Algoritmos de búsqueda para el problema: TSP (Traveling Salesman Problem)

Matias Ramirez[1], Raul Galvan[1], Ricardo Gaona[1], Cesar Gonzalez[1]

[1] Facultad Politécnica, Universidad Nacional de Asunción

[1]{ramirezmatias946,ramako72, gaonaricardo06, giuli1297.gg}@fpuna.edu.py

Resumen. En este documento recabamos información acerca de algoritmos de búsqueda de soluciones para el problema del vendedor, información que comprende la característica de cada algoritmo implementado, su desempeño y comparándolos unos con otros. Estos algoritmos son los de búsqueda ciega, entre ellos un algoritmo de backtracking para encontrar la solución óptima entre todas, y un algoritmo de Las Vegas, que encuentra la primera solución ocurrente. Finalmente, implementaremos un algoritmo de búsqueda avara al cual le aplicaremos una mejora denominada 2-otp.

Palabras Clave: Backtracking, NP Completo, Heurística, Optimización, Algoritmo Avaro, Algoritmo Las Vegas.

1 Introducción

Un viajero desea poder visitar un conjunto de ciudades una vez empezando de una ciudad inicial. Uno de los problemas del viajero sería encontrar el camino más corto. El problema del viajero es conocido en varias ramas como matemáticas, ciencia de computación, e investigaciones de operaciones. Cuando se empezó a desarrollar la teoría con respecto a los NP completo, el TSP fue uno de los primeros problemas que fueron demostrados que tenga la dificultad NP.

El algoritmo que presentamos como solución es el de búsqueda avara con una optimización 2-opt, en el cual aplicamos una heurística de distancia al nodo inicial.
Describimos en profundidad el problema de viaje del vendedor en la sección 2, para luego exponer los algoritmos utilizados en la sección 3. Finalmente, mostramos los resultados experimentales en la sección 4.

2 Descripción del problema

El problema de viaje del vendedor es un problema que busca encontrar una ruta de mínimo costo para que un viajero pueda viajar desde una ciudad inicial a todas las ciudades destino, una sola vez sin repetir las ciudades en el trayecto.
Este es un problema de dificultad NP-Difícil, que busca optimización.
En este problema las ciudades pueden ser tomadas como entidades diferentes, como puntos en el espacio para problemas de astrología y movimiento de telescopio, o puntos de soldado para optimizar los movimientos de una máquina de soldado, interpretaciones que le dan distintas aplicaciones a los algoritmos que existen para resolver el problema en cuestión

3 Algoritmos implementados

3.1 Backtracking

El backtracking es una técnica algorítmica en la cual se intenta resolver un problema computacional de manera recursiva probando todos los casos posibles. Es considerado una de las técnicas más ineficientes en el peor de los casos, en la mayoría de los casos se emplea el backtracking como un último recurso para resolver el problema. Aun así, es usado constantemente, aún más en problemas de tipo NP completo.

Para el caso del problema del TSP el backtracking se aplica visitando recursivamente todos los nodos del grafo que representa a las ciudades y sus conexiones entre sí. El grafo se representa mediante su matriz de adyacencia y el algoritmo empieza en el nodo de origen. Dentro de la llamada desde el nodo actual se recorre todos sus vecinos y se realiza la llamada recursiva siempre y cuando el nodo vecino no ha sido visitado aún en el recorrido en curso y se agrega el nodo en la lista que representa el camino que se va recorriendo.

Se sabe que se ha encontrado una solución si se llega al último nodo del grafo y se comprueba que existe una conexión con el primer nodo visitado, es decir, el origen. En ese caso se agrega el camino recorrido junto con su costo a la lista de soluciones. Cuando ya se han encontrado todas las soluciones, se elige como solución óptima aquel que tenga el menor costo de ruta.

