AEDS3 – TP3

Nome: João Pedro Samarino

1. Introdução

O trabalho teve como objetivo a implementação de um programa que dada uma área fictícia, um conjunto de robôs deve monitorar determinados pontos pré estabelecidos, porem existe o problema da alocação de pontos para cada robô. Para que se tenha eficiência o programa deve alocar um grupo de pontos para cada agente robótico de maneira a conseguir utilizar a menor quantidade de robôs, considerando também que a eficiência esta indiretamente relacionada ao raio de alcance máximo dos robôs, pois quanto maior o raio menos robôs seriam necessários.

Os problemas de coloração são bem comuns na área de ciência da computação, este problema em questão pode ser classificado nesta categoria, pois é possível interpretar este problema como um grafo, onde deve-se colorir cada vértice, dessa maneira cada cor faz referencia a um conjunto de pontos (vértices) para um dado robô.

Então representando este problema como um grafo temos que os vértices são pontos de referencia e as arestas representam que um robô naquele ponto não alcança o outro ponto interligado. Dessa maneira se colorirmos o grafo teremos o conjunto de pontos para cada robô, no entanto esse problema é do tipo NP completo, ou seja, a única solução exata conhecida é fatorial, a mesma foi implementada nesse trabalho de forma paralela e seqüencial , também foi desenvolvida uma heurística que obtém na maioria dos casos um bom resultado.

2. Solução Proposta

A idéia por trás da solução deste problema é tentar achar todas as combinações de cores possíveis entre os vértices do grafo, esse tipo de solução tem um custo computacional muito elevado, porem nesse caso é o único modo de achar a solução exata do problema como foi pedido. Também foi desenvolvido uma heurística que muitas vezes tem um bom resultado.

Para a solução desse problema foram usados seis tipos abstratos de dados com o intuito de facilitar a manipulação das estruturas pelas funções desenvolvidas no mesmo, já os procedimentos é funções foram subdivididas pensando na modularidade do código, ou seja, utilizando a idéia de reaproveitar partes de rotinas já escritas.

A solução proposta para o problema constitui em duas idéias, uma para obter a solução exata, outra para obter uma solução aproximadamente boa para a maioria dos casos (heurística). A solução exata é composta por algoritmos que fazem a permutação de todos os vértices para a coloração, ou seja, o algoritmo verifica todas as colorações possíveis é guarda a melhor solução. No algoritmo da heurística a idéia é ordenar os vértices pela quantidade de arestas é colorir na ordem dessa ordenação, fazendo esse procedimento é possível obter um bom

resultado, pois essa técnica reduz o numero de conflitos de corres , sendo possível utilizar menos cores diante algumas outras técnicas pensadas.

2.1. Estrutura de Dados:

Para entendermos o funcionamento do programa primeiro temos que entender as estruturas de dados abstratas criadas nesta solução, para então explorarmos os algoritmos de maneira completa. Foram criadas cinco estruturas básicas para armazenamento de registros de maneira mais abstrata, ou seja, para facilitar a manipulação pelas funções dos registros lá armazenados. As estruturas de dados estão contidas nos arquivos (lista.h) e (grafos.h), separadas por finalidades, no arquivo (lista.h) existem somente estruturas relacionadas a lista implementada, já em (grafos.h) existem as estruturas relacionadas ao grafo do problema .

As estruturas de dados abstratos são:

- typedef struct _Celula { int chave; double x; double y; struct _Celula * proximo; struct _Celula * anterior; } Celula;

A estrutura (Celula) é a estrutura básica do programa, todas as outras estruturas a contem internamente, está é responsável por armazenar os itens para o funcionamento correto das funções, estes itens são, (chave) responsável pro guardar a chave indicador da célula , (x e y) responsável pelas coordenadas dos pontos, (*proximo) ponteiro responsável por apontar para a célula seguinte da lista, (*anterior) ponteiro responsável por apontar para a célula anterior.

