Tecnológico de Monterrey

Evidencia 2

TC2038: Análisis y diseño de algoritmos avanzados
Grupo 601

Alejandro Arouesty
Tec de Monterrey,
Campus Santa Fe
a01782691@tec.mx

Andrés Tarazona
Tec de Monterrey,
Campus Santa Fe
a01023332@tec.mx

Joaquín Badillo
Tec de Monterrey,
Campus Santa Fe
a01026364@tec.mx

Bajo la instrucción de
Víctor de la Cueva

30 de Noviembre de 2023

1 Reflexión

1.1 Árbol de Expansión Mínima (MST)

```cpp
void mst(const utils::AdjMatrix& adj_matrix) {
  MinHeap pq;
  std::unordered_set<int> permanent;
  std::vector<Edge> mst;

  pq.push({
    .id = 0,
    .tag = 0,
    .weight = 0
  });

  while (!pq.empty() && mst.size() < adj_matrix.size() - 1) {
    Edge current = pq.top();
    pq.pop();

    if (permanent.find(current.id) != permanent.end())
      continue;

    if (current.id != current.tag)
      mst.push_back(current);
      permanent.insert(current.id);

    for (int i = 0; i < adj_matrix.size(); i++) {
      if (i == current.id || permanent.find(i) != permanent.end())
        continue;

      pq.push({
        .id = i,
        .tag = current.id,
        .weight = adj_matrix[current.id][i]
      });
    }
  }

  for (auto edge : mst)
    std::cout << "("  << edge.tag << ", "  << edge.id  << ")\t";

  std::cout << std::endl;
}
```

Implementación 1: Algoritmo de Prim

El algoritmo de Prim, el cual fue implementado como se muestra en la Implementación 1 nos permite encontrar el árbol de expansión mínima de un grafo no dirigido y ponderado. Es decir, el conjunto de aristas que conectan todos los vértices del grafo con el menor costo posible. Este algoritmo es muy similar al algoritmo de Dijkstra, con una variación en la función de costos, mientras que Dijkstra acumula el costo (puesto que busca un camino), el algoritmo de Prim solo utiliza el costo de la arista actual.

El algoritmo de Prim es un algoritmo voraz, es decir, en cada iteración toma la mejor decisión local y si los costos son positivos, el algoritmo siempre encontrará el árbol de expansión mínima.

Como se estaba buscando una forma de conectar múltiples centrales, de tal manera que existiera un camino entre cualesquiera dos centrales y minimizar el uso de fibra óptica, el algoritmo de Prim resulta ser eficiente y correcto, dado que las distancias son estrictamente positivas.

1.1.1 Complejidad Computacional

La complejidad computacional del algoritmo de Prim es de $\mathcal{O}(E \log V)$, donde E es el número de aristas y V es el número de vértices. Esto se debe a que el algoritmo utiliza un min-heap para obtener la arista de menor costo en cada iteración.

Como la inserción de un elemento a un min-heap de tamaño N tiene una complejidad en timepo $\mathcal{O}(\log N)$, al igual que para extraer el mínimo. En el peor escenario se puede considerar un min-heap que contiene todas las aristas del grafo, que como sugiere la matriz de adyacencia son $\mathcal{O}(V^2)$ aristas. Por lo tanto, la complejidad computacional del algoritmo de Prim es

$$\mathcal{O}(E \log V^2) = \mathcal{O}(2E \log V) = \mathcal{O}(E \log V) \quad \blacksquare. \tag{1}$$

1.2 Problema del Agente Viajero (TSP)

El problema del agente viajero es un problema de optimización que consiste en encontrar un camino que recorra todos los vértices de un nodo y regrese al inicio, de tal manera que el costo total del camino sea mínimo. Se ha demostrado que este problema es NP-Díficl, es decir que es **al menos** tan difícil como cualquier otro problema en NP. La implicación que es de nuestro interés de tal demostración es que no se conoce un algoritmo para una máquina determinística que lo resuelva en tiempo polinomial (además de que su existencia es poco probable) y por lo tanto para grafos suficientemente grandes es virtualmente insoluble.

