UFMG - UNIVERSIDADE FEDERAL

DE MINAS GERAIS

AEDS 3 -TRABALHO PRÁTICO 3

Nome: João Pedro Samarino

Matricula:2013048933

Belo Horizonte

2014

UFMG

**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.

**Abre arquivo de entrada e saída**

**INICIO**

**Chama função nova\_instancia, ela retorna verdadeiro?**

**Captura números de agentes e o raio deles**

**NÃO**

**SIM**

**Fim do arquivo de entrada?**

**SIM**

**NÃO**

**Captura coordenada do ponto da linha atual**

**FIM**

**Chama funções limpa\_l, limpa\_grafos , para limpar a memória alocada.**

**Salva o ponto e a sua coordenada em uma lista buffer**

**Chama função imprime\_dados para imprimir dados processados**

**Chama funções forca\_bruta\_p, forca\_bruta e heuristica para resolver o problema**

**Chama função Mont\_grafo enviando a lista buffer é o grafo buffer.**

**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.

**3.Análise de Complexidade**

Procedimento Aloca: O procedimento tem uma complexidade de tempo O(n\*m) onde n é o numero de linhas da matriz a ser alocada e m representa as colunas, esta complexidade se da devido aos loops que percorrem toda a estrutura. A complexidade de espaço do programa é O(n\*m), pois esse procedimento tem como objetivo alocar uma estrutura em forma de matriz através da função malloc().

Procedimento marca\_bloqueio: O procedimento tem uma complexidade de tempo e espaço é O(1), pois faz somente uma operação de atribuição.

Procedimento faz\_matriz\_custo: O procedimento tem uma complexidade no pior caso O((n\*m)²), onde n e m se referem consecutivamente a quantidades de linhas e colunas da matriz resposta, está complexidade no pior caso é divido a possibilidade de cada (n) células da matriz chegarem a fazer (n) chamadas, já a complexidade no melhor caso é O(n\*m) onde o procedimento marca todos os valores corretamente. A complexidade de espaço é O(1) pois são geradas neste procedimento somente variáveis de buffer.

Procedimento acha\_qt\_caminhos e limpa\_estrutura: Os procedimentos tem uma complexidade de tempo O(n\*m), onde n e m se referem consecutivamente a quantidades de linhas e colunas da matriz de custo, esta complexidade é divido ao fato de que a função pode fazer no máximo (m\*n) chamadas recursivas pois o numero de menores caminhos é limitado ao numero de células.

A complexidade de espaço é O(1) pois são geradas neste procedimento somente variáveis de buffer.

Função main – função principal: a função principal faz 1 varreduras que podem varia de 0 a X , onde X é a quantidade de instancias informada no arquivo de entrada, no final de cada ciclo o programa chama as funções da TAD , porem ele pode chamar ate X vezes uma função O((n\*m)²) alem de executar comando O(1) . Temos no pior caso O(x)\* O((n\*m)²)+O(1)=O((n\*m)²\*x) . Porem essa analise só pode ser feita considerando que todas as matrizes (cidades) da instancia vão ter o mesmo tamanho mantendo assim a relação “n”,”m” é sempre igual, se o tamanho das matrizes varia a complexidade de tempo pode variar também.A complexidade de espaço é O(n\*m), o programa pode chamar X vezes uma função de complexidade O(n\*m) porem a cada novo ciclo o espaço alocado e limpado para poder ser novamente alocado com um novo tamanho.

# 4.Implementação

Para desenvolver este trabalho, o mesmo foi dividido em três 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.

