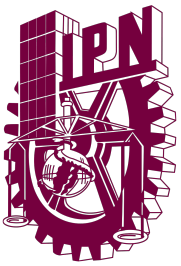

INSTITUTO POLITÉCNICO NACIONAL

CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

T E S I S

Optimización combinatoria usando técnicas de aprendizaje por refuerzo

QUE PARA OBTENER EL GRADO DE:

MAESTRÍA EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:

Ing. Erick González Arciniega

DIRECTORES DE TESIS:

Dr. Rolando Menchaca Méndez
Dr. Mario Eduardo Rivero Ángeles

Ciudad de México Junio 2022

SIP-13
REP 2017

INSTITUTO POLITÉCNICO NACIONAL
SECRETARIA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REGISTRO DE TEMA DE TESIS
Y DESIGNACIÓN DE DIRECTOR DE TESIS

Ciudad de México, a 10 de julio del 2020

El Colegio de Profesores de Posgrado del **Centro de Investigación en Computación** en su Sesión
(Unidad Académica)

Ordinaria No. 5 celebrada el día 29 del mes mayo de 2020, conoció la solicitud presentada por el (la) alumno (a):

Apellido Paterno:	GONZÁLEZ	Apellido Materno:	ARCINIEGA	Nombre (s):	ERICK

Número de registro: A 2 0 0 3 6 7

del Programa Académico de Posgrado: **Maestría en Ciencias de la Computación**

Referente al registro de su tema de tesis; acordando lo siguiente:

1.- Se designa al aspirante el tema de tesis titulado:

"Optimización combinatoria usando técnicas de aprendizaje por refuerzo"

Objetivo general del trabajo de tesis:

Desarrollar y caracterizar experimentalmente una serie de algoritmos de optimización combinatoria para el problema del viajero que utilicen técnicas de aprendizaje por refuerzo.

2.- Se designa como Directores de Tesis a los profesores:

Director: **Dr. Rolando Menchaca Méndez** 2° Director: **Dr. Mario Eduardo Rivero Ángeles**

No aplica: ☐

3.- El Trabajo de investigación base para el desarrollo de la tesis será elaborado por el alumno en:

Centro de Investigación en Computación

que cuenta con los recursos e infraestructura necesarios.

4.- El interesado deberá asistir a los seminarios desarrollados en el área de adscripción del trabajo desde la fecha en que se suscribe la presente, hasta la aprobación de la versión completa de la tesis por parte de la Comisión Revisora correspondiente.

Director(a) de Tesis

Dr. Rolando Menchaca Méndez

2º Director de Tesis

Dr. Mario Eduardo Rivero Ángeles

Aspirante

C. Erick González Arciniega

Presidente del Colegio

Dr. Marco Antonio Moreno Ibarra

INSTITUTO POLITECNICO NACIONAL
CENTRO DE INVESTIGACION
EN COMPUTACION
DIRECCION

SIP-14
REP 2017

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESIS

En la Ciudad de México siendo las 12:00 horas del día 15 del mes de junio del 2022 se reunieron los miembros de la Comisión Revisora de la Tesis, designada por el Colegio de Profesores de Posgrado de: Centro de Investigación en Computación para examinar la tesis titulada: **"Optimización combinatoria usando técnicas de aprendizaje por refuerzo"** del (la) alumno (a):

Apellido Paterno:	GONZÁLEZ	Apellido Materno:	ARCINIEGA	Nombre (s):	ERICK

Número de registro: A 2 0 0 3 6 7

Aspirante del Programa Académico de Posgrado: **Maestría en Ciencias de la Computación**

Una vez que se realizó un análisis de similitud de texto, utilizando el software antiplagio, se encontró que el trabajo de tesis tiene 10 % de similitud. **Se adjunta reporte de software utilizado**.

Después que esta Comisión revisó exhaustivamente el contenido, estructura, intención y ubicación de los textos de la tesis identificados como coincidentes con otros documentos, concluyó que en el presente trabajo **SI** ☐ **NO** ☒ **SE CONSTITUYE UN POSIBLE PLAGIO.**

JUSTIFICACIÓN DE LA CONCLUSIÓN: *(Por ejemplo, el % de similitud se localiza en metodologías adecuadamente referidas a fuente original)*
El porcentaje de similitud se localiza en frases de uso común en el área, asi como en formulaciones matemáticas que aparecen en las referencias citadas en la tesis.

****Es responsabilidad del alumno como autor de la tesis la verificación antiplagio, y del Director o Directores de tesis el análisis del % de similitud para establecer el riesgo o la existencia de un posible plagio.**

Finalmente y posterior a la lectura, revisión individual, así como el análisis e intercambio de opiniones, los miembros de la Comisión manifestaron **APROBAR** ☒ **SUSPENDER** ☐ **NO APROBAR** ☐ la tesis por **UNANIMIDAD** ☒ o **MAYORÍA** ☐ en virtud de los motivos siguientes:
Cumple con los requisitos académicos establecidos para un trabajo de maestría.

COMISIÓN REVISORA DE TESIS

Dr. Rolando Menchaca Méndez
Director de Tesis

Dr. Grigori Sidorov

Dr. Marco Antonio Moreno Ibarra

Dr. Mario Eduardo Rivero Ángeles
2° Director de Tesis

Dr. Ricardo Menchaca Méndez

Dr. Rolando Quintero Tellez

INSTITUTO POLITÉCNICO NACIONAL
CENTRO DE INVESTIGACIÓN
EN COMPUTACIÓN

Dr. Francisco Hiram Calvo Castro
PRESIDENTE DEL COLEGIO DE
PROFESORES

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA DE AUTORIZACIÓN DE USO DE OBRA PARA DIFUSIÓN

En la Ciudad de México el día _30_ del mes de _Junio_ del año __2022__, el (la) que suscribe _Erick González Arciniga_ alumno(a) del programa _Maestría en Ciencias de la Computación_ con número de registro _A200367_, adscrito(a) a _Centro de Investigación en Computación__ manifiesta que es autor(a) intelectual del presente trabajo de tesis bajo la dirección de _Rolando Menchaca Méndez__ y cede los derechos del trabajo intitulado _Optimización combinatoria usando técnicas de aprendizaje por refuerzo_, al Instituto Politécnico Nacional, para su difusión con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expresado del autor y/o director(es). Este puede ser obtenido escribiendo a las siguiente(s) dirección(es) de correo. __arciniegaerick@gmail.com__. Si el permiso se otorga, al usuario deberá dar agradecimiento correspondiente y citar la fuente de este.

Erick González Arciniega

Nombre completo y firma autografá del (de la) estudiante

Resumen

La clase de problemas NP-difíciles (NP-hard) está compuesta por un conjunto de problemas que tienen la propiedad de que un algoritmo que resuelva a cualquiera de ellos puede ser usado para resolver en tiempo polinomial cualquier problema de la clase de problemas NP. En la actualidad, no se conoce ningún algoritmo que resuelva un problema NP-difícil y tampoco se ha podido demostrar que esto sea imposible. Uno de estos problemas NP-difíciles es el problema del viajero (TSP, Traveling Salesman Problem), cuya versión de decisión, forma parte de los 21 problemas que Richard Karp demostró pertenecen a la clase NP-difícil. Dada la complejidad impuesta por el problema del viajero, a la fecha se han propuesto un número importante de algoritmos que buscan soluciones cercanas al óptimo en tiempo razonable. En esta tesis se presenta un algoritmo para el problema del viajero que utiliza elementos del Aprendizaje por refuerzo, en particular análisis de k pasos adelante (k-steps lookahead) y completado de episodios mediante heurísticas (rollout), que busca balancear la calidad de las soluciones con el tiempo de ejecución. Usando instancias de la biblioteca TSPLIB se presenta un análisis comparativo del desempeño del algoritmo propuesto contra el desempeño de un algoritmo voraz, un algoritmo basado en colonia de hormigas, y un algoritmo exacto basado en una formulación de programación entera.