- typedef Celula* Apontador;

Esta estrutura feita para abstrair o apontamento da estrutura (celula), ou seja, simplificando o uso do ponteiro quando necessário.

- typedef int TipoChave;

Esta estrutura define o indicador da célula como um inteiro, foi escolhido esse tipo de variável, pois ela e a menor estrutura que atendia os requisitos impostos pelos algoritmos.

- typedef struct {Apontador primeiro; Apontador fim;} Lista;

A estrutura (Lista) é responsável por armazenar dois ponteiros do tipo (Apontado), a estrutura foi constituída desta maneira porque a alocação das células e feita de maneira dinâmica, assim os ponteiros (primeiro) e (fim) definem só as células do inicio e do fim da lista, desta maneira podemos começar a percorrer a lista do inicio ou do fim, melhorando o tempo de acesso dos dados da estrutura.

-typedef struct {Lista* indicador; int *cores, *cores_f; int qt_v; int qt_cor;} Grafo;

A estrutura (Grafo) é responsável por armazenar ponteiros dos tipos (Lista) e (int) , o ponteiro (indicador) é responsável por armazenar a lista de vértices conectado ao vértice do índice

referente , é os ponteiros (cores, cores_f) são responsáveis consecutivamente por armazenar as cores da coloração atual e o outro as cores da melhor coloração. As variáveis (qt_v, qt_cor) são responsáveis por armazenar a quantidade de vértices e a quantidade de cores utilizadas na melhor coloração.

- typedef struct{ Grafo g; int *buffer; } Paralelo;

Essa estrutura foi feita somente para enviar os dados para a função paralela, pois era necessário condensar os dados em uma única estrutura antes de enviá-los. O apontador (buffer) é responsável por mandar a seqüência da coloração para a função que colore o grafo de maneira paralela.

