Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC2133 - Estructura de Datos y Algoritmos

Informe Tarea 0

22 de junio de 2024
Manuel Espinoza Quintero
20642598

Motivación

Esta tarea es una gran oportunidad para poner en práctica mis conocimientos de estructuras de datos y algoritmos. Cada problema me desafía a encontrar soluciones eficientes y mejorar mis habilidades de programación, para aplciarlas en problemas del mundo real.

Informe

Problema 1: Cerebros Codiciosos

Algoritmo: Para resolver este problema, use un enfoque de programación dinámica. Primero, ordene las posiciones de los planetas. Luego, use una estrategia de barrido para colocar las naves de manera que se minimice su número.

Código: El código para resolver este problema se implementó en C, utilizando una estructura de datos de matriz para mantener las posiciones de los planetas. A continuación se muestra un fragmento del código:

```
int minCerebros(int planetas[], int n, int R) {
    qsort(planetas, n, sizeof(int), compare);
    int cerebros = 0, i = 0;
    while (i < n) {
        cerebros++;
        int loc = planetas[i] + R;
        while (i < n && planetas[i] <= loc) i++;
        loc = planetas[--i] + R;
        while (i < n && planetas[i] <= loc) i++;
    }
    return cerebros;
}
```

Complejidad: La complejidad temporal del algoritmo es $O(N \log N)$ debido a la ordenación inicial, seguida de $O(N)$ para el barrido, dando un total de $O(N \log N)$. La complejidad espacial es $O(1)$ adicional, ya que solo utilizamos un número constante de variables adicionales.

Problema 2: Acumulando Basura

Este problema se aborda mediante programación dinámica similar al problema de la "mochila". Utilizamos una tabla para almacenar la cantidad mínima de basurales necesarios para cada valor de masa posible hasta M.

Código: El código se implementó en C, usando un array para la tabla de programación dinámica:

```
int minBasurales(int basurales[], int n, int M) {
    int INF = M + 1;
    int *dp = (int *)malloc((M + 1) * sizeof(int));
```

```
    for (int i = 0; i <= M; i++) dp[i] = INF;
    dp[0] = 0;
    for (int i = 0; i < n; i++) {
        for (int j = basurales[i]; j <= M; j++) {
            if (dp[j - basurales[i]] != INF) {
                dp[j] = dp[j] < dp[j - basurales[i]] + 1 ?
                dp[j] : dp[j - basurales[i]] + 1;
            }
        }
    }
    int result = dp[M] == INF ? -1 : dp[M];
    free(dp);
    return result;
}
```

Complejidad: La complejidad temporal del algoritmo es $O(N \times M)$, donde N es el número de tipos de basurales y M es la masa objetivo. La complejidad espacial también es $O(M)$ debido al uso del array dp.

Problema 3: Identificación de Regiones

Algoritmo: Para este problema, utilizamos el algoritmo de Búsqueda en Profundidad (DFS). Mantenemos un array de nodos visitados y contamos cuántas veces iniciamos una nueva búsqueda DFS, lo cual indica un nuevo componente conectado.

Código: El código se implementó en C, usando una matriz de adyacencia para representar el grafo:

```
void DFS(int u, bool *visitado, int **grafo, int N) {
    visitado[u] = true;
    for (int v = 0; v < N; v++) {
        if (grafo[u][v] && !visitado[v]) {
            DFS(v, visitado, grafo, N);
        }
    }
}

int main(int argc, char **argv) {
    check_arguments(argc, argv);
    FILE *input_file = fopen(argv[1], "r");
    FILE *output_file = fopen(argv[2], "w");
    int NODOS_GRAFO, NUM_ARISTAS;
    fscanf(input_file, "%d %d", &NODOS_GRAFO, &NUM_ARISTAS);
    int **grafo = (int **)malloc(NODOS_GRAFO * sizeof(int *));
    for (int i = 0; i < NODOS_GRAFO; i++) grafo[i] = (int *)calloc(NODOS_GRAFO, sizeof(i
    for (int q = 0; q < NUM_ARISTAS; q++) {
        int nodo1, nodo2;
        fscanf(input_file, "%d %d", &nodo1, &nodo2);
        grafo[nodo1][nodo2] = grafo[nodo2][nodo1] = 1;
    }
    bool *visitado = (bool *)calloc(NODOS_GRAFO, sizeof(bool));
    int componentes_conectados = 0;
    for (int i = 0; i < NODOS_GRAFO; i++) {
        if (!visitado[i]) {
            DFS(i, visitado, grafo, NODOS_GRAFO);
            componentes_conectados++;
        }
    }
    fprintf(output_file, "%d\n", componentes_conectados);
    for (int i = 0; i < NODOS_GRAFO; i++) free(grafo[i]);
    free(grafo);
```

```
    free(visitado);
    fclose(input_file);
    fclose(output_file);
    return 0;
}
```

Complejidad: La complejidad temporal del algoritmo DFS es $O(N + A)$, donde N es el número de nodos y A es el número de aristas. La complejidad espacial es $O(N^2)$ debido al uso de la matriz de adyacencia.

Conclusión

En este informe, se abordaron tres problemas diferentes utilizando técnicas y estructuras de datos adecuadas. Se demuestra cómo los algoritmos de programación dinámica y búsqueda en grafos pueden ser aplicados para resolver problemas complejos de manera eficiente. Las soluciones presentadas no solo son óptimas en términos de complejidad temporal y espacial, sino que también son fácilmente implementables en lenguajes de programación como C, como se vio en clases.

Referencias

[1] GeeksforGeeks, *Comparator function of qsort() in C*, [online] Available: `https://www.geeksforgeeks.org/comparator-function-of-qsort-in-c/`.