UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
Departamento de Ingeniería Informática

IMPLEMENTACIÓN DEL ALGORITMO VARIABLE NEIGHBORHOOD SEARCH PARA BÚSQUEDA DE ÓRDENES DE CONSTRUCCIÓN EFICIENTES EN STARCRAFT II: EXPERIENCIA ZERG

Brandon Estefan Nuñez Maturana

Profesor guía: Manuel Villalobos Cid

Tesis para optar al título de Ingeniero de Ejecución en Computación e Informática

Santiago – Chile
2021

RESUMEN

En la actualidad, la industria del entretenimiento ha crecido de manera increíble en temas económicos, cantidad de adeptos alrededor del mundo y también en términos de variedad de videojuegos (Practicodeporte, 2019), entre las muchas categorías de videojuegos que hay en la actualidad, existen los juegos RTS (*Real-Time Strategy*), los cuales, se caracterizan por ser muy competitivos y contar con una gran cantidad de adeptos. Es aquí donde destaca Starcraft II, el cual, es un juego que cuenta con muchísimas mecánicas y estrategias, además de una escena competitiva muy activa, es por esto que siempre los jugadores y profesionales del juego alrededor del mundo están en busca de formas de mejorar su rendimiento en partidas o de encontrar nuevos órdenes de construcción (*build-orders*), que les permitan obtener las unidades más fuertes y de forma más rápida, permitiendo así aumentar las probabilidades de victoria en cada partida. Para este cometido es que buscar aproximaciones de *build-orders* óptimos se convierte en algo clave, puesto que, permitiría mejorar en este ámbito, enriquecería la experiencia de juego y la gama de estrategias existentes; y es para este tipo de búsqueda de *build-orders* que este trabajo propone usar metaheurísticas de población para la creación de un modelo algorítmico capaz de resolver problemas complejos de optimización, como lo es el algoritmo de búsqueda de vecindades variables (VNS por sus siglas en inglés), el cual fue escogido ya que las metaheurísticas proporcionan buenas aproximaciones de óptimos globales en problemas de tipo NP-Hard.

Palabras Claves: Starcraft II, build-order, optimización, metaheurísticas, VNS, búsqueda de vecindades variables, variable neighborhood search, NP-hard, RTS, Real-Time Strategy.

Levántate y camina...después de todo, tienes dos piernas para hacerlo.
-Edward Elric

AGRADECIMIENTOS

Agradezco a mi Madre y mi Padre, por darme su apoyo cuando más lo necesitaba, dándome un tecito cada vez estaba mal y soportarme en esas largas noches de estrés, cansancio y mal humor.

Agradezco a mis amigos por ayudarme con los trabajos y tareas que no entendía, por cada junta de estudio en la biblioteca, por cada noche de juegos para pasar las penas, y por darme su tiempo para conversar y reírnos en los asientos del negocio de metalurgia, despejando la mente de la universidad que a veces agobia.

Agradecer a mi profesor guía Manuel Villalobos Cid, por su paciencia y disponibilidad durante el desarrollo de esta memoria, a pesar de que yo no fuera tan diligente algunas veces.

Agradezco a todos los que me han apoyado en el transcurso de mi carrera y me han impulsado a seguir mejorando, tanto profesores como personal del Departamento de Ingeniería Informática.

También agradezco a Peter Parker, por enseñarme desde pequeño a nunca rendirme por muy dura que parezca la situación, siempre hay que ponerse de pie.

Agradezco a AuronPlay por alegrarme los días en los semestres finales de mi carrera, cuando la pandemia estaba en su peor momento, él siempre supo como sacarme una sonrisa.

Finalmente agradezco a Francisca, mi novia, por darle color a mi vida y darme la motivación que necesito para levantarme cada mañana y las ganas de convertirme en un mejor hombre.

TABLA DE CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE ILUSTRACIONES

CAPÍTULO 1. INTRODUCCIÓN

1.1 ANTEDECEDENTES Y MOTIVACIÓN

Starcraft II es un juego de estrategia en tiempo real (*RTS* por sus siglas en inglés) y es uno de los máximos exponentes de este género de videojuegos (Huertos, 2019). En *Starcraft II*, los jugadores pueden elegir entre tres razas para poder jugar, *Protoss, Terran y Zerg.* Cada raza tiene características particulares y un árbol de tecnologías propio, de modo que para construir unidades o edificios de un nivel avanzado se deben cumplir con alguna precondición y contar con los recursos necesarios (tiempo, minerales, gas, etc.), de esta forma el jugador debe ser capaz de crear un ejército para derrotar a su rival. El modo de juego podría compararse a un tablero de ajedrez, pero extremadamente avanzado, donde el movimiento de las piezas varía entre una raza y otra, y donde los movimientos del enemigo no pueden ser vistos por el jugador, lo que lo convierte en un juego mucho más complejo, creando miles de millones de combinaciones de movimientos posibles de realizar por el jugador y muchos factores externos que considerar, como los tiempos de construcción, velocidad de movimiento de las unidades, los diferentes mapas, movimientos del enemigo, unidades del ejército enemigo, etc.

Starcraft II, al ser un juego tan masivo, con una gran comunidad y millones de jugadores alrededor del mundo, se realizan periódicamente torneos internacionales masivos, con grandes sumas de dinero como premio, permitiéndole a los jugadores profesionales generar ingresos, un ejemplo de esto fue el torneo de *Intel Extreme Masters de Katowice 2020* llevado a cabo en marzo, donde el premio total fue de 400.000 dólares (Practicodeporte, (2019, octubre 30), aproximadamente 305.680.000 millones de pesos. Ahora bien, en cuanto a la raza *Zerg*, la cual, tiene una estética similar a la de los monstruos de la película Alíen, cuenta con un modo de juego peculiar, ya que, no puede edificar en cualquier sitio sino que solo en la *biomateria Zerg*, la cual es una red de suministro constituida por venas, conductos, órganos y líquidos vitales que dotan de sustento a las construcciones *Zerg* para poder funcionar, en cuanto a sus unidades, estas se caracterizan por el hecho de que no se reparan, sino que se regeneran a partir de larvas disponibles únicamente en el edificio básico *Zerg*. En este contexto es donde las metaheurísticas toman gran relevancia, ya que, sirven como guías para encaminar la búsqueda de soluciones en estos tipos de problemas de complejidad elevada, específicamente en esta memoria se ve el uso del algoritmo *Búsqueda de Vecindades Variables (VNS)*.

VNS es un algoritmo que usa la aleatoriedad y la repetición de procesos para generar vecindades de soluciones, de las cuales se pueden obtener máximos y mínimos locales, los que posteriormente son usados para generar nuevos puntos de partida y a su vez nuevas vecindades, y

repitiendo este proceso millones de veces se logra obtener una buena aproximación a un máximo o mínimo global o en otras palabras un “óptimo”. Para la implementación de este algoritmo se requieren tres herramientas importantes, la primera es el tamaño de la vecindad, la cual viene dada por la cantidad de árboles (hilos) o de soluciones que se generan antes de pasar al siguiente punto, el cual, es el comparador de soluciones, que es el encargado de asignarle un puntaje o valor a cada solución encontrada, y finalmente la elección de la mejor solución local (Talbi, 2009).