2.2.Funções e Procedimentos

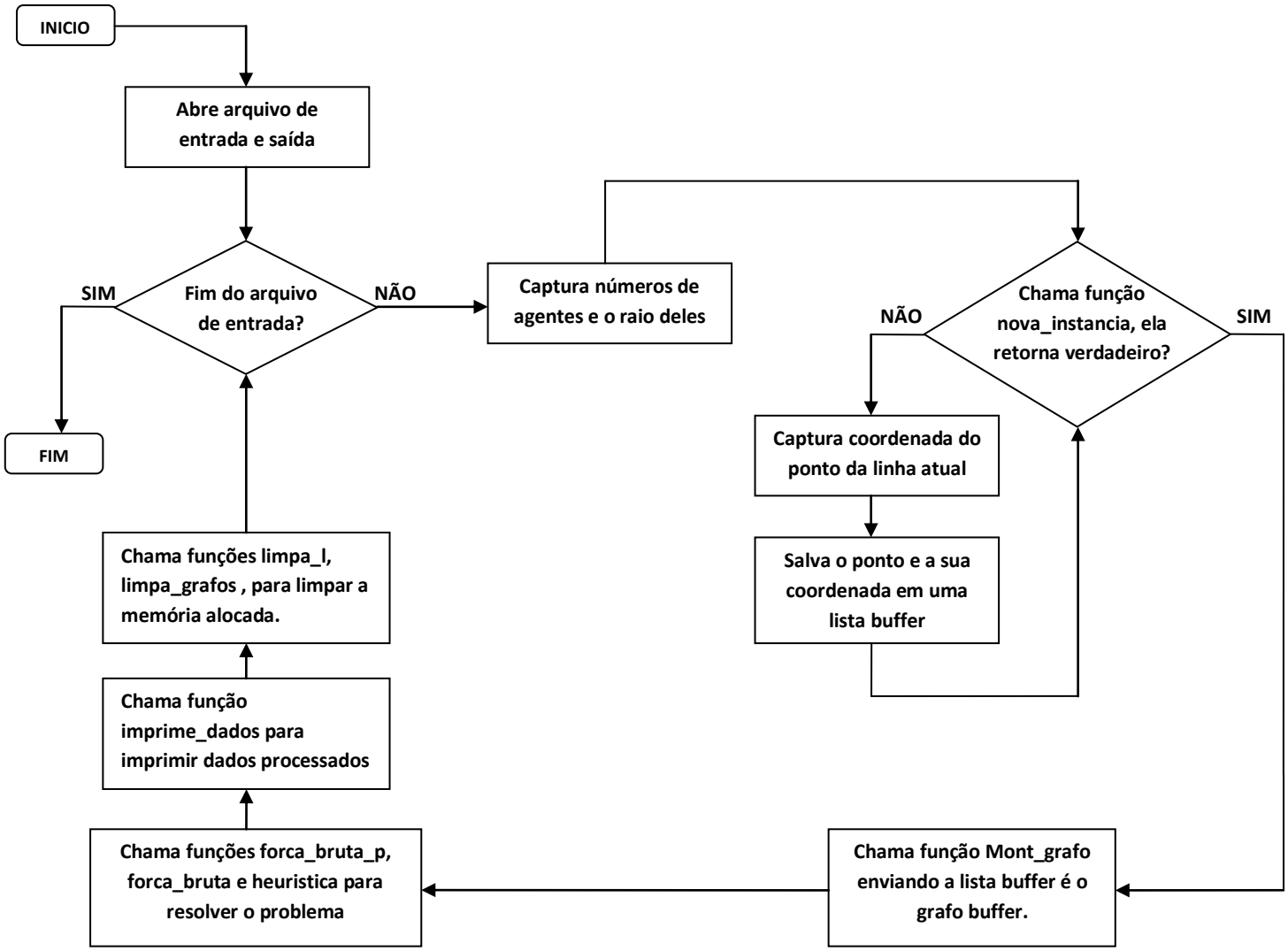
No programa foram criadas funções é procedimentos para manipulação de dados é execução da proposta de solução que tem como objetivo gerar o arquivo solução. O arquivo (main.c) contem procedimentos do fluxo principal e o (grafos.c) possui procedimentos relacionados ao TAD da manipulação de grafos, contendo os procedimentos e funções das colorações necessárias. No arquivo (lista.c) existem os procedimentos de manipulação da estrutura da lista encadeada que são usados continuamente pelas funções da TAD de grafos.

Funções e procedimentos do arquivo (main.c) :

*- int nova_instancia(FILE *arq):* Essa função recebe o ponteiro do arquivo de entrada é tem como objetivo verificar se uma nova instancia começou em relação a leitura atual do arquivo ,lembrando que para fazer essa verificação a mesma só verifica se existe na leitura atual duas quebras de paginas consecutivas se houver ela retorna (1) caso contrario (0).

-*void imprime_dados(FILE *out, Grafo *g, int n_agt):* Esse procedimento é responsável por imprimir os dados no arquivo de saída especificado pelo ponteiro (*out*), o mesmo faz a leitura do grafo já processado é imprime as informações relevantes, o mesmo ainda verifica se a solução é possível ,ou seja, se a quantidade de cores é menor que o numero de agentes especificado pela variável (*n_agt*) se não for imprime que não houve solução.

- Função Principal *(int main(int argc, char *argv[])):* A função “main” compõem o fluxo principal do programa , deste modo tem como objetivo principal capturar informações e realiza todas as chamadas ao TAD de grafos quando necessário, por fim enviar os valores obtidos das funções que trabalham com o arquivo de saída. Esta função é a mais extensa do programa porem a mesma é extremamente simples, pois não realiza nenhuma operação para o processamento dos dados do problema, somente direciona o fluxo do programa é as funções que devem ser utilizadas em determinadas ocasiões. O fluxograma na próxima pagina da função “main” mostra como funciona a lógica de execução do programa detalhadamente e nele é possível entender melhor o funcionamento do programa é sua idéia como um todo.