Algoritmo 1. Implementación de una solución al problema TSP mediante backtracking, que recibe como parámetros la matriz de adyacencia del grafo, el número n de nodos, una lista v que indica los nodos visitados, el camino

```
def tsp_backtracking(grafo, n, v, camino, soluciones, current,
count, costo, contador):
    contador.append(1)
    if count == n and checkConnection(grafo, current):
        found_path = camino.copy()
        found_path.append(1)
          soluciones.append({'costo': costo + grafo[current][0],
'camino': found_path})
        return

    for i in range(n):
        if v[i] == False and checkConnection(grafo, current, i):
            v[i] = True
            camino.append(i+1)
            tsp_backtracking(grafo, n, v, camino, soluciones, i,
count + 1, costo + grafo[current][i], contador)
            camino.pop()
            v[i] = False
```

3.2 Las vegas

Las vegas es una forma de programar algoritmos de búsqueda de soluciones en árboles de búsqueda generados a través de backtracking, aplicado al problema del viajero o TSP, se busca primeramente definir las restricciones y con ellas generar un conjunto de nodos frontera al actual, dentro de este conjunto de nodos debemos definir una variable de decisión, que será el próximo nodo a ser visitado por el algoritmo, como elegimos este nodo es donde entra en acción “Las Vegas”, ya que el criterio propuesto por este algoritmo es elegir un nodo al azar, de esta forma apostamos a una primera solución encontrada al azar, al seguidamente seleccionar nodos al azar dentro de la frontera del nodo actual, hasta encontrar un camino hamiltoniano desde el nodo inicial. Con esto no solo apostamos a la optimalidad de la solución, sino también en el caso de que no se pueda comprobar que existe una solución la apuesta es en el tiempo de ejecución, si no agregamos un tiempo máximo de búsqueda este tiempo puede ser el máximo. El algoritmo de las vegas no busca la solución más óptima si no una correcta, por lo que sí existe una o varias soluciones, la complejidad ejecución del algoritmo será como máximo NP-Difícil, resaltando que generalmente en cuanto a complejidad temporal y complejidad espacial es mejor que algoritmo de búsqueda ciega para la solución óptima (Backtracking), pero no siempre retorna una solución óptima.

Algoritmo 2. Implementación del algoritmo de las vegas, tiene como parámetros principales el nodo actual, los nodos visitados, el nodo inicial y la cantidad de nodos, y retorna el camino de solución encontrado.

```
def vegasTSP(nodo_actual, n_visitados, nodo_inicial, N, co,
ruta):
    nodos_visitados = n_visitados.copy()
    fronteras = []
    froteras_visitadas = []
    co.append(1)
    tree_node_number = len(co)
    for ar in nodo_actual.aristas:
            if (ar.nodo2 not in nodos_visitados or ar.nodo2 ==
nodo_inicial) and ar.nodo2 != nodo_actual:
             fronteras.append(ar.nodo2)
            if (ar.nodo1 not in nodos_visitados or ar.nodo1 ==
nodo_inicial) and ar.nodo1 != nodo_actual:
             fronteras.append(ar.nodo1)
    nodos_visitados.append(nodo_actual)
    while True:
        if nodo_inicial in fronteras:
            fronteras.remove(nodo_inicial)
            if len(nodos_visitados) == N:
                print("x", file=sys.stdout)
                ruta.extend(nodos_visitados)
                return nodos_visitados
             fronterasv2 = [x for x in fronteras if x not in
froteras_visitadas]
        if not fronterasv2:
            return []
        target = random.choice(fronterasv2)
        froteras_visitadas.append(target)
                 solution = vegasTSP(target, nodos_visitados,
nodo_inicial, N, co, ruta)
        if solution:
            return solution
```

3.3 El método avaro con optimización local 2-opt

El método Avaro se basa también en backtracking, aplicado al problema del viajero o TSP, se busca primeramente definir las restricciones y con ellas generar un conjunto de nodos frontera al actual, los nodos frontera se ordenan ascendentemente con respecto al coste(distancia que toma viajar desde el nodo actual al nodo frontera) sumado a la entropía(distancia que toma viajar desde el nodo inicial al nodo frontera), y se visitan en ese mismo orden, de este nodo el primero nodo visitado será el nodo con el menor coste.

El Avaro con optimización se basa en buscar en la solución que se obtuvo del método Avaro si hay posibilidad de hacer intercambios entre dos pares de caminos adyacentes, de manera a generar un menor coste del recorrido total.

