Anotação de seqüências e expansão do sistema EGene/CoEd

Ricardo Yamamoto Abe   Nº USP: 3670866
Orientador: Alan Mitchell Durham – IME/USP
Co-orientador: Arthur Gruber – ICB/USP

Sumário

I Iniciação científica **3**

1 Introdução **4**
1.1 O sistema EGene/CoEd 4
1.2 Objetivos 5
1.3 Tecnologias utilizadas 5

2 Modelagem do banco de dados de evidências **7**
2.1 Similaridade 7
2.2 Multi-Intervalo 7
2.3 Estatística 8
2.4 Gráfico 8

3 Expansão do sistema EGene/CoEd **9**
3.1 Exemplo de uso do EGene/Coed 10
3.2 Atualização do EGene: *forks* e seletores 12
3.2.1 *Forks* 12
3.2.2 Seletores 14
3.2.3 Controle de paralelismo 15

4 Geração de evidências **17**

5 Conclusão **19**

II Parte subjetiva **20**

6 Parte subjetiva **21**
6.1 Dificuldades e frustrações encontrados 21
6.2 Interação com orientadores e o grupo de bioinformática 22
6.3 Disciplinas do BCC mais relevantes 22
6.4 Futuro 24

Referências bibliográficas **25**

Parte I

Iniciação científica

Capítulo 1

Introdução

O seqüenciamento de DNA em larga escala tornou-se uma metodologia muito utilizada para desvendar a complexidade bioquímica dos organismos vivos. Além disso, a quantidade de genomas procarióticos seqüenciados vem crescendo de forma exponencial.

Inicialmente, as seqüências devem ser pré-processadas antes da montagem, agrupamento e anotação dos genes. As etapas comuns necessárias nesse processo incluem a avaliação da qualidade, mascaramento de vetor, aparamento de extremidades, filtragem por tamanho e identificação e filtragem de seqüências contaminantes. Assim, o processamento de seqüências em biologia computacional envolve várias tarefas computacionais interconectadas, cada um com um protocolo de entrada e saída de dados distinto.

Usualmente, é criado um script, chamado *pipeline* que executa as tarefas em ordem determinada pelo usuário (cada tarefa é denominada como componente). Entretanto, a criação de cada script torna-se onerosa no sentido de que é necessário considerar que cada par de programas que troca dados necessita de um processamento adicional para que a saída de um seja compatível com a entrada do outro.

Com o processamento das seqüências é possível obter algumas informações, chamadas evidências, a partir das quais um biólogo pode gerar anotações, ou seja, vincular partes de uma seqüência a funções reguladoras, componentes celulares, entre outros.

1.1 O sistema EGene/CoEd

Dada a complexidade de implementação de *pipelines*, várias soluções foram criadas pela comunidade científica. O PipeOnline [2] e o ESTAnnotator [10] são exemplos de ferramentas de pré-anotação, enquanto o Genescript [11]

e o GeneQuiz [1] são ferramentas de anotação. Neles encontramos *scripts* para UNIX realizando a integração dos diversos módulos. Entretanto, esses sistemas não provêem as funcionalidades necessárias para todas as etapas de seqüenciamento e anotação. Além disso, a arquitetura encontrada nos mesmos não foi projetada de modo a permitir uma customização fácil, de modo que alguns projetos podem não se beneficiar de algumas dessas ferramentas.

Nesse cenário, foi criado o EGene [7], um sistema de geração de pipelines que torna mais fácil a implementação dos mesmos. O EGene original contém ainda uma série de componentes que permitem a construção de *pipelines* completos de pré-anotação.

Juntamente com o EGene, foi construído o CoEd, uma ferramenta gráfica que facilita a criação dos arquivos de configuração utilizados pelo EGene. Mais do que isso, o CoEd é um *framework* de editores de configuração, permitindo que outros sistemas sejam beneficiados por uma interface gráfica de configuração.

No EGene, cada componente é um *script* Perl que lê da entrada padrão (STDIN) e escreve na saída padrão (STDOUT), sendo que os dados utilizados podem ser uma string XML com todos os dados de uma dada seqüência, ou um identificador de seqüência dentro de um banco de dados. Como a entrada e saída são bem definidas, a integração dos componentes torna-se fácil. Além disso, novos programas podem ser integrados ao sistema, bastando que seja criado um componente EGene que siga o mesmo protocolo.