Abstract

The NP-hard class of problems is composed of a set of problems with the property that an algorithm that solves any of them can be used to solve any problem that belongs to the NP class of problems in polynomial time. To this date, either providing an algorithm for any of the NP-hard problems or showing that such an algorithm does not exist remains an open problem. The Traveling Salesman Problem (TSP) is one of the 21 problems that Richard Karp showed to belong to the NP-hard class. Given the complexity of the TSP, up to date, there have been a number of proposals that provide algorithms that look for optimized solutions in reduced time. In this thesis, we present an algorithm for the STP that employs elements of Reinforcement Learning, in particular, k-steps look ahead and rollout, to balance the quality of the solutions with the algorithm's running time. We use instances of the TSPLib to perform a comparative performance analysis of the proposed algorithm against a greedy algorithm, an ant colony optimizatión algorithm, and an exact algorithm based on an integer programming formulation.

Agradecimientos

Agradezco al Consejo Nacional de Ciencia y Tecnología de México (CONACyT) y al Instituto Politécnico Nacional (IPN) por el financiamiento recibido, sin el cual esta tesis no seria posible.

Agradezco a mis directores, Dr. Rolando Menchaca Méndez y Dr. Mario Eduardo Rivero Ángeles, por su constante apoyo, enseñanzas y paciencia, por que gracias a su guía muchas ideas se hicieron claras y por que sin su apoyo esta Tesis no hubiese sido posible.
Finalmente, le doy las gracias a todos los profesores que me brindaron su apoyo y orientación, así como a mi familia y amigos que siempre fueron un pilar en el desarrollo de este trabajo.

ÍNDICE

CAPÍTULO 1

INTRODUCCIÓN

Los procesos de optimización pueden encontrarse cotidianamente en nuestras vidas, tal como maximizar beneficios y al contrario minimizar cosas que nos perjudiquen. Un ejemplo de esto es al transportarnos a la escuela o trabajo, donde buscamos minimizar nuestro tiempo de viaje para así maximizar la estancia en casa o la puntualidad en el destino. Pero encontrar la opción óptima no siempre es sencillo, estos problemas pueden escalar de cientos a miles de posibilidades.
Generalmente los problemas de optimización combinatoria tienden a ser difíciles de resolver por lo cual muchos de estos terminan estando dentro de la clase de complejidad *NP-Hard*. Un ejemplo de esto es el *Problema del Agente Viajero (TSP)*. Estos problemas son de gran interés puesto que es poco probable encontrar algoritmos que resuelvan dichos problemas en tiempo polinomial.

Dentro de la complejidad computacional un problema se clasifica en la clase P cuando existe una solución en tiempo polinomial, así mismo, si un problema no tiene una solución en tiempo polinomial pero si se puede verificar en tiempo polinomial, lo situamos dentro de la clase de problemas NP. Un problema X que pertenece a la clase NP que cumple con la propiedad de que cualquier otro problema en NP se puede reducir a él, se dice que pertenece a la clase *NP-Completo* [1].

Por lo tanto, todo problema en P pertenece a NP. Pese a esto aún no se ha determinado la relación exacta entre ambas clases, si $P = NP$ o $P \neq NP$. Partiendo de estas dos afirmaciones, podemos tener enfoques distintos, tal que si suponemos $P = NP$, podemos decir que si un problema X puede ser verificado polinomial-mente, entonces también se puede resolver de forma polinomial; por otro lado, si partimos de $P \neq NP$ entonces existen problemas en NP que son estrictamente más difícil que todo problema en P..

En este contexto, dentro de la optimización combinatoria existe una serie de problemas con aplicaciones interesantes, un ejemplo de esto es el clásico problema del agente viajero, el cuál ya se ha abordado desde un amplio panorama de soluciones, tales como algoritmos voraces (*greedy*), relajaciones de programación lineal, algoritmos genéticos y últimamente utilizando técnicas de inteligencia artificial, tales como el aprendizaje por refuerzo.

Dentro de la complejidad computacional, los problemas como el del agente viajero o el agente viajero múltiple (mTSP, por sus siglas en ingles "multiple Traveling Salesman Problem"), son problemas *NP-Hard* por lo cuál es probable que no haya soluciones en tiempo polinomial *(si suponemos que $P \neq NP$)*, por consiguiente en la búsqueda de encontrar la solución más cercanas a la óptima, recientemente se ha optado por técnicas de aprendizaje máquina como el que se aborda en este trabajo, con el aprendizaje por refuerzo.

Asimismo los problemas TSP y mTSP tienen diversas aplicaciones prácticas, logrando destacar su uso en problemas de enrutamiento de vehículos, transporte y entregas, cableado de computadores, entre otros

Un ejemplo clásico de enrutamiento de vehículos es, si consideramos 28 ciudades donde deben instalarse cabinas telefónicas, y cada cabina debe ser visitada una o dos veces por semana para retirar el dinero y cada jornada laboral dura 445 minutos, comenzando y finalizando en la misma ciudad, el problema es minimizar el numero de días que las cabinas deben ser visitadas así como el tiempo total de viaje. [8] [9] [10]

Si tenemos n ciudades $i(1 \leq i \leq n)$ son visitadas por m jornadas de trabajo, partiendo y regresando al deposito $*$ el tiempo de viaje entre la ciudad i y j es $d_{ij} = d_{ji}$ minutos para $i, j \varepsilon \{1, ..., n\} \cup \{*\}$ [2]

1.1 Planteamiento del problema

El objetivo del problema TSP es determinar el camino mas corto en un conjunto de ciudades (nodos), ordenados de forma aleatoria. Cada ciudad debe ser visitada una única vez, en un camino completamente conectado y regresando a la ciudad inicial o nodo deposito. Aunque este problema resulte ser muy útil en la practica, existen situaciones donde necesitamos otro tipo de aproximación, es aquí donde el problema del agente viajero múltiple (mTSP) resulta de gran utilidad. En el mTSP tenemos un grupo de n agentes viajeros los cuales deben visitar un conjunto de m ciudades, visitando cada ciudad una única vez y por un solo agente viajero. Así mismo, cada vendedor tiene un punto de partida o deposito donde debe regresar al finalizar su trayecto, esto nos da como resultado un conjunto de caminos por los que optó cada vendedor, formando así nuestra solución óptima. Cabe destacar que estos problemas se encuentran dentro de la comple-

jidad *NP-Hard*.

En cuanto a los problemas *NP-Hard* y su complejidad al buscar soluciones que sean óptimas o que se aproximen al óptimo, es necesario buscar con diferentes técnicas y algoritmos soluciones que se acerquen lo suficiente al óptimo para así ser utilizadas en un ambiente práctico. Cabe destacar que en la práctica los requerimientos pueden ser menos complejos que teóricamente, haciendo énfasis en requisitos muy específicos. Actualmente problemas como el agente viajero múltiple (mTSP) tienen distintos enfoques prácticos, por ejemplo en mapeo de rutas para transporte, donde al depender de requerimientos prácticos se puede optar por algoritmos que sean cercanos al óptimo.

No obstante la búsqueda de una solución óptima o muy cercana al óptimo para el problema mTSP es particularmente difícil, por lo cual es importante buscar herramientas que nos permitan alcanzar estas soluciones, desarrollando así algoritmos basados en distintas técnicas como lo son los algoritmos genéticos y en el caso especifico de este trabajo, el aprendizaje por refuerzo.

