	×	✓
additive fluents	×	×	×	×	×	×	×	✓

[a] $\mathcal{C}+$supports durative actions as long as the duration is an integer

we can represent the state S where Block C is on the table and Block B is on top of Block C as follows:

$$\begin{aligned} S = \quad & \{on(B, C), on(C, Table), \\ & clear(B), clear(Table)\}. \end{aligned} \tag{2.1}$$

Action predicates, on the other hand, denote actions that an agent can execute. In the blocks world, a predicate of the form $move(block, loc, loc')$ can be introduced to denote the action of moving the block $block$ from a location loc onto location loc'. For instance, $move(B, C, Table)$ expresses the action of moving Block B from the top of Block C onto the $Table$. We describe actions in terms of its preconditions and effects.

In ADL, preconditions of actions are expressed by a set of formulas. For instance, let $X = move(block, loc, loc')$ where $block \neq loc$ and $loc \neq loc'$. One precondition of X

is that $block$ is at location loc, another is that loc' is clear. Also $block$ needs to be clear. Therefore, the precondition set $\mathcal{P}$ for X is

$$\mathcal{P} = \{on(block, loc), clear(loc'), clear(block)\}. \tag{2.2}$$

If the preconditions of X are satisfied at a state S, then X can be executed at S. For instance, $move(B, C, Table)$ can be executed at state (2.1).

Effects of an action are grouped into two categories: add effects and delete effects. These effects are represented by sets of fluents. They describe the changes to be added to and deleted from the state at which that action is executed. For instance, an add effect of $move(block, loc, loc')$ is $clear(loc)$: after a $block$ is moved from loc to loc' then loc gets clear. The add effects of $move(block, loc, loc')$ can be described as follows:

$$\mathcal{A} = \{clear(loc), on(block, loc')\}. \tag{2.3}$$

A delete effect of $move(block, loc, loc')$ is $on(block, loc)$: after $block$ is moved from loc to loc' then $block$ is not on loc anymore.

Some of the effects are conditional. For example, a delete effect of $move(block, loc, loc')$ is $clear(loc')$: after a $block$ is moved loc to loc', the location loc' is not clear anymore. This delete effect makes sense if $loc' \neq Table$, since $Table$ is always clear (otherwise, we would not be able to move a block onto the table). For that reason, we need to specify $clear(loc')$ as a delete effect if $loc' \neq Table$ holds. We can represent a conditional effect of an action by an expression of the form $F\ if\ G$ where F is a fluent and G is a set of formulas that do not involve actions. For instance, the delete effects of $move(block, loc, loc')$ can be described as follows:

$$\mathcal{D} = \{clear(loc')\ if\ \{to \neq Table\}, on(block, loc)\}. \tag{2.4}$$

With such an add set $\mathcal{A}$ and a delete set $\mathcal{D}$, we can define the effect of executing an action X at a state S (i.e., the state S' reached by executing X at S) as follows. First we "compile" a new add set $\mathcal{A}_S$ and a new delete set $\mathcal{D}_S$ from the add set $\mathcal{A}$ (2.3) and the delete set $\mathcal{D}$ (2.4) with respect to a state S, getting rid of conditionals:

$$\mathcal{A}_S = \{Y : Y \in \mathcal{A} \cap S\} \cup \{Y : Y\ if\ F \in \mathcal{A}, S \models F\},$$
$$\mathcal{D}_S = \{Y : Y \in \mathcal{D} \cap S\} \cup \{Y : Y\ if\ F \in \mathcal{D}, S \models F\}.$$

Then, we compute the next state S' as follows:

$$S' = (S \setminus \mathcal{D}_S) \cup \mathcal{A}_S.$$

For instance, the compiled effects of $move(B, C, Table)$ with respect to state S (2.1) are

$$\mathcal{A}_S = \{on(B, Table), clear(C)\},$$
$$\mathcal{D}_S = \{on(B, C)\}.$$

After executing $move(B, C, Table)$ at state S, we reach at a new state S' of the world obtained from S by deleting the elements of $\mathcal{D}_S$ and adding the elements of $\mathcal{A}_S$:

$$\begin{aligned} S' = \; & \{on(B, Table), on(C, Table), \\ & clear(B), clear(C), clear(Table)\}. \end{aligned}$$

2.2.1.2 Describing a Planning Problem

In a classical planning problem, we are given an initial state S_i and a goal G (or a goal state S_g), and want to find a sequence of actions that would lead the initial state S_i to a goal state S_g. For instance, in the Blocks World with three blocks $\{A, B, C\}$, a planning problem with the initial state S_i where A is on C and C is on B, and the goal state S_g where A is on B and B is on C can be described as follows:

$$\begin{aligned} S_i = \; & \{on(A, C), on(C, B), on(B, Table), \\ & clear(A), clear(Table)\}, \\ S_g = \; & \{on(A, B), on(B, C), on(C, Table), \\ & clear(A), clear(Table)\}. \end{aligned} \tag{2.5}$$

Note that, in this problem, the goal is to swap the positions of the blocks B and C.

2.2.2 Action Description Language $\mathcal{C}+$

Let us describe briefly the high-level representation formalism ($\mathcal{C}+$ [33]) where we describe action domains by "causal laws".

2.2.2.1 Syntax of Causal Laws

We start with a *(multi-valued propositional) signature* that consists of a set σ of *constants* of two sorts, along with a nonempty finite set $Dom(c)$ of *value names*, disjoint from σ, assigned to each constant c. An *atom* of σ is an expression of the form $c = v$ ("the value of c is v") where $c \in \sigma$ and $v \in Dom(c)$. A *formula* of σ is a propositional combination of atoms. If c is a Boolean constant, we use c (resp. $\neg c$) as shorthand for the atom $c = True$ (resp. $c = False$).

A signature consists of two sorts of constants: *fluent constants* and *action constants*. Intuitively, fluent constants denote "fluents" characterizing a state; action constants denote "actions" characterizing an event leading from one state to another.

A *fluent formula* is a formula such that all constants occurring in it are fluent constants. An *action formula* is a formula that contains at least one action constant and no fluent

constants.

An *action description* is a set of *causal laws* of three sorts. *Static laws* are of the form

$$\textbf{caused } F \textbf{ if } G \tag{2.6}$$

where F and G are fluent formulas. *Action dynamic laws* are of the form (2.6) where F is an action formula and G is a formula. *Fluent dynamic laws* are of the form

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H \tag{2.7}$$

where F and G are as above, and H is a fluent formula. In (2.6) and (2.7) the part **if** G can be dropped if G is True; the part F is called the *head*.

While describing action domains, we can use some abbreviations. For instance, we can describe the (conditional) direct effects of actions using expressions of the form

$$c \textbf{ causes } F \textbf{ if } G$$

which abbreviates the fluent dynamic law

$$\textbf{caused } F \textbf{ if } \mathit{True} \textbf{ after } c \wedge G.$$

This abbreviation expresses that "executing c at a state where G holds, causes F."

We can formalize that F is a precondition of executing c by the following expression

$$\textbf{nonexecutable } c \textbf{ if } \neg F$$

which stands for the fluent dynamic law

$$\textbf{caused } \mathit{False} \textbf{ if } \mathit{True} \textbf{ after } c \wedge \neg F.$$

Similarly, we can prevent the concurrent execution of two actions c and c' by the expression

$$\textbf{nonexecutable } c \wedge c'.$$

We can represent the "commonsense law of inertia" also by using abbreviations. For instance, we can describe that "the value of a fluent F remains to be true unless it is caused to be false" by the expression

$$\textbf{inertial } F$$

that stands for the fluent dynamic causal law

caused F **if** F **after** F.

2.2.2.2 Semantics for Action Descriptions

The meaning of an action description can be represented by a "transition system" [33]. Let D be an action description with a signature, with a set $\mathbf{F}$ of fluent constants and a set $\mathbf{A}$ of action constants. Then the transition system $\langle S, V, R\rangle$ *described* by D is defined with a set S of states, a value function V mapping every fluent constant P at each state s to a truth value, and a set R of transitions:

(i) S is the set of all interpretations s of $\mathbf{F}$ such that, for every static law (2.6) in D, s satisfies $G \supset F$;

(ii) $V(P, s)$ is the value of the fluent constant P in s;

(iii) R is the set of all triples $\langle s, A, s'\rangle$ such that s' is the unique interpretation of $\mathbf{F}$ which satisfies the heads F of all

- static laws (2.6) in D for which s' satisfies G, and
- fluent dynamic laws (2.7) in D for which s' satisfies G and $s \cup A$ satisfies H;

and A is the unique interpretation of $\mathbf{A}$ which satisfies the heads F of all action dynamic laws (2.7) in D for which $s \cup A$ satisfies G.

The laws included in *(iii)* above are those that are *applicable* to the transition from s to s' caused by executing A: the static causal laws make sure that s' is a state, and handles the ramifications and the qualifications of A; whereas the dynamic causal laws handle the preconditions and the direct effects of A, as well as other sorts of change.

A transition diagram can be thought of as a labeled directed graph. Every state s is represented by a vertex labeled with the function $P \mapsto V(P, s)$ from fluent constants to their values. Every triple $\langle s, A, s'\rangle \in R$ is represented by an edge leading from s to s' and labeled A.

2.2.2.3 Queries

Given an action domain description represented in a fragment of $\mathcal{C}+$ as described above, we can perform various reasoning tasks over it, such as planning, prediction, postdiction and diagnosis. Such reasoning problems are represented using queries in an "action query language" as described in [30]. We consider a variation of the action query language $\mathcal{Q}$ introduced in [30]. In this language, an *atomic query* is one of the two forms, F **holds at** t

or A **holds at** t, where F is a fluent formula, A is an action formula, and t is a time step. A *query* is a propositional combination of atomic queries.

Let D be an action description and $T(D) = \langle S, V, R \rangle$ denote the transition system described by D, with a set S of states, a value function V mapping, at each state s, every fluent P to a truth value, and a set R of transitions. A *history* of D of length n is a sequence

$$s_0, A_1, s_1, \ldots, s_{n-1}, A_n, s_n \tag{2.8}$$

where each $\langle s_i, A_{i+1}, s_{i+1} \rangle$ $(0 \leq i < n)$ is in R. We say that a query Q of the form F **holds at** t (resp. A **occurs at** t) is *satisfied* by a history (2.8) if s_t satisfies F (resp. if A_t satisfies A). For nonatomic queries, *satisfaction* is defined by truth tables of propositional logic. We say that a query Q is *satisfied* by an action description D, if there is a history H of D that satisfies Q.

Let us give now an example of the use of queries for planning. Suppose that F and G are fluent formulas denoting an initial state and goal conditions respectively. We can describe the problem of finding a plan of length n, with a query of the form

$$F \textbf{ holds at } 0 \ \wedge G \textbf{ holds at } n.$$

We can also solve variations of these problems, where some intermediate states are specified or where the specified actions are not executed consecutively. This allows us to enforce, for instance, further constraints in a planning problem.

2.3 Planners

There are various planners and reasoning systems to solve planning problems. Some of them are listed in Table2.2. In the following, we describe two planners: TLPLAN(which supports ADL) and CCALC(which supports $\mathcal{C}+$).

2.3.1 TLPLAN

The input language of TLPLAN supports ADL. In addition, it provides some predefined functions/predicates and allows us to define new functions/predicates, to be included in preconditions and effects of actions.

For instance, we can put a restriction on the length of a plan (e.g., its length is at most k) by adding the following expression

$$planlength < k$$

to the set of preconditions. Here $planlength$ is a predefined function that returns the length of a plan constructed so far.

Table 2.2: Comparison of planners

	LAMA	FF	FD-AUTOTUNE	C3	ROAMER	TLPLAN	CCALC
supports predicate representations	✓	✓	✓	✓	✓	✓	✓
supports object fluent representations	×	×	×	×	×	✓	✓
supports typed representations	✓	✓	✓	✓	✓	×	✓
supports untyped representations	✓	✓	✓	✓	✓	✓	✓
supports schematic representations	✓	✓	✓	✓	✓	✓	✓
supports grounded representations	✓	✓	✓	✓	✓	✓	✓
supports negative conditions	✓	✓	✓	✓	✓	✓	✓
supports first-order formulas	✓	✓	✓	×	✓	✓	✓
supports conditional effects	✓	✓	✓	×	✓	✓	✓
supports universal effects	✓	✓	✓	✓	✓	✓	✓
supports derived predicates	✓	×	✓	×	✓	✓	✓
supports recursive effects	×	×	×	×	×	✓	✓
supports external predicates	×	×	×	×	×	×	✓

In Blocks World, we can ensure that a good tower built from bottom up is not disassembled, by including $\neg goodTower(loc)$ to the precondition set $\mathcal{P}$ of $move(block, loc, loc')$. Here $goodTower$ is a predicate that can be defined by the user as a first-order logic formula [1]. Such predicates are called "derived predicates", i.e., predicates not directly added or deleted by actions but derived from fluents at every state.

Given a domain description (i.e., action descriptions) and a planning problem (i.e., initial state and goal), TLPLAN computes a plan if one exists. For instance, with the domain description of the blocks world presented in the previous section, TLPLAN computes the following plan for the planning problem (2.5):

$$\langle move(A, C, Table), move(C, B, Table),$$
$$move(B, Table, C), move(A, Table, B)\rangle.$$

This plan consists of four actions: moving A from the top of C onto the table, moving C from the top of B onto the table, moving B from the table to the top of C, and moving A from the table to the top of B; the length of this plan is 4.

To find a plan of length k, TLPLAN performs a forward search in the state space: starting from the initial state, it explores the states that are reachable from the initial state by a sequence of k actions, until a goal state is found or some failure condition is reached (i.e., all the states reachable by a sequence of k actions are explored and the goal state is not reached).

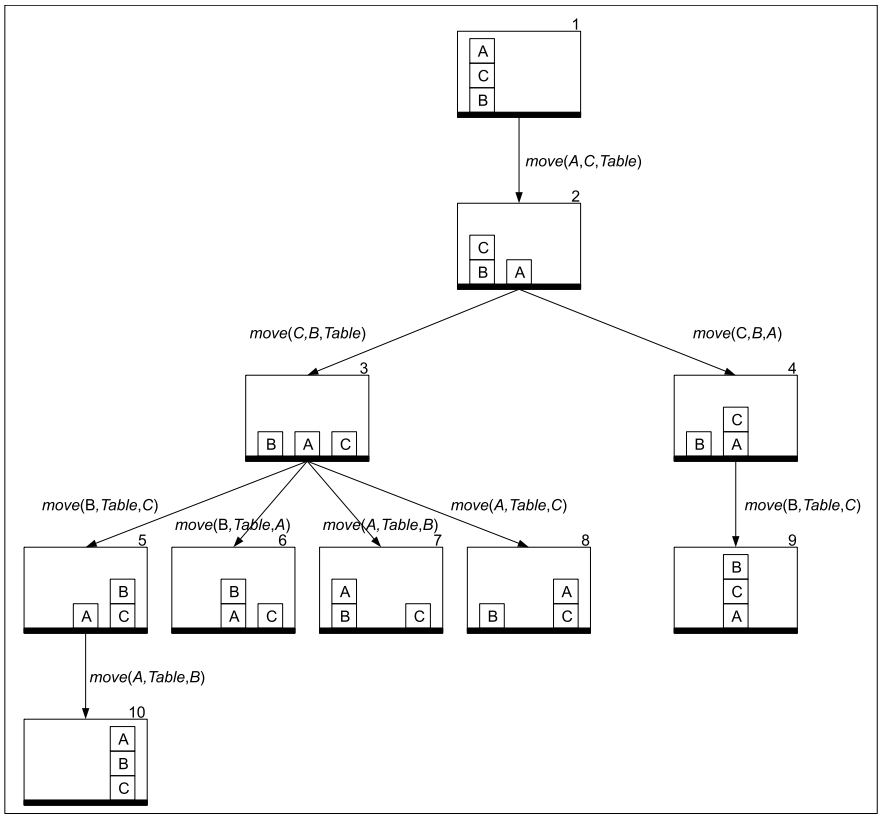

Figure 2.2: A sample search tree. Nodes represent states and edges represent actions.

There are various forward search strategies that affect the order of the explored states, such as domain-independent depth-first and breadth-first strategies or heuristic search strategies such as best-first, and A* [49]. TLPLAN allows us to specify a search strategy as part of input. Consider, for instance, the sample search tree (shown in Fig. 2.2) for the planning problem (2.5). Here, the specified search strategy determines the order of the nodes explored during search. With the depth-first search strategy, the order is $1, 2, 3, 5, 10$; with the breadth-first search strategy, the order is $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$. Some of these strategies are optimal (i.e., they guarantee finding the minimum cost/length solution in the search tree). For instance, if we are trying to minimize the plan length, breadth-first search is optimal. If we are trying to minimize the total cost of a plan, uniform-cost search is optimal [49].

The reason we chose to use TLPLAN over other planners is mainly its expressivity and its features that allows extensive control over the search, which provide the means to experiment with different search strategies and heuristics. Table 2.2 compares the features of TLPLAN to top performing planners in IPC'08 and IPC'10. Note that, most of the competition planners are meant to be used on problems without any prior domain specific information, and then use domain independent heuristics for improved search speed. TLPLAN, on the other hand, does not have built-in heuristics, but allows us to express search control information, tailored towards specific domains.

2.3.2 CCALC

The Causal Calculator (CCALC) [43] is a reasoning system, that performs reasoning tasks over an action domain description represented in a fragment of $\mathcal{C}+$ described above. To present formulas to CCALC, conjunctions $\wedge$, disjunctions $\vee$, implications $\supset$, negations $\neg$ are replaced with the symbols `&`, `++`, `->>`, and `-` respectively. Once an action domain description is given, we can perform various reasoning tasks via queries in an action query language, like the variation of the action query language $\mathcal{Q}$ described above. For instance, we can present a query to CCALC as follows:

```
:- query
maxstep :: 0..infinity;
0: loc(b) = c;
maxstep: loc(b) = a.
```

A query of the form F **holds at** t (resp. A **occurs at** t) is presented to CCALC as `t: F` (resp. `t: A`). The third line in the query above describes the initial state at time step 0, and the last line describes the goal condition at time step `maxstep`. With this query, CCALC successively tries to find a plan of length `maxstep=0,1,...,infinity`.

Given a domain description and a query, CCALC checks whether the query is satisfied by the domain description (in the sense of satisfiability planning of [37]) as follows: 1) it transforms the causal laws into a propositional theory Γ_D, via "causal logic" [33]; 2) it transforms the query into a propositional theory Γ_P; 3) it checks whether $\Gamma_D \cup \Gamma_P$ is satisfiable; 4) if $\Gamma_D \cup \Gamma_P$ is unsatisfiable, it returns No; 5) otherwise, it returns Yes and presents an example extracted from a satisfying interpretation for $\Gamma_D \cup \Gamma_P$. The transformations in the first two steps are different: the one in 1) is based on literal completion [43], whereas the one in 2) is based on a simpler procedure (see the work of Giunchiglia et al. [33] for a detailed description). Such a difference allows one to check the satisfiability of other queries (for instance, for replanning) without executing the first step again. Step 3) is done automatically by a state-of-the-art SAT solver, such as MINISAT [19] or its parallel variant MANYSAT [34]. Inheriting the advantages of $\mathcal{C}+$, CCALC allows reasoning about nondeterminism, concurrency, ramifications, and provides useful utilities, such as external predicates and additive fluents.

2.4 Example: Blocks World

In this section, we give two examples of modeling the blocks world problem as a planning problem, one in the input language of TLPLAN, using ADL operators; the other in the input language of CCALC using causal laws.

2.4.1 Solving Blocks World with TLPLAN

A basic domain model of blocks world in TLPLAN can be found in Fig. 2.3. We have two fluents (as part of the described symobls in TLPLAN): $on(block, loc)$ and $clear(loc)$, both of which were explained earlier. There is also another predicate, $block(x)$, that denotes x is a block, however we do not consider it as a fluent since its value does not change over time.

We have a single action, $move(block, to - loc)$, this time omitting the location we take the block from since it is not actually a parameter of the problem (there can be only one location a block can be on at a given moment). Given a block $block$, we can ask TLPLAN to find us a value for loc that would make $on(block, loc)$ true, by using the $exists$ built-in method.

A simple problem for the described domain model can be found in Fig. 2.4.

2.4.2 Solving Blocks World with CCALC

A basic domain model of blocks world in CCALC can be found in Fig. 2.5.

The first line describes that there are two sorts of objects, $location$s and $block$s, where each $block$ is also a $location$. Even though it describes $table$ as a $location$, it does not mention the blocks, which is specified in the problem description since they are problem specific.

