Práctica Final - Ramificación y poda

Miguel Ferrer Castellá

Junio 2024

Índice

1. Introducción

En este informe se va a proporcionar un análisis detallado de la implementación de un algoritmo que utiliza el método de ramificación y poda. Este algoritmo busca solucionar el problema del camino de mínimo coste. La eficiencia del mismo se ha incrementado mediante el uso de cotas, tanto positivas como negativas, que también se han utilizado para reducir el espacio de búsqueda.

El enfoque de este trabajo está en la base teórica, la estructura de datos utilizada y los resultados obtenidos a través de diversas pruebas y evaluaciones. El objetivo es demostrar que el método de ramificación y poda es un enfoque útil y eficiente para la resolución de problemas complejos, tanto teórica como computacionalmente. Se prevé que la investigación establezca una base sólida para futuras mejoras en aplicaciones en optimización algorítmica.

2. Descripción del Algoritmo

2.1. Estructura del Nodo

Cada nodo en el árbol de búsqueda contiene la siguiente información:

- **x, y**: Coordenadas actuales en la matriz.
- **cost**: Costo acumulado hasta el nodo actual.
- **optimisticCost**: Estimación del costo mínimo para llegar al destino desde el nodo.
- **path**: Vector que almacena el camino recorrido hasta el nodo actual.

```
struct Node {
    int x, y; // Coordenadas del nodo.
    int cost; // Costo acumulado para llegar al nodo.
    int optimisticCost; // Costo optimista estimado desde el nodo
    hasta el destino.
    vector<pair<int, int>> path; // Camino recorrido hasta el nodo.

    // Sobrecarga del operador < para comparar nodos basándose en el
    costo optimista.
    bool operator<(const Node& other) const {
        return optimisticCost > other.optimisticCost;
    }
};
```

2.2. Cota Optimista

La cota optimista se calcula utilizando la distancia Chebysov, que estima el costo mínimo para llegar desde la posición actual hasta la meta.

```
int optimistic(int x1, int y1, int x2, int y2) {
    return max(abs(x2 - x1), abs(y2 - y1));
}
```

Se estudió la posibilidad de usar la distancia de Manhattan, pero resultó menos eficiente que la distancia de Chebysov, ya que esta última proporciona la cantidad mínima de celdas que se necesitan visitar para alcanzar la casilla final.

2.3. Cota Pesimista

En una primera instancia se intentaron utilizar dos cotas pesimistas en el algoritmo mediante las funciones Mpresimistic() y mcp_greedy():

- **Mpresimistic()**: Construye una matriz de pesos con el coste del camino de mínimo peso calculado para la versión restringida del problema desde cada nodo hasta el nodo final, es decir el nodo de posición (n-1, m-1).
- **mcp_it_vector()**: Devuelve una primera estimación del valor del camino de mínimo coste hallado con la iterativa con un vector como almacén de la versión restringida, elaborada en la práctica 6.2.

2.3.1. Mpesimistic

Esta función impidió el cálculo posterior de los caminos, ya que podaba muchas ramas impidiendo almacenarlas de manera correcta en la tupla de nodos visitados. En principio se quería utilizar una única vez en la función principal del programa previo a la ejecución del algoritmo mcp_bb(), para calcular una cota pesimista para cada nodo y saber si era prometedor de una manera muy eficiente.

El cálculo de esta matriz es la que conseguía eliminar nodos gracias a la cota pesimista, como no se ha sabido implementar correctamente, esta estadística no se contabiliza en el algoritmo, ya que no se producen podas por esta regla.

```
void Mpesimistic(vector<vector<int>> &M, vector<vector<int>> &mat,
                 int n, int m) {
    // Inicializar la última celda con 0 ya que no hay costo para permanecer
    en la meta.
    M[n][m] = 0;

    // Inicializar el borde derecho y el borde inferior de la matriz.
    for (int i = n - 1; i >= 0; --i) {
        M[i][m] = mat[i][m] + M[i + 1][m];
    }
    for (int j = m - 1; j >= 0; --j) {
        M[n][j] = mat[n][j] + M[n][j + 1];
    }

    // Calcular el camino de menor dificultad para cada elemento de la matriz.
    for (int i = n - 1; i >= 0; --i) {
        for (int j = m - 1; j >= 0; --j) {
            int right = M[i][j + 1] + mat[i][j + 1];
            int down = M[i + 1][j] + mat[i + 1][j];
            int diag = M[i + 1][j + 1] + mat[i + 1][j + 1];
            M[i][j] = min({right, down, diag});
        }
    }
}
```