Algoritmo 3. Implementación del algoritmo de búsqueda Avara, tiene como parámetros principales el nodo actual, los nodos visitados, el nodo inicial y la cantidad de nodos, y retorna el camino de solución encontrado.

```
def avaroTSP(nodo_actual, n_visitados, nodo_inicial, N, co,
ruta):
    nodos_visitados = n_visitados.copy()
    fronteras = []
    froteras_visitadas = []
    co.append(1)
    tree_node_number = len(co)
    def getCost(e):
        return e.costo+distanciaRecta(nodo_inicial, e.nodo2)
    nodo_actual.aristas.sort(key=getCost)
    for ar in nodo_actual.aristas:
           if (ar.nodo2 not in nodos_visitados or ar.nodo2 ==
nodo_inicial) and ar.nodo2 != nodo_actual:
            fronteras.append(ar.nodo2)
           if (ar.nodo1 not in nodos_visitados or ar.nodo1 ==
nodo_inicial) and ar.nodo1 != nodo_actual:
            fronteras.append(ar.nodo1)
    nodos_visitados.append(nodo_actual)
    while True:
        if nodo_inicial in fronteras:
            fronteras.remove(nodo_inicial)
            if len(nodos_visitados) == N:
                # print(nodos_visitados, file=sys.stdout)
                print("x", file=sys.stdout)
                ruta.extend(nodos_visitados)
                return nodos_visitados
            fronterasv2 = [x for x in fronteras if x not in
froteras_visitadas]
        if not fronterasv2:
            return []
        target = fronterasv2[0]
        froteras_visitadas.append(target)
                solution = avaroTSP(target, nodos_visitados,
nodo_inicial, N, co, ruta)
        if solution:
            return solution
```

Algoritmo 4. Implementación del algoritmo de mejora 2-otp, tiene como parámetro principal la lista de los nodos solución, es decir el camino solución, y retorna un camino mejorado si es posible.

```
def optimizacion2opt(graph):
    cambio_min = 0
    for i in range(len(graph)-2):
```

```
        for j in range(i+2, len(graph)-1):
            cambio=0
                                costo_actual  =  distancia(graph[i],
graph[i+1])+distancia(graph[j], graph[j+1])
                if distancia(graph[i], graph[j]) is not None and
distancia(graph[i + 1], graph[j + 1]) is not None:
                                 costo_nuevo  =  distancia(graph[i],
graph[j])+distancia(graph[i+1], graph[j+1])
                    cambio = costo_nuevo - costo_actual
            if cambio < cambio_min:
                cambio_min = cambio
                i_min = i
                j_min = j
    if cambio_min < 0:
        graph[i_min+1:j_min+1]=graph[i_min+1:j_min+1][::-1]
    return graph
```

4 Resultados experimentales

El entorno de prueba considerado para los algoritmos propuestos consiste en un generador de grafos completamente conexos en el cual agregamos como parámetro la cantidad de nodos del grafo, en este cada nodo se genera con una coordenada (x, y) que se utiliza para determinar la distancia de un nodo a otro en línea recta, luego otro parámetro es la distancia máxima agregada que se utiliza para generar el costo de ruta de un nodo a otro, asignando este costo a cada arista representando la distancia en ruta de una ciudad a otra en el contexto del problema del viajero, este costo será mayor al de línea recta entre dos nodos.

Con esto, hemos logrado los resultados mostrados en la tabla Fig.1, en la cual queda evidenciado que el desempeño de algoritmos de búsqueda ciega como el de backtracking tienen un alto costo en complejidad temporal y espacial, mientras que un algoritmo de búsqueda ciega que tiene como meta una solución completa pero no óptima como los es el de “Las Vegas” demuestra tener un coste constante bajo, ya que al ser un grafo completamente conexo un camino hamiltoniano siempre será encontrado al elegir nodos siguientes al azar. Lo resaltante viene dado con el algoritmo de búsqueda avara con mejora 2-otp, que demuestra un crecimiento menor al del de backtracking, el costo de elegir un nodo mejor en términos del costo de ruta y la heurística de distancia de línea recta viene dado con un crecimiento exponencial NP-Difícil, que es generalmente menor al de del backtracking.

