UNIVERSIDAD DE SANTIAGO DE CHILE

FACULTAD DE INGENIERÍA

Departamento de Ingeniería Informática

Evolución de algoritmos para pares de problemas de optimización combinatoria

Tamara Sáez Riquelme

Profesor guía: Víctor Parada Daza

Tesis de grado presentada en conformidad a los requisitos para obtener el grado de Magíster en Ingeniería Informática

Santiago – Chile

2016

TABLA DE CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE ILUSTRACIONES

Figuras del capítulo 2

Figuras del capítulo 3

CAPÍTULO 1. INTRODUCCIÓN

1.1 ANTECEDENTES Y MOTIVACIÓN

Es común encontrar en la literatura problemas de optimización combinatoria similares entre sí, en los cuales uno de ellos posee un algoritmo polinomial que lo resuelve (Papadimitriou & Steiglitz, 1982) y el otro es de tipo NP-Hard. A un par de problemas que tienen estas características, en este proyecto se les denomina problema fácil - problema difícil. El problema fácil es aquel que puede ser resuelto por un algoritmo polinominal, mientras que el otro se denomina problema difícil (Cook et al., 1995).

Estos problemas relacionados tienen diversos orígenes, los cuales pueden ser problemas del mundo real o problemas en el mundo teórico formulados en grafos. Un caso típico lo constituye el problema del camino mínimo (PCM) (Papadimitriou & Steiglitz, 1982), el cual consiste en identificar la ruta más corta entre dos puntos de un grafo dado. Donde puede existir más de un camino entre ellos, los arcos tienen una distancia que es dada por un número entero en general, no negativo (problema fácil). Este problema ha sido ampliamente estudiado en la literatura, Dijkstra (1959) y otros científicos propusieron un algoritmo polinomial que lo resuelve conocido como el algoritmo de Dijkstra, que permite encontrar todos los caminos mínimos entre los pares de nodos (Cook et al., 1995). El problema difícil asociado es el problema del camino mínimo con restricción (PCMR), el cual pertenece a la clase NP-Hard (Handler & Zang, 1980), consiste en determinar el mejor camino que une un par de nodos dados, pero considerando como restricción que, el camino posee un costo asociado que no se debe sobrepasar, lo que constituye una restricción del tipo mochila (PM). El problema del árbol de cobertura de costo mínimo (PACCM)(Papadimitriou & Steiglitz, 1982) tanmbién es un problema típico, el cual consiste en encontrar un árbol de costo mínimo, que cubra todos los vértices de un grafo, y el problema relacionado es el problema del árbol de cobertura mínima generalizado (PACMG) (Dror et al., 2000), que consiste en encontrar en un grafo no dirigido un árbol de cobertura de costo mínimo, Los vértices de cada grupo, donde el árbol de cobertura debe incluir sólo un vértice de cada grupo. Este problema es de gran interés debido a las diversas aplicaciones, tales como, aplicaciones en la física (Kansal & Torquato, 2001), riesgos agrícolas (Dror et al., 2000), además de aplicaciones en el área de la toma de decisiones de ubicaciones de centro de distribución, entre otros (Myung et al., 1995). Otro par de problemas relacionados es el PACCM y el problema del vendedor viajero

generalizado (PVVG) (Srivastava et al., 1969). El problema consiste en encontrar un recorrido en el que existen grupos predefinidos y el viajero debe visitar exactamente un nodo en cada grupo minimizando el costo total del viaje. El problema posee una serie de aplicaciones, entre las que destacan el despacho del correo (Laporte et al., 1996), orden de selección en bodegas (Noon & Bean, 1991), secuenciamiento de archivos computacionales (AL, 1969) entre otros.

Los métodos que se han utilizado para abordar el problema complejo a partir del correspondiente problema fácil, son generalizaciones del método que resuelve el problema fácil. Por ejemplo, la determinación de un circuito hamiltoniano que corresponde a una solución factible para el problema del vendedor viajero (problema difícil), puede ser encontrada de manera heurística generando el árbol de cobertura de costo mínimo (problema fácil) y adaptándolo como solución para el vendedor viajero (Applegate et al., 2009). La generalización de este método fácil es el que da origen comúnmente a un método heurístico que resuelve el problema complejo, debido a que hasta ahora nadie ha encontrado un método polinomial que resuelva el problema difícil. Esto conduce a pensar que los elementos que se utilizan para resolver el problema fácil, es decir, los elementos algorítmicos que se utilizan para el problema sencillo, pueden ser reutilizados para dar solución al problema complejo, aunque esto de alguna forma genera un sesgo, debido a que los nuevos algoritmos que se van a generar para el problema difícil van a estar inspirados en el problema fácil. Se podría pensar que la exploración de todo el dominio de posibilidades algorítmicas que existen para resolver el problema difícil es mucho más amplio, sin embargo, este tema ha sido poco explorado en la literatura y los resultados recientes a través de hiper-heurística o de generación automática de algoritmos, algo de luz muestran sobre esta problemática de explorar otras posibilidades para generar soluciones a los problemas difíciles.

Los problemas relacionados considerados en este trabajo de investigación son el PCM - PCMR; PACCM - PACMG y PACCM - PVVG.

Para dar solución a los problemas mencionados que pertenecen a la clase NP-Hard se requieren algoritmos que encuentren en tiempos razonables para esto. Dentro de la optimización heurística existe una rama llamada Computación Evolutiva, que se encarga de encontrar dichos algoritmos. Esta rama pertenece a un área de la inteligencia artificial que se compone de un conjunto de técnicas basadas en los procesos que propuso Charles Darwin, que son: la evolución y selección natural de las especies, donde a partir de una población compuesta por diferentes soluciones que resuelven el problema de forma progresiva, se van generando nuevas soluciones respetando el principio de que las soluciones mas aptas, son las que dan origen a las nuevas poblaciones de solución (Koza, 1999).

Una de las técnicas utilizadas de la Computación Evolutiva es la Programación Genética donde las soluciones representan componentes elementales de programas, lo que permite que a partir de algunas especificaciones del problema en estudio, se produzca automáticamente un algoritmo que lo resuelva. Con ello, se puede decir que es el computador el que genera la nueva solución factible. A partir de este enunciado es posible beneficiarse de heurísticas elementales para este problema de optimización y de forma evolutiva, generar nuevas soluciones. Para conseguir estas nuevas soluciones, se seleccionan tales heurísticas y se combinan de manera evolutiva, para obtener una huerística genérica que pueda resolver cualquier instancia de un problema de optimización. La combinación de heurísticas se conoce como hiper-heurística (Burke et al., 2010), la cual realiza una búsqueda en el espacio de las heurísticas en vez de buscar en el espacio de la solución del problema. La Programación Genética ha generado distintos descubrimientos e inventos, donde su principal exponente, John Koza, a través de distintos experimentos, genera productos electrónicos patentables, como antenas o circuitos (Koza, 2003). En cuanto a la generación de algoritmos, se ha utilizado para resolver problemas NP-Hard como el problema de coloración de vértices (Bolton et al., 2013) y el problema de la mochila (Parada et al., 2015), dejado en evidencia la efectividad de esta técnica.

A pesar de los enormes esfuerzos existentes en la literatura para resolver los pares de problemas relacionados que pertenecen a los problemas difíciles, todavía existe una gran brecha para lograr que algoritmos exactos puedan ser utilizados para resolver instancias de cualquier tamaño del problema. Pocos investigadores se han preocupado de estudiar la relación que existe entre los componentes elementales de los algoritmos polinomiales y los potenciales algoritmos que existen para los problemas difíciles. Aunque la computación evolutiva, específicamente la Programación Genética ha sido utilizada para generar heurísticas en un concepto que es conocido como hiper-heurística, no se detectan nuevos algoritmos para el problema, más bien métodos de resolución que son difíciles de comprender que generalmente están escritos en lenguaje Lisp, al partir del cual es difícil descifrar como se relacionaron esos componentes algorítmicos para dar origen a estos nuevos métodos de resolución. En consecuencia es poco conocimiento el que se puede obtener a partir de esas estructuras debido a que no fue el enfoque esencial de esos autores en su trabajo. Generar automáticamente algoritmos para problemas difíciles a partir de los componentes del problema directamente relacionado, podría transformarse en una nueva metodología de resolución que se puede aplicar y extender a todos los problemas de optimización combinatoria que pertenecen a la clase difícil.

Generar automáticamente algoritmos eficientes para problemas de optimización facilitaría el trabajo de muchas personas que manualmente buscan una heurística eficiente para un

determinado problema. La potencialidad de poder encontrar algoritmos eficientes para los problemas, en términos prácticos, algoritmos que resuelven en corto tiempo la solución con un alto nivel de precisión tiene consecuencias enormes en el mundo de la gestión. La mayoría de los problemas que surgen en la gestión de operaciones, tales como: la planificación de rutas, tareas, operaciones del procesamiento de las máquinas de un sistema productivo entre otros, día a día enfrentan este problema manualmente, típicamente resolviéndolos con heurísticas que provienen del mismo mundo real. Esto se debe a que las mismas herramientas computacionales para resolver estos problemas en la práctica no existen. Debido a que no es posible crear una herramienta que resuelva todos los casos posibles y de existir, sería costosa. Los resultados de esta tesis podrían ir en directo beneficio para el desarrollo de una herramienta computacional que sea capaz de abordar pares de problemas relacionados.

1.2 DESCRIPCIÓN DEL PROBLEMA

Los problemas que se abordan en el presente trabajo (PCMR, PACMG y PVVG) presentan diversos estudios en la literatura, a pesar de esto siguen siendo un desafío computacional. Entre los estudios relacionados existen algunos que utilizan la Programación Genética de forma tradicional, sin embargo, no existen estudios que utilizen los componentes elementales de los problemas fáciles para dar solución a los problemas difíciles utilizando Programación Genética.

Para el desarrollo de este estudio se analiza el comportamiento de la generación de algoritmos basados en los componentes elementales de los problemas fáciles y considerando la incorporación gradual de nuevos elementos de refinamiento que permiten encontrar mejores resultados computacionales para cada uno de los problemas en estudio. Para el desarrollo de éstos, surgen diversas preguntas que son consideradas como parte de este trabajo, entre ellas: ¿los algoritmos generados utilizando sólo los componentes elementales son suficientes para dar solución a los problemas?, ¿estos algoritmos son eficientes?, ¿cómo afecta al desempeño computacional de los algoritmos generados los grupos de instancias de adaptación? y ¿cómo afecta al desempeño computacional de los algoritmos generados la función de evaluación?.

1.3 SOLUCIÓN PROPUESTA

1.3.1 Características de la solución

Utilizando Computación Evolutiva, se pretende generar algoritmos que sean capaces de resolver problemas de optimización combinatoria. Estos algorimos son creados por una máquina computacional de forma automática utilizando componentes del problema fácil como heurística para la generación de algoritmos que resuelvan el problema difícil. Los algoritmos obtenidos serán estudiados comparativamente con los propuestos en la literatura utilizando las instancias propias de los problemas que se tratarán en este trabajo de tesis, los cuales son: el PCMR, el PACMG y el PVVG.

1.3.2 Propósito de la solución

El propósito de la solución es aportar en el campo de la Computación Evolutiva y la optimización combinatoria, generando algoritmos automáticamente mediante el uso de componentes de los problemas fáciles, para los problemas difíciles que aún no poseen solución en un tiempo razonable.

1.4 OBJETIVOS Y ALCANCES DEL PROYECTO

En ésta sección se presenta el objetivo general, los objetivos específicos y el alcance y limitaciones que componen la presente investigación.

1.4.1 Objetivo general

Diseñar automáticamente algoritmos para problemas de optimización combinatoria considerados computacionalmente difíciles, a partir de componentes elementales que provienen de problemas estrechamente relacionados.

Las parejas de problemas que se abordan son: problema del camíno mínimo - problema del camíno mínimo con restricción; problema del árbol de cobertura de costo mínimo - problema del árbol de cobertura mínima generalizado; problema del árbol de cobertura de costo mínimo - problema del vendedor viajero generalizado.

1.4.2 Objetivos específicos

- Diseñar el proceso evolutivo para la creación de algoritmos.
- Definir las estructuras de datos necesarios para el PCMR, el PACMG y el PVVG, que soporten las instancias computacionales de los problemas y las respectivas soluciones algorítmicas.
- Definir los componentes a utilizar como funciones y terminales para la generación de los nuevos algoritmos.
- Realizar un conjunto de experimentos computacionales para generar los algortimos que resuelvan los casos de estudio.
- Analizar la complejidad, principios de funcionamiento y calidad de la soluciones obtenidas por los mejores algoritmos generados.

1.4.3 Alcances

- El trabajo de ésta investigación se concentra escencialmente en descubrir nuevos algoritmos para el PCMG, el PACMG y el PVVG, basandose en los componentes algorítmicos del

problema para el cual se conoce su algoritmo polinomial. Los algoritmos se van a construir evolutivamente mediante el uso de programación genética.

