Relatório — Técnicas de Busca e Ordenação — Trabalho 1

Alunos: Fábio Henrique Pascoal, Gessica de Souza Silva e Nathan Garcia Freitas

1. Explicação de uso e tratamentos.

Neste trabalho, para rodar o código que executa a busca do menor caminho entre nós em um sistema, deve-se escolher a TAD desejada para testes. Caso queira usar a estrutura de Heap Binária, execute na linha de comando: make -f Makefile heap; Caso deseje usar a estrutura de Árvore de Busca Binária, execute na linha de comando: make -f Makefile bst. Ambos irão gerar o mesmo executável: main.out. Para executar o programa, basta seguir a linha de comando indicada nas especificações deste trabalho.

Quanto ao tratamento dos dados de entrada, fez-se a escolha de interpretar distâncias "bomba" contidas nos arquivos de input como impossibilidades de acesso entre dois nós, atribuindo o valor "0" como a distância padrão para caminhos bloqueados.

2. Implementação em Heap

A TAD da Heap Binária possui: um array de ponteiros genéricos (void *) para armazenar os dados, o tamanho desse array, o tamanho máximo que esse array pode possuir e por fim, um ponteiro para função de comparação referente aos dados que serão armazenados na heap binária. A heap possui certas peculiaridades em comparação com um vetor, tornando essa estrutura uma fila de prioridade. As principais propriedades da Heap são:

1. O mínimo do heap é sempre sua raiz (heap mínimo).
2. A relação entre um nó pai de índice *i* e seus filhos é baseado na fórmula: filho da esquerda = 2*i + 1; já o filho da direita = 2*i + 2; isso para heaps binárias com início no índice 0 (caso de nosso trabalho). Já de um nó de índice *j*, para descobrir o índice de seu pai basta a fórmula: j/2
3. Ao inserir um nó no array, inserimos no último slot do array, e com isso, realizamos a operação de heapify up nesse nó.
4. Ao removermos o mínimo de um heap, nós pegamos o último nó do array de dados e colocamos ele na raiz, com isso realizamos a operação de heapify down nesse nó.

Os algoritmos de heapify up e heapify down servem para a ordenação do vetor. Heapify up consiste em comparar o nó atual com seu pai, caso o nó atual seja menor, troque o filho com seu pai. Esse processo se perpetua até que o nó esteja em seu devido lugar. O heapify down segue o mesmo princípio mas para a direção oposta. O nó pai compara seus dois filhos (caso haja dois filhos) e verifica o menor, caso esse nó seja menor que seu filho, eles trocam. Caso esse nó possua apenas um filho, a comparação ocorre com esse filho.

A respeito das operações principais, temos: inicialização da heap, inserção, remoção do menor (heap mínimo), obtenção do tamanho, verificação se a heap está vazia, e por fim, destruição da heap.

3. Implementação em BST

Buscando uma segunda TAD para realizar a busca do menor caminho para o nó desejado, foi escolhido uma Binary Search Tree (BST). O conceito da BST consiste numa árvore com as seguintes propriedades:

1. A árvore possui diversas folhas, também chamadas de nós. Esses nós possuem sempre duas folhas, uma à esquerda e outra à direita.
2. As folhas que estão à esquerda de um nó sempre possuem a chave de identificação menor do que a desse nó.
3. As folhas que estão à direita de um nó sempre possuem a chave de identificação maior do que a desse nó.
4. Não há nós com chaves repetidas.

As funções implementadas consistem em: criação, destruição, inserção, remoção, obtenção do menor, remoção do menor, obtenção do maior, remoção do maior e verificação se a árvore está vazia.

Para realizar o Algoritmo de Dijkstra, segue o mesmo princípio da Heap Binária, visto que, como ambas são Priority Queues (Filas de Prioridade), e a implementação do algoritmo estava genérica para quaisqueres filas de prioridades, não houve mudanças.

4. Comparação entre implementações

A respeito agora da complexidade, temos:

Algoritmo	Inserção	Remoção do menor	Obtenção do menor
Heap Binário	$O(\log(n))$	$O(\log(n))$	1
Árvore Binária de Busca	$O(N)$	$O(N)$	$O(N)$

Tabela 1: comparação entre a complexidade de tempo de operações padrão de uma fila de prioridade para as duas implementações realizadas

Observando a Tabela 1, nota-se que o heap binário sempre possuirá uma complexidade de tempo menor do que a árvore binária de busca NÃO BALANCEADA. Além disso, como na notação Big O estamos tratando o pior caso dos algoritmos, e visto que o pior caso de uma BST consiste numa árvore degenerada, com um comportamento de lista encadeada, ambas operações de inserção e remoção possuem complexidade linear, enquanto para o heap, a inserção e remoção possuem a mesma

complexidade de $O(\log(n))$, devido aos algoritmos de ajuste da propriedade da heap. Já a obtenção do menor, como dito anteriormente nas propriedades da Heap Binária, em um heap mínimo, a raiz é sempre o menor valor de todo o array, garantindo um acesso constante ao menor dado da heap. Para que haja de fato uma comparação justa, a árvore que deve-se comparar a um heap binário é uma árvore balanceada, como por exemplo uma árvore rubro-negra, que possuirá inserção, remoção e obtenção do menor com complexidade: $O(\log(n))$, sendo um competidor a altura do heap binário.

Partindo para a comparação no quesito do tempo, seguem os resultados obtidos para os tempos de execução dos algoritmos para cada entrada disponibilizada. Os valores finais de cada tempo de execução de algoritmo por entrada é a média de cinco execuções do script.

Entrada (.txt)	**Tempo BST (segundos)**	**Tempo Heap (segundos)**
caso_teste_muito_pequeno_1	0.000004	0.000002
caso_teste_muito_pequeno_2	0.000006	0.000004
caso_teste_pequeno_1	0.000542	0.000589
caso_teste_pequeno_2	0.000470	0.000487
caso_teste_pequeno_3	0.071429	0.068873
caso_teste_pequeno_4	0.072947	0.068912
caso_teste_medio_1	3.755824	3.390927
caso_teste_medio_2	3.372645	3.482253
caso_teste_medio_3	3.560783	3.609790
caso_teste_medio_4	5.267723	5.260532

Tabela 2: comparação entre o tempo de execução do algoritmo de Dijkstra para as duas diferentes implementações de uma fila de prioridade

Como pode-se observar na Tabela 2, os tempos entre os algoritmos são extremamente aproximados para as entradas fornecidas, porém, para uma entrada massiva de dados, a fila de prioridade implementada usando Heap será consideravelmente mais eficiente em termos de tempo de execução do que a implementada com BST.

Uma particularidade a se comentar a respeito do uso da BST na implementação de uma fila de prioridade mínima, é que na remoção do elemento de menor prioridade, considerando uma sequência de inserções completamente ordenada, essa operação terá custo O(1), ou seja, constante, visto que, o menor nó sempre será a raiz, embora nesse caso a operação de inserção fique totalmente prejudicada e seja O(N).