1.2 ESTADO DEL ARTE

Para trabajar en la solución a una problemática, es necesario conocer las tecnologías que trabajan en el ámbito en cuestión, en este caso, optimización y algoritmos de búsqueda en videojuegos. En Chile no se ha encontrado ningún trabajo ni documentación que tenga relación con la optimización de algoritmos para videojuegos de esta índole, pero en el ámbito internacional si se encuentran múltiples trabajos al respecto, los cuales se mencionan a continuación y también se ven resumidos en la Tabla 1.1.

1. **Actual IA en Starcraft II:** la inteligencia artificial de *Starcraft II* actualmente sigue siendo muy simple, a pesar de que el juego fue lanzado al mercado en el año 2010, durante todo este tiempo la *IA* ha sufrido pequeñas o nulas modificaciones, y en general cuenta solo con *build-orders* predefinidas de antemano por programadores, haciendo que derrotarla tenga distintos niveles de dificultad, dependiendo de cuál sea la *build-order* usada durante la partida, pero dado que la *IA* toma decisiones muy básicas en términos de estrategias y ataques al oponente es muy fácil de derrotar por un jugador que tenga algún grado mayor de experiencia en el juego.

2. **Continual online evolutionary planning for in-game build order adaptation:** en *Starcraft* existe un programa capaz de crear *build-orders* adaptativos, las cuales son creadas y modificadas a partir de un registro de partidas de jugadores de alto nivel, donde el programa toma decisiones basado en acciones específicas del rival, por ejemplo, en un enfrentamiento de *Zerg* contra *Terrans*, puede ser que uno de los jugadores cree unidades de ataque aéreas (*Mutalisco* o *crucero de batalla* respectivamente), las cuales pueden ser derrotadas solamente por la unidad específica de la raza rival, por lo que la *IA* busca en la base de datos un *build-order* utilizado con anterioridad, que le permita llegar a construir dicha unidad de la forma más rápida posible, para así intentar contrarrestar la estrategia del rival (Justesen, 2017). Pero esta estrategia tiene ciertas debilidades, las cuales son, que al momento de buscar un *build-order* en su base de datos, esta no toma en cuenta el *build-*

order que ya estaba utilizando, por lo que podría en algunos casos, comenzar a construir nuevamente unidades o edificios ya existentes. Por otro lado, este problema podría hacer que la *IA* terminará retrasando aún más sus estrategias si es que el jugador rival cambia su estrategia en medio de la batalla o si usa varios tipos de unidades diferentes produciendo cierta "confusión" en la *IA*. Cabe destacar que esta *IA* solo ha sido utilizada en *Starcraft* y no en *Starcraft II*, a pesar de que su implementación en este último podría ser basada en gran parte en el software anterior.

3. **JamesBot Q-learning:** *JamesBot* utiliza un algoritmo basado en un modelo de decisiones de Markov, el cual consiste en una toma de decisiones basada en estados actuales y futuros, donde cada estado futuro depende solamente del estado actual, y en cada iteración se busca un "óptimo local", es decir, se busca el mejor estado futuro al que se puede llegar desde el estado actual, tomando en consideración variables como la potencia de combate, economía, población máxima que se puede alcanzar, entre otras (Kowalczuk, 2019). El algoritmo que utiliza este *Bot* ha demostrado tener buenos resultados, superando ampliamente a la *IA* por defecto del juego. La desventaja de este algoritmo es que sus *build-orders* son limitadas y muy cambiantes, ya que dependen en gran medida del estado inicial de la partida, y sin tomar en cuenta la estrategia rival, por lo que un jugador experimentado podría contrarrestar fácilmente la estrategia utilizada por el *Bot*.

4. **Winner Predictor:** Existe un algoritmo que intenta predecir el ganador de un enfrentamiento 1 contra 1 de jugadores reales, teniendo en cuenta su historial de partidas, donde se puede comparar las *build-orders* usadas por cada uno, obteniendo una estimación de cuál es más efectiva. Esta forma de predicción mostró ser muy efectiva, consiguiendo un porcentaje de aciertos del 76%, lo que es realmente elevado y apoya fuertemente la hipótesis de que una *build-order* es un factor determinante a la hora de jugar (Volz & Bonde, 2019).

5. **Robust continuous build-order:** Es un agente de mejora para la *IA* actual del juego de *Starcraft*, la cual ayuda a mejorar la toma de decisiones basándose en valores estadísticos creados a partir de la cantidad de unidades de combate, unidades de recolección, recursos y población existentes, generando dos tipos de parámetros para la medición, llamados potencia de fuego y fuerza económica, las cuales se obtienen mediante una función matemática simple. Haciendo uso de estos parámetros y comparándolos con los del enemigo se puede tomar decisiones respecto a enfrentamientos, ayudando directamente a la eficiencia de la *build-order* usada, ya que evita la pérdida innecesaria de unidades de combate en muchos de los casos, evitando que se utilicen recursos valiosos en la recuperación de las unidades perdidas (Churchill & Kelly, 2019).

6. **Human-like build-order selection:** Este algoritmo creado en 2015 intenta imitar la toma

de decisiones de un jugador humano, para esto se selecciona una *build-order* inicial que depende de la raza escogida, donde para cada raza se tiene al menos tres posibles *build-orders* iniciales, de las cuales se elige una de forma aleatoria. Durante la partida se intenta entender cuál es la estrategia del enemigo (que *build-order* está usando) para intentar adaptarse y contrarrestarla, a partir del estado actual. La desventaja de esto es que hay variables que pasa por alto, ya que el algoritmo busca una "solución rápida" para contrarrestar al enemigo, intentando producir la unidad necesaria lo más rápido posible, pasando por alto cosas que se deben tener en cuenta, como la cantidad de unidades recolectoras o la fuerza económica del jugador, por lo que en algunos casos se malgastan recursos que pudieran o no ser usados de mejor manera en la construcción de otros edificios o unidades necesarias para la defensa o para una estrategia ofensiva diferente (Takino & Hoki, 2015).

1.2.1 Resumen estado del arte

Tabla 1.1: Resumen tecnologías actuales

Tecnología	Ventaja	Desventaja	Build-orders
Actual IA en Starcraft II	Diversidad de Build-orders y varios niveles de dificultad	Build-orders predefinidas y nivel de estrategia básica	Predefinida
Continual Online evolutionary planning for in-game build-order adaptation	Multiples build-orders y contraresta a los rivales	Puede "confundirse" dependiendo de las acciones del rival	Obtenidas de una base de datos de jugadores experimentados
JamesBot Q-learning	crea build-orders nuevas a partir de otras	Demasiado disperso en sus soluciones y fácil de contrarestasr	Data base con build-orders predefinidas
Winner Predictor	Buen porcentaje de acierto basándose en historial de jugadores y variables de importancia	No considera habilidades de los jugadores	No crea, solo compara
Robust continuous build-order	Uso de variables de importancia para la toma de decisiones	No puede ser usado por todo publico, solo por la IA del juego	Adapta build-orders preexistentes
Human-like build-order selection	Diversidad de estrategias y build-orders	Poca eficiencia en el uso de recursos y solo prioriza tiempo	Data base con build-orders preexistentes