1.2 Objetivos

O trabalho realizado durante a iniciação científica envolveu duas partes: a expansão do sistema EGene/Coed e modelagem do banco de dados de evidências para auxílio no processo de anotação.

Paralelamente a tudo isso, eu ajudei na criação dos componentes EGene de geração de evidências.

1.3 Tecnologias utilizadas

A parte de programação da iniciação científica envolveu basicamente a utilização de Perl e Java. Na parte Java, utilizei a IDE Eclipse para auxiliar no desenvolvimento e depuração.

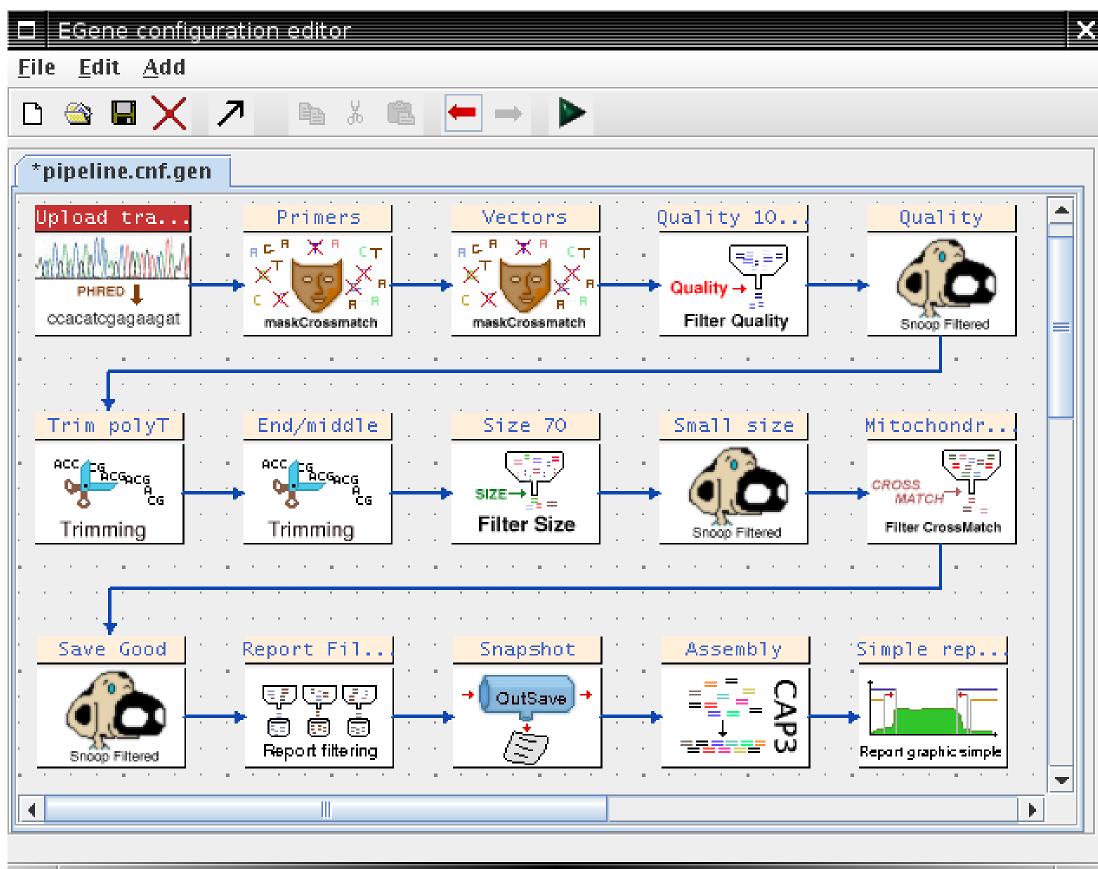

Figura 1.1: *Um pipeline gerado pelo CoEd, com um componente para carregar os dados de um arquivo no formato FASTA, uma filtragem por qualidade das bases e geração de relatório.*

Capítulo 2

Modelagem do banco de dados de evidências

Os componentes do EGene original permitem basicamente realizar o pré-processamento das seqüências, eliminando contaminantes e subseqüências de baixa qualidade. Em nosso grupo de bioinformática, a aluna Milene Ferro desenvolveu componentes para geração de evidências e componentes de anotação automática [9] no formato *feature table* de submissão de seqüências anotadas.

