Grafos e Algoritmos – Tarefa T1.2

Verificação de Clique e Percurso em Largura

Ricardo C. Corrêa

17 abril 2024

1 Problemas

Esta tarefa envolve a realização de experimentos. Os problemas tratados nos experimentos desta tarefa são definidos a seguir. Os tipos abstratos `GRAFO` e `CONJ` são discutidos na sequência.

1.1 Verificação de Clique

ENTRADA: `GRAFO` G, `CONJ` S.
QUESTÃO: verificar se S define, ou não, uma clique em G.

1.2 Percurso em Largura

ENTRADA: `GRAFO` G, vértice r de G.
QUESTÃO: determinar uma ordem em largura dos vértices de G a partir do *vértice de partida* r.

2 Tipos Abstratos

2.1 Conjuntos

Os tipos abstratos de base para as operações sobre grafos estão associados a manipulação de conjuntos. As funções desses tipos são as seguintes.

```
#define NACO unsigned long long   // tipo do NACO

#define bool NACO
#define true (NACO) 1
#define false (NACO) 0

// Tipo CONJ
typedef struct {
} CONJ;

// Cria um novo conjunto com a amplitude indicada.
// Retorna o conjunto criado.
CONJ conjunto(int ampl) {
}

// Apaga um conjunto da memória, liberando a memória por ele ocupada
void apagar(CONJ * R) {
}
```

```c
// Retorna a cardinalidade de um conjunto
static inline unsigned int cardinalidade(const CONJ * R) {
}

// Replica um conjunto especificado em um novo conjunto.
// O conjunto especificado não pode ser alterado.
// Retorna o conjunto criado.
CONJ replicar(const CONJ * R) {
}

// Copia os elementos de um conjunto de origem R para um conjunto de destino S.
// Ambos os conjuntos devem ter a mesma amplitude.
// O conjunto de origem não pode ser alterado.
static inline void copiar(const CONJ * R, CONJ * S) {
}

// Remove todos os elementos de um conjunto especificado, deixando-o vazio.
static inline void esvaziar(CONJ * R) {
}

// Verifica se um valor pertence a um conjunto.
static inline bool pertence(unsigned int i, const CONJ * R) {
}

// Incluir um elemento em um conjunto.
static inline void incluir(unsigned int i, CONJ * R) {
}

// Retirar um elemento em um conjunto.
static inline void retirar(unsigned int i, CONJ * R) {
}

// Operações sobre conjuntos

// Verifica se um conjunto R é subconjunto de outro conjunto S.
// Ambos os conjuntos devem ter a mesma amplitude.
static inline bool subconj(const CONJ * R, const CONJ * S) {
}

// Atribui a um conjunto R a sua interseção com outro conjunto S.
// Ambos os conjuntos devem ter a mesma amplitude.
// O conjunto S não pode ser alterado.
static inline void manter(CONJ * R, const CONJ * S) {
}

// Atribui a um conjunto R a sua diferença para outro conjunto S.
// Ambos os conjuntos devem ter a mesma amplitude.
// O conjunto S não pode ser alterado.
static inline void suprimir(CONJ * R, const CONJ * S) {
}

// Atribui a um conjunto R a sua união com outro conjunto S.
// Ambos os conjuntos devem ter a mesma amplitude.
```

```c
// O conjunto S não pode ser alterado.
static inline void agregar(CONJ * R, const CONJ * S) {
}

// Tipo Nó utilizado para enumeração de CONJ
typedef struct {
} NO;

// Cria e retorna um novo nó cujo alvo vale -1.
static inline NO no(CONJ * R) {
}

// Retorna o alvo de um nó.
// O nó especificado não pode ser modificado.
static inline unsigned int alvo(const NO * r) {
    return r->a;
}

// Enumeração de conjuntos
static inline NO partida(CONJ * const R) {
}
static inline bool chegada(NO * r) {
}
static inline NO * avancar(NO * r) {
}
```

2.2 Grafos

Utilizando o tipo abstrato CONJ, o tipo abstrato para grafos é implementado conforme mostrado a seguir.

```c
// Tipo GRAFO
// A qtd de vértices não pode ser alterada.
typedef struct {
    CONJ * const viz;
    int nvert;
} GRAFO;

// Cria e retorna um grafo sem arestas com a qtd de vértices especificada.
GRAFO grafo(int n) {
    CONJ * a = (CONJ *) calloc(n, sizeof(CONJ));
    for (int i = 0; i < n; i++) {
        CONJ c = conjunto(n);
        memcpy(a+i, &c, sizeof(CONJ));
    }
    GRAFO g = {a, n};
    return g;
}

// Apaga um grafo da memória, liberando a memória por ele ocupada
void apagar_grafo(GRAFO * g) {
    for (int i = 0; i < g->nvert; i++)
        apagar(&g->viz[i]);
    free(g->viz);
```

```
    g->nvert = 0;
}

// Operações sobre grafos

// Retorna a qtd de vértices do grafo
static inline unsigned int amplitude(const GRAFO * g) {
    return g->nvert;
}

// Retorna o conjunto de vizinhos de um vértice e um grafo.
// O conjunto retornado não pode ser modificado.
static inline const CONJ * vizinhos(const GRAFO * g, unsigned int v) {
    return &g->viz[v];
}

// Inclui uma aresta definida pelos seus extremos em um grafo.
// Não deve ser usada com u=v
static inline void incluir_aresta(unsigned int u, unsigned int v, GRAFO * g) {
    incluir(v, &g->viz[u]);
    incluir(u, &g->viz[v]);
}

// Verifica se uma aresta definida pelos seus extremos está presente em um grafo.
static inline bool existe_aresta(unsigned int u, unsigned int v, GRAFO * g) {
    return pertence(v, vizinhos(g, u));
}

// Retorna qtd de arestas no grafo
unsigned long n_arestas(const GRAFO * g) {
    unsigned long m = 0;
    for (int i = 0; i < g->nvert; i++)
        m += cardinalidade(vizinhos(g, i));
    return m >> 1;
}
```