El problema del agente viajero múltiple se suele formular con una formulación de programación entera con doble índice, para lo cual definimos nuestra variable binaria como.

$$x_{ij} \begin{cases} 1 \text{ si } (i,j) \text{ esta en la solución,} \\ 0 \text{ en otro caso} \end{cases}$$

Con lo cual podemos hacer la siguiente formulación

$$\text{minimizar} \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{1.1}$$

$$\sum_{j=2}^{n} x_{1j} = m \tag{1.2}$$

$$\sum_{j=2}^{n} x_{j1} = m, \tag{1.3}$$

$$\sum_{i=1}^{n} x_{ij} = 1, j = 2, ..., n, \tag{1.4}$$

$$\sum_{j=1}^{n} x_{ij} = 1, i = 2, ..., n, \tag{1.5}$$

$$x_{ij} \in \{0,1\}, \forall (i,j) \in A \tag{1.6}$$

Es importante señalar que en esta tesis se trabajará para el caso de un único viajero, es decir para $m = 1$.

1.2 Objetivo general y objetivos particulares

1.2.1 Objetivo general

Diseñar, implementar y caracterizar un algoritmo de optimización combinatoria basado en las técnicas de *rollout* y *lookahead* multi-paso, para el problema del viajero.

1.2.2 Objetivos específicos

- Realizar una revisión detallada del estado del arte referente a algoritmos para el problema del viajero.
- Implementar alternativas algorítmicas basadas en programación matemática y algoritmos genéticos para el problema del viajero.
- Utilizar las técnicas de *rollout* y *lookahead* multi-paso para implementar un algoritmo escalable y efectivo para el problema del viajero.
- Realizar un análisis comparativo del desempeño de los diferentes algoritmos en conjuntos de prueba estándar.

1.3 Justificación

Ante el creciente desarrollo tecnológico es importante apoyarnos de nuevas técnicas para resolver problemas, particularmente en este caso el problema TSP, buscando una solución más cercana al óptimo que los competidores más cercanos como los algoritmos genéticos y con un menor costo computacional.

En este trabajo se emplea un grupo de técnicas utilizadas en el contexto del aprendizaje por refuerzo para diseñar algoritmos que logren un balance adecuado entre su complejidad temporal y la calidad de las soluciones que generan. Estos algoritmos pueden ser utilizados en aplicaciones de logística para el caso que los conjuntos de entrada sean de tamaño medio a moderadamente grande, es decir de decenas de nodos hasta unas cuantas centenas.

CAPÍTULO 2

MARCO TEÓRICO

2.1 Optimización Combinatoria

El objetivo principal de la optimización es realizar una tarea determinada con el mejor manejo de recursos posible, maximizando o minimizando una función objetivo.
Por lo que ha recibido principal interés de áreas como la computación y matemáticas pero también en el área de la investigación de operaciones.

2.2 Complejidad y problemas NP-Hard

Generalmente la complejidad de los problemas computacionales se mide de acuerdo al tiempo de ejecución, si éste es polinomial o no. En tal sentido que para cada problema existe una *complejidad inherente*, la cuál permite clasificar el problema de acuerdo a su complejidad. No obstante demostrar que para un problema no existe un algoritmo eficiente que lo resuelva, puede volverse considerablemente difícil, principalmente cuando tratamos con problemas de inteligencia artificial, optimización combinatoria, entre otros. Dicho de otro modo, para el caso de los problemas NP-Hard "no conocemos un algoritmo que resuelva estos problemas en tiempo polinomial pero al mismo tiempo no podemos probar que dicho algoritmo no exista"[1]. Aunado a esto, si suponemos que existe un algoritmo polinomial que resuelva un problema de este tipo, puede significar que existe un algoritmo para todos estos problemas.

Si bien es cierto que todo un problema se considera inherentemente difícil si su solución requiere recursos importantes sin importar que algoritmo se utilice para su solución [**?**]. Generalmente cuando hablamos de complejidad computacional, los parámetros que se utilizan están basados en el tiempo de ejecución y los requerimientos de memoria de acuerdo al tamaño del problema.

2.2.1 Reducción Polinómica

Cuando tratamos con problemas *NP* podemos utilizar la técnica de reducción, la cual se basa en comparar la complejidad de dos problemas, suponiendo así, que si tenemos un problema X y este es al menos tan difícil como otro problema Y, por lo tanto si podemos resolver X entonces podemos resolver Y, en otras palabras Y es reducible a X, así mismo si suponemos que X puede ser resuelto en tiempo polinomial, y las llamadas asociadas a X pueden resolver Y en un número polinomial de pasos, podemos decir que *"Y es polinomial-mente reducible de X"*, de esta manera $Y \leq_P X$.
Por lo cual si Y puede resolverse en tiempo polinomial, entonces X también puede resolverse en tiempo polinomial, y viceversa, si Y no puede resolverse en polinómica-mente entonces X tampoco [1].

2.2.2 Problemas intratables

Conforme al tiempo de ejecución de un algoritmo podemos clasificarlo como polinomial o exponencial, partiendo de esto un algoritmo puede ser lo *suficientemente eficiente* o por el contrario *demasiado ineficiente*. Si tenemos un algoritmo cuya complejidad temporal es $O(p(n))$ se puede clasificar como polinomial y todo algoritmo que se encuentre fuera de esta complejidad, se clasifica como *exponencial* o *no polinomial*.
Los algoritmos que solamente tienen soluciones en tiempo exponencial, son considerados como *demasiado ineficientes* dado que este deberá presentar una búsqueda muy grande lo cual en tiempo y coste computacional puede resultar ineficiente e incluso intratable.

Cuando se trata de intratabilidad computacional, es importante definir la caracterización del problema, así como su reducibilidad, por ende, si definimos la entrada de un problema computacional como una cadena binaria finita s con una longitud $|s|$ y teniendo un problema de decisión X con un conjunto de cadenas, en el cual la respuesta es *"si"*, definimos un algoritmo A el cuál tiene como entrada las cadenas s y retorna el valor *"Si"* ó *"No"*, entonces tendremos $A(s)$, si decimos que A resuelve X para todas las cadenas s entonces $A(s) = "Si"$ si y solo si $s \in X$.
Ahora bien decimos que A es un algoritmo que se ejecuta en tiempo polinomial, si para cada cadena de entrada s existe una función polinomial $p(\cdot)$ de modo que para cada cadena de entrada s, el algoritmo A termina en s en un máximo de $O(p(|s|))$ pasos [1].

Cabe considerar, por otra parte, la comprobación eficiente, con la cual podemos corroborar si existe o no una solución en tiempo polinomial, utilizando un algoritmo comprobador B donde tenemos una cadena t que contiene la evidencia de que s es una instancia de S de X. Siempre y cuando se cumpla que B sea un algoritmo con ejecución en tiempo polinomial que toma como entrada s y t, además existe una función polinomial p tal que para cada cadenas, tenemos $s \in X$ si y solo si existe una cadena t tal que $|t|p(|s|)$ y $B(s,t) =$ Sí [1].

2.2.3 Problemas *NP* y *NP-Completo*

Generalmente cuando se busca una cadena t para que la comprobación eficiente acepte la entrada s, se trata de una búsqueda no determinista, siendo así como recibe su nombre la clase NP partiendo del acrónimo en ingles *"nondeterministic polynomial time"*.

Podemos así definir la clase NP como el conjunto de todos los problemas que tienen una comprobación eficiente, por lo tanto, podemos decir que $P \subseteq NP$.