Tomando por base uma modelagem das bases de dados do sistema original, foi feita uma ampliação da mesma de modo a permitir a inclusão dos dados de evidências. O arquivo modeloER.png, incluído no pacote, contém toda a estrutura desenvolvida. Foram identificados 4 tipos de evidências: similaridade, multi-intervalo, estatística e gráfico.

2.1 Similaridade

Uma evidência desse tipo indica se uma seqüência possui uma subseqüência com alguma similaridade com outra encontrada em alguma base de dados. Guardamos os dados de início e fim da subseqüência alvo e a da base de dados, os *scores* e seus rótulos, uma descrição e posições de *gap*. Um *gap* é uma região dentro da seqüência alvo ou a da base de dados que não faz parte do alinhamento.

2.2 Multi-Intervalo

Essa evidência dá um rótulo e um valor para determinados trechos da subseqüência.

2.3 Estatística

Uma evidência estatística basicamente é uma tabela de rótulos e valores. Ao contrário das evidências de similaridade e multi-intervalo, a estatística envolve a seqüência como um todo.

2.4 Gráfico

Armazenamos os pontos que formam gráficos obtidos a partir do processamento da seqüência. No momento, não temos componentes que geram evidências gráficas.

Capítulo 3

Expansão do sistema EGene/CoEd

A versão original do EGene permite uma única arquitetura de interligação de componentes, denominada *pipeline*. Num *pipeline*, os dados seguem um único fluxo, sendo processados seqüencialmente por cada componente.

Figura 3.1: *Arquitetura possível dentro do EGene original.*

Cada componente do EGene é um programa feito em Perl que possui um arquivo de configuração específico e que utiliza o SequenceObject para troca de dados. O SequenceObject mantém um padrão sobre formatação dos dados das seqüencias processadas. Ele utiliza as facilidades dos *pipes* do UNIX, de modo que num *pipeline* com vários programas ocorre um ganho com paralelismo de forma transparente, sem a necessidade de haver um controle explícito de concorrência, já que o próprio sistema operacional é encarregado disso.

O programa responsável pelo controle de execução de um pipeline é o **bigou.pl**. O bigou recebe como parâmetro o arquivo de configuração gerado pelo CoEd, arquivo esse que contém os programas que fazem parte do *pipeline* e seus respectivos parâmetros. Ao realizar o *parser*, o bigou cria um arquivo de configuração para cada um dos componentes do *pipeline*. Ao fim do processo, ele gera uma linha de comando com a chamada de todos os componentes e a executa.

3.1 Exemplo de uso do EGene/Coed

Utilizando como exemplo o *pipeline* da figura 1.1, obtemos o seguinte arquivo de configuração:

```
#**************************************
PIPELINE=pipeline
#**************************************

#=======================================
PHASE=upload_fasta
program = upload_fasta.pl
multifastafile = entrada.fasta
#=======================================

#=======================================
#                 ||                  #
#                _||_                 #
#                \  /                 #
#                 \/                  #
#=======================================

#=======================================
PHASE=filter_quality
program = filter_quality.pl
minimum_quality_other_bases = 10
percentage_good_bases_in_window = 95
minimum_quality_in_window = 15
percentage_readings_in_sequence = 0
invalid_letters = XxNn
window_size = 200
#=======================================

#=======================================
#                 ||                  #
#                _||_                 #
#                \  /                 #
#                 \/                  #
#=======================================

#=======================================
PHASE=report_graphic_complete
program = report_graphic_complete.pl
report_dir = dir
report_file = report.html
alias_database_file = /dev/null
primer_database = primer
#=======================================
```

Serão gerados 3 arquivos de configuração com nomes criados aleatoriamente. Por exemplo, poderíamos ter os arquivos `a.cnf`, `b.cnf` e `c.cnf`, com o seguinte conteúdo:

1. `a.cnf`

```
#========================================
PHASE=upload_fasta
program = upload_fasta.pl
multifastafile = entrada.fasta
#========================================
```

