UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ESTUDO DE PROPRIEDADES DA ESTRUTURA *SHADOW TREE* PARA O PROBLEMA DE *PERFECT PHYLOGENY HAPLOTYPING*

TRABALHO DE GRADUAÇÃO

2005.1

Aluno: Enio Felipe da Rocha (efr@cin.ufpe.br)
Orientador: Katia Silva Guimarães (katia@cin.ufpe.br)

Recife, 10 de Agosto de 2005

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

Enio Felipe da Rocha

ESTUDO DE PROPRIEDADES DA ESTRUTURA *SHADOW TREE* PARA O PROBLEMA DE *Perfect Phylogeny Haplotyping*

Trabalho apresentado ao Programa de Graduação em Ciência da Computação da Universidade Federal de Pernambuco como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientadora: Katia Silva Guimarães

Recife

10 de agosto de 2005

Resumo

O *Perfect Phylogeny Haplotyping Problem* (PPH) tem sido alvo de grande interesse por parte da comunidade científica. Isto se deve, principalmente, à descoberta de que *haplotypes* desenvolvem um papel central no entendimento da evolução das espécies. Inúmeros artigos propuseram soluções para este problema e recentemente foi elaborada por Ding, Filkov e Gusfield [1] uma solução em tempo linear para o problema. Esta solução utiliza uma estrutura de dados batizada de S*hadow Tree* e operações sobre esta estrutura.

Este trabalho apresenta um estudo das principais características desta estrutura de dados e o efeito das operações que são feitas sobre ela. Além disto é feita, através de exemplos intuitivos, uma verificação dos lemas e teoremas propostos no artigo e uma análise do desempenho de cada procedimento do algoritmo.

Agradecimentos

Agradeço primeiramente a Deus pelo presente da vida e a Nossa Senhora, mãe de todos nós.

À minha inesquecível mãe Maria Lúcia (*in memorian*), meu porto seguro, minha fonte de inspiração, meu exemplo de vida a quem devo absolutamente tudo o que conquistei e o que sou. Obrigado mãe pela grandiosa lição de vida que você me deu e por ter me concedido o privilégio de viver 18 anos ao seu lado

À toda minha família, principalmente meu irmão Gerd, meu pai Valdecir, minhas sobrinhas Amanda, Ana Clara e Camila, minha cunhada Patrícia, minhas tias Marlúcia, Marlene, Salete, Ivanilda, Lourdes, meu tio Mário, todos os meus primos e primas especialmente Ygor, Pablo, Juca, Kylza e Edna pelo apoio durante toda a vida e por compartilharmos tantos momentos felizes.

À minha namorada Nara Leylane que foi a pessoa que mais me aconselhou a não desistir do curso nos momentos difíceis, e pelos quase 2 anos juntos.

Aos meus amigos, em especial à Carol, Adriano, Abner, Marília, Gustavo, Pyunghwa, Junghwa, Debra Dukes, Linda Butler simplesmente pela amizade demonstrada.

À minha orientadora Katia Silva Guimarães pelos conhecimentos ensinados e pela perseverança em oferecer as disciplinas do perfil de Bioinformática. Muito sucesso na temporada no **NIH**!!!!

À professora Ana Lúcia Bezerra Candeias do Departamento de Engenharia Cartográfica, de quem fui bolsista nos anos de 2001 e 2002, pelas oportunidades que me ofereceu.

Aos meus colegas de curso, principalmente Émerson, Gilberto, Diego e Max pelas horas que passamos juntos fazendo os projetos.

Sumário

1. Introdução

A Biologia Computacional é uma área que vem despertando grande interesse dos pesquisadores bem como da própria indústria. Esta área se caracteriza pela interdisciplinaridade entre dois grandes campos de pesquisa: a Ciência da Computação e a Biologia Molecular. A Biologia Computacional tem como principal objetivo a resolução de problemas de Biologia através do emprego de técnicas de matemática aplicada, estatística, computação e engenharia. Alguns dos principais problemas que têm sido bastante pesquisados incluem o alinhamento de seqüências, predição de estrutura de proteínas, análise de expressão gênica entre outros.

1.1 – Motivação Biológica

A maioria das células humanas possui 46 cromossomos em seu núcleo que são agrupados em pares, formando, portanto, 23 pares. Cada um desses pares é formado por um cromossomo herdado do pai e outro cromossomo herdado da mãe do indivíduo.

Os cromossomos, porém, não são transmitidos através das gerações como cópias idênticas. Durante a formação das células sexuais, conhecidos como gametas, os cromossomos de um indivíduo sofrem um processo chamado de *recombinação*. Neste processo o cromossomo materno e o paterno se alinham e trocam pedaços, formando assim um novo cromossomo que contém material genético tanto do pai quanto da mãe do indivíduo. Por exemplo, digamos que um indivíduo X, do sexo masculino, tenha em suas células diplóides o par de cromossomos com os seguintes alelos: (**A B C D)** (cromossomo paterno) e (**a b c d)** (cromossomo materno).

No processo de formação das células sexuais estes cromossomos irão dar origem a um novo cromossomo, o qual conterá certos genes do cromossomo paterno e outros do cromossomo materno. No exemplo descrito o cromossomo

resultante depois do processo de recombinação poderia ser (**A b c d**), ou ainda (**A B c d**) entre outras possibilidades. Para cada par de cromossomos de uma célula diplóide, será formado um único cromossomo nos gametas. Desse modo os gametas contêm apenas metade dos cromossomos de uma célula diplóide (23 cromossomos), sendo por isso chamados de células *haplóides*. Digamos agora que um gameta deste indivíduo X, contendo os alelos (**A B c d**) fecunde um óvulo contendo os alelos (**a b C D**). O filho deste cruzamento terá em suas células diplóides o seguinte material genético: (**A B c d**) (cromossomo paterno) e (**a b C D**) (cromossomo materno).

Com o passar das gerações, segmentos de cromossomos ancestrais são misturados devido a múltiplos eventos de recombinação. Porém, alguns segmentos dos cromossomos dos ancestrais são encontrados em diversos indivíduos de uma população (Figura 1). Estes segmentos não foram alvos de nenhum evento de recombinação e estão separados por segmentos onde eventos de recombinação ocorreram. Os segmentos conservados dos cromossomos e que parecem imunes a eventos de recombinação são chamados de *haplotypes*. Por exemplo, usando o mesmo caso do indivíduo X, digamos que o seu descendente, chamado de X1, realmente recebeu no seu cromossomo paterno o trecho (**A B c d**) e que durante a produção dos seus gametas os genes **A** e **B** sejam sempre herdados como um grupo. Este grupo, nesse caso formado pelos genes **A** e **B** é que é chamado de haplotype.

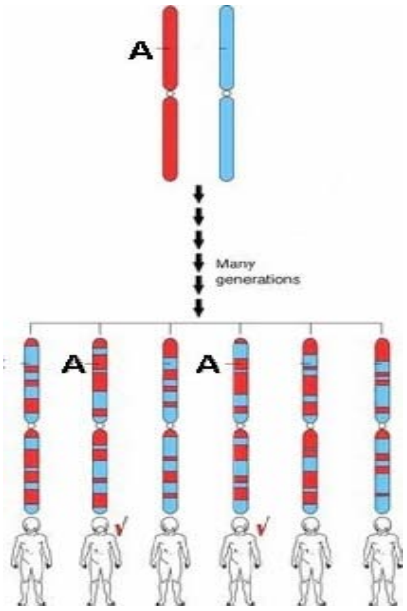


Figura 1: Com o passar das gerações pedaços dos cromossomos ancestrais são misturados, devido a recombinação, resultando em diferentes descendentes. Se um determinado alelo, chamado de A, provoca uma determinada doença os descendentes que herdaram aquele gene terão um maior risco de desenvolve-la.

Baseado nestas descobertas de que genes podem ser herdados como grupos conservados por várias e várias gerações, o **NIH** (*National Institutes of Health*) deu início a um projeto internacional que tem como objetivo comparar os genomas de vários indivíduos para identificar exatamente quais regiões dos cromossomos são compartilhadas entre eles. Essa descoberta tem impacto na comunidade científica pois a partir destes resultados poderão ser identificados genes que estão ligados ao aparecimento de certas doenças [6].

As diferenças entre bases individuais são as formas mais comuns de variação genética e são conhecidas como *Polimorfismos de Base Única* ou pela sigla em inglês, SNP que é abreviação de *Single Nucleotide Polymorphism*. [1, 3, 6]. Desse modo ao descobrir os aproximadamente 10 milhões de SNPs presentes no genoma humano, o HapMap terá dado um grande passo no entendimento das diferenças que ocorrem entre os indivíduos. Esses polimorfismos têm uma importância muito grande uma vez que eles agem como marcadores de certos genes[2]. Por exemplo,

considerando que um SNP aumenta o risco de certas pessoas desenvolverem diabetes mas não se sabe em que posição o gene está localizado no cromossomo, é possível então comparar as seqüências de indivíduos que têm a doença com as seqüências de outros que não têm. A partir disto pode-se ter uma idéia de onde o gene causador da diabetes está localizado. Na Figura 2 é mostrada a relação entre SNPs e haplotypes.

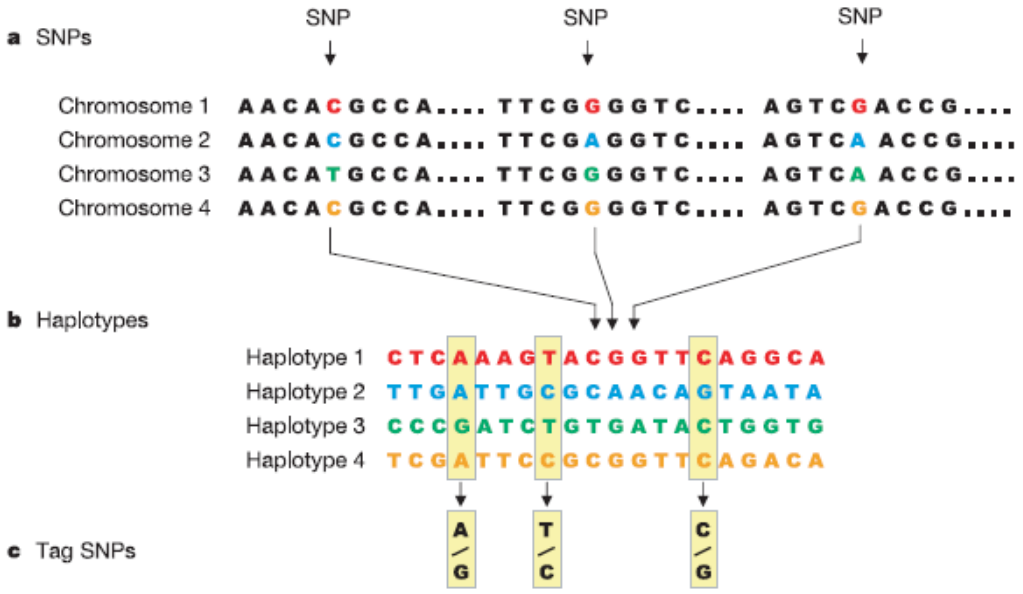


Figura 2: (a) seqüências de um mesmo cromossomo em quatro pessoas diferentes. Grande parte de suas bases são idênticas diferindo apenas em três sítios (SNPs). (b) Os haplotypes são compostos por uma combinação particular de alelos. Aqui estão mostrados vinte SNPs (apenas as bases que não são iguais nos quatro cromossomos são mostradas) que são mais comuns em determinada população.

1.2 – Definição do Problema

Como foi mostrado na Figura 2, a maioria dos SNPs é formada por apenas duas bases, sendo portanto possível atribuir os valores 0 e 1 para estas bases que diferem. A obtenção de dados de haplotypes é custosa e não-realista [1, 3] e

normalmente os dados de genotypes são obtidos. Estes dados são basicamente vetores cujos sítios (ou colunas) podem ter entrada 0, 1 ou 2. Um sítio *i* no vetor *g* tem entrada 0 se ambos os vetores de haplotypes apresentam 0 neste sítio, o mesmo acontecendo para a entrada 1. Quando um sítio *j* apresenta entrada 2 no vetor de genotypes é porque um dos vetores de haplotypes tem entrada 0 para este sítio enquanto o outro tem entrada 1.

O problema descrito acima é conhecido na literatura como *Haplotype Inference* [1, 2, 3, 4] e é formalmente definido do seguinte modo: Dado um conjunto de *n* vetores de genotypes de tamanho *m*, deve-se achar um par de vetores de haplotypes binários para cada vetor de genotypes do conjunto. De tal modo que os vetores de genotypes possam ser gerados pelos vetores de haplotypes. Por exemplo, considere o vetor de genotypes g = (1, 0, 0, 2). Há apenas dois pares de vetores de haplotypes que geram este vetor de genotypes, sendo eles p1 = [(1, 0, 0, 0), (1, 0, 0, 1)] e p2 = [(1, 0, 0, 1), (1, 0, 0, 0)]. Claramente é possível notar que haverão 2^{d-1} soluções, onde *d* é igual ao número de entradas 2 no vetor de genotype.

Como foi citado, pode haver diversas soluções para o problema de inferência de haplotypes para um determinado vetor de entrada. Porém, o que se deseja é achar a solução que corresponda a mais correta, ou seja, aquela que tenha valor do ponto de vista da genética. Gusfield [1, 2, 3] sugeriu que a solução mais significativa deste problema teria que formar uma filogenia perfeita. Este problema foi batizado de *The Perfect Phylogeny Haplotyping Problem* (PPH) e é formalmente definido da seguinte forma:

- Dada uma matriz *S* com *n* linhas e *m* colunas, contendo *n* genotypes de tamanho *m*, ache *n* pares de haplotypes que gerem *S* e formem uma filogenia perfeita.

Neste problema, as folhas da árvore filogenética são os *haplotypes* inferidos e a sua raiz é a seqüência com m 0's.

Gusfield, além de formular o problema [2], apresenta a primeira solução para ele, cujo tempo de execução é $O(nm\alpha(nm))$, onde α é a função de Ackerman. A solução proposta neste artigo é uma redução do PPH para outro problema bastante conhecido e para o qual uma solução quase linear já havia sido proposta há mais de quinze anos, o *Graph-realization problem*. Entretanto esta solução para o PPH mostrou-se inadequada, apesar de sua corretude, pois é muito difícil de ser implementada.

Posteriormente duas novas abordagens foram propostas para solucionar o PPH em tempo linear que, apesar de serem mais simples, se mostraram menos eficientes uma vez que apresentaram tempo de execução $O(nm^2)$.

Muito recentemente, Ding, Filkov e Gusfield [1] apresentaram um algoritmo linear para o problema PPH, baseado em propriedades de uma estrutura de dados chamada *Shadow Tree*, uma árvore direcionada, cujas arestas apontam na direção da raiz, e que serve para representar soluções para o problema PPH.

O objetivo central deste trabalho é realizar um estudo teórico das:

- Operações *edge addition*, *class flipping* e *class merging* com seus respectivos efeitos, e
- Propriedades da estrutura *Shadow Tree* que são mais centrais para a prova de correção do algoritmo linear para o problema PPH.

Por ser uma proposta muito recente, a estrutura *Shadow Tree*, não é discutida em qualquer outra fonte bibliográfica, exceto o artigo de Ding, Filkov e Gusfield [1]. Este artigo apresenta algumas operações no texto e provas de algumas propriedades no Apêndice A. Devido a problemas de espaço, no entanto, o texto é conciso. A nossa proposta é produzir uma apresentação detalhada das provas, enriquecida com uma série de exemplos, que ajudem o leitor a ter uma compreensão mais rápida e clara das operações e propriedades discutidas.