3 Estrutura de Dados

Para a realização dos experimentos, o tipo abstrato CONJ deve ser implementado com uma estrutura de dados. Recomenda-se utilizar mapa de bits com os parâmetros especificados a seguir.

```
#define MAX_VERTICES 8192
#define LOG_MAX_VERTICES 13
#define NBYTES_NACO 8       // tamanho do NACO: sizeof(unsigned long long) = 8
#define LOG_NBYTES_NACO 3
#define NBITS_NACO 64
#define LOG_NBITS_NACO 6

#define N_NACOS_(n) n >> LOG_NBITS_NACO
#define N_NACOS(n) (((N_NACOS_(n)) << LOG_NBITS_NACO) == n ? (N_NACOS_(n)) : ((N_NACOS_(n))+1))
#define IDX_NACO(v) (v >> LOG_NBITS_NACO)
#define IDX_EM_NACO(v) (v - (IDX_NACO(v) << LOG_NBITS_NACO))
```

```c
#define NACO_DE(a,v) a[IDX_NACO(v)]
```

Algumas das operações sobre conjuntos implementados com mapa de bits podem requerer manipulações de nacos. As funções a seguir podem ser úteis nesse contexto. Fontes: https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html e https://www.gnu.org/software/libc/manual/html_node/Copying-Strings-and-Arrays.html.

```c
#include <string.h>

// Returns the number of 1-bits in x.
int __builtin_popcountll (unsigned long long x);

// Returns one plus the index of the least significant 1-bit of x,
// or if x is zero, returns zero.
int __builtin_ffsll (long long x);

// The memset() function shall copy c (converted to an unsigned char)
// into each of the first n bytes of the object pointed to by s. The memset()
// function shall return s; no return value is reserved to indicate an error.
void *memset(void *s, int c, size_t n);

// The memcpy function copies size bytes from the object beginning at from
// into the object beginning at to. The behavior of this function is undefined
// if the two arrays to and from overlap; use memmove instead if overlapping
// is possible. The value returned by memcpy is the value of to.
void * memcpy (void *restrict to, const void *restrict from, size_t size);
```

4 Experimentos a Realizar

O objetivo dos experimentos é estimar o comportamento do tempo de execução de algoritmos para resolver os problemas **Verificação de Clique** e **Percurso em Largura** quando implementados com o tipo abstrato `GRAFO`. As funções que resolvem esses problemas são apresentadas a seguir.

```c
// Função para verificar se um subconjunto define uma clique no grafo
bool verifica_clique(const GRAFO * g, const CONJ * subset) {
    CONJ R = replicar(subset);
    NO r = partida(&R);
    while (!chegada(&r)) {
        retirar_do_conjunto(&r);
        if (!subconj(&R, vizinhos(g, alvo(&r)))) {
            apagar(&R);
            return false;
        }
        avancar(&r);
    }
    apagar(&R);
    return true;
}

// Função para percorrer um grafo com percurso em largura
void percurso_largura(const GRAFO * g, unsigned int r, CONJ * A, CONJ * B) {
    unsigned int F[amplitude(g)];
    unsigned int i = 0,f = 0;
```

```
    esvaziar(A);
    esvaziar(B);

    F[f++] = r;
    incluir(r, A);

    while (i < f) {
        unsigned int u = F[i++];
        copiar(vizinhos(g, u), B);
        suprimir(B, A);
        agregar(A, B);

        NO w = partida(B);
        while (!chegada(&w)) {
            F[f++] = alvo(&w);
            avancar(&w);
        }
    }
}
```

Os tempos obtidos em um experimento com uma implementação com mapa de bits são apresentados no gráfico da Figura 1.