- Los algoritmos van a ser probados con un grupo de instancias típicas para cada problema abordado en el presente trabajo.
- Se medirá el tiempo computacional de los algoritmos y la calidad de solución que se encuentren y en base a ellos se va a obtener una conclusión sobre los descubrimientos principales que se realicen en este trabajo.

1.5 METODOLOGÍAS Y HERRAMIENTAS UTILIZADAS

A continuación se especifica la metodología, herramientas y ambiente de desarrollo que se utilizará para llevar a cabo esta tesis.

1.5.1 Metodología de trabajo

La metodología para desarrollar el presente trabajo se basa en los siguientes objetivos: desarrollar, medir y evaluar la experiencia. El proceso a seguir para lograr estos objetivos consiste en las siguientes tres etapas:

- Desarrollo y obtención del conjunto de algoritmos: para el desarrollo del programa necesario para obtener los algoritmos a evaluar, se utiliza el método propuesto por Riccardo Poli para usar la Programación Genética Poli et al. (2008). Para resolver un problema aplicando Programación Genética, se deben tomar en consideración 5 pasos preparatorios, los cuales son:
    1. Conjunto de terminales: son entradas externas al programa, funciones sin argumento y constantes.
    2. Conjunto de funciones: son funciones definidas de acuerdo al dominio del problema. Éstas deben cumplir dos criterios:

- Clausura: se refiere a que una función u operador debería ser capaz de aceptar como entrada cualquier salida producida por cualquier función u operador del conjunto de no terminales.
- Suficiencia: hace referencia al hecho de que el poder expresivo del conjunto de no terminales debe ser suficiente para representar una solución para el problema en cuestión.

3. Función de evaluación o *fitness*: se definen inicialmente las instancias de datos con que será evaluado el problema y en base a éstos, se definen funciones que midan el rendimiento y comportamiento de los resultados.
4. Parámetros de la Programación Genética: se definen los parámetros propios de la Programación Genética. Los principales son: tamaño de la población, porcentaje de mutación y cruzamiento, casos de *fitness*.
5. Criterio de término y diseño de la solución: se definen los parámetros asociados a la cantidad de generaciones que se deben correr y cuando estas deben detenerse, o resultado al que se espera llegar para obtener la solución.

- Desempeño computacional de los algoritmos: esta etapa mide la calidad de los algoritmos generados por el proceso evolutivo y que son seleccionados para ser estudiados. En este caso, la calidad de un algoritmo se determina a partir de su capacidad para encontrar buenas soluciones con instancias distintas a las utilizadas durante el proceso evolutivo y evaluar sus resultados numéricos a fin de determinar qué tan buenos son realmente.
- Evaluación: consiste en comparar los algoritmos con otras técnicas. Estas técnicas son las propuestas por Derrac et al. (2011), donde se utiliza una serie de evaluaciones estadísticas para comparar el rendimiento de algoritmos como los que se obtienen en el desarrollo de este trabajo.

1.5.2 Herramientas de desarrollo

Para el desarrollo y ejecución de los experimentos se utilizarán dos equipos con las siguientes características:

- Computador personal:Procesador:Intel®Core™i5-3317U CPU @ 1.70GHZ y 8 GB de memoria RAM DDR3

- Clúster del Departamento de Ingeniería Industrial de la Universidad de Chile: 9 nodos, Intel (R) Xeon (R) CPU E5620 2.40GHZ y una memoria total de 400 GB.

  las herramientas de software que se utilizarán son:

- Sistema operativo: Ubuntu GNU/Linux.
- Sistema de computación evolutiva: ECJ 23
- Plataforma de desarrollo: Eclipse IDE for Java Developers.
- Herramienta ofimática: LaTeX.
- Herramienta estadística: STATA.

CAPÍTULO 2. ASPECTOS TEÓRICOS Y REVISIÓN DE LA LITERATURA

En esta sección se abarcan los aspectos relacionados al conocimiento general para la comprensión del presente trabajo (aspectos teóricos) y la revisión de la literatura asociada al trabajo presentado en esta tesis. Para realizar un análisis de la Programación Genética aplicada a problemas NP-Hard es necesario conocer la base teórica de ésta. Para ello, se exponen los conceptos fundamentales de la computación evolutiva y la Programación Genética. La sección 2.1 se centra en explicar aquellas partes fundamentales al tema que se trata en esta tesis. De la revisión de la literatura se desprende lo presentado en la sección 2.2.

2.1 ASPECTOS TEÓRICOS

2.1.1 Computación Evolutiva

A lo largo de la historia, la adaptabilidad de un organismo a un medio en constante variación ha sido la clave para la supervivencia y la evolución (Darwin 1968). Así nace la computación evolutiva (CE), cuyo principio propone el mismo concepto de supervivencia y transformación.

Los acontecimientos a los que un organismo se ve sometido conforman la selección que el medio impone sobre éste, y junto a su comportamiento determinan la adaptabilidad del individuo frente a la demanda variable a la cual es expuesto, siendo capaz incluso de encontrar las maneras más extraordinarias para lograr su objetivo.

Se considera entonces que la evolución consta de dos fases o etapas:los acontecimientos o variables aleatorias de los organismos y su posterior selección.

Los principios de la CE y sus algoritmos de optimización son la selección impuesta sobre una población inicial de individuos cuyos problemas son creados a partir de soluciones obtenidas al azar o con conocimiento del asunto en cuestión. El resultado de la selección determina una función objetivo (*fitness*), cuyo proceso reiterarivo concluye en generaciones donde quedan los

individuos mejor catalogados mediante la reproducción,la selección, cruzamiento y mutación.

La reproducción supone la transferencia del código genético de cada individuo a su descendencia, donde la capacidad reproductiva de las especies es enorme y el número de individuos podría incrementarse exponencialmente si todos sus individuos se pudieran reproducir con éxito simultáneamente, es por eso que existe una probabilidad de que la reproducción se lleve a cabo de forma exitosa. La mutación sucede debido a que los errores en la replicación durante el proceso de transferencia de información genética son inevitables, además de ser necesario incluir variabilidad en las especies. La selección es el resultado de la reproducción de las especies en un medio de competencia dentro de un espacio físico finito donde no hay espacio o recursos para todos y quienes sobreviven son los más aptos.

En la Figura 2.1 se muestra la estructura general de la computación evolutiva, donde se destacan los componentes más importantes de ésta. Estos componentes son: representación (definición de individuos), función de evaluación (o función de *fitness*), población, operadores de variación, cruzamiento, mutación y mecanismo de supervivencia (selección).

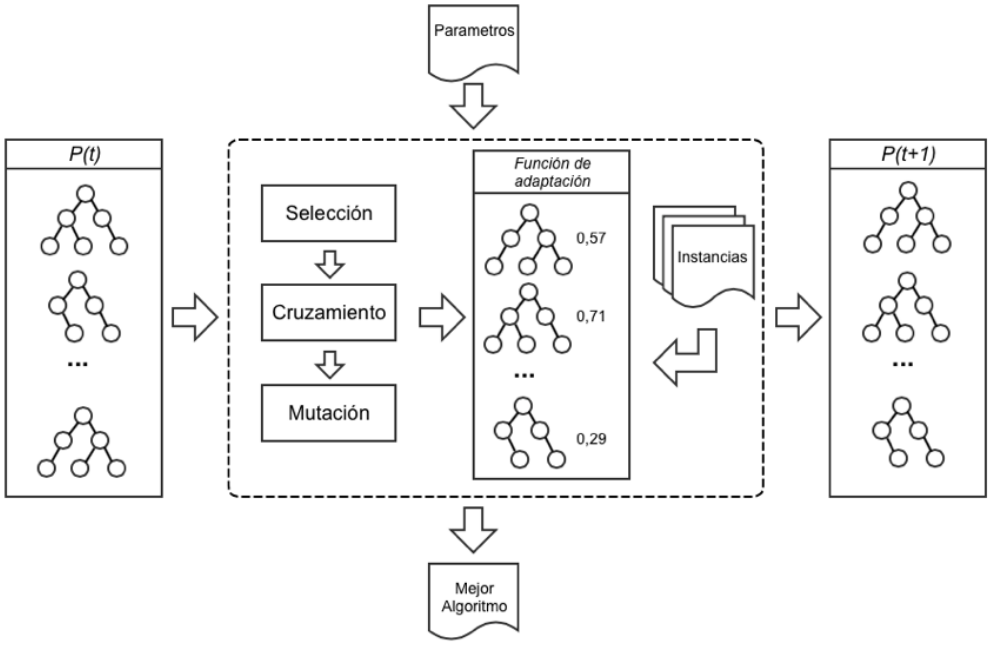

Figura 2.1: Estructura general de la computación evolutiva (Contreras Bolton et al., 2013).

Los componentes principales para la CE son:

- *Representación*: es una representación del problema a tratar en el mundo real, esto quiere

decir que debe existir coherencia entre el espacio de solución a recorrer y el problema en si, para posteriormente poder interpretar la solución producida por la evolución.

- *Población*: son los individuos que se encuentran en una generación. Se les llama individuos a los posibles candidatos que pueden dar solución al problema que se quiere resolver. La población compone la unidad de evolución, donde los individuos se mantienen estáticos, mientas que la población va cambiando.
- *Inicialización*: el objetivo es generar la población inicial, la cual puede ser de forma aleatoria o utilizando alguna heurística. Generalmente,se utiliza la generación aleatória, ya que es la propia CE la que se encargará de mejorar a los individuos.
- *Función de mejora o fitness*: es la encargada de medir la aptitud de un individuo de la población (poli et. al 2008). Esta función permite discriminar quiénes son los individuos que pasarán a las siguientes generaciones y quiénes no. Es así, dependiendo del problema, la evolución toma una tendencia de máximizar o mínimizar, como guía para generar buenas soluciones.
- *Operadores de variación*: los operadores permiten generar nuevos individuos a la población a partir de los ya existentes. Su objetivo principal es recorrer nuevas y mejores soluciones no producidas en las generaciones anteriores, con el fin de resolver el problema planteado de la mejor forma. Los operadores clasicos son tres: cruzamiento, mutación y selección (Falkenauer,1998).
  - *Cruzamiento*: es una operación binaria que permite el intercambio de material entre dos individuos, teniendo como resultado, dos nuevos individuos con información híbrida en cada uno.
  - *Mutación*: opera sobre un individuo, y se aplica para generar diversidad de población.
  - *Selección*: se encarga de obtener los mejores individuos que sobrevivirán al proceso.
- *Criterio de parada*: los criterios de parada se pueden diferenciar en dos casos: el primero, es conocer el valor óptimo de la solución y que éste sea alcanzado. El segundo puede ser que se han acabado los recursos tales como el máximo tiempo de CPU, un período de tiempo, no hay variación en la población, etc. Dentro de la computación evolutiva, existen diversos métodos que utilizan este concepto de una forma más específica de acuerdo a su representación. En el artículo (Kouchakpour et al., 2009) se presenta una de las clasificaciones, que se pueden realizar de las variantes de CE. Ésta puede ser vista en la Figura 2.2.

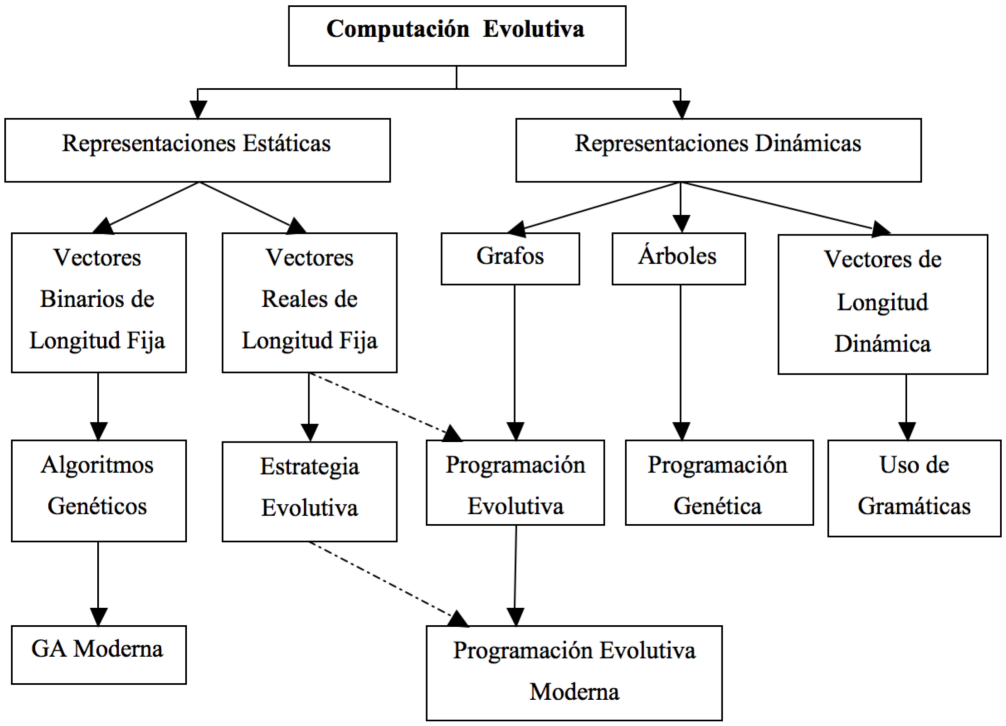

Figura 2.2: Métodos de la CE basados en el tipo de representación (Kouchakpour et al., 2009).