2. Contexto

Na Computação, as árvores são estruturas amplamente usadas que emulam o formato de uma árvore real através de um conjunto de nós interligados. Cada um desses nós tem zero ou mais filhos que são posicionados abaixo dele. Um nó poderá ter como pai apenas um outro nó, com exceção da raiz da árvore que não possui pai. Dentre as várias classificações dadas às árvores, as principais se referem à presença ou não de um nó raiz ou ainda ao número de filhos que cada nó pode ter.

Uma árvore filogenética, ou filogenia, é uma representação das ligações evolucionárias entre diferentes espécies ou mesmo entre diferentes indivíduos que tenham possivelmente um ancestral em comum. Em uma árvore filogenética cada nó representa o mais recente ancestral em comum dos seus filhos e os nós internos são chamados formalmente de Unidade Hipotética de Taxonomia, pois seres com tais características podem ou não ter existido.

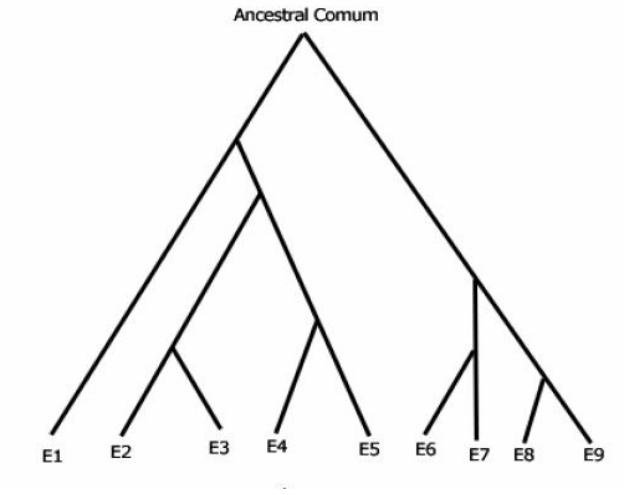


Figura 3: Exemplo de árvore filogenética. Os nós mais próximos tendem a apresentar características mais semelhantes.

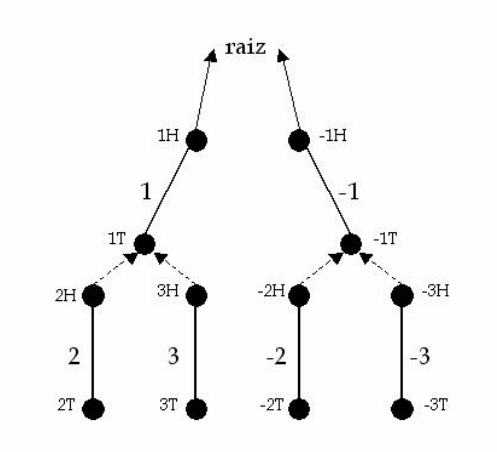
2.1 – Descrição da Estrutura de Dados

A estrutura de dados *shadow tree* possui dois diferentes tipos de arestas, as *tree edges* e as *shadow edges*. Numa *shadow tree* cada *tree edge* é identificada pelo número da coluna da matriz *S* fornecida como entrada, o mesmo acontecendo para as *shadow edges,* porém, para efeito de diferenciação, as *shadow edges* são representadas por números negativos. Ou seja, para cada *tree edge* cujo identificador é *i* há na mesma árvore uma *shadow edge* com identificador *-i*

Além dos números identificadores, as arestas possuem estruturas chamadas de *connectors* em suas extremidades. Estes *connectors* também são divididos em dois grupos, representados pelas letras *H* (da palavra inglesa *Head*) e *T* (da palavra inglesa *Tail*). Um *connector* do tipo *H* está presente na cabeça da aresta (a extremidade mais próxima da raiz) enquanto um conector do tipo *T* está localizado na cauda da aresta (mais distante da raiz). Os conectores são identificados pela aresta que o contém seguido do seu tipo, como por exemplo 1H ou -1T.

Nas *shadow trees* também existem estruturas que ligam conectores de diferentes arestas. Estas estruturas, denominadas *links,* ligam um conector do tipo *H* de uma aresta a um conector (*H* ou *T*) de outra aresta. Semelhantemente às duas estruturas anteriormente descritas, os *links* são agrupados em dois diferentes tipos: *fixed-links* e *free-links*, os quais serão mais detalhados adiante.

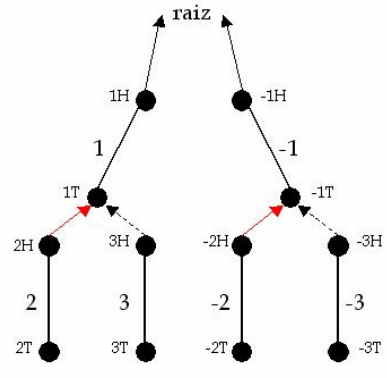
A última estrutura presente nas *shadow trees* é a *class*. Uma *class* é formada por *tree edges*, *shadow edges* e *fixed-links*. No caso mais simples os componentes de uma *class* são uma *tree edge* e a sua *shadow edge* correspondente. Quando acontece uma operação de *merge*, duas *classes* que estejam conectadas por *free-links* são agrupadas e os *links* que conectavam suas arestas se transformam em *fixed-links*. As Figuras 4 e 5 mostram os componentes de uma *shadow tree*.



Classe 1 = (raiz, 1, -1) / Classe 2 = (2, -2) / Classe 3 = (3, -3)

Figura 4: O nó mais alto é a raiz ao qual se liga às arestas 1 e -1. Os free-links são representados por linhas pontilhadas entre os conectores de duas classes distintas enquanto os fixed-links são linhas sólidas entre conectores de uma mesma classe.

Em cada classe, se os *links* forem contraídos, as arestas formarão duas sub-árvores com raiz (com exceção da classe raiz, a qual conterá apenas uma sub-árvore). As raízes destas sub-árvores são chamadas de "*class roots*" e por definição estas "*class roots*" são conectores do tipo *H*. Como pode ser visto na figura acima, cada classe X se liga à apenas uma única "classe-pai" (com exceção novamente da raiz), denominada p(X) através de *free-links* que saem das *class roots* de X e se conectam a conectores da classe p(X), que são denominados *join points*.

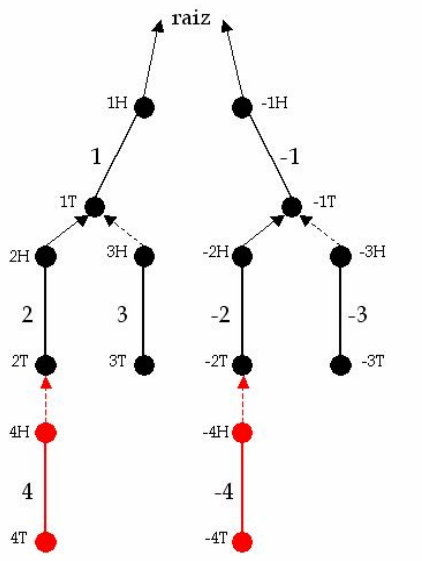


Classe 1 = (raiz, 1, -1, 2, -2) / Classe 3 = (3, -3)

Figura 5: Depois de uma operação de merge entre as classes 1 e 2 elas formam agora uma única classe.

2.2 - Operações sobre Shadow Trees

Existem basicamente três operações que podem ser realizadas numa *shadow tree*. A primeira é a adição de arestas, a qual é feita durante o processamento da matriz de entrada *S*. Como o próprio nome já diz, esta operação irá adicionar a *tree edge* X e sua correspondente *shadow edge* –X à uma árvore T da seguinte forma: é criada uma classe cujas únicas arestas são X e –X e estas arestas são ligadas através de *free-links* a certos conectores da árvore. Um exemplo desta operação pode ser observado na Figura 6.



Classe 1 = (raiz, 1, -1, 2, -2) / Classe 3 = (3, -3) / Classe 4 = (4, -4)

Figura 6: Após adição das edges 4 e –4 uma nova classe está presente na árvore. Uma propriedade importante que se deve notar numa shadow tree é que partindo das folhas da árvore em direção à raiz, os números das arestas são obrigatoriamente decrescentes.

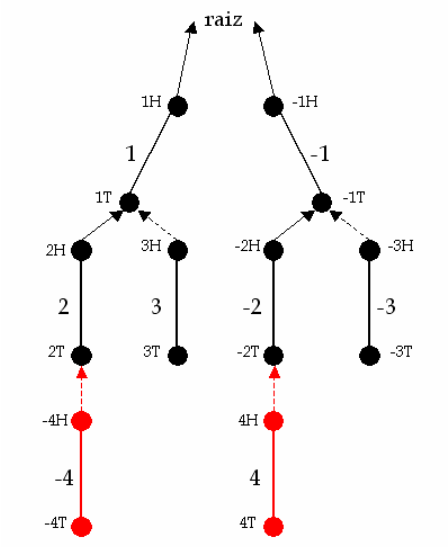
A segunda operação que pode ser aplicada em uma *shadow tree* é a operação de *class flipping*. Nesta operação uma classe X inverte a posição de suas arestas em relação à sua classe-pai, ou seja, dada uma aresta X em uma árvore T, a operação de *flipping* irá trocar a posição das arestas X e –X bem como de qualquer outra

aresta que esteja conectada a elas. Um exemplo da operação de *class flipping* pode ser visualizado na Figura 7.

A última das operações é a operação de *class merging*. Suponha duas classes X e X' numa árvore T, uma operação de *merge* só poderá ser aplicada nestas classes se elas satisfizerem uma, e somente uma, das seguintes condições:

- Uma das classes é pai da outra, por exemplo: p(X) = X'. Nesta situação, os *free-links* que ligam as *class roots* de X a p(X) se tornam *fixed-links* e as *class roots* de p(X) são designados *class roots* da nova classe. Um exemplo deste cenário é visto na Figura 7.

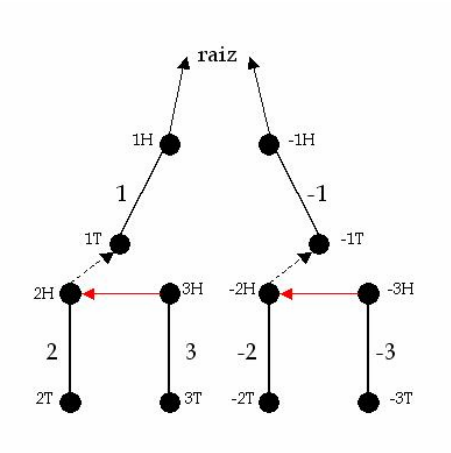
- As classes X e X' têm como pai a mesma classe Y, ou seja: p(X)=p(X') = Y. Neste caso os *links* que ligam as *class-roots de* X a p(X) se tornam *fixed-links* e passam a apontar para as *class roots* de X'. Após isso as *class roots* de X' se tornam os *class roots* da nova classe como é mostrado na Figura 8.



Classe 1 = (raiz, 1, -1, 2, -2, 4, -4) / Classe 3 = (3, -3)

Figura 7: Resultado dsa operações de flip e merge sobre a árvore da Figura 5.

Basicamente o que aconteceu foi que as arestas 4 e –4 trocaram de posição na árvore. Caso houvesse alguma aresta ligada as arestas 4 e –4 estas seriam transportadas junto com a classe envolvida no flip. Também houve neste exemplo uma operação de *merge* entre as classes 2 e 4. As *class roots* da classe 1 continuam sendo 1H e –1H.



Classe 1 = (1, -1) Classe 2 = (2, -2, 3, -3)

Figura 8: Árvore resultante de merge das classes 2 e 3 da Figura 5.

Como as classes 2 e 3 possuem o mesmo pai (classe 1) esta operação pode ser realizada. As *class roots* desta nova classe são 2H e –2H.

3. Propriedades do Algoritmo

Ding, Filkov e Gusfield [1] formularam um algoritmo para o problema PPH usando como estrutura de dados *Shadow Trees* e as operações de *merge*, *flip* e *edge addition*. O algoritmo mencionado recebe como entrada uma matriz de genotypes (seqüências ternárias sobre o alfabeto 0, 1 e 2) e tem como saída uma *shadow tree* que simboliza a solução do problema ou uma mensagem informando que tal solução não existe.

Algumas restrições são feitas para que o algoritmo funcione corretamente. A primeira delas é que a matriz *S* tenha suas colunas ordenadas em ordem decrescente de *leaf count*. O valor de *leaf count* para cada coluna é igual ao número de entradas 2 nesta coluna mais duas vezes o número de entradas 1. Ele então processa uma linha da matriz por vez e produz, ao final de cada linha $k + 1$, uma árvore correspondente à solução do PPH para as primeiras $k + 1$ linhas.

Este algoritmo de tempo linear mantém três listas durante o processamento de cada linha. A primeira delas é *OldEntryList* a qual é preenchida da seguinte forma: um número de coluna C_i estará armazenado em *OldEntryList* para a linha $k + 1$ se nesta linha a coluna C_i tem entrada 2 e em pelo menos uma das k linhas anteriores esta coluna teve entrada diferente de 0. A segunda lista é chamada de *NewEntryList*, a qual armazena quais colunas na linha $k + 1$ contém a entrada 2 pela primeira vez. Por último vem a lista *CheckList* que armazena os números das colunas de *OldEntryList* que precisam ser checados no procedimento *SecondPath*.

Algumas funções auxiliares também são necessárias para o funcionamento deste algoritmo. A função *col* recebe como entrada uma aresta ou conector e retorna o número da coluna correspondente. Uma segunda função *te* recebe uma coluna ou aresta e devolve a *tree edge* correspondente. Semelhantemente, a função *se* tem como entrada uma aresta ou número de coluna e como saída a *shadow edge* com este número.

3.1 – Descrição do algoritmo

Como já foi mencionado anteriormente, este algoritmo processa as n linhas da matriz de entrada, uma a cada vez. Ao final do processamento de cada linha k + 1 ele gera uma árvore T(k +1). Uma observação importante que deve ser ressaltada é que todas as arestas rotuladas por colunas que apresentam entrada 2 na linha k + 1 devem formar dois caminhos em direção a raiz e nenhum deles deve conter arestas cuja coluna correspondente tenha entrada 0 nesta linha.

Basicamente o algoritmo é dividido em quatro procedimentos. O primeiro deles *FirstPath* constrói um caminho em direção à raiz contendo algumas arestas presentes em *OldEntryList*. Ao fim da execução deste procedimento para uma linha k + 1 da matriz é gerada uma árvore denominada $T_{FP}(k)$. Do mesmo modo o procedimento *SecondPath* constrói um outro caminho com arestas cujos números estão em *CheckList* a partir da árvore $T_{FP}(k)$. Ao final deste procedimento a árvore $T_{SP}(k)$ é produzida. O subgrafo definido por estes caminhos formados pelos procedimentos citados anteriormente é chamado de *hipercaminho*. Em seguida é executado o procedimento *FixTree*, o qual ajusta a árvore $T_{SP}(k)$ identificando novas operações que necessitem ser feitas. As alterações que por ventura sejam feitas nas arestas pertencentes ao *hipercaminho* durante a execução deste procedimento resulta em um *hipercaminho estendido*. Finalmente é chamado o procedimento *NewEntries*. Este procedimento processa as colunas em *NewEntryList* e dá como resultado a árvore T(k +1).