2. `b.cnf`

```
#========================================
PHASE=filter_quality
program = filter_quality.pl
minimum_quality_other_bases = 10
percentage_good_bases_in_window = 95
minimum_quality_in_window = 15
percentage_readings_in_sequence = 0
invalid_letters = XxNn
window_size = 200
#========================================
```

3. `c.cnf`

```
#========================================
PHASE=report_graphic_complete
program = report_graphic_complete.pl
report_dir = dir
report_file = report.html
alias_database_file = /dev/null
primer_database = primer
#========================================
```

Em seguida, o bigou irá chamar uma linha de comando similar a que encontra-se abaixo:

```
upload_fasta.pl -conf a.cnf | filter_quality.pl -conf b.cnf |
report_graphic_complete.pl -conf c.cnf;
```

A partir daí, o UNIX realiza o controle de paralelismo e concorrência via *pipes*. Com isso, o *pipeline* é executado.

3.2 Atualização do EGene: *forks* e seletores

Durante a iniciação científica, foram criadas duas novas estruturas para o processamento: *forks* e seletores. Além disso, foi necessário atualizar a ferramenta CoEd para que ela fosse compatível com as novas funcionalidades.

3.2.1 *Forks*

Num *fork*, um dado pode ser enviado para vários componentes diferentes, gerando novas possibilidades de fluxo.

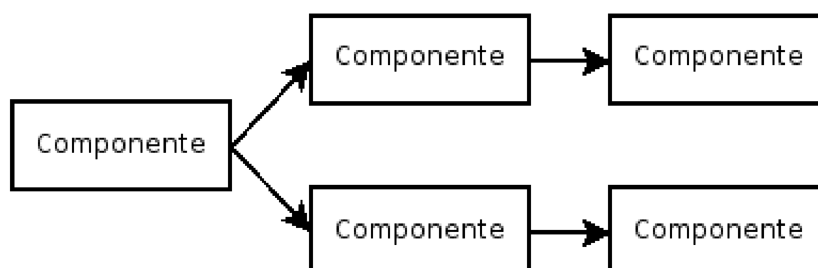

Figura 3.2: *Estrutura típica de um fork.*

Para criação do *fork*, utilizamos *named pipes*. Um *named pipe* é um arquivo UNIX que funciona como um *pipe* normal. Dessa forma, podemos representar a seguinte linha de comando:

```
programa1 | programa2
```

pela criação de um *named pipe* chamado `np` e a seguinte linha de comando:

```
programa1 > np & programa2 < np;
```

Tendo em mente a utilização dos *named pipes*, o bigou foi alterado e foram criados os programas **named_pipe_fork.pl**, **upload_named_pipe.pl** e **file_fork.pl**.

upload_named_pipe.pl

É um programa que lê um *named pipe* e escreve na saída padrão.

named_pipe_fork.pl

Esse programa recebe como parâmetro todos os *named pipes* que compõe o *fork*. Ele lê uma linha da entrada padrão e a escreve em cada um dos *named pipes*.

file_fork.pl

Funciona da mesma forma que o named_pipe_fork, mas ao invés de ler da entrada padrão, ele lê um arquivo. Isso ocorre quando a opção `manyfiles` é usada. Com ela, o bigou executa um componente de cada vez, sem paralelismo. Essa opção é utilizada para fins de depuração.

bigou.pl

A alteração do bigou envolveu uma mudança no *parser* do arquivo de configuração. Ao encontrar a string `FORK` no arquivo, o bigou começa a procurar pelos parâmetros `branch`. Um exemplo disso está abaixo:

```
#==============================
FORK
branch=report_graphic_complete
branch=filter_quality
#==============================
```

Cada parâmetro *branch* contém o nome de fase de um dos componentes do *pipeline*. Supondo que os nomes de fases são únicos, o bigou armazena esses nomes de fase em um *hash*. Ao encontrar esse nome de fase num momento posterior, dentro do arquivo de configuração, o bigou altera a linha de comando que está sendo gerada para chamar o upload_named_pipe antes do componente definido pelo nome da fase.

