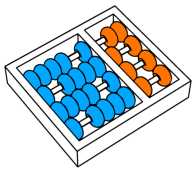
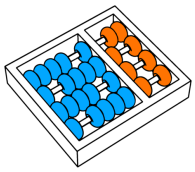


Victor de Abreu Iizuka

"Programação por Restrições aplicada a Problemas de Rearranjo de Genomas"

CAMPINAS
2012



Universidade Estadual de Campinas
Instituto de Computação

Victor de Abreu Iizuka

"Programação por Restrições aplicada a Problemas de Rearranjo de Genomas"

Orientador(a): **Prof. Dr. Zanoni Dias**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

Este exemplar corresponde à versão final da Dissertação defendida por Victor de Abreu Iizuka, sob orientação de Prof. Dr. Zanoni Dias.

Assinatura do Orientador(a)

CAMPINAS
2012

FICHA CATALOGRÁFICA ELABORADA POR
ANA REGINA MACHADO - CRB8/5467
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA - UNICAMP

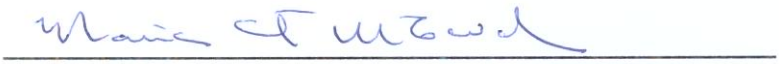
Ii1p	Iizuka, Victor de Abreu, 1987- Programação por restrições aplicada a problemas de rearranjo de genomas / Victor de Abreu Iizuka. – Campinas, SP : [s.n.], 2012. Orientador: Zanoni Dias. Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação. 1. Biologia computacional. 2. Genomas. 3. Programação por restrições. 4. Programação inteira. I. Dias, Zanoni,1975-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

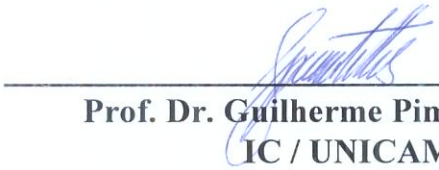
Título em inglês: Constraint proramming applied to genome rearrangement problems
Palavras-chave em inglês:
Computational biology
Genomes
Constraint programming (Computer science)
Integer programming
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Zanoni Dias [Orientador]
Maria Emília Machado Telles Walter
Guilherme Pimentel Telles
Data de defesa: 19-12-2012
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 19 de Dezembro de 2012, pela Banca examinadora composta pelos Professores Doutores:



Profª. Drª. Maria Emília Machado Telles Walter
CIC / UNB



Prof. Dr. Guilherme Pimentel Telles
IC / UNICAMP

Prof. Dr. Zanoni Dias
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Programação por Restrições aplicada a Problemas de Rearranjo de Genomas

Victor de Abreu Iizuka[1]

19 de Dezembro de 2012

Banca Examinadora:

- Prof. Dr. Zanoni Dias (Orientador)
- Prof. Dra. Maria Emília Machado Telles Walter
  Departamento de Ciência da Computação – CIC – UnB
- Prof. Dr. Guilherme Pimentel Telles
  Instituto de Computação – Unicamp
- Prof. Dr. Nalvo Franco de Almeida Junior (Suplente)
  Faculdade de Computação – UFMS
- Prof. Dr. João Meidanis (Suplente)
  Instituto de Computação – Unicamp

[1]Suporte financeiro: bolsa CNPq, processo número 134380/2009-6, de 03/2009 a 03/2011.

Abstract

The darwin's natural selection theory states that living beings of nowadays are descended from ancestors, and through evolution, genetic mutations led to the appearance of different kinds of living beings. Many mutations are point mutations, modifying the DNA sequence, which may prevent the information from being expressed, or may express it in another way. The sequence comparison is the most common method to identify the occurrence of point mutations, and is one of the most discussed problems in Computational Biology. Genome Rearrangement aims to find the minimum number of operations required to change one sequence into another. These operations may be, for example, reversals, transpositions, fissions and fusions. The concept of distance may be defined for these events, for example, the reversal distance is the minimum number of reversals required to change one sequence into another [9] and the transposition distance is the minimum number of transpositions required to change one sequence into another [10]. We will deal with the cases in which reversals and transpositions events occur separately and the cases in which both events occur simultaneously, aiming to find the exact value for the distance. We have created Constraint Programming models for sorting by reversals and sorting by reversals and transpositions, following the research line used by Dias and Dias [16]. We will present Constraint Logic Programming models for sorting by reversals, sorting by transpositions and sorting by reversals and transpositions, based on Constraint Satisfaction Problems theory and Constraint Optimization Problems theory. We made a comparison between the Constraint Logic Programming models for sorting by transpositions, described in Dias and Dias [16], and with the Integer Linear Programming formulations for sorting by reversals, sorting by transpositions and sorting by reversals and transpositions, described in Dias and Souza [17].

Resumo

A teoria da seleção natural de Darwin afirma que os seres vivos atuais descendem de ancestrais, e ao longo da evolução, mutações genéticas propiciaram o aparecimento de diferentes espécies de seres vivos. Muitas mutações são pontuais, alterando a cadeia de DNA, o que pode impedir que a informação seja expressa, ou pode expressá-la de um modo diferente. A comparação de sequências é o método mais usual de se identificar a ocorrência de mutações pontuais, sendo um dos problemas mais abordados em Biologia Computacional. Rearranjo de Genomas tem como objetivo encontrar o menor número de operações que transformam um genoma em outro. Essas operações podem ser, por exemplo, reversões, transposições, fissões e fusões. O conceito de distância pode ser definido para estes eventos, por exemplo, a distância de reversão é o número mínimo de reversões que transformam um genoma em outro [9] e a distância de transposição é o número mínimo de transposições que transformam um genoma em outro [10]. Nós trataremos os casos em que os eventos de reversão e transposição ocorrem de forma isolada e os casos quando os dois eventos ocorrem simultaneamente, com o objetivo de encontrar o valor exato para a distância. Nós criamos modelos de Programação por Restrições para ordenação por reversões e ordenação por reversões e transposições, seguindo a linha de pesquisa utilizada por Dias e Dias [16]. Nós apresentaremos os modelos de Programação por Restrições para ordenação por reversões, ordenação por transposições e ordenação por reversões e transposições, baseados na teoria do Problema de Satisfação de Restrições e na teoria do Problema de Otimização com Restrições. Nós fizemos comparações com os modelos de Programação por Restrições para ordenação por transposições, descrito por Dias e Dias [16], e com as formulações de Programação Linear Inteira para ordenação por reversões, ordenação por transposições e ordenação por reversões e transposições, descritos por Dias e Souza [17].

Agradecimentos

Gostaria de agradecer aos meus pais que fizeram todo o possível para eu conseguir chegar até aqui, dando apoio e aquele "puxão de orelha" sempre quando precisou, ao meu irmão que sempre me ajuda quando preciso e a todos os familiares que me apoiaram durante esse período.

Agradeço também ao professor Zanoni por ter auxiliado em todo o momento em que estive com dúvidas e pela paciência dispensada durante a orientação.

Finalmente, agradeço a todos os meus amigos, porque sem eles certamente não seria possível continuar esse trabalho.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A teoria da seleção natural de Darwin afirma que os seres vivos atuais descendem de ancestrais, e ao longo da evolução, mutações genéticas propiciaram o aparecimento de diferentes espécies de seres vivos.

Muitas mutações são pontuais, alterando a cadeia de DNA, o que pode impedir que a informação seja expressa, ou pode expressá-la de um modo diferente. Tais alterações debilitam, na maioria dos casos, o organismo portador ou proporcionam vantagens no processo de seleção natural.

A comparação de sequências é o método mais usual de se identificar a ocorrência de mutações pontuais, sendo um dos problemas mais abordados em Biologia Computacional. Um método de comparar duas sequências é encontrar a distância de edição [31], que é o número mínimo de operações de inserções, remoções e substituições necessárias para transformar uma sequência em outra.

A distância de edição é uma medida capaz de estimar a distância evolutiva entre duas cadeias, mas não possui a informação de quais operações ocorreram para a transformação de uma sequência em outra.

Outra abordagem usada é a de Rearranjo de Genomas, que tem como objetivo encontrar o menor número de operações que transformam um cromossomo em outro. Essas operações podem ser, por exemplo, reversões, onde um bloco do cromossomo é invertido, transposições, onde dois blocos adjacentes no cromossomo trocam de posição, fissões, efetua a quebra do cromossomo em dois cromossomos, e fusões, junta dois cromossomos em um único cromossomo.

O conceito de distância de rearranjo pode ser definido para estas operações, por exemplo, a distância de reversão é o número mínimo de reversões que transformam um genoma[1] em outro [9] e a distância de transposição é o número mínimo de transposições que transformam um genoma em outro [10].

[1]Quando usamos o termo genoma, estamos referindo a um determinado cromossomo.

Estudos mostram que os rearranjos de genomas são mais apropriados que mutações pontuais quando se deseja comparar genoma de certas espécies [30], por exemplo nas espécies de plantas *Brassica*, e no DNA cloroplasto das espécies *Tobacco fervens* e *Lobelia fervens* [9]. Nesse contexto, a distância evolutiva entre dois genomas pode ser estimada pelo conceito de distância para uma ou mais operações de rearranjo.

Neste trabalho, trataremos os casos em que os eventos de transposição e reversão ocorrem de forma isolada e os casos quando os dois eventos ocorrem ao mesmo tempo.

A operação de reversão ocorre quando um bloco do genoma é invertido. O problema da distância de reversão é encontrar o número mínimo de reversões necessárias para transformar um genoma em outro. Neste problema é importante saber se a orientação dos genes é conhecida, pois existem algoritmos polinomiais para este caso. Entretanto, se a orientação dos genes não é conhecida o problema da distância de reversão pertence à classe de problemas NP-Difíceis, com a prova apresentada por Caprara [14]. Neste caso, o melhor algoritmo de aproximação conhecido possui razão de 1.375 apresentado por Berman, Hannenhalli e Karpinski [12].

A operação de transposição ocorre quando dois blocos adjacentes no genoma trocam de posição. O problema da distância de transposição é encontrar o número mínimo de transposições necessárias para transformar um genoma em outro. Este problema pertence à classe dos problemas NP-Difíceis e a prova foi apresentada por Bulteau, Fertin e Rusu [13]. O melhor algoritmo de aproximação conhecido possui razão de 1.375 e foi apresentado por Elias e Hartman [18].

Na natureza um genoma não sofre apenas eventos de reversão ou de transposição isoladamente, ele pode sofrer mutações causadas por operações mutacionais diferentes. Para esta situação, iremos estudar o caso onde as operações de reversão e transposição ocorrem simultaneamente sobre um genoma. Os trabalhos de Walter, Dias e Meidanis [28, 33] e Lin e Xue [24] estudaram o problema de encontrar o número mínimo de reversões e transposições necessárias para transformar um genoma em outro.

Nós criamos modelos de Programação por Restrições para ordenação por reversões e ordenação por reversões e transposições, seguindo a linha de pesquisa utilizada por Dias e Dias [16]. Nós apresentaremos os modelos de Programação por Restrições (CP[2]) que buscam os resultados exatos para os problemas de ordenação por reversões, ordenação por transposições e ordenação por reversões e transposições, baseados na teoria do Problema de Satisfação de Restrições (CSP[3]) e na teoria do Problema de Otimização com Restrições (COP[4]). O modelo de programação por restrições para o problema de ordenação por reversões e transposições foi apresentado no artigo "*Constraint Logic Programming Mo-*

[2]Do inglês *Constraint Programming*.
[3]Do inglês *Constraint Satisfaction Problems*.
[4]Do inglês *Constraint Optimization Problems*.

dels for Reversal and Transposition Distance Problems" [22], publicado no *VI Brazilian Symposium on Bioinformatics (BSB'2011)*.

Nós fizemos comparações com os modelos de Programação por Restrições para ordenação por transposições, descrito por Dias e Dias [16], e com as formulações de Programação Linear Inteira (ILP[5]) que buscam os resultados exatos para os problemas de ordenação por reversões, ordenação por transposições e ordenação por reversões e transposições, descritos por Dias e Souza [17]. Tanto os modelos de CP quanto as formulações de ILP foram escritas usando softwares proprietários e softwares de código aberto, com o objetivo de comparar seus desempenhos, verificando se os softwares proprietários são inferiores ou superiores aos software de código aberto.

O texto da dissertação está dividido da seguinte maneira. O Capítulo 2 apresenta conceitos básicos necessários para o entendimento deste trabalho. O Capítulo 3 descreve os modelos usados neste trabalho. O Capítulo 4 traz a análise dos resultados obtidos durante o trabalho. O Capítulo 5 apresenta as conclusões da dissertação.

[5]Do inglês *Integer Linear Programming*.

Capítulo 2

Conceitos Básicos

Neste capítulo faremos uma apresentação dos conceitos básicos necessários para o entendimento e desenvolvimento deste trabalho. Na Seção 2.1 mostraremos as formalizações usadas pelos problemas de rearranjos de genomas. As seções 2.2, 2.3 e 2.4 descrevem, respectivamente, os problemas de ordenação por reversões, ordenação por transposições e ordenação por reversões e transposições. A Seção 2.5 explica o conceito de programação por restrições.

2.1 Definições

Para todos os problemas usamos as seguintes formalizações.

Permutação. Um genoma é representado por uma n-tupla de genes, e quando não há genes repetidos, essa n-tupla é chamada de permutação. Uma permutação é representada como $\pi = (\pi_1\ \pi_2\ \ldots\ \pi_n)$, para $\pi_i \in \mathbb{N}$, $0 < \pi_i \leq n$ e $i \neq j \leftrightarrow \pi_i \neq \pi_j$. A permutação identidade é representada como $\iota = (1\ 2\ 3\ \ldots\ n)$. Como exemplo, usaremos a permutação $\pi = (4\ 7\ 3\ 6\ 2\ 5\ 1)$.

Eventos de rearranjo. Os eventos de rearranjo tratados neste trabalho são os eventos de transposição e reversão ocorrendo isoladamente e ocorrendo de forma conjunta. Os eventos são representados por ρ e são aplicados a π de uma maneira específica. Por exemplo, uma reversão $\rho(1,3)$ aplicada a permutação $\pi = (4\ 7\ 3\ 6\ 2\ 5\ 1)$ será representada por $\pi\rho = (\underline{3\ 7\ 4}\ 6\ 2\ 5\ 1)$, e uma transposição $\rho(2,4,6)$ aplicada a permutação $\pi = (4\ 7\ 3\ 6\ 2\ 5\ 1)$ será representada por $\pi\rho = (4\ \underline{6\ 2}\ \underline{7\ 3}\ 5\ 1)$.

2.2 Ordenação por Reversões

Um evento de reversão ocorre quando um bloco do genoma é invertido. Uma reversão $\rho(i, j)$, para $1 \leq i < j \leq n$, aplicada ao genoma $\pi = (\pi_1\ \pi_2\ \ldots\ \pi_n)$ gera a permutação $\pi\rho = (\pi_1\ \ldots\ \pi_{i-1}\ \pi_j\ \pi_{j-1}\ \ldots\ \pi_{i+1}\ \pi_i\ \pi_{j+1}\ \ldots\ \pi_n)$, caso a orientação de π não seja conhecida (Figura 2.1), e $\pi\rho = (+\pi_1\ \ldots +\pi_{i-1} -\pi_j -\pi_{j-1}\ \ldots -\pi_{i+1} -\pi_i +\pi_{j+1}\ \ldots +\pi_n)$, caso a orientação de π seja conhecida (Figura 2.2).

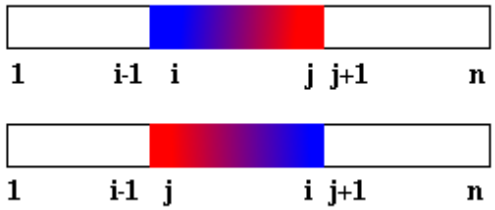


Figura 2.1: Reversão em uma permutação não orientada.

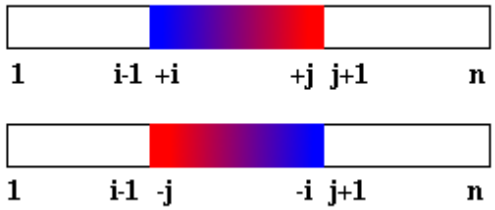


Figura 2.2: Reversão em uma permutação orientada.

O problema da distância de reversão é encontrar o número mínimo de reversões necessárias para transformar um genoma em outro. A distância de reversão entre duas permutações π e σ é representada por $d_r(\pi, \sigma)$. Note que a distância de reversão entre π e σ é igual à distância de reversão entre $\sigma^{-1}\pi$ e ι. Então, sem perda de generalidade, podemos dizer que o problema da distância de reversão é equivalente ao problema de ordenação por reversões, que é a distância de reversão entre a permutação π e a permutação identidade ι, denotado por $d_r(\pi)$.

Em um estudo inicial sobre este problema, Bafna e Pevzner [9] apresentaram um algoritmo de aproximação com razão 1.5 quando a orientação de genes é conhecida e 1.75, caso contrário.

Conhecer a orientação dos genes em um genoma é um fator importante no problema de reversão, pois existem algoritmos polinomiais para o caso em que a orientação é conhecida. No caso em que não se conhece a orientação dos genes, o problema de encontrar a distância de reversão pertence à classe dos problemas NP-Difíceis [14].

O primeiro algoritmo polinomial para o problema de reversão com orientação conhecida foi criado por Hannenhalli e Pevzner [21] que fez uso de várias operações aplicadas a uma estrutura intermediária conhecida como grafo de *breakpoints*. A estratégia usada por Hannenhalli e Pevzner foi simplificada no trabalho de Bergeron [11]. Atualmente já existe um algoritmo com complexidade sub quadrática [32] e, quando apenas a distância é necessária, um algoritmo linear pode ser usado [8].

Um resultado importante obtido por Meidanis, Walter e Dias [27] mostrou que toda teoria sobre reversões desenvolvida para genomas lineares pode ser adaptada facilmente para genomas circulares, que são comuns em seres como bactérias e plantas, por exemplo *Brassica oleracea*.

Quando a orientação dos genes não é conhecida, existem algoritmos de aproximação que seguiram a ideia do trabalho de Bafna e Pevzner citado anteriormente como, por exemplo, o algoritmo implementado por Berman, Hannenhalli e Karpinski [12] com razão de aproximação de 1.375.

O conceito de grafo de *breakpoints* foi introduzido no trabalho de Bafna e Pevzner [9]. Inicialmente a permutação π é estendida adicionando o elemento $\pi_0 = 0$ e $\pi_{n+1} = n + 1$. Dois elementos consecutivos π_i e π_{i+1}, $0 \leq i \leq n$, são *adjacentes* quando $|\pi_i - \pi_{i+1}| = 1$, e são *breakpoints* caso contrário. Define-se um grafo de arestas coloridas $G(\pi)$ com $n + 2$ vértices $\{\pi_0, \pi_1, \ldots, \pi_n, \pi_{n+1}\}$. Unimos os vértices π_i e π_j com uma aresta preta se (π_i, π_j) for um *breakpoint* na permutação π. Unimos os vértices π_i e π_j com uma aresta cinza se $|\pi_i - \pi_j| = 1$ e π_i, π_j não são consecutivos em π. Denotamos por $b_r(\pi)$ o número de *breakpoints* existentes em π com relação a permutação identidade ι. A Figura 2.3 mostra o grafo de *breakpoints* da permutação $\pi = (4\ 7\ 3\ 6\ 2\ 5\ 1)$. Neste caso o número de *breakpoints* é $b_r(\pi) = 8$.

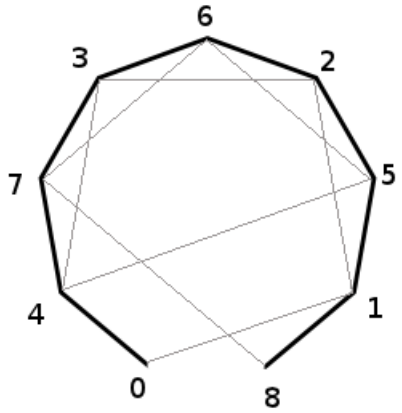


Figura 2.3: Grafo de *breakpoints* da permutação $\pi = (4\ 7\ 3\ 6\ 2\ 5\ 1)$.

Usando o conceito de *breakpoints*, temos que uma reversão atua em dois pontos em uma permutação e, portanto, pode reduzir [1] o número de *breakpoints* em pelo menos um e no máximo dois [9], levando ao Teorema 2.1.

Teorema 2.1. *Para qualquer permutação* π*,*

$$\frac{1}{2}b_r(\pi) \leq d_r(\pi) \leq b_r(\pi).$$

Um ciclo em $G(\pi)$ é chamado de *alternado* se as cores de duas arestas consecutivas são diferentes ao longo do ciclo. Assim dizemos que todos os ciclos pertencentes ao grafo serão ciclos alternados. O *comprimento* de um ciclo é a sua quantidade de arestas pretas. Um k-ciclo é um ciclo que contém k arestas pretas. Um *ciclo longo* é um ciclo de comprimento maior que dois.