Fuente: Elaboración Propia (2021)

1.3 DESCRIPCIÓN DEL PROBLEMA

Una vez contextualizada la situación, el problema surge a la hora de encontrar buenos órdenes de construcción en términos de tiempo y poder ofensivo para cada partida y para cada jugador, ya que existen billones de combinaciones y secuencias disponibles a la hora de crear unidades o construcciones, y se podría resumir de la siguiente manera:

- *¿Cómo encontrar buenos órdenes de construcción que permitan conseguir determinada entidad, más rápido que la aplicación de órdenes de construcción aleatorias para jugadores que utilicen la raza Zerg?*

1.4 SOLUCIÓN PROPUESTA

Para abordar el problema de optimización multi-objetivo se propone un algoritmo basado en una metaheurística de solución única, que sea capaz de realizar búsquedas considerando restricciones del juego y árbol de tecnologías de la raza *Zerg*, además de que permita obtener una solución única, la cual, debe cumplir con ser una “buena aproximación a un óptimo” y ser multi-objetivo. La literatura indica que entre las múltiples metaheurísticas, destaca la llamada *Variable Neighborhood Search*, es por esto, que la solución propuesta consiste en un algoritmo *VNS* que como parámetros de entrada reciba una unidad objetivo y la cantidad de estas que desea, además de ser capaz de retornar una secuencia de pasos a seguir *(build-order)*, minimizando el tiempo y maximizando la utilidad.

1.4.1 Características de la solución

Para abordar el problema de optimización multi-objetivo se propone un algoritmo metaheurístico que sea capaz de generar aproximaciones de soluciones a óptimos globales utilizando los parámetros preestablecidos y también los entregados por el usuario.

A continuación se presenta en la Figura 1.1 un bosquejo que intenta explicar qué es lo que busca hacer la solución, este puede variar según cómo se implemente la aplicación en la arquitectura actual del mandante.

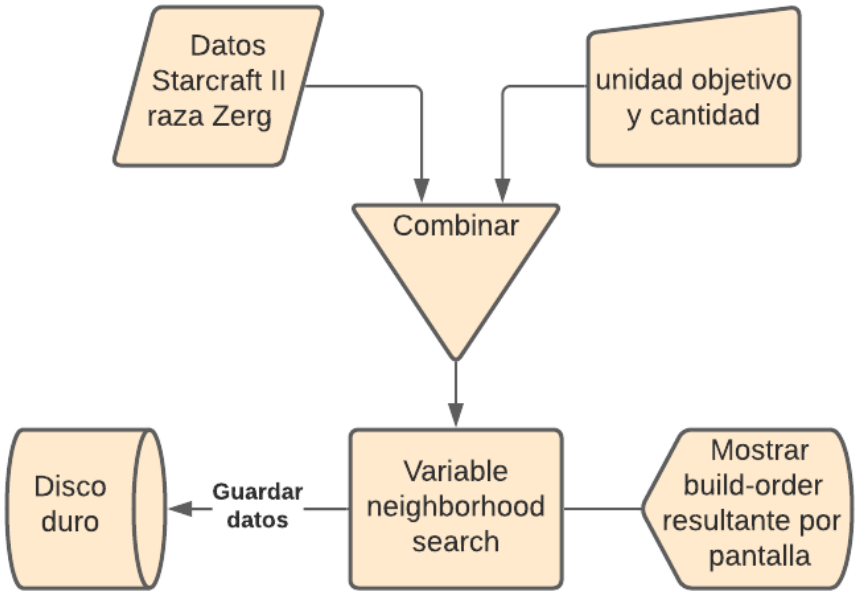

Figura 1.1: Diagrama del proceso de búsqueda de soluciones
Fuente: Elaboración Propia (2021)

1.4.2 Propósito de la solución

Comprobar que las metaheurísticas y algoritmos utilizados para encontrar buenos órdenes de construcción generen un resultado eficiente en términos de tiempo sobre un problema *RCPSP,* como lo es el juego *Starcraft II* (Churchill & Kelly, 2019), y que permitan conseguir determinada tecnología o unidades más rápido que la aplicación de órdenes de construcción aleatorias o de *build-orders* ya conocidas, aumentando con esto las probabilidades de victoria de los jugadores casuales y profesionales que usen la raza *Zerg*. Otro fin de este proyecto es aumentar la cantidad de estrategias, dándole así más versatilidad y dinamismo al juego, lo que posiblemente ayudaría a devolver el interés de los jugadores y fanáticos al modo competitivo de este juego. Por último, el código resultante puede utilizarse para resolver problemas de índoles similares, ayudando a la implementación de inteligencias artificiales o incluso con fines educacionales sobre temas de administración de recursos.

1.5 OBJETIVOS Y ALCANCE DEL PROYECTO

1.5.1 Objetivo general

Proveer una solución utilizando el algoritmo de búsqueda multi-objetivo *VNS*, que sea capaz de encontrar aproximaciones de buenas *build-orders* para la raza *Zerg* en *Starcraft II*, además de probar su eficacia en partidas reales en términos de tiempo requerido para completar dichas *build-orders*.

1.5.2 Objetivos específicos

1. Organizar los distintos conjuntos de datos del árbol de tecnología de la raza *Zerg* en un formato que pueda ser usado en la construcción del algoritmo.
2. Construir funciones para manejar y manipular los tipos de datos creados (operadores de perturbación y de mutación).
3. Implementar el algoritmo de *Variable Neighborhood Search* para la minimización de conjuntos de vecindades.
4. Evaluar el resultado del algoritmo implementado en términos de eficiencia, optimización de tiempo, convergencia de soluciones y pruebas prácticas en el juego.

1.6 METODOLOGÍA Y HERRAMIENTAS UTILIZADAS

1.6.1 Metodología

El presente proyecto a pesar de tener una componente de desarrollo posee además una de investigación. Es por esto que se optó por trabajar con una metodología por etapas, tomando como base el método científico (Popper, 1934). Ya considerado el problema, y revisadas las características generales de la solución propuesta para satisfacer dicho problema, es que se plantean tres etapas para el desarrollo de la solución. La primera etapa consiste en la revisión y organización de los conjuntos de datos. Para la segunda etapa se procede con la implementación

del algoritmo, la que engloba lo que es el manejo y manipulación de vecindades, incluyendo también la implementación misma del algoritmo. En la tercera etapa se evalúan los resultados del algoritmo implementado utilizando distintos criterios y pruebas pensadas específicamente para medir variables de importancia como el tiempo, convergencia del algoritmo y eficacia. Pero antes, el primer paso de un método científico es plantear una hipótesis acorde al problema planteado, para este proyecto en concreto se plantea la siguiente hipótesis.