Funções e procedimentos do arquivo (lista.c) :

- void FLVazia(Lista *l): Procedimento responsável por alocar na lista uma célula cabeça, este procedimento tem como finalidade de evitar erros de apontamento caso a lista esteja vazia, depois de alocar a célula cabeça ele define os apontadores da lista (l) para esta célula.

- void Insere_l(TipoChave x, Apontador posicao,Lista *l, int con_lfu): Procedimento responsável por inserir uma célula nova na lista (l) , nesta célula e atribuído os valores da chave (x) e do contador (con_lfu). A inserção e feita na frente da célula enviada (posicao), o procedimento e feito através da troca de apontamentos das células da lista, caso exista a célula cabeça na lista(l) ela e deletada para não afetar a lógica do programa.

- void limpa_l(Lista *l): Procedimento responsável por limpar a lista (l) , para fazer isso o procedimento percorre a lista e vai excluindo as células em que passa até o final da lista. No final do processo ele chama a função (FLVazia) para criar uma nova célula cabeça na lista (l).

- int Busca_l(TipoChave chave, Lista *l, Apontador *celula): Função responsável por buscar a célula que possui o valor de sua chave igual (chave) na lista (l) , a função percorre do inicio da lista em direção ao final, a cada célula que a mesma pula e incrementado um valor em uma variável genérica (i) , quando a célula procurada é achada o apontador da mesma é copiado para (*celula) é a função retorna o valor de (i) .

-int Remove_l(Apontador posicao, Lista *l):Função responsável por deletar a célula (posicao) da lista (l) , este procedimento e realizado mudando quatro ponteiros das células para não se perder a seqüência da lista em questão. Quando os ponteiros são mudados o espaço que a célula apontada pelo ponteiro (posicao) é desalocado da memória. Retorna (1) se for possível remover o item.

Funções e procedimentos do arquivo (grafos.c) :

- void Mont_grafo(Lista *l,Grafo *f,double *raio,int qt_itens): Esse procedimento tem como objetivo montar a estrutura de grafos (f), para fazer isso ele verifica a lista (l) é compara a distancia do raio do robô e de acordo com as ligações entre os pontos ele monta o grafo , sempre alocando uma fila de arestas para anexá-la a estrutura do grafo.

- void colore_grafo (Grafo *f, int* buffer): O objetivo desse procedimento é colorir o grafo (f), percorrendo os vértices pela ordem do vetor (buffer), a cada vértice colorido a função verifica se existe outro grafo interligado com a mesma cor, se houver tenta uma cor de índice maior ate obter uma cor valida para o vértice atual, o procedimento faz isso para todos os vértices do grafo, a cada vértice verifica se a melhor coloração já é melhor do que a atual, se for finaliza o procedimento, caso contrario continua a coloração, se chegar ao final do grafo armazena a coloração na estrutura de dados.

- void permuta_forca_bruta(Grafo *f, int* buffer, int k): Esse procedimento é a parte recursiva do procedimento (forca_bruta), ele gera todas as permutações do vetor buffer, e a cada vez que obtém uma nova permutação chama a função (colore_grafo) para a mesma fazer a coloração referente a essa permutação, o algoritmo funciona de maneira recursiva e a cada chamada recursiva realiza uma troca de posição no vetor (buffer) até obter uma nova permutação.

- void forca_bruta(Grafo *f): O procedimento (força_bruta) aloca o vetor buffer para enviar para a função (permuta_forca_bruta), e ao termino da mesma ela limpa a memória alocada pela variável e finaliza.

- void *colore_grafo_p(void *paramentros): Este procedimento funciona exatamente igual ao procedimento (colore_grafo) porem tem como diferença o fato de ser um procedimento paralelo , ou seja pode ser executados vários simultaneamente.