Utilizando com exemplo o *pipeline* mostrado nas figuras 3.3 e 3.4, teríamos a seguinte linha de comando:

```
a -conf a.cnf | named_pipe_fork.pl > /dev/null &
upload_named_pipe.pl | b -conf b.cnf > /dev/null &
upload_named_pipe.pl | c -conf c.cnf > /dev/null;
```

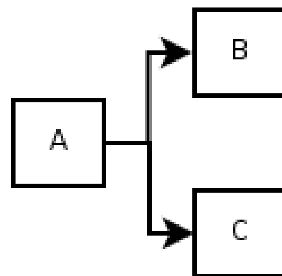

Figura 3.3: *Estrutura do fork observado pelo usuário final.*

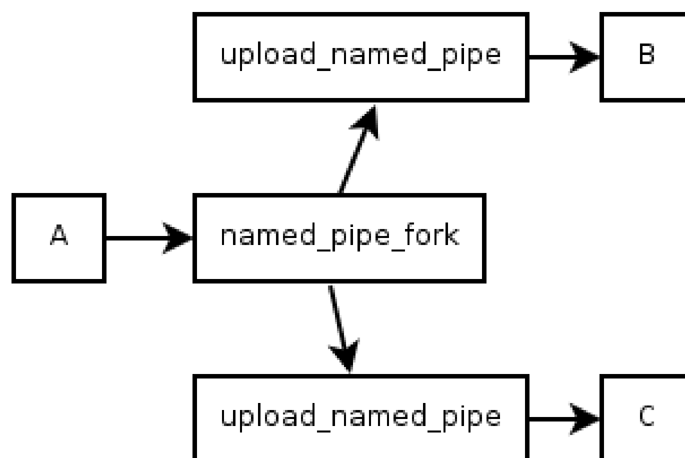

Figura 3.4: *Estrutura do fork gerado pelo bigou ao fim do processamento do arquivo de configuração.*

3.2.2 Seletores

Com seletores, é possível encapsular um componente e fazer com que uma seqüência só seja processada pelo mesmo se uma dada condição for satisfeita.

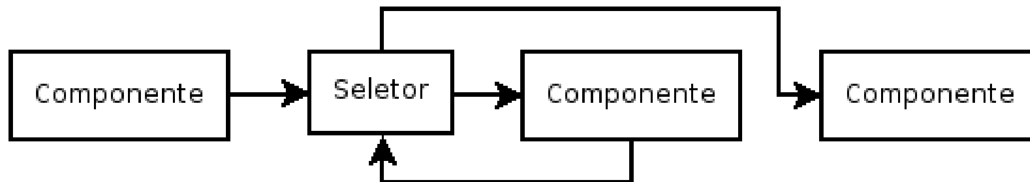

Figura 3.5: *Estrutura de um seletor.*

Para a criação dos seletores, também foram utilizados *named pipes*. Foi alterado o bigou, enquanto foram criados os programas **upload_selector.pl**, **selector_similarity.pl** e **end_selector.pl**. Com isso, podemos fazer comparações com os dados de evidência do tipo similaridade. Por exemplo, podemos verificar se uma dada seqüência pôde ser alinhada com outra dentro de uma base de humanos e, dessa forma não processar uma busca mais complexa em alguma outra base de humanos.

selector_similarity.pl

Esse programa realiza as comparações dos dados de evidência, decidindo se o componente encapsulado deve processar a seqüência ou não.

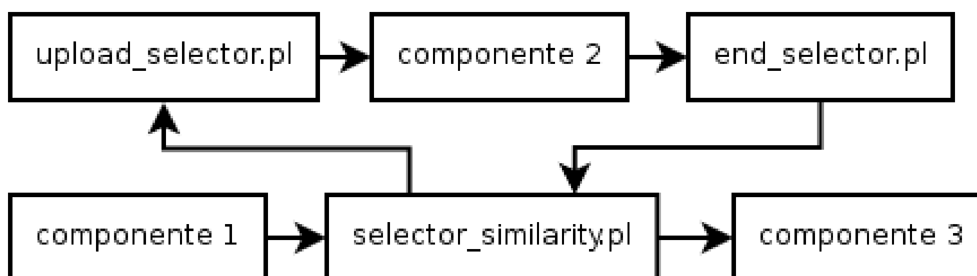

Figura 3.6: *O seletor implementado dentro do EGene.*

upload_selector.pl

Lê de um *named pipe* e envia os dados para o componente encapsulado.

end_selector.pl

Lê da entrada padrão e escreve no *named pipe* lido pelo selector_similarity.

bigou.pl

O bigou foi alterado para poder fazer o *parser* da string `SELECTOR`. A partir daí, ele começa a ler os parâmetros de condição do seletor, que podem ser: subject_id, score ou evalue. Depois, devem ser processados os parâmetros do componente encapsulado e, por fim, a string `END_SELECTOR` deve ser encontrada.