2.1.2 Programación Genética

Una de las tareas más desafiantes de la ciencia de la computación, consiste en que un computador pueda dar solución a problemas sin la necesidad de ser programados de manera explícita. Samuel (1959) declaró:

¿Cómo pueden los computadores aprender a resolver un problema, sin ser programados explícitamente para ello? En otras palabras, ¿Cómo pueden los computadores hacer lo que tienen que hacer, sin específicar exáctamente cómo han de hacerlo?

La Programación Genética (PG) es una técnica de Computación Evolutiva que mediante la generación de código computacional, es capaz de resolver problemas automáticamente, sin la necesidad de que un usuario especifíque o conozca la estructura de la solución (Poli et al., 2008). A pesar de no poder garantizar los resultados por ser esencialmente un proceso evolutivo, ha sido muy exitosa y se han encontrado formas inesperadas de resolver problemas (Holland, 1975).

En la PG, el proceso evolutivo comienza con una población inicial generada aleatoriamente, la cual es representada por programas computacionales, donde cada individuo es ejecutado y evaluado en virtud de su medida de adaptación frente a un conjunto de casos de prueba propios del problema. Esto se puede apreciar en la figura XX que presenta el diagrama de flujo de los pasos básicos de la PG.

La PG posee tres operadores genéticos, basándose en la selección natural de Darwin: la superviviencia del más apto (operador de selección),la reproducción (operador de cruzamiento) y la mutación. Los operadores se utilizan para crear las nuevas generaciones de individuos que reemplazarán las anteriores (Poli et al., 2008). Generalmente la solución inicial tiene una muy mala calidad, pero algunos individuos serán más aptos que otros. Estas diferencias de adaptación que presenta cada individuo son explotadas a través de las generaciones, y lentamente aquellas caracteristicas individuales que ayudan a resolver el problema, se harán mas comunes en la población. Este funcionamiento se puede ver apreciado en el Algoritmo 2.1

Algoritmo 2.1: Programación genética

1: *Generar Población Inicial aleatoria*
2: **while** *La condición de término sea verdadera* ***do***
3: Ejecutar cada programa y determinar su *fitness*
4: Seleccionar uno o dos individuos-programas de la población para participar en las operaciones genéticas.
5: Crear nuevo programa mediante la aplicación de las operaciones genéticas
6: ***end while***

Para resolver un problema utilizando la PG, Koza (1992a) define cinco pasos, los cuales se detallan a continuación:

- *Definición del conjunto de terminales*: Permite identificar el conjunto de terminales que van a ser utilizados por los individuos en la población. Los terminales pueden consistir en constantes que representen entradas, sensores o variables de estado, funciones sin argumento, entre otros.
- *Definición del conjunto de funciones*: Permite identificar el conjunto de funciones, las cuales corresponden a posibles operaciones elementales, de las que se sospecha puedan intervenir en el cómputo de alguna solución, tales como operaciones aritméticas ($+$,$-$,$*$,$/$), funciones matemáticas ($seno$,$coseno$,$logaritmo$), operaciones booleanas (and,or,not), operaciones condicionales (if-$then$-$else$) y funciones que causan iteraciones ($while$,for).

- *Definición de la medida de aptitud*: Es la encargada de medir la aptitud de los individuos de la población para resolver un problema. La medida de aptitud varía según el problema al que se quiere dar solución. Generalmente se utiliza el error relativo.
- *Parámetros de control*: Consiste en escoger los valores de ciertos parámetros que permiten calibrar la evolución. Estos valores pueden ser la probabilidad de cruzamiento, mutación, tipo de selección, entre otros.
- *Especificar el criterio de designación de resultado y término de ejecución*: Se puede designar al mejor individuo de todo el proceso de la evolución, a los k mejores individuos de las últimas generaciones u otra alternativa que sea apropiada al problema que se quiere resolver. Para el criterio de término, se puede fijar un número máximo de generaciones o algún equivalente de éxito como un valor de evaluación de desempeño, entre otros.

Adicionalmente, para una correcta implementacion de la PG, existen dos propiedades que se deben considerar al momento de construir las funciones y terminales:

- *Suficiencia*: Las funciones y terminales utilizados deben ser suficientes para representar una solución para el problema (Koza, 1992a). Si el diseño no es apropiado o suficientemente representativo, la PG no podrá encontrar una solución factible al problema planteado.
- *Clausura*: Cada función debe ser capaz de manejar de forma adecuada los argumentos que pueda recibir como entrada, estos son los posibles valores retornados por otra función definida en el conjunto y cualquier valor o tipo de dato que sea alcanzable desde el conjunto de terminales (Koza, 1992a). La clausura no es una propiedad indispensable, en los casos que no se puede garantizar, se debe utilizar alternativas tales como, la eliminación de individuos o la penalización sobre su medida de aptitud cada vez que algún individuo contenga alguna construcción infactible.

A partir de la última década del siglo XX, la PG ha sido considerada como un método ”humano competitivo”para la resolución de problemas. Las principales razones de esta clasificación se debe a que la mayoría de los problemas planteados en inteligencia artificial, aprendizaje automático y sistemas adaptativos pueden ser formulados como una búsqueda de programas computacionales, y la técnica de PG entrega una forma satisfactoria de guiar la búsqueda en el espacio de programas computacionales (Affenzeller, 2001).

Turing, en el año 1948, advirtió correctamente que una posible aproximación a la inteligencia artificial incluiría un proceso evolutivo, donde un programa computacional (material hereditario)

sea sometido a una modificación progresiva (mutación) bajo la dirección de la selección natural (Koza & Poli, 2005).

Un resultado generado por un método automatizado puede ser clasificado como “humano competitivo”, independiente del hecho de que sea generado de forma automática. Sin embargo, un resultado humanamente competitivo obtenido debe tener un cierto grado de complejidad, es por esto que un resultado generado por un método automatizado que resuelve un *“toy problem”* (por ejemplo, las torres de Hanoi, apilamiento de bloques, caníbales y misioneros), no sería considerado como “humano competitivo”, debido a que la solución no es publicable como un nuevo resultado científico, sólo es de interés porque fue creado de forma automática.

Así, según Koza et al. (2003), una solución generada automáticamente para un problema es humanamente competitiva si ésta satisface uno o más de los ocho criterios que se muestran en la Tabla 2.1.

Tabla 2.1: Criterios para un resultado “humano-competitivo” (Koza et al., 2003).

Nº	**Criterio**
A	El resultado fue patentado como un invento, es una mejora de una invención patentada, o podría calificar como una nueva invención patentable.
B	El resultado es igual o mejor que un resultado científico aceptado y publicado en una revista científica.
C	El resultado es igual o mejor que un resultado que fue colocado en una base de datos o archivo de resultados y gestionada por un panel internacional de expertos científicos reconocidos.
D	El resultado es publicable como un nuevo resultado científico, independiente del hecho que fue creado mecánicamente.
E	El resultado es igual o mejor que un resultado reciente de un problema que ha tenido sucesivos resultados cada vez mejores.
F	El resultado es igual o mejor que un resultado que se consideró un logro en su campo en el momento en que fue descubierto.
G	El resultado soluciona un problema de dificultad indiscutible en su campo.
H	El resultado está entre los mejores o gana una competencia regulada que incluye participantes humanos.

La obtención de resultados humanamente competitivos ha incentivado el crecimiento del campo de la computación genética y evolutiva, esto se debe a que la obtención de estos resultados depende del progreso, desarrollo y perfeccionamiento de los métodos de búsqueda genéticos y evolutivos. No obstante, a medida que resultados humanamente competitivos sean generados, irán apareciendo problemas más desafiantes.

2.2 REVISIÓN DE LA LITERATURA

2.2.1 Generación automática de algoritmos

La generación automática de algoritmos (GAA) ha sido utilizada los últimos años para resolver distintos problemas de optimización combinatoria. En este proceso, los algoritmos pueden ser generados mediante el uso de hiper-heurísticas que poseen estructuras que permiten ser interpretadas como algoritmos. Se pueden encontrar varios trabajos que utilizan la PG para realizar la GAA (Contreras Bolton et al., 2013; Drake et al., 2014; Parada et al., 2015).

El proceso para diseñar hiper-heurísticas que permitan obtener una solución a un problema de optimización combinatoria consta de tres fases (Ahandani et al., 2012). La primera corresponde a la formulación matemática del problema. La segunda fase es el diseño de heurísticas de bajo nivel que permitan entregar una solución apróximada al problema. Y la tercera fase procesa, por medio de una hiper-heurística, diferentes combinaciones de las heurísticas definidas en la segunda fase. El proceso de selección de heurísticas se puede realizar a través de métodos de adaptación que buscan los mejores resultados en el espacio de soluciones de acuerdo con una medida de rendimiento (Parada et al., 2015).

2.2.2 Heurísticas y meta-heurísticas

En la actualidad, para resolver problemas computacionalmente complejos es necesario desarrollar algoritmos más avanzados. Los algoritmos exactos suelen requerir una gran cantidad de tiempo debido al tamaño del espacio de soluciones factibles. Para solucionar éste problema, han sido desarrollados los algoritmos aproximados, que, mediante el uso de heurísticas y meta-heurísticas, permiten encontrar soluciones que se acercan a una mejor solución. Los algoritmos heurísticos utilizan funciones especiales, las que están diseñadas para encontrar el espacio de soluciones de forma inteligente.

En la Figura 2.3 se muestra la clasificación de diferentes problemas de optimización, los cuales están divididos en dos categorías: algoritmos exactos y algoritmos aproximados (Desale et al.,

2015).

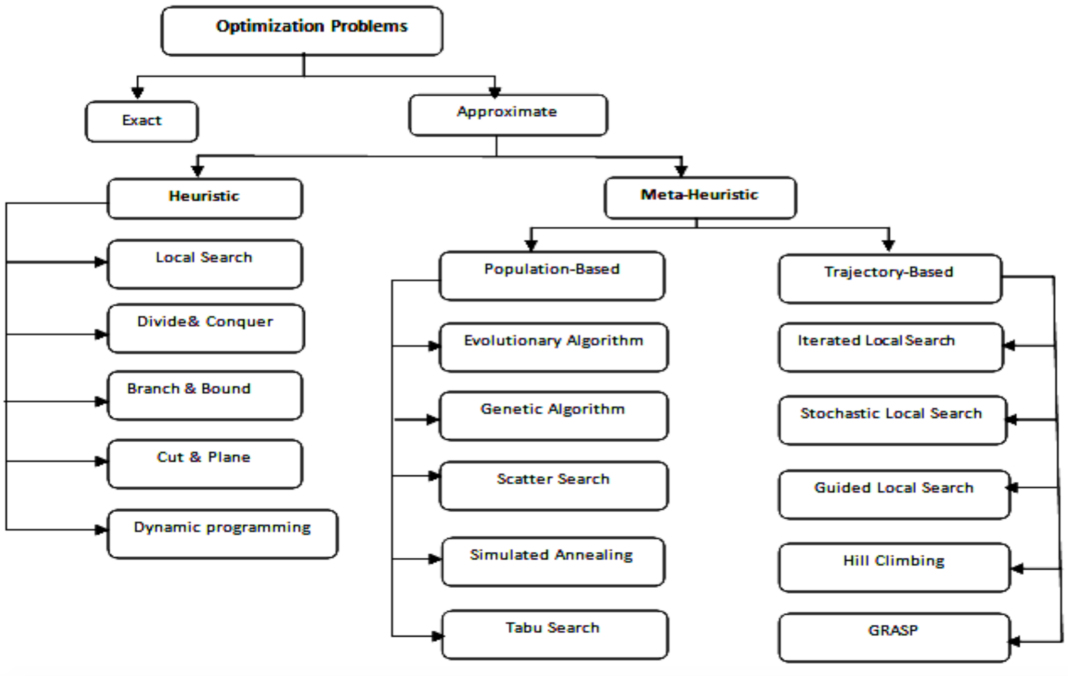

Figura 2.3: Métodos utilizados para resolver problemas de optimización (Desale et al., 2015).

Los algoritmos meta-heurísticos son el proceso de generación iterativo que guía una heurística para explorar y explotar el espacio de búsqueda.

Las heurísticas y meta-heurísticas son utilizadas para encontrar una solución óptima en un espacio de búsqueda discreto en problemas de optimización combinatoria. Un ejemplo es el problema del vendedor viajero, donde la búsqueda del espacio de soluciones factibles crece exponencialmente en función del tamaño del problema aumenta, lo que hace inviable la realización de una búsqueda exhaustiva de la solución óptima (Desale et al., 2015).

