Trabalho Prático 3 de AEDS 2

Interação Ligante Proteína Dayman Novaes

**INTRODUÇÃO**

Este é um algoritmo capaz de simular a força de interação entre ligantes e determinadas proteínas, escolhendo qual seria o ligante mais adequado, isto é, o que mais interage com a proteína.

O cálculo é simples, para um determinado raio no espaço, para cada átomo do ligante, é somado ao valor final, o número de átomos de proteína próximo à este átomo.

O algoritmo contém três TAD’s, sendo esses:

* Point
* Binder
* Octree

Os três serão detalhados a seguir.

Tendo como entrada as posições dos átomos da proteína e dos átomos do ligante, o programa deverá mostrar a força dos ligantes em ordem crescente.

**IMPLEMENTAÇÃO**

**ESTRUTURA DE DADOS**

* **Point**

Essa estrutura é a mais simples, contém apenas os valores X, Y e Z do ponto. É utilizada para representar cada átomo de uma proteína, e também cada átomo de um ligante.

* **Binder**

Essa estrutura representa o ligante, e foi simplificada. Ela não contém todos os seus átomos, apenas o nome do ligante e sua força de interação com uma proteína. Esse valor de força é atualizada no processamento de cada átomo do ligante.

* **Octree**

É a estrutura mais complexa, e representa a disposição de vários pontos em um espaço. Mas essa disposição não é feita de forma sequencial nem randômica. É uma estrutura semelhante uma árvore binária, porém, há uma divisão para cada dimensão. Como estamos falando de uma representação espacial, ou seja, três dimensões, há oito divisões, daí o nome “oct tree”.

Cada espaço (ou cada cubo) na Octree é representado por um ponto de origem (que fica no centro geométrico do espaço), e por um tamanho de aresta.

**FLUXO**

A principal função, que contém o fluxo principal do programa é a:

**void** readAllBinders(FILE \*file);

Essa função é a responsável por ler todas as entradas e definir o que deve ser feito com elas. Ela utiliza funções auxiliares como:

* **int** strIsBinderName(char \*str);
* **int** strIsProteinPoint(char \*str);
* **int** strIsBinderPoint(char \*str);

Esses auxiliares são para definir qual o tipo de entrada. Por exemplo:

* Se a entrada for o nome de uma proteína, devemos esvaziar totalmente a Octree (para inserir futuras proteínas), devemos atualizar a lista de ligantes com a proteína anterior (caso exista), devemos definir os novos parâmetros da nova Octree, e etc.
* Se a entrada for um ponto de uma proteína, devemos adicioná-lo na Octree.
* Se a entrada for um ponto de um ligante, devemos pegar esse ponto e comparar com a proteína já definida para definir a força do ligante.

A inserção dos pontos na Octree é feita pela função:

**void** insertPoint(Octree \*octree, Point point);

Ela basicamente considera os três casos de uma Octree. Se a Octree for uma folha, e tiver espaço livre, epenas insere o ponto na Octree, porém, se não há espaço vazio, a Octree é transformada em interior (não folha), seus oito filhos são criados, e o seu próprio ponto é inserido em um dos oitos, e depois o ponto em questão em inserida em um dos oitos. Observe que se essas duas inserções forem, coincidentemente, no mesmo filho, uma nova divisão ocorrerá, e assim sucessivamente. E no último caso, se a Octree não for folha, apenas inserimos o ponto em um dos oito filhos.

Já a inserção dos ligantes na lista já é feita de forma ordenada.

Sempre que a definição de um ligante termina, e a sua força de interação já foi completamente calculada, é chamada a função:

**void** insertBinder(Binder binder);

Essa função define primeiro a posição que o ligante deve ser inserido, baseado na sua força de interação, e então, após alocar mais uma posição de memória, faz um shift de uma posição de todos os ligantes seguintes. Depois insere o novo ligante na posição certa.

**ANÁLISE DE COMPLEXIDADE**

**TAD Point.h**

Todas as funções dessa estrutura são **O(1)**, pois têm custo constante independente da entrada.

**TAD Binder.h**

O mesmo caso se repete para cima.

**TAD Octree.h**

Todas as funções seguintes têm custo constante, independente da entrada:

* **int** getChildIndex(Octree \*octree, Point point);
* **Point** calculateOrigin(Point pmax, Point pmin);
* **Point** calculateHalfDimension(Point origin, Point pmax);
* **Point** calculateMaxPoint(Point origin, Point edges);
* **Point** calculateMinPoint(Point origin, Point edges);
* **Point** calculateOctreeMaxPoint(Octree \*octree);
* **Point** calculateOctreeMinPoint(Octree \*octree);

**()**

**CONCLUSÃO**

Obtive sucesso na implementação do trabalho ao obter os resultados esperados. A principal dificuldade não foi mais modularizar o código, visto que aprendi bastante com o primeiro TP. A dificuldade neste foi a alocação de memória dinâmica, pois ao lidar com várias listas diferentes, e como array de structs, e com manipulação e cópia de texto, obtive vários problemas com falha de segmentação na memória. Porém, no final consegui consertar todos e obtive um resultado satisfatório.