Antes de executar os procedimentos principais o algoritmo constrói uma *filogenia perfeita inicial* T_i. Esta filogenia é construída como se segue: seja C_1 o conjunto das colunas de S que tenha pelo menos uma entrada 1, da mesma forma que R_1 é o conjunto das linhas que tem pelo menos uma entrada 1. Primeiramente, para cada linha em R_1, se cria um caminho em direção à raiz de T_i contendo arestas rotuladas por colunas de C_1 que tenham entrada 1 para aquela linha. Depois é feita uma interseção destes diferentes caminhos (um caminho para cada linha em R_1)

formando assim a filogenia perfeita inicial. A partir dela se constrói uma *shadow tree* inicial ST_i simplesmente transformando cada aresta de T_i em uma *tree edge* de ST_i e ligando estas *tree edges* através de *fixed-links.*

Outro conceito necessário para o pleno entendimento do algoritmo é o de *raiz de uma linha i* que nada mais é do que o conector *T* da *tree edge* em ST_i rotulada pela maior coluna com entrada 1 nesta linha *i*. Para matrizes que apresentam apenas 0's e 2's a raiz de todas as linhas é sempre o nó mais alto da árvore.

Uma árvore *T* é dita *contida* em uma *shadow tree ST* se ela puder ser obtida através de operações de inversão (*flip*) seguidas de contrações de *shadow edges* e *links* de *ST*. As contrações ocorrem de forma que os conectores *T* e *H* de cada *shadow edge* se transformam em um só. O mesmo procedimento deve ser tomado para os conectores ligados através de *links*.

3.1.1 – Procedimento *FirstPath*

Considere que as listas usadas neste procedimento, bem como nos próximos estão ordenadas decrescentemente, ou seja, a coluna com maior rótulo está na cabeça da lista. É feita uma varredura em *OldEntryList* partindo da coluna com maior número que ainda não tenha sido processada (C_i). Assuma também que C_j represente a coluna com o segundo maior número da lista. Caso a lista não tenha tal coluna, C_j é igual à raiz desta linha. Considere que E_p representa o pai da aresta *te*(C_i). Caso C_i e E_p não estejam na mesma classe considere E'_p o pai de *te*(C_i) depois de uma operação de *flip* sobre a classe de C_i. Se E'_p é uma *tree edge* e *col*(E_p) ≤ *col*(E'_p) então o algoritmo realizará a operação de *flip* da classe de C_i e atribuirá E_p a E'_p. Quando E_p é igual a C_j então o algoritmo marca C_i como processada e move para a próxima coluna de *OldEntryList*.

Porém existem três casos em que novas operações precisam ser feitas. O primeiro deles é quando E_p é uma *shadow edge*. Neste caso considere $E_p = p(E_p)$ e o procedimento deve ser executado novamente. Um segundo caso acontece quando E_p é uma *tree edge* e *col*(E_p) < C_j o que indica que *te*(C_i) e *te*(C_j) não poderão jamais

estar em um mesmo caminho em direção à raiz. O algoritmo acrescenta então C_j à *CheckList*. Por fim, o terceiro caso é quando E_p é uma *tree edge* mas *col*(E_p) > C_j o que indica que *col*(E_p) não tem entrada 2 na linha k + 1(já que se ela tivesse ela estaria presente na lista *OldEntryList* após C_j), e então o algoritmo executa uma operação de *flip* na classe que contem *te*(C_i) e em seguida faz um *merge* de *te*(C_i) e E_p.

3.1.2 – Procedimento *SecondPath*

O funcionamento do procedimento *SecondPath* é bastante similar ao de *FirstPath*. As principais diferenças são:

- Enquanto *FirstPath* manipula a lista *OldEntryList*, *SecondPath* trabalha com a lista *CheckList.*
- As operações neste procedimento são feitas sobre a árvore produzida no primeiro (T_{FP}).
- Quando E_p é uma *tree edge* e *col*(E_p) < C_j o algoritmo pára e avisa que não existe solução para esta entrada.
- Quando E_p é uma *tree edge* mas *col*(E_p) > C_j irão surgir duas possibilidades:
  - Se *col*(E_p) for igual a zero na linha k + 1 então o algoritmo realiza uma operação de *merge* seguida de uma operação de *flip* nas classe de *te*(C_i).
  - Caso *col*(E_p) esteja presente em *OldEntryList* mas não em *CheckList* (o que implica que E_p está em *FirstPath*) o procedimento chama outro procedimento (*DirectSecondPath*) que irá determinar que operação deverá ser executada.

3.1.2.1 – Procedimento *DirectSecondPath*

Seja E_k a *tree edge* em *FirstPath* que tem como pai E_p. Este procedimento visa assegurar que *FirstPath* e *SecondPath* não possuam nenhuma aresta em comum e para isso faz alguns testes:

- Se após uma operação de *flip* da classe de C_i e da classe que contém E_k, E_p estiver em ambos os caminhos *FirstPath* e *SecondPath* ou em nenhum dos deles o algoritmo pára sua execução e reporta que nenhuma solução existe para a matriz *S*.
- Caso E_p esteja na mesma classe de E_k então E_p deverá fazer parte de *FirstPath*.
- Caso E_p esteja na mesma classe de *te(*C_i*)* então E_p deverá fazer parte de *SecondPath*.
- Se após uma operação de *flip* envolvendo a classe *te(*C_i*)* tal que E_p esteja em *FirstPath*, não esteja em *SecondPath* e não exista um *hipercaminho* na árvore depois desta operação, então E_p deverá fazer parte de *SecondPath*.
- Se após uma operação de *flip* envolvendo a classe E_k tal que E_p esteja em *SecondPath*, não esteja em *FirstPath* e não exista um *hipercaminho* na árvore depois desta operação, então E_p deverá fazer parte de *FirstPath*.

Se o resultado destes testes indicarem que E_p deverá pertencer a *FirstPath* e *SecondPath*, isto indica que não existe solução para o PPH. Se os teste apontarem que E_p deverá obrigatoriamente em *FirstPath* então faça uma operação de *flip* na classe de C_i tal que E_p permaneça em *FirstPath*. O mesmo acontece quando os testes indicam que E_p deverá obrigatoriamente em *SecondPath*, assim o algoritmo realiza uma operação de *flip* na classe de E_k de tal forma que E_p pertença a *SecondPath*.

Há ainda o caso que E_p poderá pertencer tanto a *FirstPath* quanto a *SecondPath*. Quando isto acontece deverá ser executada uma operação de *flip* tal que E_p esteja em *FirstPath* mas não em *SecondPath* seguida por uma operação de *merge* das classes de C_i com a classe de E_k.

3.1.3 – Procedimento *FixTree*

Após a realização dos procedimentos que criam os dois caminhos em direção à raiz que contém as arestas cujas colunas estão em *OldEntryList* e *CheckList*, o algoritmo chama o procedimento *FixTree*. Neste procedimento serão executadas operações que removem árvores *contidas* em $T_{SP}(k)$ e que não estejam presentes em nenhuma solução. O seu primeiro passo é estender *SecondPath* com *shadow edges* cujos índices estejam em *OldEntryList*. O subgrafo definido por *FirstPath* e *SecondPath* estendido é chamado de *hipercaminho estendido*.

Considerando E_1 como sendo a *tree edge* da maior coluna presente em *OldEntryList*, ou seja, a aresta mais baixa em *FirstPath* e E_2 como sendo a *tree edge* da maior coluna em *OldEntryList* que não esteja em *FirstPath*, ou seja, a aresta mais baixa de *SecondPath*. Seja *P* o maior caminho de E_2 às folhas de $T_{SP}(k)$ que consiste apenas de *shadow edges* cujos índices estão presentes em *OldEntryList* e E'_2 a aresta mais baixa de *P*, caso *P* não contém nenhuma aresta então $E'_2 = E_2$. Isto deverá ser repetido até que E_1 e E'_2 estejam na mesma classe ou até que E'_2 seja a *class root* da classe de *se*(E_1) . Quando o índice da coluna da aresta que contém a *class root* da classe de E_1 for maior que o índice da coluna da aresta que contem a *class root* de E_2, então deverá ser executada uma operação de *merge* da classe de E_1 com a classe a qual ela está ligada, caso contrário a operação de *merge* envolverá a classe de E_2 e sua classe pai.

3.1.4 – Procedimento *NewEntries*

Por fim, o algoritmo exposto em Ding, Filkov e Gusfield [1] contém o procedimento que processa as entradas da lista *NewEntryList*. Como já foi explicado anteriormente, esta lista contém o número das colunas que apresentam entrada 2 na linha $k + 1$ porém não apresentam nenhuma entrada diferente de 0 entre as linhas 1 e k. Claramente este procedimento irá realizar operações de *edge addition* sobre a árvore $T_{FT}(k)$, adicionando assim as arestas cujos índices estejam em *NewEntryList*.

Caso esta lista esteja vazia o procedimento é abortado e o algoritmo passa a processar a linha $k + 2$, caso isso não ocorra, a lista *NewEntryList* deverá ser arrumada de tal forma que o maior índice esteja mais à direita da lista e seja o primeiro a ser analisado. O procedimento então cria para cada coluna em *NewEntryList* as respectivas *tree edges* e *shadow edges.* Após isso são criados *free-links* que ligam o conector *H* de $te(C_i)$ ao conector *T* de $te(C_j)$ e outros *free*-links que ligam o conector *H* de $se(C_i)$ ao conector *T* de $se(C_j)$, onde C_j é o próximo elemento a ser analisado. Quando C_i for o último elemento da lista, ou seja, a menor entrada na lista ele é chamado de C_h e dois casos podem ocorrer.

Considerando E_1, E_2 e E'_2 como sendo os mesmos do procedimento *FixTree*. O primeiro caso acontece quando $col(E_1) < C_h$ e $te(C_h)$ e $se(C_h)$ estão conectadas aos dois finais do *hipercaminho estendido*. O procedimento então cria um *free-link* que aponta do conector *H* de $te(C_h)$ para o conector *T* de E_1 e cria também outro *free-link* que aponta do conector *H* de $se(C_h)$ para o conector *T* de E'_2, caso E'_2 e E_1 pertençam a mesma classe ou então para um conector na classe de E_1, cujo pai seja E'_2 caso não pertençam.

O segundo caso é quando $col(E_1) > C_h$. Se $col(E_2) > C_h$ então não existe solução para o PPH, caso contrário o procedimento acha duas arestas (E_{h1} e $E_{h2'}$) as quais adicionar $te(C_h)$ e $se(C_h)$ do seguinte modo: seja E_{h1} a *tree edge* de maior

índice em *OldEntryList* que seja menor do que C_h, e E_{h2} a *tree edge* de maior índice em *OldEntryList* que seja menor do que C_h e não esteja no caminho de E_{h2} à raiz. Caso E_{h1} ou E_{h2} considere-os como sendo a raiz.

O próximo passo é achar o caminho mais longo de E_{h2} em direção às folhas de $T_{FT}(k)$ composto por *shadow edges* cujos índices das colunas correspondentes estejam em *OldEntryList* e sejam menores do que C_h. Seja $E_{h2'}$ a aresta que esteja no ponto mais baixo deste caminho.

Se E_{h1} estiver no caminho de E_1 à raiz o algoritmo cria um *free-link* que aponta do conector *H* de *se*(C_h) para o conector *T* de E_{h1} e cria também um *free-link* que aponta do conector *H* de *te*(C_h) para o conector tipo *T* de $E_{h2'}$ se E_{h1} e $E_{h2'}$ pertencerem à mesma classe ou para um conector na classe de E_{h1} cujo pai é $E_{h2'}$. Caso E_{h1} estiver no caminho de E'_2 à raiz então o algoritmo cria um *free-link* que aponta do conector *H* de t*e*(C_h) para o conector *T* de E_{h1} e cria também um *free-link* que aponta do conector *H* de *se*(C_h) para o conector tipo *T* de $E_{h2'}$ se E_{h1} e $E_{h2'}$ pertencerem à mesma classe ou para um conector na classe de E_{h1} cujo pai é $E_{h2'}$.

Caso existam colunas em *NewEntryList* cujos índices sejam maiores do que *col*(E_1) considere C_t a menor delas. Neste caso *se*(C_t) é uma nova aresta que foi ligada a outra aresta pelo algoritmo. Neste caso especial o procedimento muda o link conector *H* de *se*(C_t) que passa a apontar para o conector *T* de E_1.

O algoritmo por fim executa operações de *merge* entre a classe de C_h e classes que contenham arestas com índices em *NewEntryList* que sejam menores do que *col*(E_1) e outras *merges* entre a classe de C_h com classes cujas arestas estejam em *OldEntryList* que sejam maiores do que C_h.

3.2 – Exemplo de Execução do Algoritmo sem Entradas 1

O pseudo-código do Algoritmo pode ser encontrado no Apêndice B do artigo de Ding, Filkov e Gusfield [1]. Abaixo será mostrado um exemplo de sua execução através do processamento de uma matriz de entrada *S*.

$$S = \begin{pmatrix} 2\,2\,2\,0\,0\,0\,0 \\ 2\,0\,0\,0\,2\,2\,0 \\ 2\,2\,2\;\;2\,0\,2\,2 \\ 2\,2\,2\,2\,0\,0\,0 \\ 2\,0\,0\,0\,2\,0\,0 \end{pmatrix}$$

Como já foi explicado anteriormente, o algoritmo irá processar uma linha de cada vez, Neste caso específico, como só há entradases 0 e 2 na matriz de entrada, a *shadow tree* incial ST_i é formada exclusivamente pelo nó raiz. Para a primeira linha, logicamente, os procedimentos *FirstPath*, *SecondPath* e *FixTree* não executam nenhuma ação, pois não há nenhum elemento nas listas *OldEntryList* e *CheckList*. Já a lista *NewEntryList* possui três elementos (*NewEntryList* = [1, 2, 3]) que representam as colunas que contém entradas 2 pela primeira vez. Então o algoritmo chama o procedimento *NewEntries*. Neste procedimento serão criadas as arestas cujos números de colunas estejam em *NewEntryList* de tal forma que as arestas que formam o caminho das folhas à raiz tenham rótulos decrescentes.

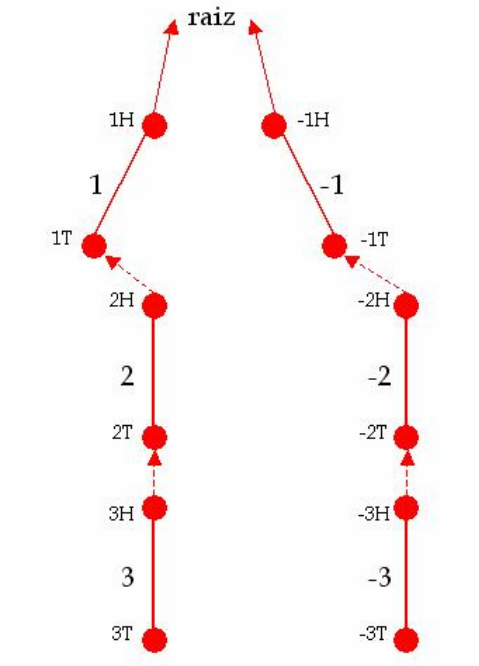


Figura 9: Árvore resultante da execução de NewEntries

O processamento da segunda linha de *S* é bastante similar ao da primeira, com a exceção de que agora os procedimentos *FirstPath* e *FixTree* são executados uma vez que há elementos na lista *OldEntryList* (*OldEntryList* = [1]). Porém nenhuma alteração sobre a árvore T(1) é feita. Desta forma $T_{FP}(1) = T_{SP}(1) = T_{FT}(1) = T(1)$. Somente quando o procedimento *NewEntries* é chamado ocorrem operações na *shadow tree*. Como esta lista tem dois elementos (*NewEntryList* = [5, 6]), a primeira coisa a se fazer é criar *tree* e *shadow edges* para estas colunas. O algoritmo então deve decidir onde posiciona-las na árvore. Como a coluna 1 está em *OldEntryList* o algoritmo decide ligar as novas arestas às arestas 1 e –1, como pode ser visto na Figura 10.