2.2.3 Hiper-heurísticas

El término de hiper-heurística fue utilizado por primera vez en el año 2000 para lograr describir una heurística que escoge heurísticas en el contexto de la optimización combinatoria. Mientras que la automatización del diseño de heurísticas se remonta a la década de 1960. La definición

de hiper-heurísticas se refiere a un método de búsqueda o mecanismo que permite seleccionar o construír la heurística que resuelve problemas de búsqueda de cálculo de aprendizaje. Se pueden diferenciar dos categorías: la selección de heurísticas y la generación de las mismas. La características de las hiper-heurísticas es que operan en un espacio de búsqueda de una heurística y no directamente en el espacio de búsqueda de soluciones del problema en cuestión (Burke et al., 2013).

El proceso para generar algoritmos en forma automática puede entenderse como una hiper-heurística, o como una metodología, cuando la generación automática de algoritmos se realiza mediante el uso de la programación genética y definición del proceso evolutivo. En ambos casos existen elementos comunes que permiten generar artefactos que resuelvan el problema en estudio, un proceso de mejora guiado por una función objetivo, entre otros. El proceso de generación de algoritmos busca obtener una estructura que se expresa mediante un árbol sintáctico y luego se decodifica para su posterior comprensión e interpretación. El objetivo del proceso es poder extraer conocimiento nuevo para recrear los algoritmos existentes en la literatura. Además, la GAA permite generar algoritmos que contemplen el uso de las mejores heurísticas o algoritmos existentes para el problema. Mientras que las hiper-heurísticas son utilizadas para resolver en forma eficiente el problema sin necesidad de realizar una interpretación del resultado obtenido (Burke et al., 2013).

2.2.4 Instancias

En los últimos años, se han desarrollado algoritmos cada vez más sofisticados para los problemas de optimización. Estos algoritmos incluyen distintos métodos o técnicas, los que son sometidas a estudios experimentales que tienen por objetivo determinar que algoritmo obtiene un mejor rendimiento, comúnmente éstos se basan en conjuntos de instancias públicas. El teorema de *no free lunch* (NFL) indica que no existe un algoritmo supere a los existentes en todas las instancas disponibles de un problema. Si un algoritmo se declarase como mejor que otro conjunto de algoritmos, entonces es posible esperar que existan instancias que aún no han sido probadas. Es por esto que una de las claves para poder realizar un correcto estudio es caracterizar las instancias del problema de acuerdo a su dificultad y al espacio de soluciones que las representa (Smith-Miles & Lopes, 2012).

Las características más sencillas de una instancia para un problema de optimización son las que

se definen como parte de la sub-clase de la instancia: características como el número de variables y restricciones, o si las matrices que almacenan los parámetros son simétricas, etc.

Para el PCMR, una forma de clasificar las instancias es mediante la distribución de costos que posee (distancias entre los nodos o ciudades), donde se muestra el número de valores distintos que poseen las distancias (Smith-Miles & Lopes, 2012). Las instancias a utilizar para este problema, son obtenidas de la TSPLib (Reinelt, 1991). Esta librería contiene instancias ampliamente utilizadas en la literatura, las que en su totalidad cuentan con su valor óptimo y, en algunos casos, con una posible solución.

FALTA.

2.2.5 Revisión de la literatura para PCMR

El PCMR, que pertenece a la clase NP-Hard (Joksch, 1966), consiste en encontrar un camino o cadena de aristas P_{cad} desde un vértice inicial v_s perteneciente a *V*, a un vértice final v_e también perteneciente a *V*, que minimice el costo total de aristas en P_{cad} y que además la suma de los recursos o pesos de cada aristas no exceda un máximo de recurso a consumir denotado *W* (Dumitrescu & Boland, 2001). Su formulación matemática es la siguiente:

$$min \sum_{i,j \in A} c_{ij} x_{ij} \tag{2.1}$$

$$sujeto \sum_{j:(i,j)\in A} x_{ij} - \sum_{k:(k,i)\in A} x_{ki} = \begin{cases} 1 & \text{si } i = v_s \\ 0 & \text{si } i = 2, ...n-1 \\ -1 & \text{si } i = v_e \end{cases} \tag{2.2}$$

$$\sum_{i,j:(i,j)\in A} w_{ij} x_{ij} \leq W \tag{2.3}$$

$$x_{ij} / x_{ij} \in \{0,1\}, (i,j) \in A \tag{2.4}$$

La ecuación (2.2) da a conocer que debe ser una cadena conexa desde el nodo inicial v_s a v_e, y

la ecuación (2.3) plantea que la suma de los pesos de las aristas consideradas en el camino no debe sobrepasar el peso W (Santos et al., 2007).

Las estrategias utilizadas para el PCMR se pueden clasificar en dos categorías principales, programación dinámica y relajación lagrangiana. Los métodos basados en la programación dinámica también se conocen como *label setting* o *label correcting*. Una de las características de este tipo de algoritmos es que en instancias de tamaño razonable, pueden ser muy veloces; sin embargo, para redes que pueden tomar dimensiones muy grandes podrían dejar de ser una buena alternativa, esto debido a que el número de etiquetas que deben ser almacenadas puede ser muy grandes y como consecuencia pueden ser imposibles de manejar.

Uno de los primeros algoritmos pertenecientes a esta categoría fue propuesto por Joksch (1966), el cual fue extendido por Dumitrescu & Boland (2003), comparandse con tres algoritmos Dumitrescu & Boland (2001); Hassin (1992); Lorenz & Raz (2001) a los que supera en término de tiempo computacional y requisitos de memoria. Los autores afirman también que la integración entre las técnicas de relajación langrageana y pre-procesamiento podrían mejorar el tiempo de ejecución.

Posteriormente, se presentan un enfoque de etapas para abordar el camino mínimo con un conjunto de restricciones o recursos (PCMG) Zhu & Wilhelm (2012). Para ello se propone un algoritmo pseudopolinomial llamado TSA en tres etapas: la primera es un pre-procesamiento de las instancias reduciendo los arcos y los vértices, disminuyendo el espacio de búsqueda, el segundo es un proceso de *setup* para transformar el PCMG a un PCM, el tercero es un procesamiento iterativo de búsqueda. El TSA es comparado con CPLEX (REFERENCIA FALTANTE) y con el algoritmo *label setting* (Dumitrescu & Boland, 2003) para cuatro tipos distintos de problemas, mostrando abordar de forma eficaz los casos de prueba, pero el tiempo de ejecición aumenta a medida que los problemas se van haciendo más grandes en cantidad de nodos y conjunto de recursos.

El segundo enfoque para dar solución al PCMR se basa en el uso de la relajación lagrangiana para resolver la formulación de programación entera del problema. La eficiencia de este enfoque se basa en la eficacia de los algoritmos de ruta más corta sin restricciones subyacentes. Un algoritmo exacto para el problema PCMR basado en relajación Langragiana al cual es denotado como LRA (lagrangian relaxation algorithm)(Santos et al., 2007). El LRA se compara con métodos de *k-shortest path* y con métodos de relajación langragiana, ambos introducidos por Handler & Zang (1980). Los resultados de estas pruebas indican que el LRA puede abordar grandes problemas teniendo ventajas sobre los otros métodos en términos de tiempo computacional y en

el uso de memoria. Sin embargo, los autores comparan su algoritmo sólo con Handler & Zang (1980), y no con los algoritmos mejorados de (Dumitrescu & Boland, 2003).

Luego, se expone un nuevo algoritmo basado en relajaciones Langragianas con enumeraciones de caminos mínimos al cual denominan LRE (*lagrangian relaxation and enumeration*), el cual se sostiene en técnicas de pre-procesamientos (Carlyle et al., 2008). En el trabajo se utiliza cuatro grupos de casos de prueba y se compara con el *label setting* (Dumitrescu & Boland, 2003), donde el LRE resuelve hasta el doble de casos que el *label setting* en un tiempo de 30 minutos. Sin embargo, existen casos que el LRE no supera al *label setting*, debido a la existencia de grandes distancias entre las cotas inferiores y superiores, teniendo un tiempo de ejecución más prolongado. Mientras que en el 2009, se da a conocer un enfoque Langragiano dual al cual se denomina eliminación agresiva de costos (AEE) (Muhandiramge & Boland, 2009). Consiste en una versión modificada del algoritmo expuesto por (Carlyle et al., 2008). AEE se compone de dos etapas principales: primero relaja el problema con métodos duales Langragianos con iteraciones, todo ello mediante la restricción de pesos con pasos de pre-procesamiento, después, aborda el problema basándose en las cotas de la red ya reducida por el pre-procesamiento. Se da a conocer que se podrían mejorar los resultados siempre y cuando se encuentren buenos parámetros iniciales, como las cotas iniciales o múltiplos iniciales, lo cual no se investigó en el artículo científico.

Posteriormente, se propone un método exacto llamado *Pulse* que maneja redes de gran escala para el problema del PCMR (Lozano & Medaglia, 2013), encontrando mejores resultados en los 180 casos propuestos en la literatura, respecto al algoritmo de Santos et al. (2007), logrando aceleraciones de hasta 60 veces. También supera favorablemente al algoritmo propuesto por Dumitrescu & Boland (2003), logrando aceleraciones de hasta 900 veces con las dos series de casos propuestos para el problema.

En uno de los últimos trabajos se propone un modelo mejorado llamado Ameba para abordar el problema del PCMR (Zhang et al., 2013). El modelo Ameba consiste en un conjunto de tubos interconectados que emulan a esta bacteria, y que posteriormente, construyen un modelo matemático (Nakagaki et al., 2001). Para resolver el problema, adaptan el modelo de Ameba al problema de PCMR y después integran un proceso de relajación Langragiana. Se afirma que el método es capaz de abordar problemas del PCMR, pero sólo con los casos de selección de rutas en redes de transporte y de computadora propuestos en su investigación.

2.2.6 Revisión de la literatura para PACMG

El PACMG, que pertenece a la clase NP-Hard (Dror et al., 2000), consiste en un grafo no dirigido cuyos nodos son divididos en k clúster, para determinar un árbol de cobertura de costo mínimo que incluya sólo un vértice de cada cluster. Se define matemáticamente por un grafo no dirigido $G = (V, E)$ de n vértices, m aristas y $V_1, ..., V_k$ una división de V en k subconjuntos (clúster), donde, $V_1 \cup V_2 \cup V_3 ....\cup V_k$ con $V_l \cap V_s = \Phi \ \ \forall j, s \in \{1, ..., s\} \ \ y \ \ l \neq s$. Las aristas son definidas sólo entre vértices que pertenecen a clusters diferentes, el costo de una arista $e = (i, j) \in E$ está dado por e_{ij}.

Su formulación matemática propuesta por Pop (2002) es la siguiente:

Considerando,

$$\text{mín} \sum_{i,j \in A}^{m} c_{ij} x_{ij}$$

$$x_e = x_{ij} = \begin{cases} 1 & \text{Sí la arista } e = (i, j) \in E \text{ es incluye en la solución} \\ 0 & \text{en otro caso} \end{cases}$$

$$z_i = \begin{cases} 1 & \text{Sí el nodo } i \text{ es incluye en la solución} \\ 0 & \text{en otro caso} \end{cases}$$

$$w_{ij} = \begin{cases} 1 & \text{Sí el arco } (i, j) \in A \text{ es incluido en la solución} \\ 0 & \text{en otro caso} \end{cases}$$

Se utilizan las notaciones vectoriales $x = (x_{ij})$, $z = (z_i)$, $w = (W_{ij})$ y la notación $x(E') = \sum_{\{i,j\} \in E'} x_{ij}$, para $E' \subseteq E$, $z(V') = \sum_{i \in V'} z_i$, para $V' \subseteq V$ y $w(A') = \sum_{(i,j) \in A'} w_{ij}$, para $A' \subseteq A$

$$\text{mín} \quad \sum_{e \in E} c_e x_e \tag{2.5}$$

$$s.t. \quad z(V_k) = 1 \qquad \forall k \in K = \{1, \ldots, k\} \tag{2.6}$$

$$x(E(S)) \leq z(S - i) \quad \forall i \in S \subset V, 2 \leq |S| \leq n - 1 \tag{2.7}$$

$$x(E) = k - 1 \tag{2.8}$$

$$x_e \in \{0, 1\} \qquad \forall e \in E \tag{2.9}$$

$$z_i \in \{0, 1\} \qquad \forall i \in V \tag{2.10}$$

La ecuación (2,6) asegura que exactamente n vértice es seleccionado desde cada clúster, la ecuación (2,7) impide que existan ciclos, y la ecuación (2,8) asegura que existan *k-1* aristas en la solución.