- void permuta_forca_bruta_p(Paralelo *p, int* buffer ,int k ,int *buffer_t, pthread_t *t_id): O procedimento é semelhante a (permuta_forca_bruta) porem a diferença está no controle de thread que o mesmo realiza, a variável (k) é o numero de thread simultâneas executando, ou seja, o procedimento vai executando e a cada permutação nova chama uma nova thread e

incrementa (k), porem quando a quantidade (k) for maior que o limite de thread o mesmo espera todas as threads acabarem para poder então continuar a executar.

- void forca_bruta_p(Grafo *f): O procedimento aloca (N) novas estruturas de (f) em (Paralelo) para o processamento paralelo , onde (N) é a quantidade máxima de thread , ou seja , esse procedimento faz uma copia dos dados de (f) para novas estruturas que são repassadas para o procedimento (permuta_forca_bruta_p) para então serem processadas.

- void heuristica(Grafo *f): Tem como objetivo achar uma ordem para a coloração que obtenha um bom resultado, o procedimento aloca um vetor buffer para guardar os índices dos vértices que vão ser executados pelo procedimento (colore_grafo), o mesmo para obter essa ordem faz uma varredura no grafo a fim de deixa o vetor (buffer) na ordem do índice do vértice que possui mais ligações "arestas" sendo a primeira posição do vetor é o que possui menos ligações sendo o ultimo do vetor, quando ordena o vetor buffer chama o procedimento (colore_grafo) para processar.

-void limpa_grafo(Grafo *f): Esse procedimento simplesmente tem como objetivo de limpar as estruturas alocadas para serem novamente usadas em outras instancias ou para a finalização do programa.

2.3. Estratégia para o processamento paralelo

Para realizar a paralelização do programa o algoritmo responsável (forca_bruta_p) realiza as seguintes tarefas, primeiro ela gera uma estrutura de acordo com a quantidade de threads que iram rodar simultaneamente, depois chama a função (permulta_forca_bruta_p) essa função gera as permutações e faz o controle das threads da seguinte maneira, enquanto houver threads livres dispara um processo para cada uma, quando acabar as threads livres espera todas acabarem para então continuar a fazer esse processo ate acabar as seqüências a serem permutadas. Quando uma thread é criada ele executa o procedimento (colore_p), durante a coloração algumas variáveis são mudadas, mais essas variáveis são únicas para cada thread, foram criadas durante a execução de (forca_bruta_p) sendo assim não foi preciso colocar nenhuma trava para fazer o controle de escrita das mesmas, no final o procedimento (forca_bruta_p) junta as informações para obter a solução final.

3.Análise de Complexidade

Procedimentos e funções, FLVazia , Insere_l , Remove_l: Estes tem uma complexidade de tempo O(1). Esta complexidade de tempo e devido às atribuições que estes procedimentos fazem, todos fazem operações ou chamadas de custo fixo, como atribuição de valores ou mudança em ponteiros. A complexidade de espaço é O(1) pois são geradas neste procedimento somente variáveis buffers é no caso de (insere_l) a mesma cria de uma célula a cada chama, continuando sendo sua complexidade de espaço O(1).

Procedimento limpa_memoria , limpa_l e limpa_grafo: São procedimento tem uma complexidade de tempo O(n), onde n e a quantidades de células na estrutura de referencia que a função recebe, esta complexidade se da devido ao loop internos que percorrem a lista ou um grafo. A complexidade de espaço é O(1) pois são geradas neste procedimento somente variáveis de buffer.

Função Busca_l : Esta função é um pouco diferente das outras , pois ela nem sempre percorre toda a lista , ela percorre somente ate achar o valor procurado ,se não achar ela chega ao final, desta maneira no pior caso tem uma complexidade de tempo O(n) ,onde n e a quantidades de células na estrutura. A complexidade de espaço é O(1) pois são geradas neste procedimento somente variáveis de buffer.