Observe que $G(\pi)$ pode ser decomposto em ciclos de arestas disjuntas, pois cada vértice tem o mesmo número de arestas incidentes cinzas e pretas. Logo existem diversas maneiras de realizar a decomposição de ciclos em $G(\pi)$. A Figura 2.4 mostra um exemplo de decomposição em ciclos para o grafo de *breakpoints* da permutação $\pi = (4\ 7\ 3\ 6\ 2\ 5\ 1)$.

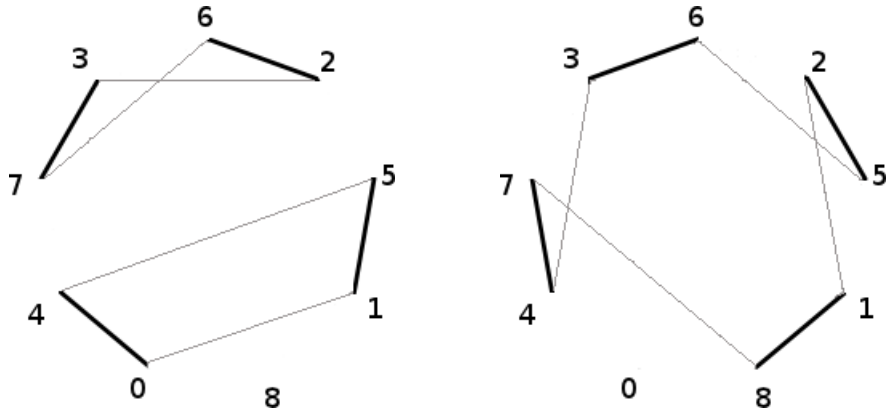


Figura 2.4: Exemplo de decomposição em ciclos de arestas disjuntas para o grafo de *breakpoints* da permutação $\pi = (4\ 7\ 3\ 6\ 2\ 5\ 1)$.

Uma reversão atua em duas arestas pretas de $G(\pi)$, se as arestas representam os *breakpoints* que são separados pela operação de reversão [15]. O Teorema 2.2, demonstrado no trabalho de Christie [15], fornece novos limites para a distância de reversão usando a quantidade de 2-ciclos, além dos breakpoints, na máxima decomposição em ciclos de $G(\pi)$.

[1] Uma reversão pode aumentar o número de *breakpoints*, mas queremos reduzir o seu número para diminuir a diferença entre o número de *breakpoints* da permutação π e o número de *breakpoints* da permutação identidade ι ($b_r(\iota) = 0$).

Teorema 2.2. *Se $c_2(\pi)$ é o número mínimo de 2-ciclos em qualquer máxima decomposição em ciclos de $G(\pi)$ então:*

$$\frac{2}{3}b_r(\pi) - \frac{1}{3}c_2(\pi) \leq d_r(\pi) \leq b_r(\pi) - \frac{1}{2}c_2(\pi).$$

2.3 Ordenação por Transposições

Um evento de transposição ocorre quando dois blocos adjacentes no genoma trocam de posição. Uma transposição $\rho(i, j, k)$, para $1 \leq i < j < k \leq n + 1$, aplicada ao genoma $\pi = (\pi_1\ \pi_2\ \ldots\ \pi_n)$ gera a permutação $\pi\rho = (\pi_1\ \ldots\ \pi_{i-1}\ \pi_j\ \ldots\ \pi_{k-1}\ \pi_i\ \ldots \pi_{j-1}\ \pi_k\ \ldots\ \pi_n)$ (Figura 2.5).

Figura 2.5: Transposição aplicada em uma permutação.

O problema da distância de transposição é encontrar o número mínimo de transposições necessárias para transformar um genoma em outro. A distância de transposição entre duas permutações π e σ é representada por $d_t(\pi, \sigma)$. Note que a distância de transposição entre π e σ é igual à distância de transposição entre $\sigma^{-1}\pi$ e ι. Então, sem perda de generalidade, podemos dizer que o problema da distância de transposição é equivalente ao problema de ordenação por transposições, que é a distância de transposição entre a permutação π e a permutação identidade ι, denotado por $d_t(\pi)$.

Este problema foi estudado por Bafna e Pevzner [10], que apresentaram um algoritmo capaz de fornecer uma resposta aproximada na razão de 1.5, além de derivar um importante limite inferior para o problema. Introduziram também o conceito de *breakpoints* em eventos de transposições, elementos adjacentes em um genoma, mas não no outro, e o conceito de grafo de ciclos, ambos ferramentas importantes utilizadas para encontrar limitantes para o problema. Foram apresentadas várias questões em aberto, como verificar a complexidade do problema da distância de transposição e o diâmetro, que é a maior distância possível entre duas permutações de tamanho n. O problema do diâmetro foi estudado por Meidanis, Walter e Dias [26].

A complexidade deste problema ficou em aberto por um longo tempo. O trabalho de Bulteau, Fertin e Rusu [13] apresentou a prova de que o problema de ordenação por

transposição pertence a classe dos problemas NP-Difíceis. Elias e Hartman [18] apresentaram um algoritmo de aproximação na razão de 1.375. O trabalho de Labarre [23] apresentou novos limitantes, além de definir classes de permutações em que a distância de transposição pode ser calculada em tempo e espaço lineares.

No problema de ordenação por transposições, um *breakpoint* é um par (π_i, π_{i+1}) tal que $\pi_{i+1} \neq \pi_i + 1$. Denota-se por $b_t(\pi)$ como sendo o número de *breakpoints* na permutação π. Sabemos que uma transposição atua em três pontos de uma permutação, logo, pode reduzir [2] o número de *breakpoints* em pelo menos um e no máximo três [10], levando ao Teorema 2.3.

Teorema 2.3. *Para qualquer permutação* π*,*

$$\frac{1}{3}b_t(\pi) \leq d_t(\pi) \leq b_t(\pi).$$

O conceito de grafo de ciclos foi introduzido por Bafna e Pevzner [10] e foi usado para obter limitantes melhores para o problema. Um grafo direcionado com arestas coloridas, denotado por $G(\pi)$, é chamado de grafo de ciclos da permutação π se possui um conjunto de vértices $\{0,\ 1,\ \ldots,\ n+1\}$ e seu conjunto de arestas é definido como para todo $1 \leq i \leq n+1$, arestas cinzas são direcionadas de $i-1$ para i e arestas pretas de π_i para π_{i-1}. A Figura 2.6 mostra o grafo de ciclos para a permutação $\pi = (4\ 7\ 3\ 6\ 2\ 5\ 1)$.

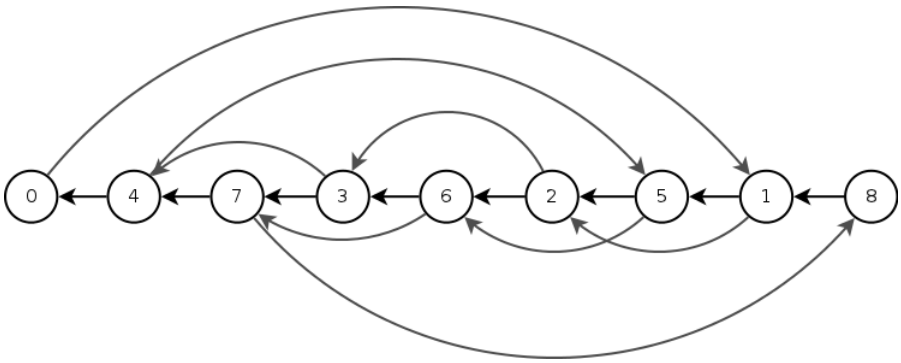


Figura 2.6: Grafo de ciclos para a permutação $\pi = (4\ 7\ 3\ 6\ 2\ 5\ 1)$.

De forma similar ao problema de ordenação por reversões, um ciclo de $G(\pi)$ é chamado de *alternado* se ele for um ciclo direcionado com arestas de cores alternadas. Para todo vértice de $G(\pi)$ toda aresta chegando é unicamente pareada com uma aresta saindo de cor diferente. Isto implica que existe uma decomposição única de ciclos alternados do

[2] Assim como no caso de reversão, uma transposição pode aumentar o número de *breakpoints*, mas queremos reduzir o seu número para diminuir a diferença entre o número de *breakpoints* da permutação π e o número de *breakpoints* da permutação identidade ι ($b_t(\iota) = 0$).

conjunto de arestas de $G(\pi)$. A seguir o termo ciclo é usado no lugar de ciclos alternados e usamos o termo k-ciclo para definir um ciclo alternado de tamanho $2k$, k-ciclo é longo se $k > 2$, e curto caso contrário. A Figura 2.7 mostra um exemplo de decomposição em ciclos para o grafo de ciclos da permutação $\pi = (4\ 7\ 3\ 6\ 2\ 5\ 1)$.

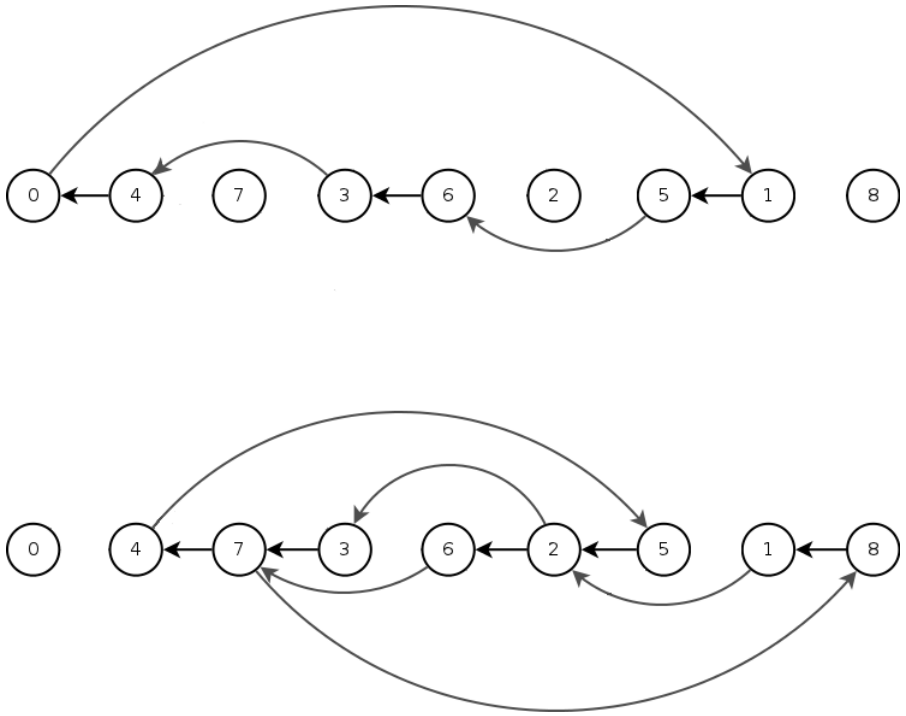


Figura 2.7: Exemplo de decomposição em ciclos de arestas disjuntas para o grafo de ciclos da permutação $\pi = (4\ 7\ 3\ 6\ 2\ 5\ 1)$.

Para melhorar o limitante, Bafna e Pevzner [10] estudaram separadamente os ciclos pares e ímpares. Um ciclo é impar se possui um número ímpar de arestas pretas e par caso contrário. Seja $c_{\text{ímpar}}(\pi)$ o número de ciclos ímpares de $G(\pi)$, para uma permutação π, e $\Delta c_{\text{ímpar}}(\pi, \rho) = c_{\text{ímpar}}(\pi\rho) - c_{\text{ímpar}}(\pi)$ a mudança no número de ciclos ímpares devido a transposição ρ, temos que $\Delta c_{\text{ímpar}} \in \{2, 0, -2\}$ gerando o resultado do Teorema 2.4. Note que a permutação identidade possui $c_{\text{ímpar}}(\iota) = n + 1$, então se a cada transposição for possível aumentar o número de ciclos ímpares da permutação π ficaremos mais próximos de transformá-la na permutação identidade ι.

Teorema 2.4. *Para qualquer permutação π,*

$$\frac{1}{2}(n + 1 - c_{\acute{i}mpar}(\pi)) \leq d_t(\pi) \leq \frac{3}{4}(n + 1 - c_{\acute{i}mpar}(\pi)).$$

2.4 Ordenação por Reversões e Transposições

Na natureza um genoma não sofre apenas eventos de reversão ou de transposição, ele está exposto a diversos eventos mutacionais diferentes. Para esta situação, iremos estudar o caso onde os eventos de reversão e transposição ocorrem simultaneamente em um genoma.

O problema da distância de reversão e transposição é encontrar o número mínimo de reversões e transposições necessárias para transformar um genoma em outro. A distância de reversão e transposição entre duas permutações π e σ é representada por $d_{rt}(\pi, \sigma)$. De forma similar ao caso em que os eventos ocorrem individualmente, podemos dizer, sem perda de generalidade, que o problema da distância de reversão e transposição é equivalente ao problema de ordenação por reversões e transposições, que é a distância de reversão e transposição entre a permutação π e a permutação identidade ι, denotado por $d_{rt}(\pi)$.

Este problema foi estudado por Hannenhalli e coautores [20], que analisaram a evolução de genomas por diferentes tipos de eventos, em especial reversões e transposições.

Em 1998, Walter, Dias e Meidanis [33] apresentaram um algoritmo de aproximação para a distância de reversão e transposição, além de limitantes para o diâmetro de reversão e transposição em permutações orientadas que foram posteriormente melhorados [28].

No trabalho de Gu, Peng e Sudborough [19] é apresentado um algoritmo 2-aproximado para computar a distância entre dois genomas com a orientação dos genes conhecida usando a operação de reversão e transposição simultaneamente.

2.5 Programação por Restrições

Programação por Restrições é um paradigma de programação que usa restrições para estabelecer as relações entre as variáveis. Diferentemente da programação imperativa, as restrições não usam passos para executar, mas usam as propriedades da solução a ser encontrada. Resumidamente, uma restrição sobre uma sequência de variáveis é a relação entre seus domínios. Pode ser vista como um requisito que diz quais combinações de valores dos domínios das variáveis serão admitidas. Um problema é então simplificado usando entidades e seus relacionamentos. As entidades de um modelo de programação por restrição são chamadas de variáveis e os relacionamentos de restrições.

Definição 2.1. *Um modelo de programação por restrições P é formado por:*

- *Um conjunto de variáveis $X = \{x_1, \ldots, x_n\}$, com seus respectivos domínios $D_1, \ldots, D_n$.*
- *Um conjunto finito de restrições C, cada um sobre uma subsequência de X.*

Então, o modelo pode ser escrito como $P = \langle C; x_1 \in D_1, \ldots, x_n \in D_n \rangle$. A solução é a associação $\{(x_1, d_1), \ldots, (x_n, d_n)\}$, onde $d_i \in D_i$, que satisfaz todas as restrições em C. Um modelo P é chamado *consistente* (ou *viável*) se possuir pelo menos uma solução, caso contrário, é chamado de *inconsistente* (ou *inviável*).

Os modelos que usam a teoria do Problema de Satisfação de Restrições (CSP[3]) são descritos conforme a Definição 2.1 acima. Para exemplificar os modelos que usam a teoria CSP, iremos usar o problema das n rainhas, que é um dos problemas mais conhecido baseado na teoria CSP.

O problema das n rainhas consiste em posicionar as rainhas em um tabuleiro $n \times n$, onde $n \geq 3$, de modo que nenhuma rainha seja atacada por outra. A Figura 2.8 mostra uma das soluções para o problema quando temos $n = 8$.

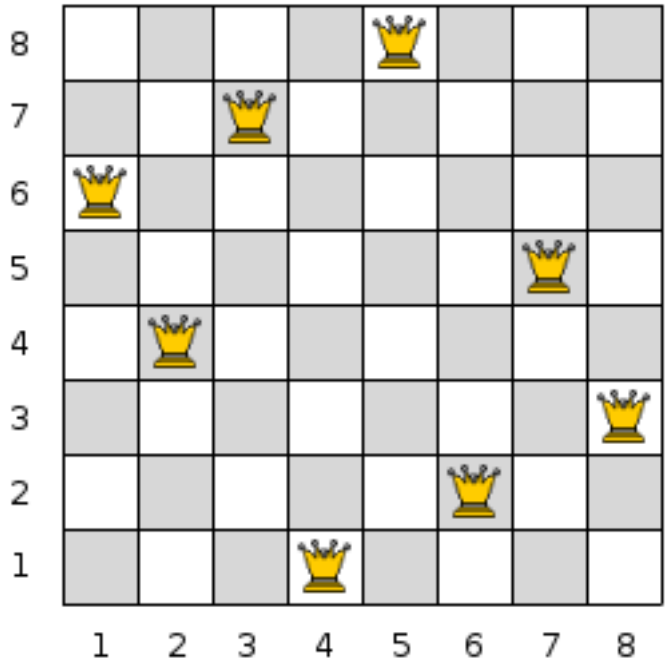


Figura 2.8: Uma das soluções para o problema das 8 rainhas.

Uma das possíveis representações usando a teoria CSP para o problema usa n variáveis, $X = \{x_1, \ldots, x_n\}$, onde $1 \leq x_i \leq n$ para todo $1 \leq i \leq n$. Então, uma variável x_i representa a posição da rainha posicionada na i-ésima coluna do tabuleiro. Por exemplo, a solução mostrada na figura 2.8 corresponde aos valores $X = \{6, 4, 7, 1, 8, 2, 5, 3\}$, sendo que a primeira rainha da esquerda foi posicionada na sexta linha de baixo para cima, a segunda rainha foi posicionada na quarta linha, e segue desta maneira para as outras rainhas.

[3]Do inglês *Constraint Satisfaction Problems*.

O conjunto de restrições pode ser formulado usando as seguintes desigualdades para $1 \leq i \leq n-1$ e $i+1 \leq j \leq n$:

- $x_i \neq x_j$ (Não permite duas rainhas na mesma linha),
- $x_i - x_j \neq i - j$ (Não permite duas rainhas na diagonal ascendente),
- $x_i - x_j \neq j - i$ (Não permite duas rainhas na diagonal descendente).

Os modelos que são baseados na teoria do Problema de Otimização com Restrições (COP[4]) possuem o objetivo de encontrar a melhor solução de um conjunto de restrições, usando uma função de custo, ou seja, considerando um modelo da teoria CSP, $P_{csp} = \langle C; x_1 \in D_1, \ldots, x_n \in D_n \rangle$, e uma função de custo, $custo : D_1 \times \ldots \times D_n \rightarrow R$, queremos encontrar a solução $\{(x_1, d_1), \ldots, (x_n, d_n)\}$ de P_{csp}, para qual o valor $custo(d_1, \ldots, d_n)$ seja ótimo. Logo, os modelos baseados na teoria COP são representados como $P_{cop} = \langle P_{csp}, custo \rangle$. Para exemplificar os modelos que usam a teoria COP, iremos usar o problema da mochila, um problema bastante famoso na área de otimização combinatória.

O problema da mochila é preencher a mochila com um conjunto de objetos cujo valor total seja máximo, sem ultrapassar o peso total da mochila. Formalizando, nós temos n objetos com pesos $A = \{a_1, \ldots, a_n\}$ e valores $B = \{b_1, \ldots, b_n\}$ e o peso máximo que a mochila suporta w. Usaremos o conjunto de variáveis binárias $X = \{x_1, \ldots, x_n\}$, que serão usadas para determinar se o objeto i será colocado ou não dentro da mochila, sendo que $x_i = 1$ caso o objeto é colocado na mochila, e $x_i = 0$ caso contrário.

A restrição do problema é se a mochila suporta o peso total dos objetos colocados:

$$\sum_{i=1}^{n} a_i \cdot x_i \leq w$$

Nós então procuramos a solução para esta restrição, que será a seguinte soma:

$$\sum_{i=1}^{n} b_i \cdot x_i$$

Como o modelo é baseado na teoria COP, precisamos otimizar a solução. Logo, a função de custo será:

$$\max \sum_{i=1}^{n} b_i \cdot x_i$$

Nesta dissertação, usamos duas abordagens diferentes para a criação dos modelos, uma baseada na teoria do Problema de Satisfação de Restrições, e outra baseada na teoria do

[4]Do inglês *Constraint Optimization Problems*.

Problema de Otimização com Restrições. Recomendamos a leitura de Apt [6], Apt e Wallace [7] e Marriot e Stuckey [25] para um aprofundamento maior sobre programação por restrições.