1.6.2 Hipótesis

"El uso del algoritmo Variable Neighborhood Search sobre el problema RCPSP de optimización de build-orders en starcraft II entrega buenas aproximaciones a óptimos globales"

1.6.3 Etapas

- **Etapa 1:** Organización de los datos: esta etapa está alineada con el objetivo específico 1. En esta etapa se realiza una organización y ordenamiento de los conjuntos de datos derivados del árbol de tecnologías y las dependencias entre los elementos, en los cuales primero se crea un tipo de dato, el cual permite que pueda ser trabajado de forma más fácil que con los datos en bruto, para así poder crear las vecindades del algoritmo seleccionado.
- **Etapa 2:** Implementación: relacionado con los objetivos específicos 2 y 3, esta etapa consiste en la implementación del algoritmo, incluyendo la creación de las vecindades y la realización de distintos tipos de filtros dentro de la vecindad, para obtener elementos que estén más cercanos al óptimo, creando así una nueva vecindad para repetir el proceso. Para esto primero se implementan funciones para la manipulación de los datos, por ejemplo, funciones para cortar, eliminar y juntar conjuntos de datos entre otras.

  Pasando a la implementación del algoritmo metaheurístico, se definen los criterios para la optimización multi-objetivo. Después se estudian e implementan distintos operadores del algoritmo para eliminar elementos que no sirvan y así generar mejores vecindades. Posteriormente, se realiza una parametrización de los resultados obtenidos, es decir, se buscan los mejores parámetros de entrada para el algoritmo *VNS*, lo que se logra utilizando el *software* de *IRace* explicado en la Sección 2.4. Las soluciones se transforman a un formato comprensible para los usuarios, y finalmente se realizan pruebas locales, probando los *build-orders* obtenidos por el algoritmo en partidas reales, por jugadores de diferentes niveles.

- **Etapa 3:** Evaluación de resultados: esta última etapa se relaciona con el objetivo específico 4. En esta última etapa se evaluará el algoritmo implementado en términos de eficiencia y tiempos finales de las *build-orders* obtenidas. Como primera actividad, se evaluará el algoritmo utilizando comparaciones de las métricas con otras *build-orders* ya existentes (como tiempo, fuerza económica o fuerza de ataque). Luego se pasará a una evaluación práctica de los resultados, mediante pruebas en partidas reales llevadas a cabo tanto por jugadores principiantes y profesionales. Al finalizar esta fase de pruebas se obtendrá un porcentaje de éxito (porcentaje de victorias) para ser comparado con la tasa de victorias del jugador antes de usar las *build-orders* resultantes del algoritmo.

1.6.4 Herramientas de desarrollo

Las herramientas que se poseen para el desarrollo del proyecto corresponden a las siguientes, comenzando con la implementación del algoritmo, se usa el lenguaje **Python 3**, el cual es un lenguaje de programación multiparadigma, ya que soporta los paradigmas de orientación a objetos, imperativo y, en menor medida, programación funcional. Es un lenguaje interpretado, usa tipado dinámico y es multiplataforma. Se opta por este lenguaje dado que es de alto nivel, lo cual facilita la curva de aprendizaje y se contaba con conocimiento previo sobre este lenguaje, además cuenta con múltiples herramientas y bibliotecas que facilitan mucho el trabajo, como por ejemplo **google colaboratory**.

Google Colaboratory o ”Colab” para abreviar, es un producto de Google Research. Permite a cualquier usuario escribir y ejecutar código arbitrario de Python en el navegador. Es especialmente adecuado para tareas de aprendizaje automático, análisis de datos y educación. Es por estas razones que se decide usar esta herramienta para la implementación del código, facilitando enormemente las pruebas de desempeño, eficiencia y también la instalación de bibliotecas y complementos.

Por otro lado, también se usa **Google Drive**, el cual, es un servicio de alojamiento de archivos que fue introducido por la empresa estadounidense *Google*. Es también uno de los sitios de alojamiento más conocidos en el mundo, por lo que se decidió usarlo a modo de herramienta de versionamiento del código, ya que, cuenta con una conexión con *google colaboratory*, haciendo más fácil respaldar los avances.

Dentro de la implementación también se usa **Microsoft Excel** para almacenar datos importantes referentes al árbol de tecnologías de la raza *Zerg* y la relación de dependencia que tienen sus unidades y edificios, los cuales son leídos directamente de dos archivos con extensión *“.csv”* para posteriormente crear objetos de la clase *Unidad* dentro del programa.

También es importante el uso de los datos obtenidos directamente del videojuego **Starcraft II**, como los costos de producción de las unidades, cantidades iniciales de unidades o minerales, el árbol de tecnología y ciertas restricciones como el límite de unidades recolectoras que pueden estar extrayendo mineral o gas desde un cierto punto.

Para la documentación se utiliza el editor de documentos en línea de **Overleaf**, el cual, es una *startup* y empresa social que construye herramientas de creación de documentos colaborativas. Su principal producto es un editor de documentos escritos en el lenguaje de etiquetas para composición tipográfica **LaTeX** en tiempo real y colaborativo (Overleaf, 2021). *LaTeX* es un sistema de composición de documentos que incluye funcionalidades diseñadas para la producción de documentación técnica y científica, *LaTeX* es el estándar de facto para la comunicación y publicación de documentos científicos. Está disponible bajo una licencia de software gratuito (LaTeX, 2019).

Para la implementación y diseño de los gráficos utilizados en la etapa de pruebas se utilizó **PowerBI**, que es una colección de servicios de software, aplicaciones y conectores que funcionan conjuntamente para convertir orígenes de datos sin relación entre sí en información coherente, interactiva y atractiva visualmente. Sus datos pueden ser una hoja de cálculo de Excel o una colección de almacenes de datos híbridos locales y basados en la nube. PowerBI permite conectarse con facilidad a los orígenes de datos, visualizar y descubrir qué es importante y compartirlo con cualquiera o con todos los usuarios que desee (Microsoft, 2021).

Por último, para la creación de diagramas y modelos se utiliza **LucidChart** que es una herramienta de diagramación basada en la web, que permite a los usuarios colaborar y trabajar juntos en tiempo real, creando diagramas de prácticamente cualquier índole (LucidChart, 2021).

1.6.5 Ambiente de desarrollo

El proceso de desarrollo será en un computador Asus el cual presenta las siguientes especificaciones:

- Sistema Operativo Windows 10 pro.
- Procesador Intel® Core i5 4670u de 4 nucleos (CPU 3,4GHz x 4)
- Memoria RAM de 16 GB (2 unidades de 8GB)
- Tarjeta Gráfica Nvidia GTX 1060 de 3GB.
- SSD con capacidad de 256GB.

- Disco duro de 2 TB.

Este trabajo de memoria se ha llevado bajo el contexto mundial de la pandemia *COVID-19*, no obstante, no ha tenido mayores implicancias más que un aumento en el tiempo de desarrollo con base a las dificultades para llevar a cabo las reuniones y en algunos casos la obtención de recursos dada la contingencia.

1.7 ALCANCES Y LIMITACIONES

Respecto a los alcances de la solución se encuentran los siguientes:

1. La implementación de un algoritmo metaheurístico *VNS*, para la búsqueda de aproximaciones de *build-orders* óptimos.
2. La implementación debe considerar parámetros como, los recursos existentes, velocidad de extracción de recursos, tiempos y costos de creación de unidades y edificios, el árbol de tecnologías de la raza *Zerg* y la unidad o edificio especificado por el usuario, la cual, corresponde al objetivo del *build-order* resultante, buscando siempre minimizar el tiempo necesario para completarla y aumentar con ello las probabilidades de victoria.