Este problema ha sido abordado con diversos enfoques. Uno de los enfoques en la programación entera, en los cuales se han utilizado distintas formulaciones tales como: *formulations based on tree properties*, *flow based formulations*, *formulations based on arborescence properties* y *based on Steiner tree properties* (Pop, 2009). Sin embargo, no son capaces de resolver problemas de gran tamaño y a pesar de la variedad de formulaciones, sólo algunas estrategias encuentran buenas cotas inferiores, pero el tiempo de ejecución es demasiado (Ferreira et al., 2012) o son capaces de encontrar cotas para instancias grandes, debido a los altos recursos de memoria requeridos.

Otros enfoques para resolver el PACMG son las heurísticas constructivas y de mejora. Las heurísticas constructivas proporcionan la solución paso a paso, de manera tal, que se van añadiendo aristas a la solución hasta construir un PACMG. Típicamente, se utilizan adaptaciones de los conocidos algoritmos polinomiales de *Prim*, de *Kruskal* y de *Sollin* (Papadimitriou & Steiglitz, 1982). Las heurísticas de mejora comienzan con una solución inicial que se modifica iterativamente con el objetivo de encontrar mejores soluciones. Las primeras cuatro heurísticas propuestas para dar solución al problema (dos de mejora y dos constructivas), fueron propuestas por Dror et al. (2000), siendo la más efectiva, una de las constructivas.

Otro de los enfoques utilizados para abordar el problema es la combinación de diferentes técnicas, heurísticas y metaheurísticas. Uno de los enfoques es el de *greedy* llamado PROGRES, que es un algoritmo *greedy* aleatorio que incorpora tres técnicas de diversificación: aleatoriedad, perturbación y penalización (Haouari & Chaouachi, 2006), obteniendo un 92,5 % de resultados óptimos, de los casos de prueba que son generados al azar con un máximo de 1000 vértices y 1000 aristas.

Posteriormente, se proponen cinco heurísticas (Ferreira et al., 2012), mostrando un buen desempeño tanto en instancias de pocos vértices como en instancias de gran tamaño, en la calidad de solución y en el tiempo computacional. Además, proponen mejoras que combinan el procesamiento con una búsqueda local, mejorando las soluciones considerablemente, pero aumentando el tiempo de cómputo. Luego, se proponen seis versiones de GRASP (Talbi, 2009) que consisten en la combinación de los métodos anteriores y la técnica de *path-relinking*, lo cual aumenta el rendimiento para las seis heurísticas de GRASP, permitiendo mejorar aún más las soluciones, pero también incrementando el tiempo de cómputo.

Uno de los últimos trabajos propone combinaciones heurísticas produciendo nuevos algoritmos para el PACMG de forma automática, utilizando parte de las técnicas ya existentes en la literatura (Contreras-Bolton et al., 2016). Los algoritmos son construidos utilizando componentes heurísticos elementales, obteniendo desde los métodos descritos en la literatura, más las estructuras de control que dan forma al algoritmo, mediante el uso de la computación evolutiva, obteniendo algoritmos competitivos, en términos de la calidad de la solución obtenida con las heurísticas existentes para abordar el problema, donde se obtiene que el error promedio obtenido por el algoritmo, de los 251 casos de prueba utilizados, es más bajo que el eror promedio de las heurísticas propuestas por Golden et al. (2005) y Ferreira et al. (2012).

2.2.7 Revisión de la literatura para PVVG

El problema PVVG, que pertenece a la clase NP-Hard (Srivastava et al., 1969; AL, 1969), consiste en encontrar un recorrido en el que existen *clusters* o grupos predefinidos y el viajero debe visitar exactamente un nodo en cada *cluster* minimizando el costo total del viaje. La formulación matemática es el siguiente:

$$\sum_{i=1}^{m}\sum_{j=1}^{m} c_{ij}x_{ij} \tag{2.11}$$

Sujeto a

$$\sum_{i=1}^{m}\sum_{j=1}^{m} c_{ij}x_{ij} = 1 \tag{2.12}$$

$$\sum_{j=1}^{m}\sum_{i=1}^{m} c_{ij}x_{ij} = 1 \tag{2.13}$$

$$\sum_{j=1}^{m} x_{ji} - \sum_{j=1} x_{ij} = 0 \tag{2.14}$$

$$x_{ij} \in \{0,1\} \tag{2.15}$$

Las ecuaciones (2.12) y (2.13) garantizan que cada uno de los vértices es visitada exactamente una vez , y la ecuación (2.14) asegura que no existan ciclos.

El PVVG es un problema al cual se le ha dedicado bastantes trabajos. Diversos investigadores (Noon & Bean, 1993; Laporte & Semet, 1999; Lien et al., 1993; Ben-Arieh et al., 2003) han propuesto transformar el problema en una instancia del problema del vendedor viajer (PVV). En una primera mirada, el concepto de transformar un problema que no ha sido tan trabajado a otro muy conocido, da la impresión de que se podrían obtener resultados prometedores. Sin embargo, al tratarse de una conversión, se debe lidear con aspectos técnicos que afectan los resultados. Uno de ellos, se requiere una solución exacta de la instancia PVV, debido a que una aproximación al óptimo puede representar una solución infactible para el PVVG (por ejemplo, qu la solución del PVV imprique recorrer más de un nodo por conjunto en el PVVG). Por otro lado, la estructura de las instancias del PVVG suelen ser distintas a las e los sistemas capaces de entregar una solución a instancias del PVV.

Una aproximación más eficiente para dar solución exacta al PVVG, es el algoritmo de ramificación y corte (Fischetti et al., 1995). Utilizando este algoritmo, se ha logrado obtener soluciones a varias instancias, con un tamaño de hasta 89 grupos. Sin embargo, resolver instancias de mayor tamaño de forma óptima, es todavía una tarea muy complicada de realizar, aun con el poder de computo actual.

Posteriormente, se propone una heurística sofisticada llamada GI^3 (Generalización de inicialización, Inserción y Mejora) (Renaud & Boctor, 1998). GI^3 es una generalización de la heurística I^3 presentado por Renaud et al. (1996). Esta heurística se compone de tres fases: la inicialización, en el que se construye una solución parcial, la inserción de un nodo de clústeres no visitado a la ruta hasta que se haya completado, y una fase que mejora la solución. El autor se compara con la heurística propuesta por Fischetti et al. (1997), en la que de 36 casos utilizados GI^3 entrega una mejor solución en 20 de los 36 casos, utilizando un tiempo computacional en promedio de 83,9

segundos.

Los algoritmos evolutivos también son utilizados para resolver al PVVG, uno de ellos es un algoritmo aleatorio de clave genética (RKGA) hibridado con una búsqueda local (Snyder & Daskin, 2006). Se utiliza codificación de clave aleatoria para representar las soluciones. La ventaja de este esquema de codificación es mantener la viabilidad de las soluciones durante cruce y mutación. En un conjunto de 41 problemas de prueba estándar con distancias simétricas y hasta 442 nodos, la heurística encuentra soluciones que son óptimas en la mayoría de los casos y está dentro del 1 % del óptimo en gran parte de los problemas, exceptuando los de mayor dimensión, con tiempos de ejecución de 10 segundos en promedio.

Posteriormente, se propone un algoritmo memético, un nuevo procedimiento de cruce que se basa en *large neighborhood search* (Bontoux et al., 2010). Con el objetivo de mejorar el rendimiento, se hibrididan el algoritmo con procedimientos de búsqueda local, es decir *2-opt, 3-opt, de Lin-Kernighan* y mover procedimientos. Los resultados muestran que es un algoritmo con buen rendimiento, donde los 41 casos de prueba se abordan de manera óptima y en 37 de esos casos, la solución óptima se encuentra en cada ejecución del algoritmo propuesto.

Uno de los últimos trabajos es un algoritmo evolutivo inspirado por la competencia imperialista, llamado algoritmo competitiva imperialista (ICA). ICA tiene mejor rendimiento que los algoritmos genéticos y optimización por enjambre de partículas, el cual es mejorado empleando un nuevo esquema de codificación, un nuevo método de cruce y el nuevo mecanismo de mejora de los planes de desarrollo llamado imperialistas, el cual permite que el algoritmo sea más eficaz para explorar todo el espacio de soluciones (Ardalan et al., 2015). ICA obtiene mejores rsultados en promedio en 27 casos de 53 superando el algoritmo de Snyder & Daskin (2006). En otras obras, el ICA se aplica al problema de localización cubo por Mohammadi et al. (2014), el suministro de diseño de la red de la cadena por Devika et al. (2014), el problema de flujo de potencia con funciones de costos no lisos por Ghasemi et al. (2014), entre otros.

CAPÍTULO 3. DISEÑO DE LA EXPERIMENTACIÓN

La experimentación se lleva a cabo ejecutando los 3 problemas explicados en los capítulos anteriores, considerando tres etapas distintas. La tabla 3.1 presenta de manera esquemática el diseño experimental completo. La primera columna de la tabla representa cada uno de los problemas y la primera fila corresponde a las distintas etapas de la experimentación realizada. En la segunda columna se encuentran los algoritmos redescubiertos, basados en sus componentes elementales para los problemas básicos que se encuentran en la literatura mediante la PG. La tercera columna, que corresponde a la versión generalizada, presenta los algoritmos generados mediante la PG utilizando los componentes elementales que permitieron redescubrir los algoritmos de la primera columna. Finalmente, la cuarta columna corresponde a algoritmos que son encontrados mediante la PG, considerando los componentes elementales y de mejora o constructivos que permiten refinar los resultados.

Tabla 3.1: Resumen del diseño experimental.

	Etapa I	Etapa II	Etapa III
PCMR	$A_{11}(g_1)$	$A_{12}(g_1, g_2, g_3)$	$A_{13}(g_1, g_2, g_3)$
PACMG	$A_{21}(g_1)$	$A_{22}(g_1, g_2, g_3, g_4)$	$A_{23}(g_1, g_2, g_3, g_4)$
PVVG		$A_{32}(g_1, g_2)$	$A_{33}(g_1, g_2)$

La estrategia seguida para obtener los algoritmos que se presentan en la tabla 3.1, considera la incorporación gradual de nuevos terminales que permiten encontrar mejores resultados computacionales para cada uno de los problemas estudiados. En el caso de los algoritmos de la *Etapa I*, se utilizan los componentes elementales del algoritmo de Prim y Dijkstra, además de componentes adicionales que permiten dar diversidad a la evolución los cuales son utilizados como terminales en la GAA. Los algoritmos de la *Etapa II* utilizan los mismos terminales que en el caso de la *Etapa I*, pero adaptados a las características propias de cada problema en particular. Para los algoritmos de la *Etapa III*, son utilizados los mismos terminales de la *Etapa II*, además de terminales de mejora que permiten refinar los resultados obtenidos. Por ejemplo, el término $A_{23}(g1, g2, g3, g4)$ corresponde a un algoritmo que es encontrado para el problema 2 (PACMG) en el experimento 3 (versión generalizada con terminales de refinamiento), considerando los grupos de instancias $g1$, $g2$, $g3$ y $g4$.

3.1 LÓGICA DEL DISEÑO

Para realizar la evaluación propuesta se desarrollan ocho experimentos generando algoritmos utilizando la PG, de los cuales dos corresponden a redescubrir los algoritmos polininomiales Dikjstra correspondientes al PCM y Prim, que da solución al PACCM utilizando sus componentes elementales como terminales. Posteriormente tres experimentos que consisten en utilizar los componentes elementales del algoritmo Dijkstra para el PCMR y Prim para el PACMG y PVVG. Finalmente, tres experimentos que utilicen componentes elementales de los algoritmos polinomiales, más algunas componentes de refinamiento que permiten mejorar los resultados. Los experimentos relacionados a la PG siguen la estructura utilizada por otros autores para la GAA (Contreras Bolton et al., 2013; Drake et al., 2014; Parada et al., 2015). Los experimentos son los siguientes:

- **Experimento 1**: tiene por objetivo realizar la GAA utilizando la PG para redescubrir el algoritmo Dijkstra.
- **Experimento 2**: tiene por objetivo realizar la GAA utilizando la PG para redescubrir el algoritmo Prim.
- **Experimento 3**: tiene por objetivo realizar la GAA utilizando la PG implementando los componentes elementales del algoritmo de Dijkstra, los cuales se adaptan para el PCMR.
- **Experimento 4**: tiene por objetivo realizar la GAA utilizando la PG implementando componentes elementales del algoritmo de Prim, los cuales se adaptan para el PACMG.
- **Experimento 5**: tiene por objetivo realizar la GAA utilizando la PG implementando componentes elementales del algoritmo de Prim, los cuales se adaptan para el PVVG.
- **Experimento 6**: tiene por objetivo realizar la GAA utilizando la PG ocupando los mismos componentes que en el experimento 3 y componentes de mejora para el PCMR.
- **Experimento 7**: tiene por objetivo realizar la GAA utilizando la PG ocupando los mismos componentes que en el experimento 4 y componentes de mejora para el PACMG.
- **Experimento 8**: tiene por objetivo realizar la GAA utilizando la PG ocupando los mismos componentes que en el experimento 5 y componentes de mejora para el PVVG.