2.3.2. mcp_it_vector

Es la función para el cálculo de un valor pesimista inicial que consigue podar muchas ramas por no ser prometedoras, debido a que la cota optimista del nodo no es tan buena como la solución inicial que obtiene el algoritmo iterativo, que además es computacionalmente muy eficiente.

Se barajó la posibilidad de realizar la misma función con el algoritmo voraz realizado en la práctica 7 pero su eficiencia empeora la del algoritmo iterativo.

```
int mcp_it_vector(int n,int m, vector<vector<int>> &mat){
    vector<int> a(m,SENTINEL);
    return mcp_it_vector(a,n,m,mat);
}

int mcp_it_vector(vector<int> &a, int n,int m, vector<vector<int>> &mat){

    a[0] = mat[0][0];

    for (int j = 1; j <m; ++j){
        a[j] = a[j - 1] + mat[0][j];
    }

    for (int i = 1; i < n; ++i){
        int diagonal = a[0];
        a[0] += mat[i][0];

        for (int j = 1; j < m; ++j) {
            int temp = a[j];

            a[j] = mat[i][j] + min(a[j],min(a[j-1],diagonal));

            diagonal = temp;
        }
    }

    return a[m-1];
}
```

2.4. Algoritmo de Búsqueda Ramificación y Poda

El algoritmo elaborado con el método de ramificación y poda utiliza las cotas optimista y pesimista mencionadas anteriormente para podar ramas que nacen de nodos no prometedores. Utiliza una cola de prioridad basándose en el valor de la cota optimista para la ordenación de búsqueda de nodos prometedores.

Podará ramas si el nodo actual no es prometedor por ser su cota optimista mayor que la mejor solución hasta el momento, y si no es así, se estudian las 8 diferentes posibilidades de movimiento estudiando los respectivos nodos a los que se accederá siguiendo la misma lógica anterior para descartarlo y almacenarlo en caso de que sea prometedor para estudiarlo en un futuro como una posibilidad real para construir el camino.

Además, en la misma función se calcula al mismo tiempo el camino seguido guardado en la variable best_path para optimizar el tiempo de cálculo si se quiere imprimir uno de los dos caminos, esto perjudica las complejidades temporal y espacial de la función en el caso de que los caminos no quieran ser visualizados, pero se ha desarrollado esta solución pensando que la finalidad es mostrar el correcto funcionamiento de todas las opciones del programa, mejorando por lo tanto la eficiencia de este.