Para o *pipeline* com seletor definido na figura 3.6, teríamos a seguinte linha de comando:

```
componente1 -conf arq1.cnf | selector_similarity.pl -conf arq2.cnf |
componente3 -conf arq3.cnf > /dev/null &
upload_selector.pl -conf arq4.cnf | componente2 -conf arq5.cnf |
end_selector.pl -conf arq6.cnf > /dev/null;
```

3.2.3 Controle de paralelismo

Para fins de otimizar o acesso aos arquivos, é usual que os comandos de escrita não sejam executadas imediatamente, mas enviados para um *buffer*. Esse processo ocorre até que o *buffer* seja totalmente preenchido; nesse momento, a gravação no disco é efetivamente feita.

Essa característica trouxe problemas para o uso de seletores. Observando o exemplo da linha de comando referente à figura 3.6, os programas ficavam em espera, sem realizar qualquer processamento. Ao realizar a depu-

ração, verifiquei que o programa selector_similarity, ao enviar dados para o upload_selector, não preenchia totalmente o *buffer*. Assim, os programas componente2 e end_selector também não recebiam seus dados. O problema encontrado é que o selector_similarity fica esperando os dados do end_selector, para preservar a ordem de processamento das seqüências.

Para contornar isso, foi utilizada a opção *autoflush* para todos os *named pipes* utilizados. Com isso, a escrita é executada sem a necessidade de preenchimento dos *buffers*. Além disso, selector_similarity envia caracteres de controle para upload_selector e end_selector envia esses caracteres para selector_similarity. Esse controle foi criado pois não havia como garantir, utilizando o *autoflush*, um componente fosse finalizado sem processar as seqüências. Isso pode ocorrer se um dos programas tentar abrir um *named pipe* para leitura sem que exista outro processo pronto para escrita.

Capítulo 4

Geração de evidências

Durante a iniciação científica, auxiliei a aluna Milene na construção de vários componentes para geração de evidência. Minhas tarefas principais eram a instalação dos programas no servidor do ICB e a criação das expressões regulares utilizadas no *parser* das saídas desses programas.

A lista de componentes criados é a seguinte:

- annotation_blast.pl
- annotation_glimmerhmm.pl
- annotation_orf.pl
- annotation_sim4.pl
- annotation_trf.pl
- annotation_estscan.pl
- annotation_glimmerm.pl
- annotation_phat.pl
- annotation_snap.pl
- annotation_trna.pl
- annotation_exonerate.pl
- annotation_hmmer.pl
- annotation_rpsblast.pl

- annotation_string.pl
- annotation_twinscan.pl
- annotation_genscan.pl
- annotation_mreps.pl
- annotation_signalP.pl
- annotation_tmhmm.pl

Capítulo 5

Conclusão

Com forks e seletores, o EGene é capaz de implementar o processamento de programas em praticamente toda arquitetura possível em grafos direcionados acíclicos. Uma única estrutura não é implementável com as alterações: dois ou mais componentes enviando dados para um outro. Entretanto, essa limitação é aceitável no contexto de processamento de seqüências, dado que ela não é utilizada usualmente.

Além do aumento de arquiteturas ocorre também um ganho de desempenho. Com os seletores, é possível ignorar a execução de um ou mais componentes sobre uma dada seqüência. Com os *forks* temos ainda mais paralelismo do que o oferecido pelos *pipes* encontrados nos sistemas UNIX.

Outro ganho importante refere-se ao CoEd, pois com essas novas estruturas, ele tornou-se mais flexível como *framework* de editor de configuração, de modo que agora ele pode ser usado em um número maior de cenários.

Em relação ao componentes de evidência, a grande maioria já está pronta. De fato, falta apenas o *parser* do programa Interpro. Com isso, tais componentes estão começando a ser usados efetivamente no mestrado da Milene.