En consecuencia es difícil identificar si todos los problemas en NP no pertenecen a P, generando así una de las mayores preguntas en el área de algoritmos y computación, y es si:

$$P = NP \text{ ó } P \neq NP$$

Dado que resulta difícil pensar que NP sea igual a P, se suele trabajar con la segunda hipótesis ($P \neq NP$).

Por consiguiente resulta interesante buscar cuáles son los problemas *más difíciles* en NP, los cuales podemos definir, tomando en cuenta las siguientes dos propiedades.

(i) $X \in NP$

(ii) $\forall Y \in NP, Y \leq_P X$

Como resultado si asumimos que X es un problema *NP-Completo* entonces X tiene una solución polinómica, si y solo si $P = NP$, por el contrario si cualquier problema que se encuentre en *NP-Completo* no puede resolverse en tiempo polinomial, entonces ningún problema en *NP-Completo* puede resolverse en tiempo polinomial.
Sin duda buscar problemas que se encuentren en *NP-Completo* resulta interesante y si suponemos que $P \leq NP$ nos ayuda a saber que dicho problema no tiene una solución en tiempo polinomial, para esto podemos optar por la siguiente estrategia propuesta en [1], si suponemos un problema X

1. Probar que $X \in NP$
2. Escoger un problema Y que se sepa que es *NP-Completo*
3. Probar que $Y \leq_P X$

2.2.4 Máquinas de Turing

La máquina de Turing es uno de los conceptos fundamentales dentro de la computación, dado que todo computo se puede expresar como una secuencia finita o infinita ordenada, donde un estado y símbolo avanzan cambiando de estado cuando se escribe un nuevo símbolo.

2.2.4.1 Máquina de Turing Determinista

Una máquina de Turing se considera Determinista cuando para cada par de estado-símbolo, solo existe una posibilidad de ejecución. Si M es una maquina de Turing determinista y x es una palabra de entrada, entonces M y x siguen un único camino computacional [**?**].

2.2.4.2 Máquina de Turing No -Determinista

Si M es una *Máquina de Turing No-Determinista* y x es una palabra de entrada, entonces M y x especifican un árbol computacional donde la raíz es la configuración inicial de M, y los hijos de un nodo son las configuraciones que se ejecutan en un movimiento. [**?**]

2.3 Programación Dinámica

La programación lineal se basa en la colección de soluciones de una serie de subproblemas que provienen del problema original las cuales deben satisfacer las siguientes características:

- La solución original se puede calcular fácilmente partiendo de la solución a los subproblemas.
- Existe solamente un numero polinomial de sub problemas.

2.4 Programación lineal

La programación lineal se encarga de buscar soluciones óptimas, asignando de la mejor manera los recursos limitados en un problema, buscando una solución que logre minimizar costos o maximizar ganancias [**?**].

Los problemas de programación lineal cubren las siguientes condiciones:

- Sus variables de decisión se puede describir como una función lineal, la cual recibe el nombre de *función objetivo*.
- Las alternativas de soluciones están regidas por un conjunto de ecuaciones o desigualdades lineales también conocidas como restricciones del problema.

Aunado a esto, la programación lineal tiene diversas aplicaciones y puede resultar interesante su aplicación en problemas de optimización combinatoria, al buscar la solución mas cercana al óptimo.

2.5 Algoritmos Avaros (Greedy)

Las heurísticas son algoritmos que están diseñados para un problema en especifico, dado que se emplean métodos prácticos y específicos en cada problema, por lo cual una de sus mayores limitantes es el no garantizar la solución mas optima. Una de las heurísticas mas utilizadas son los algoritmos avaros (Greedy).

Los Algoritmos Avaros (Greedy) son heurísticas que se caracterizan por hacer uso de los datos disponibles del problema para construir una solución paso a paso, esta búsqueda puede tomar distintos criterios pero siempre utilizando la mejor opción que tenga mas cerca, de aquí reciben el nombre de avaros, dado que siempre buscaran el beneficio mas cercano.

Para comenzar a buscar la solución a un problema es común optar por soluciones que no nos garanticen una solución óptima, en este punto los algoritmos avaros (greedy) pueden volverse de gran ayuda.

2.6 Algoritmos evolutivos: fundamentos y limitantes

Debido a la complejidad de tratar problemas NP-Hard y la necesidad de encontrar soluciones óptimas o que nos den el mayor costo-beneficio, se puede optar por diversas técnicas, entre ellas se encuentran los algoritmos de propósito general también conocidos como meta-heurísticas, atribuyéndoles este nombre al ser algoritmos que generalmente se desempeñan con mayor eficacia que las heurísticas clásicas; así mismo existe una gran variedad de algoritmos de propósito general, muchos de estos inspirados en la naturaleza, e.g. principios de evolución, fenómenos físicos, comportamiento animal, entre otros [3].

Concretizando las Meta-heurísticas son una clase de métodos aproximados, diseñados para tratar problemas de optimización combinatoria difíciles, guiados por una heurísticas subordinada combinando diferentes conceptos para explorar y explotar los espacios de búsqueda, con el fin de encontrar de manera eficiente soluciones cercanas al óptimo.[4]

CAPÍTULO 3

TÉCNICAS DE APRENDIZAJE POR REFUERZO PARA LA SOLUCIÓN DE PROBLEMA COMBINATORIOS

La programación dinámica (DP-Dynamic Programing), es utilizada principalmente para la solución de problemas de optimización gracias a que se basa en reducir el tiempo de ejecución de un algoritmo conforme a la resolución de subproblemas que pueden ser utilizados para resolver todo el problema. Un ejemplo de esto es el subproblema de la cola, del cuál se hablara más a fondo en este capitulo. Así mismo resulta interesante la sinergia que tiene la programación dinámica con el aprendizaje por refuerzo para la aproximación de soluciones óptima a problemas complejos, como podremos analizarlo a continuación.

3.1 Programación Dinámica Determinista

Dentro de la programación dinámica, un primer acercamiento puede darse a través de problemas deterministas, tomando en cuenta que todos los problemas de programación dinámica implican un sistema dinámico de tiempo discreto, generando una secuencia de estados bajo una política de control; en el caso de problemas de horizonte finito, el sistema avanza conforme a un numero finito N de *estados*, estos estados se denotan en un tiempo k como x_k y con un control como u_k. [6]

3.1.1 Problemas Deterministas

Los problemas de programación dinámica deterministas involucran un sistema de tiempo discreto de la siguiente forma:

$$x_{k+1} = f_k(x_k, u_k), k = 0, 1, ..., N-1 \tag{3.1}$$

Donde:
k es el índice de tiempo,
x_k es el estado del sistema,
u_k es la variable de control a ser seleccionada en un tiempo k de algún conjunto $U_k(x_k)$ dado que depende en x_k,
f_k es una función de (x_k, u_k) que describe el mecanismo por el cual el el estado se actualiza desde el tiempo k hasta el tiempo $k+1$,
N es el horizonte o número de veces que se aplica el control. [6]
Hay que mencionar además que tenemos el conjunto de todo los posibles x_k el cuál es el espacio de estados en tiempo k. De modo que puede ser cualquier conjunto y puede depender de k. De manera análoga tenemos el espacio de control en tiempo k, siendo este, el conjunto de todos los u_k posibles.