It then declares the constants:

- $loc(block)$ is a functional fluent (i.e. $loc(block) = value$ has the same meaning as $on(block, value)$).
- $above(block, location$ is kind of like a derived predicate, even though it is not directly modified by the actions, its value can change as an indirect effect of actions.
- $move(block)$ is the only action that moves a block
- $destination(block)$ is an attribute of $move(block)$ (i.e., it specifies where to move the block)

One important thing to note in the rest of the formulation is that it allows concurrency (i.e. multiple $move$ actions can take place in a single turn). Using the $noconcurrency.$ command would disable it.

Fig. 2.6 shows a problem file for this formulation. The query asks if there is a plan of length at most 3 (and at least 2), that can transform the initial state to the goal state.

Figure 2.3: Domain description of blocks world in TLPLAN's input language

```
(declare-described-symbols
    (predicate block 1)
    (predicate on 2)
    (predicate clear 1)
)

(declare-defined-symbols
    (predicate move-effects 2)
)

;operator for moving a block
(def-adl-operator (move ?block ?to-loc)
    (pre
        (?block) (block ?block)
        (?to-loc) (clear ?to-loc)
        (and
            ;the block should be clear
            (clear ?block)

            ;the block should not be moved on itself
            (not (= ?block ?to-loc))

            ;the block should be moved somewhere else
            (exists (?from-loc)(on ?block ?from-loc)
                (not (= ?to-loc ?from-loc)))))

    ;resolve the effects of the action
    (move-effects ?block ?to-loc))

;effects of moving a block
(def-defined-predicate (move-effects ?block ?loc)
    (and
        ;the block is no longer on its previous loc
        (exists (?prev-loc)(on ?block ?prev-loc)
            (del (on ?block ?prev-loc)))

        ;it is now on its new location
        (add (on ?block ?loc))

        ;the new location is no longer clear
        (implies (not (= ?loc Table)) (del (clear ?loc)))

        ;the old location is now clear
        (exists (?prev-loc)(on ?block ?prev-loc)
            (add (clear ?prev-loc)))))
```

Figure 2.4: Problem description of blocks world in TLPLAN's input language

```
;
;         Initial state:              Goal:
;
;                                     3   6
;           1   3   5                 2   5
;           2   4   6                 1   4
;         -------------             ---------

(set-initial-world

    (block 1)(block 2)(on 1 2)(on 2 Table)(clear 1)
(block 3)(block 4)(on 3 4)(on 4 Table)(clear 3)
(block 5)(block 6)(on 5 6)(on 6 Table)(clear 5)
    (clear Table)
)

(set-goal
    (on 3 2) (on 2 1) (on 1 Table)
    (on 6 5) (on 5 3) (on 4 Table)
)
```

Figure 2.5: Domain description of blocks world in CCALC

```
% File 'bw': The blocks world

:- sorts
  location >> block.

:- objects
  table                          :: location.

:- constants
  loc(block)                     :: inertialFluent(location);
  above(block,location)          :: sdFluent;
  move(block)                    :: exogenousAction;
  destination(block)             :: attribute(location) of move(block).

:- variables
  B,B1                           :: block;
  L                              :: location.

% two blocks can't be on the same block at the same time
constraint loc(B)=loc(B1) ->> loc(B)=table where B @< B1.

% definition of above

caused above(B,L) if loc(B)=L.
caused above(B,L) if loc(B)=B1 & above(B1,L).
default -above(B,L).

% a block cannot be above itself
constraint -above(B,B).

% effect of moving a block
move(B) causes loc(B)=L if destination(B)=L.

% a block can be moved only when it is clear
nonexecutable move(B) if loc(B1)=B.

% a block can't be moved onto a block that is being moved also
nonexecutable move(B) & move(B1) if destination(B)=B1.

% a block can't be moved to its current position
nonexecutable move(B) if destination(B)=loc(B).
```

Figure 2.6: Problem description of blocks world in CCALC

```
% File 'bw-test': Planning problem in the blocks world
%
%        Initial state:             Goal:
%
%                                   3   6
%          1   3   5                2   5
%          2   4   6                1   4
%        -------------            ---------

:- include bw.

:- objects
  1..6       :: block.

:- show loc(L).

:- query
  maxstep :: 2..3;
  0:       loc(1)=2, loc(2)=table, loc(3)=4,
           loc(4)=table, loc(5)=6, loc(6)=table;
  maxstep: loc(1)=table, loc(2)=1, loc(3)=2,
           loc(4)=table, loc(5)=4, loc(6)=5.
```

Chapter 3

AI Planning for Genome Rearrangement

In biology, phylogenies can be reconstructed by comparing the genomes of species [51]. One metric of evolutionary distance for a comparison of two genomes is the number of rearrangement events (i.e., genome-wide mutations that change the order, orientations, presence of genes in a genome) that converts one genome to the other; a smaller number of such events implies a closer lineage. Finding the minimum number of rearrangement events between genomes is called the genome rearrangement problem.

The study of genome rearrangement problems has started with a focus on reversal distance, pioneered by David Sankoff [50, 38]. Since then various efficient polynomial time algorithms have been developed [35, 6, 36, 45, 3] for finding the exact reversal distance. However, no such method is known for finding the exact transposition distance.

Bafna and Pevzner [2] provide a polynomial time approximation algorithm and conjecture that the problem of finding the exact transposition distance is NP-hard. Blanchette et al. [7] introduce a greedy search algorithm with lookahead to find near-optimal solutions to this problem. More recent studies, on the other hand, are based on the idea of simulating genome rearrangement events by means of Double-Cut-Join (DCJ) operations introduced by Yancopoulos et al. [61]. Basically, a DCJ operation makes two cuts on a genome, and then rejoins the four cut points in a possible way. Even though a DCJ operation is not an evolutionary event encountered in nature, it can simulate such events: Any inversion, fusion (joining two chromosomes) or fission (breaking a chromosome into two) can be simulated by a single DCJ operation; and any transposition can be simulated by two successive DCJ operations (namely a fission to cut the desired part out, followed by a fusion to reinsert it to the desired location). If the genomes are of equal gene content and do not contain duplicate genes, then the minimum DCJ distance between two genomes can be found in linear time [61]. Bergeron et al. [4, 5] extend this result for genomes with possibly multiple chromosomes (both linear and circular), but still requiring equal gene content and no gene duplications. Some studies [20, 60] extend the canonical approach that only considers permutations, by allowing gene duplications or unequal gene

content; however, efficient computation of the edit distance is not achieved. A method that estimates the edit distance with gene duplications and unequal gene content is proposed [55]; however, only inversions are considered. Lin et al. [41] describe a method that estimates the minimum DCJ distance by using duplication and gene loss operations along with DCJ operations; however, the relative frequencies of these three types of operations are required for better accuracy, and identifying these frequencies for real data is problematic.

We view the genome rearrangement problem as an AI planning problem as in our earlier work [24, 59]. In a planning problem, the goal is to find a plan (i.e., a sequence of actions that leads an agent from an initial state to a goal state) whose length is at most a given nonnegative integer k. The idea is then to describe genome rearrangement events as actions, and consider one of the genomes as the initial state and the other one as the goal state; and prompt an AI planner to find a sequence of at most k actions (rearrangement events) that leads the initial state to the goal state.

We introduce a computational method to solve the genome rearrangement problem for single chromosome circular genomes with duplicate genes and unequal gene content. We consider transpositions, inversions, transversions, insertions and deletions as rearrangement events [59, 58]. We formulate the genome rearrangement problem as a planning problem differently and develop a genome rearrangement software, called GENOMEPLAN, based on these methods. We show the applicability and effectiveness of ur methods using GENOMEPLAN, with real datasets and randomly generated datasets.

Our formulation of the genome rearrangement problem as a planning problem differs from that of Erdem and Tillier in the following ways: First of all, it extends the representation of genomes to handle duplicate genes and the descriptions of events (transpositions, inversions, transversions). It introduces new operators for insertions and deletions. The goal-check is done in a more computationally-efficient way by means of checking the breakpoint distance (instead of checking the whole gene orders of the genomes). Also, some heuristics (e.g., the breakpoint heuristic, discarding irrelevant genes) are embedded in the action descriptions; Erdem and Tillier specify the breakpoint heuristic as a search control strategy (separate from the action descriptions) and do not discard irrelevant labels. We allow insertion/deletion of a single gene, whereas Lin et al. [41] allow deletion/duplication of a block of genes.

In the following, we introduce a precise definition of a genome rearrangement problem and a planning problem (Section 3.1), and explain how we model genome rearrangement as planning (Section 3.2). We discuss the results of our experiments (Section 3.3) and conclude (Section 3.4).

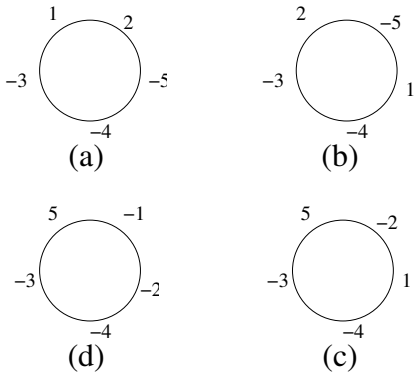

Figure 3.1: (a) A genome; (b) a transposition of (a); (c) an inversion of (b); (d) a transversion of (c).

3.1 Genome Rearrangement Problem

We describe the genome rearrangement problem as an AI planning problem, and then use the AI planner TLPLAN to compute solutions. Before we explain our planning-based approach to genome rearrangement, let us briefly go over some preliminaries via examples.

We represent a circular genome of a single-chromosome organism by circular configurations of numbers $1, \ldots, n$, with a sign $+$ or $-$ assigned to each of them. For instance, Fig. 3.1(a) shows a genome for $n = 5$. Numbers $\pm 1, \ldots, \pm n$ will be called *labels*. Intuitively, a label corresponds to a gene, and its sign corresponds to the orientation of the gene. By $(l_1, \ldots, l_n)$ we denote the genome formed by the labels $l_1, \ldots, l_n$ ordered clockwise. For instance, each of the expressions $(1, 2, -5, -4, -3), (2, -5, -4, -3, 1), \ldots$ denotes the genome in Fig. 3.1(a).

About genomes g, g', we say that g' is a *transposition* of g (or *can be obtained from g by a transposition*) if, for some labels $l_1, \ldots, l_n$ and numbers k, m $(0 < k, m \leq n)$,

$$g = (l_1, \ldots, l_n),$$
$$g' = (l_k, \ldots, l_m, l_1, \ldots, l_{k-1}, l_{m+1}, \ldots, l_n).$$

Here $l_k, ..., l_m$ is moved after l_n. For instance, the genome in Fig. 3.1(b) is a transposition of the genome in Fig. 3.1(a). Given two genomes g and g', the problem of finding the smallest number of successive transpositions by which g' can be obtained from g is conjectured to be NP-hard [2].

Similarly, about genomes g, g', we say that g' can be obtained from g by a *deletion* (or g can be obtained from g' by an *insertion*) if, for some labels $l_1, \ldots, l_n$ and a number m $(0 < m \leq n)$,

$$g = (l_1, \ldots, l_n),$$
$$g' = (l_1, \ldots, l_{m-1}, l_{m+1}, \ldots, l_n).$$

Other events, inversions and transversions, can be defined as in Erdem and Tillier's work [24]. About genomes g, g', we say that g' is an *inversion* of g (or *can be obtained from g by an inversion*) if, for some labels $l_1, \ldots, l_n$ and a number m $(0 < m \leq n)$,

$$g = (l_1, \ldots, l_n),$$
$$g' = (-l_m, \ldots, -l_1, l_{m+1}, \ldots, l_n).$$

For instance, the genome in Fig. 3.1(c) is an inversion of the genome in Fig. 3.1(b). Given two genomes g and g', the problem of finding the smallest number of successive inversions by which g' can be obtained from g is in P [35].

About genomes g, g', we say that g' is a *transversion* (or *inverted transposition*) of g (or *can be obtained from g by a transversion*) if, for some labels $l_1, \ldots, l_n$ and numbers k, m $(0 < k, m \leq n)$,

$$g = (l_1, \ldots, l_n),$$
$$g' = (-l_m, \ldots, -l_k, l_1, \ldots, l_{k-1}, l_{m+1}, \ldots, l_n).$$

Here $l_k, ..., l_m$ is inverted and then moved after l_n. For instance, the genome in Fig. 3.1(d) is a transversion of the genome in Fig. 3.1(c).

We say that there is a *breakpoint* between two genomes if one of the genomes includes the pair l, l' and the other genome includes neither the pair l, l' nor the pair $-l', -l$. For instance, there are 3 breakpoints between $(1, 2, 3, 4, 5)$ and $(1, 2, -5, -4, 3)$. The number of breakpoints between two genomes is called their *breakpoint distance*.

The *genome rearrangement problem* can be defined as follows: given two genomes g and g', and a positive integer k, decide whether g' can be obtained from g by at most k successive events.

3.2 Methods

We view the genome rearrangement problem as a planning problem as follows:

> given two genomes g and g', and a nonnegative integer k, find a sequence of at most k events that reduces the number of breakpoints between g and g' to 0.

3.2.1 Describing Genomes

We view the gene order of a whole genome as a signed permutation of gene labels, and represent the permutation by specifying the clockwise order of the labels. For that, we introduce a fluent of the form $cw(l, l')$ which expresses that label l' comes after label l in clockwise direction. However, such a representation alone is not sufficient to describe

genomes with duplicate genes. For example, the genome $(1, 2, 3, 2, 4, 2)$ can be represented as follows:

$$cw(1, 2), cw(2, 3), cw(3, 2), cw(2, 4), cw(4, 2), cw(2, 1).$$

Here, $(1, 2, 4, 2)$ can be erroneously considered as a subsequence of the genome. For this reason, we treat duplicate genes as different genes but also keep track of them. To identify which genes are duplicates, we introduce a predicate $dup(x, y)$ ("the gene labeled as x is originally labeled as y"). For example, the duplication in the genome $(1, 2, 3, 2, 4, 2)$ can be represented by relabeling two of the 2's as a 5 and a 6 and by specifying that the genes labeled as 5 and 6 are originally labeled as 2:

$$cw(1, 2), cw(2, 3), cw(3, 5), cw(5, 4), cw(4, 6), cw(6, 1),$$
$$dup(2, 2), dup(5, 2), dup(6, 2).$$

In general, we represent multiple copies $l_1, \ldots, l_c$ of the same gene m by the set of fluents $dup(l_i, m)$. In this way, we can keep the number of dup predicates linear in the size of duplicate genes (as opposed to quadratic number of dup predicates when every pair of duplicates is specified).

3.2.2 Genome Rearrangement as a Planning Problem

In the planning problem that describes a genome rearrangement problem, both genomes g and g' are specified in the initial state S_i. We assume that the rearrangement events are applied to the genome g. We describe the gene order of g by fluents of the form $cw(l, l')$, and the gene order of g' by fluents of the form $cw'(l, l')$. The effects of actions not only change the order of genes in g, but also the number of breakpoints between g and g'. Therefore, we represent the number of breakpoints between g and g' by a functional fluent $bpcount$. Suppose that we are given two genomes, $g = (1, 2, 3, 2, 4)$ and $g' = (1, 4, -3, -2)$. In the corresponding planning problem, the initial state S_i is defined as follows:

$$S_i = \{cw(1, 2), cw(2, 3), cw(3, 5), cw(5, 4), cw(4, 1),$$
$$cw'(1, 4), cw'(4, -3), cw'(-3, -2), cw'(-2, 1),$$
$$dup(2, 2), dup(2, 5), bpcount = 4\}.$$

The goal S_g is defined as $S_g = \{bpcount = 0\}$. Note that any state $Z \supset S_g$ is considered a goal state.

Given the planning problem described by S_i and S_g above and the action descriptions discussed in the following subsections, TLPLAN computes a 2-step plan, $\langle transvert(2, 3, 4), delete(5)\rangle$, according to which the genome $(1, 2, 3, 2, 4)$ can be transformed to $(1, 4, -3, -2)$ as follows: first $2, 3$ is inverted and then inserted after 4, next the first appearance of 2 is deleted.

3.2.3 Describing Rearrangement Events

We introduce five actions to describe transpositions, inversions, transversions, insertions and deletions, and represent them in the input language of TLPLAN.

Consider the action $transpose(x, y, z)$ ("the gene sequence starting with the gene labeled as x and ending at the gene labeled as y is inserted after the gene labeled as z") that describes a transposition. The preconditions of this action are described with the set:

$$\begin{aligned}\mathcal{P} = \{&label(x), label(y), label(z), x \neq z, y \neq z,\\ &planlength < k, \neg cw(z, x), notbetween(z, x, y)\}\end{aligned} \tag{3.1}$$

The first three conditions describe that x, y and z are labels to denote genes. The fourth and the fifth conditions ensure that the segment $x..y$ of the genome is not inserted after x or y. The fifth condition is that the length of the plan constructed so far is less than the given nonnegative integer k. The sixth condition describes that x does not come after z. The last condition describes that z is not between x and y. Here $notbetween$ is a derived predicate defined as a first-order logic formula. We can represent $notbetween(z, x, y)$ in first-order logic as follows:

$$notbetween(z, x, y) \equiv (length(y, z) \leq length(x, z))$$

Here, $length(x, y)$ is a function evaluating to the length of the sequence $x \ldots y$. [1]

In a state S that satisfies the preconditions (3.1) and where the genome is of the form $(x_1, x..y, y_1..z, z_1, ...)$, the effects of a transposition (i.e., insertion of $x..y$ after z) are described by an add set $\mathcal{A}$ and a delete set $\mathcal{D}$:

$$\begin{aligned}\mathcal{A} &= \{cw(x_1, y_1), cw(z, x), cw(y, z_1)\},\\ \mathcal{D} &= \{cw(x_1, x), cw(y, y_1), cw(z, z_1)\}.\end{aligned} \tag{3.2}$$

According to these sets, we discard the clockwise orderings $x_1\ x$, $y\ y_1$ and $z\ z_1$, and instead consider the following clockwise orderings $x_1\ y_1$, $z\ x$ and $y\ z_1$. Thus, after the execution of action $transpose(x, y, z)$, we obtain the genome $(x_1, y_1..z, x..y, z_1, ...)$.

Similarly, other genome rearrangement events are defined as ADL-operators in the input language of TLPLAN.

[1] We keep the positions of genes in a genome (as if the genome is represented as a vector) using a function of the form $pos(X) = P$ ("the position of gene X is P"). The positions of genes are generated for the initial genome and updated with each rearrangement event. Then, the length of the sequence $x \ldots y$ ($length(x, y)$) is computed by finding the difference between the positions of x and y, taking into account that the genome is circular.

3.2.4 Swapping Duplicates

Simply renaming the duplicate genes as described in Section 3.2.1 can lead to different lengths of plans. Consider, for instance, two genomes $(1, 2, 3, 2, 4)$ and $(1, 2, 3, 4, 2)$. Renaming one of the 2's in each genome as 5 leads to four possible problems. One of these problems is defined by $(1, 5, 3, 2, 4)$ and $(1, 2, 3, 4, 5)$, and the other by $(1, 2, 3, 5, 4)$ and $(1, 2, 3, 4, 5)$. The former problem can be solved in two steps, by transposing 5 after 4 and transposing 3 after 2; the latter, on the other hand, can be solved in a single step, by transposing 5 after 4.

To be able to find shorter plans, we introduce an auxiliary action, $swap(x, y)$, for swapping two genes that are duplicates of each other. We assign a cost of 0 to $swap(x, y)$ so that a series of $swap$ operations can allow switching between any two different relabelings of the same genome with a total cost of 0. The idea is then to assign a cost of 1 to each rearrangement event, and to try to find a plan whose total cost is as small as possible.

3.2.5 Embedding Heuristics in Action Descriptions

For a more efficient computation, we embed some heuristics in action descriptions to reduce the search space.

3.2.5.1 The Breakpoint Heuristic

We enforce the breakpoint heuristic in genome rearrangements, i.e., we ensure that the number of breakpoints decreases at the each step of the plan, by modifying the effects of actions. For instance, the effects of a transposition characterized by the action $transpose(x, y, z)$ are modified to take into account the change in the number of breakpoints, by including the following equality in the add set:

$$bpcount = bpcount - relievedbp\_transpose(x, y, z)$$

where $relievedbp\_transpose$ calculates the number of breakpoints that are eliminated by the transposition. Recall that, with $relievedbp\_transpose$, breakpoints are not counted from the scratch at each step: they are counted initially, and after that the number of breakpoints is decreased by each application of a transposition.

Similarly, the breakpoint heuristic is embedded in the description of each genome rearrangement event as well as the auxiliary action of swapping duplicates' labels.