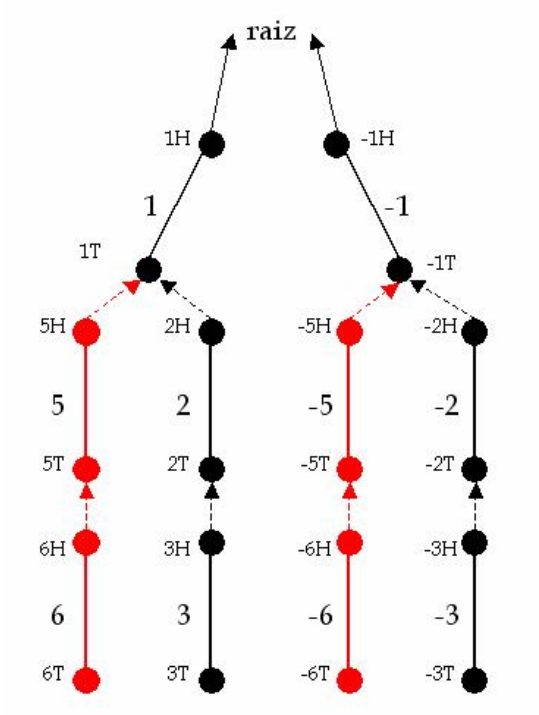


Figura 10: Árvore resultante da execução de NewEntries com a segunda linha da matriz.

A partir da terceira linha é que transformações mais profundas podem ser vistas nas árvores. Isto é devido ao fato de que nesta linha várias colunas estarão presentes em *OldEntryList*.

Quando o procedimento *FirstPath* é executado ele irá criar um caminho em direção à raiz usando algumas arestas cujos números de colunas estejam em

OldEntryList. Para isso ele irá executar operações de *flip* e *merge*. A árvore resultante da execução de *FirstPath* sobre a terceira linha de *S* é mostrado na Figura 11. É interessante notar que o caminho construído por este procedimento consiste nas arestas 6, -5 e 1. Durante a execução de *FirstPath*, as entradas 2 e 3 que estavam em *OldEntryList* são transferidas para *CheckList*.

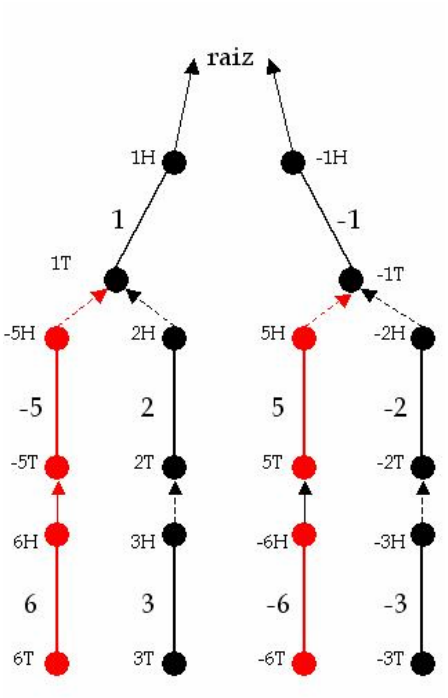


Figura 11: $T_{FP}(2)$, árvore resultante da execução de FirstPath com a terceira linha da matriz.O caminho gerado por este procedimento contém as arestas 6, -5 e 1.

Agora é então executado o procedimento *SecondPath*. Este procedimento é responsável por construir um segundo caminho à raiz constituído de arestas presentes em *CheckList* e que não use nenhuma aresta de *FirstPath*. Neste caso particular, *SecondPath* não vai conseguir criar este segundo caminho citado, pois a aresta 1 já está presente em *FirstPath*, ele então delega esta tarefa de decidir se a arestas 1 faz parte de *FirstPath* ou *SecondPath* a um procedimento auxiliar denominado *DirectSecondPath*.

Claramente, neste caso, é indiferente a posição da *tree edge* 1. Portanto para efeito de facilidade o procedimento *DirectSecondPath* deixa a aresta 1 em *FirstPath* e forma *SecondPath* com as arestas 3, 2 e –1. A Figura 12 mostra a árvore $T_{SP}(2)$.

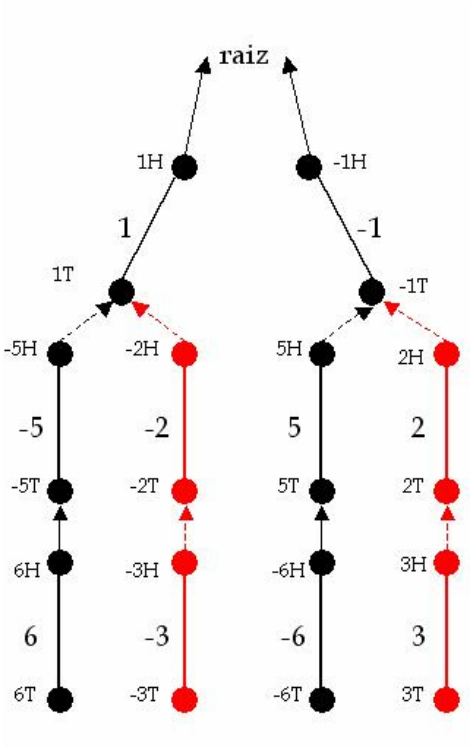


Figura 12: $T_{SP}(2)$, árvore resultante da execução de SecondPath sob a terceira linha da matriz.O caminho gerado por este procedimento contém as arestas 3, 2 e -1.

Após isto é necessário identificar quais árvores estão contidas em $T_{SP}(2)$ mas que não fazem parte de nenhum solução do problema. Neste caso o algoritmo irá realizar uma operação de *merge* entre as classes (5, -5, 6, -6) e (2, -2) conforme pode ser verificado na Figura 13.

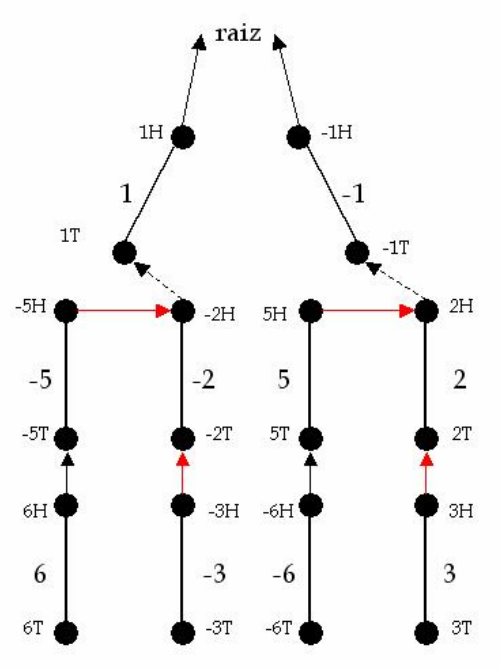


Figura 13: $T_{FT}(2)$, árvore resultante da execução de FixTree sob a terceira linha da matriz. A operação de merge realizada impede que soluções erradas estejam na shadow tree final.

A lista *NewEntriesList*, para a terceira linha, contém os elementos 4 e 7. O procedimento *NewEntries* deve achar uma forma de anexar as arestas cujas colunas estão em *NewEntriesList* à $T_{FT}(2)$. A Figura 14 mostra a árvore T(3).

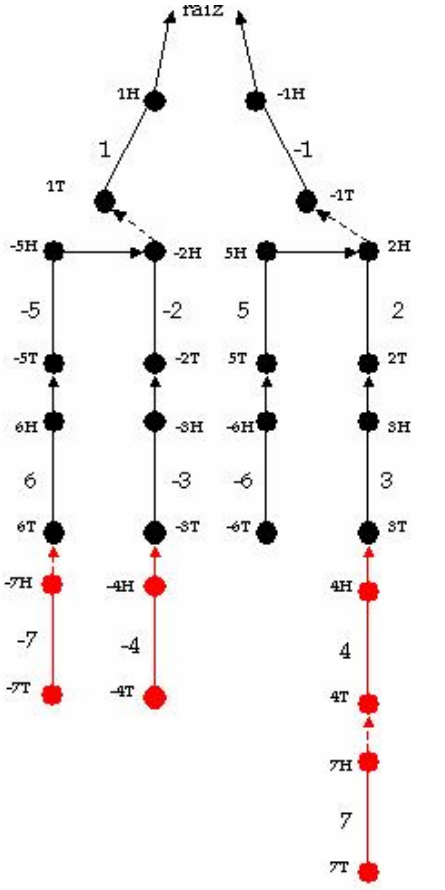


Figura 14: T(3), árvore resultante da execução de NewEntries sob a terceira linha da matriz.

A partir da linha 4 da matriz *S* não existem mais colunas em *NewEntriesList*, pois todas elas já tiveram entradas 2 entre as linhas 1 e 3. Então o procedimento *NewEntries* não pode mais alterar a *shadow tree*. O algoritmo deve agora executar apenas operações de *flip* e *merge* sobre à arvore T(3).

Na linha 4, a lista *OldEntryList* contém os elementos 1, 2, 3 e 4. O objetivo do procedimento *FirstPath* deve ser então construir um caminho em direção à raiz de T(3) usando o máximo de arestas cujas colunas estejam em *OldEntryList*. Neste caso uma simples operação de *flip* possibilita que esta condição seja feita e ainda faz com que *FirstPath* contenha todos os elementos de *OldEntryList* tornando desnecessária a execução de *SecondPath* como pode ser identificado na Figura 15.

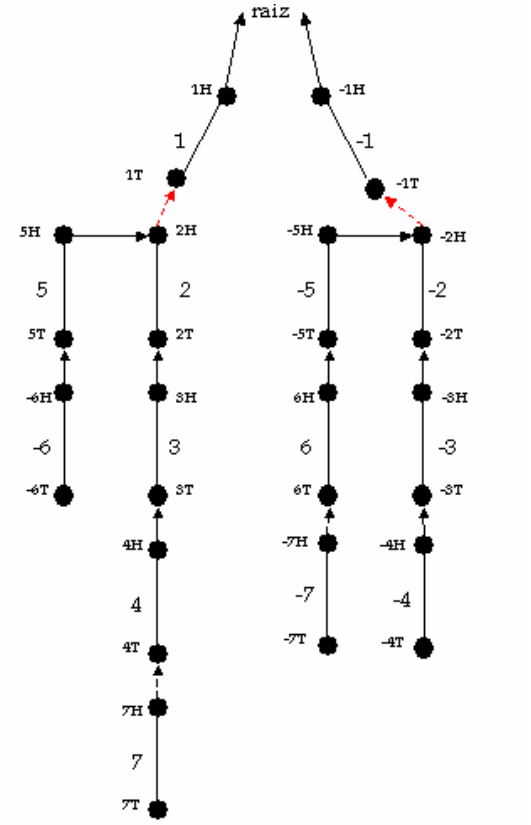


Figura 15: $T_{FP}(3) = T_{SP}(3) = T_{FT}(3) = T(4) = T(5)$, árvore resultante da execução de FirstPath sob a quarta linha da matriz.

Finalmente a última linha da matriz é processada. Novamente não existe nenhum elemento em *NewEntryList* e o algoritmo tratará de arrumar a árvore de tal forma que ela represente todas as soluções para o problema. Como *OldEntryList* contém apenas dois elementos (1 e 5) é fácil verificar que o procedimento *FirstPath* não realizará nenhuma operação, pois estas duas arestas já estão em um caminho em direção à raiz. Da mesma forma *SecondPath* e *FixTree* não irão alterar a árvore da Figura 15. Portanto a representação das soluções para o PPH para a matriz *S* é mostrada na figura anterior.

3.3 – Exemplo de Execução do Algoritmo com Entradas 1

O funcionamento do algoritmo para matrizes que tenham entradas 1 é exatamente igual ao descrito anteriormente mas é interessante mostrar um exemplo de seu funcionamento pois a shadow tree inicial que é construída antes da chamada do primeiro procedimento não é constituída apenas pela raiz. Considerando a matriz:

$$S = \begin{pmatrix} 10220 \\ 10200 \\ 12000 \\ 22002 \end{pmatrix}$$

A *shadow tree* inicial, obviamente não é formada somente pela raiz. Sua composição é exibida na Figura 16.

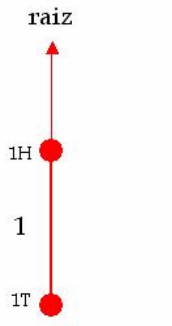


Figura 16: Shadow tree inicial (ST_i) para a matriz S.

Neste caso R_1 = (1, 2, 3) e C_1 = (1). Para se formar ST_i é preciso construir, para cada linha em R_1, um caminho em direção à raiz constituído apenas por arestas cujo rótulo esteja em C_1 e depois fazer a interseção destes caminhos. Todas as arestas de ST_i são ligadas através de *fixed-links*. Não há *shadow edges* na shadow tree inicial.

A partir da construção desta *shadow tree* inicial é iniciado o processamento da primeira linha. Mais uma vez o algoritmo não executa os passos *FirstPath, SecondPath* e *FixTree* já que *OldEntryList* e *CheckList* estão vazias. Porém, a execução de *NewEntries* se faz oportuna, pois *NewEntryList* contém os elementos 3 e 4. Sendo assim são criadas as *tree edges* e *shadow edges* correspondentes e estas são anexadas através de *fixed-links* à raiz de St_i, que neste exemplo é o *T connector* 1T., conforme pode ser verificado na próxima figura.

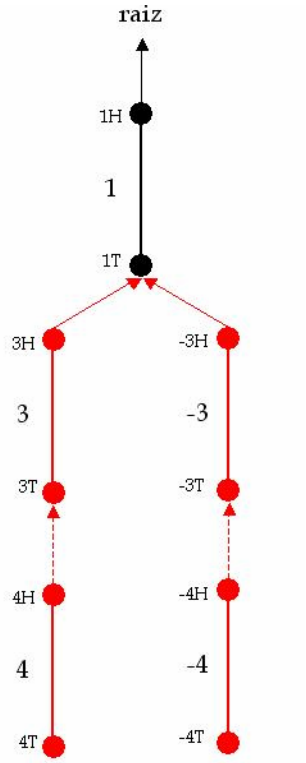


Figura 17: As arestas 3, -3, 4 e –4 são criadas e colocadas na árvore de tal forma que a aresta com menor número se ligue através de fixed-links à raiz (1T)

Quando do processamento da segunda linha de *S* o algoritmo verifica que *OldEntryList* possui apenas um elemento e o pai dele é uma *tree edge* (1 é pai de 3) então o algoritmo não executa nenhuma alteração na árvore. Por isso a Figura 17 é a representação de T(1) e T(2).

Já na linha 3 as arestas 2 e –2 devem ser anexadas à T(2), já que *NewEntryList* contém a coluna 2. Por outro lado *OldEntryList* está vazia e por isso os

procedimentos *FirstPath, SecondPath* e *FixTree* não executam nenhuma ação. No procedimento *NewEntries* o algoritmo tentará ligará as arestas 2 e –2 à raiz desta linha que é mais uma vez 1T. A Figura 18 mostra o resultado.