3.1.2 Principio de Optimalidad

Principio de Optimalidad

Sea $\{u_0^*, ..., u_{N-1}^*\}$ una secuencia óptima de control, que junto con x_0 determina la secuencia de estados correspondiente $\{x_1^*, ..., x_N^*\}$ a través de la ecuación (3.1). Considerando el subproblema por el cual comenzamos en x en el tiempo k y deseamos minimizar el *"costo por recorrer"* desde el tempo k hasta el tiempo N,

$$g_k(x_k^*, u_k) + \sum_{m=k+1}^{N-1} g_m(x_m, u_m) + g_N(x_N),$$

sobre $\{u_k, ..., u_{N-1}\}$ con $u_m \in U_m(x_m), m = k, ..., N-1$, Entonces la secuencia de control óptima truncada $\{u_k^*, ..., u_{N-1}^*\}$ es óptima para este subproblema.

El subproblema utilizado en el *Principio de Optimalidad* es conocido como el *subproblema de la cola ó "tail subproblem" en ingles*, el cuál comienza en x_k^*. Ahora bien, por el *Principio de Optimalidad* sabemos que *"La cola de una secuencia óptima es óptima para el subproblema de la cola"*. [6]

Es por esto que si la secuencia de control truncada $\{u_k, ..., u_{N-1}\}$ no es óptima como se supuso en un principio, podemos reducir el costo, cambiando a una secuencia óptima para el subproblema, una vez que lleguemos a x_k^*ya que las elecciones anteriores de control $\{u_0^*, ..., u_{k-1*}\}$, no son restrictivas para futuras decisiones.

Teniendo en cuenta, el *Principio de Optimalidad*, el costo óptimo de una función puede ser construido de adelante hacía atrás, de forma que primero se compute la función de costo óptimo para el *subproblema de la cola*, que involucra el ultimo estado, después se resuelva el subprob-

lema de la cola, que involucra los dos últimos estados y así sucesivamente hasta construir la *función de costo óptima* para todo el problema.

3.1.3 Algoritmo de Programación Dinámica

El algoritmo de programación dinámica se basa en el *Principio de Optimalidad*, resolviendo todos los subproblemas de cola de un periodo de tiempo, usando la solución de dichos problemas en un periodo más corto de tiempo.

Algoritmo de PD para Problemas Deterministas de Horizonte Finito

Comenzamos con

$$J_N^*(x_N) = g_N(x_N), \qquad \forall x_N, \tag{3.2}$$

y para $k = 0,...,N-1$, sea

$$J_k^*(x_k) = \min_{u_k \in U_k(x_k)} [g_k(x_k,u_k) + J_{k+1}^*(f_k(x_k,u_k))], \qquad \forall x_k. \tag{3.3}$$

Ahora bien, es necesario que se calculen en (3.3) todos los estados x_k para el *tiempo* k, antes de pasar al *tiempo* $k-1$. El punto clave del algoritmo de Programación Dinámica, es que para cada estado inicial x_0 el número $J_0^*(x_0)$ obtenido en el ultimo paso, es igual al costo óptimo $J^*(x_0)$. Por lo cual se puede demostrar de forma más general, que *para todo* $k = 0,1,...,N-1$, y todos los estados x_k en el tiempo k, tenemos que

$$J_k^*(x_k) = \min_{\substack{u_m \in U_m(x_m) \\ m=k,...,N-1}} J(x_k;u_k,...,u_{N-1)}, \tag{3.4}$$

donde:

$$J(x_k;u_k,...,u_{N-1}) = g_N(x_N) + \sum_{m=k}^{N-1} g_m(x_m,u_m), \tag{3.5}$$

es decir que $J_k^*(x_k)$ es el costo óptimo del subproblema de cola que comienza en el estado x_k y tiempo k para el periodo de tiempo $(N-k)$ y que termina en tiempo N. Por consiguiente $J_k^*(x_k)$ puede nombrarse como el costo óptimo de ir del estado x_k y tiempo k, y J_k^* como la función de costo óptimo en el tiempo k. [6]

El algoritmo de PD resuelve cada uno de los subproblemas de cola, por lo cual se puede decir que es la minimización del coste acumulado a partir de un estado intermedio hasta el final del horizonte.

Partiendo de las funciones $J_0^*,...,J_N^*$, podemos utilizar el siguiente algoritmo para construir una secuencia de control óptima $\{u_0^*,...,u_{N-1}^*\}$ y su trayectoria de estados correspondiente $\{x_1^*,...,x_N^*\}$ para el estado inicial dado x_0.

Construcción de una Secuencia de Control Óptima

Sea

$$u_0^* \in \arg \min_{u_0 \in U_0(x_0)} [g_0(x_0, u_0) + J_1^*(f_0(x_0, u_0))],$$

y

$$x_1^* = f_0(x_0, u_0^*).$$

Secuencialmente, yendo hacia adelante, para $k = 1, 2, ..., N-1$, sea

$$u_k^* \in arg \min_{u_k \in U_k(x_k^*)} [g_k(x_k^*, u_k) + J_{k+1}^*(f_k(x_k^*, u_k))], \tag{3.6}$$

y

$$x_{k+1}^* = f_k(x_k^*, u_k^*)$$

3.1.4 Aproximación en espacio valor

Construir una *Secuencia de Control Óptima* solo es posible cuando ya hemos calculado $J_k^*(x_k)$ por programación dinámica para todo x_k y k. Teóricamente puede resultar factible, pero en la practica, esto puede resultar relativamente imposible, considerando que el tiempo puede ser demasiado grande con base en el numero de posibles x_k por lo cuál k puede ser muy largo. Para estos casos podemos utilizar una solución algorítmica similar remplazando J_k^* por una aproximación, denotada como $\widetilde{J_k}$. Siendo esta la base para la *Aproximación en espacio valor*, la cual construye una solución sub-óptima $\{\widetilde{u}_0, ..., \widetilde{u}_{N-1}\}$, basándonos en reemplazar J_k^* por $\widetilde{J_k}$ en la ecuación (3.5), podemos obtener lo siguiente.

Aproximación en el Espacio Valor

Utilizando $\widetilde{J_k}$ en vez de J_k^*

Comenzamos con

$$\widetilde{u}_0 \in \arg \min_{u_0 \in U_0(x_0)} [g_0(x_0, u_0) + \widetilde{J}_1(f_0(x_0, u_0))],$$

y sea

$$\widetilde{x}_1 = f_0(x_0, \widetilde{u}_0).$$

Secuencialmente, yendo hacia adelante, para $k = 1, 2, ..., N-1$, sea

$$\widetilde{u}_k \in arg \min_{u_k \in U_k(x_k^*)} [g_k(\widetilde{x}_k, u_k) + \widetilde{J}_{k+1}(f_k(\widetilde{x}_k, u_k))], \tag{3.7}$$

y

$$\widetilde{x}_{k+1} = f_k(\widetilde{x}_k, \widetilde{u}_k)$$

3.2 Problemas deterministas de la ruta más corta (shortest path)

Existe una gran variedad de problemas que pueden emplear la *Programación Dinámica* para minimizar el coste y optimizar su solución, uno de ellos es el problema del agente viajero, puesto que tiene las características necesarias para que su resolución con programación dinámica resulte interesante, hay que mencionar, además, que existen otros problemas y variaciones donde se busca encontrar la ruta más corta, por consecuencia tomaremos un problema determinista de donde se busca encontrar su *"shortest path"* ó ruta más corta para emplear el algoritmo de *Programación Dinámica*. [6]

Supongamos que tenemos un grafo dirigido con un nodo especial también llamado destino, entonces tendremos el conjunto de nodos $\{1,2,...,N,t\}$ donde el nodo t es el destino y siendo a_{ij} el costo de moverse del nodo i al nodo$j$, necesitamos encontrar el camino más corto desde el nodo i al nodo j, siendo este una secuencia de aristas, tal que el nodo final de cada arista en la secuencia es el nodo inicial de la siguiente arista y la longitud del camino es la suma de los costos de las aristas que lo conforman.