Embedding the breakpoint heuristic in the description of $swap$ is quite important, which brings along also some complications. Although a series of swaps of duplicates' labels allows switching between any two different relabelings of the same genome where each gene is uniquely labeled, note that there are too many possible relabelings that may

lead to too many swaps (not to mention redundant swaps). Therefore, forcing a swap operation to relieve at least one breakpoint reduces the number of these possibilities (and thus the size of the search space) and eliminates the risk of getting stuck in an infinite loop of 0-cost $swap$ actions in the search. On the other hand, embedding the breakpoint heuristic in swap operations may prevent switching from one relabeling to another one that would lead to a plan with a smaller cost. Consider, for instance, the rearrangement of the genome $(1, 5, 6, 4, 2, 3, 7)$ into $(1, 2, 3, 4, 5, 6, 7)$, and assume that 5 is a duplicate of 2 and 6 is a duplicate of 3. In this example, $swap(2, 5)$ alone does not modify the number of breakpoints (two existing breakpoints are relieved but two new ones emerge), nor does $swap(3, 6)$; but together, they can relieve 4 breakpoints. To take into account such cases, we introduce two additional auxiliary actions, namely $block\_swap$ and $reverse\_block\_swap$ (also subject to the breakpoint heuristic). The auxiliary action $block\_swap$ takes two non-overlapping gene segments of equal length k such that, for each $1 \leq i \leq k$, the i'th gene of the first segment is a duplicate of the i'th gene of the second segment; after that, for each $1 \leq i \leq k$, it swaps the i'th gene in the first segment with the i'th gene in the second segment. The auxiliary action $reverse\_block\_swap$ is similar, except it swaps the genes in reverse order (i.e., swaps the i'th gene of the first segment with the $k - i + 1$'th gene of the second segment, for each i). Although these two auxiliary actions guide the search by picking relabelings of genomes, they do not guarantee that these relabelings lead to shorter plans. Consider, for instance, rearranging $(1, 4, 2, 3, 5)$ into $(1, 2, 3, 4, 5)$ where 2, 3 and 4 are duplicates of each other. Clearly, these genomes are the same; however, there are no $swap$, $block\_swap$ or $reverse\_block\_swap$ operation applicable to reduce the number of breakpoints.

Let us now discuss how the breakpoint heuristic affects the computational efficiency and the optimality of plans. Note that the branching factor of the search tree without the breakpoint heuristics is $O(n^3)$ where n is the genome length (since a transposition is specified by 3 genes). The introduction of the breakpoint heuristic reduces the branching factor to $O(n^2)$ (since the number of transpositions that break at least one breakpoint is $O(n^2)$). Such a decrease in the branching factor speeds up the search. On the other hand, the breakpoint heuristic possibly does not preserve optimality with an optimal-cost search strategy.

3.2.5.2 Maintaining the Good Segments

We call a segment of a genome a *good segments* if none of the adjacent pairs in the segment have a breakpoint between them. We maintain the good segments in a genome while rearranging (i.e. we never create a new breakpoint by seperating a pair of genes that are also adjacent in the goal). For that, we extend the preconditions of actions with further conditions.

For example, we extend the preconditions (3.1) of transpositions by including the

following predicates:

$$\begin{aligned} &\neg goodbefore(x), \neg goodafter(y), \neg goodafter(z), \\ &goodlink(z, x) \vee goodlink(y, z_1) \vee goodlink(x1, y1). \end{aligned} \tag{3.3}$$

Here, $goodbefore(x)$, $goodafter(y)$, $goodafter(z)$ and $goodlink(z, x)$ are all derived predicates. The predicate $goodbefore(x)$ expresses that the gene that comes before x in the genome g described by cw fluents is at its goal position with respect to g'. In other words, the sequence $x_1\, x$ that occurs in g occurs also in the other genome g' described by cw' fluents; in such a case, we say that x_1 and x form a *good link*. Similarly, $goodafter(y)$ expresses that the gene that comes after gene y in g is in its goal position with respect to g', whereas $goodlink(z, x)$ expresses that the sequence $z\, x$ in g is in its goal position.

By including (3.3) in $\mathcal{P}$, we ensure that a transposition $transpose(x, y, z)$ is applicable to a genome g if the genes before x, after y, and after z in g are not in goal positions relative to genome g', but at least one of these genes will be in its goal position after $transpose(x, y, z)$. By this way, including (3.3) in $\mathcal{P}$ enforces a transposition to relieve at least one breakpoint by forming at least one good link.

Note also that maintaining good segments (in addition to the breakpoint heuristic) reduces the branching factor further (from $O(n^2)$) to $O(b^2)$ where b is the number of breakpoints.

3.2.5.3 Discarding Irrelevant Labels

We consider a label (or a gene) *irrelevant* if it forms two good links as expected, i.e., it is in its goal position. To improve the computational efficiency, we remove the irrelevant labels at each step of the search by modifying the delete effects of actions.

For instance, consider $transpose(x, y, z)$ that rearranges the genome $(x_1, x..y, y_1..z, z_1, ...)$ into $(x_1, y_1..z, x..y, z_1, ...)$. After this transposition, if $x_1\, y_1$ forms a good link and y_1 forms a good link with the label that comes right after y_1, then y_1 becomes an irrelevant label and can be discarded during the search. This heuristic is expressed by adding the following to the delete effects of $transpose(x, y, z)$:

$$label(y_1)\ if\ \{goodlink(x_1, y_1), goodafter(y_1)\}.$$

A similar conditional effect is included for the other case, when $z\, x$ forms a good link and makes z irrelevant.

Discarding irrelevant labels reduces the number of candidate rearrangement events at each step of the search, and thus provides a good speed-up. However, it does not reduce the branching factor further since irrelevant labels are genes with good links on both sides.

3.2.6 Assigning Costs and Priorities to Events

To get more plausible results from the point of view of biology, one can incorporate domain-specific information into action descriptions by assigning costs and priorities to actions to guide the search. For instance, transpositions may occur more often in some species, then we can assign a lower cost to transpositions to reflect this domain-specific information. In this way, with a cost-based search strategy, we expect to obtain a plan that involves transpositions instead of more expensive actions. Priorities of actions affect the ordering of successor states in search: actions with higher priorities are selected before the ones with lower priorities if the search strategy is priority-based.

Costs and priorities can be defined as specific numbers or as terms that evaluate to numbers. For instance, we can define the cost of a transposition $transpose(x, y, z)$ by a function:

$$cost\_transpose(x, y, z) = ct + originalLength(x, y)$$

where ct is a number and $originalLength(x, y)$ is a derived function that returns the length of the original gene segment (before preprocessing) denoted by $x..y$. If we want TLPLAN to take into account the costs of transpositions as defined above, we include the following expression in the preconditions:

$$cost = cost\_transpose(x, y, z).$$

If we want TLPLAN to prefer transpositions that relieve more number of breakpoints per cost, we can define the priority of transpositions accordingly:

$$priority\_transpose(x, y, z) = \\ relievedbp\_transpose(x, y, z)/cost\_transpose(x, y, z)$$

and include the following expression in the preconditions:

$$priority = priority\_transpose(x, y, z).$$

Alternatively, we can define priorities of events with respect to an expected occurrence of events specified in terms of percentages. For instance, the priority of an inversion can be defined with respect to a specified percentage (say, 80%):

$$priority\_invert(x, y) = \\ epi + (relievedbp\_invert(x, y, z)/cost\_invert(x, y, z))$$

where epi is set to 100 (resp. 0) at 80% (resp. 20%) of the steps. In this way, if the plan consists of 10 steps, the priority of an inversion is set to a higher value (and thus the search is biased towards an inversion) at 8 steps.

3.3 Results

Based on the planning approach described above, we implemented a software system, called GENOMEPLAN, that can solve genome rearrangement problems based on the gene-order data of whole genomes, using the planner TLPLAN. GENOMEPLAN can handle genomes with unequal gene contents or duplicate genes, and it considers transpositions, inversions, transversions, insertions and deletions as rearrangement events. Also, it can solve variations of genome rearrangement where we specify costs/priorities of events by functions.

We performed three sorts of experiments:

- **Experiments with Real Data.** To show the usefulness of our planning-based approach to genome rearrangement, we experimented with three sets of real data using GENOMEPLAN: mitochondrial genomes of *Metazoa* (animals with a nervous system, and muscles) [8], chloroplast genomes of *Campanulaceae* (flowering plants) [13], and chloroplast genomes of various land plants and green algae [15]. Only in the first data set, genomes are of unequal content and with duplicate genes.

- GENOMEPLAN **vs.** DERANGE 2**.** To compare our planning-based approach to genome rearrangement with duplicate genes, with the naive approach (where we relabel duplicates uniquely and use an existing genome rearrangement software system that can handle transpositions, inversions and transversions and then whose goal is to find a parsimonious solution), we experimented with a set of randomly generated problem instances.

- GENOMEPLAN **vs.** TD-ESTIMATOR**.** Another available genome rearrangement software that can handle inversions, transpositions, and transversions is TD-ESTIMATOR of Lin et al. [41]. Unlike GENOMEPLAN and DERANGE 2, TD-ESTIMATOR tries to approximate the true distance between two genomes in terms of DCJ operations, gene losses and duplications. We experimented with a set of randomly generated problem instances with duplicates and equal gene content.

Before solving the problem instances, we applied two preprocessing methods to reduce their sizes: "safe" deletions and "condensing". According to the former method, if a gene is present in one of the genomes only, then we delete all its copies from that genome. After that, according to the latter method (like in GRAPPA), common subsequences in the genomes are replaced by some new identifiers.

All experiments are run on a workstation with two 1.60GHz Intel Xeon E5310 Quad-Core Processor and 16 GB RAM, running Centos 64bit (Version 5.3). All the benchmark data are available at `http://krr.sabanciuniv.edu/projects/GenomePlan`.

3.3.1 Experiments with Real Data

In these experiments, for each data set, first we computed the distance (i.e., the number of rearrangement events) between each pair of genomes using GENOMEPLAN. Then, based on these distances, we constructed a phylogeny using the program NEIGHBOR [27] with default values. We studied these phylogenies in comparison with the published phylogenies, to analyze the accuracy of our approach on the real data.

In all the experiments using GENOMEPLAN, the planner TLPLAN was run with the default search strategy, namely depth-first-priority. The cost of insertions and deletions were assigned to 1; and the cost of swaps were assigned to 0 as described in Section 3.2.4. For the inversions and transpositions, we tried five different cost assignments: 1 and 1, 1 and 1.5, 1 and 2, 1.5 and 1, 2 and 1, respectively. We assumed that the cost of a transversion is identical to the cost of a transposition. Recall that the goal is to find a plan with a small total cost (rather than a shortest plan). We set the maximum total cost k of a plan to a large value, to see the effectiveness of our methods with heuristics.

The priorities of inversions, transpositions and transversions were defined as the number of relieved breakpoints per cost, as described in Section 3.2.6. Swaps have 0-cost and they can help forming good links with no cost, as described in Section 3.2.4; therefore, to make swaps applicable whenever possible, the priority of swaps was set to a high value (2000 plus the number of relieved breakpoints). Consider, for instance, rearranging $(1, 4, 2, 3, 5)$ into $(1, 2, 3, 4)$ where 4 and 5 denote the same gene: if swaps have a higher priority, then the rearrangement can be achieved by the swap of 4 and 5 followed by the deletion 5; otherwise, the rearrangement can be achieved with a solution of higher cost where the deletion of 5 is followed by a transposition of 4 after 3. The priorities of insertions and deletions were set to high values as well with the following intuition: insertions/deletions are predestined to occur a fixed number of times in a small-cost plan and thus are unavoidable, and applying them as early as possible helps reduce the problem size.

We also experimented with these data sets where the priority is defined relative to a specified percentage of inversions. For each data set, inversion percentages of 25, 50 and 75 were tried, where the costs of all events were set to 1.

3.3.1.1 Chloroplast genomes of land plants and green algae

We considered the chloroplast genomes of 7 species studied by Cui et al. [15]. These genomes share 85 genes; and each genome is of length 87–97. When the rearrangement events were assigned the same cost of 1, GENOMEPLAN computed a plan for each pair of genomes. Overall, 21 plans, each with 8–48 events, were computed in 67 minutes. The number of rearrangement events included in these plans are summarized in Table 3.1. Based on these results, an unrooted tree (Fig. 3.2) was constructed using the distance

Table 3.1: The distance matrix computed by GENOMEPLAN for the chloroplast genomes of 7 land plants and green algae: *Nicotiana (NI)*, *Marchantia (MA)*, *Chaetosphaeridium (CM)*, *Chlorella (CA)*, *Chlamydomonas (CS)*, *Nephroselmis (NE)*, and *Mesostigma (ME)*.

	NI	*MA*	*CM*	*ME*	*NE*	*CS*	*CA*
NI	0						
MA	8	0					
CM	13	10	0				
ME	25	19	21	0			
NE	30	26	26	25	0		
CS	48	43	43	44	45	0	
CA	40	35	35	32	34	48	0

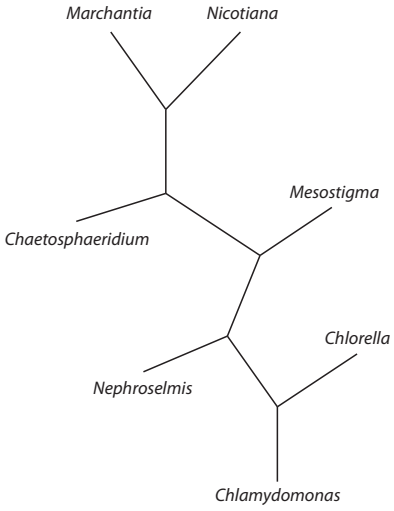

Figure 3.2: The tree computed by NEIGHBOR with the matrix in Table 3.1.

matrix program NEIGHBOR.

The unrooted tree shown in Fig. 3.2 groups *Nicotiana* and *Marchantia* with *Chaetosphaeridium*, thus grouping the land plants and charophyte algae; it also groups *Chlorella* and *Chlamydomonas* with *Nephroselmis*, thus grouping the chlorophyte algae; *Mesostigma* is an outlier. These results conform with the biological evidence based on the analysis of 50 concatenated proteins [15].

In all other experimental settings (with different costs of events and the percentages of inversions), exactly the same unrooted tree was computed. Since the distances between genomes are large, the modifications of costs/priorities do not change the outcome.

In the work by Cui et al. [15], the authors handle duplications in a different way, considering inversions only. If a gene has multiple copies, instead of renaming the duplicate genes, they choose to keep only one of these genes and discard the rest. In this way, they construct multiple datasets with equal gene content and with no duplicates. Then, for each

Table 3.2: The distance matrix computed by GENOMEPLAN for 13 chloroplast genomes of *Campanulaceae*: *Wahlenbergia (WA)*, *Merciera (ME)*, *Trachelium (TM)*, *Symphyandra (SY)*, *Campanula (CA)*, *Adenophora (AD)*, *Legousia (LE)*, *Asyneuma (AS)*, *Triodanus (TS)*, *Codonopsis (CO)*, *Cyananthus (CY)*, *Platycodon (PL)*, *Tobacco (TO)*.

	TO	*PL*	*CY*	*CO*	*ME*	*WA*	*TS*	*AS*	*LE*	*SY*	*AD*	*CA*	*TM*
TO	0												
PL	8	0											
CY	6	9	0										
CO	6	10	5	0									
ME	9	14	9	9	0								
WA	8	12	9	9	4	0							
TS	8	11	9	9	8	6	0						
AS	8	11	9	9	9	7	2	0					
LE	9	12	9	10	8	6	2	4	0				
SY	7	11	8	8	6	3	5	6	5	0			
AD	8	11	8	9	6	4	6	7	6	3	0		
CA	7	10	8	8	5	3	5	6	5	1	2	0	
TM	7	10	9	8	5	2	4	5	4	1	2	1	0

dataset, a phylogeny is computed using breakpoint medians. The dataset that yields the best tree is chosen for a full evaluation by GRAPPA. With this method, the computation of the phylogeny above took almost 25 days in the study of Cui et al. [15].

3.3.1.2 Chloroplast genomes of *Campanulaceae*

We considered the chloroplast genomes of 13 *Campanulaceae* species, each with 105 genes, as in the work by Cosner et al. [13]. In the case of rearrangement events having the same cost, all 78 plans (each with 1–14 events) were computed by GENOMEPLAN in 46 CPU seconds. These results are summarized in a distance matrix (Table 3.2).

According to the unrooted tree constructed by NEIGHBOR (Fig. 3.3) over this distance matrix, we observe the following: *Wahlenbergia* and *Merciera* are grouped together; *Trachelium*, *Symphyandra*, *Campanula*, *Adenophora* are grouped together; *Legousia*, *Asyneuma*, *Triodanus* are grouped together; *Codonopsis*, *Cyananthus*, *Platycodon*, *Tobacco* are grouped together, separate from the others. These groupings are identical to the ones in the consensus tree presented in Fig. 4 of [13]. The major division between the grouping of *Codonopsis*, *Cyananthus* and *Platycodon*, and the others conform with the most recent results [14] based on the sequence analysis; also this division corresponds to the distribution of pollen morphology characteristics, unlike the previous results.

The experiments with other cost/priority settings resulted in the same major groupings, except for the case where the cost of inversions were set to 2 and the cost of transpositions and transversions was set to 1. In this case, *Merciera* was not grouped with

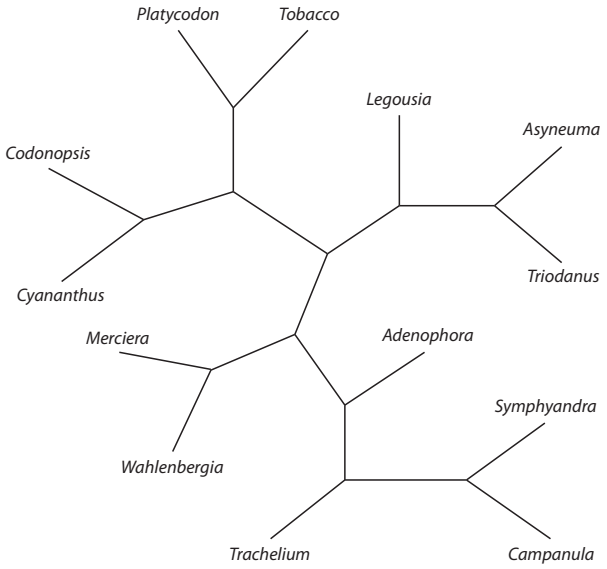

Figure 3.3: The tree computed by NEIGHBOR with the matrix in Table 3.2.

Wahlenbergia but closer to the group of *Codonopsis*, *Cyananthus*, *Platycodon*, *Tobacco*. However, penalizing inversions with a higher cost is contradictory to the general belief that inversions occur more often in chloroplast genomes and this belief might explain the discrepancy observed between the phylogenies.

3.3.1.3 Mitochondrial genomes of *Metazoa*

We considered the mitochondrial genomes of 11 species, each with 36 genes, studied by Blanchette et al. [8].

When the events were assigned the same cost of 1 and the desired percentage of inversions was set to 50%, GENOMEPLAN computed all 55 plans (each with 1–22 events) in 622 CPU seconds; these results are summarized in Table 3.3. The phylogeny constructed by NEIGHBOR for these distances is shown in Fig. 3.4. This phylogeny groups chordates (Human) and echinoderms (*Asterina pectinifera*, *Paracentrotus lividus*) together; arthropods (*Drosophila yakuba*, *Artemia franciscana*), some molluscs (*Katharina tunicata*) and annelids (*Lumbricus terrestris*) together; nematodes (*Ascaris suum*, *Onchocerca volvulus*) are a sister to these two groupings. These results conform with the results of [47] based on morphological data. Groupings of chordates and echinoderms, and molluscs and annelids also conform with the most widely accepted view of Metazoan Systematics and Tree of Life, based on the analysis of molecular data (18S rRNA sequences). On the other hand, this phylogeny does not group all molluscs together.

In the other settings (with different costs and priorities of events), different phylogenies were obtained; however, these phylogenies do not confirm much with the widely accepted trees mentioned above. For instance, when the priority was defined as the number

Table 3.3: The distance matrix computed by GENOMEPLAN for 11 mitochondrial genomes of *Metazoa*: Human (HU), *Asterina pectinifera (AP)*, *Paracentrotus lividus (PL)*, *Drosophila yakuba (DY)*, *Artemia franciscana (AF)*, *Albinaria coerulea (AC)*, *Cepaea nemoralis (CN)*, *Katharina tunicata (KT)*, *Lumbricus terrestris (LT)*, *Ascaris suum (AS)*, *Onchocerca volvulus (OV)*.

