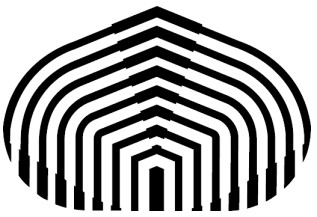

UNIVERSIDAD SIMÓN BOLÍVAR

CI5437

Informe Proyecto Final

Estudiantes:

Pietro Iaia 15-10718

Diego Peña 15-11095

Sartenejas, Julio 2021

1 Introducción

El problema del agente viajero (TSP, por sus siglas en inglés, Traveling Salesman Problem) es un problema matemático muy conocido, en el cual, dada una lista de ciudades y las distancias entre cada una de ellas, se debe determinar la ruta más corta (O de menor costo) para que el agente recorra todas las ciudades exactamente una vez y regresé a la ciudad original. Las primeras formulaciones matemáticas del problema vienen de principios del siglo XIX en Europa, cuando W.R Hamilton y Thomas Kirkman lo plantearon como parte de una discusión acerca de ciclos hamiltonianos (Biggs, et al, 1986). Desde este punto de vista, se puede considerar el TSP como el problema de hallar el ciclo hamiltoniano más corto.

Desde entonces ha sido un problema ampliamente estudiado debido a su enorme utilidad práctica: Hallar una solución para una instancia del mismo puede representar un ahorro importante para una empresa que tenga que hacer una serie de entregas o para compañías de transporte, ya que pueden planificar la ruta más corta y más económica para sus vehículos. Sin embargo, el problema es NP-Hard, lo que dificulta su resolución para contextos en los que haya muchas ciudades o puntos que recorrer.

El objetivo de este informe es comparar distintas técnicas de resolución del problema. Algunos son métodos exactos, que, de terminar deben devolver la solución óptima, mientras que otros son aproximados, que devuelven una solución que en la mayoría de los casos es buena (En algunos casos incluso puede que lleguen a la óptima), pero que no ofrecen garantías en ese aspecto

2 Metodología:

A continuación se dará una breve explicación de los métodos utilizados para intentar resolver el problema

2.1 Técnicas exactas

2.1.1 Búsqueda en el espacio de estados

Se utilizó el modelo de espacio de estados para representar el problema, de acuerdo a la representación vista en clases:

- Sea *S* el conjunto de secuencias de ciudades, tales que solo se puede repetir la primera ciudad de la secuencia, y esta repetición debe aparecer al final del recorrido
- Sea A el conjunto de operaciones posibles sobre una secuencia: *Append(s, i)* que añade la ciudad *i* a la secuencia *s* siempre y cuando *i* no pertenezca a *s* y *Fin(s)* que añade la ciudad inicial de la secuencia al final del recorrido siempre y cuando *s* contenga todas las ciudades.
- Sea s_{init} la secuencia vacía
- Sea S_G el estado objetivo, es decir una secuencia que contenga el ciclo que pasa por todas las ciudades solo una vez (Excepto por la primera, que obviamente también es la última, porque es un ciclo). Nótese que para resolver el problema correctamente se debe seleccionar el estado $s \in S_G$ de menor costo
- La función de transición *f(s, a)* aplica la acción especificada al estado actual
- El costo de una acción es la distancia entre la ciudad al final de la secuencia *s* y la ciudad a añadir a la secuencia

Para hacer el recorrido del espacio de estado se utilizó un algoritmo de la familia Branch and Bound, particularmente el Depth First Branch and Bound. Como heurística se utilizó la vista en clases calcular el costo del árbol mínimo cobertor (MST por sus siglas en inglés, Minimum Spanning Tree) de las ciudades que no estuvieran en el recorrido parcial, más los dos lados de menor costo que conectaran un vértice que estuviera en el recorrido parcial con el MST.

Esta implementación de branch and bound requería una cota superior inicial. Se probaron dos metodologías para obtenerlas:

- Generar un ciclo hamiltoniano inicial aleatorio: Se creó un camino Hamiltoniano que iba de la primera ciudad a la última. Se ejecutó una permutación aleatoria y se añadió el primer esto de esta permutación al final.
- Generar un ciclo hamiltoniano con un algoritmo aproximado: Particularmente, se utilizó el algoritmo 2-opt, el cual se detallará más adelante.