3.2 METODOLOGÍA DEL EXPERIMENTO

Para el desarrollo de este experimento se utiliza la PG tradicional que fue descrita en la sección 2.1.2. Este experimento está dividido en dos etapas: proceso de evolución y el proceso de evaluación.

3.2.1 El proceso evolutivo

Se considera un proceso de aprendizaje para identificar la combinación adecuada de componentes que permitan resolver el problema. Para realizar el proceso evolutivo, son necesarios los siguientes pasos:

- Definir una estructura de datos acorde al problema.
- Definir un conjunto de funciones y terminales que cumplan con las propiedades de suficiencia y clausura.
- Construir una función de evaluación acorde al problema para evaluar el rendimiento de los algoritmos generados.
- Seleccionar un conjunto de instancias del problema para adaptar los algoritmos a generar.
- Determinar los métodos de evolución y el valor de los parámetros de control del proceso evolutivo.
- Ejecutar el proceso evolutivo un número determinado de veces y recopilar estadísticas de los individuos generados.
- Seleccionar un conjunto de individuos para ser estudiados.

3.2.2 El proceso de evaluación

El proceso de evaluación mide el desempeño computacional que poseen los algoritmos generados en el proceso evolutivo. El desempeño de los algoritmos se determina a partir de que tan efectivos son para resolver el problema y cuánto tiempo tardan en hacerlo. Para esto se selecciona un conjunto de instancias distinto al grupo utilizado en la evolución. Los algoritmos son evaluados en este nuevo conjunto y para medir su calidad se utiliza el error relativo promedio(ERP), que consiste en el error relativo de los algoritmos obtenidos. Para cada grupo de instancias de evaluación S, se determina el porcentaje promedio por el cual el beneficio obtenido z_i se encuentra distanciado de la mejor solución o_i para cada instancia del conjunto S. Esto se representa en la ecuación 3.1, donde n_s representa el número de instancias del conjunto S.

$$ERP_s = \frac{1}{n_s} \cdot \sum_{i=1}^{n_s} \frac{z_i - o_i}{z_i} \cdot 100\,\% \tag{3.1}$$

3.3 CONSIDERACIONES GENERALES

Para trabajar con la PG en los ocho experimentos, se tienen en consideración algunos aspectos comunes tales como: el diseño, construcción, parametrización, calibración y *testing* que se realizan previo a la ejecución de la PG. Esto debido a que es un proceso iterativo en cada experimento, y los efectos de un cambio en el diseño deben implementarse junto con realizar *testing* para conocer sus efectos reales.

Se realiza a continuación una descripción de las consideraciones.

3.3.1 Resultados

En los experimentos se busca encontrar algoritmos que solucionen los problemas planteados en la hipótesis, adicionalmente, estos resultados deben dar resultados similares o mejores a los

obtenidos en trabajos previos.

3.3.2 Instancias

En los experimentos se utilizan instancias disponibles en la *web* para uso investigativo y reconocidas por la comunidad científica asociada al área, las cuales difieren en cantidad para cada problema.

3.3.3 Ejecuciones

Aunque la PG utiliza generación aleatoria de números, es decir, una semilla distinta para cada ejecución, se observa que bastan tres ejecuciones para probar que los resultados son estadísticamente verdaderos. Según Cantú-Paz & Goldberg (2003); Li & Ciesielski (2004) no es necesario realizar múltiples ejecuciones, ya que esto está sujeto a varios factores, como la variabilidad de las funciones, terminales, el problema en sí, entre otros. Es por esto que se realizó un *test* estadístico, utilizando la herramienta STATA (*Data analysis and statistical software*) para comprobar si los datos se encuentran normalizados, para luego analizar si existe diferencia entre los resultados entregados por cada una de las ejecuciones realizadas.

3.4 ESTRUCTURAS DE DATOS DE LOS PROBLEMAS

Los algoritmos generados fueron construidos utilizando varias estructuras de datos. Las estructuras se diseñaron a partir de las caracteristicas de cada problemática y sus terminales y funciones. A continuación se dan a conocer los componentes de la estructura de datos para cada uno de los problemas:

PCMR

- *Listas de Adyacencia de Grafo unidireccional (LAGU)*: Se guarda en esta lista de adyacencia, el grafo correspondiente a las instancias de evolución de los algoritmos. Este dato es estático, pues solo corresponde a información que deben leer los individuos y resolver durante la evolución.
- *Listas de incidencia de Grafo unidireccional (LIGU)*: Se guarda en esta lista de incidencia, el grafo correspondiente a las instancias que es usada en la evolución de los algoritmos, donde se busca almacenar las aristas que llegan a los nodos. Este dato es estático, pues solo corresponde a información que deben tomar los individuos y resolver durante la evolución.
- *Lista de etiquetas de solución (LES)*: Lista con las etiquetas del grafo. Se tiene una lista de etiquetas de tamaño DIM_i, siendo i la instancia correspondiente de la etiqueta. Esta se inicializa con los valores de recursos y costo en infinito y con antecesor nulo.
- *Lista de nodos sin revisar (LNSR)*: Posee todos los nodos que no han sido etiquetados, vale decir, no se ha cambiado en la estructura *LES*.
- *Último (U)*: Es el último nodo marcado que fue de menor costo y que, a partir de él, se siguen etiquetando nodos.

PACMG

- Matriz de adyacencia (MAAC): En esta estructura se almacenan las distancias entre nodos, considerando la distancia euclidiana estipulado en las instancias. Las dimensiones de esta matriz son de nxn, dónde n es la cantidad de nodos.
- Mapa de relación cluster-nodo (MACN): Esta estructura almacena la relación existente entre el cluster y las características del nodo, además de almacenar por cada nodo, todos los posibles nodos con los que se puede conectar.
- Listas de listas de la solución (LLS): En esta estructura se almacena el árbol de solución, donde cada posición i de la lista (raiz), contiene los nodos asociados i (hijos).
- Lista de Solución (LSAC): Almacena una lista de enteros, siendo estos, los nodos incluidos en la solución por cada cluster.

PVVG

- Matriz de adyacencia (MAVV): En esta estructura se almacenan las distancias entre nodos, considerando la distancia euclidiana estipulado en las instancias. Las dimensiones de esta matriz son de NxN, dónde N es la cantidad de nodos (Referencia Parada Teoría de grafos).
- Lista de nodos (LN): En esta estructura se almacenan estructura de nodos. Dicha estructura posee las características primordiales de un nodo: id, coordenadas y clúster al cual pertenece el nodo.
- Listas de adyacencia de clúster (LAC): Es una lista de listas. Cada posición i de la lista, contiene una lista de nodos asociados al clúster $i + 1$ (Referencia Montgomery diseño y análisis).
- Lista de nodos cercanos al centro (LNCC): Se construye una lista de nodos ordenados de menor a mayor según el centro de la instancia. El tamaño de esta estructura solo almacena $n/2$ nodos.
- Lista de nodos cercanos al centro (LNLC): Se construye una lista de nodos ordenados de menor a mayor según el centro de la instancia. El tamaño de esta estructura solo almacena $n/2$ nodos.
- Lista de Solucion (LSVV): Almacena una lista de enteros, siendo estos, los nodos incluidos en la solución. El circuito se cierra conectado conectando el último de nodo de la lista con el primero de esta esta estructura.

3.5 IDENTIFICACIÓN DE COMPONENTES ELEMENTALES

En esta sección se presenta el análisis para la obtención de los componentes elementales de los problemas fáciles.

3.5.1 Descomponiendo Algoritmo Dijkstra

El algoritmo de Dijkstra permite resolver de forma polinomial el PCM y en el algoritmo 3.1 se presenta su pseudocódigo.

Algoritmo 3.1: Algoritmo de Dijkstra

```
1:  Inicializar Etiquetas con antecesor nulo y costo INFINITO
2:  Inicializar una lista L con Todos los vétices del Grafo
3:  Etiquetar Vértice Inicial=[-,0]
4:  Marcar X = vértice Inicial
5:  Eliminar Vértice Incial de L
6:  while Quedan Vértices en lista L do
7:      Actualizar etiquetas vecinos de X solo si el costo es menor.
8:      Buscar en L, el nodo con Etiqueta con costo más bajo (Nmin).
9:      Macar X=Nmin.
10:     Eliminar Nmin de L
11: end while
```

En el algoritmo 3.1, de la línea 1 hasta la 5 se inicializan las estructuras de datos, mientras que en la línea 6 se puede observar el ciclo que permite realizar las iteraciones hasta encontrar el camino mínimo. La línea 7 es la encargada de actualizar las etiquetas de los vecinos del vértice seleccionado, la línea 8 busca en la lista de nodos disponibles el vértice seleccionado, la línea 9 es la encargada de marcar el vértice en la solución y la línea 10 es la que elimina el vértice marcado de la lista de vértices disponibles.

De esta forma el algoritmo se ejecuta hasta que todos los vértices hayan sido marcados. Esto se cumple con el criterio de término del ciclo *while* del pseudocódigo, ya que a medida que se extraen los nodos de la lista *L*, se van marcando y etiquetando los vértices del grafo.

Se puede observar que los componentes elementales del algoritmo están en la línea 6, que es la que permite que el ciclo siga hasta encontrar la solución, línea 7, encargada de actualizar las etiquetas y la línea 9 marcar el vértice en la solución. Generando así las siguientes funciones elementales:

- **Marca costo menor**: marca el vértice de costo menor en la solución.
- **Etiquetar Dijkstra**: Actualiza las etiquetas del problema.

- **Quedan vertices**: Verifica si ya se encontró el vértice final.

3.5.2 Descomponiendo Algoritmo Prim

El algoritmo de Prim permite resolver de forma polinomial el problema del aŕbol de cobertura de costo mínimo y en el algoritmo 3.2 se presenta su pseudocódigo.

Algoritmo 3.2: Algoritmo de Prim

```
1: Inicializar una lista L con Todos los vétices del árbol
2: Inicializar el árbol con un vértice arbitrario.
3: while Quedan Vértices sin utilizar do
4:     Añadir la arista de menor peso que se conecte con el árbol.
5:     Eliminar la arista de menor peso de la lista L.
6: end while
```

En el algoritmo 3.2, en las líneas 1 y 2 se inicializan las estructuras de datos que almacena el árbol de solución, mientras que en la línea 3 se puede observar el ciclo que permite realizar las iteraciones hasta encontrar el aŕbol de cobertura de costo mínimo, la línea 4 es la que va agregando cada vértice al árbol, y finalmente la línea 5 elimina el vértice agregado al árbol. De esta forma el algoritmo se ejecuta hasta que todos los vértices hayan sido agregados al árbol.

Se puede observar que los componentes elementales del algoritmo están en la línea 3, que es la que permite que el ciclo siga hasta encontrar la solución y la línea 4, que permite ir construendo el árbol agregando la arísta de costo menor. Generando así los siguientes funciones elementales para el redescubrimiento:

- **Quedan Cluster**: Verifica si aún existen vértices disponibles que no sean parte de la solución.
- **IniAristaMenorMasNodos**: Agrega dos vértices iniciales a la solución parcial, los cuales son parte de la arista de menor costo.
- **constAristaMenor**: Agrega un nuevo vértice a la solución parcial. El vértice seleccionado es aquel que tiene la arista de menor costo entre los nodos que pertenecen a la solución.

3.6 DEFINICIÓN DE FUNCIONES Y TERMINALES

Se define como funciones todos aquellos componentes que requieren de argumentos y, como terminales, todos aquellos componentes que no lo requieran, donde además, tanto funciones como terminales deben retornar algún valor (Koza, 1992b).

Las funciones y terminales son las operaciones elementales de las estructuras de datos anteriormente definidas. Por lo tanto, su definición es fundamental para generar algoritmos con la capacidad de utilizar dichas estructuras con el fin de alcanzar una solución que minimice el costo para el PCMR, PACMG y PVVG. Se construyen terminales en base a los componentes elementales de los problemas fáciles, además de algunos terminales de refinamiento, y funciones que permitan operar en diversas combinaciones sobre estos terminales.

3.6.1 Conjunto de funciones

Las funciones que conforman los algoritmos generados para los ocho experimentos contienen instrucciones básicas utilizadas en su mayoría por todos los lenguajes de programación. Desde el punto de vista de la PG, las funciones corresponden a los nodos internos del árbol (Koza & Poli, 2005) y estas se presentan a continuación:

- *While(a1,a2):* Realiza el argumento *a2* mientras el argumento *a1* retorne verdadero. Lo característico de este *While* es tiene como límite de iteraciones el número total de elementos. Esta condición son necesaria debido a que el argumento *a1* del *While* puede retornar siempre verdadero, rompiendo así la propiedad de clausura de la evolución (Koza et al., 2003). La función retorna verdadero si al menos realiza un cambio en el total de iteraciones realizadas. Si en ninguna de las iteraciones ejecutadas realiza un cambio en la estructura de datos de solución, retorna falso.
- *If_then(a1,a2):* Se ejecuta el argumento *a1*. Si éste retorna verdadero, se ejecuta el argumento *a2* y la función retorna verdadero. En caso contrario, no se ejecuta *a2* y la función retorna falso.
- *And(a1,a2):* Se ejecuta el argumento *a1* y el argumento *a2*. La función retorna verdadero si

a1 y *a2* son verdaderos, en cualquier otro caso retorna falso.