	OV	*AS*	*LT*	*KT*	*CN*	*AC*	*AF*	*DY*	*PL*	*AP*	HU
OV	0										
AS	10	0									
LT	18	19	0								
KT	17	17	11	0							
CN	18	17	16	17	0						
AC	18	17	18	16	3	0					
AF	19	18	15	11	16	16	0				
DY	19	17	15	11	17	17	2	0			
PL	17	17	15	16	16	16	16	16	0		
AP	17	18	16	16	17	17	16	17	1	0	
HU	18	18	16	14	17	18	12	11	14	13	0

of breakpoints per cost, chordates were located closer to arthropods than to echinoderms in the phylogeny. When the costs of inversions were set to higher values or when the desired percentage of inversions was reduced to 25%, nematodes were grouped with molluscs and echinoderms were placed as a sister to this grouping. We observe that higher costs of inversions or lower percentage of inversions lead to less plausible groupings; these results suggest a bias towards inversions. We also observe the variety of phylogenies obtained in different settings. This variety can be attributed to the small variations between pairwise distances: A small change in the distance matrix results in comparatively large changes in the associated phylogeny computed by NEIGHBOR. These experiments show also that GENOMEPLAN, with its flexibility of setting costs/priorities, can be useful for better understanding which events might have occurred more often.

3.3.2 GENOMEPLAN vs. DERANGE 2

For these experiments, we randomly generated genome rearrangement problem instances. For a given genome length n, and a number d of duplicate genes, a single random problem instance was generated as follows: First, we generated the identity permutation $1, \ldots, n-d$, and, for d times, we added to the end of the permutation a random number from $\{1, \ldots, n-d\}$. After shuffling the resulting sequence, we obtained a random genome g of length n, which contains d duplicate genes. Given the number of each event, we generated the other genome g' by applying a series of randomly generated instances of these events on g.

We generated two datasets, each consisting of 1000 problem instances. In the first

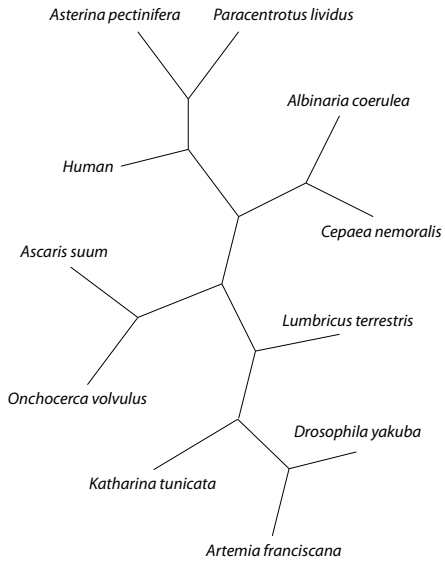

Figure 3.4: The tree computed by NEIGHBOR with the matrix in Table 3.3.

dataset, $n = 100$ and $d = 0$ (no duplicates); in the second dataset, $n = 100$ and $d = 50$. In both datasets, g' is generated from g by applying 10 inversions, 5 transpositions and 5 transversions. We presented the second dataset to both GENOMEPLAN and DERANGE 2, after relabeling the duplicates in the problem instances in random order. We provided some information to GENOMEPLAN about the duplicates as explained in Section 3.2.1. The cost of each rearrangement event was set to 1, and the priority was defined as the number of relieved breakpoints per cost; the priority of a swap was set to 2000 plus the number of relieved breakpoints. DERANGE 2 was run with look-ahead=4.

In the duplicate-free dataset, out of 1000 instances, GENOMEPLAN found more parsimonious solutions to 242 instances whereas DERANGE 2 computed more parsimonious solutions to 112 instances; for the rest of the instances, both systems computed solutions of equal cost. In the dataset with duplicates, out of 1000 instances, GENOMEPLAN found more parsimonious solutions to 277 instances whereas DERANGE 2 computed more parsimonious solutions to 112 instances. These results suggest that allowing 0-cost swaps of (originally) duplicate genes after relabeling duplicates is a more effective method than the naive approach of relabeling duplicates.

3.3.3 GENOMEPLAN vs. TD-ESTIMATOR

We also compared GENOMEPLAN with TD-ESTIMATOR [41], the only other software that can handle duplications and unequal gene content, on randomly generated instances. These instances were generated as described in the previous section. The randomly generated sequences contained 40% duplicate genes. Since GENOMEPLAN and TD-ESTIMATOR handle unequal gene content differently (GENOMEPLAN uses single in-

sertions and deletions whereas TD-ESTIMATOR uses block duplications and single gene losses), for a fair comparison these random instances were generated with equal gene content.

As mentioned before, Lin et al. try to estimate the actual number of events whereas we are trying to find the minimum number of events. Due to this fundamental difference of the problems addressed by TD-ESTIMATOR and GENOMEPLAN, comparing the results with respect to the actual distance or with respect to the minimal distance is unfair. To establish a fair comparison criteria, we made use of the following observation: a software that always correctly estimates the exact half of the true evolutionary distance leads to better phylogenies than a software that finds solutions closer to the actual distance but with some deviations. Following this observation, we compared GENOMEPLAN and TD-ESTIMATOR in terms of consistency in the estimations of the true evolutionary distances. In order to do that, we normalized the average of their estimations for the given problem instances to the true distance; and then we compared the squares of the errors of the normalized distances.

In our experiments we tested the accuracy of GENOMEPLAN and TD-ESTIMATOR in two different settings: we fixed the length of the genomes and increased the number of events (i.e., inversions, transpositions and transversions); and we fixed the number of events and increased the lengths of the genomes. Also, in each setting, we considered three cases where the ratio of the number of inversions to the number of transpositions and transversions was set to 1/1, 1/2, or 2/1. The cost of an inversion was set to 1; and the cost of a transposition/transversion was set to 1.5 (which is the derived cost of a transposition in Lin et al.'s experiments). The priority of an inversion/transposition/transversion was defined as the number of relieved breakpoints per cost. The priority of a swap was defined as 2000 plus the number of relieved breakpoints.

3.3.3.1 Fixed genome length but varying number of events

Table 3.4 shows the results of the experiments where we fixed the length of the genome to 125. The rows identified by A1–A5 (resp. B1–B5 and C1–C5) summarize the results of the experiments for the case where the ratio of the number of inversions to the number of transpositions and transversions was set to 1/1 (resp. 1/2 and 2/1). In each of these three cases, the number of events ranged from 12 (A1, B1, C1) to 60 (A5, B5, C5).

Each line in the table summarizes the results of experiments over 100 randomly generated instances. Consider, for instance, the row identified by A4. This row summarizes the results of experiments over 100 randomly generated instances, where each instance is generated by 30 inversions, 15 transpositions and 15 transversions as explained in the previous section; hence, the actual cost of the plan is 60. The average cost of the plans computed by GENOMEPLAN (resp. TD-ESTIMATOR) is 53.32 (resp. 58.12). The standard deviation for GENOMEPLAN (resp. tde) computed after normalization of these

Table 3.4: Comparison of GENOMEPLAN and TD-ESTIMATOR in the case with fixed genome length and increasing number of operations.

ID	genome length	#of inversions	#of transpositions	#of transversions	actual cost	average cost TD-ESTIMATOR	average cost GENOMEPLAN	normalized stdev TD-ESTIMATOR	normalized stdev GENOMEPLAN
A1	125	6	3	3	15	14.78	14.68	1.19	0.98
A2	125	12	6	6	30	29.11	28.47	2.52	1.77
A3	125	18	9	9	45	43.75	42.05	3.81	2.33
A4	125	24	12	12	60	58.12	53.32	4.90	3.11
A5	125	30	15	15	75	72.86	62.24	6.59	3.60
B1	125	4	4	4	16	15.53	15.86	1.15	1.02
B2	125	8	8	8	32	30.69	31.17	2.56	1.59
B3	125	12	12	12	48	46.48	45.02	3.48	2.43
B4	125	16	16	16	64	61.56	56.36	4.99	2.96
B5	125	20	20	20	80	76.95	64.37	6.80	3.46
C1	125	8	2	2	14	13.57	13.58	1.13	1.01
C2	125	16	4	4	28	27.56	26.49	1.79	1.49
C3	125	24	6	6	42	41.30	38.89	3.15	2.22
C4	125	32	8	8	56	54.83	50.05	4.41	2.64
C5	125	40	10	10	70	67.39	58.96	5.52	3.44

average costs to the actual cost is 3.11 (resp. 4.90).

In Table 3.4, we observe that the average costs calculated by GENOMEPLAN and TD-ESTIMATOR tend to be lower than the actual cost. As the number of events increases, the estimations of TD-ESTIMATOR usually remain just below the actual cost while GENOMEPLAN finds evolutionary distances with higher deviation from the actual cost; this is expected since TD-ESTIMATOR tries to estimate the actual cost while GENOMEPLAN tries to optimize the total cost. The normalized standard deviation increases as the number of events increase. These error values are lower for GENOMEPLAN, making it more advantageous over TD-ESTIMATOR. Similar observations are made with different ratios of the number of inversions.

3.3.3.2 Fixed number of events but varying genome length

Table 3.5 summarizes our experiments where the number of events was set to 30 and the genome length varied from 125 to 2000. It has a similar structure as in Table 3.4. The rows identified by D1–D5 (resp. E1–E5 and F1–F5) summarize the results for the case where the ratio of the number of inversions to the number of transpositions and transversions was set to 1/1 (resp. 1/2 and 2/1). In each of these three cases, the genome length varied from 125 (D1, E1, F1) to 2000 (D5, E5, F5).

In each case, we observe that the average costs of solutions increase and the normalized standard deviations decrease as the genome size increases. We also observe that, in shorter genomes (resp. longer genomes), the normalized standard deviation of the solutions computed by GENOMEPLAN (resp. TD-ESTIMATOR) is lower. The average cost of the solutions GENOMEPLAN becomes slightly higher than the actual cost when the length of the genome is increased to 2000. This behavior of GENOMEPLAN can be explained as follows. While generating the instances, as the genome length increases, it

Table 3.5: Comparison of GENOMEPLAN and TD-ESTIMATOR in the case with fixed number of events and increasing genome lengths.

	genome	#of	#of	#of	actual	average cost		normalized stdev	
ID	length	inversions	transpositions	transversions	cost	TD-ESTIMATOR	GENOMEPLAN	TD-ESTIMATOR	GENOMEPLAN
D1	125	15	8	7	37.5	36.74	35.90	2.67	1.94
D2	250	15	8	7	37.5	37.03	36.64	1.90	1.25
D3	500	15	8	7	37.5	37.46	37.38	1.41	1.21
D4	1000	15	8	7	37.5	37.05	37.32	1.07	0.78
D5	2000	15	8	7	37.5	37.44	37.84	0.70	1.02
E1	125	10	10	10	40	39.21	38.62	3.28	1.98
E2	250	10	10	10	40	39.33	39.54	2.00	1.58
E3	500	10	10	10	40	39.82	39.87	1.57	1.06
E4	1000	10	10	10	40	39.84	40.15	1.12	1.01
E5	2000	10	10	10	40	39.86	40.24	0.73	0.78
F1	125	20	5	5	35	34.29	33.01	3.03	2.06
F2	250	20	5	5	35	34.50	33.57	1.86	1.09
F3	500	20	5	5	35	34.85	34.52	1.37	1.06
F4	1000	20	5	5	35	34.85	34.88	0.79	0.83
F5	2000	20	5	5	35	35.06	35.12	0.66	0.92

becomes less likely for a rearrangement event to operate on a breakpoint generated by a previous event. Then, the ratio of the number of breakpoints to the number of events that generated them increases up to a point where the sequence of randomly generated events gets closer to an optimal plan. Therefore, as the genome length increases, harder problems are generated. On the other hand, recall that GENOMEPLAN does not guarantee finding optimal solutions.

3.4 Summary of Contributions

We have extended the work of Erdem and Tillier [24] with the following changes:

To handle duplicates, we include in the representation of genomes the information about which genes are duplicates of each other, and we introduce a 0-cost auxiliary action of swapping gene segments. After assigning a cost of 1 to the actions that characterize rearrangement events, the genome rearrangement problem can be reformulated as a planning problem that asks for a plan whose total cost is at most a given nonnegative integer k. By allowing 0-cost swaps of duplicates, we avoid enumerating all possible relabelings of the duplicates and solving the genome rearrangement problem for each possible relabeling as in Cui et al.'s work [15].

To improve the computational efficiency, we embed three heuristics in the action descriptions. The breakpoint heuristic ensures that the number of breakpoints decreases at each step of the plan; it reduces the branching factor of the search tree from $O(n^3)$ to $O(n^2)$ where n is the genome length. If a gene segment occurs in both genomes then the second heuristic identifies this gene segment as a "good segment" and maintains it as is through the search. It further reduces the branching factor of the search tree to $O(b^2)$ where b is the number of breakpoints. According to the third heuristic, a gene is consid-

ered "irrelevant" if it forms good segments with its two neighbor genes; if an action makes a gene irrelevant then that gene is discarded after the application of that action. Discarding irrelevant genes reduces the number of candidate rearrangement events considered at each step of the search.

To get more plausible results from the point of view of biology, one can incorporate domain-specific information into descriptions of actions that characterize rearrangement events. Such an incorporation can be achieved by assigning costs and priorities to these actions so that the search is guided towards more plausible solutions. For instance, transpositions may occur more often in some species, then we can assign a lower cost to transpositions to reflect this domain-specific information. In this way, with a cost-based search strategy where the goal is to find a small-cost plan, we expect to obtain a plan that involves transpositions instead of more expensive actions. On the other hand, priorities of actions affect the ordering of successor states in search: actions with higher priorities are selected before the ones with lower priorities if the search strategy is priority-based. The flexibility of assigning costs and priorities to events is also important in understanding the frequency of different events [52] in the evolution of species.

We implemented a genome rearrangement software, called GENOMEPLAN, based on the methods described above, utilizing the facilities of the AI planner TLPLAN [1]. Although the genomes of many species (in particular, the chloroplast genomes) have unequal gene content and duplicate genes, most of the existing genome rearrangement software (e.g., GRIMM [57], GRAPPA [46], DERANGE 2 [7], MGR [9]) cannot handle them directly. Being able to represent and modify genome rearrangement problems in a high-level formalism, and to choose the search strategy and cost/priority settings to solve the problem allows us a flexible tool to analyze and better understand evolutionary history of species. In this sense, GENOMEPLAN provides an alternative tool to solving genome rearrangement problems.

We illustrated the applicability and the effectiveness of our planning-based approach to genome rearrangement in three sorts of experiments using GENOMEPLAN. To show the usefulness of our planning-based approach to genome rearrangement, we experimented with three sets of real data: mitochondrial genomes of *Metazoa* (animals with a nervous system, and muscles) [8], chloroplast genomes of *Campanulaceae* (flowering plants) [13], and chloroplast genomes of various land plants and green algae [15]. We observed that our results conform with the most recent and widely accepted results. To compare our planning-based approach to genome rearrangement with duplicate genes, with the naive approach (where we relabel duplicates uniquely and use an existing genome rearrangement software system that can handle transpositions, inversions and transversions and whose goal is to find a parsimonious solution), we experimented with a set of randomly generated problem instances. In these experiments, we used DERANGE 2 since it is the only such available system. We observed that GENOMEPLAN computes

more parsimonious solutions compared to DERANGE 2. Another available genome rearrangement software that can handle inversions, transpositions, and transversions is TD-ESTIMATOR [41]. Unlike GENOMEPLAN and DERANGE 2, TD-ESTIMATOR tries to approximate the true distance between two genomes in terms of the Double-Cut-Join operator, gene losses and duplications. We experimented with a set of randomly generated problem instances with duplicates and with equal gene content. observed that GENOMEPLAN is comparable with TD-ESTIMATOR in terms of accuracy (deviation of the estimated cost from the actual cost, after normalization).

Chapter 4

Decoupled Planning for Multiple Teams of Robots

Consider a domain of multiple teams of robots with each team located in a separate workspace, working toward completion of their assigned tasks. Let each team be composed of several types of robots with different capabilities and let some types of robots be able to change their end-effectors to perform different actions. Given the state of each workspace and designated tasks for each team, the goal is for all the teams to complete these tasks in a minimum number of steps. We can divide this problem into several independent planning problems (i.e., one plan for each team); and thus can solve it by planning the actions of each team.

To make more efficient use of shared resources (e.g., robots), let us assume that teams can exchange robots: at any step, a team can lend one of its robots to another team. A transportation delay (in terms of number of steps) is associated with such exchanges, since it takes time for a robot to move from one workspace to another. This assumption on the exchange of robots between teams complicates the whole problem, since the problem cannot always be divided into independent smaller problems. One straightforward way to solve this modified problem might seem to formalize the whole domain, and pose the problem above as a single planning problem; however, the domain description and the search space gets too large.

We propose to solve a restricted version of this problem where a team can either lend or borrow a robot, but not both and a team can not lend or borrow more than one robot. We also assume that all actions of robots are discrete and take a single step, and the teams start executing their plans at the same time. The goal is to find a plan (for each team, possibly with robot exchanges) where the tasks of all teams are completed in minimum number of steps.

Our solution is based on the following idea (Fig. 4):

- Each team has a representative agent. The representative agents can find optimal plans (with concurrent actions) for that team. The representative agents can also answer certain types of yes/no queries, such as “can your team complete its task in

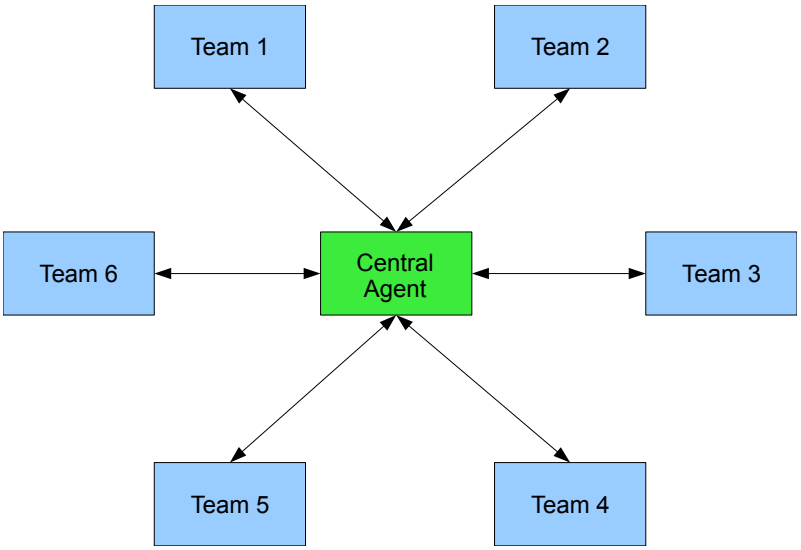

Figure 4.1: Our general approach.

k steps, while also lending a robot before step k'?".

- There is a central agent, that communicates with the representative agents through these kinds of queries, in order to find a minimum length plan.

Our solution involves a combination of causal reasoning and decoupled planning:

- We represent the domain in the action language $\mathcal{C}+$ [33] so that every team can reason about the domain using the causal reasoner CCALC [43] with respect to their own parameters (e.g., number of robots).

- We introduce an intelligent algorithm for the central agent to decide how to decouple plans (i.e., which teams lend robots to which other teams, and when) to find an optimal overall plan, by communicating with the representatives of the teams.

Our approach can be applied to many challenging domains with multiple heterogenous teams of self-reconfigurable robots (e.g., search and rescue robots, cognitive factories). We show its applicability on a Cognitive Painting Factory scenario, providing also a good case study for future intelligent factories [64]. We also discuss how our optimal decoupled planning algorithm can be embedded in an execution and monitoring framework, allowing the effective reuse of previously computed results in case of plan failures (e.g., when a robot gets broken, or tasks are reassigned).

The rest of this chapter is organized as follows: We first briefly introduce the Cognitive Painting Factory in Section 4.1, followed by explaining how we model the workspace of a single team as a planning problem in $\mathcal{C}+$, in order to answer different kinds of queries asked by the central agent (Section 4.2). We then explain the algorithm our central agent utilizes to find a collective optimal decoupled plan and analyze it in Section 4.3. Following a description of how we embed our optimal decoupled planning algorithm in an

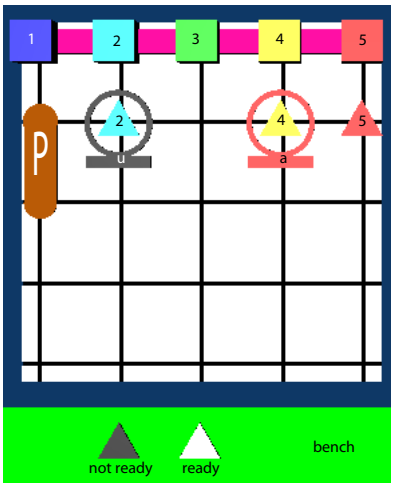

Figure 4.2: A sample workspace.

execution and monitoring framework in Section 4.4, we discuss the related work in Section 4.5 and conclude.