Respecto a las limitaciones de la solución se encuentran los siguientes:

1. **Velocidad de ejecución:** el usuario es una limitación muy grande en el contexto de la medición de eficacia del modelo, ya que el *build-order* está limitado a ser una especie de guía a seguir, pero la velocidad de la ejecución de las órdenes recaen en las manos del jugador mismo, por lo que a pesar de tener un *build-order* muy eficiente puede terminar perdiendo la partida debido a errores externos a la solución propuesta.
2. **El mapa:** en *Starcraft II*, existe una gran cantidad de mapas, y cada cierto tiempo van agregando o quitando algunos. Cada mapa cuenta con una extensión y terreno muy diferente, por lo que hace imposible incorporar todas las variables existentes en estos terrenos al modelo de solución de este problema.
3. **Movimientos de combate:** la decisión de los movimiento de las unidades recae en el jugador, por lo que el mal uso de estas es una variable que no se puede considerar dentro del algoritmo, ya que incluso si se pudiera encontrar un *build-order* óptimo para cierto conjunto de unidades, este podría ser mal ejecutado debido a los ataques del rival o situaciones de combate entre jugadores que podrían retrasar los tiempos de ejecución de las órdenes, por lo que se opta por no considerar estos errores dentro del algoritmo.

1.8 ORGANIZACIÓN DEL DOCUMENTO

- Capítulo 1 ”*Introducción*”: En el primer capítulo se entrega información acerca del contexto del proyecto, indicando las motivaciones y la problemática. Además se otorgan objetivos y alcances sobre la solución a desarrollar y la metodología que permite un trabajo ordenado para finalizar el proyecto.
- Capítulo 2 ”*Marco teórico*”: En este capítulo se muestra el marco teórico, con el propósito de otorgar todos los conocimientos necesarios para poder ayudar a la explicación y entendimiento de las tecnologías a aplicar durante el desarrollo de la solución.
- Capítulo 3 ”*Algoritmo Propuesto y experimentos*”: se expone el funcionamiento del algoritmo *VNS* y todas funcionalidades necesarias para implementarlo, además se detallan el diseño, estructura, aspectos de implementación y se explica cada uno de los prototipos creados. Por ultimo en este capítulo se dan a conocer las pruebas realizadas en el sistema para el cumplimiento de los objetivos propuestos.
- Capítulo 4 ”*Resultados*”: Se muestran los resultados de los experimentos y pruebas para ser analizados y verificar si se cumple o no lo requerido para efectos de esta memoria.
- Capítulo 5 ”*Conclusiones*”: Se presentan las conclusiones del proyecto basándose en los objetivos generales y específicos, las limitaciones y alcances logrados, además de posibles mejoras a futuro y las reflexiones finales referentes a este proyecto.

CAPÍTULO 2. MARCO TEÓRICO

A continuación, se presenta el marco conceptual y bases de este trabajo, lo que incluye un contexto e información relevante para el desarrollo de esta memoria y su correcta comprensión.

2.1 STARCRAFT II Y RCPSP

El problema que se quiere solucionar se puede considerar un problema de programación de proyectos con recursos limitados (*RCPSP* por sus siglas en inglés), este tipo de problemas considera recursos de disponibilidad limitada, actividades de duración conocida y coste de recursos, vinculadas por relaciones de precedencia (Artigues, 2021). Aplicando esto al contexto actual, el objetivo es encontrar un *build-order* de duración mínima asignando un tiempo de inicio a cada actividad de manera que se respeten las relaciones de precedencia dadas por el arbol de tecnologías de la raza *Zerg* y las disponibilidades de recursos existentes. Y es aquí donde las metaheurísticas toman importancia, ya que, según la literatura, son la mejor manera de abordar este tipo de problemas de complejidad elevada (Talbi, 2009).

Más formalmente, el *RCPSP* se puede definir como un problema de optimización combinatoria. Un problema de de este tipo se define por un espacio de solución X, que es discreto o que puede reducirse a un conjunto discreto, y por un subconjunto de soluciones factibles $Y \subseteq X$ asociadas con una función objetivo $f : Y \rightarrow \mathbb{R}$. Un problema de optimización combinatoria tiene como objetivo encontrar una solución factible $y \in Y$ tal que $f(y)$ se minimice o maximice. Un problema de programación de proyectos con recursos limitados es un problema de optimización combinatoria definido por una tupla (V, p, E, R, B, b).

Considerando el problema de *resource-constrained project scheduling problem (RCPSP)* con respecto al criterio de minimización del lapso de tiempo de *build-orders*. El problema explica las limitaciones tecnológicas de la precedencia de las actividades junto con las limitaciones de recursos mediante las restricciones del árbol de tecnologías y la recolección de recursos durante la partida. No se permiten reembolsos de actividades (no se recuperan recursos al deshacer un edificio o unidad), debido a ello es que esta problemática se considera como *NP-hard*. Se propone un algoritmo de *Variable Neighborhood Search* con n vecindarios. Los experimentos numéricos basados en el conjunto de datos de prueba estándar *RCPSP j120* de la biblioteca *PCPLIB* demostraron que el algoritmo propuesto produce mejores resultados que los algoritmos existentes en la literatura para instancias de gran tamaño. Para algunos casos del conjunto de datos *j120*, se mejoraron las soluciones heurísticas más conocidas (Goncharov, 2019).

2.2 STARCRAFT II

Cada juego es un mundo diferente, es por esto que explicar el funcionamiento y algunos términos es de gran ayuda para la comprensión del documento. Comenzando con el funcionamiento y mecánica de una partida promedio en el juego, su estado inicial, las variables y tiempos a considerar dentro del algoritmo implementado.

Figura 2.1: Vista general del juego
Fuente: www.deejayfool.com, 2021

El videojuego *Starcraft II*, consiste básicamente en enfrentamiento entre jugadores, los cuales deben construir bases y ejércitos además de gestionar correctamente los recursos disponibles (Figura 2.2) que existen en el terreno de juego (mapa), con el fin de lograr doblegar a su enemigo y destruir su base o lograr que este se rinda. Esta mecánica o modo de enfrentamiento es sin turnos como en muchos otros juegos de estrategia, pues al ser un RTS (*Real-Time Strategy*) cada jugador juega aprovechando al máximo cada segundo de la partida para potenciar su ejercito o debilitar al del rival.

Figura 2.2: Visor de recursos (mineral, gas vespeno y población)
Fuente: Videojuego Starcraft II, Blizzard Entertainment, 2021

Ahora bien, antes de dar inicio a la partida, cada jugador tiene la opción de seleccionar una de las tres razas disponibles en *Starcraft II*, donde cada una de ellas posee unidades, mecánicas y características únicas, pero todas se encuentran sujetas a las mismas reglas de juego. Dado que este proyecto hace énfasis en la raza *Zerg*, las definiciones, métodos, procesos y ejemplos presentados en los capítulos siguientes se limitan a tratar solamente los temas derivados de esta raza.

Al momento de dar inicio una partida, cada jugador comienza con ciertos elementos básicos (puntuados a continuación), para luego dejar en sus manos el destino de su ejercito y sus tácticas de combate para derrotar al rival.