Con fines prácticos, asumiremos que no existen ciclos negativos en el grafo, así como que la distancia de un nodo i al mismo nodo i sea igual a cero, o dicho de otra forma $a_{ii} = 0$, así mismo es necesario asumir que para cada nodo i existe un camino de i a t.
Podemos formular el problema como un problema determinista de Programación Dinámica con N etapas, donde los estados de cada etapa $0,...,N-1$ son los nodos de $1,..,N$, con el nodo t como destino, siendo este el único estado en N, y siendo el control las *aristas (i,j)*, tal que para cada estado i se selecciona un control (i,j) con un costo a_{ij}.[6]
Podemos así escribir el *Algoritmo de Programación Dinámica* para este problema con la función oprima de costo J_k^* de la siguiente forma

$$J_k^*(i) = \text{costo optimo de ir de } i \text{ a } j \text{en } N-k \text{ movimientos.}$$

para $i = 1,...,N$ y $k = 0,...,N-1$. El costo del camino óptimo de i a j es $J_0^*(i)$, que visto desde el algoritmo de PD es:

$$J_k^*(i) = \min_{\text{todas las aristas}(i,j)} [a_{ij} + J_{k+1}^*(j)], \qquad k = 0,1,...,N-2,$$

con

$$J_{N-1}^*(i) = a_{it}, \qquad i = 1,2,...,N.$$

Este algoritmo tambíen es conocido como el *Algoritmo Bellman-Ford para caminos cortos*.

3.3 Optimización determinista discreta

Los problemas de optimización discretos se pueden formular típicamente como problemas de *Programación Dinámica* dividiendo cada solución factible en una secuencia de decisiones/controles; Aunque esta formulación resulta en un calculo intratable cuando existe una gran cantidad de estados ya que estos crecen de forma exponencial, por ende es común utilizar métodos de aproximación, como ejemplo podemos tomar el problema del agente viajero (TSP). [6]

En el TSP existen una cantidad N de ciudades y un coste de viaje entre ciudad y ciudad, donde se busca encontrar el camino con el menor costo de viaje. Para formular este problema en un problema de Programación Dinámica, consideramos un grafo completo dirigido, por lo cual para cada nodo existe una arista hacia todo los nodos de $k = 1,...,N$ y una etapa de decisión por cada estado, haciendo que el problema crezca exponencialmente, por lo cuál es necesario optimizarlo para encontrar una solución óptima.

De una forma más general, la formulación de un poblema de optimización discreta determinista es la siguiente

$$\text{minimizar } G(u) \qquad \text{sujeto a} u \in U,$$

donde U es un conjunto finito de soluciones factibles y $G(u)$ es la funcion de costo. Si suponemos que cada solución u tiene N componente, entonces queda de la siguiente forma $u = (u_1,...u_N)$, donde N es un entero positivo. Ahora entonces podemos ver el problema como un problema de decisión secuencial, donde los componentes $u_1,...u_N$ son seleccionados solamente una vez. Una $k-$tupla $(u_1,...,u_k)$ consiste en los primeros k componentes de una solución llamada $k-$solución y se asocian las $k-$soluciones con la k etapa para $k = 1,...,N$ quedando de la siguiente forma.

$$U_{k+1}(u_1,...,u_k) = \{\widetilde{u}_k + 1 | \text{existe una solución de la forma} (u_1,...,u_k,\widetilde{u}_{k+1},...,\widetilde{u}_N \in U\}$$

3.4 Problemas con un estado de termino

Es común encontrarnos con problemas con un estado de termino o estado final, el cual podemos considerar como un estado t que es libre de costo y absorbente de tal forma que

$$g_k(t,u_k,w_k) = 0, \qquad f_k(t,u_k,w_k) = t, \qquad \forall u_k \in U_k(t), k = 0,1,...$$

De esta forma el proceso de control termina cuando se llega a t incluso cuando esto sucede antes de llegar al final del horizonte. Aunque cabe destacar que también se puede llegar a t por elección si se dispone de una decisión de parada especial, o por medio de una transición desde otro estado. [6]

3.5 Problemas generales de aproximación en el espacio valor

En cuestión del espacio valor, tenemos dos principales problemas.

- Obtener $\widetilde{J}_k$: de a cuerdo al método de computar las funciones *lookahead* $\widetilde{J}_k$ que están involucradas en la minimización *lookahead*.
- Selección de control: el método para realizar la minimización e implementar la política subóptima $\widetilde{\mu}_k$

3.5.1 Métodos para calcular aproximaciones en el espacio valor

El cálculo de $\widetilde{J}_k$ se puede hacer de 4 formas:

- *Aproximación del problema*: Las funciones $\widetilde{J}_k$ en la ecuación de aproximación en el espacio valor (3.7) se obtienen como las funciones de costo óptimas o casi óptimas de un problema de optimización simplificado. Las simplificaciones pueden incluir explotar la estructura que se puede descomponer, ignorar varios tipos de incertidumbres y reducir el tamaño del espacio de estado.
- *Optimización aproximada "On-line"*: Estos métodos a menudo involucran el uso de una política sub-óptima o heurística, que se aplica "On-line" cuando es necesario para aproximar los valores óptimos verdaderos de costo de uso.
- *Aproximación de costos paramétricos*:Las funciones $\widetilde{J}_k$ en la ecuación de aproximación en el espacio valor (3.7) se obtienen a partir de una clase paramétrica dada por las funciones $\widetilde{J}_k(x_k, r_k)$ donde r_k es un vector de parámetros, seleccionado por un algoritmo adecuado.
- *Agregación*: En este caso las funciones $\widetilde{J}_k$ se derivan de las funciones de costo óptimas del problema agregado. La partición del espacio de estado puede ser arbitraria, pero a menudo se determina mediante el uso de características (los estados con características "similares" se agrupan). Además, la agregación se puede combinar de manera complementaria con los métodos anteriores, y se puede utilizar como punto de partida una función aproximada de costos por recorrer producida por cualquiera de estos métodos.

[6]

3.6 Multistep Lookahead

El sistema *Multistep Lookahead* se basa en la el control que se obtiene de la minimización, podemos ilustrarlo con el caso de *"two-step lookahead"* en tiempo k y estado x_k

$$\widetilde{\mu}_k(x_k) \in arg \min_{u_k \in U_k(x_k)} E\{g_k(x_k, u_k, w_k) + \widetilde{J}_{k+1}(f_k(x_k, u_k, w_k))\},$$

donde para todos los posibles estados x_{k+1} que se pueden generar por el sistema de ecuaciones, partiendo de x_k donde

$$\widetilde{J}_{k+1}(x_{k+1}) = \min_{u_{k+1} \in U_{k+1}(x_{k+1})} E\{g_{k+1}(x_{k+1}, u_{k+1}, w_{k+1}) + \widetilde{J}_{k+2}(f_{k+1}(x_{k+1}, u_{k+1}, w_{k+1}))\},$$

donde $\widetilde{J}_{k+2}$ es una aproximación de la función óptima de costo para ir J^*_{k+2}.
De esta forma también podemos definir políticas con un lookahead de $\ell > 2$ etapas, con un estado inicial x_k y $\widetilde{J}_{k+l}$ como la función de costo terminal. [6] [7] [11] [**?**]