4.1 A Cognitive Painting Factory Scenario

Consider a Painting Factory with multiple workspaces, where each workspace produces different colored boxes (by successively painting, waxing and stamping an unpainted box) with a team of robots consisting of several self-reconfigurable worker robots and a single carrier robot. Worker robots can move horizontally and change their end-effectors to do different tasks, while a carrier robot can move both horizontally and vertically and push or pull the worker robots, after attaching to them. Each workspace is depicted as a grid, as shown in Fig. 4.1, contains an assembly line along the north wall to move the boxes and a pit stop area where the worker robots can change their end-effectors.

To make more efficient use of shared resources, teams can exchange robots: at any step, a team can lend one of its worker robots through their pit stop such that after a transportation delay the worker robot shows up in the pit stop of a borrowing team. Here, we are assuming that the end effector (e.g., green painter) of the robot is removed after it leaves the lending team's workspace and a new end effector (e.g., red painter) is mounted on it before it enters the borrowing team's workspace; and all this extra work is accounted for in the transportation delay.

Initially, we are given each workspace's state, with the number of boxes to be painted; the goal is for all teams to paint the specified number of boxes in minimum number of steps. While a plan is being executed, it is possible that a robot may get broken so that it can not attach to another robot or work on a box, or that tasks assigned to teams (e.g., the number of orders) may be modified.

4.2 Representing the Painting Factory Domain

We describe the painting factory domain in the action description languag $\mathcal{C}+$ and the reasoning problems in the action query language accepted by the causal reasoner CCALC, like in [12, 25]. Inheriting the advantages of $\mathcal{C}+$, CCALC allows reasoning about true concurrency, ramifications, defaults, and various reasoning tasks (planning with temporal constraints, prediction, etc.) over the given domain description. See Chapter 2 for detailed information about $\mathcal{C}+$ and CCALC.

Our main goal is to be able to answer the following kinds of queries, for a single team:

- *Can the team complete its task in k steps?*
- *Can the team complete its task in k steps, while lending a robot before step k'?*
- *Can the team complete its task in k steps, by borrowing a robot after step k'?*

Let us first consider a single team in a single workspace, explain the domain description in $\mathcal{C}+$, and show how a single team can use CCALC with this formalization for planning to answer the first type of queries. After that, we explain how to modify the domain description to allow exchanges of robots. Last, we show how to find plans with a smaller number of actions (i.e., spend less energy to achieve the same goals).

4.2.1 Domain Description: No Robot Exchanges

We view the workspace as a 3×5 grid (Fig. 4.1) with the lower left corner being $(1, 1)$. We represent the carrier robot by the constant `c1`; n worker robots by the constants `w1`, `w2` ... `wn`; and each one of b boxes with a distinct number in $\{1 \dots b\}$.

4.2.1.1 Fluents

The fluents used in describing the Painting Factory are summarized in Table 4.1.

The robots are supposed to be located at grid squares; therefore, the location of a robot `R` is specified by two functional fluents, `xpos(R)=X` and `ypos(R)=Y`.

The boxes are supposed to be located in some order on the assembly line; therefore, the location of a box `B` on the assembly line is specified by a single fluent `linePos(B)=L` where `L` is in $\{1-b, \dots, 5+b\}$. Here $\{1-b, \dots, 0\}$ and $\{6, \dots, 5+b\}$ denote the boxes that completed the painting process, or that are not yet taken in the painting process.

The status of a box `B` is denoted by the functional fluent `workDone(B)=WS` where `WS` stands for a work stage: `0`–unprocessed, `1`–painted, `2`–waxed, `3`–stamped.

A newly painted box is wet and it has to be left to dry before it can be waxed; to formalize this transition constraint, we need a relational fluent `wetpaint(B)` to show that box `B` is wet.

Table 4.1: Fluents for representing the Painting Factory.

Fluent	Description
`xPos(R)=X, yPos(R)=Y`	Coordinates of robot `R` on the grid.
`linePos(B)=L`	Position of box `B` on the assembly line.
`workDone(B)=WS`	Work stage of a box `B`. `WS` can be `unprocessed, painted, waxed, stamped`.
`wetPaint(B)`	Indicates the box `B` has wet paint.
`endEffector(W)=E`	Worker `W` has the end effector `E`; which can be: `painter, waxer, stamper`.
`attached(C,W)`	Carrier `C` is attached to worker `W`.

The functional fluent `endEffector(W)=E` denotes that the worker robot `W` has the end-effector `E`; the value of `E` denotes the role of the worker relative to the work stage: `1`–painter, `2`–waxer, `3`–stamper.

The carrier robot needs to attach to and detach from the worker robots, to be able to carry them along the vertical axis. Therefore, we need a relational fluent `attached(C,W)` to express that the carrier robot `C` is attached to the worker robot `W`.

Using fluents only, we can describe state constraints by means of static causal laws. For instance, we formalize that at no state of the world two worker robots `W1` and `W2` are on the same grid cell, by the causal law

```
caused false if xpos(W1)=xpos(W2) & ypos(W1)=ypos(W2) where W1\=W2.
```

4.2.1.2 Actions

The actions used in describing the Painting Factory are summarized in Table 4.2.

A robot `R` (which may be a worker `W` or carrier `C`) can move in the direction `D` by one unit; we denote this action by `move(R,D)`. A worker robot `W` can perform the following actions: `swapEndEffector(W,E)`–changing its end-effector to `E`, `workOn(W,B)`–working on a box `B` to proceed to the next work stage. A carrier robot can perform the following actions: `attach(C,W)`–attaching to a worker robot `W`, `detach(C)`–detaching from the worker robot it is attached to, `push(C)`–pushing the worker robot (it is attached to) vertically by one unit, `pull(C)`–pulling the worker robot (its is attached to) vertically by one unit. In addition to the actions of robots in a team, there is also the action of shifting the assembly line, denoted by `lineShift`.

In $\mathcal{C}+$, we describe actions and change by causal laws. Consider, for instance, the action `workOn(W,B)`. We formalize by the following causal law that this action, as its direct effect, increments the work stage `WS` of a box `B` if the worker robot `W` is working on `B`:

Table 4.2: Actions for representing the Painting Factory.

Action	Description
`move(R,D)`	Robot `R` moves 1 step towards direction `D`. Workers cannot move vertically.
`push(C), pull(C)`	Carrier `C` pushes/pulls the worker it is attached to. Only way for the workers to move vertically.
`attach(C,W)`	Carrier `C` attaches to worker `W`. `C` must be on top of `W`.
`detach(C)`	Carrier `C` detaches from the worker it is attached to.
`swapEndEffector(W,E)`	Worker `W` swaps its end effector to `E`. `W` must be in the pit area.
`workOn(W,B)`	Worker `W` works on the box `B`. Current work-stage of `B` and end effector of `W` must match.
`lineShift`	Line shifts, causing all the boxes to move.

```
workOn(W,B) causes workDone(B)=WS if workDone(B)=WS-1.
```

The action `workOn(W,B)` denotes painting if the current work stage is `0`, i.e., `workDone(B)=0`. Therefore, we formalize that painting a box `B` causes the box to have wet paint, by the causal law

```
workOn(W,B) causes wetpaint(B) if workDone(B)=0.
```

Similarly, we describe the direct effects of other actions by causal laws.

We can describe change that does not directly involve an action of a robot. For instance, we formalize that a box with wet paint gets dry, by the causal law

```
caused -wetpaint(B) after wetpaint(B).
```

We describe preconditions of actions by causal laws as well. For instance, we formalize by the following causal law that the action `move` is not possible if the robot is at the rightmost border (`maxX`):

```
nonexecutable move(R,right) if xpos(R)=maxX.
```

We describe that a robot `W` cannot work on a box `B` that still has wet paint, by the causal law:

```
nonexecutable workOn(W,B) if wetpaint(B).
```

and that a worker robot `W` can work on a box `B` only if the worker has the appropriate end-effector for the next work stage, by the causal law:

```
nonexecutable workOn(W,B) if endEffector(W)=WS & workDone(B)\=WS-1.
```

Similarly, we can express that a worker robot can work on a box if it is right next to the assembly line and it is aligned with the box, that the worker robots do not move vertically, that a worker robot can swap its end-effector only if it is in the pit area, and other preconditions of the worker's actions, etc.

We can also formalize the preconditions of a carrier's actions: A carrier attaches to a worker only if it is right on top of it; a carrier can not push/pull a robot if it is not attached to it.

Concurrent actions are allowed unless specified otherwise. We express the nonexecutability of two actions concurrently, also by causal laws. For instance, we can express that a robot cannot move in two different directions (to prevent diagonal moves) by the causal law

```
nonexecutable move(R,D1) & move(R,D2) where D1\=D2.
```

a carrier cannot detach and pull at the same time by the causal law

```
nonexecutable detach(C) if pull(C).
```

Similarly, we prevent the following concurrent actions: A worker cannot work on a box while the line is shifting; the pushing/pulling carrier robot and the pushed/pulled worker robot cannot be involved in any other action; and a moving robot cannot attach or detach or work on a box.

4.2.1.3 Finding Plans without Robot Exchanges

We present planning problems to CCALC by means of queries. For instance, the following query asks for a shortest plan whose length is at least 18 and at most 100, for a team with one worker (`w1`) and one carrier (`c1`):

```
:- query
maxstep :: 18..100;
0: % no robot is attached to another
 [/\C /\W |-attached(C,W)],
 % no block has wetpaint
 [/\B | -wetpaint(B)=0],
 % worker is at (1,3); carrier is at (1,1)
 xpos(w1)=1, ypos(w1)=3, xpos(c1)=1, ypos(c1)=1,
 % boxes are not yet processed
 [/\B | linePos(B)=B+lineLength, workDone(B)=0];
maxstep: % all boxes are painted
 linePos(maxBox)=0, [/\B | workDone(B)=3].
```

Then, with the domain description whose some parts are explained above, CCALC finds a shortest plan of length 29 in 10 CPU seconds (on a workstation with two 1.60GHz Intel Xeon E5310 Quad-Core Processor and 16 GB RAM, running Centos 64bit (Version 5.3)) using the parallel SAT solver MANYSAT [34] as its search engine; it spends 2.4 CPU secs to find a plan of length 29 (4784 atoms and 28897 clauses) and 7.6 CPU secs to verify the nonexistence of a plan of length $k = 18, ..., 28$.

4.2.2 Domain Description: Exchanges of Robots

We extend the basic formulation for CCALC above to answer two different kinds of queries, namely, for a specific intermediate Step k':

- "*Can the team complete its task in k steps, while lending a robot before step k'?*"
- "*Can the team complete its task in k steps, by borrowing a robot after step k'?*"

For this extension, we introduce a new fluent `bench(W)` to describe that a worker robot `W` is at the bench area, and thus it cannot be part of any of the previous actions, and it is also forced to be outside the workspace (`ypos(W)=0`). We also add two new actions, namely `giveRobot(W)` which puts `W` to the bench (`W` needs to be in the pit stop area first) and `takeRobot(W,X,Y)` which takes `W` from the bench and puts it at `(X,Y)` in the pit stop area.

The direct effect of `takeRobot(W,X,Y)` is described by the causal laws:

```
takeRobot(W,X,Y) causes xpos(W)=X.
takeRobot(W,X,Y) causes ypos(W)=Y.
```

whereas the following describe some of its preconditions:

```
nonexecutable takeRobot(W,X,Y) if -bench(W).
```

With these additional fluents and actions, we can express the two sorts of queries mentioned above. CCALC allows us to add constraints as parts of queries. For example, we can add before the statement of goal in the query above, the line

```
10: workDone(1)=3;
```

to introduce a subgoal that the first box should be stamped by Step 10. We can add the following constraint to ensure that at least one worker is in the bench at step k':

```
k': [\/W | xpos(W)=minX-1], [\/W | bench(W)];
```

By this way, we can answer the query "*Can the team complete its task in k steps, while lending a robot before step k'?*", provided also that no robot is borrowed meanwhile with:

```
nonexecutable takeRobot(W,X,Y).
```

Disabling taking robots prevents the cases where a robot is given before step k', remains in the bench at step k', and then taken back.

The queries that involve borrowing a robot are handled similarly. This time we add an extra robot to the problem, say `we`, and force it to be in the bench at step k':

```
k': bench(we);
```

We also disable the lending of robots, similar as before.

4.2.3 Eliminating Redundant Actions

In the plans found by CCALC, we have observed some unnecessary actions, such as:

- A worker swapping its end effector two steps in a row
- The carrier attaching to a worker and detaching immediately after
- Robots moving in one direction and then the opposite direction (including push and pulls)

These are undesirable behaviours in a factory and they should be eliminated. To this end, we further modify the formulation, to force CCALC to find plans with less redundancy. However, while eliminating the redundancies, we should not eliminate a solution. In the following, we give detailed explanations of how we eliminate the redundancies related to end effector swappings, carrier attachment-detachments, and robot movement. We then discuss the issues with redundancy elimination and how we deal with them.

4.2.3.1 Eliminating Redundant Swaps

First of all notice that instead of swapping its end effector twice in a row, a worker can just sit in the pit stop for one step and swap its end effector to the desired one in the next step. This kind of modification to the plan does not interfere with the work of other robots, therefore does not break the completeness of the method (i.e., a solution is not eliminated). One method of eliminating swap redundancies is with the following constraint:

```
nonexecutable swapEndEffector(W,E1) after swapEndEffector(W,E2).
```

which prevents a worker from swapping its end effector twice in a row. However, this constraint leaves out the case where the worker swaps its end effector every other step (or every three steps etc.).

To find a more general solution to the problem, we propose a *token* system, where robots obtain tokens for performing specific tasks and remove them by performing other

tasks. A token penalyzes a robot by not allowing it to perform some actions. For example, in the example above, we can give a relevant token, *swap token* to any worker `W` that swaps its end effector. While a worker has a *swap token*, it can no longer perform other swaps, until its token is removed by working on a box. To include this idea in our formulation, we introduce a new fluent, `swaptoken(W)`, to denote that `W` has a *swap token*, and extend the formulation with the following causal laws:

```
swapEndEffector(W,E) causes swaptoken(W).
nonexecutable swapEndEffector(W,E) if swaptoken(W).
workOn(W,B) causes -swaptoken(W).
```

Also, while finding a plan with redundancy elimination, a worker `W` always starts without a *swap token* and must not have a *swap token* in the goal state. This can be achieved with the following additions to a query:

```
0: [/\ W | -swaptoken(W)].
maxstep: [/\ W | -swaptoken(W)].
```

The intuition behind this method can be explained by *purpose* and *commitment*: the only purpose of swapping an end effector is to be able to work on a box; so once a worker swaps its end effector, it actually commits to work on a box. It is not able to perform further swaps until it fulfills this commitment. If it does not fulfill the commitment during the plan, that means there was no purpose in swapping the end effector in the first place and therefore it was a redundant action. Note that our solution with tokens also eliminates the cases where a worker swaps its end effector and not work on a box for the rest of the plan, or where a robot swaps, moves left, then moves right and swaps again etc.

4.2.3.2 Eliminating Redundant Attachments/Detachments

To eliminate redundancies of attachments, followed by a detachment, we introduce a fluent `attachtoken(C)` to denote that `C` has an *attach token*. Then we specify that a carrier receives an *attach token* when it attaches to any worker and can not detach while it has the token. An *attach token* can be removed only by pushing or pulling the attached worker (since this is the purpose of attaching in the first place). We add the following causal laws for *attach token*s:

```
attach(C,W) causes attachtoken(C).
nonexecutable detach(C) if attachtoken(C).
push(C) causes -attachtoken(C).
pull(C) causes -attachtoken(C).
```

To eliminate a similar case of redundancy, where a carrier detaches from a worker and attaches to it again, we use a *detach token*. Note that this is a token for a carrier-worker pair, which is given when carrier `C` detaches from worker `W`. While this token is

active `C` can not attach to `W`. The token is removed when `C` attaches to a different worker (which is an acceptable excuse for detaching), or when the worker performs a horizontal move (which is also an acceptable excuse, since a worker and a carrier can not move horizontally while attached). We use the fluent `detachtoken(C,W)` to denote that the pair `C,W` has a *detach token*. We extend the formulation with the following causal laws:

```
detach(C) causes detachtoken(C,W) if attached(C,W).
nonexecutable attach(C,W) if detachtoken(C,W).
attach(C,W) causes -detachtoken(C,W1).
move(W,D) causes -detachtoken(C,W).
```

As with the swap eliminations, we also force no *attach token*s or *detach token*s to be present in the initial and goal states:

```
0: [/\ C | -attachtoken(C)], [/\ C /\ W| -attachtoken(C,W)].
maxstep: [/\ W | -attachtoken(C)], [/\ C /\ W| -attachtoken(C,W)].
```

4.2.3.3 Eliminating Movement Redundancies

Elimination of redundant carrier movement is easier, compared to eliminating redundant worker movement. In our factories carriers can not collide with any other robots and therefore can roam freely and can always choose to move between any two points $((x_1, y_1), (x_2, y_2))$ on the grid, using the minimum number of steps. Note that this minimum number of steps is equal to $|x_2 - x_1| + |y_2 - y_1|$, and can be achieved by never moving in opposite directions during movement (i.e., if the carrier moves left, then it should not move right). To force this optimal movement for the carrier, we use four tokens, one for each direction: *no-left token*, *no-right token*, *no-up token*, *no-down token*. Whenever the carrier moves left, it receives a *no-right token*, and is not able to move right while it has a *no-right token* (similar rules for the other three directions). Any such token is removed immediately when the carrier attaches to a worker, since the carrier's reason for moving is fulfilled. We introduce the fluent `norighttoken(C)` which denotes that the carrier `C` has a *no-right token* (three other fluents for other directions) and we add the following causal laws (for one direction, similar laws for the other directions):

```
move(C,left) causes norighttoken(C) if -attached(C).
nonexecutable move(C,right) if norighttoken(C).
attach(C,W) causes -norighttoken(C).
```

Eliminating the redundancies in worker's moves on the grid is more tricky. The two primary purposes of worker movement are: 1) to get to a grid cell to perform some task (work on a box, swap end effector, meet halfway with the carrier, go to the pit stop to be given to another team), or 2) to let another robot pass. Our approach for eliminating

redundant worker movement is based on using *trail token*s: A worker leaves a *trail token* on a grid cell, when it leaves the cell (including push/pulls) and a worker can not move (or pushed/pulled) on top of its own *trail token*. All the trail of a robot is erased when it works on a box, changes its end effector, it is attached to, or it is lent to another team. Also, other workers can erase parts of the trail of a worker when they move over it. This is to justify the cases where a worker moves away to avoid collisions. We introduce the fluent `trailtoken(X,Y,W)` to denote that worker `W` has left a *trail token* on the grid cell (`X,Y`) and extend the formulation with the following causal laws:

```
caused trailtoken(X,Y,W) if -[\/W1 | xpos(W1)=X & ypos(W1)=Y]
                         after xpos(W)=X & ypos(W)=Y.

workOn(W,B) causes -trailtoken(X,Y,W).
swapEndEffector(W,E) causes -trailtoken(X,Y,W).
attach(C,W) causes -trailtoken(X,Y,W).

caused -trailtoken(X,Y,W) if xpos(W1)=X & ypos(W1)=Y
                           where W \= W1.

caused false if xpos(W) = X & ypos(W) = Y & trailtoken(X,Y,W).
```

As with other forms of redundancy eliminations, we also force no *no-left token*, *no-right token*, *no-up token*, *no-down token*, or *trail token*s to be present in the initial and goal states:

```
0: [/\ C | -nouptoken(C)], [/\ C | -nodowntoken(C)],
[/\ C | -norighttoken(C)], [/\ C | -nolefttoken(C)],
[/\ X /\ Y /\ W | -trailtoken(X,Y,W)].
maxstep: [/\ C | -nouptoken(C)], [/\ C | -nodowntoken(C)],
[/\ C | -norighttoken(C)], [/\ C | -nolefttoken(C)],
[/\ X /\ Y /\ W | -trailtoken(X,Y,W)].
```

4.2.3.4 Discussion

The effects of using redundancy elimination methods (on the size of the formulation, planning time and number of actions in plans) are shown in Table 4.3. There is a 25% decrease in the number of actions in the plan computed with redundancy elimination, at the cost of increased planning time. There are two problems with our methods:

Planning Time: Plans with redundancy elimination can take significantly more time to find (more than 3 times, for the instance in Table 4.3). The reason for this is the increased formulation size (almost 3 times the number of atoms and clauses, compared to the no redundancy elimination version). To avoid this issue as much as possible, we do

Table 4.3: Effects of using redundancy elimination in the formulation of a workspace with 2 workers, 1 carrier, 4 boxes, while finding a plan of length 30.

redundancy elimination	# of atoms	# of clauses	time	# of actions
none	7952	64931	2.34964	75
all	14994	113576	5.40018	57

not use the formulation with redundancy elimination to answer the queries asked by the central agent. We use redundancy elimination only when a decoupled plan is found with our optimal decoupled planning algorithm (Algorithm 8). This is discussed in more detail in Section 4.3.