Parte II

Parte subjetiva

Capítulo 6

Parte subjetiva

6.1 Dificuldades e frustrações encontrados

Ao começar a iniciação científica em bioinformática, eu encontrei certa dificuldade para associar os diversos programas, algoritmos e teorias com a parte biológica envolvida. Além disso, inicialmente foi relativamente difícil interagir com o grupo do ICB, dado que eles, mais envolvidos em biologia do que computação, possuem uma forma de pensar e também um vocabulário muito diferentes dos encontrados dentro do IME.

Dessa forma, antes de qualquer coisa, estudei partes do livros *Cell* [3], para compreender a teoria básica sobre funcionamento de uma célula, DNA e RNA, e do *Bioinformatics* [13], que explica os algoritmos e programas mais utilizados. Com isso já foi possível entender alguns dos problemas principais e quais as abordagens mais comuns para resolvê-los. Isso também permitiu que eu pudesse conversar de maneira mais fácil com os biólogos.

Na parte computacional, o maior obstáculo foi a utilização da linguagem Perl. Ela é poderosa e muito útil na área de biológica computacional, mas é extremamente fácil criar um programa Perl complicado de ser entendido por outras pessoas ou mesmo pelo próprio criador do código. Pelo menos para mim, a existência de referências de vetor ou *hash*, o fato de matrizes passadas como argumento serem transformadas em vetores e a possibilidade de chamar uma função com um número arbitrário de parâmetro, tornaram difíceis o entendimento, alteração e manutenção dos programas já existentes.

Além disso, criar o controle de paralelismo para os seletores tomou bastante tempo. Com isso, não foi possível fazer a modelagem XML para anotações.

6.2 Interação com orientadores e o grupo de bioinformática

Durante a iniciação científica, fui orientado por duas pessoas: na área mais relacionada com computação, o prof. Alan M. Durham, enquanto que na parte biológica, o prof. Arthur Gruber. Ambos mostraram muita disposição para me explicar os vários conceitos envolvidos e responder todas as minhas dúvidas. De modo geral, acredito que os dois são ótimos orientadores e que realmente se preocupam com a formação de seus alunos.

Participei de diversos seminários do grupo de bioinformática, de modo que pude discutir vários tópicos da área. Além disso, apresentei alguns desses seminários, o que contribuiu para minha formação, já que tive que aprender a criar boas transparências e a falar em público.

A expansão do EGene e a criação de boa parte das ferramentas de anotação foi feita em conjunto com Milene Ferro, atualmente aluna de mestrado do ICB. Nessa interação eu pude compreender outros assuntos de biologia e ferramentas de bioinformática, além de tentar repassar meu conhecimento de computação. Acredito que foi um trabalho bastante proveitoso para ambas as partes.

6.3 Disciplinas do BCC mais relevantes

No bacharelado em Ciências da Computação, muitas disciplinas foram importantes tanto em relação à iniciação científica quanto à minha formação – e, com poucas exceções, só identifiquei o quão útil uma disciplina era **após** terminar de cursá-la (imagino que esse seja um problema encontrado por boa parte dos alunos). Listo abaixo aquelas que considero como mais relevantes:

- **MAC0122 – Princípios de Desenvolvimento de Algoritmos:** nessa disciplina obtemos conceitos básicos que são usados em todo o restante do curso.

- **MAT0139 – Álgebra Linear para Computação:** um bom entendimento dessa matéria ajudou a eliminar o trauma de matemática que eu tinha (talvez não totalmente), além de servir de base para compreensão de diversos temas, como o método simplex e redes neurais.

- **MAC0315 – Programação Linear:** a primeira disciplina vista na graduação que trata mais profundamente de teorias matemáticas dentro da computação.

- **MAE0228 – Noções de Probabilidade e Processos Estocásticos:** sem ela, seria impossível compreender diversos algoritmos e teorias utilizados em bioinformática, como por exemplo HMM (*Hidden Markov Model*).

- **MAC0239 – Métodos Formais em Programação:** essa disciplina trouxe uma base importante para o entendimento de provas de teoremas. Por isso, imagino que deveria haver, no primeiro semestre do BCC, alguma matéria que oferecesse uma introdução dos conceitos de lógica vistos em MAC0239. Acredito que isso traria diversos benefícios em matérias que exigem que o aluno construa provas formais, como por exemplo Programação Linear e Álgebra 2.