Procedimento Mont_grafo e heuristica: Estes realizam operações que causam uma complexidade $O(n^2)$ onde n é a quantidade de cédulas da estrutura, pois ambas as funções fazem comparações de todas as celas com cada uma delas. A complexidade de espaço do procedimento (Mont_grafo) é $O(n^2)$ no pior caso onde todos os vértices estão interligados a todos ou outros, já o procedimento (heurística) não aloca nenhuma memória considerável , apenas variáveis de buffers.

Procedimento colore_grafo e colore_grafo_p: Estes realizam operações que causam uma complexidade de tempo O(V + A) onde V é o numero de vértices e A o numero de arestas do grafo , essa complexidade é devido ao fato do programa percorrer todos os vértices é em cada um dele verifica os vértices que estão interligados . Já a complexidade de tempo é O(1), pois não aloca memória intermitente.

Procedimento forca_bruta e forca_bruta_p : Estes realizam chamadas a um procedimento O(FAT(V)) , onde V é a quantidade de vértices , alem da chamada executam operações constantes , sua complexidade então é a mesma. Já a complexidade de espaço de força_bruta é O(v), pois a mesma aloca um vetor buffer para enviar para outras funções como parte da lógica de execução, e força_bruta_p divide os dados para cada processo paralelo então sua complexidade de espaço é $O(n^2*N)$, onde n representa o numero de células da estrutura é N representa o numero de processos que vão rodar em paralelo.

Procedimento permuta_forca_bruta_p e permuta_forca_bruta: Esses são os procedimentos que fazem todas as permutações possíveis, ou seja, tentam todas as cores para colorir o grafo , ambos são procedimentos recursivos que possuem uma complexidade fatorial devido a quantidade de permutações que geram a cada nova chamada, dentro de cada chamada recursiva chamam uma função O(V+A) , porem isso não ira afetar a ordem de grandeza da complexidade de tempo que é O(FAT(V)). A complexidade de espaço é O(1), pois não aloca memória permanente durante o processo.

Função main – função principal: Podemos analisar a complexidade somente de cada instancia, pois as instancias variam de tamanho então a complexidade de cada uma será distinta, então para uma instancia genérica a função main executa chamadas a procedimentos O(1) , O(n) é O(FAT(v)), logo sua complexidade de tempo será de O(FAT(v)). Já sua complexidade de espaço é $O(n^2*N)$ que é gerado pelo procedimento que realiza operações em paralelo, o qual é chamada na função principal.

4.Implementação

Para desenvolver este trabalho, o mesmo foi dividido em cinco arquivos com o intuito de melhorar a organização e normalizar as funções e estruturas de acordo com os seus objetivos.

4.1.Arquivos utilizados

(main.c): Arquivo principal, responsável pelo fluxo do programa, contem funções relacionadas a entrada e saída de dados dos arquivos, contem a função (main()), que controla todas as chamadas ao TAD.

(grafos.c): Arquivo responsável pelos procedimentos relacionados ao TAD, ou seja, aqueles que iram fazer operações diretamente com a estrutura de dados relacionada.

(grafos.h): Este arquivo contem o cabeçalho das funções relacionadas ao TAD que estão contidas no arquivo (grafos.c), Também possui as estruturas de dados ligadas ao programa e suas definições.

(Lista.c): Arquivo relacionado ao TAD da lista encadeada implementada no programa.

(Lista.h) : Este arquivo contem o cabeçalho das funções relacionadas ao TAD da lista e estruturas de dados as suas definições.

4.2. Compilação

O programa deve ser compilado no compilador GCC através de um makefile ou do seguinte comando no terminal:

gcc main.c lista.c grafos.c -pthread -o tp3

4.3. Execução

A execução do programa tem como parâmetros:

• Arquivo de entrada.

• Arquivo de saída.

O comando no terminal para a execução do programa e da forma:

./tp3 < arquivo de entrada > < Arquivo de saída >

OBS: Como foram implementados três algoritmos com a mesma função, para rodar somente com um, se deve comentar a chamada dos outros na função (main), na mesma já existe a indicação de onde deve ser feito o comentário.