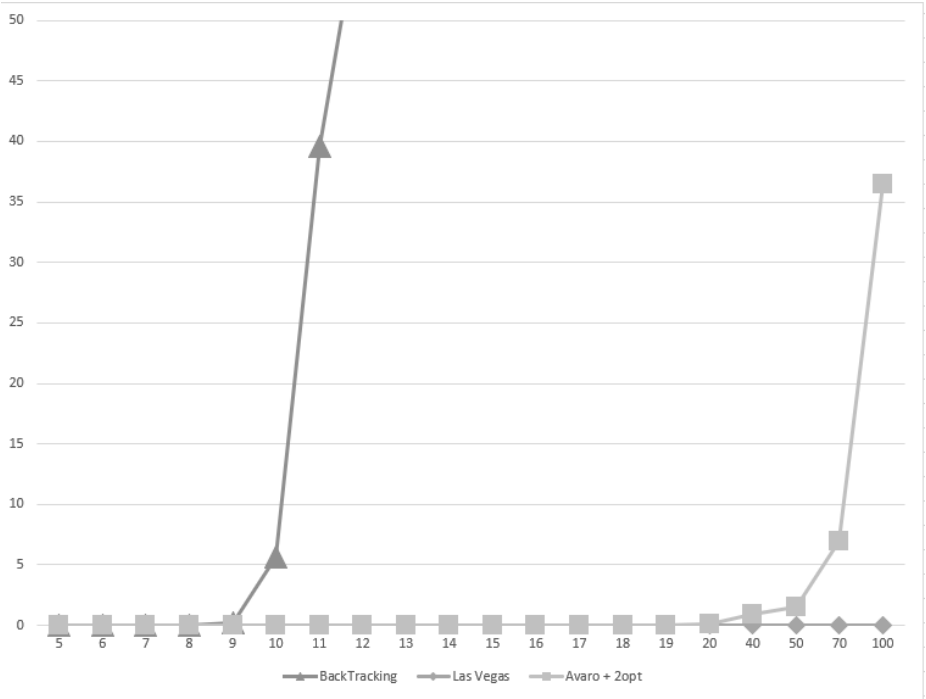

Fig. **1.** Una gráfica que en el “eje x” tiene la cantidad de nodos que tiene como entrada cada algoritmo, y en el “eje y” contiene los segundos que le tomó a cada algoritmo encontrar una solución para el problema de TSP.

5 Conclusiones

En el presente trabajo hemos implementado tres algoritmos y una mejora para resolver el problema de TSP(Traveling Salesman Problem), el algoritmo de Backtracking es el de mayor consumo en cuanto a procesamiento y a complejidad espacial, porque debe recorrer todas las opciones posibles, pero por esta misma razón el algoritmo siempre encuentra la solución óptima, el método las Vegas es el de menor coste de procesamiento y complejidad espacial, es poco probable obtener la solución óptima, pero se obtiene una solución razonable con respecto al tiempo de búsqueda, el Método Avaro es 2do mejor con respecto a procesamiento y complejidad espacial, presenta una mejor solución con respecto al método Las Vegas, ya que tiene una mayor probabilidad de encontrar la solución óptima o por lo menos mejor, por último el método Avaro con optimización local 2-opt tiene similar coste espacial al de Avaro pero tiene un coste de procesamiento ligeramente mayor correspondiente a la mejora 2-opt, la solución obtenida resulta con una pequeña mejora con respecto a la solución del Avaro simple, y es capaz de mejorar la solución del algoritmo avaro, a un punto probable de convertirlo en una solución óptima.

Tenemos cuatro formas de resolver un problema, y con respecto a cuál sería el mejor algoritmo, la respuesta es depende, porque depende de que tan grande sea el problema, depende de si preferimos priorizar el tiempo de búsqueda, la complejidad espacial, y también depende si buscamos la solución óptima o si nos conformamos con una solución razonable con respecto al tiempo de búsqueda.

En cuanto a trabajos futuros similares a este, cabría la posibilidad de probar mecanismos de procesamiento paralelos para comparar soluciones de distintos hilos y elegir el mejor, también se podría investigar más métodos de evaluación de algoritmos para una mejor elaboración de los resultados experimentales.

Referencias

1. Jünger, M.; Reinalt, G.; Rinaldi, G.: The Traveling Salesman Problem.
2. Cummings, N.: A brief History of the Traveling Salesman Problem. The Operational Research Society.

https://www.theorsociety.com/about-or/or-methods/heuristics/a-brief-history-of-the-travelling-salesman-problem/