2.1.2 Programación lineal entera:

Se utilizó un solver para programación lineal entera para resolver los problemas. El primer paso fue modelar el TSP como un problema de programación lineal. Para ello se utilizó el modelo para n ciudades planteado por Dantzig-Fulkerson-Johnson en el cual:

- Variables X_{ij} con $i, j \in \{1..n\} \wedge i \neq j$ tal que $X_{ij} = 1$ si en la solución, se utiliza el arco que va desde i hasta j y $X_{ij} = 0$en caso contrario
- La función objetivo es $\sum_{i=1}^{n} \sum_{j=1\, i \neq j}^{n} C_{ij} X_{ij}$ donde C_{ij}es la distancia del arco entre i y j
- Las restricciones son
    - Para cada ciudad i, solo un arco puede salir de ella hacia otra ciudad $(\forall i \mid i \in \{1..n\} : \sum_{j=1,\, i \neq j}^{n} X_{ij} = 1)$
    - Para cada ciudad j, solo un arco puede salir de ella hacia otra ciudad $(\forall j \mid j \in \{1..n\} : \sum_{i=1,\, i \neq j} X_{ij=1})$
    - Las dos restricciones anteriores permiten que el solver devuelva como solución una conjunto de ciclos disjuntos, en lugar de un solo ciclo hamiltoniano. En caso de que pase esto, deben agregarse restricciones adicionales. Hay varias formas de modelar esto, la utilizada fue: Sea S el conjunto de variables que representan uno de los ciclos anteriormente mencionados, se tiene $\sum_{X_{ij} \in S} X_{ij} \leq |S| - 1$. Deben seguirse agregando este tipo de restricciones iterativamente hasta que la solución sea un ciclo hamiltoniano
- Las variables deben ser enteras

2.1.3 Algoritmo Held-Karp

Uno de los algoritmos utilizados para el estudio fue el algoritmo de Herd-Karp, también llamado el algoritmo de Bellman-Herd-Karp, el cual es un algoritmo de programación dinámica usado para solucionar el travelling salesman problem (TSP). Este algoritmo acepta como parámetro una matriz con las distancias unidireccionales entre cada par de ciudades que componen el problema. El objetivo es encontrar un camino óptimo que visite cada ciudad exactamente una vez antes de regresar al punto de partida.

Para que este algoritmo funcione correctamente, primero debemos enumerar las ciudades 1, 2, …, n. Donde la ciudad 1 representa el punto de partida, o ciudad origen, en el problema.

Aparte de esto, la posición <n, m> en la matriz de distancias representa la distancia desde la ciudad n a la ciudad m. Herd-Karp entonces empieza calculando, por cada set de ciudades S y cada ciudad no contenida en estos set, el camino unidireccional más corto desde la ciudad origen 1 hasta cada ciudad no contenida en S pasando por todas las ciudades en S una sola vez. Este algoritmo funciona como un algoritmo de programación dinámica debido a que primero podemos resolver los caminos con menos ciudades, para luego usar su resultado y resolver los caminos con más ciudades. Es decir, primero calculamos los caminos óptimos para todos los sets de tamaño |S| = i, para luego usar sus resultados y calcular los caminos óptimos para todos los sets de tamaño |S| = i+1. Esto se repite hasta encontrar el camino óptimo entre la ciudad origen y sí misma, es decir, encontrar el camino óptimo para el único set de tamaño |S|=n-1, que sería el set que contiene a todas las ciudades sin incluir la ciudad inicial.

2.2 Técnicas de aproximación

2.2.1 Vecino más cercano (Nearest Neighbour) randomizado

Uno de los algoritmos aproximados que se puede utilizar para el problema de TSP, es un algoritmo greedy, conocido como nearest neighbour, que selecciona una ciudad inicial para su recorrido, y partir de entonces, siempre añade al recorrido parcial la ciudad más cercana a la que esté de último en el recorrido parcial. En este caso, la selección de la ciudad inicial se hace aleatoriamente

2.2.2 Vecino más cercano (Nearest Neighbour) iterativo:

Esta técnica utiliza el mismo algoritmo descrito anteriormente (Nearest Neighbour), pero lo ejecuta N veces (N es el número de ciudades): Una partiendo de cada ciudad en el problema. AL final devuelve el recorrido de menor costo de todos los que calculó

2.2.3 Algoritmo 2-opt

Este algoritmo parte de un recorrido completo cualquiera y toma dos puntos cualesquiera (Salvo la ciudad de origen/destino) e invierte el recorrido entre esos dos puntos. El algoritmo selecciona sistemáticamente cada combinación de puntos, construye el camino dado por la inversión del segmento entre ellos, y si el recorrido resultante es mejor, lo guarda y reinicia el proceso de selección de puntos, solo que esta vez toma el nuevo recorrido recién conseguido como base. Este proceso se repite hasta que no se consiga un recorrido mejor.