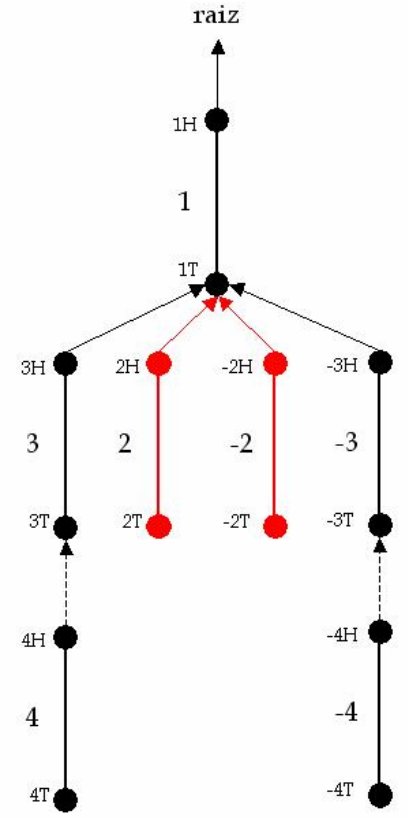


Figura 17: As arestas 2 e -2 são criadas e ligadas à raiz da linha 3 através de fixed-links

Por fim é processada a última linha. Durante a execução de *FirstPath* acontece o caso ideal, quando os elementos de *OldEntryList* formam um caminho em direção à raiz formado apenas por *tree edges*. Sendo assim nenhum elemento é adicionado à CheckList e novamente o algoritmo não realiza nenhuma ação nos três primeiros procedimentos. Na execução de *NewEntries* o algoritmo cria as arestas 5 e –5 e as anexa à aresta 2 e à raiz respectivamente, conforme é visto na Figura 18.

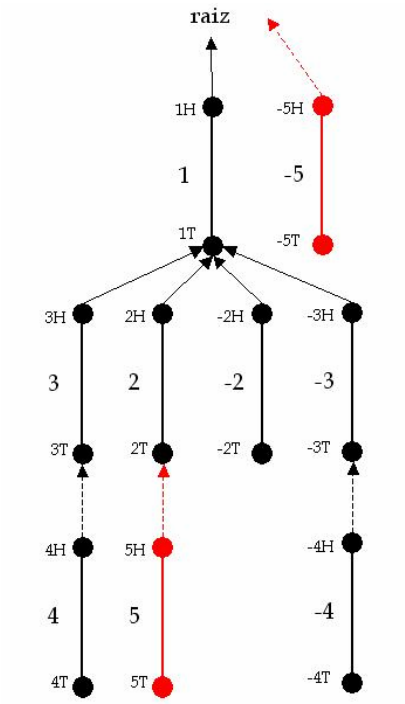


Figura 18: As arestas 5 e -5 são criadas e ligadas à aresta 2 e à raiz da linha 4.

3.4 – Mapeamento das *Shadow Tree* para a solução do PPH

Na definição do problema de inferência de haplotypes foi dito que o número de soluções para cada linha de S é dado pela fórmula (2^{d-1}), onde d é igual ao número de entradas 2 nesta linha. Entretanto a maioria destas soluções não é solução para o PPH uma vez que elas não formam uma filogenia perfeita. Mesmo assim pode acontecer de mais de uma solução ser possível para uma entrada S e todas elas devem estar representadas na *shadow tree* final produzida pelo algoritmo. O número de soluções representadas por uma árvore ST é dado pela fórmula 2^{p-1} onde p é o número de classes distintas em ST.

Uma árvore T é dita *contida* em uma *shadow tree* ST se ela puder ser obtida através de operações de inversão (*flip*) seguidas de contrações de *shadow edges* e *links* de ST. As contrações ocorrem de forma que os conectores T e H de cada *shadow edge* se transformam em um só. O mesmo procedimento deve ser tomado

para os conectores ligados através de *links*. As Figuras 19 e 20 mostram exemplos destes conceitos.

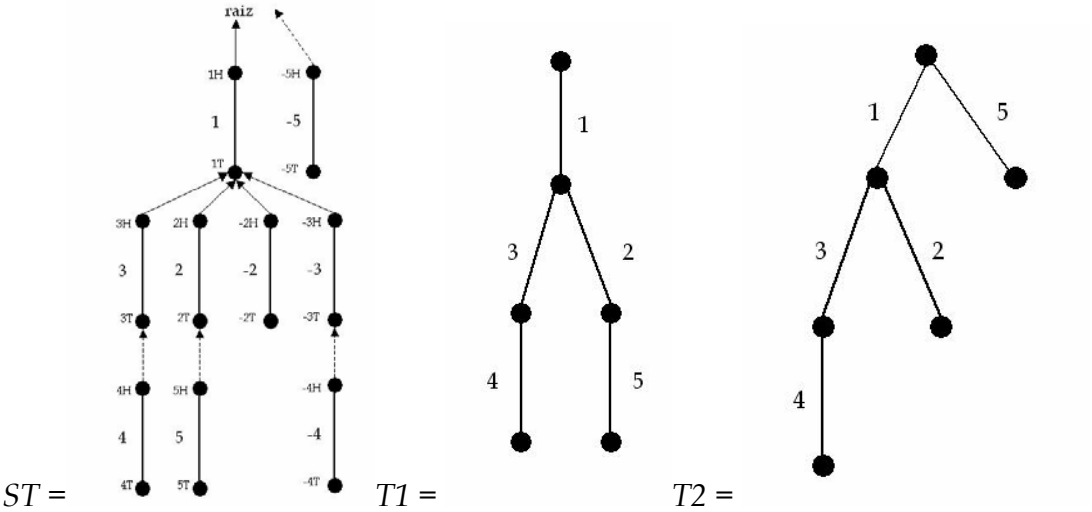


Figura 19: A shadow tree ST é uma representação de todas as soluções para a matriz S mostrada na página 34.

Como é possível notar existem quatro soluções distintas para a matriz S, pois *ST* apresenta três classes. Ao lado dela são exibidas algumas árvores que estão contidas em *ST*. A árvore *T1* é obtida através da contração de todos os *links* e *shadow edges* de *ST*. Já a árvore *T2* é obtida depois de uma operação de *flip* da classe (5, -5) seguida das contrações necessárias. O modo de se obter as seqüências binárias através destas árvores será explicado mais adiante.

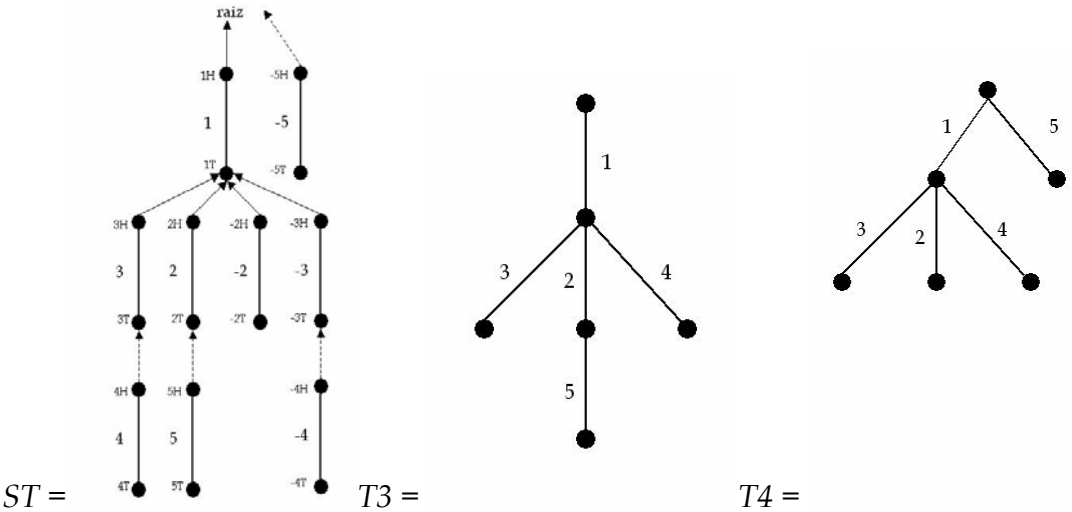


Figura 20: As outras árvores contidas em ST

Ao lado de *ST* estão as outras duas soluções para o problema. A árvore *T3* foi gerada através de uma operação de *flip* da classe(4, -4) seguida da contração de *links* e *shadow edges*. Já na árvore *T4* foram executadas 2 inversões, a primeira envolveu a classe (4, -4) e a outra foi na classe (5, -5).

O artigo de Ding, Filkov e Gusfield [1] não sugere uma maneira de extrair as seqüências a partir das árvores contidas na *shadow tree* final embora isto possa ser trivialmente obtido. Através de comunicação pessoal com um dos autores do artigo [11] foi sugerido que para obter tais seqüências em tempo razoável o seguinte algoritmo é sugerido:

- Para cada linha de *S* seja *c* a entrada 2 mais à direita (com maior número de coluna). Nesta coluna a primeira seqüência de haplotype vai ter entrada 1 e a segunda entrada 0. Para cada uma das outras entradas 2 desta linha, se o seu número de coluna rotular alguma aresta da árvore no caminho de *c* à raiz então esta coluna vai ter a primeira seqüência de haplotype (*hap1*) com entrada 1 e a segunda (*hap2*) com entrada 0. Caso contrário a primeira seqüência tem entrada 0 naquela coluna e a segunda tem entrada 1.

Considere a árvore T1 da Figura 19. O algoritmo exposto acima funciona da seguinte forma: a linha 1 é (1 0 2 2 0), sendo assim sua coluna mais à direita é $c = 4$. A coluna de número 4 da primeira seqüência de haplotype desta linha vai ter entrada 1 e a segunda entrada 0. A única coluna nesta linha que apresenta 2 é a coluna 3. O algoritmo verifica que em T1 a aresta 3 está no caminho da aresta 4 à raiz. Desta forma a entrada da coluna 3 na primeira seqüência vai ser 1 e na segunda 0. Como esta linha não tem mais entradas 2 ambas as seqüências possuem entradas, nas outras colunas, iguais às da linha processada. Portanto as seqüências que explicam a primeira linha são: *hap1* = "1 0 1 1 0" e *hap2* = "1 0 0 0 0".

No caso da segunda linha acontece a mesma coisa. A coluna com entrada 2 de maior índice vai ser a coluna 3, então *hap1* terá entrada 1 nesta coluna e *hap2* terá entrada 0. As seqüências que explicam a linha 2 são: *hap1* = "1 0 1 0 0" e *hap2* = "1 0 0 0 0". A linha 3 é bastante similar à linha 2já que *hap1* e *hap2* serão iguais em 4 de suas colunas, diferindo apenas na segunda coluna, onde *hap1* terá entrada 1 e *hap2* entrada 0: *hap1* = "1 1 0 0 0" e *hap2* = "1 0 0 0 0".

A última linha não difere em nada das 3 primeiras. Como existe um caminho em T1 formado pelas arestas cujos índices são exatamente iguais aos das colunas que têm entrada 2 nesta linha *hap1* terá entrada 1 nestas colunas enquanto *hap2* terá 0: *hap1* = "1 1 0 0 1" e *hap2* = "0 0 0 0 0".

Considerando agora a árvore T4 o processamento das linhas vai ser um pouco diferente. No caso da primeira linha a coluna com entrada 2 de maior índice é novamente 4. Neste caso *hap1* terá 1 nesta coluna e *hap2* entrada 0. Porém não existe caminho da aresta 4 à raiz que passe pela aresta 3, que também tem entrada 2 nesta linha. Sendo assim *hap1* vai ter 0 na terceira coluna enquanto *hap2* terá entrada 1. Desse modo: *hap1* = "1 0 0 1 0" e *hap2* = "1 0 1 0 0". Nas linhas 2 e 3 não há diferença em relação à T1 porém na linha 4 ocorre uma pequena diferença. Nesta linha a coluna de maior índice que tem entrada 2 é a coluna 5, desse modo *hap1* tem entrada 1 nesta coluna e *hap2* entrada 0. Claramente não há nenhum caminho que vá da aresta 5 à raiz e que passe por 2 ou 1 (as outras colunas com entrada 2 nesta linha). Portanto em ambas as colunas *hap1* terá entrada 0 e *hap2* entrada 1. Abaixo são mostradas as seqüências que resolvem *S* para cada árvore.

$$S = \begin{pmatrix} 10220 \\ 10200 \\ 12000 \\ 22002 \end{pmatrix}$$

Considerando T1:

Linha 1 = 1 0 **2 2** 0

Hap1 = 1 0 **1 1** 0

Hap2 = 1 0 **0 0** 0

Linha 2 = 1 0 **2** 0 0

Hap1 = 1 0 **1** 0 0

Hap2 = 1 0 **0** 0 0

Linha 3 = 1 **2** 0 0 0

Hap1 = 1 **1** 0 0 0

Hap2 = 1 **0** 0 0 0

Linha 4 = **2 2** 0 0 **2**

Hap1 = **1 1** 0 0 **1**

Hap2 = **0 0** 0 0 **0**

Considerando T2:

Linha 1 = 1 0 **2 2** 0

Hap1 = 1 0 **1 1** 0

Hap2 = 1 0 **0 0** 0

Linha 2 = 1 0 **2** 0 0

Hap1 = 1 0 **1** 0 0

Hap2 = 1 0 **0** 0 0

Linha 3 = 1 **2** 0 0 0

Hap1 = 1 **1** 0 0 0

Hap22 = 1 **0** 0 0 0

Linha 4 = **2 2** 0 0 **2**

Hap1 = **0 0** 0 0 **1**

Hap22 = **1 1** 0 0 **0**

Considerando T3:

Linha 1 = 1 0 **2 2** 0

Hap1 = 1 0 **0 1** 0

Hap2 = 1 0 **1 0** 0

Linha 2 = 1 0 **2** 0 0

Hap1 = 1 0 **1** 0 0

Hap2 = 1 0 **0** 0 0

Linha 3 = 1 **2** 0 0 0

Hap1 = 1 **1** 0 0 0

Hap22 = 1 **0** 0 0 0

Linha 4 = **2 2** 0 0 **2**

Hap1 = **1 1** 0 0 **1**

Hap22 = **0 0** 0 0 **0**

Considerando T4:

Linha 1 = 1 0 **2 2** 0

Hap1 = 1 0 **0 1** 0

Hap2 = 1 0 **1 0** 0

Linha 2 = 1 0 **2** 0 0

Hap1 = 1 0 **1** 0 0

Hap2 = 1 0 **0** 0 0

Linha 3 = 1 **2** 0 0 0

Hap1 = 1 **1** 0 0 0

Hap22 = 1 **0** 0 0 0

Linha 4 = **2 2** 0 0 **2**

Hap1 = **0 0** 0 0 **1**

Hap22 = **1 1** 0 0 **0**

3.5 – Exemplo Completo de Execução do Algoritmo

Depois de apresentados todos os conceitos e formas de execução do algoritmo, é válido mostrar um exemplo de execução deste a partir do começo, desde a ordenação das colunas até a obtenção das seqências que explicam a matriz de entrada.

$$S = \begin{pmatrix} 2 & 2 & 0 & 2 & 0 & 2 \\ 0 & 2 & 2 & 1 & 0 & 0 \\ 2 & 2 & 2 & 2 & 0 & 0 \\ 0 & 2 & 2 & 1 & 2 & 0 \end{pmatrix}$$

2 4 3 6 1 1

O valor de *leaf count* para cada uma das colunas está em negrito abaixo destas. Elas devem ser ordenadas agora de tal forma que a coluna mais à esquerda tenha o maior valor de *leaf count*.

$$S = \begin{pmatrix} 2\,2\,0\,2\,0\,2 \\ 1\,2\,2\,0\,0\,0 \\ 2\,2\,2\,2\,0\,0 \\ 1\,2\,2\,0\,2\,0 \end{pmatrix}$$

As linhas devem agora ser ordenadas para que a primeira linha tenha a entrada 1 mais à direita, a segunda linha a segunda entrada 1 mais à direita e assim por diante. A matriz ordenada e pronta para ser executada é:

$$S = \begin{pmatrix} 1\,2\,2\,0\,0\,0 \\ 1\,2\,2\,0\,2\,0 \\ 2\,2\,0\,2\,0\,2 \\ 2\,2\,2\,2\,0\,0 \end{pmatrix}$$

Primeiramente deve-se construir a filogenia inicial para a matriz. Novamente esta filogenia é composta apenas da raiz e da *tree edge* 1. A seguir é processada a primeira linha. Como em todos os outros casos, a execução do algoritmo dos procedimentos *FirstPath, SecondPath* e *FixTree* para a primeira linha não modifica a filogenia inicial. Todas as modificações ocorrem no procedimento *NewEntries*.

