Notas de Aula - AED1 – Matrizes Esparsas
Prof. Jefferson T. Oliva

Matriz (bidimensional)
- Arranjo bidimensional: composto por m linhas e n colunas
- Utilizada para representação de diversos tipos de dados
→ Dados numéricos
→ Imagens
→ etc

Representação de matrizes por alocação estática:

```
int mat[4][4];
mat[0][0] = 0;
mat[0][1] = 3;
mat[0][2] = 0;
mat[0][3] = 1;
mat[1][0] = 0;
mat[1][1] = 0;
...
mat[3][3] = 0;
```

ou

```
int mat[4][4] = {{0, 3, 0, 1}, {0, 0, 5, 0}, {0, 0, 0, 0}, {2, 7, 1, 0}};
```

Representação de matrizes por alocação dinâmica:

```
int i;
int **mat = malloc(4 * sizeof(int*));

for (i = 0; i < n; i++)
    mat[i] = malloc(4 * sizeof(int));
```

Além da representação de uma matriz poder ser custosa, na mesma podem haver elementos nulos (com valor irrelevante, e.g. zero).

Matriz Esparsa

Matriz em que a maioria dos elementos possui um valor padrão (0, por exemplo), ou a maioria dos valores são faltantes
- Por exemplo: representar o contorno de uma imagem em preto e branco

Nos exemplos do slide 7 seria um desperdício gastar m × n posições na memória para representá-los: apenas uma pequena quantidade dos elementos tem um valor diferente de zero (preto).

Matrizes esparsas podem para evitar o desperdício de recursos computacionais (espaço e tempo para processamento): apenas elementos com valores diferentes de zero serão alocados.

Matrizes esparsas são aplicados em diversos problemas de:
- Método das malhas para a resolução de circuitos elétricos
- Sistemas de equações lineares
- Armazenamento de dados (e.g. planilhas eletrônicas, mapas de bits)
- Etc

Exemplos nos slides 10 e 11

Para representação de grafos, a matriz de adjacência pode acarretar em desperdício de memória.

Nesse caso, a lista de adjacência pode ser vantajosa para a representação de grafos "esparsos".

A representação de matrizes esparsas podem ser representadas por meio de um vetor de listas encadeadas.
- Cada elemento desse vetor representaria uma linha da matriz
- Em grafos, a lista de adjacência é similar à matriz esparsa

Implementação de Matrizes Esparsas

Ver exemplos nos slides de 14 a 18

Cada célula (Cell) contém:
- A informação (item)
- A coluna da matriz (col)
- O ponteiro para a próxima célula (next)

typedef struct Cell Cell;

```
struct Cell{
    int item;
    int col;
    Cell *next;
};
```

Representação de uma matriz esparsa:
- Quantidade de linhas (n_lin)
- Quantidade de colunas (n_col)
- Arranjo de ponteiros do tipo Cell (lin)

```
typedef struct{
      Cell *head;
}ListaE;


typedef struct Spa_Mat{
      int n_lin;
      int n_col;
      ListaE **lin;
};
```

Operações básicas em uma matriz esparsa
- Criar uma matriz esparsa
- Buscar item
- "Alterar" item
- Imprimir matriz

A operação de busca é a mesma aplicada em listas encadeadas, mas ela pode ser na matriz inteira (ou seja, nas n_lin listas)

A inserção é feita em ordem crescente e de acordo com o índice da coluna, considerando os seguintes casos:
- Caso exista uma célula na posição e o valor a ser inserido for igual a zero, então a célula deve ser removida
- Caso exista uma célula na posição e o valor a ser inserido for diferente de zero, então o valor existente deve ser substituído
- Caso não exista uma célula na posição e o valor a ser inserido for diferente de zero, então uma nova célula é criada
- Caso não exista uma célula na posição e o valor a ser inserido for igual a zero, então não é necessário fazer algo

No caso de uma matriz esparsa, a operação de inserção e remoção é a mesma: depende do valor a ser inserido.

Para a impressão da matriz, para as "células inexistentes" é impresso 0 e para das demais células, é impresso os seus respectivos valores

Implementação da operação criar:

```
Spa_Mat* criar(int l, int c){
    Spa_Mat* mat = (Spa_Mat*) malloc(sizeof(Spa_Mat));
    int i;

    mat->n_col = c;
    mat->n_lin = l;
    mat->lin = (ListaE**) malloc(sizeof(ListaE*) * l);

    for (i = 0; i < l; i++){
        mat->lin[i] = (ListaE*) malloc(sizeof(ListaE));
        mat->lin[i]->head = NULL;
    }

    return mat;
}
```

Implementação da operação buscar:

```
static int procurar_lista(int item, ListaE *l){
    Cell *aux;

    if (l != NULL){
        aux = l->head;
        while ((aux != NULL) && (aux->item < item))
        aux = aux->next;
    }

    if ((aux != NULL) && (aux->item == item))
        return 1;
    else
        return 0;
}

int buscar(int item, Spa_Mat* mat){
    int i;
    int aux = 0;

    for (i = 0; (i < mat->n_lin) && (aux == 0); i++)
        aux = procurar_lista(item, mat→lin[i]);

    return aux;
}
```