5. Experimentos

Os experimentos realizados tiveram como objetivo verificar o funcionamento do programa é o tempo gasto para execução do mesmo, verificando assim a qualidade dos algoritmos implementados e seus desempenhos.

5.1. Maquina utilizada

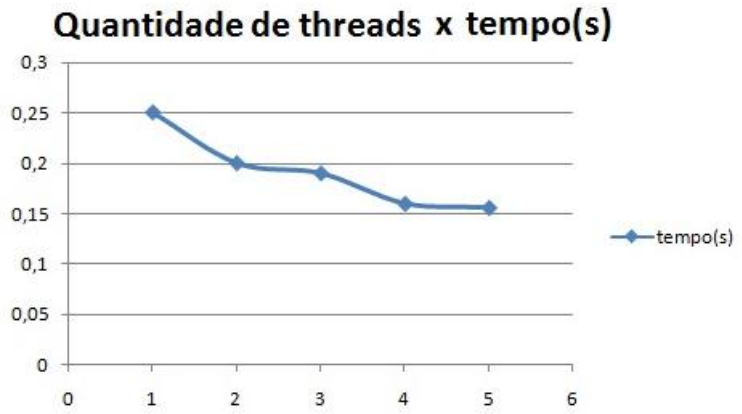
Os testes e a compilação do programa foram realizadas em um Pentium core 2 duo, com 3 Gb de memória principal, o sistema operacional utilizado foi o Ubuntu Linux para arquitetura AMD64 (64 bits por ciclo de maquina) .

5.2. Teste

O primeiro teste realizado foi o de numero de pontos por tempo, foram criadas instancias genéricas com as seguintes quantidades de pontos : 5,6,7,8,9 e 10 . O gráfico abaixo mostra a relação de tempo por quantidade de pontos na instancia, vale ressaltar que o algoritmo rodado foi o da força bruta que tem uma complexidade de tempo fatorial.

Podemos perceber por esse gráfico que o tempo da um salto gigantesco, mais isso e devido à complexidade do problema, tal saída já era espera, pois o problema foi resolvido através da técnica força bruta.

O segundo gráfico mostra a relação de tempo para solucionar uma mesma instancia, porem o que varia agora é a quantidade de threads rodando simultaneamente para resolver o problema.



Podemos notar que com a variação inicial da quantidade de threads existe uma queda maior, no entanto com o aumento da mesma a queda tende a não ser mais acentuada, isso é devido ao tempo de sincronização do programa, cada thread processa uma parte do problema separadamente, porem depois de processada existe um tempo para juntar as soluções, quanto maior é a quantidade de threads maior ficara o tempo para juntar os dados, então a

quantidade de threads rodando simultaneamente deve ser avaliada em relação a necessidade sempre olhando para o custo beneficio da paralisação do problema.

5. Conclusão

Para realizar este trabalho foi necessário aprender um pouco do funcionamento da biblioteca de paralelismo "*Pthreads"*, a elaborar uma heurística e desenvolver um algoritmo "força bruta" para fazer a permutação em um determinado conjunto.

Houve muitas dificuldades no decorrer do processo, principalmente em relação à parte paralelizada do programa, por não conhecer o funcionamento e as praticas desse tipo de programação o resultado obtido foi satisfatório apesar de que com um numero grande de threads rodando simultaneamente o programa gera alguns erros e outras pequenas peculiaridades, os outros dois algoritmos (heurística) e (força bruta) tiveram uma implementação mais limpa é direta, obtendo um ótimo desempenho.

Com esse trabalho agreguei conhecimento dos assuntos práticos passados em sala de aula, heurística, NP Completo e a elaboração de algoritmos força bruta. Cheguei à conclusão que para solucionar esse tipo de problema em tempo hábil não existe uma forma de sempre achar a melhor solução mais na maioria dos problemas é possível obter um algoritmo que gera um resultado satisfatório com um gasto computacional aceitável.