Debido a este contexto, si es de vital importancia resolver el problema del agente viajero de forma exacta, se debe recurrir a los algoritmos de búsqueda y aunque su orden en el peor escenario sea al menos exponencial, el uso de heurísticas puede reducir el tiempo de ejecución promedio considerablemente. Por esta razón, en la Implementación 2 se utilizó la técnica de *Branch and Bound* para reducir el espacio de búsqueda.

```
1  void tsp(utils::AdjMatrix adj_matrix, int start) {
2    utils::AdjMatrix initial = adj_matrix;
3
4    for (int i = 0; i < initial.size(); i++) {
5        initial[i][i] = INT_MAX;
6    }
7
8    int best = INT_MAX;
9
10   Element bestElement = {
11       .node = -1,
12       .cost = INT_MAX,
13       .level = -1,
14       .reduced = initial,
15       .path = {}
16   };
17
18   std::pair<utils::AdjMatrix, int> reduced = initialReducer(initial, 0);
19   std::priority_queue<Element, std::vector<Element>, compare> pq;
```

```
    pq.push({
        .node = start,
        .cost = reduced.second,
        .level = 0,
        .reduced = reduced.first,
        .path = {start}
    });

    while(!pq.empty() && pq.top().cost < best) {
        Element current = pq.top();
        pq.pop();

        if (current.level == adj_matrix.size() - 1) {
            best = std::min(best, current.cost + adj_matrix[current.node][start])
                ;
            bestElement = current;
            continue;
        }

        for (int i = 0; i < current.reduced.size(); i++) {
            if (i == current.node)
                continue;

            int min = INT_MAX;
            for (int j = 0; j < current.reduced[0].size(); j++) {
                if (current.reduced[i][j] >= INT_MAX)
                    continue;

                min = std::min(min, current.reduced[i][j]);
            }

            if (min < INT_MAX) {
                std::pair<utils::AdjMatrix, int> reduced = reducer(current.
                    reduced, i, current.node, current.cost);
                std::vector<int> path = current.path;
                path.push_back(i);

                pq.push({
                    .node = i,
                    .cost = reduced.second,
                    .level = current.level + 1,
                    .reduced = reduced.first,
                    .path = path
                });
            }
        }
    }

    int length = 0;
    std::cout << "Path: ";

    int prev = -1;
    for (auto node : bestElement.path) {
        if (prev != -1)
            length += adj_matrix[prev][node];
```

```
73
74        prev = node;
75        std::cout << (char)(node + 65) << " ";
76    }
77
78    std::cout << (char)(start + 65) << std::endl;
79
80    length += adj_matrix[prev][start];
81
82
83    std::cout << "Length: " << length << std::endl;
84
85 }
```

Implementación 2: Branch and Bound para TSP

1.2.1 Complejidad Computacional

En el peor escenario, se puede considerar que en la cola de prioridad están todos los nodos del árbol de búsqueda. Como este árbol en cada nivel reduce en 1 elemento la cantidad de hijos, por el principio de conteo se deduce que tiene $n!$ nodos. Por lo tanto, asíntóticamente el ciclo puede correr $\mathcal{O}(n!)$ veces. En cada una de estas interaciones se calcula la heurísitca reduciendo una matriz, lo cual tiene una complejidad $\mathcal{O}(n^2)$ dado que se tiene una matriz de adyacencia y se obtiene un elemento del heap lo cual tiene complejidad $\lg(n!)$.

Por lo tanto, aunque branch and bound en promedio tendrá una ejecución muy rápdia, en el peor escenario su complejidad será

$$\mathcal{O}\left(\lg(n!) + n^2\right) \cdot n! \tag{2}$$

Por lo que se mencionó en clase, resulta interesante utilizar una máscara de bits y programación dinámica en una futura implementación.

1.3 Distancia Más Cercana

El último problema que se abordó fue el de encontrar la ubicación de la central más cercana para una instalación nueva. El problema de asociar puntos (o vectores) a partir de su distancia es de hecho de gran interés para la computación moderna, ya que las emergentes tendencias por la adoptación de la inteligencia artificial y el aprendizaje automático se han visto beneficiados por la indexación de datos de acuerdo con su cercanía en un espacio vectorial, como se puede observar con las bases de datos de vectores (*embeddings*).

La búsqueda por cercanía en un contexto geográfico también resulta interesante en servicios de mapas y en el *marketing* ya que es mucho más probable que para una persona, un producto o servicio sea más atractivo si se encuentra cerca de su ubicación. De hecho aunque en esta situación se utilizó la norma L^2 (o distancia euclideana) para determinar la cercanía entre dos puntos, si consideramos las ubicaciones en la Tierra, dicha norma podría implicar atravesar la superficie terrestre. Por lo tanto si se utiliza la longitud y latitud como sistema coordenado (y se considera la Tierra como una esfera ideal), la distancia de nuestro interés es en realidad la de un segmento de un círculo máximo, dado que esta es la geodésica en una esfera.

```
1 Point findClosestPair(const std::vector<Point>& points) {
2   std::string x_str;
```

```
  std::cout << "Choose the x coordinate: ";
  std::cin >> x_str;

  std::string y_str;
  std::cout << "Choose the y coordinate: ";
  std::cin >> y_str;

  Point point = std::make_pair(std::stod(x_str), std::stod(y_str));


  double minDistance = std::numeric_limits<double>::max();
  std::pair<double, double> closestPair = {-1.0, -1.0};

  for (int i = 0; i < points.size(); ++i) {
      double distance = euclideanDistanceSquared(points[i].first, points[i].
         second, point.first, point.second);
      if (distance < minDistance) {
          minDistance = distance;
          closestPair = points[i];
      }
  }

  return closestPair;
}
```

Implementación 3: Ubicación más cercana

1.3.1 Complejidad Computacional

Para ahorrar el trabajo computacional de la raíz cuadrada, se uso la norma euclideana al cuadrado ya que es monotónicamente creciente y $d_1(x,y)^2 >= d_2(x,y)^2 \implies d_1(x,y) >= d_2(x,y)$ y por lo tanto es una métrica igual de válida. Por lo tanto la complejidad de esta función se reduce a hacer una búsqueda lineal en el arreglo de puntos, lo cual tiene una complejidad.

$$\mathcal{O}(n) \tag{3}$$