La función mcp_bb() devuelve una tupla con el mínimo peso, tupla con las posiciones visitadas en el camino y las estadísticas para ver la cantidad de nodos visitados, explorados, nodos hoja, nodos descartados por diferentes motivos, etc. que se solicitaba en el enunciado.

```
tuple<int, vector<pair<int, int>>, vector<int>> mcp_bb(int n, int m,
vector<vector<int>> &grid) {
    priority_queue<Node> pq;
    Node startNode = {0, 0, grid[0][0],
    grid[0][0] + optimistic(0, 0, n - 1, m - 1), {{0,0}}};
    pq.emplace(startNode);

    vector<vector<int>> cost(n, vector<int>(m, INT_MAX));
    cost[0][0] = grid[0][0];

    // Calcular cota pesimista inicial
    int best_cost = mcp_it_vector(n, m, grid)+1; // Sumamos 1 para asegurar
    que encuentra un nodo prometedor y realizar bien el camino

    while (!pq.empty()) {
        Node current = pq.top();
        pq.pop();
        ++nvisit;

        // Si el costo optimista del nodo actual es mayor o
        igual al mejor costo conocido, se descarta.
        if (current.optimisticCost >= best_cost) {
            ++npromising_but_discarded;
            continue;
        }

        // Si se llega a la meta, se actualiza el mejor camino si es necesario.
        if (current.x == n - 1 && current.y == m - 1) {
            nleaf++;
            if(current.cost < best_cost){
                best_path = current.path;
                best_cost = current.cost;
                cost[current.x][current.y] = current.cost;
                ++nbest_solution_updated_from_leafs;
            }
            continue;
        }

        // Expansión de los nodos vecinos.
        for (int i = 0; i < 8; ++i) {
            int nx = current.x + dx[i];
            int ny = current.y + dy[i];

            if (isValid(nx, ny, n, m)) {
                nvisit++;

                int new_cost = current.cost + grid[nx][ny];
                int optimisticCost = new_cost + optimistic(nx, ny, n - 1, m - 1);

                // Si el costo optimista es mayor o igual al mejor costo conocido,
                se descarta.
                if(optimisticCost >= best_cost){
                    nnot_promising++;
                    continue;
                }
```

```
                // Si se encuentra un camino mejor, se actualizan los costos y se
                encola el nuevo nodo.
                if (optimisticCost < best_cost and new_cost < cost[nx][ny]) {
                    cost[nx][ny] = new_cost;
                    vector<pair<int, int>> new_path = current.path;
                    new_path.push_back({nx, ny});
                    Node nextNode = {nx, ny, new_cost, optimisticCost, new_path};
                    pq.emplace(nextNode);
                    ++nexplored;
                }
            } else {++nunfeasible;}
        }
    }
    return make_tuple(best_cost, best_path,
                      vector<int>{nvisit, nexplored, nleaf, nunfeasible,
                      nnot_promising,
                                  npromising_but_discarded,
                                  nbest_solution_updated_from_leafs,
                                  nbest_solution_updated_from_pessimistic_bound});
}

// Función para imprimir el camino en formato 2D.
void printPath2D(int n, int m, const vector<pair<int, int>>& path) {
    vector<vector<char>> display(n, vector<char>(m, '.'));
    for (const auto& p : path) {
        display[p.first][p.second] = 'x';
    }
    for (const auto& row : display) {
        for (char cell : row) {
            cout << cell;
        }
        cout << endl;
    }
}
```

3. Resultados y Evaluación.

Para evaluar el rendimiento del algoritmo, se realizaron pruebas utilizando varias matrices de diferentes tamaños y configuraciones de costos facilitadas. A continuación se presentan algunos ejemplos de resultados obtenidos, dónde el óptimo es 1966.

Algoritmos en 100.map		
Algorítmo	**Tiempo (ms)**	**Resultado**
Recursivo Memoización	0.146	2360
Iterativo vector	0.022	2360
Voraz	0.034	3297
Backtracking	-	-
Ramificación y Poda	3.742	1966

Se muestra a continuación una demostración de la eficiencia de este algoritmo, comparando su resultado para el mapa 100.map para algunos de los distintos algoritmos realizados anteriormente.

Tiempo de ejecución	
Mapa	**Tiempo (ms)**
002.map	0.017
020.map	0.491
050.map	2.659
200.map	25.613
500.map	270.596
700.map	702.420
1K.map	2083.650
2K.map	16858.800
3K.map	54956.800
4K.map	124974

4. Conclusiones.

Se ha descubierto que el algoritmo de ramificación y poda es una técnica útil para resolver problemas de optimización para el problema del camino de mínimo coste frente a otros algoritmos, como se demuestra en el estudio. Los resultados demuestran que el uso de límites optimistas y pesimistas no sólo mejora significativamente el rendimiento frente al algoritmo de vuelta atrás realizado en la práctica 8, sino que también permite una reducción significativa en el espacio de búsqueda. La solidez y eficiencia del método propuesto han sido confirmadas mediante la resolución de un mapa específico.

El algoritmo visto en esta práctica final puede aplicarse en problemas de búsqueda muy grandes y complejos logrando una eficiencia inalcanzable por algoritmos vistos durante la asignatura como Programación Dinámica o Vuelta Atrás.