Métodos de Solução dos Problemas de Programação por Restrições. Para encontrar soluções em um modelo de programação por restrições, utiliza-se algoritmos de propagação de restrições, cujo objetivo é reduzir o espaço de busca nos domínios das variáveis. Esses algoritmos fazem a redução do problema em outro mais simples de solucionar. Para lidar com essa situação, usamos a Definição 2.2.

Definição 2.2. *Considere dois modelos de programação por restrições P_1 e P_2 e uma sequência X das suas variáveis comuns ($X \subset X_1$ e $X \subset X_2$, onde X_1 e X_2 são, respectivamente, sequências das variáveis de P_1 e P_2). Então, P_1 e P_2 são equivalentes em relação a X, se:*

- *para toda solução d em P_1, existe uma solução em P_2 que coincide com d nas variáveis em X.*
- *para toda solução e em P_2, existe uma solução em P_1 que coincide com e nas variáveis em X.*

Mas, em muitos casos, os modelos de programação por restrições possuem restrições que não se ligam às restrições simples ou são um grupo de restrições de diversos tipos. Então, nestes casos utiliza-se métodos baseados em buscas sobre os domínios das variáveis. A seguir, listaremos alguns métodos de buscas mais utilizados.

- **Busca Local:** Classe de algoritmo, usada para os modelos que usam as teorias CSP e COP, que possui o objetivo de encontrar uma solução utilizando uma atribuição inicial definida sobre as variáveis (chamada neste contexto de *estado*) e tenta melhorar sua *qualidade* a cada iteração, fazendo pequenas mudanças locais, chamadas de *movimento*. A qualidade do estado é definida por uma função de custo (Ex: Número de restrições violadas pelo estado. Então a qualidade de uma solução será 0.).

  O conceito principal de uma busca local é a utilização de *vizinhança*, que tem o objetivo de associar para cada estado um conjunto de estados, chamados de *vizinhos*. Então, a busca local começa do estado inicial, entra em um *loop*, no qual realiza um movimento de um estado para seu vizinho. O estado final é ou uma solução para o modelo, ou um estado de parada, indicando que nenhuma solução foi encontrada até este estado.

- **Busca Top-Down:** Método de busca mais usado em modelos que usam a teoria CSP. Utiliza a estratégia de *branching* em conjunto aos algoritmos de propagação de restrições. O *branching* possibilita a divisão de um modelo CSP em dois ou mais CSPs, sendo que a união destes é equivalente ao problema inicial. A propagação de restrição permite transformar um dado modelo CSP em um equivalente mais simples. A Busca top-down alterna os métodos de *branching* e de propagação de algoritmos, usando uma árvore que é chamada de *árvore de busca*. As folhas desta árvore são ou CSPs inconsistentes ou uma solução para um dos CSPs gerados pela técnica. A Figura 2.9 apresenta um exemplo de uma árvore de busca, note que a árvore é gerada *on-the-fly*[5].

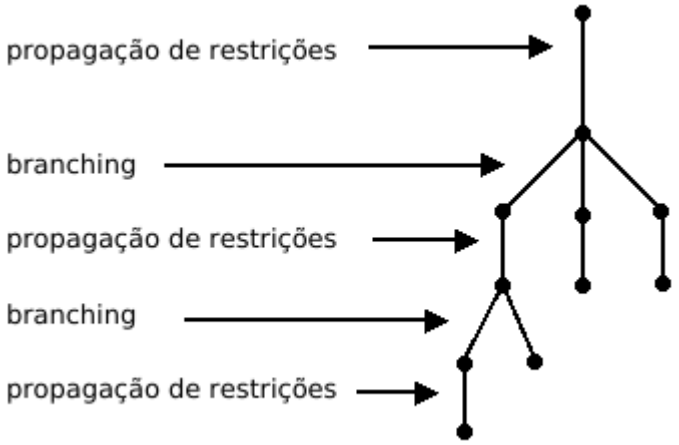


Figura 2.9: Exemplo de árvore de busca para um modelo CSP.

O procedimento padrão de uma busca top-down é o *backtracking*. Ele inicia a busca pelo nó raiz da árvore e segue para o primeiro nó descendente. O processo continua até que uma folha é encontrada, neste caso, ele retorna para o nó ancestral mais próximo que possua outro nó descendente, e então o processo recomeça. Se o controle voltou para o nó raiz e todos os descendentes foram visitados, o processo termina.

Um exemplo de *branching* é a técnica conhecida como *labelling*, que divide um domínio (finito) de uma variável em domínios unitários, correspondendo a uma busca sistemática de todos os valores de uma determinada variável. Uma forma de propagação de restrições, combinada com o *branching*, é aplicada ao longo da árvore, removendo valores dos domínios das variáveis que não participam de nenhuma solução.

[5]Gerada no momento da execução do modelo.

- **Busca Branch and Bound:** Método de busca mais usado em modelos que usam a teoria COP. Utiliza a técnica de *backtracking* levando em conta a função de custo. Suponha que o objetivo do problema é encontrar uma solução com o valor mínimo para a função de custo. Durante a busca usamos uma variável *bound* para guardar o *melhor valor encontrado*. O valor inicial desta variável é ∞. Então a cada vez que uma solução menor é encontrada, este valor é guardado em *bound*.

  Há diversas variações no algoritmo de branch and bound, mas um ponto importante a ser considerado é o que fazer após encontrar uma solução com melhor custo. O método mais simples é reiniciar o processamento com a variável *bound* inicializada com o novo valor para o custo.

  Uma alternativa é continuar a busca por soluções melhores sem reiniciar o processamento. Neste caso a restrição $custo(x_1, \ldots, x_n) < bound$ é usada e a cada vez que encontramos uma solução com custo melhor, esta solução é adicionada dinamicamente à restrição $custo(x_1, \ldots, x_n) < bound$. A propagação de restrição é acionada por esta restrição, levando à poda da árvore de busca ao identificar que as soluções a partir de um determinado nó não pode gerar uma solução com custo melhor que o atual, mantido pela variável *bound*. A Figura 2.10 mostra o estado de uma árvore de busca usando o método branch and bound. As linhas pontilhadas representam as partes ignoradas durante a busca por não gerar uma solução melhor.

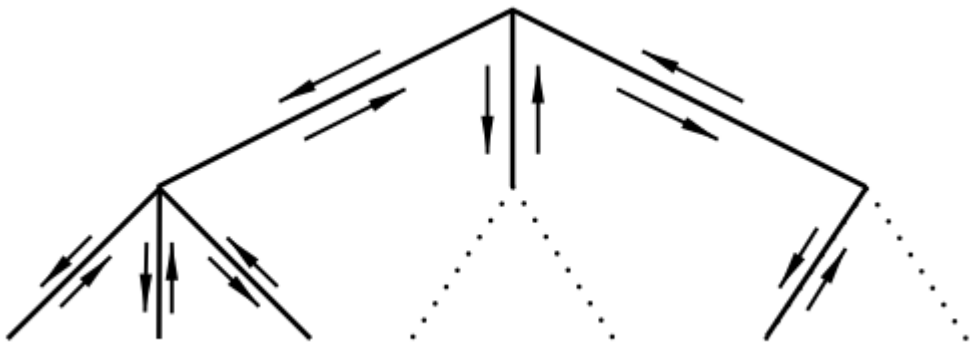


Figura 2.10: Exemplo de árvore de busca usando método branch and bound.

Capítulo 3

Modelos

Neste capítulo nós apresentaremos a descrição dos modelos de programação por restrições (Seção 3.1) e programação linear inteira (Seção 3.2) usados para os problemas de ordenação por transposições, ordenação por reversões e ordenação por reversões e transposições.

3.1 Programação por Restrições

O modelo de programação por restrições usado para o problema de ordenação por transposições é o descrito por Dias e Dias [16]. Nós usamos as definições contidas no Capítulo 2 para criar as formulações para os problemas de ordenação por reversões e ordenação por reversões e transposições, baseadas nas teorias do Problema de Satisfação de Restrições (CSP) e do Problema de Otimização com Restrições (COP). As formulações foram descritas usando a notação *prolog-like* de Marriot [25]. O artigo [22], publicado no *VI Brazilian Symposium on Bioinformatics (BSB'2011)*, apresenta o modelo de programação por restrições para o problema de ordenação por reversões e transposições. Primeiramente iremos apresentar os predicados que são comum às duas formulações, na Seção 3.1.1 apresentaremos o modelo baseado na teoria do Problema de Satisfação de Restrições e na Seção 3.1.2 o modelo baseado na teoria do Problema de Otimização com Restrições.

Em Prolog as variáveis são descritas por *strings* iniciadas com letra maiúscula ou "_" (*underscore*) caso a variável seja anônima. As letras gregas π e σ representam listas nesta notação. A construção $X :: [i .. j]$ significa que X (ou cada elemento de X se X for uma lista) pode assumir um valor do intervalo $[i .. j]$. Átomos são constantes de texto que representam relações, funções ou objetos, são iniciados com letras minúsculas, caso contrário é necessário usar aspas simples. Alguns exemplos de átomos são x, azul, 'Aluno', 'quem é você?'. Um predicado por ser representado usando a seguinte representação: *átomo/aridade*, onde o átomo representa o nome do predicado, e aridade é o número de parâmetros que o predicado pode receber, por exemplo, o predicado *length/2* possui o

nome *length* e possui dois parâmetros.

A representação da permutação (3.1) e o efeito das operações de reversão (3.2) e transposição (3.3) podem ser vistas da mesma maneira que são descritas pelos problemas. Neste modelo a permutação π é uma lista de elementos $(\pi_1,\ \pi_2,\ \ldots\ ,\ \pi_n)$ onde $\pi_i \in \mathbb{N}$, $0 < \pi_i \leq n$ e $\pi_i \neq \pi_j$ para $i \neq j$.

$$\begin{aligned} &\mathit{permutation}(\pi, N)\ \text{:-} \\ &\quad \mathit{length}(\pi, N), \\ &\quad \pi\ ::\ [1\ ..\ N], \\ &\quad \mathit{all\_different}(\pi). \end{aligned} \tag{3.1}$$

Na reversão $\rho(i, j)$, $0 < i < j \leq n$, dividimos a lista em três sublistas $C_1C_2C_3$ onde $C_1 = (\pi_1\ ..\ \pi_{i-1})$, $C_2 = (\pi_i\ ..\ \pi_j)$ e $C_3 = (\pi_{j+1}\ ..\ \pi_n)$. Depois fazemos a reversão na sublista C_2, resultando na lista R_{C_2}. Então juntamos a nova lista R_{C_2} com as sublistas C_1 e C_3 para formar $\pi\rho = C_1R_{C_2}C_3$. Observe que as listas C_1 e C_3 podem ser vazias.

$$\begin{aligned} &\mathit{reversal}(\pi, \sigma, I, J)\ \text{:-} \\ &\quad \mathit{permutation}(\pi, N), \\ &\quad \mathit{permutation}(\sigma, N), \\ &\quad 1 \leq I < J \leq N, \\ &\quad \mathit{split}(\pi, I, J, C_1, C_2, C_3), \\ &\quad \mathit{reverse}(C_2, R_{C_2}), \\ &\quad \sigma = C_1, R_{C_2}, C_3. \end{aligned} \tag{3.2}$$

Na transposição $\rho(i, j, k)$, $0 < i < j < k \leq n$, dividimos a lista em quatro sublistas $C_1C_2C_3C_4$ onde $C_1 = (\pi_1\ ..\ \pi_{i-1})$, $C_2 = (\pi_i\ ..\ \pi_{j-1})$, $C_3 = (\pi_j\ ..\ \pi_{k-1})$ e $C_4 = (\pi_k\ ..\ \pi_n)$. Trocamos de posição os blocos C_2 e C_3 e as juntamos na ordem C_1, C_3, C_2 e C_4 para formar $\pi\rho = C_1C_3C_2C_4$. Observe que as sublistas C_1 e C_4 podem ser vazias.

$$\begin{aligned} &\mathit{transposition}(\pi, \sigma, I, J, K)\ \text{:-} \\ &\quad \mathit{permutation}(\pi, N), \\ &\quad \mathit{permutation}(\sigma, N), \\ &\quad 1 \leq I < J < K \leq N, \\ &\quad \mathit{split}(\pi, I, J, K, C_1, C_2, C_3, C_4), \\ &\quad \sigma = C_1, C_3, C_2, C_4. \end{aligned} \tag{3.3}$$

3.1.1 Modelo CSP

Primeiramente modelaremos o problema usando a teoria CSP, mas o número de variáveis é desconhecido devido ao fato de precisarmos do valor da distância de reversão $d_r(\pi)$

para criar as restrições e variáveis que representam as permutações. Por esta razão, nós escolhemos um valor candidato para a distância R tal que $R \in [LB \,..\, UB]$, onde LB é um limitante inferior e UB é um limitante superior, ambos conhecidos, para o problema, e tentamos achar uma combinação apropriada de R reversões que solucionam o problema. Se o modelo CSP falha (não existe combinação que soluciona o problema com o valor escolhido) com o candidato R, nós escolhemos outro valor R apenas incrementando seu valor. O valor de R é escolhido usando uma estratégia *bottom-up*, ou seja, a verificação inicia pelo valor do limitante inferior LB e termina quando o valor é maior que o limitante superior UB. Na transposição, o processo é o mesmo que na reversão, trocando apenas o valor da distância de reversão ($d_r(\pi)$) para o valor da distância de transposição ($d_t(\pi)$). A variável *Model* recebe um átomo que representa qual o modelo e o limitante que serão usados para solucionar o problema.

$$
\begin{aligned}
&\mathit{reversal\_distance}(\iota, 0, \_Model).\\
&\mathit{reversal\_distance}(\pi, R, Model) \text{ :-}\\
&\quad \mathit{bound}(\pi, Model, LB, UB),\\
&\quad R :: [LB \,..\, UB],\\
&\quad \mathit{indomain}(R),\\
&\quad \mathit{reversal}(\pi, \sigma, \_I, \_J),\\
&\quad \mathit{reversal\_distance}(\sigma, R-1, Model).
\end{aligned}
\tag{3.4}
$$

$$
\begin{aligned}
&\mathit{transposition\_distance}(\iota, 0, \_Model).\\
&\mathit{transposition\_distance}(\pi, T, Model) \text{ :-}\\
&\quad \mathit{bound}(\pi, Model, LB, UB),\\
&\quad T :: [LB \,..\, UB],\\
&\quad \mathit{indomain}(T),\\
&\quad \mathit{transposition}(\pi, \sigma, \_I, \_J, \_K),\\
&\quad \mathit{transposition\_distance}(\sigma, T-1, Model).
\end{aligned}
\tag{3.5}
$$

O predicado *rev_trans_dist/3* (3.6) retorna o valor da distância de reversão e transposição. O predicado *event/2* escolhe o melhor evento entre o predicado *reversal/4* (3.2)

e o predicado *transposition/5* (3.3) para minimizar o valor da distância.

$$
\begin{aligned}
&\mathit{rev\_trans\_dist}(\iota, 0, \_Model).\\
&\mathit{rev\_trans\_dist}(\pi, N, Model) \text{ :-}\\
&\quad \mathit{bound}(\pi, Model, LB, UB),\\
&\quad N :: [LB \mathinner{..} UB],\\
&\quad \mathit{indomain}(N),\\
&\quad \mathit{event}(\pi, \sigma),\\
&\quad \mathit{rev\_trans\_dist}(\sigma, N-1, Model).
\end{aligned}
\tag{3.6}
$$

O predicado *indomain(X)* em (3.4), (3.5) e (3.6) analisa o domínio da variável X e escolhe o menor elemento dele (no caso, o valor do limitante inferior). Se o modelo retorna para o predicado *indomain* devido a uma falha, o elemento que a originou será removido do domínio e um outro valor será escolhido.

Os modelos CSP para os problemas de ordenação possuem a estrutura mostrada acima, trocando apenas os limitantes usados. A seguir apresentaremos os predicados que lidam com a escolha do modelo a ser usado.

Em comum para os todos eventos descritos temos o modelo *def_csp* que usa limitantes triviais[1].

$$
\begin{aligned}
&\mathit{bound}(\pi, def\_csp, LB, UB) \text{ :-}\\
&\quad LB = 0,\\
&\quad \mathit{length}(\pi, UB).
\end{aligned}
\tag{3.7}
$$

Modelos específicos para ordenação por reversões:

- *rev_br_csp*: Modelo que usa o conceito de *breakpoints* em reversões para calcular os limitantes descritos no Teorema 2.1.
- *rev_cg_csp*: Modelo que usa o número de 2-ciclos na máxima decomposição em ciclos de $G(\pi)$ para calcular os limitantes descritos no Teorema 2.2.

[1]Modelos que não usam nenhum limitante, ou seja, o limitante inferior é igual à 0 e o limitante superior é igual ao tamanho da permutação fornecida.

$$
\begin{aligned}
&bound(\pi, rev\_br\_csp, LB, UB) \text{ :-} \\
&\quad calc\_reversal\_breakpoints(\pi, B), \\
&\quad LB = B/2 \\
&\quad UB = B. \\
&bound(\pi, rev\_cg\_csp, LB, UB) \text{ :-} \\
&\quad calc\_reversal\_breakpoints(\pi, B), \\
&\quad find\_2\_cycle(\pi, B, C), \\
&\quad LB = (2 * B - C)/3, \\
&\quad UB = B - C/2.
\end{aligned}
\tag{3.8}
$$

Modelos específicos para ordenação por transposições:

- *tra_br_csp*: Modelo que usa o conceito de *breakpoints* em transposições para calcular os limitantes conforme descrito no Teorema 2.3.
- *tra_cg_csp*: Modelo que usa o conceito de grafo de ciclos em transposições, fazendo a decomposição de ciclos e analisando os ciclos ímpares separadamente para calcular os limitantes conforme descrito no Teorema 2.4.

$$
\begin{aligned}
&bound(\pi, tra\_br\_csp, LB, UB) \text{ :-} \\
&\quad calc\_transposition\_breakpoints(\pi, B), \\
&\quad LB = B/3, \\
&\quad UB = B. \\
&bound(\pi, tra\_cg\_csp, LB, UB) \text{ :-} \\
&\quad length(\pi, N), \\
&\quad calc\_oddcycles\_transposition(\pi, N, C), \\
&\quad LB = (N + 1 - C)/2, \\
&\quad UB = (3 * (N + 1 - C))/4.
\end{aligned}
\tag{3.9}
$$

Modelos específicos para ordenação por reversões e transposições:

- *r_t_br_csp*: Melhor limitante superior entre o limitante de *breakpoints* para reversões e o limitante de *breakpoints* para transposições.
- *r_t_bc_csp*: Melhor limitante superior entre o limitante de *breakpoints* para reversões e o limitante do grafo de ciclos para transposições.

- *r_t_cc_csp*: Melhor limitante superior entre o limitante do grafo de ciclos para reversões e o limitante do grafo de ciclos para transposições.

$$
\begin{aligned}
&bound(\pi, r\_t\_br\_csp, 0, UB) \text{ :-}\\
&\quad bound(\pi, rev\_br, \_RLB, RUB),\\
&\quad bound(\pi, tra\_br, \_TLB, TUB),\\
&\quad min(RUB, TUB, UB).\\
&bound(\pi, r\_t\_bc\_csp, 0, UB) \text{ :-}\\
&\quad bound(\pi, rev\_br, \_RLB, RUB),\\
&\quad bound(\pi, tra\_cg, \_TLB, TUB),\\
&\quad min(RUB, TUB, UB).\\
&bound(\pi, r\_t\_cc\_csp, 0, UB) \text{ :-}\\
&\quad bound(\pi, rev\_cg, \_RLB, RUB),\\
&\quad bound(\pi, tra\_cg, \_TLB, TUB),\\
&\quad min(RUB, TUB, UB).
\end{aligned}
\tag{3.10}
$$

O predicado *bound/4* (3.7), (3.8), (3.9), (3.10) recebe na variável *Model* um átomo que representa o modelo a ser usado. Este átomo conecta-se com o predicado que retorna o limitante superior e inferior apropriado para o modelo. Observe que o limitante inferior é igual a 0 no caso dos modelos de ordenação por reversões e transposições. Isto ocorre devido ao fato que, a cada nova iteração do modelo, pode surgir um limitante inferior melhor, simplesmente fazendo a troca entre as operações de reversão e transposição.

3.1.2 Modelo COP

Uma outra alternativa é modelar o problema usando a teoria COP. Os modelos que usam esta abordagem necessitam de um limitante superior, portanto serão feitas algumas alterações nos predicados definidos anteriormente. Nós usamos uma lista de variáveis binárias B para indicar quando uma operação de reversão ou de transposição modificou ou não a permutação fornecida ($B_k = 1$ se a k-ésima operação modifica a permutação e $B_k = 0$ caso contrário.).