En esta implementación, el recorrido base inicial que usa el algoritmo se obtiene a partir del algoritmo de vecino más cercano iterativo

Todas estas técnicas se aplicaron sobre un conjunto de 11 datasets para TSP, con una cantidad de ciudades variable. Iban desde las 5 ciudades hasta las 127. Los datasets FIVE y P01 son exclusivos de John Burkard, mientras que el resto provienen la TSPLIB mantenida por Gerhard Reinelt. Varios de los casos utilizados de la TSPLIB son los mismos que utiliza John Bukard. Se usaron esos ya que el en su página él incluía las instancias representadas como matriz de costos, y este era el formato utilizado en la implementación. Estos casos de prueba se encuentran en el repo de Github del proyecto en la carpeta Datasets.

Todos los algoritmos se ejecutaron por un máximo de 10 minutos

3 Implementación

El algoritmo de branch and bound, el de Held-Karp y los métodos heurísticos fueron implementados en C++ sin librerías de terceros. Para ejecutarlos debe entrar al archivo algorithms.cpp y quitar los comentarios del algoritmo que se desea ejecutar, así como de las líneas de tsart, tend y chrono que vienen inmediatamente antes o después del algoritmo seleccionado. Luego, debe hacerse make y ejecutar el programa main en la terminal con el nombre del archivo que contiene el caso de prueba. La implementación de Branch and Bound se hizo de forma que solo tuviera que guardarse un estado en memoria, el cual era almacenado globalmente (Similar a la implementación de IDA* del proyecto 1)

Para el modelo de programación lineal se utilizó el solver de la librería pulp de python, que permite resolver problemas de programación lineal entera. El solver en sí es de la librería, sin embargo, se implementó código adicional para revisar si la solución al modelo era efectivamente un único ciclo o un conjunto de ciclos. En este último caso, el código implementado agrega las restricciones adicionales al modelo y vuelve a ejecutar el solver con las nuevas restricciones. Este proceso se repite hasta que se consigue un único recorrido que abarca todas las ciudades. Por las propiedades matemáticas del modelo, este recorrido es la solución óptima.

Para el formato de los casos de prueba, se debe utilizar un archivo que contenga la matriz de costos con header. El header debe ser una única línea al con la letra c, para distinguirlo de las demás líneas. La matriz de costos solo debe representarse solo con sus contenidos, es decir, nada de nombres para las filas o columnas de la matriz. Los algoritmos

implementados las enumeran automáticamente y saben que, por ejemplo M[0][2] se refiere al costo de ir de la ciudad 0 a la ciudad 2.

Se usan unos scripts de python para ejecutar todos los casos de una vez y anotarlos en un csv. Estos son runBnB (Para branch and bound), runHeuristic (Para los aproximados), runHeldKarp (Para el algoritmo homónimo) y runLPS (Para el solver lineal). Cabe destacar, que para cualquiera de los primeros tres casos, el usuario debe asegurarse de que está ejecutándose el archivo correspondiente en el main de algorithms.cpp.

4 Resultados:

A continuación se incluyen tablas que muestran el desempeño promedio de los algoritmos utilizados en los casos seleccionados. Nótese, que para el cálculo de los tiempos promedios, solo se tomaron en cuenta los casos resueltos por los algoritmos

- BnB rp: Branch and Bound con cota inicial por recorrido random.
- BnB 2op: Branch and Bound con cota inicial por algoritmo 2-opt.