No caso da primeira linha a lista *NewEntryList* contém os elementos 2 e 3. O algoritmo então cria as *tree* e *shadow edges* correspondentes a estas colunas e liga as arestas 3 e –3 às arestas 2 e –2 respectivamente.

O processamento da segunda linha também é simples. Nesta linha a lista *OldEntryList* contém as colunas 2 e 3 e, como ambas já estão presentes na *shadow tree* T(1) e a classe que contém 2 é a classe-pai da classe que contém 3, nenhuma operação é realizada pelos procedimentos *FirstPath, SecondPath* e *FixTree*. Ainda nesta linha *NewEntryList* contém a coluna 5 e as arestas correspondentes (5 e -5)

devem ser ligadas às folhas de T(1). A figura abaixo mostra o resultado depois do processamento das duas primeiras linhas.

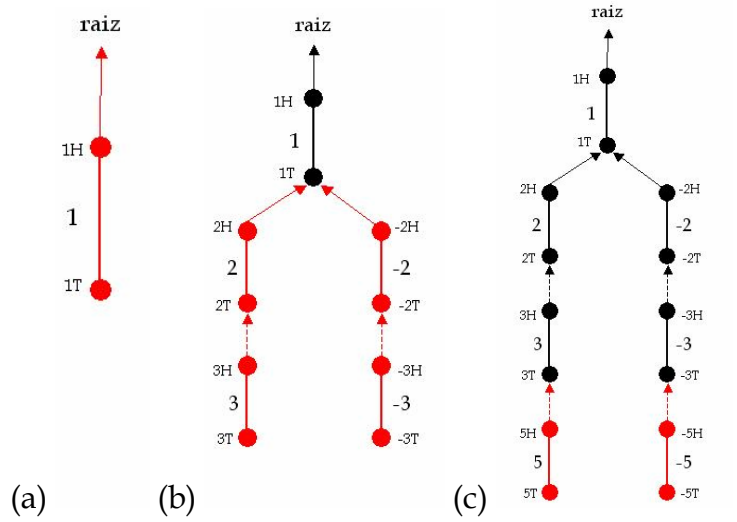


Figura 21. (a) Filogenia inicial construída para a matriz S. (b) A execução do algoritmo para a primeira linha adiciona as arestas 2 e 3 à filogenia inicial. (c) O algoritmo adiciona as arestas 5 e –5 de acordo com o primeiro caso explicado na descrição de NewEntries, quando col(E_1) < C_h.

O processamento da terceira linha é um pouco diferente do das duas primeiras. Mesmo assim os procedimentos *FirstPath*, *SecondPath* e *FixTree* novamente não realizam nenhuma operação sobre a árvore, pois as colunas em *OldEntryList* (1 e 2) já estão ordenadas, ou seja, uma é pai da outra (a aresta 1 é pai da aresta 2). Porém no processamento das novas entradas ocorre um caso interessante. Nesta linha, a lista *NewEntryList* é formada pelas colunas 4 e 6 e o algoritmo deve achar o lugar adequado para anexar as arestas correspondentes. Semelhantemente ao que ocorreu na linha 1, as arestas 4, -4, 6 e –6 são criadas e as duas últimas são ligadas através de *free-links* às duas primeiras. O algoritmo identifica as arestas E_1 (aresta correspondente à maior coluna em *OldEntryList*), E_2 (*tree edge* correspondente a maior aresta cujo número de coluna esteja em *OldEntryList* e que não faça parte do caminho de E_1 à raiz) e E'_2 (a folha do maior caminho composto por *shadow edges* cujos números de colunas estejam em *OldEntryList*) que são, respectivamente, as arestas 2, raiz e raiz.

Cabe agora compara o número da menor coluna em *NewEntryList* (4) e o número de coluna da aresta E_1 (2). Como 4 é maior do que 2 o algoritmo decide ligar a sub-árvore formada pelas arestas (4, 6) à E_1 e a sub-árvore (-4, -6) à E'_2. O resultado do processamento da terceira linha é mostrado na figura abaixo.

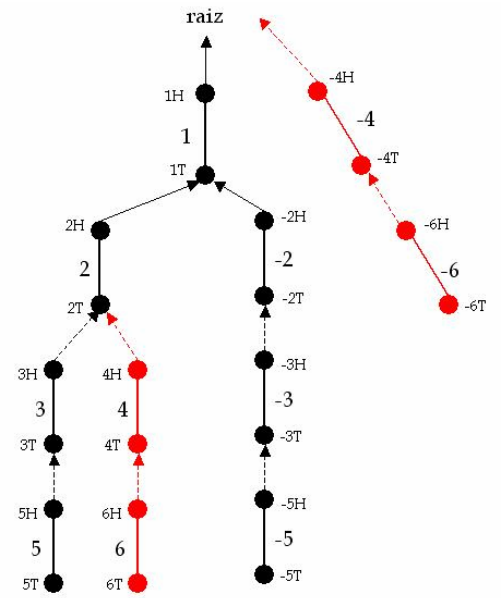


Figura 22: Resultado da adição das arestas 4, -4, 6 e –6. Esta adição se encaixa com o primeiro caso citado na descrição do procedimento NewEntries

Ao contrário das três primeiras linhas, quando nenhuma alteração nas árvores foi feita pelos procedimentos *FirstPath, SecondPath* e *FixTree*, durante o processamento desta linha estes procedimentos realizarão algumas operações, enquanto *NewEntries* não será mais executado, já que a lista *NewEntryList* está vazia.

Como já foi dito anteriormente o procedimento *FirstPath* tentará criar um caminho em direção a raiz da linha 3 usando o número máximo de arestas cujos rótulos estejam em *OldEntryList* que nesta linha é composta pelas colunas (1,2,3,4). Obviamente este caminho não poderá conter todas as colunas desta lista pois as arestas 3 e 4 jamais poderão fazer parte de um mesmo caminho. Em resumo o algoritmo decide que as arestas 3, 2 e 1 devem formar um caminho em direção a

raiz e que o outro caminho deve ser formado exclusivamente pela aresta 4, conforme mostra a Figura 23.

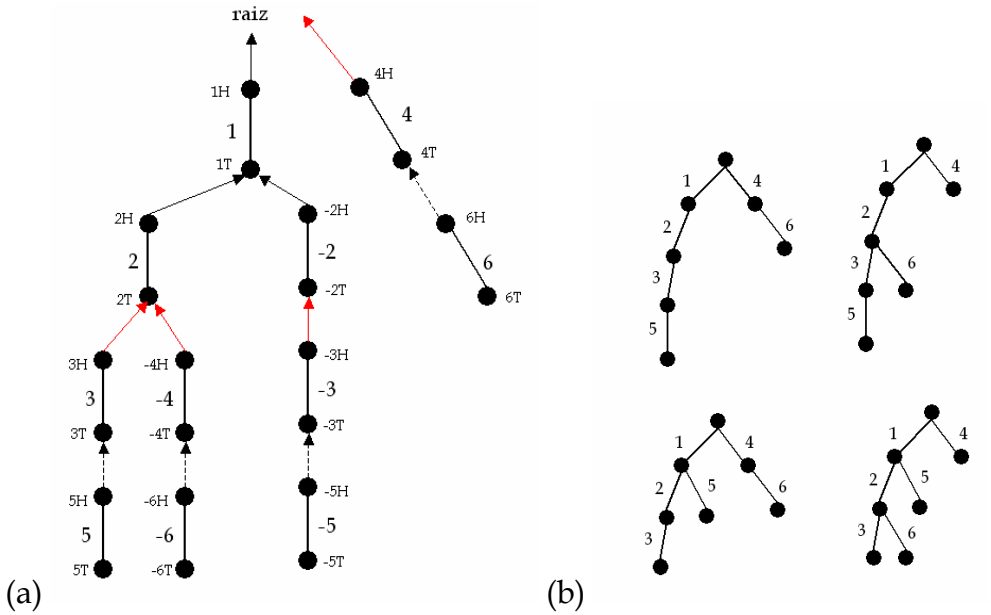


Figura 23:(a) Shadow tree final para a matriz S. Esta shadow tree contém três classes (b) e portanto representa quatro soluções distintas para o problema.

As seqüências de haplotypes para cada linha da matriz são dadas abaixo. Considerando T1 como sendo a árvore da Figura 23 (b) mais à esquerda, T2 a árvore abaixo de T1, T3 a direita de T1 e T4 a árvore abaixo de T3.

Considerando T1:

Linha 1 = 1 **2 2** 0 0 0

Hap1 = 1 **1 1** 0 0 0

Hap2 = 1 **0 0** 0 0 0

Linha 2 = 1 **2 2** 0 **2** 0

Hap1 = 1 **1 1** 0 **1** 0

Hap2 = 1 **0 0** 0 **0** 0

Linha 3 = **2 2** 0 **2** 0 **2**

Hap1 = **0 0** 0 **1** 0 **1**

Hap2 = **1 1** 0 **0** 0 **0**

Linha 4 = **2 2 2 2** 0 0

Hap1 = **0 0 0 1** 0 0

Hap2 = **1 1 1 0** 0 0

Considerando T2

Linha 1 = 1 **2 2** 0 0 0

Hap1 = 1 **1 1** 0 0 0

Hap2 = 1 **0 0** 0 0 0

Linha 2 = 1 **2 2** 0 **2** 0

Hap1 = 1 **0 0** 0 **1** 0

Hap2 = 1 **1 1** 0 **0** 0

Linha 3 = **2 2** 0 **2** 0 **2**

Hap1 = **0 0** 0 **1** 0 **1**

Hap2 = **1 1** 0 **0** 0 **0**

Linha 4 = **2 2 2 2** 0 0

Hap1 = **0 0 0 1** 0 0

Hap2 = **1 1 1 0** 0 0

Considerando T3:

Linha 1	=	1 **2 2** 0 0 0
Hap1	=	1 **1 1** 0 0 0
Hap2	=	1 **0 0** 0 0 0

Linha 2	=	1 **2 2** 0 **2** 0
Hap1	=	1 **1 1** 0 **1** 0
Hap2	=	1 **0 0** 0 **0** 0

Linha 3	=	**2 2** 0 **2** 0 **2**
Hap1	=	**0 0** 0 **0** 0 **1**
Hap2	=	**1 1** 0 **1** 0 **0**

Linha 4	=	**2 2 2 2** 0 0
Hap1	=	**0 0 0 1** 0 0
Hap2	=	**1 1 1 0** 0 0

Considerando T4:

Linha 1	=	1 **2 2** 0 0 0
Hap1	=	1 **1 1** 0 0 0
Hap2	=	1 **0 0** 0 0 0

Linha 2	=	1 **2 2** 0 **2** 0
Hap1	=	1 **0 0** 0 **1** 0
Hap2	=	1 **1 1** 0 **0** 0

Linha 3	=	**2 2** 0 **2** 0 **2**
Hap1	=	**0 0** 0 **0** 0 **1**
Hap2	=	**1 1** 0 **1** 0 **0**

Linha 4	=	**2 2 2 2** 0 0
Hap1	=	**0 0 0 1** 0 0
Hap2	=	**1 1 1 0** 0 0

3.6 – Tempo de Execução do Algoritmo

Sem dúvida a principal característica do algoritmo exposto anteriormente é a sua eficiência. Diferentemente dos outros algoritmos propostos para o problema [2, 3, 4, 5], este algoritmo é executado em tempo linear. Essa característica é facilmente verificada através de uma análise um pouco mais cuidadosa dos procedimentos *FirstPath, SecondPath, FixTree* e *NewEntries.* É interessante ressaltar que cada uma das operações (*flip, merge* e *edge addition*) é realizada em tempo constante. Como o processamento de cada linha é limitado por $O(m)$, onde m é o número de colunas da matriz, e cada procedimento é executado no máximo uma vez para esta linha, o tempo total de execução do algoritmo é limitado por $O(nm)$.

3.6.1 – Desempenho de *FirstPath* e *SecondPath*

Para cada uma das n linhas da matriz de entrada, os procedimentos *FirstPath* e *SecondPath* são ativados no máximo uma vez, para processar as colunas em *OldEntryList* e *CheckList*, respectivamente. Por definição, no pior caso estas listas irão conter um número de elementos igual a m, o número de colunas. No caso de *OldEntryList* isto ocorre quando a linha processada é formada por m entradas 2 e todas as colunas já tiveram uma entrada 2 anteriormente; além disso, o tamanho da lista *CheckList* é sempre menor ou igual ao tamanho da *OldEntryList* na mesma iteração.

O tempo de execução dos procedimentos *FirstPath* e *SecondPath* é limitado por uma constante vezes o número de colunas. O que pode acontecer é que o algoritmo execute um determinado *loop* várias vezes no processamento de uma coluna (quando a aresta correspondente à coluna que está sendo processada (C_i) tem como pai uma *shadow edge* e ambas pertencem a uma mesma classe). Neste caso, o algoritmo realiza uma atribuição (tempo constante) e entra no *loop* novamente sem ter terminado o processamento da coluna C_i. Intuitivamente, o pior caso acontece quando a aresta correspondente à coluna C_i tem acima dela *m*-1 *shadow edges*, o que provoca que o procedimento execute o *loop*, para a coluna C_i, *m*–1 vezes. Porém esta restrição implica que, para todas as outras colunas das listas o *loop,* será executado apenas uma vez, já que todas as *shadow edges* da árvore vão estar em uma sub-árvore diferente. Uma ilustração do pior caso é vista na Figura 24. Em qualquer caso, o número de vezes que este *loop* é executado não passa de 2 * (*m* - 1).

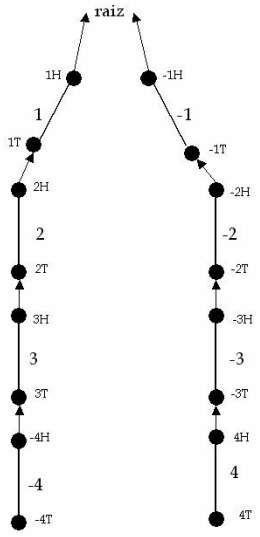


Figura 24: Considere que o procedimento FirstPath está sendo executado e que OldEntryList contém as colunas (1, 2, 3, 4). A shadow tree mostra que todas as arestas acima da aresta 4 são shadow edges, implicando na execução do loop por 3 vezes. Porém a partir daí o processamento de cada coluna da lista (OldEntryList ou CheckList) só vai executar este loop uma única vez.