O primeiro predicado que precisamos criar para o evento de reversão é o *reversal_cop/5* (3.11). Primeiramente, dada uma reversão $\rho(i, j)$, adicionamos uma nova restrição para permitir $(i, j) = (0, 0)$. Se $(i, j) = (0, 0)$ então $\pi\rho = \pi$. Então, adicionamos um novo argumento ao predicado *reversal_cop* que recebe a variável B_k (como último parâmetro).

$$
\begin{aligned}
&reversal\_cop(\iota, \iota, 0, 0, 0).\\
&reversal\_cop(\pi, \sigma, I, J, 1) \text{ :- } reversal(\pi, \sigma, I, J).
\end{aligned}
\tag{3.11}
$$

O predicado equivalente para o evento de transposição é o *transposition_ cop/6* (3.12). Neste caso, dada uma transposição $\rho(i, j, k)$, adicionamos uma nova restrição para permitir $(i, j, k) = (0, 0, 0)$. Se $(i, j, k) = (0, 0, 0)$ então $\pi\rho = \pi$.

$$\begin{aligned} &\mathit{transposition\_cop}(\iota, \iota, 0, 0, 0, 0). \\ &\mathit{transposition\_cop}(\pi, \sigma, I, J, K, 1) \text{ :- } \mathit{transposition}(\pi, \sigma, I, J, K). \end{aligned} \tag{3.12}$$

Para calcular a distância de reversão nos modelos baseados na teoria COP, implementamos o predicado *reversal_ distance_ cop/3* (3.13), que ajusta as variáveis B_k usando o valor do limitante superior e restringe as permutações fazendo $\pi_k = \pi_{k-1}\rho_k$. O predicado *length/2*, predicado interno do Prolog, é usado para criar uma lista de variáveis não instanciadas com o tamanho dado. A função de custo *Cost* é a soma das variáveis B_k associadas com cada ρ_k, $Cost = \sum_{k=1}^{UB} B_k$, onde UB é um limitante superior conhecido. A distância de reversão é o valor mínimo da função de custo $d_r = \min Cost$. Para evitar processamentos desnecessários, o valor de $Cost$ precisa ser maior ou igual a qualquer limitante inferior. O predicado equivalente para o problema de ordenação por transposições é o *transposition_ distance_ cop/3* (3.14).

$$\begin{aligned} &\mathit{reversal\_distance\_cop}(\pi, R, Model) \text{ :-} \\ &\quad \mathit{bound}(\pi, Model, LB, UB), \\ &\quad \mathit{length}(B, UB), \\ &\quad \mathit{upperbound\_constraint\_rev}(\pi, B, Model, UB), \\ &\quad \mathit{sum}(B, Cost), \\ &\quad Cost \geq LB, \\ &\quad \mathit{minimize}(Cost, R). \end{aligned} \tag{3.13}$$

$$\begin{aligned} &\mathit{transposition\_distance\_cop}(\pi, T, Model) \text{ :-} \\ &\quad \mathit{bound}(\pi, Model, LB, UB), \\ &\quad \mathit{length}(B, UB), \\ &\quad \mathit{upperbound\_constraint\_trans}(\pi, B, Model, UB), \\ &\quad \mathit{sum}(B, Cost), \\ &\quad Cost \geq LB, \\ &\quad \mathit{minimize}(Cost, T). \end{aligned} \tag{3.14}$$

O predicado equivalente para o modelo de ordenação por reversões e transposições é o *rev_ trans_ dist_ cop/3* (3.15). O predicado *upperbound_ constraint_ event/4* escolhe o melhor evento entre a reversão, usando o predicado *upperbound_ constraint_ rev/4* (3.16),

e a transposição, usando o predicado *upperbound_constraint_trans/4* (3.17), para minimizar o valor da distância. Como o modelo é baseado na teoria COP, ele irá avaliar os dois casos e escolher qual deles retorna o menor valor para a distância.

$$\begin{aligned}
&rev_trans_dist_cop(\pi, N, Model) :- \\
&\quad bound(\pi, Model, LB, UB), \\
&\quad length(B, UB), \\
&\quad upperbound_constraint_event(\pi, B, Model, UB), \\
&\quad sum(B, Cost), \\
&\quad Cost \geq LB, \\
&\quad minimize(Cost, N).
\end{aligned} \tag{3.15}$$

O predicado *upperbound_constraint_rev/4* (3.16) aplica na permutação os efeitos de ρ_k e retorna o valor apropriado de B para cada reversão ρ_k. Uma restrição importante, $UB \geq LB$, é verificar se é possível ordenar a permutação usando o número restante de reversões para evitar processamento desnecessário. O predicado equivalente para o evento de transposição é o *upperbound_constraint_trans/4* (3.17).

$$\begin{aligned}
&upperbound_constraint_rev(\iota, [\,], _Model, _UB). \\
&upperbound_constraint_rev(\pi, [B|Bt], Model, UB) :- \\
&\quad reversal_cop(\pi, \sigma, _I, _J, B), \\
&\quad bound(\pi, Model, LB, _UB), \\
&\quad UB \geq LB, \\
&\quad upperbound_constraint_rev(\sigma, Bt, Model, UB - 1).
\end{aligned} \tag{3.16}$$

$$\begin{aligned}
&upperbound_constraint_trans(\iota, [\,], _Model, _UB). \\
&upperbound_constraint_trans(\pi, [B|Bt], Model, UB) :- \\
&\quad transposition_cop(\pi, \sigma, _I, _J, _K, B), \\
&\quad bound(\pi, Model, LB, _UB), \\
&\quad UB \geq LB, \\
&\quad upperbound_constraint_trans(\sigma, Bt, Model, UB - 1).
\end{aligned} \tag{3.17}$$

Os modelos baseados na teoria COP possuem a estrutura acima, trocando apenas os limitantes usados. Os limitantes são os mesmos usados para os modelos CSP, modificados para os modelos COP. Então temos os seguintes limitantes: *def_cop*, *rev_br_cop*, *rev_cg_cop*, *tra_br_cop*, *tra_cg_cop*, *r_t_br_cop*, *r_t_bc_cop* e *r_t_cc_cop*.

3.2 Programação Linear Inteira

A abordagem utilizada para programação linear inteira é a descrita no trabalho de Dias e de Souza [17]. O modelo é específico para os eventos de reversão, transposição ou reversão e transposição quando ocorrem simultaneamente. Uma característica importante do modelo é o seu tamanho polinomial de variáveis e de restrições em relação ao tamanho da permutação fornecida como entrada.

Primeiramente vamos apresentar as variáveis e restrições que são comuns para todos os modelos. A ideia é assegurar que só estamos tratando com permutações válidas.

Gerando permutações válidas a cada iteração. As variáveis B_{ijk} indicam se a i-ésima posição de π possui o valor j depois da k-ésima operação ter sido executada, para todo $1 \leq i, j \leq n$ e todo $0 \leq k < n$.

$$B_{ijk} = \begin{cases} 1, & \text{se } \pi[i] = j \text{ depois da } k\text{-ésima operação} \\ 0, & \text{caso contrário} \end{cases}$$

As restrições (3.18) e (3.19) garantem que a permutação inicial e a final são corretas.

$$B_{i,\pi[i],0} = 1, \text{ para todo } 1 \leq i \leq n. \tag{3.18}$$

$$B_{i,\sigma[i],n-1} = 1, \text{ para todo } 1 \leq i \leq n. \tag{3.19}$$

A restrição (3.20) garante que cada posição de uma permutação possui exatamente um valor associado a ela. Já a restrição (3.21) garante que todo valor esteja associado a uma posição de cada permutação.

$$\sum_{j=1}^{n} B_{ijk} = 1, \text{ para todo } 1 \leq i \leq n,\ 0 \leq k < n. \tag{3.20}$$

$$\sum_{i=1}^{n} B_{ijk} = 1, \text{ para todo } 1 \leq j \leq n,\ 0 \leq k < n. \tag{3.21}$$

Distância de reversão. Para o problema da distância de reversão definimos os seguintes conjuntos de variáveis e restrições. As variáveis binárias r_{abk} indicam quando a k-ésima operação de reversão afeta o blocos $\pi[a \,..\, b]$ de π, para todo $1 \leq a < b \leq n$ e todo $1 \leq k < n$.

$$r_{abk} = \begin{cases} 1, & \text{se } \rho_k = \rho(a, b) \\ 0, & \text{caso contrário} \end{cases}$$

As variáveis binárias r_k são usadas para decidir se a k-ésima operação de reversão modificou a permutação, para todo $1 \leq k < n$.

$$r_k = \begin{cases} 1, & \text{se } \rho_k = \rho(x,y) \text{ e } \pi\rho_k\rho_{k-1} \ \ldots \ \rho_1 \neq \pi\rho_{k-1} \ \ldots \ \rho_1 \\ 0, & \text{caso contrário} \end{cases}$$

As restrições (3.22) e (3.23) são necessárias para identificar as reversões que fazem parte da solução. A restrição (3.22) garante que se a k-ésima reversão não alterar a permutação, nenhuma das reversões seguintes poderá alterar. Já a restrição (3.23) garante que no máximo uma reversão poderá ser feita por iteração.

$$r_k \leq r_{k-1}, \text{ para todo } 1 \leq k < n. \tag{3.22}$$

$$\sum_{a=1}^{n-1} \sum_{b=a+1}^{n} r_{abk} \leq r_k, \text{ para todo } 1 \leq k < n. \tag{3.23}$$

As próximas restrições lidam com as modificações na permutação causadas pela reversão a cada iteração da execução. A análise será dividida em dois casos onde, para cada caso, analisamos cada posição i da permutação para verificar seu valor após a operação de reversão $\rho(a,b)$ ser completada.

1. $i < a$ ou $i > b$: A operação de reversão não modifica estas posições.

$$\sum_{a=i+1}^{n-1} \sum_{b=a+1}^{n} r_{abk} + \sum_{a=1}^{n-1} \sum_{b=a+1}^{i-1} r_{abk} + (1 - r_k) + B_{i,j,k-1} - B_{ijk} \leq 1, \quad \text{para todo } 1 \leq i, \ j \leq n \text{ e todo } 1 \leq k < n. \tag{3.24}$$

2. $a \leq i \leq b$: A operação de reversão altera os elementos armazenados nestas posições. Para não ser redundante, a desigualdade precisa ter os dois primeiros termos com valor 1. Neste caso, $B_{ijk} = 1$, implica que o elemento j foi salvo na posição $b+a-i$.

$$r_{abk} + B_{b+a-i,j,k-1} - B_{ijk} \leq 1, \quad 1 \leq a < b \leq n, \ a \leq i \leq b, \ 1 \leq j \leq n, \ 1 \leq k < n. \tag{3.25}$$

Distância de transposição. Para o problema da distância de transposição, usaremos os seguintes conjuntos de variáveis e restrições. As variáveis binárias t_{abck} indicam quando a k-ésima operação de transposição realiza a troca de lugares dos blocos $\pi[a \,..\, b-1]$ e $\pi[b \,..\, c-1]$ da permutação π, para todo $1 \leq a < b < c \leq n+1$ e todo $1 \leq k < n$.

$$t_{abck} = \begin{cases} 1, & \text{se } \rho_k = \rho(a,b,c) \\ 0, & \text{caso contrário} \end{cases}$$

As variáveis binárias t_k são usadas para decidir se a k-ésima operação de transposição modificou a permutação, para todo $1 \leq k < n$.

$$t_k = \begin{cases} 1, & \text{se } \rho_k = \rho(x,y,z) \text{ e } \pi\rho_k\rho_{k-1} \ \ldots \ \rho_1 \neq \pi\rho_{k-1} \ \ldots \ \rho_1 \\ 0, & \text{caso contrário} \end{cases}$$

As restrições (3.26) e (3.27) são necessárias para identificar as transposições que fazem parte da solução. A restrição (3.26) garante que se a k-ésima transposição não alterar a permutação, nenhuma das transposições seguintes poderá alterar. Já a restrição (3.27) garante que no máximo uma transposição poderá ser feita por iteração.

$$t_k \leq t_{k-1}, \text{para todo } 1 \leq k < n. \tag{3.26}$$

$$\sum_{a=1}^{n-1} \sum_{b=a+1}^{n} \sum_{c=b+1}^{n+1} t_{abck} \leq t_k, \text{para todo } 1 \leq k < n. \tag{3.27}$$

As próximas restrições refletem as modificações na permutação causadas por uma transposição a cada passo da execução. A análise será dividida em três casos onde, para cada caso, analisamos cada posição i da permutação para verificar seu valor após a operação de transposição $\rho(a,b,c)$ ser completada.

1. $i < a$ ou $i \geq c$: A operação de transposição não altera estas posições.

$$\sum_{a=i+1}^{n-1} \sum_{b=a+1}^{n} \sum_{c=b+1}^{n+1} t_{abck} + \sum_{a=1}^{n-1} \sum_{b=a+1}^{n} \sum_{c=b+1}^{i} t_{abck} + (1 - t_k) + B_{i,j,k-1} - B_{ijk} \leq 1, \quad \text{para todo } 1 \leq i,j \leq n \text{ e todo } 1 \leq k < n. \tag{3.28}$$

2. $a \leq i < a + c - b$: Após a operação de transposição ser completada, estas posições serão ocupadas pelos elementos que estavam nas posições de b a $c-1$. Para não ser redundante, esta desigualdade precisa ter os dois primeiros termos com o valor 1. Neste caso, temos que $B_{ijk} = 1$, implicando que o elemento j foi salvo na posição $b - a + i$.

$$t_{abck} + B_{b-a+i,j,k-1} - B_{ijk} \leq 1, \quad 1 \leq a < b < c \leq n+1, a \leq i < a + c - b, 1 \leq j \leq n, 1 \leq k < n. \tag{3.29}$$

3. $a + c - b \leq i < c$: Após a operação de transposição ser completada, estas posições serão ocupadas pelos elementos que estavam nas posições de a a $b-1$. Similarmente ao caso anterior, esta desigualdade é redundante se os valores dos dois primeiros termos não forem iguais a 1. Isto significa que a k-ésima transposição move $B^{k-1}[a \,..\, b-1]$ para as posições que precedem a posição c. Por definição, i representa uma das posições que receberão um elemento deste subvetor. Então,

temos que $B^k[i] = B^{k-1}[b-c+i]$, para todo $i \in [a+c-b \,..\, c-1]$ e os últimos dois termos se anulam.

$$\begin{aligned} t_{abck} + B_{b-c+i,j,k-1} - B_{ijk} \leq 1, \\ 1 \leq a < b < c \leq n+1, a+c-b \leq i < c, 1 \leq j \leq n, 1 \leq k < n. \end{aligned} \tag{3.30}$$

Distância de reversão e transposição. Para o problema da distância de reversão e transposição usaremos todas as variáveis definidas anteriormente, com a adição das variáveis binárias z_k, que é usada para indicar quando uma k-ésima operação, seja ela uma reversão ou uma transposição, realmente modificou a permutação. Então, para todo $1 \leq k < n$, temos que:

$$z_k = \begin{cases} 1, & \text{se } \rho_k = \rho(x,y) \text{ ou } \rho_k = \rho(x,y,z) \text{ e } \rho_k\rho_{k-1} \ \ldots \ \rho_1\pi \neq \rho_{k-1} \ \ldots \ \rho_1\pi \\ 0, & \text{caso contrário} \end{cases}$$

Usaremos todas as restrições definidas anteriormente, com exceção das restrições (3.22) e (3.26) que serão substituídas pelas restrições (3.31) e (3.32). A restrição (3.31) garante que se não ocorreu nenhuma operação em uma iteração então, não ocorrerá nenhuma operação nas iterações seguintes. A restrição (3.32) garante que no máximo uma operação é executada a cada iteração.

$$z_k \leq z_{k-1}, \text{para todo } 1 \leq k < n. \tag{3.31}$$

$$r_k + t_k = z_k, \text{para todo } 1 \leq k < n. \tag{3.32}$$

Precisamos modificar as restrições (3.24) e (3.28), substituindo r_k e t_k por z_k, resultando nas restrições (3.33) e (3.34).

$$\begin{aligned} \sum_{a=i+1}^{n-1} \sum_{b=a+1}^{n} r_{abk} + \sum_{a=1}^{n-1} \sum_{b=a+1}^{i-1} r_{abk} + (1 - z_k) + B_{i,j,k-1} - B_{ijk} \leq 1, \\ \text{para todo } 1 \leq i,\ j \leq n \text{ e todo } 1 \leq k < n. \end{aligned} \tag{3.33}$$

$$\begin{aligned} \sum_{a=i+1}^{n-1} \sum_{b=a+1}^{n} \sum_{c=b+1}^{n+1} t_{abck} + \sum_{a=1}^{n-1} \sum_{b=a+1}^{n} \sum_{c=b+1}^{i} t_{abck} + (1 - z_k) + B_{i,j,k-1} - B_{ijk} \leq 1, \\ \text{para todo } 1 \leq i, j \leq n \text{ e todo } 1 \leq k < n. \end{aligned} \tag{3.34}$$

3.2.1 Função Objetivo

Considerando as variáveis e restrições descritas anteriormente para cada um dos três problemas de distâncias, temos a função objetivo $\omega_r = \min \sum_{k=1}^{n-1} r_k$, para o problema da distância de reversão, a função objetivo $\omega_t = \min \sum_{k=1}^{n-1} t_k$, para o problema da distância de transposição, e a função objetivo $\omega_{rt} = \min \sum_{k=1}^{n-1} z_k$, para o problema da distância de reversão e transposição quando ocorrem simultaneamente.

3.2.2 Tamanho do modelo

É fácil observar que o modelo descrito possui tamanho polinomial em relação ao tamanho da permutação fornecida como entrada. A Tabela 3.1 mostra o tamanho do modelo para os três problemas de distâncias com relação ao parâmetro n (tamanho da permutação de entrada).

Tabela 3.1: Tamanho dos modelos em relação à n.

Modelo	**Variáveis**	**Restrições**
Distância de Reversão	$O(n^3)$	$O(n^5)$
Distância de Transposição	$O(n^4)$	$O(n^6)$
Distância de Reversão e Transposição	$O(n^4)$	$O(n^6)$

Capítulo 4

Análise dos Resultados

Neste capítulo apresentaremos os resultados obtidos pelos modelos descritos no Capítulo 3. A Seção 4.1 mostra as características do computador utilizado para executar os testes. A Seção 4.2 descreve como os testes foram executados. A Seção 4.3 apresenta a análise dos resultados obtidos durante este trabalho.

4.1 Especificações Técnicas

O computador[1] utilizado para executar os testes possui as seguintes características:

- Processador: Intel® Core™ 2 Duo 2.33GHz.
- Memória RAM: 3 GB.
- Sistema Operacional: Ubuntu Linux com kernel 2.6.31.
- Compilador para linguagem C++: gcc 4.4.3 [3].

Todos os modelos de programação por restrições foram implementados usando as seguintes ferramentas:

- Sistema de programação de código aberto *ECLiPSe-6.0* [1], usando a linguagem própria, baseada no prolog.
- O código foi escrito na linguagem C++, usando o pacote proprietário *IBM® ILOG® CPLEX® CP Optimizer v 2.3*[2] [4].

[1] Computador disponível com as licenças dos softwares proprietários.

[2] Quando usamos o termo ILOG CP, estamos falando sobre este pacote de programação por restrições.

Todas as formulações de programação linear inteira foram implementadas usando as seguintes ferramentas:

- Sistema de programação de código aberto *GLPK-4.35* [2], usando a linguagem de modelagem *GNU MathProg*.
- O código foi escrito na linguagem C++, usando o pacote proprietário *IBM® ILOG® CPLEX® Optimizer v 12.1*[3] [5].

4.2 Descrição dos Testes

Os testes foram separados de acordo com o tamanho das permutações. Uma instância contém um conjunto de permutações com tamanho n, onde $n > 2$ devido ao fato de ser trivial ordenar uma permutação com tamanho 2. Para cada instância, geramos 50 permutações aleatórias com tamanho n.

Todas as instâncias foram executadas nos softwares indicados na Seção 4.1. Para cada instância foi dado o tempo máximo de 25 horas, decidido após alguns testes iniciais, onde foi possível observar que havia um número pequeno de instâncias que não eram solucionadas devido ao limite de memória do sistema. Os modelos de programação por restrições e as formulações de programação linear inteira produzem o resultado exato para a distância escolhida. Fazemos a comparação dos modelos baseando nos tempos médios usados para resolver cada instância. Como referência usamos os modelos de programação linear inteira descritos na Seção 3.2.

4.3 Análise dos Resultados

As tabelas 4.1, 4.2, 4.3 apresentam os tempos médios usados para resolver cada instância dos testes. O caractere "-" significa que o modelo não conseguiu solucionar todas as permutações da instância dentro do limite de 25 horas. O caractere "*" significa que o modelo não conseguiu terminar devido ao limite de memória do sistema.