Implementação da operação inserir/remover

```
static int validar_pos_matriz(int lin, int col, Spa_Mat* mat){
   return (lin >= 0) && (lin < mat->n_lin) && (col >= 0) && (col < mat->n_col);
}


// Esta função só é chamada quando item > 0
// Caso a célula com coluna col exista, basta mudar o seu respectivo valor. Caso contrário,
// deve ser criada uma nova célula
static void trocar_inserir_na_lista(int item, int col, ListaE *l){
   Cell *auxA, *auxP, *novo;

   // Verificar se a lista da matriz tem a cabeça vazia ou a coluna em que a coluna em que
   // haverá inserção é menor em comparação com a da primeira célula da lista.
   // Caso esse teste retornar verdadeiro, a célula será inserida na primeira posição da lista.
   if ((l->head == NULL) || (col < l->head->col)){
      novo = criar_celula(item, col);
      novo->next = l->head;
      l->head = novo;
   // Se a lista não estiver vazia, é verificado se o seu primeiro elemento tem a coluna igual
   // a col. Caso positivo, apenas o valor da célula é trocado
   }else if ((l->head != NULL) && (col == l->head->col)){
      l->head->item = item;
   }else{
      auxA = l->head; // a nova célula deverá ser colocada entre auxP e auxA
      auxP = auxA;

      // Encontrar uma posição adequada para inserção/remoção
      while ((auxP != NULL) && (auxP->col < col)){
         auxA = auxP;
         auxP = auxP->next;
      }

      // Se a coluna col existir, basta trocar o valor de item
      if ((auxP != NULL) && (col == auxP->col))
         auxP->item = item;
      // Caso contrário, uma nova célula deve ser criada
      else{
         novo = criar_celula(item, col);
         novo->next = auxA->next; // auxP
         auxA->next = novo;
      }
   }
}
```

```
// Esta função só é chamada quando item <= 0
// Caso a célula com coluna col exista, basta remover a céluna. Caso contrário, nada é feito.
static void remover_na_lista(int col, ListaE *l){
    Cell *auxA, *auxP = NULL;

    // Verificar se a célula que deverá ser removida é o primeiro elemento da lista.
    if (l->head != NULL){
        if (col == l->head->col){
            auxP = l->head;
            l->head = l->head->next;
            free(auxP);
        }else{
            auxA = l->head;
            auxP = auxA; // Se for feita a remoção, auxP será removida e auxA->next = auxP->next

            // Encontrar uma posição adequada para inserção/remoção
            while ((auxP != NULL) && (auxP->col < col)){
                auxA = auxP;
                auxP = auxP->next;
            }

            if ((auxP != NULL) && (col == auxP->col)){
                auxA->next = auxP->next;

                free(auxP);
            }
        }
    }
}


// Se item for <= 0, uma célula poderá ser removida. Caso contrário, poderá ocorrer um dos seguintes
// cenários:
// 1 - Se a coluna existir na l-ésima lista, a sua respectiva célula terá o seu valor alterado.
// 2 - Caso contrário, uma nova célula deverá ser criada e inserida de forma ordenada (por coluna)
//     na l-ésima lista
void trocar(int item, int l, int c, Spa_Mat* mat){
    if (validar_pos_matriz(l, c, mat)){
        if (item > 0)
            trocar_inserir_na_lista(item, c, mat->lin[l]);
        else
            remover_na_lista(c, mat->lin[l]);
    }
}
```

Implementação da operação imprimir:

```
void imprimir(Spa_Mat* mat){
	int i, j;
	Cell* aux;

	for (i = 0; i < mat->n_lin; i++){
		aux = mat->lin[i]->head;
		j = 0;

		while (aux != NULL){
			while (j < aux->col){
				printf("0 ");
				j++;
			}

			printf("%d ", aux->item);
			aux = aux->next;
		}

		for (j; j < mat->n_col; j++)
			printf("0 ");

		printf("\n");
	}
}
```

Complexidade
- Para criar uma matriz esparsa vazia, a complexidade é de O(m), onde m é quantidade de linhas
- Para criar uma matriz esparsa não vazia, a complexidade é de O(m * n), onde n é quantidade de colunas
- Para buscar/inserir/remover um item, a complexidade é de O(n), ou seja, cada operação é proporcional a quantidade de itens guardados em cada linha
→ Dependendo da implementação da operação de busca, a complexidade pode ser na ordem de O(m * n)
- Para imprimir, a complexidade é de O(m * n)

Vantagens
- Mantém a característica bidimensional da imagem (para a forma de representação desse tipo de matriz apresentada até o momento)
- Economia de memória

Desvantagem: acesso sequencial

Outras formas de representação: ver slides 29 e 30

Referências

Oliva, J. T. Matrizes Esparsas. AE22CP - Algoritmos e Estrutura de Dados I. Notas de Aula. Engenharia de Computação. Dainf/UTFPR/Pato Branco, 2020.

Roman, N. T.; Digiampietri, L. A. Matriz Esparsa. ACH2023 - Algoritmos e Estrutura de Dados I. Notas de Aula. Sistemas de Informação. EACH/USP/São Paulo, 2018.