- **MAC0426 – Sistemas de Bancos de Dados:** uma disciplina fundamental em qualquer curso sério de computação, essencial para que eu pudesse atualizar o modelo para incorporar evidências para fins de anotação de genes.

- **MAC0414 – Linguagens Formais e Autômatos:** é inviável realizar uma iniciação científica em biológica computacional sem essa disciplina, dado que diversos algoritmos dessa área têm como base a teoria encontrada na mesma.

- **MAC0441 – Programação Orientada a Objetos & MAC0413 – Tópicos de Programação Orientada a Objetos:** a alteração do CoEd, que foi feito em linguagem Java, exigiu que eu tivesse conhecimento de padrões de programação, como aqueles encontrados no livro *Design Patterns* [8]. Acredito inclusive que, devido à importância que as linguagens orientadas a objetos possuem tanto na área acadêmica quanto no mercado de trabalho, a disciplina Programação Orientada a Objetos deveria ser obrigatória no currículo do BCC.

- **MAC0422 – Sistemas Operacionais:** importante tanto para entender como um computador funciona quanto para, baseado nesse conhecimento, criar programas melhores.

- **MAC0438 – Programação Concorrente:** essa disciplina foi bastante útil durante a criação dos seletores, devido aos inúmeros problemas encontrados com o uso de vários programas Perl que trocavam informações por meio dos *named pipes* fornecidos pelo Linux.

6.4 Futuro

Com o fim de minha iniciação científica, pretendo agora seguir no mestrado em bioinformática, sendo que o prof. Alan já aceitou ser meu orientador. Também devo criar os seletores para evidências do tipo multi-intervalo, estatística e gráfico.Além disso, quero continuar a atualizar a plataforma CoEd, não apenas para suporte ao EGene, mas para expandir suas funcionalidades de *framework* de editor de configurações.

Referências Bibliográficas

[1] Andrade, M. A., Brown, N. P., Leroy, C., Hoersch, S., de Daruvar, A., Reich, C., Franchini, A., Tamames, J., Valencia, A., Ouzounis, C., Sander, C. Automated genome sequence analysis and annotation. *Bioinformatics*, 15(5), 391-412. 1999.

[2] Ayoubi, P., Jin, X., Leite, X., Liu, X., Martajaja, J., Abduraham, A., Wan, Q., Yan, W., Misawa, E. & Prade, R.A. PipeOnline 2.0: automated EST processing and funcional data sorting. *Nucleic Acids Res.* 30(21), 4761-4769. 2002.

[3] B.Alberts, D.Bray, J. Lewis, M.Raff, K.Roberts and J.D. Watson. Molecular Biology of the Cell - Third Edition. Garland Publishing, Inc., New York, NY, 1994.

[4] Cerami, Ethan. XML for bioinformatics. Springer, 2005.

[5] CPAN. http://www.cpan.org. Novembro de 2006.

[6] Date, C.J. Introdução a sistemas de banco de dados. Elsevier, 2003.

[7] Durham, A.M., et al. EGene: a configurable pipeline generation system for automated sequence analysis. *Bioinformatics,* 21(12): 2812-2813. 2005.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[9] Ferro, Milene. Desenvolvimento de componentes de anotação de seqüências para o sistema EGene de geração de pipelines, 2006.

[10] Hotz-Wagenblatt, A., Hankeln, T., Ernst, P., Glatting, K. H., Schmidt, E. R. & Suhai, S. ESTAnnotator: a tool for high throughput EST annotation. *Nucleic Acids Res.*, 31(13), 3716-3719, 2003.

[11] Hudek, A. K., Cheung, J., Boright, A. P., Scherer, S. W. Genescript: DNA sequence annotation pipeline. *Bioinformatics* 19, 1177-1178. 2003.

[12] Java. http://www.sun.com/java. Novembro de 2006.

[13] Mount, David. Bioinformatics: Sequence and Genome Analysis. Cold Spring Harbor Laboratories Press, 2001.

[14] Wall, Larry; Christiansen, Tom; Orwant, John. Programação Perl. Campus, 2001.