Podemos observar nos três casos que os modelos de programação por restrições baseados na teoria COP possuem os piores tempos de execução e os modelos baseados na teoria CSP possuem os melhores resultados.

O modelo baseado na teoria COP tem como objetivo otimizar o resultado do problema. Seu mecanismo de busca consiste em encontrar uma solução base para depois encontrar uma solução melhor, usando um valor melhor para a função de custo. Com isso, ele acaba

[3]Quando usamos o termo ILOG CPLEX, estamos falando sobre este pacote de programação linear.

gerando um espaço de busca maior do que o modelo correspondente baseado na teoria CSP, que usa uma estratégia *bottom-up*.

Também podemos notar que, quanto melhor os limitantes, menor é o tempo necessário para solucionar as instâncias, conseguindo resolver mais instâncias com permutações "maiores"[4]. Isto ocorre pela redução do conjunto das possíveis soluções do problema.

A seguir faremos a análise separadamente para cada caso. A Seção 4.3.1 contém os resultados para o problema de ordenação por reversões. A Seção 4.3.2 contém os resultados para o problema de ordenação por transposições. A Seção 4.3.3 contém os resultados para o problema de ordenação por reversões e transposições. A Seção 4.3.4 apresenta a comparação das ferramentas utilizadas nos testes.

4.3.1 Ordenação por Reversões

Nos modelos de ordenação por reversões, Tabela 4.1, podemos notar que alguns modelos não conseguiram solucionar as permutações devido ao limite de memória do sistema. Isto ocorreu com as permutações de tamanho $n = 10$ usando os três limitantes nos modelos baseados na teoria CSP desenvolvido para o *ILOG CP*, com as permutações de tamanho $n = 13$ para o limitante *rev_ cg_ csp* no modelo baseado na teoria CSP e com as permutações de tamanho $n = 6$ para o limitante *rev_ cg_ cop* no modelo baseado na teoria COP desenvolvidos para o *ECLiPSe*.

No *ECLiPSe*, é possível observar que nos modelos baseados na teoria COP, quanto melhor o limitante, maior é o tempo necessário para resolver as instâncias. O principal motivo é a complexidade existente para encontrar os melhores limitantes, juntamente com o aumento do espaço de busca gerado pelo modelo COP.

A diferença na complexidade pode ser observada quando comparamos os resultados de ordenação por reversões com os resultados de ordenação por transposições (Tabela 4.2). É possível notar que, nos resultados dos limitantes triviais[5], colunas *def_ cop* e *def_ csp*, e dos limitantes no grafo de *breakpoints* para reversões, colunas *rev_ br_ cop* e *rev_ br_ csp* e dos limitantes que usam *breakpoints* para transposições, colunas *tra_ br_ cop* e *tra_ br_ csp*, os tempos dos modelos de ordenação por reversões é melhor, em relação aos modelos de transposições. Isto ocorre, devido à procura dos blocos na permutação, que irá sofrer o evento escolhido. A reversão irá alterar apenas um bloco, ou seja, necessita apenas escolher duas posições na permutação. No caso da transposição, o evento trocará dois blocos adjacentes de lugar, necessitando escolher três posições na permutação. Logo o

[4]Nenhum modelo conseguiu resolver instâncias com permutações de tamanho $n > 14$, dentro do tempo limite de 25 horas.

[5]Limitante inferior igual à 0 e limitante superior igual ao tamanho da permutação fornecida como entrada.

espaço de busca dos modelos de ordenação por transposição é maior do que o espaço dos modelos de ordenação por reversões.

Entretanto, se focarmos apenas nos melhores limitantes dos dois tipos de ordenação, colunas *rev_ cg_ cop*, *rev_ cg_ csp*, *tra_ cg_ cop* e *tra_ cg_ csp*, ocorre justamente o inverso. Este comportamento pode ser explicado usando a decomposição em ciclos. Na transposição a decomposição é única, para todo vértice do grafo de ciclos, toda aresta chegando é unicamente pareada com uma aresta saindo de cor diferente, então basta encontrar o número de ciclos ímpares. No caso da reversão, todo vértice possui o mesmo número de arestas incidentes cinzas e pretas no grafo de *breakpoints*. Logo existem diversas maneiras para realizar a decomposição em ciclos. Como o modelo usa os limitantes definidos por Christie [15], é necessário mais processamento para encontrar a decomposição máxima em 2-ciclos, aumentando o espaço de busca dos modelos de ordenação por reversões.

A diferença na complexidade pode ser notado, também, nos modelos baseados na teoria CSP desenvolvidos para o *ILOG CP*. Isto pode ser causado por dois motivos:

1. Forma da modelagem: Como o modelo foi pensado para o *ECLiPSe*, o modelo não aproveita características específicas do *ILOG CP* que poderiam melhorar sua performance.

2. Mecanismo de *backtracking*[6]: O *ECLiPSe* usa uma linguagem baseada no Prolog, que possui o paradigma de programação lógica. Uma das suas características é ter o mecanismo de *backtracking* embutido na linguagem. No caso do *ILOG CP*, o modelo foi escrito usando a linguagem C++, que é uma linguagem orientada à objetos e não tem o mecanismo de *backtracking* por padrão. Portanto, o *ECLiPSe* realiza o *backtracking* de forma "mais natural" do que o *ILOG CP*.

Nenhum modelo de ordenação por reversões conseguiu solucionar as instâncias com permutações de tamanho $n > 13$ dentro do tempo limite de 25 horas.

4.3.2 Ordenação por Transposições

Nos modelos de ordenação por transposições, Tabela 4.2, os modelos baseados na teoria COP que usam o limitante *tra_ br_ cop* apresentaram os piores resultados, não apresentando nenhuma vantagem em relação ao modelo que usa limitantes triviais *def_ cop*.

No *ECLiPSe*, é possível observar que, diferentemente do modelo de ordenação por reversões, nos modelos baseados na teoria COP, o melhor limitante, *tra_ cg_ cop*, obteve o melhor tempo de execução. Este limitante conseguiu reduzir o espaço de busca por ser mais preciso que os outros. Neste caso, a redução do espaço de busca supriu a necessidade

[6]Algoritmo para encontrar soluções para um problema computacional. Ele pode eliminar múltiplas soluções apenas se decidir que elas não são viáveis para o problema.

da quantidade de processamento para encontrar o número de ciclos ímpares no grafo de ciclos da ordenação por transposições, ao contrário do modelo de ordenação por reversões que usa o limitante *rev_cg_cop*. Em relação aos modelos baseados na teoria CSP, o modelo de ordenação por transposições que usa o limitante *tra_cg_csp* obteve tempos de execução melhores que o modelo de ordenação por reversões que usa o limitante *rev_cg_csp*.

Similar ao modelo de ordenação por reversões, nos modelos baseados na teoria CSP desenvolvidos para o *ILOG CP*, quanto melhor o limitante, maior é o tempo necessário para resolver as instâncias.

Nenhum modelo de ordenação por transposições conseguiu solucionar as instâncias com permutações de tamanho $n > 14$ dentro do tempo limite de 25 horas.

4.3.3 Ordenação por Reversões e Transposições

Os modelos de ordenação por reversões e transposições, Tabela 4.3, obtiveram os piores resultados. Isto já era esperado, devido ao fato que os modelos utilizam tanto a operação de reversão como a operação de transposição, resultando em um espaço de busca maior e no aumento do tempo necessário para encontrar a solução.

O modelo baseado na teoria CSP que usa o limitante *r_t_cc_csp* desenvolvido para o *ECLiPSe* não conseguiu resolver a instância com permutações de tamanho $n = 7$ devido ao limite de memória do sistema. Foi o único modelo de ordenação por reversões e transposições que ocorreu este problema.

É possível notar claramente o rápido crescimento do espaço de busca dos modelos conforme o tamanho das permutações. Como exemplo, podemos pegar os modelos baseados na teoria CSP desenvolvidos para o *ILOG CP*. O modelo que obteve o pior tempo para a instância com permutações de tamanho $n = 10$ foi o modelo *def_csp*, que não utiliza limitantes, sendo o tempo de execução para essa instância 0.012 segundos. Um tempo excelente para o modelo que, diferentemente dos modelos que utilizam as operações isoladamente, conseguiu solucionar esta instância. Porém para as instâncias com permutações de tamanho $n > 10$, os modelos não conseguiram solucionar todas permutações dentro do limite de tempo.

Nenhum modelo de ordenação por reversões e transposições conseguiu solucionar as instâncias com permutações de tamanho $n > 10$ dentro do tempo limite de 25 horas.

4.3.4 Comparação das ferramentas

Um dos objetivos deste trabalho é analisar se os softwares proprietários são inferiores ou superiores aos softwares de código aberto. Usaremos a Tabela 4.4 para analisar o número de instâncias resolvidas por cada ferramenta utilizada.

No caso das formulações em programação linear inteira, podemos notar que as formulações desenvolvidas para o *ILOG CPLEX* obtiveram os melhores tempos para as instâncias com permutações de tamanho $n < 7$, e as formulações desenvolvidas para o *GLPK* foram mais rápidas nas permutações de tamanho $n = 7$, para todas formulações, e $n = 8$, no caso de ordenação por reversões (Tabela 4.1) mas não conseguiu resolver instâncias com permutações de tamanho $n > 8$, ao contrário do *ILOG CPLEX*. Se analisamos somente o número de instâncias resolvidas (Tabela 4.4) podemos afirmar que *ILOG CPLEX* é o mais adequado para os problemas de ordenação, devido ao fato de resolver uma instância a mais em relação ao *GLPK*. Apesar do pequeno número de instâncias resolvidas pelas formulações de programação linear inteira, podemos notar que escrever as formulações usando o pacote do *ILOG CPLEX* é mais adequado para os três problemas de ordenação.

No caso dos modelos de programação por restrições, iremos dividir a análise em dois casos. Nos modelos baseados na teoria COP, podemos notar que, nos três casos, o *ILOG CP* foi superior ao *ECLiPSe*, tanto nos tempos de execução, quanto no número de instâncias resolvidas, sendo 70 instâncias resolvidas pelo *ILOG CP* e 30 instâncias resolvidas pelo *ECLiPSe*. Então, para os modelos baseados na teria COP, podemos afirmar que o pacote do *ILOG CP* é o mais adequado para os problemas de ordenação.

Para os modelos baseados na teoria CSP, o *ECLiPSe* não só obteve tempos melhores, como também resolveu instâncias maiores. No caso do problema de ordenação por reversões, o *ECLiPSe* resolveu 24 instâncias com permutações de tamanho até $n = 12$, enquanto que o *ILOG CP* resolveu 21 instâncias com permutações de tamanho até $n = 9$. No caso do problema de ordenação por transposições, o *ECLiPSe* resolveu 24 instâncias com permutações de tamanho até $n = 14$, enquanto que o *ILOG CP* resolveu 21 instâncias com permutações de tamanho até $n = 9$. No caso do problema de ordenação por reversões e transposições, o *ILOG CP* foi melhor, resolvendo 32 instâncias com permutações de tamanho até $n = 10$, enquanto o *ECLiPSe* resolveu 19 instâncias com permutações de tamanho até $n = 7$. Então, para o problema de ordenação por reversões e transposições, o *ILOG CP* é o mais adequado, mas para os outros problemas, podemos afirmar que o *ECLiPSe* é o mais adequado.

Tabela 4.1: Tempo médio (em segundos) para o modelo de ordernação por reversões. O caractere "-" significa que o modelo não conseguiu terminar o conjunto de testes dentro do limite de 25 horas. O caractere "*" significa que o modelo não conseguiu terminar devido ao limite de memória do sistema. Os softwares de código aberto são: *ECLiPSe* para programação por restrições e *GLPK* para programação linear inteira. Os softwares proprietários são: *ILOG CP* para programação por restrições e *ILOG CPLEX* para programação linear inteira. Os limitantes usados para a teoria CSP, coluna *CSP*, são: *def_csp*, não usa nenhum limitante, *rev_br_csp* usa os limitantes descritos no Teorema 2.1 e *rev_cg_csp* usa os limitantes descritos no Teorema 2.2. Os limitantes equivalentes para a teoria COP, coluna *COP*, são: *def_cop*, *rev_br_cop* e *rev_cg_cop*.

Modelos de Ordenação por Reversões														
size	**CP**												**ILP**	
	ECLiPSe						**ILOG CP**						**GLPK**	**ILOG CPLEX**
	COP			**CSP**			**COP**			**CSP**				
	def	**rev_br**	**rev_cg**	**def**	**rev_br**	**rev_cg**	**def**	**rev_br**	**rev_cg**	**def**	**rev_br**	**rev_cg**		
3	0.008	0.003	0.013	0.004	0.004	0.003	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
4	0.368	0.339	1.490	0.025	0.004	0.003	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.002
5	31.676	39.424	93.984	0.531	0.014	0.008	0.019	0.018	0.019	0.001	0.006	0.004	0.176	0.039
6	-	-	*	9.001	0.032	0.014	0.144	0.148	0.126	0.001	0.023	0.020	2.118	0.653
7	-	-	-	442.825	0.179	0.072	1.962	1.807	1.331	0.007	0.156	0.225	3.896	21.911
8	-	-	-	-	1.075	0.449	25.831	26.275	30.244	0.001	0.642	2.757	3.006	280.528
9	-	-	-	-	5.015	1.401	503.421	473.005	405.242	0.003	11.416	54.560	-	258.051
10	-	-	-	-	51.800	8.639	-	-	-	*	*	*	-	-
11	-	-	-	-	1664.691	194.266	-	-	-	-	-	-	-	-
12	-	-	-	-	-	671.979	-	-	-	-	-	-	-	-
13	-	-	-	-	-	*	-	-	-	-	-	-	-	-

Tabela 4.2: Tempo médio (em segundos) para o modelo de ordernação por transposições. O caractere "-" significa que o modelo não conseguiu terminar o conjunto de testes dentro do limite de 25 horas. Os softwares de código aberto são: *ECLiPSe* para programação por restrições e *GLPK* para programação linear inteira. Os softwares proprietários são: *ILOG CP* para programação por restrições e *ILOG CPLEX* para programação linear inteira. Os limitantes usados para a teoria CSP, coluna *CSP*, são: *def_csp*, não usa nenhum limitante, *tra_br_csp* usa os limitantes descritos no Teorema 2.3 e *tra_cg_csp* usa os limitantes descritos no Teorema 2.4. Os limitantes equivalentes para a teoria COP, coluna *COP*, são: *def_cop*, *tra_br_cop* e *tra_cg_cop*.

Modelos de Ordenação por Transposições														
size	**CP**												**ILP**	
	ECLiPSe						**ILOG CP**						**GLPK**	**ILOG CPLEX**
	COP			**CSP**			**COP**			**CSP**				
	def	**tra_br**	**tra_cg**	**def**	**tra_br**	**tra_cg**	**def**	**tra_br**	**tra_cg**	**def**	**tra_br**	**tra_cg**		
3	0.005	0.006	0.004	0.003	0.002	0.003	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
4	0.479	1.557	0.049	0.012	0.004	0.003	0.004	0.001	0.001	0.001	0.001	0.001	0.001	0.005
5	174.445	1275.804	2.922	0.318	0.020	0.005	0.023	0.009	0.004	0.001	0.008	0.008	0.376	0.054
6	-	-	33.311	15.957	0.185	0.009	0.321	0.117	0.072	0.001	0.035	0.041	3.948	2.308
7	-	-	-	394.574	0.519	0.015	1.747	1.313	0.321	0.005	0.112	0.173	5.558	66.497
8	-	-	-	-	7.254	0.069	29.714	45.806	27.883	0.002	0.651	5.895	-	-
9	-	-	-	-	99.152	0.365	641.954	838.988	320.615	0.004	2.285	53.673	-	-
10	-	-	-	-	-	5.162	-	-	-	-	-	-	-	-
11	-	-	-	-	-	33.122	-	-	-	-	-	-	-	-
12	-	-	-	-	-	43.893	-	-	-	-	-	-	-	-
13	-	-	-	-	-	521.840	-	-	-	-	-	-	-	-
14	-	-	-	-	-	1223.512	-	-	-	-	-	-	-	-
15	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Tabela 4.3: Tempo médio (em segundos) para o modelo de ordenação por reversões e transposições. O caractere "-" significa que o modelo não conseguiu terminar o conjunto de testes dentro do limite de 25 horas. O caractere "*" significa que o modelo não conseguir terminar devido ao limite de memória do sistema. Os softwares de código aberto são: *ECLiPSe* para programação por restrições e *GLPK* para programação linear inteira. Os softwares proprietários são: *ILOG CP* para programação por restrições e *ILOG CPLEX* para programação linear inteira. Os limitantes usados para a teoria CSP, coluna *CSP*, são: *def_csp*, não usa nenhum limitante, *r_t_br_csp* usa o melhor limitante superior entre o limitante de *breakpoints* para reversões e o limitante de *breakpoints* para transposições, *r_t_bc_csp* usa o melhor limitante superior entre o limitante de *breakpoints* para reversões e o limitante do grafo de ciclos para transposições e *r_t_cc_csp* usa melhor limitante superior entre o limitante do grafo de ciclos para reversões e o limitante do grafo de ciclos para transposições. Os limitantes equivalentes para a teoria COP, coluna *COP*, são: *def_cop*, *r_t_br_cop*, *r_t_bc_cop* e *r_t_cc_cop*.

Modelos de Ordenação por Reversões e Transposições																		
size	CP																ILP	
	ECLiPSe								ILOG CP								GLPK	ILOG CPLEX
	COP				CSP				COP				CSP					
	def	r_t_br	r_t_bc	r_t_cc	def	r_t_br	r_t_bc	r_t_cc	def	r_t_br	r_t_bc	r_t_cc	def	r_t_br	r_t_bc	r_t_cc		
3	0.035	0.010	0.004	0.006	0.004	0.005	0.004	0.008	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
4	6.434	10.333	0.248	0.687	0.022	0.026	0.035	0.081	0.002	0.002	0.002	0.003	0.001	0.001	0.001	0.001	0.008	0.008
5	-	-	30.824	64.170	0.379	0.414	0.680	2.066	0.019	0.020	0.022	0.023	0.001	0.001	0.001	0.001	0.466	0.059
6	-	-	519.803	-	11.566	12.776	21.264	89.760	0.340	0.327	0.319	0.319	0.001	0.001	0.001	0.001	4.642	1.374
7	-	-	-	-	400.904	441.946	792.422	*	2.052	2.062	2.066	2.088	0.004	0.004	0.002	0.004	5.094	82.241
8	-	-	-	-	-	-	-	-	11.894	11.727	12.367	12.369	0.002	0.004	0.004	0.004	-	-
9	-	-	-	-	-	-	-	-	310.012	304.331	331.275	328.956	0.006	0.006	0.004	0.005	-	-
10	-	-	-	-	-	-	-	-	-	-	-	-	0.012	0.010	0.010	0.011	-	-
11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Tabela 4.4: Quantidade de instâncias resolvidas por cada ferramenta utilizada.

Número de instâncias Resolvidas							
	ECLiPSe		**ILOG CP**		**GLPK**	**ILOG CPLEX**	**Total**
	COP	**CSP**	**COP**	**CSP**			
Reversões	9	24	21	21	6	7	88
Transposições	10	24	21	21	5	5	86
Reversões e Transposições	11	19	28	32	5	5	100
Total	30	67	70	74	16	17	274

Capítulo 5

Conclusões

Neste trabalho, nós criamos modelos de Programação por Restrições para ordenação por reversões e ordenação por reversões e transposições, seguindo a linha de pesquisa utilizada por Dias e Dias [16]. Nós apresentamos os modelos de Programação por Restrições que buscam os resultados exatos para os problemas de ordenação por reversões, ordenação por transposições e ordenação por reversões e transposições, baseados na teoria do Problema de Satisfação de Restrições e na teoria do Problema de Otimização com Restrições.

No Capítulo 2 apresentamos os conceitos usados nesta dissertação. Na Seção 2.1 mostramos as formalizações usadas pelos problemas de rearranjos de genomas. Nas seções 2.2, 2.3 e 2.4 descrevemos os problemas de ordenação por reversões, ordenação por transposições e ordenação por reversões e transposições, respectivamente. Na Seção 2.5 explicamos o conceito de Programação por Restrições. Os modelos de Programação por Restrições e as formulações de Programação Linear Inteira foram descritos no Capítulo 3.

No Capítulo 4 apresentamos os resultados obtidos. Nós fizemos comparações com os modelos de Programação por Restrições para ordenação por transposições, descrito por Dias e Dias [16], e com as formulações de Programação Linear Inteira que buscam os resultados exatos para os problemas de ordenação por reversões, ordenação por transposições e ordenação por reversões e transposições, descritos por Dias e Souza [17].