3.6.1 Multistep Lookahead y horizonte "Rolling"

Calcular funciones lookahead $\widetilde{J}_{k+\ell}$ en $\ell - pasos$ puede resultar muy similar a calcular *one-step lookahead*, pero si tenemos un lookahead lo suficientemente largo podemos capturar suficiente del problema de programación dinámica en cuestión, para asñi no necesitar de $\widetilde{J}_{k+\ell}$.
En particular podemos decir que $\widetilde{J}_{k+l}(x_{k+l}) \equiv 0$, o decimos que

$$\widetilde{J}_{k+\ell}(x_{k+\ell}) = g_N(x_{k+\ell}).$$

La idea es utilizar un número lo suficientemente grande de pasos lookahead ℓ para garantizar una aproximación razonablemente fiel de $Q-$factores Q_k o funciones de costo de ir $J^*_{k+\ell}$, también conocido como el *rollong horizon approach*.
Cabe destacar que los sistemas de *rolling horizon* tienden a funcionar bien si la distribución de probabilidad del estado $k + \ell$ pasos adelante, es aproximadamente independiente del estado y control actual.[7]

Hay que mencionar, además, que a medida que el tamaño de ℓ del lookahead se elije más grande, la necesidad de una buena elección de $\widetilde{J}_{k+\ell}$ tiende a disminuir. Esto sucede dado que la aproximación efectiva de la función costo de ir en ℓ-paso lookahead consta se dos componentes:

- el costo de un problema de $(\ell - 1)$ paso, que involucra las últimas $(\ell - 1)$ etapas de la anticipación de ℓ pasos.
- la aproximación del costo terminal $\widetilde{J}_{k+\ell}$

3.7 Rollout

La *política de rollout* se basa en el uso de una heurística o política suboptima, también conocida como *política base*, la cual se busca mejorar partiendo de la minimización lookahead, utilizando la heurística al final [6][5] [13] [14]

En el caso de *one-step lookahead* la implementación de la política rollout resulta de cierta forma mas sencilla, ya que es una aproximación en el espacio valor con los valores aproximados

de la funci'on costo de ir $\widetilde{J}_{k+1}(x_{k+1})$, calculándose con la política base, partiendo de casa posible estado x_{k+1}. En el caso de problemas de horizonte largo, la ejecución de la política base puede truncarse, es decir, puede usarse para un número limitado de pasos, con alguna aproximación de función de costo en el final para tener en cuenta el costo de los pasos restantes.

3.7.1 On-Line Rollout para problemas deterministas de estado finito

Si consideramos un problema de programacion dinamica determinista con un numero finito de control y con un estado inicial x_k en tiempo k, el rollout considera todos los subproblemas de cola que comienzan en todos los siguientes estados posibles x_{k+1}, y los resuelve de forma suboptima utilizando la heurística base. Por lo tanto cuando se encuentra en x_k, *rollout* genera *on-line* los siguientes estados x_{k+1} que corresponden a todo $u_k \in U_k(x_k)$, y utiliza la heuristica base para calcular la secuencia de estados $\{x_{k+1},...,x_N\}$ y controles $\{u_{k+1},...,u_{N-1}\}$ tal que

$$x_{i+1} = f_i(x_i, u_i), \qquad i = k,...,N-1$$

Entonces el algoritmo de rollout aplica el control que minimiza sobre $u_k \in U_k(x_k)$ la expresión del costo de cola para las etapas k a la N

$$g_k(x_k, u_k) + g_{k+1}(x_{k+1}, u_{k+1}) + ... + g_{N-1}(x_{N-1}, u_{N-1}) + g_N(x_N). \tag{3.8}$$

CAPÍTULO 4

TRABAJO RELACIONADO

4.1 Introducción

La técnica propuesta en esta tesis consiste en utilizar el *"rollout"* y *"k-steps lookahead"* en conjunto, para dar una solución menos lejana al óptimo para el problema del agente viajero ó *Traveling Salesman Problem (TSP)* siendo este uno de los problemas más estudiados en optimización, el cuál se basa en encontrar la ruta más corta por la que puede optar un agente viajero, acorde a un conjunto de ciudades o nodos, conectadas por aristas, visitando cada ciudad una única vez.

Considerando que el *TSP* es un problema clásico y estudiado desde distintos enfoques, en este capitulo se pretende dar a conocer el estado del arte, partiendo de algoritmos ya propuestos para el problema del agente viajero y que cuentan con soluciones cercanas al óptimo en mayor o menor medida, así como las ventajas y desventajas que estos pueden tener.

Es importante destacar, que en el presente trabajo nos enfocaremos en grafos completos, para los cuales existen distintos ciclos hamiltonianos, de los cuales se busca encontrar la solución menos lejana al óptimo.

4.2 Formulación de programación lineal del problema del agente viajero

El funcionamiento de la mayoría de los algoritmos con relación a la solución de problemas de optimización combinatoria decrece al incrementar el número de instancias, haciendo así que el tiempo para encontrar una solución óptima, se vea incrementado en gran medida. Dicha

situación presenta una problemática para algoritmos como el de programación entera, que nos pueden dar una solución óptima a un gran coste computacional para problemas muy grandes.

Para esto es necesario destacar que para la formulación de *Programación Lineal Entera* podemos encontrar distintas formulaciones, utilizando en el presente trabajo la propuesta por *Miller, Tucker y Zemlin (MTZ)* en 1960 [15].

Si suponemos un conjunto de ciudades $\{1,2,...,n\}$, el costo de viaje de la ciudad i a la ciudad j es c_{ij}

Apoyándonos de una decisión binaria y considerando un tour dado t

$$x_{ij} = \begin{cases} 1 \; si \; el \; agente \; viaja \; de \; la \; ciudad \; i \; a \; la \; ciudad \; j \\ 0 \; otro \; caso \end{cases} \tag{4.1}$$

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{4.2}$$

$$\min \sum_{i=1}^{n} x_{ij} = 1 \qquad j = 1,2,...,n \tag{4.3}$$

$$\min \sum_{j=1}^{n} x_{ij} = 1 \qquad j = 1,2,...,n \tag{4.4}$$

$$u_i - u_j + x_{ij} \leq n-1, \qquad 2 \leq i \neq j \leq n, \tag{4.5}$$

$$x_{ij} \in \{0,1\} \qquad i,j = 1,2,...,n \qquad i \neq j \tag{4.6}$$

Ahora bien con las restricciones (4.3) y (4.4) definimos que de cada ciudad se parta y se llegue una sola vez, y con las restricciones (4.5) y (4.6) evitamos que se formen subtours.
Esta formulación es general, por lo cuál existen muchas versiones, con el fin de evitar la formación se *subtours*, a su vez el TSP tiene variaciones como el *problema del agente viajero múltiple (mTSP)* entre otros, y para cada uno existen formulaciones distintas que parten de la función objetivo de minimizar costos.

4.3 Algoritmo Avaro (Greedy)

La heuristica greedy o algoritmo avaro es una de las más utilizadas para generar soluciones a problemas de optimización, sin embargó también es una de las menos exactas, dado que nos brinda soluciones que pueden encontrarse más lejanas al óptimo, esto a cambio de menor tiempo de ejecución; al emplear una estrategia avara, el coste computacional se reduce en gran medida.

Esta técnica consiste en dos etapas, una de búsqueda global y otra de búsqueda local, en la

primer etapa se busca dentro del espacio de soluciones, aquellas que puedan ser potencialmente buenas, para posteriormente hacer una búsqueda local, al tratarse de una búsqueda avara, esta heurística puede quedarse atrapada en un óptimo local, dado que no se guarda memoria de los pasos realizados, toma decisiones locales con lo que tiene en el momento, dando soluciones que pueden no resultar óptimas, sin embargo su fácil implementación la hace una opción clara al tratar con problemas de optimización.