- 12 Drones ó trabajadores (Figura 2.3)
- 1 Hatchery ó edificio principal (Figura 2.4)
- 3 Larvas (Figura 2.5)
- 0 Mineral
- 0 Gas Vespeno

Figura 2.3: Drone
Fuente: Videojuego Starcraft II, Blizzard Entertainment, 2021

Figura 2.4: Hatchery
Fuente: www.deejayfool.com, 2021

Figura 2.5: Larvas
Fuente: Videojuego Starcraft II, Blizzard Entertainment, 2021

2.2.1 Árbol de Tecnologías

Cada raza posee elementos diferentes entre sí, pero todas las razas se dividen en tres categorías que incluyen la totalidad de los elementos del árbol de tecnologías en cuestión, las que son *unidades, edificios y tecnologías*, donde cada una cuenta con características diferentes, pero también tienen en común que todas ellas poseen requisitos de alguna clase.

- **Unidades:** son todos los elementos del árbol de tecnologías producidos por un edificio o evolucionadas a partir de *larvas*, y su rol es principalmente el enfrentamiento o combate, pero también existen los Drones, que son unidades obreras, encargadas de la recolección de recursos y también la creación de edificios.
- **Edificios:** son estructuras creadas a partir de los Drones o evolucionadas de otros edificios,

que se encuentran en una posición fija dentro del mapa, donde cada edificio creado tiene una funcionalidad o permite la creación de cierta unidad o tecnología, y también existen edificios con la capacidad de brindar apoyo ofensivo (capaces de atacar).

- **Tecnologías:** estos elementos son mejoras que ofrecen distintas ventajas de tiempo, aumento de estadísticas (bonus de ataque o defensa) o desbloqueo de habilidades especificas para algún tipo de unidad.

Refiriéndose a estos tres tipos de elementos como entidades es que se puede expresar un árbol de tecnologías como *las relaciones de dependencia que tienen unas entidades con respecto a otras* , y es gracias a estas conexiones que se obtienen los requisitos y condiciones para desbloquear cada elemento como se aprecia en la Figura 2.6.

Figura 2.6: Árbol de tecnologías Zerg
Fuente: Videojuego Starcraft II, Blizzard Entertainment, 2021

2.2.2 Recursos

Como todo buen sistema neoliberal, cualquiera de las entidades antes descritas tiene un costo asociado que varía según la utilidad de cada una. Existen dos tipos de recursos no renovables distribuidos a lo largo del terreno de juego, que son recolectados por medio de las unidades obreras *(Drones)* y que son restados cada vez que la creación de una entidad lo requiere

y en las cantidades correspondientes. Es importante aclarar que estos recursos no pueden ser menores que cero en ningún caso.

Los minerales son la fuente principal de recursos en *Starcraft II*, pues todas las unidades y edificios lo requieren para su creación. Estos minerales son extraídos directamente de los campos de minerales que se encuentran distribuidos a lo largo del terreno, y deben ser llevados a la *Hatchery*, donde pasan a ser sumados a la cantidad de minerales obtenidos.

Figura 2.7: Minerales
Fuente: Videojuego Starcraft II, Blizzard Entertainment, 2021

Al inicio de la partida, la *Hatchery* del jugador *Zerg* cuenta con ocho campos de minerales cercanos, cuatro grandes y cuatro pequeños, lo que se traduce en un total de 10.800 minerales por *Hatchery* existente, pero a pesar de que estos recursos son limitados y no renovables, el jugador puede ir en busca de nuevos campos de minerales a lo largo del campo de batalla, pero es recomendable siempre crear una *Hatchery* cercana a estos para hacer que la recolección de minerales sea eficiente.

En cuanto a la extracción de estos minerales, un *Drone* tarda 1,99 segundos desde el inicio de la acción hasta que extrae las cinco unidades de mineral que puede cargar, para luego esperar por, aproximadamente, 0,3571 segundos antes de iniciar su viaje de regreso a la *Hatchery* para depositar los minerales y generar un aumento de cinco minerales en el contador de recursos obtenidos. De forma general, dos *Drones* pueden encargarse de extraer minerales de una misma piedra de mineral sin que tengan que esperar a que el otro finalice su extracción, pero, al ser el objetivo principal recolectar la mayor cantidad posible de mineral por segundo, es que se ha calculado por la comunidad del videojuego que el óptimo es de tres obreros por mineral (PiousFlea, 2010), donde el tercer *Drone* obtiene menos minerales por unidad de tiempo, pero se maximiza el ingreso total por cada mineral. Teniendo esto en cuenta, los cálculos realizados por un usuario de la comunidad de *StarCraft II* arrojan un total aproximado de 102 minerales por minuto al contar con tres trabajadores por mineral.

El otro recurso importante es el *Gas Vespeno*, el cual, es un recurso más limitado, ya que solo se cuenta con dos géiseres de gas por *Hatchery*, que además requiere de un edificio llamado *"extractor"* para llevar a cabo la extracción por parte de los *Drones*. También cabe destacar que el gas es un recurso que no es necesario para la creación de todas las entidades del árbol de tecnologías, pero no por esto deja de ser un elemento vital, ya que, por norma general, las

unidades mas poderosas requieren de este recurso.

Figura 2.8: Gas Vespeno
Fuente: Videojuego Starcraft II, Blizzard Entertainment, 2021

La extracción de este gas por medio de un *Drone* tarda aproximadamente 1,415 segundos, para luego dirigirse a la *Hatchery* mas cercana a depositarlo, lo que aumenta en cuatro unidades el contador de *gas vespeno* obtenido. De forma similar a la extracción de minerales, el Extractor de gas alcanza su máxima capacidad con tres *Drones*, es decir, un total de seis por cada *Hatchery*, entregando un total de 114 unidades de gas vespeno por minuto de juego.

Un ultimo recurso, aunque con un comportamiento diferente a los anteriores, es el de límite de población o *control*, el cual, es un valor entero positivo que limita la cantidad de unidades que puede crear el jugador, ya que cada unidad tiene un coste de capacidad asociado y con la creación de estas se alcanza un límite de población que debe ser solucionado con la creación de *Overlords*, que es una unidad especial que aumenta el límite de población en ocho unidades.

2.2.3 Velocidades

Como juego *RTS* que es *Starcraft II*, la velocidad de juego es algo importante, y es común que las acciones del juego sean medidas en segundos, pero en este caso particular, el juego nos brinda la opción de cambiar la velocidad a la que se desarrolla el juego, por este hecho es que se debe explicar como funciona esta mecánica. El juego cuenta con cinco velocidades que pueden ser seleccionadas al crear una partida personalizada o es puesta en la velocidad mas alta en caso de tratarse de partidas clasificatorias.

Los llamados **factores de velocidad**, son valores que aceleran o retrasan las acciones dentro del juego, desde la extracción de minerales, hasta la creación de edificios se ven afectadas por este valor, el cual se detalla en la Tabla 2.1

Tabla 2.1: Factores de velocidad del juego

Velocidad de Juego	Factor velocidad	1 minuto en tiempo de juego
Slower	0.6	100
Slow	0.8	75
Normal	1.0	60
Fast	1.2	50
Faster	1.4	42.86

Fuente: Elaboración Propia (2021)

Un ejemplo practico de la variación de velocidad de juego es la creación de una *Hatchery*, la cual, en una partida normal tarda 71 segundos, pero si fuera una partida con velocidad *Faster*, el tiempo sería de 50,72 seg, pero el juego lo aproxima al número entero inferior más cercano, en este caso, 50 segundos tardaría en ser creado dicho edificio.