Os resultados foram analisados observando os tempos médios usados para resolver cada instância dos testes. Analisamos também qual ferramenta é a mais adequada para o problema, no caso de programação linear inteira usamos o *GLPK* e o *ILOG CPLEX*, e no caso de programação por restrições usamos o *ECLiPSe* e o *ILOG CP*, sendo que o *GLPK* e o *ECLiPSe* são softwares de código aberto, e o *ILOG CPLEX* e o *ILOG CP* são softwares proprietários.

Os resultados mostraram que os modelos de programação por restrições baseados na teoria CSP obtiveram os melhores tempos em relação às formulações de programação linear, já os modelos de programação por restrições baseados na teoria COP obtiveram os

piores tempos, no modelo de ordenação por reversões, e as formulações de programação linear inteira obtiveram os piores resultados nos outros problemas de ordenação.

O modelo de programação por restrições para o problema de ordenação por reversões e transposições foi apresentado no artigo "*Constraint Logic Programming Models for Reversal and Transposition Distance Problems*" [22], publicado no *VI Brazilian Symposium on Bioinformatics (BSB'2011)*, realizado em Brasília, DF em 2011.

Apesar de ser mais uma ferramenta para solucionar os problemas de ordenação por reversões, ordenação por transposições e ordenação por reversões e transposições, esta abordagem ainda é inviável na prática.

Para trabalhos futuros, heurísticas podem ser estudadas para melhorar o desempenho dos modelos de programação por restrições (observe que não foi utilizada nenhuma heurística na criação dos modelos). As heurísticas vão desde a escolha de qual permutação deve ser analisada primeiro até a análise de permutações que possuem alguma característica em comum. Em ambos os casos, o objetivo é reduzir o espaço de busca do problema.

Podemos melhorar a formulação de programação linear inteira, utilizando relaxação lagrangeana, ou escrever uma nova formulação sem preocupação com o seu tamanho para aplicar técnicas como geração de colunas e *branch-and-cut* [29, 34].

Apêndice A

Códigos Fontes

Este apêndice apresentará os códigos fontes utilizados nesta dissertação.

A.1 ECLiPSe

Nesta seção apresentaremos o código fonte desenvolvido para o sistema *ECLiPSe*. Dividimos por partes para simplificar o entendimento dos modelos.

O início do arquivo (A.1) contém predicados simples que facilitam a codificação dos predicados de ordenação.

Listing A.1: Predicados básicos

```
1  :- lib(ic).
2  :- lib(branch_and_bound).
3
4  % ordered/1: Verifica se a lista dada está ordenada
5  ordered([]).
6  ordered([H | T]) :-
7          ordered(H, T).
8  ordered(_,[]).
9  ordered(E, [H | T]) :-
10         E #=< H,
11         ordered(H, T).
12
13 % element_is_different/2: Verifica se o elemento X é
     diferente de todos os elementos da lista L=[H|T]
```

```
14 element_is_different(_, []).
15 element_is_different(X, [H|T]) :-
16         X \= H,
17         element_is_different(X, T).
18
19 % all_different/1: Verifica se a lista L=[H|T] possui
      elementos repetidos
20 all_different([]).
21 all_different([H|T]) :-
22         element_is_different(H, T),
23         all_different(T).
24
25 % is_identity/1: Verifica se a lista é a permutação
      identidade
26 is_identity(H) :-
27         ordered(H),
28         all_different(H).
29
30 % in_range/2: Verifica se todos elementos da lista L=[H|T]
      está dentro do intervalo M .. N
31 in_range(_, _, []).
32 in_range(M, N, [H|T]) :-
33         H :: M .. N,
34         in_range(M, N, T).
35
36 % split3/6: Divide a lista L=[H|T] em 3 sublistas
37 split3([H|T], I, J, [H|C1], C2, C3) :-
38         length([H|T], N),
39         I :: 0 .. N,
40         indomain(I),
41         J :: 0 .. N,
42         indomain(J),
43         I1 is I - 1,
44         J1 is J - 1,
45         split3(T, I1, J1, C1, C2, C3).
46 split3([H|T], 0, J, [], [H|C2], C3) :-
47         length([H|T], N),
48         J :: 0 .. N,
```

```
49          indomain(J),
50          J1 is J - 1,
51          split3(T, 0, J1, [], C2, C3).
52  split3(L, 0, 0, [], [], L).
53
54  % split4/8: Divide a lista L=[H|T] em 4 sublistas
55  split4([H|T], I, J, K, [H|C1], C2, C3, C4) :-
56          length([H|T], N),
57          I :: 0 .. N,
58          indomain(I),
59          J :: 0 .. N,
60          indomain(J),
61          K :: 0 .. N,
62          indomain(K),
63          I1 is I - 1,
64          J1 is J - 1,
65          K1 is K - 1,
66          split4(T, I1, J1, K1, C1, C2, C3, C4).
67  split4([H|T], 0, J, K, [], [H|C2], C3, C4) :-
68          length([H|T], N),
69          J :: 0 .. N,
70          indomain(J),
71          K :: 0 .. N,
72          indomain(K),
73          J1 is J - 1,
74          K1 is K - 1,
75          split4(T, 0, J1, K1, [], C2, C3, C4).
76  split4([H|T], 0, 0, K, [], [], [H|C3], C4) :-
77          length([H|T], N),
78          K :: 0 .. N,
79          indomain(K),
80          K1 is K - 1,
81          split4(T, 0, 0, K1, [], [], C3, C4).
82  split4(L, 0, 0, 0, [], [], [], L).
83
```

Os predicados a seguir (A.2) são usados para calcular os limitantes, usando as ferramentas para ordenação por reversões e ordenação por transposições citadas no Capítulo 2.

Listing A.2: Calculando os limitantes

```
84 %
85 % Breakpoints
86 %
87
88 % calc_breakpoint/3: Encontra o número de breakpoints dada
      uma permutação Pi, de acordo com o modelo escolhido
89 calc_breakpoint([],0, _) :- !.
90 calc_breakpoint(Pi, N, rev) :-
91         length(Pi, M),
92         calc_breakpoint_rev(0, Pi, N, M), !.
93 calc_breakpoint(Pi, N, trans) :-
94         length(Pi, M),
95         calc_breakpoint_trans(0, Pi, N, M), !.
96
97 % breakpoints para reversões
98 calc_breakpoint_rev(E, [], 0, L) :-
99         E >= L - 1,
100        E =< L + 1, !.
101 calc_breakpoint_rev(E, [], 1, L) :-
102        E =\= L - 1,
103        E =\= L + 1, !.
104 calc_breakpoint_rev(E, [H | T], N, L) :-
105        E >= H - 1,
106        E =< H + 1,
107        calc_breakpoint_rev(H, T, N, L), !.
108 calc_breakpoint_rev(E, [H | T], N, L) :-
109        E =\= H - 1,
110        E =\= H + 1,
111        calc_breakpoint_rev(H, T, N1, L),
112        N is N1 + 1, !.
113
114 % breakpoints para transposições
115 calc_breakpoint_trans(E, [], 0, L) :-
116        E =:= L - 1, !.
117 calc_breakpoint_trans(E, [], 1, L) :-
118        E =\= L - 1, !.
119 calc_breakpoint_trans(E, [H | T], N, L) :-
```

```
120         E =:= H - 1,
121         calc_breakpoint_trans(H, T, N, L), !.
122 calc_breakpoint_trans(E, [H | T], N, L) :-
123         E =\= H - 1,
124         calc_breakpoint_trans(H, T, N1, L),
125         N is N1 + 1, !.
126
127 %
128 % Grafo de ciclos para transposições
129 %
130
131 %black_edges_cg_trans/2: Cria o conjunto de arestas pretas
132 black_edges_cg_trans([A, B], [[B, A]]) :- !.
133 black_edges_cg_trans([A, B | T], [[B, A] | R]) :-
134         black_edges_cg_trans([B|T], R).
135
136 %gray_edges_cg_trans/2| Cria o conjunto de arestas cinzas
137 gray_edges_cg_trans([], N, I) :-
138         I > N, !.
139 gray_edges_cg_trans([[I, J]| T], N, I) :-
140         I =< N,
141         J is I + 1,
142         gray_edges_cg_trans(T, N, J).
143
144 %look_for_next/3: Procura a aresta vizinha a [U,V] no
    conjunto de arestas dados
145 look_for_next([_, V], [[V, J] | _], [I, J]) :-
146         I is V, !.
147 look_for_next([U, V], [[I, _] | T], E) :-
148         I =\= V,
149         look_for_next([U, V], T, E).
150 look_for_next(_, [], []).
151
152 %remove_edge/3: Remove a aresta do conjunto.
153 remove_edge(E, [E | T], T) :- !.
154 remove_edge(E, [H | T], [H | Ts]) :-
155         remove_edge(E, T, Ts), !.
156 remove_edge(_, [], []).
```

```
157
158 %create_cycle_trans/4: Encontra um ciclo começando por
       arestas cinzas. Usa os predicados create_cycle_gray/5 e
       create_cycle_black/5 para alternar as cores no ciclo
159 create_cycle_trans([H | TBlack], Gray, [H | T], N) :-
160         create_cycle_gray(H, TBlack, Gray, T, R),
161         N is R + 1, !.
162
163 %create_cycle_gray/5: Encontra a aresta cinza e segue para
       uma aresta preta
164 create_cycle_gray(H, Black, Gray, [E | T], N) :-
165         look_for_next(H, Gray, E),
166         length(E, NE),
167         NE > 0,
168         remove_edge(E, Gray, NGray),
169         create_cycle_black(E, Black, NGray, T, N), !.
170 create_cycle_gray(H, _, Gray, [], 0) :-
171         look_for_next(H, Gray, E),
172         length(E, NE),
173         NE =< 0, !.
174
175 %create_cycle_black/5: Encontra a aresta preta e segue para
       uma aresta cinza
176 create_cycle_black(H, Black, Gray, [E | T], N) :-
177         look_for_next(H, Black, E),
178         length(E, NE),
179         NE > 0,
180         remove_edge(E, Black, NBlack),
181         create_cycle_gray(E, NBlack, Gray, T, R),
182         N is R + 1, !.
183 create_cycle_black(H, Black, _, [], 0) :-
184         look_for_next(H, Black, E),
185         length(E, NE),
186         NE =< 0, !.
187
188 %remove_cycle/5: Efetua a remoção das arestas que pertencem
       ao ciclo. OBS: Dois ciclos não possuem arestas em comum
189 remove_cycle([H | TC], B, G, NBlack, NGray):-
```

```
190         remove_edge(H,B,NB),
191         remove_cycle_G(TC, NB, G, NBlack, NGray), !.
192 remove_cycle([], B, G, B, G).
193
194 remove_cycle_G([H | TC], B, G, NBlack, NGray) :-
195         remove_edge(H, G, NG),
196         remove_cycle(TC, B, NG, NBlack, NGray), !.
197 remove_cycle_G([], B, G, B, G).
198
199 %count_odd_cg_trans/3: Retorna o número de ciclos ímpares
      usando o conjunto de arestas pretas e cinzas
200 count_odd_cg_trans([], _, 0) :- !.
201 count_odd_cg_trans(_, [], 0) :- !.
202 count_odd_cg_trans(Black, Gray, C) :-
203         create_cycle_trans(Black, Gray, Cycle, N),
204         1 is N mod 2,
205         remove_cycle(Cycle, Black, Gray, NewBlack, NewGray),
206         count_odd_cg_trans(NewBlack, NewGray, R),
207         C is R + 1, !.
208 count_odd_cg_trans(Black, Gray, C) :-
209         create_cycle_trans(Black, Gray, Cycle, N),
210         0 is N mod 2,
211         remove_cycle(Cycle, Black, Gray, NewBlack, NewGray),
212         count_odd_cg_trans(NewBlack, NewGray, C), !.
213
214 %calc_odd_cycles_transposition/3: Cria a representação do
      grafo de ciclos para a permutação Pi e encontra a
      quantidade de ciclos ímpares
215 calc_odd_cycles_transposition(Pi, N, C) :-
216         extend_pi(Pi, EPi),
217         black_edges_cg_trans(EPi, Black),
218         gray_edges_cg_trans(Gray, N, 0),
219         count_odd_cg_trans(Black, Gray, C).
220
221 %
222 % Grafo de ciclos para reversões
223 %
224
```

```
225 %create_black_edges_cg_rev/2: Cria a lista de arestas pretas
       para o grafo de ciclos para reversões
226 create_black_edges_cg_rev([_], []).
227 create_black_edges_cg_rev([U, V| T], [[U, V] | RT]) :-
228         V =\= U + 1,
229         V =\= U - 1,
230         create_black_edges_cg_rev([V | T], RT), !.
231 create_black_edges_cg_rev([U, V| T], RT) :-
232         V =< U + 1,
233         V >= U - 1,
234         create_black_edges_cg_rev([V | T], RT), !.
235
236 %neighbors/4: Verifica se dois elementos são vizinhos em Pi
237 neighbors(_, _, [], 0) :- !.
238 neighbors(A, B, [A, B | _], 1) :- !.
239 neighbors(A, B, [B, A | _], 1) :- !.
240 neighbors(A, B, [_ | T], R) :-
241         neighbors(A, B, T, R), !.
242
243 %create_gray_edges_cg_rev_aux/3 e create_gray_edges_cg_rev/2:
       Cria a lista de arestas cinzas para o grafo de ciclos
       para reversões
244 %create_gray_edges_cg_rev_aux/3
245 create_gray_edges_cg_rev_aux(_, [_],[]).
246 create_gray_edges_cg_rev_aux(P, [U | T], [[U , N] | RT]) :-
247         N is U + 1,
248         neighbors(U, N, P, R),
249         R =:= 0,
250         create_gray_edges_cg_rev_aux(P, T, RT), !.
251 create_gray_edges_cg_rev_aux(P, [U | T], RT) :-
252         N is U + 1,
253         neighbors(U, N, P, R),
254         R =\= 0,
255         create_gray_edges_cg_rev_aux(P, T, RT), !.
256
257 %create_gray_edges_cg_rev/2
258 create_gray_edges_cg_rev([], []).
259 create_gray_edges_cg_rev(P, G) :-
```

```
260         create_gray_edges_cg_rev_aux(P, P, G).
261
262 %connect_edges/4: Verifica se uma aresta [E,F] conecta as
      arestas [A,B] e [C,D]
263 connect_edges([],_,_,0) :- !.
264 connect_edges(_,[],_,0) :- !.
265 connect_edges(_,_,[],0) :- !.
266 connect_edges([A, _], [C, _], [E, F], 1) :-
267         E =:= A,
268         F =:= C, !.
269 connect_edges([A, _], [C, _], [E, F], 1) :-
270         F =:= A,
271         E =:= C, !.
272 connect_edges([A, _], [_, D], [E, F], 1) :-
273         E =:= A,
274         F =:= D, !.
275 connect_edges([A, _], [_, D], [E, F], 1) :-
276         F =:= A,
277         E =:= D, !.
278 connect_edges([_, B], [C, _], [E, F], 1) :-
279         E =:= B,
280         F =:= C, !.
281 connect_edges([_, B], [C, _], [E, F], 1) :-
282         F =:= B,
283         E =:= C, !.
284 connect_edges([_, B], [_, D], [E, F], 1) :-
285         E =:= B,
286         F =:= D, !.
287 connect_edges([_, B], [_, D], [E, F], 1) :-
288         F =:= B,
289         E =:= D, !.
290 connect_edges(_, _, _, 0) :- !.
291
292 %neighbour_edges/3: Verifica se duas arestas são vizinhas
293 neighbour_edges(_, [], 0) :- !.
294 neighbour_edges([], _, 0) :- !.
295 neighbour_edges([A,_], [A,_], 1) :- !.
296 neighbour_edges([A,_], [_,A], 1) :- !.
```

```
297 neighbour_edges([_,A], [A,_], 1) :- !.
298 neighbour_edges([_,A], [_,A], 1) :- !.
299 neighbour_edges(_, _, 0) :- !.
300
301 %remove_neighbour_edges/3: Remove as arestas vizinhas
302 remove_neighbour_edges([], _, _) :- !.
303 remove_neighbour_edges(_, [], []) :- !.
304 remove_neighbour_edges(A, [B | GT], NG) :-
305         neighbour_edges(A, B, R),
306         R =:= 1,
307         remove_neighbour_edges(A, GT, NG), !.
308 remove_neighbour_edges(A, [B | GT], [B |NG]) :-
309         neighbour_edges(A, B, R),
310         R =:= 0,
311         remove_neighbour_edges(A, GT, NG), !.
312
313 %find_gray_edges/4: Encontra as arestas cinzas que ligam as
    arestas A e B
314 find_gray_edges([], [], _, []) :- !.
315 find_gray_edges(_, _, [], []) :- !.
316 find_gray_edges(A, B, [E | GT], [E | ET]) :-
317         connect_edges(A, B, E, R),
318         R =:= 1,
319         remove_neighbour_edges(E, GT, NGT),
320         find_gray_edges(A, B, NGT, ET), !.
321 find_gray_edges(A, B, [E | GT], ET) :-
322         connect_edges(A, B, E, R),
323         R =:= 0,
324         find_gray_edges(A, B, GT, ET), !.
325
326 %create_matching_graph_aux/6 e create_matching_graph/4: Criam
    um emparelhamento para encontrar a decomposição de ciclos
    do grafo de ciclos para reversões
327 %create_matching_graph_aux/6
328 create_matching_graph_aux(_ , _, [] , _  , [], []) :- !.
329 create_matching_graph_aux(_ , _, _  , [] , [], []) :- !.
330 create_matching_graph_aux([], _, _  , _  , [], []) :- !.
331 create_matching_graph_aux(U, [V | BT], B, G, [[U, V] | FT],
```

```
332                             [Edges | FGT]) :-
333         find_gray_edges(U, V, G, Edges),
334         length(Edges, N),
335         N =:= 2,
336         create_matching_graph_aux(U, BT, B, G, FT, FGT), !.
337 create_matching_graph_aux(U, [V | BT1], B, G, FT, FGT) :-
338         find_gray_edges(U, V, G, Edges),
339         length(Edges, N),
340         N =\= 2,
341         create_matching_graph_aux(U, BT1, B, G, FT, FGT), !.
342 create_matching_graph_aux(_, [], [H | BT], G, FT, FGT) :-
343         create_matching_graph_aux(H, BT, BT, G, FT, FGT), !.
344
345 %create_matching_graph/4
346 create_matching_graph(_, [], [], []).
347 create_matching_graph([], _, [], []).
348 create_matching_graph([H | BT], G, F, FG) :-
349         create_matching_graph_aux(H, BT, BT, G, F, FG), !.
350
351 %check_adjacency/3: Verifica se A possui adjacência com algum
    elemento da lista L=[B|T]
352 check_adjacency([], _, 0) :- !.
353 check_adjacency(_, [], 0) :- !.
354 check_adjacency(A, [B|_], 1) :-
355         neighbour_edges(A,B,R),
356         R =:= 1, !.
357 check_adjacency(A, [B|T], RES) :-
358         neighbour_edges(A,B,R),
359         R =:= 0,
360         check_adjacency(A, T, RES), !.
361
362 %maximum_cardinality_matching/3: Escolhe qual aresta faz
   parte parte do emparelhamento máximo
363 maximum_cardinality_matching([], [], _) :- !.
364 maximum_cardinality_matching([FH | FT], [-1| XT], []) :-
365         maximum_cardinality_matching(FT, XT, [FH]).
366 maximum_cardinality_matching([FH | FT], [-1| XT], M) :-
367         check_adjacency(FH, M, R),
```

```
368         R =:= 0,
369         maximum_cardinality_matching(FT, XT, [FH | M]).
370 maximum_cardinality_matching([FH | FT], [0| XT], M) :-
371         check_adjacency(FH, M, R),
372         R =:= 1,
373         maximum_cardinality_matching(FT, XT, M).
374
375 %mcm_res/3: Retorna a lista com as arestas que fazem parte da
        solução
376 mcm_res([], _, []) :- !.
377 mcm_res(_, [], []) :- !.
378 mcm_res([-1 | T], [H | FT], [H | MT]) :-
379         mcm_res(T, FT, MT), !.
380 mcm_res([0 | T], [_ | FT], M) :-
381         mcm_res(T, FT, M), !.
382
383 %find_maximum_cardinality_matching/4: Encontra o
       emparelhamento máximo. OBS: Usa a teoria COP para
       encontrar o emparelhamento máximo
384 find_maximum_cardinality_matching(F, FG, M, MG) :-
385         length(F, N),
386         length(X, N),
387         minimize((maximum_cardinality_matching(F, X, []),
388                   sum(X, Cost)), Cost),
389         mcm_res(X, F, M),
390         mcm_res(X, FG, MG).
391
392 %create_lg_aux/4: Cria a lista de ladder cycles (2-ciclos que
        compartilha arestas cinzas)
393 create_lg_aux(_, _, [], []) :- !.
394 create_lg_aux(U, [V | MGT], MG, [[U, V] | LE]) :-
395         neighbour_edges(U, V, R),
396         R =:= 1,
397         create_lg_aux(U, MGT, MG, LE), !.
398 create_lg_aux(U, [V | MGT], MG, LE) :-
399         neighbour_edges(U, V, R),
400         R =:= 0,
401         create_lg_aux(U, MGT, MG, LE), !.
```

```
402 create_lg_aux(_, [], [H | MGT], LE) :-
403         create_lg_aux(H, MGT, MGT, LE), !.
404
405 %join_ladder_aux/3 e join_ladder/2: Cria a lista dos vértices
        do conjunto de ladder cycles
406 %join_ladder_aux/3
407 join_ladder_aux(E, [], E) :- !.
408 join_ladder_aux(E, [H | T], R) :-
409         append(E, H, TEMP),
410         join_ladder_aux(TEMP, T, R), !.
411
412 %join_ladder/2
413 join_ladder([], []) :- !.
414 join_ladder([H | T], R) :-
415         join_ladder_aux(H, T, R), !.
416
417 %in_list/3: Verifica se um elemento já está na lista
418 in_list(_, [], 0) :- !.
419 in_list(H, [H | _], 1) :- !.
420 in_list(H, [_ | T], R) :-
421         in_list(H, T, R), !.
422
423 %remove_equals_aux/3 e remove_equals/2: Retira os elementos
       repetidos da lista
424 %remove_equals_aux/3
425 remove_equals_aux(E, [], [E]) :- !.
426 remove_equals_aux(E, [H | T], [E | R]) :-
427         in_list(E, [H | T], B),
428         B =:= 0,
429         remove_equals_aux(H, T, R), !.
430 remove_equals_aux(E, [H | T], R) :-
431         in_list(E, [H | T], B),
432         B =:= 1,
433         remove_equals_aux(H, T, R), !.
434
435 %remove_equals/2
436 remove_equals([], []) :- !.
437 remove_equals([H | T], R) :-
```

```
438         remove_equals_aux(H, T, R), !.
439
440 %vertex_in_ladder/2: Cria a lista com os vértices
      pertencentes aos ladder cycles (ladder vertices)
441 vertex_in_ladder([],[]).
442 vertex_in_ladder(Ladder, Vertex) :-
443         join_ladder(Ladder, List),
444         remove_equals(List, Vertex), !.
445
446 %find_independent_vertex_aux/3 e find_independent_vertex/2:
      Encontra os 2-ciclos independentes (Não possuem nenhuma
      aresta em comum com os outros ciclos) e os ladder vertices
447 %find_independent_vertex_aux/3
448 find_independent_vertex_aux([], _, []) :- !.
449 find_independent_vertex_aux([H | MGT], LE, [H | LI]) :-
450         in_list(H, LE, R),
451         R =:= 0,
452         find_independent_vertex_aux(MGT, LE, LI), !.
453 find_independent_vertex_aux([H | MGT], LE, LI) :-
454         in_list(H, LE, R),
455         R =:= 1,
456         find_independent_vertex_aux(MGT, LE, LI), !.
457
458 %find_independent_vertex/3
459 find_independent_vertex(A, [], A) :- !.
460 find_independent_vertex(MG, Ladder, LI) :-
461         vertex_in_ladder(Ladder, LVertex),
462         find_independent_vertex_aux(MG, LVertex, LI), !.
463
464 %create_ladder_graph/3: Cria o ladder graph usando o
      emparelhamento criado a partir do grafo de ciclos
465 create_ladder_graph([],[],[]) :- !.
466 create_ladder_graph([H | MGT], LEdges, LInd) :-
467         create_lg_aux(H, MGT, MGT, LEdges),
468         find_independent_vertex([H | MGT], LEdges, LInd), !.
469
470 %calc_2_cycle/3: Calcula o número de 2-ciclos no grafo de
      ciclos para reversões
```

```
471 calc_2_cycle([], [], 0).
472 calc_2_cycle(LE, LI, C) :-
473         length(LI, Z),
474         length(LE, Y),
475         C is Z + Y.
476
477 %find_2_cycle/2: Retorna o número de 2-ciclos no grafo de
      ciclos para reversões
478 find_2_cycle([], 0) :- !.
479 find_2_cycle(Vertex, C) :-
480         create_black_edges_cg_rev(Vertex, BEdges),
481         create_gray_edges_cg_rev(Vertex, GEdges),
482         create_matching_graph(BEdges, GEdges, FEdges, FGEdges
              ),
483         find_maximum_cardinality_matching(FEdges, FGEdges,
              _MEdges, MGEdges),
484         create_ladder_graph(MGEdges, LEdges, LIndependent),
485         calc_2_cycle(LEdges, LIndependent, C), !.
486
```