3.6.2 – Desempenho de *FixTree* e *NewEntries*

Durante o processamento de uma linha i, o procedimento *FixTree* é executado no máximo uma vez. Chamemos E_1 a aresta com maior rótulo (mais à esquerda) em *OldEntryList* e E_2 a aresta com maior rótulo em *OldEntryList* que não esteja no caminho de E_1 à raiz de i (na Figura 12, E_1 e E_2 são as arestas 6 e 3, respectivamente). O único processamento feito por *FixTree* que não toma tempo constante é o de encontrar o maior caminho entre E_2 e as folhas de $T_{SP}(i - 1)$, tal que este caminho contenha a maior quantidade de *shadow edges* cujos números de colunas estejam em *OldEntryList*. Isso é feito com uma busca a partir de E_2, a qual cobre menos de m arestas. Do mesmo modo, o único processamento que não toma tempo constante no procedimento *NewEntries* é a busca por um caminho semelhante ao caminho encontrado no procedimento *FixTree*, porém este caminho este caminho é entre duas arestas diferentes. Assim como em *FixTree*, esta busca cobre menos de m arestas.

3.7 – Correção do Algoritmo

Os lemas apresentados no algoritmo são expostos abaixo acompanhados de exemplos que ilustrem seus significados.

Lema 1: Em todas as árvores contidas em $T(k + 1)$ existem dois caminhos em direção a raiz que passam por arestas correspondentes a todas as colunas com entrada 2 na linha $k + 1$ e que não possuem nenhuma aresta em comum.

As Figuras 25 e 26 mostram a execução do algoritmo para todas as linhas de uma matriz S.

$$S = \begin{pmatrix} 1\,0\,0\,0 \\ 1\,2\,0\,0 \\ 1\,2\,2\,2 \end{pmatrix}$$

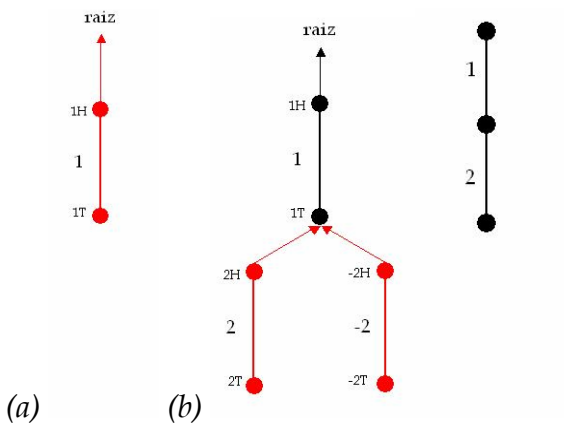


Figura 25: (a) A filogenia inicial contém apenas a aresta 1.
(b) Shadow tree resultante do processamento da segunda linha. Ao seu lado é exibida a única árvore contida nela.

Vê-se na figura anterior que a filogenia inicial para esta matriz S contém apenas a aresta 1. Como a primeira linha de *S* não contém nenhuma entrada 2, o Lema 1 é verdadeiro neste caso.

Para o lema ser verdadeiro devem haver 2 caminhos em direção a raiz (1T) que não tenham nenhuma aresta em comum e que passe por todas as arestas cujos números de coluna são 2. Na Figura 25 (b), o primeiro caminho é formado pelas arestas (1, 2) e o segundo caminho é vazio, o que comprova o Lema 1.

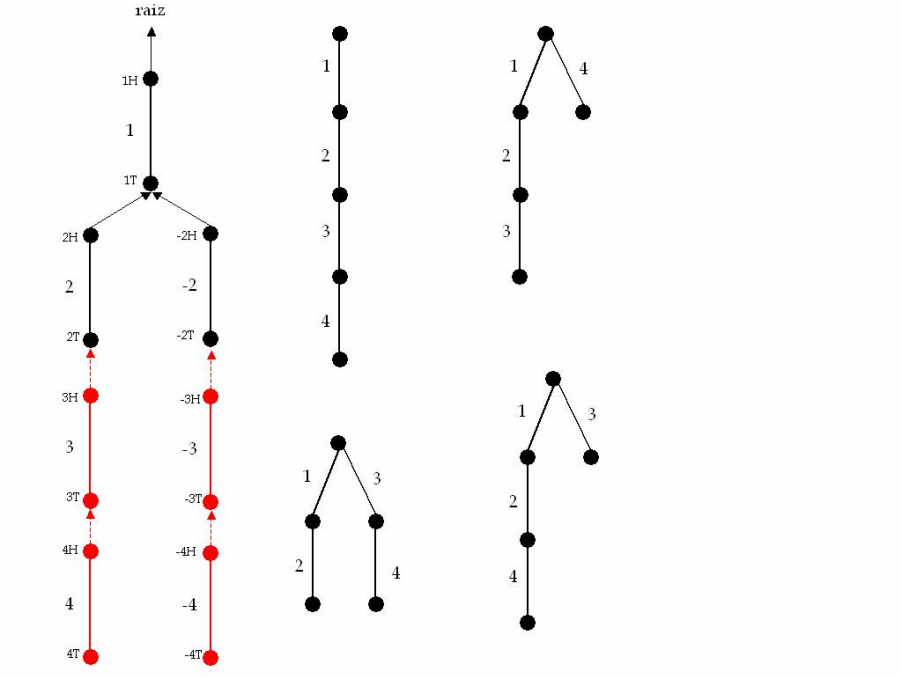


Figura 26: Shadow tree resultante do processamento da última linha de S e as árvores que estão contidas nela e que representam solução para o PPH.

Pode ser verificado que o Lema 1 é verdadeiro para todas as árvores contidas na *shadow tree* final. Na primeira os dois caminhos são: Caminho1 = (1, 2, 3, 4), Caminho2 = vazio. Na segunda: Caminho1 = (1, 2, 3), Caminho2 = (4). Na terceira: Caminho1 = (1, 2), Caminho2 = (3, 4) e na última árvore: Caminho1 = (1, 2, 4) e Caminho2=(3).

Lema 2: Se existe alguma solução para o PPH para a matriz *S* então entradas 1 devem estar obrigatoriamente à esquerda de entradas 2.

A justificativa passa pela forma como as linhas e colunas são ordenadas antes do processamento, a saber, primeiramente as colunas são ordenadas por valor decrescente de *leaf count*, e em seguida, as linhas são ordenadas colocando primeiramente aquelas que têm entrada 1 mais à direita. Recorde que todas as arestas rotuladas por colunas que tenham entrada 2 numa linha *i* devem formar dois caminhos distintos em direção a alguma aresta da filogenia inicial perfeita, e

que a partir desta última aresta há um caminho em direção à raiz da árvore contendo arestas cujas colunas apresentam entrada 1 nesta linha (aresta 2 da Figura 21, por exemplo). Caso haja uma entrada 2 à esquerda de uma entrada 1 nesta linha *i*, haverá uma violação da propriedade que afirma que nunca uma aresta com rótulo *c* poderá estar acima de uma aresta com rótulo menor do que *c* em um caminho em direção à raiz.

Lema 3: Assuma que todas as soluções para o PPH, restritas às colunas na *shadow tree T(k)*, estão contidas em *T(k)*. Suponha que o algoritmo realize uma operação de *flip/merge* no procedimento *FirstPath* para a linha $k + 1$. Qualquer árvore contida em *T(k)* que é perdida depois das operações de *flip/merge* não corresponde à nenhuma solução para o PPH.

O único caso em que ocorre uma operação de *flip* seguida de uma operação de *merge* no procedimento *FirstPath* durante o processamento de uma linha $k + 1$ é quando a coluna que está sendo processada (C_i) tem entrada 2 nesta linha e a coluna correspondente ao pai de *te*(C_i) tem entrada 0. Neste caso o pai de *te*(C_i) não pode estar no caminho de *te*(C_i) à raiz, já que nenhuma aresta com entrada 0 em uma linha poderá fazer parte de um caminho entre uma aresta cuja coluna apresente entrada 2 nesta linha e a raiz, conforme definição do Lema 1. Lembrando que T(k) é a árvore produzida pelo algoritmo depois do processamento das k primeiras linhas e que o algoritmo está prestes a executar uma operação de *flip* seguida de uma de *merge* sobre esta árvore pode-se notar claramente que existirá uma diferença entre T(k) e $T_{FP}(k)$. Basicamente algumas árvores que estão contidas em T(k) deixarão de existir em $T_{FP}(k)$. O Lema 3 afirma que as árvores perdidas depois destas operações não estão presentes em nenhuma solução para o PPH para a matriz *S*.

As Figuras 27 e 28 mostram um exemplo do significado deste lema.

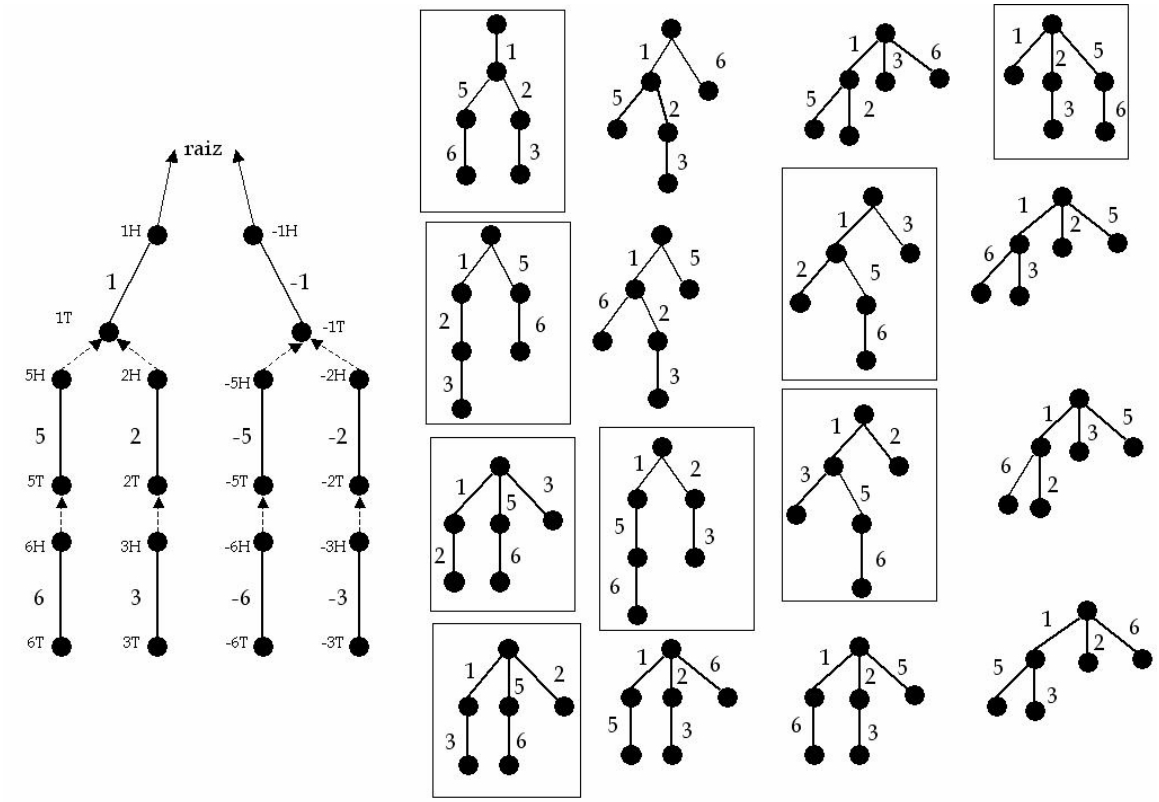


Figura 27: A shadow tree do lado esquerdo da figura é obtida após o processamento de uma linha k de determinada matriz. Ao seu lado estão representadas as 16 árvores contidas em T(k).

O algoritmo verifica que as arestas 5 e 6 não podem estar no mesmo caminho pois na linha $k + 1$ a coluna 5 apresenta entrada 0 e a coluna 6 entrada 2. Sendo assim é realizada uma operação de *flip* e logo depois uma operação de *merge*. Ainda nesta figura, as árvores destacadas são aquelas que serão perdidas depois das operações e que não estarão presentes em nenhuma solução do PPH para *S*.

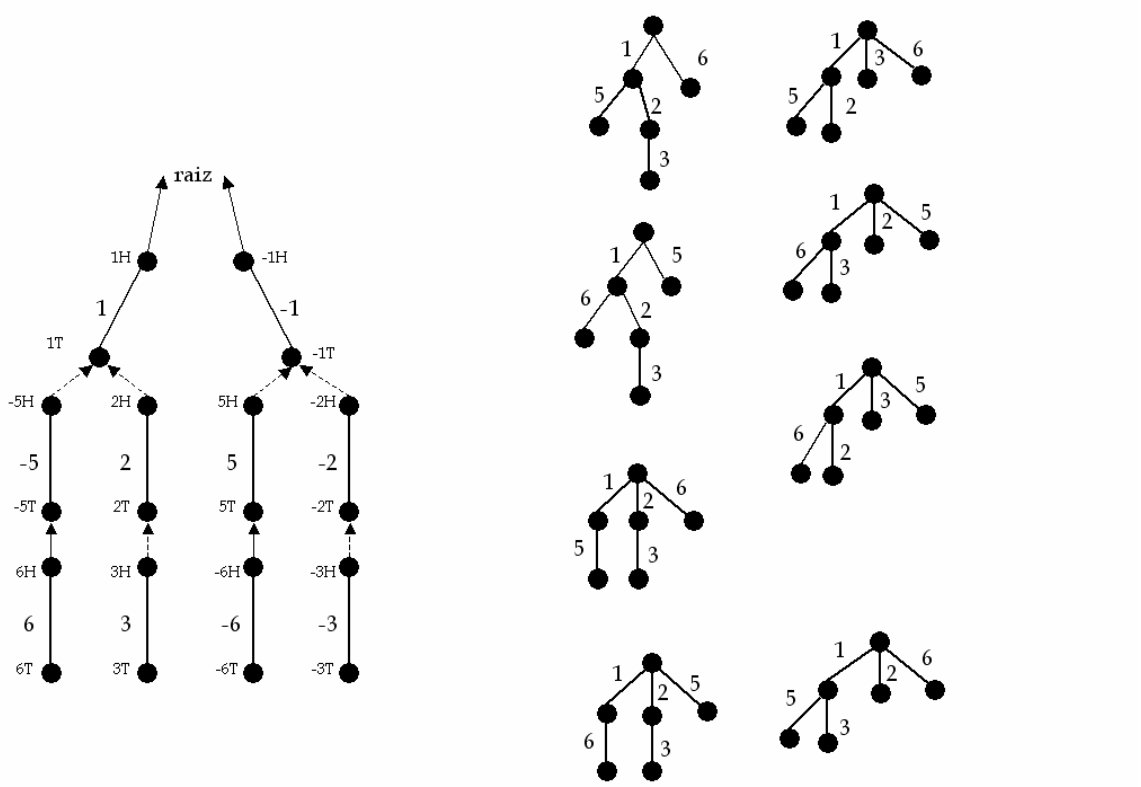


Figura 28: Depois de uma operação de flip da classe (6, -6) seguida de uma operação de merge desta classe com sua classe pai e posteriormente outro flip da nova classe

É facilmente observável que o número de árvores contidas nesta nova *shadow tree* é metade do número de árvores contidas na *shadow tree* da Figura 27. Interessante notar que apenas as árvores que tinham as arestas 5 e 6 no mesmo caminho em direção à raiz foram removidas.