Para el correcto entendimiento de este proyecto, se debe tener en cuenta que la velocidad de juego considerada para la implementación del código es la *faster*, ya que es la utilizada en partidas clasificatorias y torneos oficiales.

2.2.4 Build-orders

Para comprender la solución, es bueno dar una definición más concreta y visual a este concepto. Primero, cada vez que un jugador comienza la producción de una unidad, la construcción de un edificio o la investigación de una mejora se considera que el jugador está realizando una acción, la cual, ocurre en un tiempo determinado contado a partir del inicio de la partida en segundos. Al reunir estas acciones en una secuencia, ordenada según sus marcas de tiempo respectivas, se tiene lo que se denomina orden de construcción, el cual, se muestra al final de cada partida. También es habitual en la comunidad de *Starcraft II* que los jugadores compartan estos órdenes de construcción con la finalidad de que sean evaluados por otros jugadores, por lo que no es extraño que se observen distintos formatos a la hora de presentarlos, un ejemplo de estos se muestra a continuación en la Figura 2.9.

Figura 2.9: Ejemplo de build-order zerg
Fuente: Spawning Tool, 2021

2.3 ENFOQUES Y METAHEURÍSTICAS

Con el fin de encontrar una respuesta al problema es que se destacan cuatro distintos enfoques para la solución, con los cuales se puede trabajar en la creación de un modelo que sea capaz encontrar buenas *build-order*, los cuales se explican a continuación.

2.3.1 Enfoques de la Solución

1. **Búsqueda en Espacio de Soluciones:** este enfoque es conocido por ser el más sencillo de pensar, ya que consiste en generar todas las combinaciones posibles de solución que puedan existir, entregando un universo de soluciones bastante amplio, el cual, debe ser restringido mediante una o muchas condiciones, que para este caso específico surgen a partir del árbol de tecnologías de la raza *Zerg*, los recursos disponibles, el costo de generar cada edificio o unidad, el límite de población y restricciones especificas por parte del usuario. Este enfoque permitiría teóricamente encontrar el óptimo en términos de *build-orders*, ya que, se obtendrían todas las soluciones existentes, pero esto conlleva un costo sumamente

elevado en términos computacionales, llegando al punto de que no se podría obtener un resultado sino hasta muchos años después de iniciada la búsqueda, motivo por el cual, se hace inútil su implementación.

2. **Búsqueda en Espacio de Estados:** este enfoque es similar al anterior, pero con la gran diferencia de que entrega un abanico de posibilidades acotado, ya que cada iteración restringe el conjunto seleccionando siguiente mediante la mejor solución del estado anterior. Sin embargo, requiere incluso más recursos que la búsqueda en espacio de soluciones, ya que dependiendo de la implementación podría requerir comparaciones entre las diferentes soluciones obtenidas, las cuales en este tipo de problemas son del orden de varios billones, aumentando aún más el tiempo de búsqueda de estas. Es por eso que este tipo de enfoque tardaría un tiempo increíblemente alto, incluso años, considerando la cantidad abrumadora de combinaciones posibles, lo que también lo hace inviable.

3. **Experiencia de Jugador:** este enfoque es el más subjetivo, ya que depende de la experiencia que tenga el jugador, la cual, se obtiene ya sea, estudiando las mecánicas y *build-orders* existentes o jugando una cantidad significativa de partidas, para poder encontrar buenas *build-orders.* Pero este tipo de enfoque basado en el método de aprendizaje común no asegura encontrar las soluciones más eficientes, además este tipo de enfoque se aleja por completo del ámbito computacional y científico involucrando conocimientos y experiencias individuales de cada jugador, sin considerar el tiempo necesario para encontrar soluciones satisfactorias, lo cual puede tardar incluso años y cientos de horas de juego (Vinyals & Horgan, 2019).

4. **Enfoque metaheurístico:** las denominadas metaheurísticas proveen algoritmos que permiten encontrar aproximaciones de soluciones en problemas de índole de optimización con un universo de soluciones demasiado grande sin tener la necesidad de revisar todos los resultados. La ventaja de este enfoque es que permite encontrar soluciones eficientes en tiempos razonables, ya que son aproximaciones, aunque eso no descarta que sea el óptimo, por esto se vuelve necesario comparar con otras *build-orders* ya conocidas para saber si realmente se obtiene una mejora o no respecto a lo ya conocido (Talbi, 2009).

Dentro de las metaheurísticas existen las llamadas, metaheurísticas de población, las que según la literatura asociada, serian las que brindan una mejor aproximación para este tipo de problemas, algunas de ellas son:

1. **Simulated annealing:** también llamado Algoritmo de Recocido simulado, es un algoritmo de búsqueda metaheurística para problemas de optimización global. El objetivo general de este tipo de algoritmos es encontrar una buena aproximación al valor óptimo de una función en un

espacio de búsqueda grande. A este valor óptimo se denomina ”óptimo global”. El concepto de su nombre viene del proceso de recocido del acero, un proceso que consiste en calentar y luego enfriar lentamente el material para variar sus propiedades físicas. Su funcionamiento consiste en que, durante cada iteración, se evalúa algunos vecinos del estado actual *S* y probabilísticamente decide entre efectuar una transición a un nuevo estado *S'* ó quedarse en el estado actual. Esta evaluación de estados se realiza tomando en consideración los parámetros entregados y un estado de solución deseado. En el contexto de la raza Zerg, los parámetros iniciales serían los recursos existentes, el árbol de tecnologías de la raza, la unidad o edificio que se busca obtener y el criterio de selección para los estados encontrados sería la minimización del tiempo (Talbi, 2009).

2. **Iterated Local Search:** búsqueda local iterada es un término en matemáticas aplicadas e informática que define una modificación de los métodos de búsqueda local o Hill-Climbing para resolver problemas de optimización discretos. Una simple modificación consiste en iterar llamadas a la rutina de búsqueda local, cada vez partiendo de una configuración inicial diferente. El aprendizaje implica que la historia previa, por ejemplo, la memoria sobre los mínimos locales encontrados anteriormente se extrae para producir cada vez mejores puntos de partida para la búsqueda local. El supuesto implícito es el de una distribución agrupada de mínimos locales: cuando se minimiza una función, es más fácil determinar buenos mínimos locales cuando se parte de un mínimo local con un valor bajo que cuando se parte de un punto aleatorio (Talbi, 2009).

3. **Hill-Climbing:** este enfoque es una estrategia basada en la optimización local, la cual requiere muy poco costo computacional, es un enfoque útil si se tiene una función heurística muy buena o cuando los operadores de cambio entre estados tienen cierta independencia entre sí (conmutatividad), lo que implica que la operación de un operador no altera la futura aplicación de otro. La desventaja de usar este enfoque es que dependiendo de cuántos máximos o mínimos locales tenga el problema se vuelve una opción menos eficaz, bajando los porcentajes de efectividad mientras más grande sea el problema, por lo que aplicarlo a un problema como el de la búsqueda de aproximaciones de *build-orders* óptimas sería un error, dada la enorme cantidad de opciones o estados posibles (Talbi, 2009).