Os próximos predicados (A.3) são usados para representar permutações e permutações estendidas.

Listing A.3: Permutações

```
487 %
488 % Permutação
489 %
490
491 % permutation/2: Verifica se Pi é uma permutação
492 permutation(Pi, N) :-
493         length(Pi, N),
494         in_range(1, N, Pi), % Pi :: [1 .. N],
495         all_different(Pi).
496
497 %extend_pi/2: Adiciona 0 no início e N+1 no fim da permutação
      Pi
498 extend_pi([],[]).
499 extend_pi(Pi, EPi) :-
```

```
500         length(Pi, N),
501         N1 is N + 1,
502         append([0|Pi], [N1], EPi).
503
```

O predicado *bound*, no próximo trecho de código, escolhe qual limitante será usado e retorna os valores dos limitantes.

Listing A.4: Predicado bound

```
504 %
505 % Bounds
506 %
507
508 % bound/4: Dado um modelo retorna os limitantes inferior e
      superior para a permutação Pi
509 bound(Pi, def, LB, UB) :-
510         length(Pi, N),
511         LB is 0,
512         UB is N.
513
514 % Limitantes para reversões
515 bound(Pi, rev_br, LB, UB) :-
516         calc_breakpoint(Pi, B, rev),
517         LB is B // 2,
518         UB is B.
519 bound(Pi, rev_cg, LB, UB) :-
520         extend_pi(Pi, EPi),
521         calc_breakpoint(EPi, B, rev),
522         find_2_cycle(EPi, C),
523         LB is ((2*B - C) // 3),
524         UB is (B - (C // 2)).
525
526 % Limitantes para transposições
527 bound(Pi, tra_br, LB, UB) :-
528         calc_breakpoint(Pi, B, trans),
529         LB is B // 3,
530         UB is B.
531 bound(Pi, tra_cg, LB, UB) :-
```

```
532         length(Pi, N),
533         calc_odd_cycles_transposition(Pi, N, C),
534         LB is (N + 1 - C) // 2,
535         UB is (3 * (N + 1 - C)) // 4.
536
537 % Limitantes para reversões + transposições
538 bound(Pi, t_r_br, 0, UB) :-
539         bound(Pi, tra_br, _TLB, TUB),
540         bound(Pi, rev_br, _RLB, RUB),
541         min(RUB, TUB, UB).
542 bound(Pi, t_r_cb, 0, UB) :-
543         bound(Pi, tra_cg, _TLB, TUB),
544         bound(Pi, rev_br, _RLB, RUB),
545         min(RUB, TUB, UB).
546 bound(Pi, t_r_cc, 0, UB) :-
547         bound(Pi, tra_cg, _TLB, TUB),
548         bound(Pi, rev_cg, _RLB, RUB),
549         min(RUB, TUB, UB).
550
```

No trecho de código a seguir (A.5), apresentaremos os predicados dos problemas de ordenações usando a teoria CSP. Note que é possível modificar o código para obter quais permutações foram utilizadas para ordenar a permutação original.

Listing A.5: Modelos CSPs

```
551 %
552 % CSP models
553 %
554
555 % reversal/4: Efetua a reversão no bloco definido por (I,J)
556 reversal(Pi, Sigma, I, J) :-
557         permutation(Pi, N),
558         [I, J] #:: 0 .. N, I #< J,
559         indomain(I),
560         indomain(J),
561         split3(Pi, I, J, C1, C2, C3),
562         reverse(C2, R2),
563         append(C1, R2, C12),
```

```
564         append(C12, C3, C123),
565         Sigma = C123,
566         permutation(Sigma, N).
567
568 % transposition/5: Efetua a transposições dos blocos
      definidos por (I,J,K)
569 transposition(Pi, Sigma, I, J, K) :-
570         permutation(Pi, N),
571         [I, J, K] #:: 0 .. N, I #< J, J #< K,
572         indomain(I),
573         indomain(J),
574         indomain(K),
575         split4(Pi, I, J, K, C1, C2, C3, C4),
576         append(C1, C3, C13),
577         append(C13, C2, C132),
578         append(C132, C4, C1324),
579         Sigma = C1324,
580         permutation(Sigma, N).
581
582 % event/2: Escolhe o melhor evento entre reversão e transposi
      ção
583 event(Pi, Sigma,tra) :-
584         transposition(Pi, Sigma, _I, _J, _K).
585 event(Pi, Sigma,rev) :-
586         reversal(Pi, Sigma, _I, _J).
587 event(Pi, Sigma) :-
588         bound(Pi, tra_br, _TLB, TUB),
589         bound(Pi, rev_br, _RLB, RUB),
590         TUB =< RUB,
591         event(Pi, Sigma, tra).
592 event(Pi, Sigma) :-
593         bound(Pi, tra_br, _TLB, TUB),
594         bound(Pi, rev_br, _RLB, RUB),
595         TUB > RUB,
596         event(Pi, Sigma, rev).
597
598 % reversal_dist/3: CSP: Retorna a distância de reversão
      usando o modelo M
```

```
599 reversal_dist(Pi, 0, _M) :-
600         is_identity(Pi).
601 reversal_dist(Pi, T, M) :-
602         bound(Pi, M, LB, UB),
603         T :: LB .. UB,
604         indomain(T),
605         reversal(Pi, Sigma, _I, _J),
606         TAUX is T - 1,
607         reversal_dist(Sigma, TAUX, M), !.
608
609 % transposition_dist/3: CSP: Retorna a distância de transposi
    ção usando o modelo M
610 transposition_dist(Pi, 0, _M) :-
611         is_identity(Pi).
612 transposition_dist(Pi, T, M) :-
613         bound(Pi, M, LB, UB),
614         T :: LB .. UB,
615         indomain(T),
616         transposition(Pi, Sigma, _I, _J, _K),
617         TAUX is T - 1,
618         transposition_dist(Sigma, TAUX, M), !.
619
620 % rev_trans_dist/3: CSP: Retorna a distância de reversão+
    transposição usando o modelo M
621 rev_trans_dist(Pi, 0, _M) :-
622         is_identity(Pi).
623 rev_trans_dist(Pi, D, M) :-
624         bound(Pi, M, LB, UB),
625         D :: LB .. UB,
626         indomain(D),
627         event(Pi, Sigma),
628         DAUX is D - 1,
629         rev_trans_dist(Sigma, DAUX, M), !.
630
```

O último trecho (A.6), modela os problemas de ordenações usando a teoria COP.

Listing A.6: Modelos COPs

```
631 %
632 % COP models
633 %
634
635 % reversal_cop/5: Modificação do predicado reversal/4 do
       modelo CSP para incluir a variável B
636 reversal_cop(Pi, Sigma, 0, 0, 0) :-
637         is_identity(Pi),
638         is_identity(Sigma), !.
639 reversal_cop(Pi, Sigma, I, J, 1) :-
640         reversal(Pi, Sigma, I, J).
641
642 % transposition/6: Modificação do predicado transposition/5
       do CSP para incluir a variável B
643 transposition_cop(Pi, Sigma, 0, 0, 0, 0) :-
644         is_identity(Pi),
645         is_identity(Sigma), !.
646 transposition_cop(Pi, Sigma, I, J, K, 1) :-
647         transposition(Pi, Sigma, I, J, K).
648
649 % ub_constraint_rev/3: Aplica os efeitos da reversão. Retorna
       os valores corretos na lista B
650 ub_constraint_rev(Pi, [], _M, _UB) :-
651         is_identity(Pi).
652 ub_constraint_rev(Pi, [B|Bt], M, UB) :-
653         reversal_cop(Pi, Sigma, _I, _J, B),
654         bound(Pi, M, LB, _UB),
655         UB >= LB,
656         UBAUX is UB - 1,
657         ub_constraint_rev(Sigma, Bt, M, UBAUX).
658
659 % ub_constraint_trans/3: Aplica os efeitos da reversão.
       Retorna os valores corretos na lista B
660 ub_constraint_trans(Pi, [], _M, _UB) :-
661         is_identity(Pi).
662 ub_constraint_trans(Pi, [B|Bt], M, UB) :-
663         transposition_cop(Pi, Sigma, _I, _J, _K, B),
```

```
664         bound(Pi, M, LB, _UB),
665         UB >= LB,
666         UBAUX is UB - 1,
667         ub_constraint_trans(Sigma, Bt, M, UBAUX).
668
669 % ub_constraint_event/3: Escolhe qual evento será usado.
      Retorna os valores corretos na lista B
670 ub_constraint_event(Pi, [], _M, _UB) :-
671         is_identity(Pi).
672 ub_constraint_event(Pi, [B|Bt], M, UB) :-
673         transposition_cop(Pi, Sigma, _I, _J, _K, B),
674         bound(Pi, M, LB, _UB),
675         UB >= LB,
676         UBAUX is UB - 1,
677         ub_constraint_event(Sigma, Bt, M, UBAUX).
678 ub_constraint_event(Pi, [B|Bt], M, UB) :-
679         reversal_cop(Pi, Sigma, _I, _J, B),
680         bound(Pi, M, LB, _UB),
681         UB >= LB,
682         UBAUX is UB - 1,
683         ub_constraint_event(Sigma, Bt, M, UBAUX).
684
685 % reversal_dist_cop/3: COP: Retorna a distância de reversão
      usando o modelo M
686 reversal_dist_cop(Pi, 0, _M) :-
687         is_identity(Pi), !.
688 reversal_dist_cop(Pi, T, M) :-
689         bound(Pi, M, LB, UB),
690         length(B, UB),
691         minimize((ub_constraint_rev(Pi, B, M, UB),
692                   sum(B, Cost),
693                   Cost >= LB), Cost), T is Cost.
694
695 % transposition_dist_cop/3: COP: Retorna a distância de
      transposição usando o modelo M
696 transposition_dist_cop(Pi, 0, _M) :-
697         is_identity(Pi), !.
698 transposition_dist_cop(Pi, T, M) :-
```

```
699         bound(Pi, M, LB, UB),
700         length(B, UB),
701         minimize((ub_constraint_trans(Pi, B, M, UB),
702                   sum(B, Cost),
703                   Cost >= LB), Cost), T is Cost.
704
705 % rev_trans_dist_cop/3: COP: Retorna a distância de reversão+
      transposição usando o modelo M
706 rev_trans_dist_cop(Pi, 0, _M) :-
707         is_identity(Pi), !.
708 rev_trans_dist_cop(Pi, T, M) :-
709         bound(Pi, M, LB, UB),
710         length(B, UB),
711         minimize((ub_constraint_event(Pi, B, M, UB),
712                   sum(B, Cost),
713                   Cost >= LB), Cost), T is Cost.
714
```

Apêndice B

Constraint Logic Programming Models for Reversal and Transposition Distance Problems

Este apêndice apresentará o artigo "*Constraint Logic Programming Models for Reversal and Transposition Distance Problems*" [22], com texto em inglês publicado nos anais do *VI Brazilian Symposium on Bioinformatics (BSB'2011)*, realizado em Brasília, DF em 2011.

Este artigo apresenta modelos de programação por restrições que buscam o resultados exato para o problema de ordenação por reversões e transposições, baseados na teoria do Problema de Satisfação por Restrições e na teoria do Problema de Otimização com Restrições.

Constraint Logic Programming Models for Reversal and Transposition Distance Problems

Victor de Abreu Iizuka and Zanoni Dias

Institute of Computing, University of Campinas
Av. Albert Einstein 1251, Cidade Universitária, Campinas-SP, Brazil
victor.iizuka@students.ic.unicamp.br, zanoni@ic.unicamp.br

Abstract. Genome Rearrangements research appeared in the last years to deal with problems such as to find the minimum number of rearrangement events, for example, transposition, reversals, fusion and fissions, needed to transform one genome into another. In this paper we follow the research line of Dias and Dias [12] and we introduce Constraint Logic Programming (CLP) models for sorting by reversals and transpositions, based on Constraint Satisfaction Problems (CSP) theory and Constraint Optimization Problems (COP) theory, for unsigned and linear permutations. We made a comparison between the CLP models and the ILP polynomial models described in Dias and Souza [13].

1 Introduction

Genome rearrangements field focus on the comparison of the positions of the same blocks of genes on distinct genomes and on the rearrangement events that possibly transformed one genome into another. Previous studies show that rearrangements are more suitable than nucleotide (or amino acid) comparison when the objective is to compare the genome of two species [6, 19].

The transposition exchanges two adjacent blocks of any size in a chromosome. The transposition distance problem is to find the minimum number of transpositions that transform one genome into another. This problem is NP-hard, the proof was presented by Bulteau, Fertin e Rusu [10]. The best approximation algorithm ratio is 1.375 and was presented by Elias and Hartman [14],

The reversal inverts a block of any size in a chromosome. The reversal distance problem is to find the minimum number of reversals that transform one genome into another. The problem of reversals with unsigned permutations is NP-hard, the proof was presented by Caprara [11]. The best approximation algorithm is 1.375 and was presented by Berman, Hannenhalli and Karpinski [9].

The reversal and transposition distance problem is to find the minimum number of reversals and transpositions that transform one genome into another. Walter, Dias and Meidanis [17, 20] and Lin and Xue [15] studied this problem.

In this paper we will focus on sorting by reversals and transpositions with unsigned and linear permutations and we want to find the exact distance necessary to transform one genome into another. We follow the research line of Dias

and Dias [12] and we introduce Constraint Logic Programming (CLP) models for sorting by reversals and transpositions, based on Constraint Satisfaction Problems (CSP) theory and Constraint Optimization Problems (COP) theory. We made a comparison between the CLP models and the ILP polynomial models described in Dias and Souza [13].

This paper is divided as follow. Section 2 presents the CLP model for sorting by reversal and transposition problem. Section 3 discusses the computational tests. Finally, Section 4 exhibits the conclusions and future works.

2 CLP Model for Sorting by Reversals and Transpositions

In this Section we present a basic CLP model for sorting by reversals and transpositions. We will use the Constraint Satisfaction Problem (CSP) and Constraint Optimization Problems (COP) theories to formulate the problem. We define the formulations using the prolog-like Marriott's notation [16] as much as possible.

The representation of permutation (1) and the effects of reversal (2) and transposition (3) can be seen as the same way we described the problem. In this model the permutation π is a list of elements $(\pi_1, \pi_2, \cdots, \pi_n)$ where $\pi_i \in \mathbb{N}$, $0 < \pi_i \leq n$ and $\pi_i \neq \pi_j$ for $i \neq j$. The identity permutation ι is defined as $\iota = (1\ 2\ 3\ \cdots\ n)$

$$\begin{aligned} &\mathit{permutation}(\pi, N)\ \text{:-} \\ &\quad \mathit{length}(\pi, N), \\ &\quad \pi\ ::\ [1\ ..\ N], \\ &\quad \mathit{all\_different}(\pi). \end{aligned} \tag{1}$$

Note that in prolog variables are denoted by strings starting with an upper letter or "_" (the underscore) if the variable is anonymous. The greek letters π and σ are lists in this notation. The construction $X\ ::\ [i\ ..\ j]$ means that X (or every element of X if X is a list) ranges over the interval $[i\ ..\ j]$.