Completeness: Even though the redundancy elimination methods above work effectively most of the time, we have encountered several cases where the completeness of the method is not preserved due to our worker movement redundancy elimination method (i.e., plans of certain length and lend/borrow times are found with*out* redundancy elimination but are *not* found with redundancy elimination). For these cases where a plan with redundancy elimination is expected but not found, we simply remove the worker movement redundancy elimination parts, and continue with the redundancy eliminations for swaps, attachs-detachs, and carrier movement, all of which preserve completeness. Finding a completeness preserving version of worker movement elimination method is part of our future work.

4.3 Optimal Decoupled Planning

In this section, we show how the central agent communicates with (asks queries to) the team representatives in order to find an optimal decoupled plan.

Throughout this section, we only consider the central agent, which can directly communicate with the team representatives. Representative of team t, denoted by R_t, can be asked certain kinds of queries, such as "can your team complete its task in k steps, while also lending a robot before step k'?", and always provide a correct answer; the representative can take an arbitrary amount of time to find a plan and respond, but it does so eventually. The team representatives plan to answer their respective queries in parallel.

We also make excessive use of *summaries* for teams. $S_t^k = \langle role, l, u \rangle$ summarizes what is known about team t for plan length k. Usually the reference to team summaries are for a fixed plan length, and the k is omitted in the representation. Summaries and what exactly they summarize are discussed in detail in the following subsection.

One further note: instead of saying "a team can complete its task when it lends a robot before step k'", we simply say "a team can lend a robot before step k'" and assume that it

also has to complete its own task while lending a robot.

This section is organized as follows: We first describe how the central agent tries to find a decoupled plan of a fixed length k in Subsection 4.3.1, then we give the algorithm for finding a decoupled plan of minimum length, in Subsection 4.3.2. We follow by showing how we can modify the algorithms to reduce the number of queries asked to team representatives, by using their previous answers, in Subsection 4.3.3.

4.3.1 Finding Decoupled Plans of Fixed Length

Given a plan length k, a transportation delay d, a list of already benched robots and when they are ready to be lent to a team, and access to the representatives $R_1, R_2, \ldots R_n$ for n teams; the goal is to determine if the teams can be coordinated in such a way that each team completes their tasks in at most k steps, by lending or borrowing a single robot (or not getting involved at all).

4.3.1.1 Observations

Let's start by some observations and explaining how we use them:

- For a given plan length k, a team can either complete its task in k steps, in which case it may be able to lend a robot to another team; or it can not, in which case it may be able to complete its task by borrowing a robot. Following this observation, for any plan length k, the role of a team can be determined by trying to find a plan of length k without robot exchanges. Note that a lending team may not be able to lend a robot at all, or a borrowing team may not be able to complete its plan even with an immediately borrowed robot (in which case there can be no decoupled plan of length k), but they are still labeled as a $lender$ or a $borrower$.

- For a lending team, the following holds: "If it can lend a robot before step k', then it can definitely lend a robot before step $k'' > k'$". Similarly, the contrapositive statement also holds: "If it can not lend a robot before step k'', then it can not lend a robot before step $k' < k''$". Following these observations, we can say that there is a unique k^* for plan length k, where for all $k' = 0, 1, 2, \ldots k^* - 1$, the team can not lend a robot before step k', and for all $k' = k^*, k^* + 1, \ldots, k$, the team can lend a robot before step k'. One way of pinpointing k^* would be to perform a binary search, starting with the bounds l, u (set to $0, k + 1$ initially) and trying to find a plan where a robot is lent before $\lfloor (l + u)/2 \rfloor$. If a plan is found, we reduce u, else we increase l; when $l + 1 = u$, u is equal to k^*. At any stage of the binary search, l shows the highest known value for k' which a robot can not be lent before, and u shows the lowest known value for k' which a robot can be lent before. In our

Algorithm 1 FINDDECOUPLEDPLAN

Input: plan length k; team representatives $R_1, R_2, \ldots, R_n$, transportation delay d; an array $bench$ of benched robots, with the number of steps remaining before they can be given to a team
Output: success if a decoupled plan can be found, failure otherwise

// summaries $S_1, S_2, \ldots, S_n$ for each team, where $S_t = \langle role, l, u \rangle$;

for all teams t **do**
  $S_t \leftarrow \langle none, 0, k+1 \rangle$; *// set role, bounds*
  call DETERMINEROLE(k, S_t, R_t) as a separate thread;
while $\exists t(S_t.l + 1 \neq S_t.u)$ **do**
  wait for a team's summary to be updated;

  for all t with $\neg R_t.isPlanning$ **and** $S_t.l + 1 < S_t.u$ **do**
    call TIGHTENBOUNDS(k, S_t, R_t) as a separate thread;
return CANMATCH, $S_1, S_2, \ldots, S_n$;

Algorithm 2 DETERMINEROLE

Input: plan length k; summary $S_t = \langle role, l, u \rangle$ for team t; R_t (representative of team t)
Output: The role of the team t for plan length k ($S_t.role$) is updated when the planning stops; $R_t.isPlanning$ is true while team t is planning

$R_t.isPlanning := true$;
Ask R_t for a plan of length k, without exchanges;

wait for R_t to responds with $answer$;
if $answer = success$ **then**
  $S_t.role := lender$;
else // $answer = failure$
  $S_t.role := borrower$;
$R_t.isPlanning := false$;

decoupled planning algorithm, the known values for intermediate steps where the teams can lend/borrow robots are compared to check if a decoupled plan is found.

- For a borrowing team, similar observations can be made with one major difference: Notice that, as k' decreases, it becomes harder for a lending team to lend a robot before k', whereas it gets easier for a borrowing team to borrow a robot after step k'. For the binary search, this difference translates as the lower bound l being increased when we successfully find a plan and the upper bound u being decreased when no plan is found, instead of the other way around.

4.3.1.2 The Main Algorithm

With these observations in mind, the naive decoupled planning algorithm (Algorithm 1) is as follows: For all teams t, first determine their roles, then find their earliest lend or latest borrow times (t^*) by performing the binary search described above. If each borrowing team b (with borrow time b^*) can be matched to a unique lending team l (which can lend a robot by step $l^* < b^* - d$) or a spare robot r (that has to stay in the bench for r^* more

Algorithm 3 TIGHTENBOUNDS

Input: plan length k; summary $S_t = \langle role, l, u \rangle$ for team t; R_t (representative of team t)
Output: Tighter bounds after team t finishes planning (if the bounds are not tight already), $R_t.isPlanning$ is true while team t is planning

if $S_t.l + 1 = S.t_u$ **then** // *already tight bounds*
    return;

$k' := (S_t.l + S_t.u)/2$;
$R_t.isPlanning := true$;
if $S_t.role = lender$ **then**
    Ask R_t for a plan of length k, where a robot is lent before k';
    wait for R_t to responds with $answer$;
    if $answer = success$ **then**
        $S_t.u := k'$;
    else // $answer = failure$
        $S_t.l := k'$;
else // $S_t.role = borrower$
    Ask R_t for a plan of length k, with a robot borrowed after k';
    wait for R_t to responds with $answer$;
    if $answer = success$ **then**
        $S_t.l := k'$;
    else // $answer = failure$
        $S_t.u := k'$;
$R_t.isPlanning := false$;

Algorithm 4 CANMATCH

Input: plan length k; transportation delay d; an array of benched robots $bench$, with the number of steps remaining before they can be given to a team; summaries $S_1, S_2, \ldots, S_n$ for all teams t with $S_t = \langle role, l, u \rangle$ with full information (i.e., $role$ is set, $u - l = 1$)
Output: success if a matching exists with the roles and bounds of the teams, failure otherwise

$L, B \leftarrow$ empty sets for lend and borrow times;
$L \leftarrow$ the values from $bench$;

for all teams t **do**
    if $S_t.role = lender$ **then**
        $L := L \cup \{S_t.u + d\}$;
    if $S_t.role = borrower$ **then**
        $B := B \cup \{S_t.l\}$;

for $i = 1, 2, \ldots, |B|$ **do**
    $borrow \leftarrow i^{th}$ lowest value in B;
    $lend \leftarrow i^{th}$ lowest value in L;
    if $lend \geq borrow$ **then**
        return false; // *a borrowing team's needs can not be met*
return true; // *all borrowers are matched with a lent robot*

steps, where $r^* < b^*$), then we say a decoupled plan is found, else we say no decoupled plan can be found.

For implementing this algorithm, we keep "summaries" of teams. A summary for a team t is a triplet $S_t = \langle role, l, u \rangle$ where $S_t.role$ denotes the role of that team ($lender$,

Algorithm 5 FINDDECOUPLEDPLAN (updated with early termination)

Input: plan length k; team representatives $R_1, R_2, \ldots, R_n$, transportation delay d; an array $bench$ of benched robots, with the number of steps remaining before they can be given to a team
Output: success if a decoupled plan can be found, failure otherwise

// summaries $S_1, S_2, \ldots, S_n$ for each team, where $S_t = \langle role, l, u \rangle$;

for all teams t **do**
  $S_t \leftarrow \langle none, 0, k+1 \rangle$; *// set role, bounds*
  call DETERMINEROLE(k, S_t, R_t) as a separate thread;
loop
  wait for a team's summary to be updated;

  if CANDEFINITELYMATCH($k, d, bench, S_1, S_2, \ldots, S_k$) **then**
    return success, $S_1, S_2, \ldots, S_n$;
  if $\neg$CANPOSSIBLYMATCH($k, d, bench, S_1, S_2, \ldots, S_k$) **then**
    return failure;

  for all t with $\neg R_t.isPlanning$ **and** $S_t.l + 1 < S_t.u$ **do**
    call TIGHTENBOUNDS(k, S_t, R_t) as a separate thread;

$borrower$, or $none$ if it is not determined yet), and l, u denote the associated bounds. The summary of the team starts as $\langle none, 0, k+1 \rangle$, its most uninformative state, and is iteratively made more informative by asking queries to the team representative R_t, up to its most informative state where $S_t.l + 1 = S_t.u$. We use the functions DETERMINEROLE (Algorithm 2) and TIGHTENBOUNDS (Algorithm 3) to update the summaries of teams. Both of these functions are used to ask a query to a team representative, and update the summaries when an answer is received.

Note that the functions DETERMINEROLE and TIGHTENBOUNDS are called as separate threads, parallel to the main algorithm. These functions ask a query to a team representative, wait for its answer, update the team's summary accordingly and return. The main algorithm continues its flow and does not wait for the calls to DETERMINEROLE and TIGHTENBOUNDS to return. We assume a thread terminates on its own, once its function returns. At any given time, a team can have at most a single thread running for it (either for determining its role, or tightening its bounds).

Once the earliest lend and latest borrow times of teams are established, CANMATCH (Algorithm 4) determines if a decoupled plan is found (or finds out that none exists).

4.3.1.3 The Improved Algorithm

An immediate improvement of Algorithm 1 would be to check for a solution (or the nonexistence of one), after some team's summary is updated and thus the central agent has better information about the teams' capabilities. We utilize the functions CANDEFINITELYMATCH (Algorithm 6) and CANPOSSIBLYMATCH (Algorithm 7) to check whether the current summaries are sufficient to find a valid matching, and to check whether there is

Algorithm 6 CANDEFINITELYMATCH

Input: plan length k; transportation delay d; an array of benched robots $bench$, with the number of steps remaining before they can be given to a team; summaries $S_1, S_2, \ldots, S_n$ for all teams t with $S_t = \langle role, l, u \rangle$

Output: success if we can be sure that a matching exists with the current roles and bounds of the teams, failure otherwise

```
if ∃t s.t. S_t.l = 0 then // it may not be able to find a plan with only 1 borrowed robot
    return false;
if ∃t s.t. S_t.role = none then // it can be a borrower with S_t.l = 0
    return false;

L, B ← empty sets for lend and borrow times;
L ← the values from bench;

for all teams t do
    if S_t.role = lender then
        L := L ∪ {S_t.u + d};
    if S_t.role = borrower then
        B := B ∪ {S_t.l};

for i = 1, 2, ..., |B| do
    borrow ← i^th lowest value in B;
    lend ← i^th lowest value in L;
    if lend ≥ borrow then
        return false; // a borrowing teams needs can not be met;
return true; // all borrowers are matched with a lent robot
```

still hope for finding a decoupled plan. By a valid matching, we mean that each borrowing team is assigned to a lending team or a spare robot, and the lend times are sufficient to provide the robots before the borrow times (taking the transportation delay into account). CANDEFINITELYMATCH and CANPOSSIBLYMATCH are slightly modified versions of CANMATCH.

CANDEFINITELYMATCH interprets the summaries in the most pessimistic way possible: if a borrowing team's lower bound is still at its starting value of 0, then it is assumed that borrowing a robot (at any step) does not help that team finish its task; if a team's role is not yet determined, it is assumed to be a borrower (with a lower bound of 0); it works by trying to match the lower bounds of borrowing teams with the upper bounds of lending teams (i.e. their current verified lend/borrow times).

Conversely, CANPOSSIBLYMATCH interprets the summaries in the most optimistic manner: Any team with a not yet determined role is considered the best possible lender (i.e. can lend by step 1), and the team's best possible lend/borrow times are compared (which are $S_t.l + 1$ for lending teams and $S_t.u - 1$ for borrowing teams). Clearly, for any group of summaries for which CANMATCH returns success, there is a decoupled plan. For any group of summaries that CANPOSSIBLYMATCH returns failure, there can be no decoupled plan. Also observe that, when the summaries are fully informative, both functions return the same answer.

Algorithm 7 CANPOSSIBLYMATCH

Input: plan length k; transportation delay d; an array of benched robots $bench$, with the number of steps remaining before they can be given to a team; summaries $S_1, S_2, \ldots, S_n$ for all teams t with $S_t = \langle role, l, u \rangle$

Output: success if there is still hope for a solution if the bounds of the teams are iterated further, failure otherwise

```
L, B ← empty sets for lend and borrow times;
L ← the values from bench;

for all teams t do
    if S_t.role = lender then
        L := L ∪ {S_t.l + 1 + d};
    if S_t.role = none then // assume it to be the best possible lender
        L := L ∪ {1 + d};
    if S_t.role = borrower then
        B := B ∪ {S_t.u − 1};

for i = 1, 2, ..., |B| do
    borrow ← i^th lowest value in B;
    lend ← i^th lowest value in L;
    if lend ≥ borrow then
        return false; // a borrowing teams needs can not be met
return true; // all borrowers are matched with a lent robot
```

The improved FINDDECOUPLEDPLAN can be found in Algorithm 5. After asking for all the teams to determine their roles, it goes in a loop until CANDEFINITELYMATCH returns true with the current summaries or CANPOSSIBLYMATCH returns false. In each iteration, it first waits for a team to answer a query and update its summary, and when a team does, FINDDECOUPLEDPLAN calls CANDEFINITELYMATCH and CANPOSSIBLY-MATCH to see if the new information leads to a definite decision. If not, the team is asked to tighten its bounds.

Notice that FINDDECOUPLEDPLAN returns either success or failure, and, in the case of success, the summaries of teams (not the actual plans themselves). After a (minimum) decoupled plan is found, a team can be asked to find an executable plan, without redundant actions, where the team lends (borrows) a robot before (after) a mid step, as specified in its summary.

Proposition 1. FINDDECOUPLEDPLAN *(Algorithm 5) is sound and complete. It always terminates with at most* $O(nlogk)$ *queries to* CCALC*, where* k *is the length of the decoupled plan and* n *is the number of teams.*

Proof. Termination FINDDECOUPLEDPLAN always terminates, either by CANDEFINITELYMATCH returning true or CANPOSSIBLYMATCH returning false on the current summaries. Note that once a team's role is determined, TIGHTENBOUNDS is iteratively called for a team, until its the lower and upper bounds on its earliest lend / latest borrow time are tightened down to a difference of 1. Our claim is that, either FINDDECOUPLEDPLAN

terminates before all the teams' lower and upper bounds are tight (which we call the most informative state), or it eventually reaches the point where the bounds are tight, at which point it definitely terminates.

First, let's show that FINDDECOUPLEDPLAN reaches the point where the bounds are tight (if it does not terminate early). While a team t's bounds $(S_t.l, S_t.u)$ are not tight $(S_t.l + 1 < S_t.u)$, it is asked to tighten its bounds, with a call to TIGHTENBOUNDS which asks a single query to CCALC, which always returns an answer. It takes a finite number of calls to TIGHTENBOUNDS to determine a team's tightest bounds (for plan length k, $\lceil logk \rceil$ to be exact), and since TIGHTENBOUNDS always terminates, FINDDECOUPLED-PLAN eventually reaches the point, where for all teams t, $S_t.l + 1 = S_t.u$ (if it does not terminate early).

Our claim is that, upon reaching this point, FINDDECOUPLEDPLAN terminates. Observe that both CANDEFINITELYMATCH and CANPOSSIBLYMATCH have two stages: they first collect lend/borrow times from the summaries of teams and then compare these lend/borrow times to check for a matching, and the comparison method they use are the same. While collecting lend/borrow times, for a lending team t, CANDEFINITELYMATCH considers $S_t.u$ as the lend time, while CANPOSSIBLYMATCH considers $S_t.l + 1$. Note that these values are the same if team t's bounds are at their tightest (i.e., $S_t.l + 1 = S_t.u$). A similar case can be made for borrowing times as well where the collected borrow time from team t is either $S_t.l$ or $S_t.u - 1$, which are the same when the bounds are at their tightest. Once the collected lend/borrow times are the same, CANDEFINITELY-MATCH and CANPOSSIBLYMATCH both return the same answer (true or false). And since FINDDECOUPLEDPLAN requires CANDEFINITELYMATCH to return true or CAN-POSSIBLYMATCH to return false in order to terminate, it definitely terminates when both return the same answer.

Soundness We need to show that if FINDDECOUPLEDPLAN returns true for plan length k, than that means there is a decoupled plan of length k. Note that, for FINDDE-COUPLEDPLAN to return true, CANDEFINITELYMATCH has to return true for the current summaries of teams. Let's observe these summaries: for each team t, its summary S_t contains its role $S_t.role$, and lower and upper bounds $S_t.l$, $S_t.u$ for its earliest lend or latest borrow time. For a lending team, if $S_t.u < k$, this means that the team can lend a robot before $S_t.u$ and still complete its own task in k steps and has actually successfully found a plan for such a case, using CCALC. Similarly, for a borrowing team, if $S_t.l > 0$, this means that the team can borrow a robot after $S_t.l$ and complete its own task in k steps. We claim that the overall plan, in which each lending team i executes its plan of length k where a robot is lent before $S_i.u$ and each borrowing j team executes its plan of length k where a robot is borrowed after $S_j.l$, is a valid overall plan (i.e., all the teams complete their tasks in k steps). This is fairly easy to show:

- Each lending team i is clearly able to complete its plan, since it does not need the

help of any other team and can complete its task on its own. Even if $S_i.u = k$ it still has a plan (where it does not lend a robot), since being self sufficient is the definition of a lending team.

- Each borrowing team j has a plan where it completes its task with a robot borrowed before step $S_j.l$. The potential problem here is that a borrowing team is actually dependent on a lending team and requires it to be able to lend a robot before $S_j.l$ (minus the transportation delay d). However, since CANDEFINITELYMATCH actually matches each borrowing team j with a lending team i s.t. $S_j.l + d < S_i.u$, we know each borrowing team is lent a robot on time and can also complete its plan as well.

Completeness We need to show that if there is an overall plan P of length k, then FINDDECOUPLEDPLAN returns true (for length k); specifically, we need to show: 1) Every call of CANPOSSIBLYMATCH in FINDDECOUPLEDPLAN returns true, given that there is an overall plan of length k; 2) If every call of CANPOSSIBLYMATCH in FINDDECOUPLEDPLAN returns true, then FINDDECOUPLEDPLAN returns true.