Tipo de Búsqueda	% Solución
BnB rp	36.36%
BnB 2-op	36.36%
Todos	36.36%

Tabla 4.1: Porcentaje de solución de tipos de Branch and Bound

Tipo de Búsqueda	Cota inicial	Nodos generados	Nodos expandidos	Costo	Tiempo (segundos)
BnB 2-op	920	1149250	105571	833	118.366
BnB rp	2076	1417608	141960	833	138.258
Todos	1498	12834229	123766	833	128.312

Tabla 4.2: Comparación entre los dos tipos de búsqueda

Tipo de Búsqueda	% Solución
BnB rp	36.36%
BnB 2-op	36.36%
Held Karp	9.1%
Prog. Lineal	63.63%
Todos	40%

Tabla 4.3: Porcentaje de solución de métodos exactos

Método	Tiempo (segundos)
BnB rp	118.366
BnB 2-op	138.258
Held Karp	-
Prog. Lineal	0.027
Todos	68.445

Tabla 4.4: Porcentaje de tiempo, en segundos, para obtener la solución en métodos exactos

Método	% Finalización
2-op	100%
Iter. Nearest Neighbour	100%
Nearest Neighbour Random	100%
Todos	100%

Tabla 4.5: Porcentaje de Finalización (Es decir, que no hagan timeout) de métodos heurísticos

Método	Tiempo (segundos)	Proporción sol. est./sol. real
2-op	0.014	2.532
Iter. Nearest Neighbour	0.010	3.614
Nearest Neighbour Random	0	4.615
Todos	0.0008	3.587

Tabla 4.6: Porcentaje de tiempo, en segundos, y proporción entre la solución estimada y la real utilizando métodos heurísticos

Método	Tiempo (segundos)
BnB 2-op	138.258
BnB rp	118.366
Held Karp	-
Prog. Lineal	0.027
2-op	0.014
Iter. Nearest	0.010

Neighbour	
Nearest Neighbour Random	0
Todos	21.394

Tabla 4.7: Porcentaje de tiempo, en segundos, y tasa para obtener la solución en todos los métodos

5 Análisis de resultados

En cuanto a los resultados de Branch and Bound, se puede apreciar que no hay diferencia en términos de porcentaje de casos resueltos entre los distintos enfoques que se le dio a la selección de la cota superior inicial (Tabla 4.1). Sin embargo, sí hubo una mejoría en cota inicial, términos de tiempo y de nodos expandidos/generados (Tabla 4.2). Esto se debe a que en el caso de recorrido aleatorio, para elegir la cota inicial se toma en cuenta un único recorrido al azar, independientemente de si existe uno mejor o no. El hecho de que sea aleatorio, por temas de probabilidad,mitiga la posibilidad que el recorrido tenga un costo particularmente malo, pero también de que tenga un costo particularmente bueno. Por su parte, 2-opt hace una comparación entre varios recorridos posibles vía vecino más cercano iterativo (Que ya de por sí vecino más cercano se guía por una heurística, que es otra ventaja) y luego trata de mejorar el costo del que mejor resultado le dio, mediante inversión de segmentos. Aparte de que el promedio de costos sea inferior cuando se usa 2-opt, a nivel individual, para los datasets utilizados, todas las cotas superiores iniciales fueron menores respecto a las de su contraparte.

Sin embargo el porcentaje de resolución de casos vía branch and bound fue relativamente bajo. Esto en parte, puede ser por el cálculo de la heurística, ya que cada vez que llega a un nodo debe resolver el problema del MST para los nodos fuera del recorrido. A pesar de que el algoritmo de Prim utilizado es bastante eficiente, el tiempo de ejecutarlo miles o millones de veces se va acumulando y termina consumiendo gran parte de la ejecución del branch and bound. Quizás el uso de técnicas más modernas como PDBs pueda ayudar a reducir un poco este tiempo. Otra posible mejora, podría obtenerse mediante una representación más compacta del recorrido parcial (Se utilizó un vector de enteros).

Si se ve la comparación entre todos los métodos exactos (Tabla 4.3), se puede apreciar que el más efectivo es el solver de programación lineal entera, y el menos es el algoritmo de Held-Karp, con Branch and Bound en el medio.

El problema de Held-Karp, no es solo el hecho de ser un algoritmo exponencial $O(2^n n^2)$, cosa que era de esperarse porque el TSP es un problema exponencial, sino que no se implementaron las mejoras de programación dinámica, lo cual hizo que se repitieran muchos cálculos muchas veces. Aún si se hubieran implementado, esto solo hubiera servido para los casos más pequeños, ya que se requería indexar por conjuntos de ciudades, y aunque estos podían representarse eficientemente por cadenas de bits, ya para instancias medianas la tabla se haría gigantesca, y en los casos más grandes, abiertamente imposible de representar. Este no es un algoritmo práctico para casos grandes o medianos.