A reversal $\rho(i, j)$, $0 < i < j \leq n$, split the list in three sub-lists $C_1C_2C_3$ where $C_1 = (\pi_1..\pi_{i-1})$, $C_2 = (\pi_i..\pi_j)$ and $C_3 = (\pi_{j+1}..\pi_n)$. After that, we do a reverse on the sub-list C_2 and the result is the sub-list R_{C_2}. Finally we join the new sub-list R_{C_2} with the sub-lists C_1 and C_3 to form $\rho\pi = C_1R_{C_2}C_3$.

$$\begin{aligned} &\mathit{reversal}(\pi, \sigma, I, J)\ \text{:-} \\ &\quad \mathit{permutation}(\pi, N), \\ &\quad \mathit{permutation}(\sigma, N), \\ &\quad 1 \leq I < J \leq N, \\ &\quad \mathit{split}(\pi, I, J, C_1, C_2, C_3), \\ &\quad \mathit{reverse}(C_2, R_{C_2}), \\ &\quad \sigma = C_1, R_{C_2}, C_3. \end{aligned} \tag{2}$$

A transposition $\rho(i,j,k)$, $0 < i < j < k \leq n$, split the list in four sublists $C_1C_2C_3C_4$ where $C_1 = (\pi_1..\pi_{i-1})$, $C_2 = (\pi_i..\pi_{j-1})$, $C_3 = (\pi_j..\pi_{k-1})$ and $C_4 = (\pi_k..\pi_n)$. After we join them to form $\rho\pi = C_1C_3C_2C_4$. Note that C_1 and C_4 could be empty.

$$
\begin{aligned}
&\mathit{transposition}(\pi,\sigma,I,J,K)\ \text{:-}\\
&\quad \mathit{permutation}(\pi,N),\\
&\quad \mathit{permutation}(\sigma,N),\\
&\quad 1 \leq I < J < K \leq N,\\
&\quad \mathit{split}(\pi,I,J,K,C_1,C_2,C_3,C_4),\\
&\quad \sigma = C_1,C_3,C_2,C_4.
\end{aligned}
\tag{3}
$$

We first model the problem as CSP, but the number of variables is unknown because we need the value of distance $d_{r\_t}(\pi)$ to set the constraints and variables that represent the permutations. For this reason we pick a candidate value for distance N such that $N \in [LB..UB]$, where LB is a known lower bound and UB is a known upper bound for the problem, and try to find the appropriate combination of N reversals and transpositions. If the CSP fails with the candidate N, we choose another value for N just incrementing its value. We check the value of N using a bottom-up strategy and for definition we don't check any value higher than any upper bound UB. This behaviour is described by *rev_trans_dist/3* predicate (4).

The *event/2* predicate chooses the best event between the *reversal/4* predicate (2) and the *transposition/5* predicate (3) to minimize the distance.

The *indomain(X)* predicate gets the domain of the variable X and chooses the minimum element in it. If a fail backtracks to *indomain*, the element that generated the fail will be removed from the domain and another value will be chosen.

$$
\begin{aligned}
&\mathit{rev\_trans\_dist}(\iota,0,\_Model).\\
&\mathit{rev\_trans\_dist}(\pi,N,Model)\ \text{:-}\\
&\quad \mathit{bound}(\pi,Model,LB,UB),\\
&\quad N :: [LB..UB],\\
&\quad \mathit{indomain}(N),\\
&\quad \mathit{event}(\pi,\sigma),\\
&\quad \mathit{rev\_trans\_dist}(\sigma,N-1,Model).
\end{aligned}
\tag{4}
$$

The CSP models have the above structure changing only the bounds we used. We call **def_csp** the model that doesn't use any lower bounds, **r_t_br_csp** the model that chooses the best bound between the reversal breakpoint lower and upper bounds described by Bafna and Pevzner [7] and the transposition breakpoint lower and upper bounds described by Bafna and Pevzner [8], and

r_t_bc_csp the model that chooses the best bound between the reversal breakpoint lower and upper bounds and the transposition edge-colored cycle graph lower and upper bounds described by Bafna and Pevzner [8].

Another approach is to model the problem as a COP. This approach needs an upper bound and some changes on previous predicates. We use the binary variables B to indicate whether a event, reversal or transposition, has modified the permutation.

The first predicate that we need to create is *reversal_cop/5* (5). First of all, given a permutation $\rho(i, j)$, we add a new clause to allow $(i, j) = (0, 0)$. If $(i, j) = (0, 0)$ then $\pi\rho = \pi$. We add a new argument to the *reversal_cop/5* predicate that receive the variable B.

$$\begin{aligned} &\mathit{reversal\_cop}(\iota, \iota, 0, 0, 0). \\ &\mathit{reversal\_cop}(\pi, \sigma, I, J, 1) \text{ :- } \mathit{reversal}(\pi, \sigma, I, J). \end{aligned} \tag{5}$$

The equivalent predicate for transposition is *transposition_cop/6* (6). In this case, given a permutation $\rho(i, j, k)$, we add a new clause to allow $(i, j, k) = (0, 0, 0)$. If $(i, j, k) = (0, 0, 0)$ then $\pi\rho = \pi$.

$$\begin{aligned} &\mathit{transposition\_cop}(\iota, \iota, 0, 0, 0, 0). \\ &\mathit{transposition\_cop}(\pi, \sigma, I, J, K, 1) \text{ :- } \mathit{transposition}(\pi, \sigma, I, J, K). \end{aligned} \tag{6}$$

To calculate the reversal and transposition distance in the COP model we implemented the *rev_trans_dist_cop/3* predicate (7), which set the variables B using the upper bound and constrains the permutations by making $\pi_k = \pi_{k-1}\rho_k$. The predicate *length/2* is a prolog built-in and is used to create a list of non instantiated variables of a given size. The cost function *Cost* is the sum of variables B associated with each ρ_k, $Cost = \sum_{k=1}^{UB} B_k$, where UB is a known upper bound. The reversal and transposition distance is the minimum value of the cost function $d_{r\_t} = \min Cost$. To avoid unnecessary processing, the value of $Cost$ must be greater or equal to any lower bound.

$$\begin{aligned} &\mathit{rev\_trans\_dist\_cop}(\pi, N, Model) \text{ :-} \\ &\quad \mathit{bound}(\pi, Model, LB, UB), \\ &\quad \mathit{length}(B, UB), \\ &\quad \mathit{upperbound\_constraint\_rev\_trans}(\pi, B, Model, UB), \\ &\quad \mathit{sum}(B, Cost), \\ &\quad Cost \geq LB, \\ &\quad \mathit{minimize}(Cost, N). \end{aligned} \tag{7}$$

The *upperbound_constraint_rev_trans/4* predicate (8) applies the effects of ρ_k in permutation and returns the value of B for every reversal or transposition

ρ_k. An important constraint is to check if it is possible to sort the permutation using the remaining number of reversals and transpositions to avoid unnecessary processing. The *event_cop/3* predicate chooses the best event between reversal, using the *reversal_cop/5* predicate (5), and transposition, using the *transposition_cop/6* predicate (6), to minimize the distance.

$$
\begin{aligned}
&\mathit{upperbound\_constraint\_rev\_trans}(\iota, [\,], \_Model, \_UB).\\
&\mathit{upperbound\_constraint\_rev\_trans}(\pi, [B|Bt], Model, UB)\ \text{:-}\\
&\quad \mathit{event\_cop}(\pi, \sigma, B),\\
&\quad \mathit{bound}(\pi, Model, LB, \_UB),\\
&\quad UB \geq LB,\\
&\quad \mathit{upperbound\_constraint\_rev\_trans}(\sigma, Bt, Model, UB - 1),
\end{aligned}
\tag{8}
$$

The COP models have the above structure changing only the bounds we used. We use the same CSP bounds modified for COP models. So we have the following bounds: **def_cop**, **r_t_br_cop** and **r_t_bc_cop**.

3 Computational Experiments

All the constraint logic programming models were implemented using the open source programming system *ECLiPSe* [2] and the proprietary C++ package *IBM® ILOG® CPLEX® CP Optimizer* [3]. We recommend Apt and Wallace [5] and Marriott and Stuckey [16] as introduction for CLP using *ECLiPSe*.

All the integer programming formulations were implemented using the open source system *GLPK*[1], the models were written in *GNU MathProg modeling language* intended for describing linear mathematical programming models, and the proprietary C++ package *IBM® ILOG® CPLEX® Optimizer*[4].

We carried our tests on a microcomputer equipped with Intel® Core™ 2 Duo 2.33GHz, with 3 GB of RAM, running under a Ubuntu Linux operating system with kernel 2.6.31, *ECLiPSe-6.0*, *GLPK-4.35*, *IBM ILOG CPLEX CP Optimizer v 2.3* and *IBM ILOG CPLEX Optimizer v 12.1*.

The CLP results used the model described in previous section. The ILP results used the formulations described in Dias and Souza [13].

The Table 1 summarize the results. The column **size** represents the length of permutations used in the tests. The CPU times (in seconds) reported refers to an average of 50 instances where the permutation π was randomly generated. In all models we made a comparison between the permutation π and the identity permutation ι. The times are given in seconds and grow very fast as the instance size increases. This behavior occur due to the exponential search space, in case of CLP models, and occur due to model sizes, in case of ILP formulations. The timeout is printed when the model could not finish the entire tests within a time limit of 25 hours. Instances with $|\pi| \geq 13$ printed timeout in all models, except for the model **r_t_bc_csp**.

Table 1. Average time (in seconds) for sorting by reversals and transpositions models. The "-" means that the model could not finish the entire data test within a time limit of 25 hours.

Reversals and Transpositions Models															
		CP						ILOG CP						ILP	
size	ECLiPSe													GLPK	ILOG CPLEX
	def_cop	r_t_br_cop	r_t_bc_cop	def_csp	r_t_br_csp	r_t_bc_csp	def_cop	r_t_br_cop	r_t_bc_cop	def_csp	r_t_br_csp	r_t_bc_csp		
3	0.034	0.012	0.003	0.004	0.003	0.002	0.004	0.002	0.001	0.004	0.002	0.003	0.001	0.002
4	7.370	12.288	0.341	0.028	0.007	0.004	0.008	0.008	0.007	0.012	0.009	0.005	0.001	0.012
5	-	-	26.047	0.343	0.020	0.010	0.026	0.024	0.021	0.031	0.021	0.013	0.396	0.055
6	-	-	409.079	16.742	0.122	0.031	0.268	0.232	0.103	0.085	0.066	0.046	4.062	0.808
7	-	-	-	593.666	0.670	0.104	1.896	1.967	1.179	0.533	0.400	0.255	3.660	94.429
8	-	-	-	-	2.579	0.149	12.851	10.589	5.566	3.088	2.655	1.531	-	-
9	-	-	-	-	13.958	0.339	468.581	422.396	102.687	61.973	60.465	19.984	-	-
10	-	-	-	-	64.208	1.318	-	-	-	-	-	1189.290	-	-
11	-	-	-	-	167.423	3.327	-	-	-	-	-	-	-	-
12	-	-	-	-	1058.050	11.044	-	-	-	-	-	-	-	-
13	-	-	-	-	-	20.961	-	-	-	-	-	-	-	-
14	-	-	-	-	-	51.294	-	-	-	-	-	-	-	-
15	-	-	-	-	-	164.994	-	-	-	-	-	-	-	-
16	-	-	-	-	-	188.704	-	-	-	-	-	-	-	-
17	-	-	-	-	-	1046.984	-	-	-	-	-	-	-	-

We can see that CLP models based on CSP theory have better times in comparison to ILP formulation and the CLP models based on COP theory have the worst times. In terms of CLP models the model **r_t_bc_csp** have the best times (especially in large instances), due to breakpoint bounds with transposition edge-colored cycle graph bounds. This is an expected result if we consider the fact that the cycle graph lower bound is tighter than the breakpoint lower bound. Another interesting fact is that the ILOG models were faster than ECLiPSe at first, but in the end the ILOG models became slower. In terms of ILP models we can see that GLPK models, in average, had better times than ILOG models.

We can observe that even with the ILP polynomial model, the times grows very fast due to the order of polynomials, making the ILP models prohibitive in practice [13].

Note that in the experiment the CLP model based on COP theory have the worst CPU time. We did not use any heuristic that could improve the results, but the search mechanism of COP models is to find a solution and try to improve it by looking for new solutions with improved value of the cost function. In the end it will lead to a greater space search than CSP models that use a bottom-up strategy.

4 Conclusion and Future Works

In this paper we introduced a Constraint Logic Programming model for sorting by reversals and transpositions, based on Constraint Satisfaction Problem theory and Constraint Optimization Problem Theory. We made a comparison between the CLP models and the ILP polynomial models described in Dias and Souza [13]. The analysis shows that the CLP models based on CSP theory achieved better performance than ILP formulations.

This approach is still not viable in practice. For future works we plan to improve the models with better bounds for the problem, reducing the search space for them, or with some procedure to choose a set of reversals or transpositions that will be branched firstly.

In terms of ILP models we plan to improve the bounds of models with techniques like *Lagrangian Relaxation* or writing a new model in order to apply *Column Generation* or *Branch-and-Cut* [18, 21].

Acknowledgments. This work was supported by CNPq (472504/2007-0, 479207/2007-0, 483177/2009-1). Victor de Abreu Iizuka receives a MSc scholarship from CNPq.

References

1. GNU Linear Programming Kit. `http://www.gnu.org/software/glpk/` (2010)
2. The ECLiPSe Constraint Programming System. `http://www.eclipseclp.org/` (April 2011)
3. IBM® ILOG® CPLEX® CP Optimizer. `http://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/` (April 2011)

4. IBM® ILOG® CPLEX® Optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/ (April 2011)
5. Apt, K., Wallace, M.: Constraint Logic Programming using Eclipse. Cambridge University Press, New York, NY, USA (2007)
6. Bafna, V., Pevzner, P.A.: Sorting by Reversals: Genome Rearrangements in plant and organelles and evolutionary history of X chromossome. Molecular Biology and Evolution 12(2), 239–246 (1995)
7. Bafna, V., Pevzner, P.A.: Genome rearrangements and sorting by reversals. SIAM Journal on Computing 25(2), 272–289 (1996)
8. Bafna, V., Pevzner, P.A.: Sorting by transpositions. SIAM Journal on Discrete Mathematics 11(2), 224–240 (1998)
9. Berman, P., Hannenhalli, S., Karpinski, M.: 1.375-approximation algorithm for sorting by reversals. In: Proceedings of the 10th Annual European Symposium on Algorithms (ESA'02). pp. 200–210. Springer-Verlag, London, UK (2002)
10. Bulteau, L., Fertin, G., Rusu, I.: Sorting by transpositions is difficult. Computing Research Repository abs/1011.1157 (2010)
11. Caprara, A.: Sorting by reversals is difficult. In: Proceedings of the first annual international conference on Computational molecular biology (RECOMB'97). pp. 75–83. ACM, New York, NY, USA (1997)
12. Dias, U., Dias, Z.: Constraint programming models for transposition distance problem. Lecture Notes on Bioinformatics 5676, 13–23 (2009)
13. Dias, Z., de Souza, C.: Polynomial-size ILP models for rearrangements distance problems. In: Proceedings of the Brazilian Symposium on Bioinformatics (BSB'2007). pp. 74–85 (2007)
14. Elias, I., Hartman, T.: A 1.375-approximation algorithm for sorting by transpositions. IEEE/ACM Transactions on Computational Biology Bioinformatics 3(4), 369–379 (2006)
15. Lin, G.H., Xue, G.: Signed genome rearrangement by reversals and transpositions: Models and approximations. Lecture Notes in Computer Science 1627, 71–80 (1999)
16. Marriott, K., Stuckey, P.J.: Programming with constraints: an introduction. MIT Press (1998)
17. Meidanis, J., Walter, M.E.M.T., Dias, Z.: A lower bound on the reversal and transposition diameter. Journal of Computational Biology 9(5) (2002)
18. Nemhauser, G., Wolsey, L.: Integer and Combinatorial Optimization. Wiley-Interscience (1988)
19. Palmer, J.D., Herbon, L.A.: Plant mitochondrial DNA evolves rapidly in structure, but slowly in sequence. Journal of Molecular Evolution 27, 87–97 (1988)
20. Walter, M.E.M.T., Dias, Z., Meidanis, J.: Reversal and transposition distance of linear chromosomes. In: Proceedings of the String Processing and Information Retrieval (SPIRE'98) (1998)
21. Wolsey, L.: Integer Programming. Wiley-Interscience (1998)

Referências Bibliográficas

[1] The ECLiPSe Constraint Programming System. `http://www.eclipseclp.org/`.

[2] Glpk (GNU Linear Programming Kit). `http://www.gnu.org/software/glpk/`.

[3] GNU Compiler Collection. `http://gcc.gnu.org/`.

[4] IBM® ILOG® CPLEX® CP Optimizer. `http://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/`.

[5] IBM® ILOG® CPLEX® Optimizer. `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`.

[6] K. Apt. *Principles of Constraint Programming*. Cambridge University Press, New York, NY, USA, 2003.

[7] K. Apt and M. Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, New York, NY, USA, 2007.

[8] D. A. Bader, B. M. E. Moret, and M. Yan. A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. *Journal of Computational Biology*, 8(5):483–491, 2001.

[9] V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, 25(2):272–289, 1996.

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

[11] A. Bergeron. A very elementary presentation of the Hannenhalli-Pevzner theory. *Discrete Applied Mathematics*, 146:134–145, March 2005.

[12] P. Berman, S. Hannenhalli, and M. Karpinski. 1.375-approximation algorithm for sorting by reversals. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA'2002)*, pages 200–210, London, UK, UK, 2002. Springer-Verlag.

[13] L. Bulteau, G. Fertin, and I. Rusu. Sorting by transpositions is difficult. *Computing Research Repository*, abs/1011.1157, 2010.

[14] A. Caprara. Sorting by reversals is difficult. In *Proceedings of the first annual international conference on Computational molecular biology (RECOMB'97)*, pages 75–83, New York, NY, USA, 1997. ACM.

[15] D. A. Christie. A 3/2-approximation algorithm for sorting by reversals. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, SODA '98, pages 244–252, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[16] U. Dias and Z. Dias. Constraint programming models for transposition distance problem. In *Proceedings of the 4th Brazilian Symposium on Bioinformatics (BSB'2009)*, volume 5676 of *Lecture Notes on Computer Science*, pages 13–23, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] Z. Dias and C. de Souza. Polynomial-size ILP models for rearrangements distance problems. In *Proceedings of the Brazilian Symposium on Bioinformatics (BSB'2007)*, pages 74–85, 2007.

[18] I. Elias and T. Hartman. A 1.375-approximation algorithm for sorting by transpositions. *IEEE/ACM Transactions on Computational Biology Bioinformatics*, 3(4):369–379, 2006.

[19] Q.-P. Gu, S. Peng, and H. Sudborough. A 2-approximation algorithm for genome rearrangements by reversals and transpositions. *Theor. Comput. Sci.*, 210(2):327–339, January 1999.

[20] S. Hannenhalli, C. Chappey, E. V. Koonin, and P. A. Pevzner. Genome sequence comparison and scenarios for gene rearrangements: a test case. *Genomics*, 30:299–311, 1995.

[21] S. Hannenhalli and P. A. Pevzner. Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals). In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing (STOC'95)*, pages 178–189, New York, NY, USA, 1995. ACM.

[22] V. A. Iizuka and Z. Dias. Constraint logic programming models for reversal and transposition distance problems. In *Proceedings of the Brazilian Symposium on Bioinformatics (BSB'2011)*, pages 25–32, Brasília-DF, Brazil, 2011.

[23] A. Labarre. New bounds and tractable instances for the transposition distance. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 3(4):380–394, 2006.

[24] G.-H. Lin and G. Xue. Signed genome rearrangement by reversals and transpositions: Models and approximations. *Lecture Notes in Computer Science*, 1627:71–80, 1999.

[25] K. Marriott and P. J. Stuckey. *Programming with constraints: an introduction*. MIT Press, 1998.

[26] J. Meidanis, M. E. M. T. Walter, and Z. Dias. Transposition distance between a permutation and its reverse. In R. Baeza-Yates, editor, *Proceedings of the 4th South American Workshop on String Processing (WSP'97)*, pages 70–79, Valparaiso, Chile, 1997. Carleton Univertity Press.

[27] J. Meidanis, M. E. M. T. Walter, and Z. Dias. Reversal distance of signed circular chromosomes. Technical Report IC-00-23, Institute of Computing, University of Campinas, December 2000.

[28] J. Meidanis, M. E. M. T. Walter, and Z. Dias. A lower bound on the reversal and transposition diameter. *Journal of Computational Biology*, 9(5):743–746, 2002.

[29] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*. Wiley-Interscience, New York, NY, USA, 1988.

[30] J. D. Palmer and L. A. Herbon. Plant mitochondrial DNA evolves rapidly in structure, but slowly in sequence. *Journal of Molecular Evolution*, 27:87–97, 1988.

[31] J. C. Setubal and J. Meidanis. *Introduction to Computional Molecular Biology*. PWS Publishing Company, 1997.

[32] E. Tannier and M.-F Sagot. Sorting by reversals in subquadratic time. In *Proceedings of Combinatorial Pattern Matching (CPM'2004)*, pages 1–13, Istambul, Turkey, 2004.

[33] M. E. M. T. Walter, Z. Dias, and J. Meidanis. Reversal and transposition distance of linear chromosomes. In *Proceedings of the String Processing and Information Retrieval (SPIRE'98)*, 1998.

[34] L. Wolsey. *Integer Programming*. Wiley-Interscience, New York, NY, USA, 1998.