1) We prove our first claim observing some inequalities, over the summaries of teams, and thus, over the lend/borrow times CANPOSSIBLYMATCH tries to match. Take any team t. In a plan P of length k, team t either lends a robot at step $l_t < k$, borrows a robot at step $b_t < k$, or is not involved in any robot exchange. Let's explore these options and what they imply for our summaries and lend/borrow times CANPOSSIBLYMATCH considers:

- If t is a lender, with lend time l_t, then FINDDECOUPLEDPLAN definitely labels it as a lender. We know that CCALC never fails to find a plan if one exists, and since we also know that the team is able to lend a robot by step l_t, CCALC always finds a plan where a robot is lent before step k' where $k' > l_t$. Simply put, the lower bound on the earliest lend time is always smaller than l_t ($S_t.l < l_t$). For a lending team t, CANPOSSIBLYMATCH, considers $l'_t = S_t.l + 1$ as its lend time. Since $S_t.l < l_t$, we can say $S_t.l + 1 \leq l_t$, and thus, $l'_t \leq l_t$.

- If t is a borrower, with borrow time b_t, then it can be labeled as either a lender, or a borrower. If it is labeled as a borrower, then the upper bound on the latest borrow time is always above b_t (using arguments similar to the lending case, $S_t.u > b_t$). For a borrowing team t, CANPOSSIBLYMATCH, considers $b'_t = S_t.u - 1$ as its borrow time. Since $S_t.u > b_t$, we can say $S_t.u - 1 \geq b_t$, and thus, $b'_t \geq b_t$.

- If t does not exchange robots, then it means it can perform its task on its own, and is labeled as a lender.

Let's return to our original claim, that given there is a plan P of length k, every call of CANPOSSIBLYMATCH in FINDDECOUPLEDPLAN returns true. After collecting

lend/borrow times, CANPOSSIBLYMATCH tries to match them, and if it can't, it returns false. We now show that there is always a possible matching, where each borrower i is matched to a lender j, where j lends a robot to i in the plan P.

Since each team FINDDECOUPLEDPLAN labels as a borrower, can only be a borrower in plan P, and each lender in plan P is labeled as a lender again, we know that we can uniquely match each borrower with the lender that lends it a robot in plan P. Pick such a lender/borrower pair with lender i with a lend time of l_i in P and borrower j with a borrow time b_j in P. Since i is able to lend a robot to j in plan P, we know that $l_i + d < b_j$ must hold. Using the inequalities from the observations, we also know that for i, the lend time l'_i CANPOSSIBLYMATCH considers satisfies $l'_i \leq l_i$ and for j, the borrow time b'_j CANPOSSIBLYMATCH considers satisfies $b'_j \geq b_j$. Putting it all together, we obtain $l'_i + d \leq l_i + d < b_j \leq b'_j$ and thus $l'_i + d < b'_j$. This means, in any call to CANPOSSIBLYMATCH, the matching in plan P is also a matching in CANPOSSIBLYMATCH (with lend/borrow time constraints satisfied between the pairs), and thus, CANPOSSIBLYMATCH always returns true (given that there is such a plan P).

2) If every call of CANPOSSIBLYMATCH in FINDDECOUPLEDPLAN returns true, then FINDDECOUPLEDPLAN can never return false, and since we know it always terminates, it has to return true. So, given that there is a plan, FINDDECOUPLEDPLAN returns true.

Complexity For a plan of length k, we can determine a team's role in a single query and determine its tightest bounds with an additional $\lceil log_2 k \rceil$ queries. For n teams this means a total of $n \times \lceil log_2 k + 1 \rceil$ queries. Note that this is the worst case behaviour and FINDDECOUPLEDPLAN may terminate before all the bounds are tightened. Also note that, due to parallelism, the time to find a decoupled plan is at most as much as the time it takes for any team to answer their respective queries (at most $\lceil log_2 k + 1 \rceil$ queries for a single team), plus the negligible amount of time spent to compare the summaries for possible matchings. □

4.3.2 Finding Minimum Length Decoupled Plans

In this subsection, we show how we utilize the FINDDECOUPLEDPLAN algorithm described earlier, to find minimum length decoupled plans. FINDDECOUPLEDPLAN provides us a tool to check if there is a plan for a given length k. We can use FINDDECOUPLEDPLAN to perform a binary search on the plan length, to find the minimum plan length k^*.

The search has two phases as shown in Algorithm 8: first an upper bound on k^* is established by trying to find decoupled plans of length $k, 2k, 4k \ldots$ until one is found, that is of length $2^m k$, where $k \in \mathbb{Z}^+$ is a parameter of FINDMINIMUMDECOUPLEDPLAN. Once the upper bound on k^* is established, a binary search is performed between $2^{m-1}k$

Algorithm 8 FINDMINIMUMDECOUPLEDPLAN

Input: team representatives $R_1, R_2, \ldots, R_n$, transportation delay d; an array of benched robots $bench$, with the number of steps remaining before they can be given to a team, plan length k to begin the search with

Output: a minimum length decoupled plan

```
// team summaries S_1, S_2, ..., S_n;
l := 0; u := 0; // the lengths of the last failed and successful decoupled plan attempts
// Establish an upper bound on plan length
while u = 0 do
    outcome, S'_1, S'_2, ..., S'_n ← FINDDECOUPLEDPLAN(k, R_1, R_2, ..., R_n, d, bench);
    if (outcome = success) then
        u := k;
        for all teams t do
            S_t := S'_t;
    else
        l := k;
        k := 2k;

// find a minimum length plan
while u > l + 1 do
    k := ⌊(u + l)/2⌋;
    outcome, S'_1, S'_2, ..., S'_n ← FINDDECOUPLEDPLAN(k, R_1, R_2, ..., R_n, d, bench);
    if (outcome = success) then
        u := k;
        for all teams t do
            S_t := S'_t;
    else
        l := k;
return PLAN(S_1, S_2, ..., S_n, R_1, R_2, ..., R_n);
```

and $2^m k$ to find the value of k^*. Notice that, for the successful decoupled plans we store the team summaries; and when the search terminates, we have the team summaries for the minimum length plan. FINDMINIMUMDECOUPLEDPLAN returns a decoupled plan, associated with these summaries by calling the PLAN function which asks all the teams to find good quality plans (with redundancy elimination, as mentioned in Subsection 4.2.3), ready for execution.

Proposition 2. FINDMINIMUMDECOUPLEDPLAN *is sound and complete. It always terminates (assuming there is a plan to be found) with at most* $O(n(logk^*)^2)$ *queries to* CCALC*, where* k^* *is the length of the optimal decoupled plan and* n *is the number of teams.*

Proof. Soundness and Completeness FINDMINIMUMDECOUPLEDPLAN performs a binary search on the optimal plan length and asks FINDDECOUPLEDPLAN if there is a decoupled plan of a fixed length, at each step of the binary search. Since FINDDECOUPLEDPLAN is both sound and complete (Proposition 1), FINDMINIMUMDECOUPLEDPLAN is also sound and complete, and the found plan is of minimum length.

Termination We know that FINDDECOUPLEDPLAN always terminates and we assume

there is a decoupled plan to be found of length k^*, therefore an upperbound is eventually established after at most $\lceil log_2 k^* \rceil$ calls to FINDDECOUPLEDPLAN. After its bounds are established, the binary search terminates in a finite number of calls to FINDDECOUPLED-PLAN.

Complexity Assuming we start the first stage of the search with plan length 1, it takes exactly m calls to FINDDECOUPLEDPLAN to establish the upperbound on k^* where m is the smallest integer with $log_2 k + 1 \leq m$. When the second stage starts, the bounds for the binary search are 2^{m-1} and 2^m. It takes $m - 1$ calls to FINDDECOUPLEDPLAN before we know we have a minimum length plan. FINDDECOUPLEDPLAN makes $O(nlog2^m) = O(nm)$ planning calls each time it is called, so the total number of planning calls is $O(nm^2)$ which is $O(n(logk^*)^2)$. □

4.3.3 Inferring Bounds from Previous Searches

In FINDMINIMUMDECOUPLEDPLAN described above, the subsequent calls to FINDDE-COUPLEDPLAN are done without any information from the previous FINDDECOUPLED-PLAN calls. However, there are some simple inferences that can be made about a teams role and bounds for a plan of length k, using its roles and bounds from plans of length $l < k$ and $u > k$:

Proposition 3. *A team is labeled as a lender for plan length k, if it was labeled as a lender for plan length $u > k$. Conversely, a team is labeled as a borrower for plan length k, if it was labeled as a borrower for plan length $l < k$.*

Proof. A team being labeled as a lender (resp. borrower) is only based on its ability to complete its work on its own. If it can (resp. can not) complete its task in k steps, then it certainly can (resp. can not) complete its task in $u > k$ (resp. $l < k$) steps. □

Proposition 4. *A team can lend a robot before step k' in a plan of length k, if it can lend a robot before step k' in a plan of length $l < k$. Conversely, a team can not lend a robot before step k' in a plan of length k, if it can not lend a robot before step k' in a plan of length $u > k$.*

Proof. Given that a team can lend a robot at step k' in a plan of length l, we can simply perform that plan and then do nothing for the remaining $k - l$ steps, to achieve a plan of length k; thus the first statement holds. The second statement is the contrapositive of the first one, therefore it also holds. □

Proposition 5. *A team can borrow a robot after step k' in a plan of length k, if it can borrow a robot after step $l' = k' - a$ in a plan of length $l = k - a$, where $a \in \{1, 2, \dots k' - 1\}$. Conversely, a team can not borrow a robot after step k' in a plan of length k, if it can not borrow a robot after step $u' = k' + a$ in a plan of length $u = k + a$ where $a \in \{0, 1, \dots\}$.*

Algorithm 9 FINDMINIMUMDECOUPLEDPLAN (updated for role and bound inferrence)

Input: team representatives $R_1, R_2, \ldots, R_n$, transportation delay d; an array of benched robots $bench$, with the number of steps remaining before they can be given to a team, plan length k to begin the search with

Output: a minimum length decoupled plan

```
Create uninitialized team summaries S_1, S_2, ..., S_n ; S_1^L, S_2^L, ..., S_n^L; S_1^U, S_2^U, ..., S_n^U;
l := 0; u := 0; // the lengths of the last failed and successful decoupled plan attempts
// Establish an upper bound on plan length
while u = 0 do
    outcome, S_1, S_2, ..., S_n ← FINDDECOUPLEDPLAN(k, R_1, R_2, ..., R_n, d, bench ,
S_1^L, S_2^L, ..., S_n^L, S_1^U, S_2^U, ..., S_n^U);
    if (outcome = success) then
        u := k;
        for all teams t do
            S_t^U := S_t;
    else
        l := k;
        for all teams t do
            S_t^L := S_t;
        k := 2k;

// find a minimum length plan
while u > l + 1 do
    k := ⌊(u + l)/2⌋;
    outcome, S_1, S_2, ..., S_n ← FINDDECOUPLEDPLAN(k, R_1, R_2, ..., R_n, d, bench ,
S_1^L, S_2^L, ..., S_n^L, S_1^U, S_2^U, ..., S_n^U);
    if (outcome = success) then
        u := k;
        for all teams t do
            S_t^U := S_t;
    else
        l := k;
        for all teams t do
            S_t^L := S_t;
return PLAN(S_1^U, S_2^U, ..., S_n^U, R_1, R_2, ..., R_n);
```

Proof. Given that a team can borrow a robot at step $l' = k' - a$ in a plan of length $l = k + a$, then we can simply do nothing for a steps and then perform the plan to achieve a plan of length k where a robot is borrowed after step k'; thus the first statement holds. The second statement is the contrapositive of the first one, therefore it also holds. □

We can use these results to quicken subsequent decoupled plan attempts by inferring roles and bounds of teams, at the beginning of trying to find a decoupled plan, for that we introduce two new algorithms: INFERROLE (Algorithm 11) and INFERBOUNDS (Algorithm 12). We also present the updated versions of FINDDECOUPLEDPLAN and FINDMINIMUMDECOUPLEDPLAN in Algorithm 10 and Algorithm 9.

With the updates, while we are trying to find a decoupled plan, we first try to infer the role of the team, using the summaries from previous attempts. If we can not, then we call DETERMINEROLE as usual. Also, before calling TIGHTENBOUNDS for a team for the

Algorithm 10 FINDDECOUPLEDPLAN (updated for role and bound inferrence)

Input: plan length k; team representatives $R_1, R_2, \ldots, R_n$, transportation delay d; an array of benched robots $bench$, with the number of steps remaining before they can be given to a team; plan lengths l, u and summaries $S^L = \{S_1^L, S_2^L, \ldots, S_n^L, \}, S^U = \{S_1^U, S_2^U, \ldots, S_n^U, \}$ for the last failed and successful decoupled plan attempts respectively
Output: success if a decoupled plan can be found, failure otherwise

// summaries $S_1, S_2, \ldots, S_n$ for each team, where $S_t = \langle role, l, u \rangle$;

for all teams t **do**
  $S_t \leftarrow \langle none, 0, k+1 \rangle$; *// set role, bounds*
  $S_t.role \leftarrow$ INFERROLE($S_t^L.role, S_t^U.role$);
  if $S_t.role = none$ **then**
    call DETERMINEROLE(k, S_t, R_t) as a separate thread;
loop
  wait for a team's summary to be updated;
  if CANDEFINITELYMATCH($k, d, bench, S_1, S_2, \ldots, S_k$); **then**
    return success, $S_1, S_2, \ldots, S_n$;
  if $\neg$CANPOSSIBLYMATCH(k, d, $bench, S_1, S_2, \ldots, S_k$); **then**
    return failure, $S_1, S_2, \ldots, S_n$;

  for all t with $\neg R_t.isPlanning$ **and** $S_t.l + 1 < S_t.u$ **do**
    if INFERBOUNDS has not been called for team t yet **then**
      INFERBOUNDS ($k, l, u, S_t, S_t^L, S_t^U$);
    call TIGHTENBOUNDS(k, S_t, R_t) as a separate thread;

Algorithm 11 INFERROLE

Input: Roles r^L, r^U of a team from decoupled plan attempts of shorter and longer lengths
Output: Role of the team, if one can be inferred, $none$ otherwise

if $r^L = lender$ **then**
  return $lender$;
else if $r^U = borrower$ **then**
  return $borrower$;
else *// bound can not be inferred*
  return $none$;

first time, we first call INFERBOUNDS once. The summaries of both failed and successful decoupled plan attempts are stored, but we only store the latest ones (whose plan lengths are the tightest bounds we have on the minimum plan length at the moment). This does not result in a loss of information since all the inferences are transitive.

As an example, suppose the optimal plan length is 20 and we have found successful plans for length 32 where team t is labeled as a borrower, with an upper bound of 25 (it can not borrow after step 25); and now we are trying to find a decoupled plan of length 24. We can infer that the team t will again be a borrower, with an upper bound of 17 (25+24-32). Now suppose we have tightened the upper bound of team t down to 12 and are trying to find a decoupled plan of length 20: if we use the summary from the plan with length 32, we infer that the team is a borrower with an upper bound of 13 (25+20-32), if we instead use the summary from the plan with length 24, we end up with an upper

Algorithm 12 INFERBOUNDS

Input: Plan lengths k, l, u and summaries S_t, S_t^L, S_t^U for team t - for the current, last failed and last successful decoupled plan attempts respectively
Output: Summary S_t for team t with potentially updated bounds

if $S_t.role = lender$ **then**
  if $S_t^L.role = lender$ **and** $S_t^L.u < l$ **then**
    $S_t.u$ = $S_t^L.u$;
  if $S_t^U.role = lender$ **and** $S_t^U.l > 0$ **then**
    $S_t.l$ = $S_t^U.l$;
if $S_t.role = borrower$ **then**
  if $S_t^L.role = borrower$ **and** $S_t^L.l > 0$ **then**
    $S_t.l$ = $S_t^L.l + k - l$;
  if $S_t^U.role = borrower$ **and** $S_t^U.u < u$ **then**
    $S_t.u$ = $S_t^U.u$ + k - u;
if $S_t.l < 0$ **then**
  $S_t.l := 0$;
if $S_t.u > k$ **then**
  $S_t.u := k$;
return S_t;

bound of 8 (12 + 20 - 24), which is tighter. Also note that if the upper bound had not been tightened, we could have still inferred the 13 (17+20-24).

Even though these kind of inferences speed up the process in practice, we were not able to show the existence of a tighter asymptotical upper bound, than the one shown in Proposition 2, with the added inference rules.

4.4 Embedding Decoupled Planning in an Execution and Monitoring Framework

We have embedded our optimal decoupled plan algorithm in an execution and monitoring framework. The basic idea is, while the tasks of all teams are not completed, we find a optimal decoupled plan and start to execute it, step by step. If at any step, something goes wrong (i.e., a carrier tries to attach and fails) or we receive an order for more boxes, we start again (i.e., replan and execute). Note that, each time we call FINDMINIMUMDECOUPLEDPLAN, we assume that no robot exchange had taken place (i.e., a team can lend a robot, then after replanning lend another robot, or borrow a robot).

For the subsequent replans for an optimal length decoupled plan, we can reuse the information from the previously computed plan. Notice that, when a teams state is changed, the rest of the previously computed plan becomes non-executable and we have to find a new plan for that team. However, for a team that has the desired state, the rest of its plan is still applicable. Using this observation, when we call FINDMINIMUMDECOUPLEDPLAN, instead of starting the search by looking for a decoupled plan of length 1, we can start by looking for a decoupled plan whose length is equal to the remaining length of

the plan we were executing. This way, while for the teams that changed their states we initialize them from scratch (as usual), we can initialize the rest of the teams with their roles and bounds with the following ideas:

- A lender which has lent a robot is still a lender, since it can complete its task on its own, with its remaining robots. However, its bounds are reset to zero and the remaining plan length respectively.
- A borrower which has borrowed a robot becomes a lender, because with the borrowed robot, it can complete the rest of the plan by itself. Its bounds are reset as well.
- If a lender has not lend a robot or a borrower has not borrowed a robot, they retain their roles.
- If a team is a lender for plan length k with bounds l, u, and we execute a steps without lending a robot, if we start to replan for plan length $k - a$, its bounds become $max(0, l - a), u - a$. Same thing applies for a borrower that has not yet borrowed a robot.

Let's give an example with four teams: Team 1 has 1 worker and has to paint 4 boxes, Team 2 has 3 workers and has to paint 5 boxes, Team 3 has 2 workers and has to paint 4 boxes, and Team 4 has 1 worker and has to paint 3 boxes. The transportation delay t_d is set to 2.

In the first part of FINDMINIMUMDECOUPLEDPLAN, FINDDECOUPLEDPLAN finds an overall plan of length $k = 32$ (after trying $k = 1, 2, 4, 8, 16$). According to this plan, Team 1 receives a robot at Step 16 and Team 2 lends a robot at Step 7. The second part of FINDMINIMUMDECOUPLEDPLAN starts a binary search with $l = 16$ and $u = 32$. After trying $k = 24, 28, 26, 27$, an optimal plan of length 27 is found. According to this plan, Teams 2 and 3 can lend robots at Steps 10 and 12 respectively; Teams 1 and 4 can borrow robots at Steps 13 and 18 respectively. Notice that lend/borrow matchings of Team 2 (10) with Team 1 (13), and Team 3 (12) with Team 4 (18) are valid with respect to the transportation delay. Table 4.4 shows some parts of these plans. In this example, the largest problem CCALC solves has 20578 atoms and 186578 clauses. The average time to answer a query is 11 CPU seconds (on a workstation with two 1.60GHz Intel Xeon E5310 Quad-Core Processor and 16 GB RAM, running Centos 64bit (Version 5.3)).

4.5 Related Work

Most of the related work on multi-agent systems consider multiple agents, each agent capable of reasoning, working in the same environment. In comparison, our work focuses

Table 4.4: List of actions for all teams.

	Team1	Team2	Team3	Team4
0	move(w1,right) lineShift	move(w2,right) move(w3,right) move(c1,right) lineShift swapEndEffector(w1,1)	move(w1,right) move(w2,right) move(c1,right) lineShift	move(w1,right) lineShift
⋮				
10	move(w1,left)	lineShift giveRobot(w2) attach(c1,w3)	workOn(w1,1) workOn(w2,3)	swapEndEffector(w1,2)
11	workOn(w1,1)	workOn(w1,5) workOn(w3,3)	move(w1,left) move(w2,left)	move(w1,right)
12	move(w1,left)	move(w1,left) pull(c1)	workOn(w2,2) giveRobot(w1)	workOn(w1,1)
13	move(c1,right) takeRobot(w2,1,2) swapEndEffector(w1,2)	move(w1,left)	move(w2,left)	move(w1,left)
⋮				
26	lineShift	lineShift	lineShift	lineShift

on multiple teams of robot, each team with a single cognitive agent, working in separate environments. A parallel may be drawn between the two approaches if we view the problem we study in this thesis as multiple agents (as opposed to multiple teams) working in separate environments but still dependent on each other.