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

(caminhos.h): Este arquivo contem o cabeçalho das funções relacionadas ao TAD que estão contidas no arquivo (matriz.c), Também possui as estruturas de dados ligadas ao programa e 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 caminhos.c -o tp2*

**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:

*./tp2 < arquivo de entrada > < Arquivo de saída >*

**4.3.1. formato dos arquivos**

Arquivo de entrada: O arquivo de entrada deve conter as especificações básicas dos procedimentos que iram ser realizados pelo programa. Este ira conter na seguinte ordem formatada, (Instancias), (dimensão da cidade(1)), (números de bloqueios da cidade(1)),(localização dos bloqueios), (numero de testes) ,(localização dos lugares testados) e assim sucessivamente para cada instancia.

**5. Experimentos**

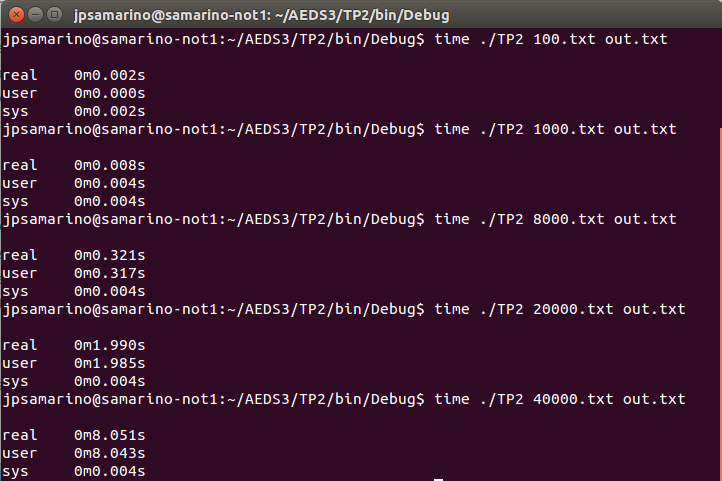
Os experimentos realizados tiveram como objetivo verificar o funcionamento do programa é o tempo gasto para execução do mesmo, verificando assim a qualidade dos algoritmos implementados no mesmo.

**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**

Para verificar o desempenho do programa foram realizados alguns testes. Os arquivos de testes contem uma instância com uma cidade de (N) igual a consecutivamente 100, 1000, 8000, 20000, 40000, sendo (N) a quantidade de cruzamentos, ou seja, (m\*n), realizando somente (1) teste por arquivo. Foi utilizado o comando time para medir o tempo de execução do programa para a entrada determinada. Podemos ver abaixo na imagem o tempo de execução para cada entrada.



Podemos notar que com o aumento de (N) o incremento do tempo não foi linear, pois como dissemos na analise de complexidade a mesma representa um polinômio que pode chegar a ser de grau (2). Abaixo existe um gráfico de (N x Tempo) onde podemos mostrar a relação entre o tempo de execução é a quantidade de instâncias processadas.

# 

Podemos notar por esse gráfico que se obtém uma solução com um polinômio aproximadamente de grau (2), como podemos visualizar, a curva não cresceu muito rapidamente, comprovando desta maneira a analise de complexidade citada na seção anterior desse trabalho.

# 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.

Foi possível com este trabalho aprender mais sobre paradigmas de programação como recursão e programação dinâmica, apesar da solução ser muito simples a elaboração da mesma foi um pouco custosa, achar uma maneira de representar computacionalmente este tipo de problema demandou uma quantidade considerável de tempo, pois a sua solução não é trivial. Durante a programação surgiram algumas dificuldades, como não tenho pratica com implementações de funções recursivas, tive dificuldades para fazer funcionarem corretamente.

Em relação os algoritmos programados cheguei à conclusão que a implementação por meio de funções recursivas deixa a etapa de codificação muito mais simples, acredito que a minha solução para este problema demanda muito tempo é deve existir uma maneira consideravelmente mais rápida, porem eu ao desenvolver essa solução não me deparei com outras formas para chegar à solução da mesma. A solução do algoritmo foi testada em alguns exemplos que eu mesmo desenvolvi e resolvi. Sempre o algoritmo acha a resposta certa, concluindo que o mesmo converge para a resposta em um tempo considerável.