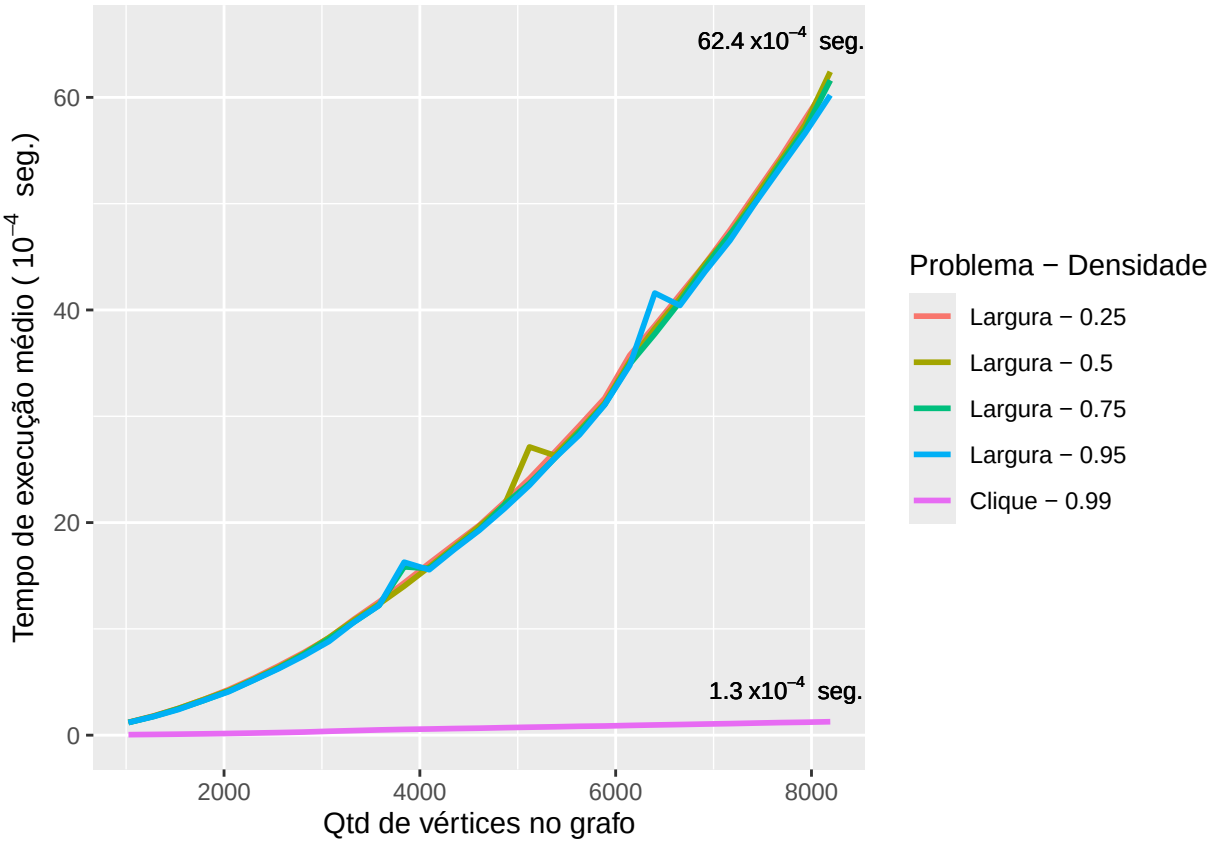

Figura 1: Tempos médios de execução com 10 grafos aleatórios para cada quantidade de vértices e cada densidade.

4.1 Etapas na realização do trabalho

1. O material a utilizar nos experimentos está disponível no SIGAA como arquivo compactado anexo à tarefa. Ao descompactar o arquivo anexo à tarefa, a seguinte estrutura de arquivos é criada:

- *clique_completo.c:* programa para os experimentos com o problema **Verificação de Clique**
- *clique_largura.pdf*: este arquivo de descrição da tarefa
- *graph.h*: tipos abstratos utilizados
- *largura.c*: programa para os experimentos com o problema **Percurso em Largura**
- *result.csv:* exemplo de planilha que organiza os tempos obtidos e pode ser usada para gerar um gráfico

2. Implementar `CONJ` e `NO`, e as respectivas funções, do arquivo *graph.h*.

3. Compilar cada um dos programas de experimentos da seguinte forma:

- *gcc largura.c -o largura*
- *gcc clique_completo.c -o clique*

4. Executar *clique* e *largura*.

5. Imprimir um gráfico com os tempos de execução obtidos seguindo o modelo do gráfico da Figura 1.

6. Criar um arquivo compactado contendo *graph.h* modificado com a sua implementação de `CONJ` e o gráfico dos tempos de execução alcançados nos experimentos.

ATENÇÃO: prever pelo menos 4h de execução dos programas.

4.2 Nota Competitiva

Todos os códigos entregues serão reexecutados no mesmo computador no qual os tempos no gráfico da Figura 1 foram obtidos.

A nota da tarefa dependerá do desempenho do programa fornecido, tanto a correção quanto o tempo de execução. Quanto mais efeciente a implementação, maior será a nota. Os que conseguirem tempos melhores que a implementação que foi usada para produzir os resultados do gráfico da Figura 1 terá um bônus de nota.