In general, there are three major types of methods used in decoupled planning to coordinate the actions of agents [42], and these types of methods can be used in conjunction with each other:

- **Coordination before planning:** These type of methods coordinate the agents before they even begin to plan, by introducing *social laws*, which the agents must follow. These laws restirct the agents in their behavior and can be used to reduce planning and coordination time. A good example for social laws might be the traffic rules. If everyone drives on the right side of the road, no coordination with oncoming cars will be required.

  Shoham and Tennenholtz, study how social laws can be created in a multi-agent system [53]. Briggs proposes the idea of flexible laws [10], where agents first try to find plans using the strictest laws but if a solution cannot be found agents are allowed to use more relaxed laws. ter Mors et al., describe a preplanning coordination method that adds a minimal set of additional constraints to the subgoals to be performed, in order to ensure a coordinated solution by independent planning [56].

- **Coordination during planning:** In these type of methods, agents find plans for themselves while sharing their plan information and adapting their plans to avoid conflicts.

  One approach like this is the Partial Global Planning (PGP) framework [18], and its extension, Generalized PGP [16, 17]. In this approach, agents share their plans using a specialized plan representation. Coordination is achieved as follows: if an agent informs a second agent of its own plan, the second agent merges this

information into its own partial global plan. Second agent then tries to improve the global plan. If it can, the improved plan is shown to the other agents who can accept/reject/modify it. An overview of PGP related approaches is given by [40].

- **Coordination after planning:** These type of methods use plan merging. Given the individual plans of all agents, plan merging constructs a joint plan for all agents. Georgeff was one of the first to propose a plan-synchronization process starting with individual plans [31, 32]. Stuart uses propositional temporal logic to guarantee that only feasible states of the environment can be reached (it can be seen as semaphores which guarantee that no event fails) [54]. Introducing restrictions on individual plans (as in coordination before planning) can be used to ensure efficient merging [62, 29].

  Another approach to merging plans is to use the search method A^* and a smart cost-based heuristic [22]. Ephrati and Rosenschein showed that dividing the work of constructing sub plans over several agents reduces the overall complexity of the merging algorithm [23].

In light of these related work, we can consider our method to be a coordination during planning method since our teams keep replanning until a final overall plan is found. A major difference is that instead of teams communicating with each other, we have a central agent which communicates with all the teams. Our assumptions for robot exchanges (can not lend/borrow more than one robot, can not lend and borrow, etc.) may be seen as social laws, however their purpose is not to avoid conflicts between teams but to simplify the problem and allow us to efficiently coordinate the teams.

4.6 Summary of Contributions

We have developed a novel algorithm for finding optimal decoupled plans for problems that involve multiple teams working in separate workspaces, that are allowed to lend or borrow at most one robot. We provided detailed termination, soundness, completeness and complexity analysis of the algorithm. We have implemented our algorithm in the Robot Operating System (ROS), so that it can be used with many different kinds of robots.

We have introduced a new problem, a Cognitive Painting Factory, and modeled it in the action description language $\mathcal{C}+$. We have shown methods to decrease the number of redundant actions performed in plans found by CCALC.

We embedded our optimal decoupled planning algorithm in an execution and monitoring framework and showed its applicability on the Cognitive Painting Factory domain.

Chapter 5

Conclusion

Let us summarize the contributions of this thesis and the ongoing and future work in two parts.

5.1 Genome Rearrangement

Extending our earlier work [59], we introduced a new computational method, based on AI planning, to solve genome rearrangement problems with duplicate genes, involving transpositions, inversions, inverted transpositions, insertions, and deletions. There are some methods [21], [15], [39], [63] and tools (e.g., GRIMM [57], GRAPPA [46], DERANGE 2 [7], MGR [9]) to solve restricted versions of these problems, e.g., by considering inversions only or by relabeling duplicates uniquely; however, none of these genome rearrangement software can handle such general genome rearrangement problems. There is another system, TD-ESTIMATOR [41], which can handle transpositions and duplicates; however, rather than solving the genome rearrangement problem, it approximates the distance between two genomes in terms of the Double-Cut-Join operation, and gene losses and duplications.

Based on our AI planning approach to genome rearrangement, we implemented a genome rearrangement software system called GENOMEPLAN, which describes the genome rearrangement problems discussed above as planning problems and use the planner TLPLAN to compute solutions. GENOMEPLAN can solve variations of genome rearrangement where we specify costs and priorities of events by functions. Being able to represent and modify genome rearrangement problems in a high-level formalism, and to choose the search strategy and settings to solve the problem by utilizing the facilities of TLPLAN, allows us a flexible tool (GENOMEPLAN) to analyze and better understand evolutionary history of species. In this way, GENOMEPLAN provides an alternative tool to solving genome rearrangement problems.

We showed the applicability and the effectiveness of GENOMEPLAN on three real

data sets: chloroplast genomes of various land plants and green algae [15], *Metazoan* mitochondrial genomes [8], and *Campanulaceae* chloroplast genomes [13]. Only in the first data set, genomes are of unequal content and with duplicate genes. We observed that the results found by GENOMEPLAN are similar to those widely accepted. As for the computation time, the first data set is evaluated in 67 minutes; note that this data set is evaluated in almost 25 days in the work by Cui et al. [15]. The second data set is evaluated in a minute, whereas the third data set is evaluated in 10 minutes. We also illustrated the usefulness of specifying costs and priorities of events in understanding the evolutionary history of *Metazoan* mitochondrial genomes. We observed that increasing the priority of inversions in the problem specification improved the accuracy of solutions.

We showed the effectiveness of our approach in handling duplicates, with some experiments over randomly generated problem instances. In particular, we compared our approach with the naive approach of relabeling duplicates uniquely and then using an existing genome rearrangement software system to solve the problems. We observed that, compared to the naive approach of relabeling genes uniquely (using DERANGE 2), GENOMEPLAN finds more parsimonious solutions.

Last but not the least, we compared our approach to solving genome rearrangement problems where the goal is to find smaller cost solutions, with Cui et al.'s approach where the goal is to find solutions that estimate the true evolutionary distance better. We experimented with a set of randomly generated problem instances with duplicates and with equal gene content, using GENOMEPLAN and TD-ESTIMATOR. In order to establish a fair comparison, we compared them in terms of the standard deviation of the computed distances from the true evolutionary distances, after normalizing the costs of solutions to the actual costs. We observed that, when the genome length is fixed but the number of events is varied, the normalized standard deviation increases as the number of events increase; these error values are lower for GENOMEPLAN, making it more advantageous over TD-ESTIMATOR. When the number of events is fixed but the genome length is varied, the normalized standard deviation decreases as the genome size increases; in shorter genomes (resp. longer genomes), the normalized standard deviation of the solutions computed by GENOMEPLAN (resp. TD-ESTIMATOR) is lower.

The results of our work are summarized in a conference paper [59], and a journal article [58].

Future work As part of our onging work, we have devised a greedy search algorithm for genome rearrangement, using the search strategies and heuristic we learned while developing GENOMEPLAN, which significantly outperforms GENOMEPLAN in terms of planning time and matches GENOMEPLAN in terms of solution quality. It allows the specification of costs of actions, but does not provide the other kinds of domain information specification offered by GENOMEPLAN. We have also extended our greedy search algorithm to a lookahead search. In preliminary experiments, we observed that

with lookahead 2, we obtain better results and even with lookahead 3, the search method performs significantly faster than GENOMEPLAN. We feel that the results obtained by the search algorithm is promising, and will focus our attention to developing it further.

5.2 Multi-Robot Systems

We have developed a novel algorithm for finding optimal decoupled plans for problems that involve multiple teams working in separate workspaces, that are allowed to lend or borrow at most one robot. We provided detailed termination, soundness, completeness and complexity analysis of the algorithm. We have implemented the algorithm in the Robot Operating System (ROS). We have shown its applicability on a Cognitive Painting Factory scenario. For that we modeled the workspaces in the action description language $\mathcal{C}+$. We have improved the representation to decrease the number of redundant actions in plans found by CCALC. We have embedded our optimal decoupled planning algorithm in an execution and monitoring framework and showed its applicability by simulations with the Cognitive Painting Factory scenario. Our work are summarized in a conference paper [26].

Future work Our work on decoupled planning is an initial study and has room for improvement.

The tightening of bounds in the decoupled plan is done naively (i.e., each team tightens its bounds separately until an overall solution is found). A more goal oriented bound tightening method can be developed. Consider, for instance, a case with only two teams, a lender and a borrower, and we are trying to find a decoupled plan of length 30. Suppose the current bounds of the lender are 19 and 20 respectively (i.e., it can give a robot by step 20, at the earliest) and the bounds of the borrower are not initialized yet (i.e., 0 and 30). Our current algorithm would first try to check if the team can borrow a robot by after 15, however checking if the team can borrow after step 20 (assuming there is no transportation delay) has two advantages: 1) If it turns out that the team can borrow after step 20, then there is a decoupled plan. In comparison, if it turns out that the team can borrow after step 15, then that does not guarantee a decoupled plan and further queries are required. 2) The query for a borrow time of 20 is more likely to fail than a query for a borrow time of 15. This way, if there is no decoupled plan, the non-existence of it is established more quickly.

We are working on a method where a bipartite graph (one part for lenders and extra robots, other for borrowers) is maintained throughout the search for a decoupled plan. There is an edge between a borrower and a lender (or a spare robot), if and only if there can be a valid matching where the teams connected by the edge match. This helps keeps track of which teams should consider which teams while tightening their bounds. The method is not fully developed and tested yet, but we believe it will increase the runtime

performance even further. The missing parts are mainly about how to choose which value to test for a team t, given the other teams that t can be matched with and their bounds. Committing to some strategies and deviating from the middle point has the potential for the number of planning calls for a single team to exceed the logarithmic bound of binary search.

Another possible extension is the removal of some of the assumptions we have made (i.e. a team can lend or borrow at most one robot). There are two extensions to this: 1) allowing a team to lend and borrow in the same plan, 2) allowing a team to lend any number of robots or borrow any number of robots (but not both).

Bibliography

[1] F. Bacchus and F. Kabanza. Using temporal logic to express search control knowledge for planning. *Artificial Intelligence*, 116(1–2):123–191, 2000.

[2] V. Bafna and P. Pevzner. Sorting by transpositions. *SIAM Journal of Discrete Mathematics*, 11:224–240, 1998.

[3] A. Bergeron. A very elementary presentation of the Hannenhalli-Pevzner theory. In *Proc. of CPM*, pages 106–117, 2001.

[4] A. Bergeron, J. Mixtacki, and J. Stoye. A unifying view of genome rearrangements. In *Proc. of WABI*, pages 163–173, 2006.

[5] A. Bergeron, J. Mixtacki, and J. Stoye. A new linear time algorithm to compute the genomic distance via the double cut and join distance. *Theoretical Computer Science*, 410(51):5300–5316, 2009.

[6] P. Berman and S. Hannenhalli. Fast sorting by reversal. In *Proc. of CPM*, pages 168–185, 1996.

[7] M. Blanchette, T. Kunisawa, and D. Sankoff. Parametric genome rearrangement. *Gene-Combis*, 172:11–17, 1996.

[8] M. Blanchette, T. Kunisawa, and D. Sankoff. Gene order breakpoint evidence in animal mitochondrial phylogeny. *Journal of Molecular Evolution*, 49:193–203, 1999.

[9] G. Bourque and P.A. Pevzner. Genome-scale evolution: Reconstructing gene orders in the ancestral species. *Genome Research*, 12(1):26–36.

[10] W. Briggs. *Modularity and Communication in Multi-Agent Planning*. PhD thesis, 1996.

[11] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[12] Ozan Caldiran, Kadir Haspalamutgil, Abdullah Ok, Can Palaz, Esra Erdem, and Volkan Patoglu. Bridging the gap between high-level reasoning and low-level control. In *Proc. of LPNMR*, 2009.

[13] M.E. Cosner, R.K. Jansen, B.M.E. Moret, L.A. Raubeson, L.S. Wang, T. Warnow, and S. Wyman. An empirical comparison of phylogenetic methods on chloroplast gene order data in *Campanulaceae*. In *Comparative Genomics*, pages 99–122. Kluwer, 2000.

[14] M.E. Cosner, L.A. Raubeson, and R.K. Jansen. Chloroplast DNA rearrangements in *Campanulaceae*: phylogenetic utility of highly rearranged genomes. *BMC Evol. Biol.*, 4(27), 2004.

[15] L. Cui, J. Leebens-Mack, L.S. Wang, J. Tang, L. Rymarquis, D.B. Stern, and C.W. dePamphilis. Adaptive evolution of chloroplast genome structure inferred using a parametric bootstrap approach. *BMC Evol. Biol.*, 6:13, 2006.

[16] K. Decker and V.R. Lesser. Generalizing the partial global planning algorithm. *International Journal of Intelligent and Cooperative Information Systems*, 1:319–346, 1992.

[17] K. Decker and V.R. Lesser. Designing a family of coordination algorithms. In *Proc. of DAI*, pages 65–84, 1994.

[18] E. H. Durfee and V. R. Lesser. Planning coordinated actions in dynamic domains. In *Proc. of the DARPA Knowledge-Based Planning Workshop*, pages 18.1–18.10, 1987.

[19] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Proc. of SAT*, pages 502–518, 2003.

[20] N. El-Mabrouk. Genome rearrangement by reversals and insertions/deletions of contiguous segments. In *Proc. of CPM*, pages 222–234, 2000.

[21] N. El-Mabrouk. Reconstructing an ancestral genome using minimum segments duplications and reversals. *J. Comput. Syst. Sci.*, 65(3):442–464, 2002.

[22] E. Ephrati and J. S. Rosenschein. Multi-agent planning as the process of merging distributed sub-plans. In *Proc. of DAI*, pages 115–129, 1993.

[23] Eithan Ephrati and Jeffrey S. Rosenschein. Divide and conquer in multi-agent planning. In *Proc. of AAAI*, 1994.

[24] E. Erdem and E. Tillier. Genome rearrangement and planning. In *Proc. of AAAI*, pages 1139–1144, 2005.

[25] E. Erdem, K. Haspalamutgil, C. Palaz, V. Patoglu, and T. Uras. Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation. In *Proc. of ICRA*, 2011.

[26] E. Erdem, K. Haspalamutgil, V. Patoglu, and T. Uras. Causality-based planning and diagnostic reasoning for cognitive factories. In *Proc. of ETFA*, 2011.

[27] J. Felsenstein. PHYLIP (phylogeny inference package) version 3.6. Distributed by the author., 2009.

[28] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[29] D. Foulser, M. Li, and Q. Yang. Theory and algorithms for plan merging. *Artificial Intelligence Journal*, 57:143–182, 1992.

[30] Michael Gelfond and Vladimir Lifschitz. Action languages. *Electronic Transactions on Artificial Intelligence*, 2:193–210, 1998.

[31] M. P. Georgeff. Communication and interaction in multi-agent planning. In *Proc. of AAAI*, pages 200–204, 1983.

[32] M. P. Georgeff. Communication and interaction in multi-agent planning. In *Readings in Distributed Artificial Intelligence*, pages 200–204, 1988.

[33] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *AIJ*, 153:49–104, 2004.

[34] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Control-based clause sharing in parallel sat solving. In *Proc. of IJCAI*, pages 499–504, 2009.

[35] S. Hannenhalli and P.A. Pevzner. Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations with reversals). In *Proc. of STOC*, pages 178–189, 1995.

[36] H. Kaplan, R. Shamir, and R.E. Tarjan. Faster and simpler algorithm for sorting signed permutations by reversals. In *Proc. of SODA*, pages 344–351, 1997.

[37] H. Kautz and B. Selman. Planning as satisfiability. In *Proc. of ECAI*, pages 359–363, 1992.

[38] J. D. Kececioglu and D. Sankoff. Efficient bounds for oriented chromosome inversion distance. In *Proc. of CPM*, pages 307–325, 1994.

[39] M. Lajoie, D. Bertrand, N. El-Mabrouk, and O. Gascuel. Duplication and inversion history of a tandemly repeated genes family. *J. of Comp. Biol.*, 14(4):462–478, 2007.

[40] V. Lesser, K. Decker, N. Carver, A. Garvey, D. Neimen, M. Prassad, and T. Wagner. Evolution of the gpgp domain independent coordination framework. Technical report, University of Massachusetts, 1998.

[41] Y. Lin, V. Rajan, K. M. Swenson, and B. M. E. Moret. Estimating true evolutionary distances under rearrangements, duplications, and losses. *BMC Bioinformatics*, 11 (16), 2010.

[42] B. J. Clement M. M. de Weerdt. Introduction to planning in multiagent systems. *Multiagent and Grid Systems An International Journal*, 5:345–355, 2009.

[43] Norman McCain and Hudson Turner. Causal theories of action and change. In *Proc. of AAAI/IAAI*, pages 460–465, 1997.

[44] D. McDermott. The formal semantics of processes in pddl. In *Proc. of ICAPS Workshop on PDDL*, 2003.

[45] B. Moret, L. Wang, T. Warnow, and S. Wyman. New approaches for reconstructing phylogenies from gene order data. *Bioinformatics*, pages 165–173, 2001.

[46] B. Moret, S. Wyman, D. Bader, T. Warnow, and M. Yan. A new implementation and detailed study of breakpoint analysis. In *Proc. of PSB*, pages 583–594, 2001.

[47] C. Nielsen. *Animal Evolution: Interrelationships of the Living Phyla*. Oxford University Press, 2001.

[48] Edwin Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. of KR*, pages 324–332, 1989.

[49] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2009.

[50] D. Sankoff. Edit distances for genome comparisons based on non-local operations. In *Proc. of CPM*, pages 121–135, 1992.

[51] D. Sankoff and M. Blanchette. Multiple genome rearrangement and breakpoint phylogeny. *J. of Comp. Biol.*, 5:555–570, 1998.

[52] D. Sankoff, J. Lefebvre, E. Tillier, A. Maler, and N. El-Mabrouk. The distribution of inversion lengths in bacteria. In *Proc. of RECOMB-CG*, pages 97–108, 2004.

[53] Y. Shoham and M. Tennenholtz. On social laws for artificial agent societies:off-line design. *Artificial Intelligence*, 73, issue = 1–2, year = 1995, pages = 231–252,.

[54] C.J. Stuart. An implementation of a multi-agent plan synchronizer. In *Proc. of IJCAI*, pages 1031–1033, 1985.

[55] K. M. Swenson, M. Marron, J. V. Earnest-DeYoung, and B. M. E. Moret. Approximating the true evolutionary distance between two genomes. In *Proc. of ISBRA*, pages 173–185, 2005.

[56] A. ter Mors, J. Valk, and C. Witteveen. Coordinating autonomous planners. In *Proc. of*, pages 795–801, 2004.

[57] G. Tesler. GRIMM: genome rearrangements web server. *Bioinformatics*, 18(3): 492–493, 2002.

[58] T. Uras and E. Erdem. Genome rearrangement and ai planning. *submitted to IEEE Transactions on Computational Biology and Bioinformatics*.

[59] T. Uras and E. Erdem. Genome rearrangement and planning: Revisited. In *Proc. of ICAPS*, 2010.

[60] S. Yancopoulos and R. Friedberg. Sorting genomes with insertions, deletions and duplications by DCJ. In *Proc. of RECOMB-CG*, pages 170–183, 2008.

[61] S. Yancopoulos, O. Attie, and R. Friedberg. Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics*, 21(16): 3340–3346, 2005.

[62] Q. Yang, D. S. Nau, and J. Hendler. Merging separately generated plans with restricted interactions. *Computational Intelligence*, 8:648–676, 1992.

[63] F. Yue, M. Zhang, and J. Tang. Phylogenetic reconstruction from transposition. *BMC Genomics*, 9, 2008.

[64] M.F. Zaeh, M. Beetz, K. Shea, G. Reinhart, K. Bender, C. Lau, M. Ostgathe, W. Vogl, M. Wiesbeck, M. Engelhard, C. Ertelt, T. Rhr, M. Friedrich, and S. Herle. The cognitive factory. In *Changeable and Reconf. Manufacturing Systems*, pages 355–371. 2009.