Algorithm 1: TSP-Greedy

Input: Numero de ciudades n, un arreglo de costos c(i,j) con i,j=1,...,n y un nodo deposito
Output: Camino de ciudades y costo total

```
C=0
costo=0
e=1 (e= nodo actual)
for r = 1 to n-1 do
    encojé un punto j con minimo=c(e,j)
    costo=costo+minimo
    e=j
    C(r)=j
costo=costo+c(e,1)
```

4.4 Metaheuristica Colonia de Hormigas

Las metaheuristicas son constantemente empleadas en problemas de optimización, en este caso se opto por la metaheuristica conococida como Colonia de hormigas o *Ant Colony System (ACS)*.

El algoritmo de colonias de hormigas emula la conducta de las hormigas para generar senderos entre su nido y fuentes de ingesta de alimentos. Una vez que las hormigas se desplazan por el camino, dejan un rastro de feromonas que otras tienen posibilidad de seguir otras hormigas, sin embargo dicho rastro se va evaporando posterior a su depósito. Por consiguiente los senderos más transitados por las hormigas son donde la feromona perdura más en el tiempo. Un camino largo, necesita un largo tiempo para ser recorrido y la feromona se evapora a una tasa más alta que su deposito. Por contrario, en senderos cortos la feromona se deposita con mayor rapidez que su evaporación. De esta forma, con el tiempo el camino entre el nido y la fuente de alimento llega a ser mínimo.

Para el problema del agente viajero se utiliza como datos de entrada la cantidad de hormigas de la colonia, el número de ciudades y las distancias entre estas.

Algorithm 2: TSP-Colina de Hormigas

```
begin
    t ⟵ 1
    Inicializa el valor de las feromonas
    while no termine la condición do
        for k=1,2,...,M do
            for n=1,2,...,N do
                genera un tour
                Se actualiza la feromona local
        Se actualiza la fermonoa global
        t ⟵ t + 1
```

CAPÍTULO 5

PROPUESTA DE SOLUCIÓN

5.1 Introducción

En los capítulos anteriores se han abordado los conceptos básicos para comprender la propuesta de solución al problema presente planteado, abordando también algoritmos que pueden servir como competidores, acorde a los algoritmos que ya han sido planteados.

Como hemos visto en los capítulos anteriores, el problema del viajero ha sido resuelto empleando distintos tipos de algoritmos, con la finalidad de encontrar el que se adapte mejor al problema; para el presente trabajo se emplean técnicas de *rollout* y *k-steps lookahead* en especifico *2-steps lookahead*.

Si generalizamos el problema y consideramos un agente viajero, el cual tiene que visitar cada ciudad de un conjunto de ciudades c=0,1,...,N-1 y regresando exactamente a la ciudad de donde partió (la cuál llamamos deposito), considerando que se trabaja con grafos completos, cada ciudad esta totalmente conectada a las demás ciudades, por lo cual la distancia entre una ciudad i y una ciudad j es c_{ij}.

CAPÍTULO 6

RESULTADOS EXPERIMENTALES

En el presente capitulo, se analizaran 3 algoritmos como principales competidores del algoritmo propuesto, así como la metodología empleada para su ejecución, utilizando los *benchmarks* de la librería *TSPLIB*, la cual nos permite evaluar el problema *TSP* con distintos tamaños de ciudades, logrando así diferenciar tiempos y costes de cada algoritmo, reportando los resultados en una tabla comparativa.

6.1 Metodología

Con la finalidad de poder tener distintas perspectivas, la experimentación se hizo con archivos *.tsp*, ya que la librería *TSPLIB* cuenta con gran cantidad de problemas y sus soluciones óptimas para el problema del viajero, en especifico se utilizo un conjunto de 11 problemas, elegidos arbitrariamente.

Comparativa de Algoritmos								
2*TSPLIB	Programación Lineal		Greedy		Rollout k-Steps lookahead		Colonia de hormigas	
	Costo	Tiempo	Costo	Tiempo	Costo	Tiempo	Costo	Tiempo
bayg29	1610	19.9s	2031.6	0.00128s	1752	0.452343s	2212	6.4688s
bays29	2020	167s	2365.8	0.001601s	2020	2.34864s	2212	5.8159s
ulysses16	6859	353s	8934	0.00253s	7598	324s	7544	85.342s

CAPÍTULO 7

CONCLUSIONES Y TRABAJO A FUTURO

7.1 Conclusiones

En el presente trabajo, se propone un algoritmo basado en técnicas de *rollout* y *k-steps lookahead*, el cual permite mejorar el coste y acercarnos más a una solución óptima, con tiempos de computo razonables. Mejorando los tiempos en comparación con algoritmos de programación entera y mejorando el coste de heurísticas, en especifico de las heurística greedy.

7.2 Trabajo a futuro

Considerando los resultados para el problema del agente viajero, podemos considerar el diguiente trabajo a futuro:

- Extensión del algoritmo para el mTSP y para el mTSP con múltiples depots.
- Paralelización del algoritmo. Cada rollout se puede hacer independiente.
- Aplicación de estas técnica a otros problemas algorítmicos

REFERENCIAS

[1] KLEIMBERG, JON y TARDOS, ÉVA, *Algorithm Design*

[2] J K LENSTRA y RINNOY KAN,

[3] NERI FERRANTE *Handbook of Memetic Algorithms*

[4] J.P. KELLY *Meta-Heuristic: Theory and Applications*

[5] Dimitrakakis, Christos, and Michail G. Lagoudakis. "Algorithms and bounds for rollout sampling approximate policy iteration." In European Workshop on Reinforcement Learning, pp. 27-40. Springer, Berlin, Heidelberg, 2008.

[6] Bertsekas, Dimitri P., John N. Tsitsiklis, and Cynara Wu. "Rollout algorithms for combinatorial optimization." Journal of Heuristics 3, no. 3 (1997): 245-262.

[7] Efroni, Yonathan, Gal Dalal, Bruno Scherrer, and Shie Mannor. "Multiple-step greedy policies in approximate and online reinforcement learning." Advances in Neural Information Processing Systems 31 (2018).

[8] Cheikhrouhou, Omar, and Ines Khoufi. "A comprehensive survey on the Multiple Traveling Salesman Problem: Applications, approaches and taxonomy." Computer Science Review 40 (2021): 100369.

[9] Picard, Jean-Claude, and Maurice Queyranne. "The time-dependent traveling salesman problem and its application to the tardiness problem in one-machine scheduling." Operations research 26, no. 1 (1978): 86-110.

[10] Baniasadi, Pouya, Mehdi Foumani, Kate Smith-Miles, and Vladimir Ejov. "A transformation technique for the clustered generalized traveling salesman problem with applications to logistics." European Journal of Operational Research 285, no. 2 (2020): 444-457.

[11] Efroni, Yonathan, Gal Dalal, Bruno Scherrer, and Shie Mannor. "How to combine tree-search methods in reinforcement learning." In Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, no. 01, pp. 3494-3501. 2019.

[12] Efroni, Yonathan, Gal Dalal, Bruno Scherrer, and Shie Mannor. "Beyond the one-step greedy approach in reinforcement learning." In International Conference on Machine Learning, pp. 1387-1396. PMLR, 2018.

[13] Boyan, Justin, and Andrew Moore. "Generalization in reinforcement learning: Safely approximating the value function." Advances in neural information processing systems 7 (1994).

[14] Guerriero, Francesca, Marco Mancini, and Roberto Musmanno. "New rollout algorithms for combinatorial optimization problems." Optimization methods and software 17, no. 4 (2002): 627-654.

[15] C. E. Miller, A. W. Tucker, and R. A. Zemlin. 1960. Integer Programming Formulation of Traveling Salesman Problems. J. ACM 7, 4 (Oct. 1960), 326329.