En cuanto a la programación lineal entera, sus mejores resultados se deben en gran parte a que es una librería externa especializada para esa tarea y seguramente cuenta con numerosas optimizaciones que facilitan su resolución. Además, como se dijo en la introducción el TSP es un problema común en el área de investigación y planificación de operaciones, donde se utilizan mucho los solvers de programación lineal. Es posible que este sea uno de los problemas "benchmarks" con los que se prueban y optimizan los solvers durante su desarrollo. Como idea para un futuro proyecto, sería interesante buscar un solver de programación lineal entera enfocado en TSP, donde las optimizaciones no estén solo en el solver en sí, sino en el proceso de verificación de si la solución dada efectivamente es un único ciclo que abarca con todas las ciudades, ya que este proceso se podría hacer de manera más rápida si se trabaja con información que el solver maneja internamente.

Para la tabla 4.4, no se incluyó el tiempo promedio del algoritmo Herd-Karp ya que solo pudo resolver uno de los casos de prueba seleccionados, y fue el más sencillo, el cual resolvió casi instantáneamente. Por esa razón, y para que no pareciera engañoso en la tabla que Held-Karp resolvió sus casos de prueba en 0 segundos, se decidió no colocarlo en el análisis de tiempo promedio.

En cuanto a los algoritmos aproximados (Tabla 4.6), los resultados fueron los esperados. Dado que vecino más cercano iterativo contiene varias iteraciones de vecino más cercano, tarda más tiempo que vecino más cercano aleatorio, que solo se ejecuta una vez, pero garantiza una solución de mejor o igual costo. Esto se evidencia en la proporción estimación/solución real (División del costo la solución ofrecida por el método aproximado y el costo de la solución exacta). Una buena estimación debe tener el valor más cercano a uno

posible, ya que esto significa que la estimación tiene un valor muy similar a la solución real. En este caso, la estimación más cercana a 1, la tiene el método iterativo

Similarmente, 2-opt toma más tiempo que vecino más cercano iterativo, pero es porque debe ejecutar ese algoritmo, tomar la solución ofrecida y realizar una serie de inversiones de segmentos para tratar de encontrar un camino de menor costo. Esto garantiza que 2-opt devuelva una solución de igual o mejor costo que vecino más cercano iterativo. Al igual que en el caso anterior, esto se puede evidenciar en la proporción estimación/solución real, donde el valor más cercano a 1 lo ofrece 2-opt (Inclusive, para el caso GR17, 2-opt devolvió la solución óptima). Si se ven las tablas que muestran los resultados individuales de cada algoritmo (Están en el repo de github del proyecto) se podrá observar que estas estimaciones son mejores en casos más pequeños. Esto puede deber a que el número de recorridos posibles es menor en estos casos, y por lo tanto, entre el proceso de ejecutar varias iteraciones de vecino más cercano, y el proceso de mejoras de 2-opt como tal, se considera una mayor proporción de todos los caminos posibles, y hay una mejor posibilidad de encontrar la solución exacta o al menos una bastante cercana

Aprovechando el poco tiempo que lleva ejecutar estos métodos, se recomienda seguir combinando este tipo de algoritmos de distintas formas para mejorar las aproximaciones que dan. Por ejemplo, se puede aplicar 2-opt sobre la solución obtenida en cada iteración de vecino más cercano iterativo y hacer la comparación. O después de aplicar 2-opt, se pueden aplicar otras técnicas de tipo k-opt, como 3-opt, que toma tres puntos, en invierte el segmento entre el primer y el segundo punto y el segundo y el tercer punto.

6 Conclusiones

- La programación lineal es un método bastante efectivo para calcular el TSP con casos pequeños y medianos
- Sin programación dinámica, el algoritmo Held-Karp resulta muy ineficiente inclusive para casos pequeños
- En el modelo de espacio de estados, el cálculo de la heurística de un estado puede influir bastante sobre el tiempo de ejecución de los algoritmos de búsqueda.
- Los algoritmos heurísticos permiten encontrar soluciones relativamente buenas en una fracción del tiempo de los algoritmos exactos
- Los algoritmos heurísticos dan mejores resultados en los casos pequeños
- La selección de la cota superior inicial puede influir en el rendimiento de los algoritmos branch and bound

7 Referencias

Biggs, N., Lloyd, E. K., & Wilson, R. J. (1986). *Graph Theory, 1736-1936*. Oxford University Press.

8 Fuentes de los casos de prueba:

- Recopilación de John Burkardt: https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html
- Recopilación TSPLIB de Gerhard Reinelt: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/