- *Or(a1,a2):* Se ejecuta el argumento *a1* y el argumento *a2*. La función retorna falso si *a1* y *a2* son falsos, en cualquier otro caso retorna verdadero.
- *Not(a1):* Se ejecuta el argumento *a1*. La función retorna la negación lógica del resultado del argumento.
- *Equal(a1,a2):* Se ejecuta el argumento *a1* y se ejecuta el argumento *a2*. Si el argumento *a1* retorna lo mismo que el argumento *a2*, la función retorna verdadero, en caso contrario, retorna falso.

3.6.2 Conjunto de terminales

Los terminales son funciones diseñadas para cada uno de los problemas. De acuerdo a la definición de la PG, un terminal es un nodo hoja (Koza & Poli, 2005). Cada uno de los terminales es una heurística elemental capaz de modificar la estructura de datos definida generando nuevas soluciones. Los terminales a utilizar son:

Redescubrimiento de Dijkstra

- *MarcaCostoMenor*: Elimina un vértice de la lista de vértices no marcados y es agregado como último vértice marcado. El vértice seleccionado es aquel que posea la etiqueta con menor costo acumulado. Retorna verdadero si se logra marcar un vértice, falso en el caso contrario.
- *EtiquetarDijkstra*: Cambia las etiquetas de los vecinos a partir del último vértice marcado. Si un vértice vecino ya fue etiquetado, se selecciona la etiqueta que posea el menor costo acumulado. Si un vértice fue marcado anteriormente, no puede volver a ser etiquetado. Retorna verdadero si se logra etiquetar un vértice, falso en el caso contrario.
- *QuedanVertices*: Verifica si aún existen vértices que no han sido marcados. Retorna verdadero si quedan vértices por recorrer, falso en el caso contrario.

- *Etiquetador*: Cambia las etiquetas de los vecinos del último vértice marcado. Los vecinos deben ser vértices que no han sido marcados. Retorna verdadero si se logra cambiar las etiquetas de los vecinos, falso en el caso contrario.
- *MarcaCostoMayor*: Elimina un vértice a la lista de vértices no marcados y es agregado como último vértice marcado. El vértice seleccionado es aquel que posea la etiqueta con mayor costo acumulado. Retorna verdadero si se logra marcar un vértice, falso en el caso contrario.
- *MarcarGradoMenor*: Elimina un vértice a la lista de vértices no marcados y es agregado como último vértice marcado. El vértice seleccionado es aquel que posea el menor grado. Retorna verdadero si se logra marcar un vértice, falso en el caso contrario.
- *MarcarGradoMayor*: Elimina un vértice a la lista de vértices no marcados y es agregado como último vértice marcado. El vértice seleccionado es aquel que posea el mayor grado. Retorna verdadero si se logra marcar un vértice, falso en el caso contrario.

Redescubrimiento de Prim

- *QuedanCluster*: Verifica si aún existen vértices disponibles que no sean parte de la solución. Retorna verdadero si aún quedan clúster por seleccionar, falso en el caso contrario.
- *constAristaMenor*: Agrega un nuevo vértice a la solución parcial. El vértice seleccionado es aquel que tiene la arista de menor costo entre los nodos que pertenecen a la solución. Retorna verdadero si se logra agregar el vértice a la solución, falso en el caso contrario.
- *IniAristaMenor*: Agrega dos vértices iniciales a la solución parcial, los cuales son parte de la arista de menor costo. Retorna verdadero si se logra agregar los dos vértices a la solución, falso en el caso contrario.
- *IniAristaMayor*: Agrega dos vértices iniciales a la solución parcial, los cuales son parte de la arista de mayor costo. Retorna verdadero si se logra agregar los dos vértices a la solución, falso en el caso contrario.
- *constAristaMayor*: Agrega un vértice a la solución parcial. El vértice seleccionado es aquel que tiene la arista de mayor costo que conecte a un clúster que no se encuentre en la solución. Retorna verdadero si se logra agregar el vértice a la solución, falso en el caso contrario.

- *constKruskalMenor*: Agrega dos vértices que no pertenecen a la solución parcial. Los vértices forman parte de la arista de menor costo encontrada entre los clusters que no se encuentre en la solución. Retorna verdadero si se logra agregar los dos vértices a la solución, falso en el caso contrario.
- *constKruskalMayor*: Agrega dos nuevos vértices a la solución parcial. Los vértices forman parte de la arista de mayor costo encontrada entre los clusters que no forman parte de la solución. Retorna verdadero si se logra agregar los dos vértices a la solución, falso en el caso contrario.

PCMR

Para el experimento 3 se utilizan los mismos terminales que para el redescubrimiento del Djkstra, pero acorde a las restricciónes, características y estuctura del PCMR:

- *MarcaCostoMenor*: Elimina un vértice a la lista de vértices no marcados y es agregado como último vértice marcado. El vértice seleccionado es aquel que posea la etiqueta con menor costo acumulado.
- *MarcaPesoMenor*: Elimina un vértice a la lista de vértices no marcados y es agregado como último vértice marcado. El vértice seleccionado es aquel que posea la etiqueta con menor peso cumulado.
- *MarcarGradoMenor*: Elimina un vértice a la lista de vértices no marcados y es agregado como último vértice marcado. El vértice seleccionado es aquel que posea el menor grado.
- *MarcarGradoMayor*: Elimina un vértice a la lista de vértices no marcados y es agregado como último vértice marcado. El vértice seleccionado es aquel que posea el mayor grado.
- *MarcarPonderado*: Elimina un vértice a la lista de vértices no marcados y es agregado como último vértice marcado. El vértice seleccionado es aquel que posea el menor valor que resulte de la multiplicación entre su peso acumulado y su costo acumulado.
- *Etiquetador*: Cambia las etiquetas de los vecinos del último vértice marcado. Los vecinos deben ser vértices que no han sido marcados.

- *EtiquetarDijkstra*: Cambia las etiquetas de los vecinos a partir del último vértice marcado. Si un vértice vecino ya fue etiquetado, se selecciona la etiqueta que posea el menor costo acumulado. Si un vértice fue marcado anteriormente, no puede volver a ser etiquetado.
- *EtiquetarVerificaPeso*: Cambia las etiquetas de los vecinos a partir del último vértice marcado verificando que el peso acumulado de la etiqueta no supere el peso máximo permitido. Si un vértice vecino ya fue etiquetado, se deja el que tenga menor costo acumulado. Si un vértice fue marcado anteriormente, no puede volver a ser etiquetado.

Para el experimento 6 se utilizan los mismos terminales que para el experimento 3 y se agregan dos terminales de refinamiento:

- *MejorarDijkstra*: Cambia la etiqueta del último vértice marcado a partir de los vértice incidentes en él. Se realiza el cambio si el costo acumulado de la nueva etiqueta es menor al que va a reemplazar.
- *MejorarDijkstraPeso*: Cambia la etiqueta del último vértice marcado a partir de los vértice incidentes en él verificando que el peso acumulado de la etiqueta no supere el peso máximo permitido. Se realiza el cambio si el costo acumulado de la nueva etiqueta es menor al que va a reemplazar.

PACMG

Para el experimento 4 se utilizan los mismos terminales que para el redescubrimiento de Prim, pero acorde a las restricciónes, características y estuctura del PACMG:

- *QuedanCluster*: Verifica si aún existen vértices disponibles que no sean parte de la solución. Retorna verdadero si aún quedan clúster por seleccionar, falso en el caso contrario.
- *constAristaMenor*: Agrega un nuevo vértice a la solución parcial. El vértice seleccionado es aquel que tiene la arista de menor costo entre los nodos que pertenecen a la solución. Retorna verdadero si se logra agregar el vértice a la solución, falso en el caso contrario.
- *IniAristaMenor*: Agrega dos vértices iniciales a la solución parcial, los cuales son parte de la arista de menor costo. Retorna verdadero si se logra agregar los dos vértices a la solución, falso en el caso contrario.

- *IniAristaMayor*: Agrega dos vértices iniciales a la solución parcial, los cuales son parte de la arista de mayor costo. Retorna verdadero si se logra agregar los dos vértices a la solución, falso en el caso contrario.
- *constAristaMayor*: Agrega un vértice a la solución parcial. El vértice seleccionado es aquel que tiene la arista de mayor costo que conecte a un clúster que no se encuentre en la solución. Retorna verdadero si se logra agregar el vértice a la solución, falso en el caso contrario.
- *constKruskalMenor*: Agrega dos vértices que no pertenecen a la solución parcial. Los vértices forman parte de la arista de menor costo encontrada entre los clusters que no se encuentre en la solución. Retorna verdadero si se logra agregar los dos vértices a la solución, falso en el caso contrario.
- *constKruskalMayor*: Agrega dos nuevos vértices a la solución parcial. Los vértices forman parte de la arista de mayor costo encontrada entre los clusters que no forman parte de la solución. Retorna verdadero si se logra agregar los dos vértices a la solución, falso en el caso contrario.

Para el experimento 7 se utilizan los mismos terminales que para el experimento 4 y se agregan cinco terminales de refinamiento:

- *MejorarConexionAltura2*: mejora la solución actual reestructurando un subárbol de altura 2 perteneciente a la solución mediante el uso del prim modificado. Posteriormente se conecta al vértice del que fue desconectado al comienzo.
- *MejorarConexionAltura3*: mejora la solución actual reestructurando un subárbol de altura 3 perteneciente a la solución mediante el uso del prim modificado. Posteriormente se conecta al vértice del que fue desconectado al comienzo.
- *MejorarConexionAltura4*: mejora la solución actual reestructurando un subárbol de altura 4 perteneciente a la solución mediante el uso del prim modificado. Posteriormente se conecta al vértice del que fue desconectado al comienzo.
- *MejoraConexiónCluster*: Busca una mejora de la solución actual desplazando las aristas de un cluster por todos sus vértices, manteniendo el vértice que mejore la solución. Este proceso finaliza cuando se recorren todos los clúster.
- *CortarConexiónSubárbolMayorAltura*: Corta la rama con mayor altura de la solución actual.

PVVG

Para el experimento 5 se utiliza una adaptación de los componentes elementales del redescubrimiento de Prim, pero acorde a las restricciónes, características y estuctura del PVVG:

- *AgregarMejorVecino*: Busca en los vertices disponibles, el vertice que agregue el menor costo al circuito y es agregado al final de éste, donde el clúster al que pertenece el vértice no forme parte de la solución parcial. Retorna verdadero si agrega el vertice, falso en caso contrario.
- *AgregarPeorVecino*: Busca en los vertices disponibles, el vertice que agregue el mayor costo al circuito y es agregado al final de éste, donde el clúster al que pertenece el vértice no forme parte de la solución parcial. Retorna verdadero si agrega la ciudad, falso en caso contrario.
- *AgregarCercaCentro*: Busca el vértice más cercano a las coordenadas del centro que se encuentre disponible. Si encuentra un vértice, éste es agregado al final del circuito, donde el clúster al que pertenece el vértice no forme parte de la solución parcial. Retorna verdadero si agrega la ciudad, falso en caso contrario.
- *AgregarLejosCentro*: Busca el vértice más lejano a las coordenadas del centro que se encuentre disponible. Si encuentra un vértice, éste es agregado al final del circuito, donde el clúster al que pertenece el vértice no forme parte de la solución parcial. Retorna verdadero si agrega la ciudad, falso en caso contrario.
- *AgregarCercano*: Busca el vértice que al ser agregado en cualquier posición del circuito, sume el menor costo a éste. Donde el clúster al que pertenece el vértice no forme parte de la solución parcial. Por ejemplo, se tiene el circuito $[A, B, C]$ donde ingresa D, siendo $A->D<B->D<C->D$. El resultado luego de la inserción es $[A, D, B, C]$. Retorna verdadero si agrega la ciudad, falso en caso contrario.
- *AgregarLejano*: Busca el vértice que al ser agregado en cualquier posición del circuito, sume el mayor costo a éste. Donde el clúster al que pertenece el vértice no forme parte de la solución parcial. Por ejemplo, se tiene el circuito $[A, B, C]$ donde ingresa D, siendo $A->D<B->D<C->D$. El resultado luego de la inserción es $[A, B, C, D]$. Retorna verdadero si agrega la ciudad, falso en caso contrario.

- *EliminarPeorArco*: Busca en el circuito dos vertices que produzcan el peor arco (el mayor costo al circuito). Ambos vertices correspondientes al peor arco son eliminadas. Retorna verdadero si logra eliminar el arco y falso en caso contrario.