O quarto lema do artigo estudado requer a definição de um outro conceito. Quando se fala em "*uma solução para o PPH restrita às colunas em uma shadow tree ST*" refere-se à árvore obtida de uma solução para o PPH depois de contrair-se todas as arestas correspondentes às colunas que não estejam em *ST*. Uma árvore *T* contida em *ST* está em uma solução para o PPH se *T* pode ser obtida através de uma solução para o PPH depois de se contrair todas as arestas correspondentes à colunas que não estejam em *ST*.

Lema 4: Assuma que todas as soluções para o PPH, restritas às colunas na *shadow tree* T(*k*) estejam contidas em T(*k*). Se o algoritmo colocar uma coluna C_j em *CheckList* durante o processamento de uma coluna C_i durante a execução de *FirstPath* para uma linha $k + 1$ então *te*(C_i) e *te*(C_j) jamais estarão em um mesmo caminho em qualquer solução para o PPH com a matriz *S*.

De acordo com a definição do lema jamais as arestas *te*(C_i) e *te*(C_j) jamais estarão no mesmo caminho, não importando quantas e quais operações sejam realizadas em T(*k*). Segundo a especificação de *FirstPath* o pai de *te*(C_i) pode ser E_p ou E'_p (estas arestas são definidas na explicação do procedimento *FirstPath*) já que só existem dois *join points* na classe do pai de *te*(C_i) (assim como em todas as classes). Desta forma *te*(C_j) nunca irá ser o pai de *te*(C_i). Ainda, como foi dito anteriormente, na definição das propriedades do algoritmo, em qualquer caminho direcionado à raiz os números das arestas são estritamente decrescentes bem como se uma aresta E foi colocada antes de uma aresta E', E' jamais estará acima de E em qualquer caminho em direção à raiz. Ocorre também que sempre *col*(E_p) > *col*(E'_p) e *col*(E_p) < C_j, portanto *te*(C_j) não poderá nunca estar acima de E_p ou E'_p. Da mesma forma $C_j < C_i$, pela própria definição de *OldEntryList*, o que indica que *te*(C_j) nunca estará abaixo de *te*(C_i) em nenhuma forma árvore resultante de operações em T(*k*). Conclui-se então que *te*(C_i) e *te*(C_j) jamais estarão em um mesmo caminho à raiz.

Em outras palavras, como *te*(C_j) não poderá estar acima do pai de *te*(C_i) nem abaixo de *te*(C_i) , estas arestas nunca estarão em um mesmo caminho. A Figura 29 mostra um exemplo ilustrativo deste lema. Considere o seguinte cenário: durante a execução do procedimento *FirstPath* para uma linha $k + 1$ de uma matriz *S*, $C_i = 6$ e $C_j = 3$.

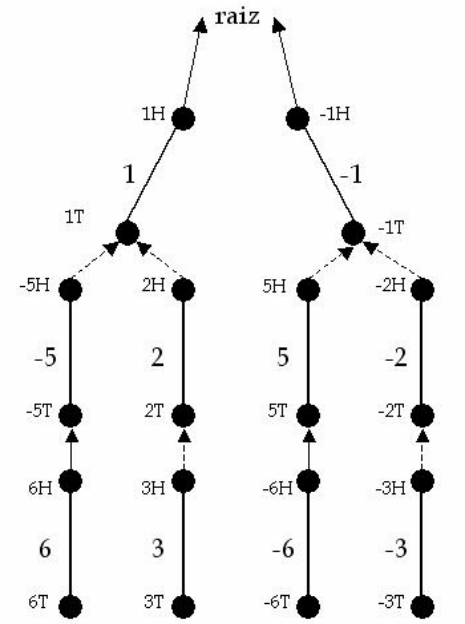


Figura 29: Ilustração do Lema 4

De acordo com o Lema 4 não existe nenhuma forma de se realizar operações sobre T(k + 1) de tal forma que as arestas 6 e 3 estejam em um mesmo caminho em direção à raiz. Isso fica óbvio na medida que a aresta 6 só pode ter como pai a aresta –5, e E_p nesse caso é igual a 1 já que 6 e –5 estão na mesma classe. Como col(E_p) < C_j então 3 nunca estará acima de 1. Além disso C_i > C_j (6 > 3) então a aresta 3 jamais estará abaixo da aresta 6 em um caminho em direção à raiz. Logo, as arestas 6 e 3 nunca pertencerão a um mesmo caminho. O mesmo acontece para as arestas 6 e 2.

Teorema 1: Todas as soluções para o PPH estão contidas na *shadow tree* final produzida pelo algoritmo. Da mesma forma todas as árvores contidas nesta *shadow tree* são soluções para o PPH.

Em resumo este teorema afirma o seguinte: todas as arestas correspondentes a elementos de *NewEntryList* para uma determinada linha k + 1 são anexadas às folhas de T(k), o que implica que todas as árvores contidas em T(k + 1) restritas às colunas em T(k) estão contidas em T(k + 1), pelo simples fato de que nenhuma

operação de *merge* foi executada. De acordo com o Lema 1, existem dois caminhos em direção à raiz de T(k + 1) que contêm todas as arestas em *OldEntryList* e que não possuem nenhuma aresta cuja coluna correspondente tenha entrada 0 na linha k + 1. Portanto a *shadow tree* final conterá p árvores e cada uma delas terá dois caminhos em direção à raiz (com nenhuma aresta em comum) para cada linha de S de tal forma que estes caminhos contenham todas as arestas cujas colunas correspondentes tenham entrada 2 nesta linha.

Agora resta verificar que toda solução para o PPH está contida na *shadow tree* final. Ding, Filkov e Gusfield [1] fazem isso através de uma indução. Primeiro é visto que todas as soluções para o PPH restritas às colunas em T(1) estão contidas em T(2). Logicamente que todas as soluções para o PPH restritas às colunas de T(1) estão contidas em T(1) já que esta árvore apresenta i classes, onde i é o número de entradas 2 na primeira linha da matriz. A indução indica que todas as soluções para o PPH restritas às colunas em T(k) estão contidas em T(k). O passo de indução é então provar que todas as soluções para o PPH restritas às colunas em T(k + 1) estão contidas em T(k + 1).

Dentre as operações que podem ser realizadas na *shadow tree* a operação de *flip* não altera o número de classes contidas em uma determinada *shadow tree*. A operação de adição de arestas à T(k) eleva o número de árvores contidas na *shadow tree*. Para cada *tree edge* adicionada a T(k) o número de árvores contidas na árvore resultante da adição dobra. Este aumento é maior ou igual ao número máximo de soluções para o PPH restritas às colunas de T(k + 1). Desta forma todas as soluções foram incluídas com estas adições, embora algumas delas sejam descartadas no processamento das próximas linhas.

A operação de *merge*, de acordo com o Lema 3, retira de T(k + 1) apenas árvores que não sejam solução para o PPH. Portanto nenhuma solução para o PPH restritas às colunas de T(k + 1) é perdida durante o processamento desta linha. Um exemplo que simboliza este teorema é mostrado na Figura 30.

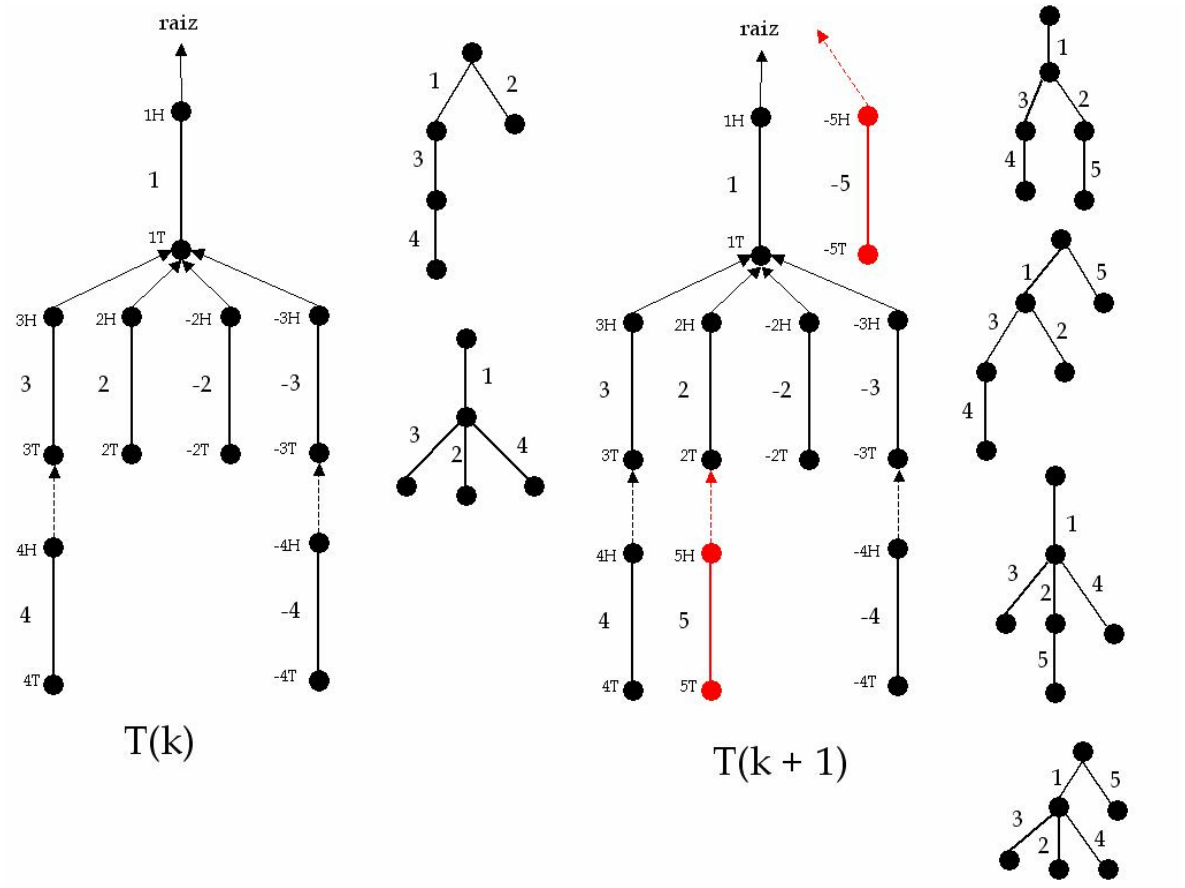


Figura 30: Exemplo de verificação do Teorema 1.

Conforme legenda na própria figura anterior, no lado esquerdo dela está representada uma *shadow tree* obtida depois do processamento de *k* linhas da matriz. Ao seu lado estão todas as árvores contidas nesta *shadow tree*. Também na figura é mostrada a *shadow tree* obtida depois do processamento da linha *k* + 1 e as árvores contidas nela. Como não houve nenhuma operação de *merge* todas as árvores contidas em T(*k* +1) restritas às colunas em T(*k*) estão também contidas em T(*k*), o que está de acordo com a segunda parte do teorema. Neste exemplo, com a adição de uma nova classe (5, -5), o número de soluções para o PPH restritas às colunas em T(k + 1) dobra em relação a T(k). Conclui-se então que nenhuma solução para o PPH deixa de ser adicionada com a adição de uma nova classe.

4. Conclusões

Este trabalho procurou demonstrar através principalmente de exemplos as idéias centrais e a grande eficiência do algoritmo exposto em Dig, Filkov e Gusfield [1]. Tentou-se explorar exemplos que abrangessem todos os casos de que fossem mais significativos.

Paralelamente ao estudo foi desenvolvida uma implementação em JAVA para o algoritmo descrito, batizada de **BioLabPPH**, que se mostrou bastante eficiente. O desempenho do programa desenvolvido não foi tão bom quanto o do programa **LPPH**, elaborado pelos autores do artigo, já que este último foi escrito na linguagem de programação C, que apresenta um melhor desempenho do que JAVA. Mas em comparação aos programas desenvolvidos em outras abordagens [2, 3, 4, 5], o **BioLabPPH** apresenta ótima performance, logicamente devido ao caráter linear de seu algoritmo.

Uma vantagem do **BioLabPPH** sobre o **LPPH** é que, pelo próprio fato dele ter sido escrito em JAVA, é muito mais simples disponibiliza-lo na *Web* para ser consultado pela comunidade científica, enquanto o **LPPH** é um aplicativo que deve necessariamente ser baixado para o computador do usuário.

Os casos de testes foram criados utilizando-se o programa **ms** [7] combinado com um outro programa desenvolvido durante o trabalho, que simplesmente combina os dados de haplotypes gerados pelo **ms** gerando vetores de genotypes. Este programa foi desenvolvido em JAVA, e é bastante simples.

Como trabalho futuro sugere-se que seja realizada uma melhora na performance do **BioLabPPH** através da alteração da estrutura de classes, utilizando um menor número dessas. Além disto, seria de muito valor uma interface gráfica mais atraente para o **BioLabPPH**, e que houvesse uma opção de interromper o algoritmo para que ele mostrasse o processamento de cada uma das linhas da matriz de entrada ao invés de apenas exibir a *shadow tree* final e as seqüências que explicam *S*.

5. Referências

[1] D. Gusfield, Z. Ding and V. Filkov. A Linear-Time Algorithm for the Perfect Phylogeny Haplotyping (PPH) Problem. Proceedings of the 9th International Conference on Computational Biology - RECOMB'05, Cambridge, MA, May 2005.

[2] D. Gusfield. Haplotyping as Perfect Phylogeny: Conceptual Framework and Efficient Solutions. Proceedings of the 6th International Conference on Computational Biology RECOMB'02, 166-175, 2002.

[3] D. Gusfield. Inference of Haplotypes from Samples of Diploid Populations: Complexity and Algorithms. Journal of Computational Biology, Vol. 8 no. 3 305-323, 2001.

[4] D. Gusfield, V. Bafna, G. Lancia and S. Yooseph. Haplotyping as Perfect Phylogeny: a Direct Approach. Journal of Computational Biology, Vol. 10 323-340, 2003.

[5] R. H. Chung and D. Gusfield. Empirical Explorations of Perfect Phylogeny Haplotyping and Haplotypers. Proceedings of the 9th International Conference on Computing and Combinatorics, Vol. 2697 5-9, 2003.

[6] THE INTERNATIONAL HAP MAP CONSORTIUM. The International HapMap Consortium. Nature, Vol. 426, 18/25, 789-796, 2003.

[7] R. R. Hudson. Generating Samples under a Wright-Fisher Neutral Model. Bioinformatics, Vol. 18 337-338, 2002.

[8] D. Gusfield, V. Bafna, S. Hannenhalli and S. Yooseph. A Note on Efficient Computing of Haplotypes via Perfect Phylogeny. Journal of Computational Biology, 2004. *In press*

[9] P. Bonizzoni, G. D. Vedova, R. Dondi and J. Li. The Haplotyping Problem: Models and Solutions. Journal of Computer Science and Technology, Vol. 18 675-688, 2003.

[10] R. H. Chung and D. Gusfield. Perfect Phylogeny Haplotyper: Haplotype Inferal Using a Tree Model. Bioinformatics, Vol. 19 780-781, 2003.

[11] Z. Ding. Disponível em: <zhding@ucdavis.edu>. Consultado nos dias 06 e 22 de junho e 01 de julho.

6. Assinaturas

_____

Katia Silva Guimarães
(Orientadora)

_____

Enio Felipe da Rocha
(Aluno)