Para el experimento 8 se utilizan los mismos terminales que para el experimento 5 y se agregan tres terminales de refinamiento:

- *Invertir*: Cambia el orden de los vertices sólo en el caso en que este cambio produzca alguna mejora. El cambio se realiza mediante la inversión de las posiciones de los extremos hacia adentro. Por ejemplo, la primera se invierte con la última, la segunda con la penúltima y así sucesivamente. Retorna verdadero si logra mejorar el costo del circuito y falso en caso contrario.
- *mejoraLKH*: Optimizador que utiliza la heurística LKH. Retorna verdadero si puede realizar alguna mejora al circuito y falso en caso contrario.

REFERENCIAS BIBLIOGRÁFICAS

Affenzeller, M. (2001). Connectionist models of neurons, learning processes, and artificial intelligence: 6th international work-conference on artificial and natural neural networks, iwann 2001 granada, spain, june 13–15, 2001 proceedings, part 1. (pp. 594–601).

Ahandani, M. A., Baghmisheh, M. T. V., Zadeh, M. A. B., & Ghaemi, S. (2012). Hybrid particle swarm optimization transplanted into a hyper-heuristic structure for solving examination timetabling problem. *Swarm and Evolutionary Computation*, *7*, 21 – 34.

AL, H. (1969). Record balancing problem-a dynamic programming solution of a generalized traveling salesman problem. *Revue Francaise D Informatique De Recherche Operationnelle*, *3*(NB 2), 43.

Applegate, D. L., Bixby, R. E., Chvatal, V., & Cook, W. J. (2009). *The Traveling Salesman Problem: A Computational Study*. Princeton, NJ, USA: Princeton University Press.

Ardalan, Z., Karimi, S., Poursabzi, O., & Naderi, B. (2015). A novel imperialist competitive algorithm for generalized traveling salesman problems. *Applied Soft Computing*, *26*, 546–555.

Ben-Arieh, D., Gutin, G., Penn, M., Yeo, A., & Zverovitch, A. (2003). Transformations of generalized atsp into atsp. *Operations Research Letters*, *31*(5), 357–365.

Bolton, C. C., Gatica, G., & Parada, V. (2013). Automatically generated algorithms for the vertex coloring problem. *PloS one*, *8*(3), e58551.

Bontoux, B., Artigues, C., & Feillet, D. (2010). A memetic algorithm with a large neighborhood crossover operator for the generalized traveling salesman problem. *Computers & Operations Research*, *37*(11), 1844–1852.

Burke, E. K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Ozcan, E., & Qu, R. (2013). Hyper-heuristics. *J Oper Res Soc*, *64*(12), 1695–1724.

Burke, E. K., Hyde, M., Kendall, G., & Woodward, J. (2010). A genetic programming hyper-heuristic approach for evolving 2-D strip packing heuristics. *IEEE Transactions on Evolutionary Computation*, *14*(6), 942–958.

Cantú-Paz, E., & Goldberg, D. E. (2003). Genetic and evolutionary computation — gecco 2003: Genetic and evolutionary computation conference chicago, il, usa, july 12–16, 2003 proceedings, part i. (pp. 801–812).

Carlyle, W. M., Royset, J. O., & Kevin Wood, R. (2008). Lagrangian relaxation and enumeration for solving constrained shortest-path problems. *Networks*, *52*(4), 256–270.

Contreras-Bolton, C., Gatica, G., Barra, C. R., & Parada, V. (2016). A multi-operator genetic algorithm for the generalized minimum spanning tree problem. *Expert Systems with Applications*, *50*, 1–8.

Contreras Bolton, C., Gatica, G., & Parada, V. (2013). Automatically generated algorithms for the vertex coloring problem. *PLoS ONE*, *8*(3), 1–9.
URL http://dx.doi.org/10.1371%2Fjournal.pone.0058551

Cook, W., Lovász, L., Seymour, P. D., et al. (1995). *Combinatorial optimization: papers from the DIMACS Special Year*, vol. 20. American Mathematical Soc.

Derrac, J., García, S., Molina, D., & Herrera, F. (2011). A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. *Swarm and Evolutionary Computation*, *1*(1), 3 – 18.

Desale, S., Rasool, A., Andhale, S., & Rane, P. (2015). Heuristic and meta-heuristic algorithms and their relevance to the real world: A survey. *International journal of computer engineering in research trends*, *2*, 296–304.

Devika, K., Jafarian, A., & Nourbakhsh, V. (2014). Designing a sustainable closed-loop supply chain network based on triple bottom line approach: A comparison of metaheuristics hybridization techniques. *European Journal of Operational Research*, *235*(3), 594–615.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, *1*(1), 269–271.

Drake, J. H., Hyde, M., Ibrahim, K., & Ozcan, E. (2014). A genetic programming hyper-heuristic for the multidimensional knapsack problem. *Kybernetes*, *43*(9/10), 1500–1511.
URL http://dx.doi.org/10.1108/K-09-2013-0201

Dror, M., Haouari, M., & Chaouachi, J. (2000). Generalized spanning trees. *European Journal of Operational Research*, *120*(3), 583–592.

Dumitrescu, I., & Boland, N. (2001). Algorithms for the weight constrained shortest path problem. *International Transactions in Operational Research*, *8*(1), 15–29.

Dumitrescu, I., & Boland, N. (2003). Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem. *Networks*, *42*(3), 135–153.

Ferreira, C. S., Ochi, L. S., Parada, V., & Uchoa, E. (2012). A grasp-based approach to the generalized minimum spanning tree problem. *Expert Systems with Applications*, *39*(3), 3526–3536.

Fischetti, M., González, J. J. S., & Toth, P. (1995). The symmetric generalized traveling salesman polytope. *Networks*, *26*(2), 113–123.

Fischetti, M., Salazar Gonzalez, J. J., & Toth, P. (1997). A branch-and-cut algorithm for the symmetric generalized traveling salesman problem. *Operations Research*, *45*(3), 378–394.

Ghasemi, M., Ghavidel, S., Rahmani, S., Roosta, A., & Falah, H. (2014). A novel hybrid algorithm of imperialist competitive algorithm and teaching learning algorithm for optimal power flow problem with non-smooth cost functions. *Engineering Applications of Artificial Intelligence*, *29*, 54–69.

Golden, B., Raghavan, S., & Stanojević, D. (2005). Heuristic search for the generalized minimum spanning tree problem. *INFORMS Journal on Computing*, *17*(3), 290–304.

Handler, G. Y., & Zang, I. (1980). A dual algorithm for the constrained shortest path problem. *Networks*, *10*(4), 293–309.

Haouari, M., & Chaouachi, J. S. (2006). Upper and lower bounding strategies for the generalized minimum spanning tree problem. *European Journal of Operational Research*, *171*(2), 632–647.

Hassin, R. (1992). Approximation schemes for the restricted shortest path problem. *Mathematics of Operations research*, *17*(1), 36–42.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. The MIT Press.

Joksch, H. C. (1966). The shortest route problem with constraints. *Journal of Mathematical analysis and applications*, *14*(2), 191–197.

Kansal, A. R., & Torquato, S. (2001). Globally and locally minimal weight spanning tree networks. *Physica A: Statistical Mechanics and its Applications*, *301*(1), 601–619.

Kouchakpour, P., Zaknich, A., & Bräunl, T. (2009). A survey and taxonomy of performance improvement of canonical genetic programming. *Journal Knowledge and Information Systems*, *21*, 1–39.

Koza, J. R. (1992a). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT press.

Koza, J. R. (1992b). *Genetic programming: on the programming of computers by means of natural selection*, vol. 1. MIT press.

Koza, J. R. (1999). Genetic programming III: Darwinian invention and problem solving. San Francisco: CA: Morgan Kaufmann.

Koza, J. R. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Norwell, MA, USA: Kluwer Academic Publishers.

Koza, J. R., Keane, M. A., Streeter, M. J., Mydlowec, W., Yu, J., & Lanza, G. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence.*. Springer US.

Koza, J. R., & Poli, R. (2005). *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, chap. Genetic Programming, (pp. 127–164). Boston, MA: Springer US.
URL http://dx.doi.org/10.1007/0-387-28356-0_5

Laporte, G., Asef-Vaziri, A., & Sriskandarajah, C. (1996). Some applications of the generalized travelling salesman problem. *Journal of the Operational Research Society*, *47*(12), 1461–1467.

Laporte, G., & Semet, F. (1999). Computational evaluation of a transformation procedure for the symmetric generalized traveling salesman problem. *INFOR*, *37*(2), 114.

Li, X., & Ciesielski, V. (2004). Analysis of genetic programming runs. http://goanna.cs.rmit.edu.au/~vc/papers/aspgp04.pdf.

Lien, Y.-N., Ma, E., & Wah, B. W.-S. (1993). Transformation of the generalized traveling-salesman problem into the standard traveling-salesman problem. *Information Sciences*, *74*(1), 177–189.

Lorenz, D. H., & Raz, D. (2001). A simple efficient approximation scheme for the restricted shortest path problem. *Operations Research Letters*, *28*(5), 213–219.

Lozano, L., & Medaglia, A. L. (2013). On an exact method for the constrained shortest path problem. *Computers & Operations Research*, *40*(1), 378–384.

Mohammadi, M., Torabi, S., & Tavakkoli-Moghaddam, R. (2014). Sustainable hub location under mixed uncertainty. *Transportation Research Part E: Logistics and Transportation Review*, *62*, 89–115.

Muhandiramge, R., & Boland, N. (2009). Simultaneous solution of lagrangean dual problems interleaved with preprocessing for the weight constrained shortest path problem. *Networks*, *53*(4), 358–381.

Myung, Y.-S., Lee, C.-H., & Tcha, D.-W. (1995). On the generalized minimum spanning tree problem. *Networks*, *26*(4), 231–241.

Nakagaki, T., Yamada, H., & Toth, A. (2001). Path finding by tube morphogenesis in an amoeboid organism. *Biophysical chemistry*, *92*(1), 47–52.

Noon, C. E., & Bean, J. C. (1991). A lagrangian based approach for the asymmetric generalized traveling salesman problem. *Operations Research*, *39*(4), 623–632.

Noon, C. E., & Bean, J. C. (1993). An efficient transformation of the generalized traveling salesman problem. *INFOR: Information Systems and Operational Research*, *31*(1), 39–44.

Papadimitriou, C. H., & Steiglitz, K. (1982). *Combinatorial optimization: algorithms and complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Parada, L., Herrera, C., Sepúlveda, M., & Parada, V. (2015). Evolution of new algorithms for the binary knapsack problem. *Natural Computing*, (pp. 1–13).

Poli, R., Langdon, W. B., & McPhee, N. F. (2008). *A Field Guide to Genetic Programming*. Lulu Press, Inc.

Pop, P. (2009). A survey of different integer programming formulations of the generalized minimum spanning tree problem. *Carpathian Journal of Mathematics*, *25*(1), 104–118.

Pop, P. C. (2002). *The generalized minimum spanning tree problem*. Twente University Press.

Reinelt, G. (1991). Tsplib - a traveling salesman problem library. *ORSA Journal on Computing*, *3*(4), 376–384.
URL `http://dx.doi.org/10.1287/ijoc.3.4.376`

Renaud, J., & Boctor, F. F. (1998). An efficient composite heuristic for the symmetric generalized traveling salesman problem. *European Journal of Operational Research*, *108*(3), 571–584.

Renaud, J., Boctor, F. F., & Laporte, G. (1996). A fast composite heuristic for the symmetric traveling salesman problem. *INFORMS Journal on Computing*, *8*(2), 134–143.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, *3*(3), 210–229.

Santos, L., Coutinho-Rodrigues, J., & Current, J. R. (2007). An improved solution algorithm for the constrained shortest path problem. *Transportation Research Part B: Methodological*, *41*(7), 756–771.

Smith-Miles, K., & Lopes, L. (2012). Measuring instance difficulty for combinatorial optimization problems. *Computers & Operations Research*, *39*(5), 875 – 889.

Snyder, L. V., & Daskin, M. S. (2006). A random-key genetic algorithm for the generalized traveling salesman problem. *European Journal of Operational Research*, *174*(1), 38–53.

Srivastava, S., Kumar, S., Garg, R., & Sen, P. (1969). Generalized traveling salesman problem through n sets of nodes. *CORS journal*, *7*, 97–101.

Talbi, E.-G. (2009). *Metaheuristics: from design to implementation*. NJ:Wiley.

Zhang, X., Zhang, Y., Hu, Y., Deng, Y., & Mahadevan, S. (2013). An adaptive amoeba algorithm for constrained shortest paths. *Expert Systems with Applications*, *40*(18), 7607–7616.

Zhu, X., & Wilhelm, W. E. (2012). A three-stage approach for the resource-constrained shortest path as a sub-problem in column generation. *Computers & Operations Research*, *39*(2), 164–178.