2.3.2 Enfoque seleccionado y justificación

Dado que la literatura relacionada propone el uso de estrategias basadas en metaheurísticas de solución única se ha optado por mantener esta línea usando una estrategia

diferente, pero de la misma índole. El enfoque seleccionado es el *"Variable Neighborhood Search" (VNS)*. El cual fue propuesto por Mladenović Hansen en 1997 (Mladenovic & Hansen, 1997). La elección de este método se debe a que proporciona buenas aproximaciones a óptimos globales y aunque haya algoritmos que tengan resultados casi igual de eficientes, dada su naturaleza y tipo de paradigma utilizado en su implementación se ajusta de mejor manera al problema *RCPSP*.

VNS es un método metaheurístico para resolver un conjunto de problemas de optimización combinatoria y optimización global. Explora vecindarios distantes de la solución actual del operador y se mueve desde allí a una nueva solo si se realizó una mejora. El método de búsqueda local se aplica repetidamente para pasar de soluciones en el vecindario a óptimos locales.

VNS realiza sistemáticamente el procedimiento de cambio de vecindad, tanto en descenso a mínimos locales como en huida de los valles que los contienen.

VNS se basa en los siguientes conceptos:

- Una estructura de vecindad, es decir, un conjunto de elementos o estructuras de datos que sirve para representar la solución del problema, pero se debe considerar que esta vecindad puede no contener las soluciones del problema.
- Un mínimo local con respecto a una estructura de vecindario no es necesariamente un mínimo local para otra estructura de vecindario.
- Un mínimo global es un mínimo local con respecto a todas las estructuras de vecindad posibles.
- Para muchos problemas, los mínimos locales con respecto a uno o varios vecindarios son relativamente cercanos entre sí.

A diferencia de muchas otras metaheurísticas, los esquemas básicos de *VNS* y sus extensiones son simples y requieren pocos o nulos parámetros. Por lo tanto, además de proporcionar muy buenas soluciones, a menudo de formas más simples que otros métodos, *VNS* da una idea de las razones de tal desempeño, que, a su vez, puede conducir a implementaciones más eficientes y sofisticadas (Talbi, 2009).

2.4 IRACE

El paquete IRACE comprende algunos componentes que juntos conforman el proceso de optimización. La Figura 2.10 describe cómo las diferentes partes de IRACE interactúan entre sí. IRACE se compone de tres entradas principales (Lopez-Ibañez & Stüzle, 2016):

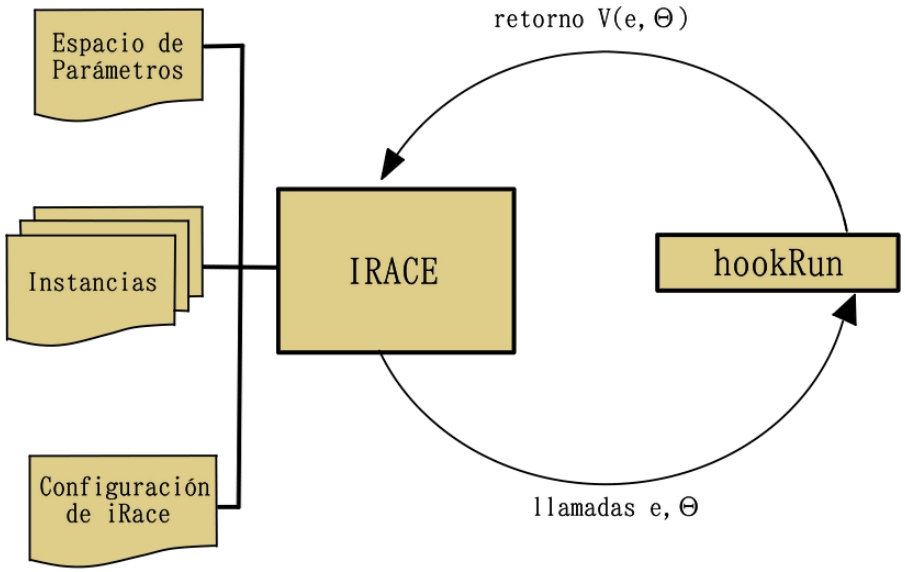

Figura 2.10: Esquema de flujo de información de IRACE
Fuente: Elaboración Propia (2021)

1. Descripción del espacio de parámetros X, es decir, los posibles parámetros que se pueden configurar, sus tipos, valores y constantes.
2. El conjunto de instancias de optimización $e_1, e_2, ...$, que es un conjunto finito utilizado para representar el problema que se está abordando.
3. Configuración de IRACE, definida por una serie de opciones, que forman parte de un archivo de definición de parámetros, presupuesto de ejecución, número de vecindades, entre otros.

Para la descripción del espacio de parámetros, IRACE utiliza un archivo de texto plano llamado por defecto *"parameters.txt"*, el cual, define todos los parámetros posibles que se pueden utilizar para configurar el algoritmo bajo revisión. Cada parámetro se define en una línea de este archivo, que consta básicamente de cinco columnas, el identificador para el parámetro y cómo es referenciado por IRACE al usuario, una clave que identifica el parámetro que será procesado por el módulo de ejecución (*hookRun*), el tipo de parámetro donde "c" corresponde a categóricos, "r" para números reales e "i" para enteros. La cuarta columna describe los posibles valores para el parámetro y finalmente, el quinto puede incluir condiciones para habilitar o no el parámetro utilizando la sintaxis del entorno R (Lopez-Ibañez & Stüzle, 2016).

CAPÍTULO 3. ALGORITMO PROPUESTO Y EXPERIMENTOS

En el presente capítulo se efectúa un análisis del trabajo realizado durante el desarrollo del proyecto utilizando el algoritmo *VNS* para la búsqueda de aproximaciones a buenas *build-orders*. En primera instancia, se definen los requisitos tanto funcionales como no funcionales y cómo han sido abordados a través de los prototipos. El algoritmo multi-objetivo implementado está basado en el algoritmo de búsquedas de vecindades variables o VNS (por sus siglas en inglés: *Variable Neighborhood Search*) para abordar el problema de *RCPSP*. *VNS* es uno de los algoritmos más utilizados en problemas de optimización multi-objetivo e incluye una buena estrategia para aumentar la diversidad de las soluciones (Villalobos-Cid & Inostroza-Ponta, 2018).

3.1 ALGORITMO PROPUESTO

En la Figura 3.1 se muestra una representación visual del funcionamiento del algoritmo, donde *VNS(x)* representa el inicio del algoritmo, $V_1(x)$ es la vecindad inicial, desde donde se evalúan soluciones y se encuentra x', punto que representa un máximo o mínimo local, el cual se usa para crear una nueva vecindad $V_k(x)$, donde repitiendo el proceso anterior, se encuentra un punto x'' que nuevamente es considerado como máximo o mínimo local. Estas Xs encontradas a lo largo del algoritmo son evaluadas y comparadas entre sí, con el fin de verificar cual de ellas está más cerca del objetivo; en el caso del ejemplo cada X' representa un óptimo local mejor que el encontrado en la vecindad anterior, iterando hasta encontrar el objetivo, que en este ejemplo, representa al mínimo Global.

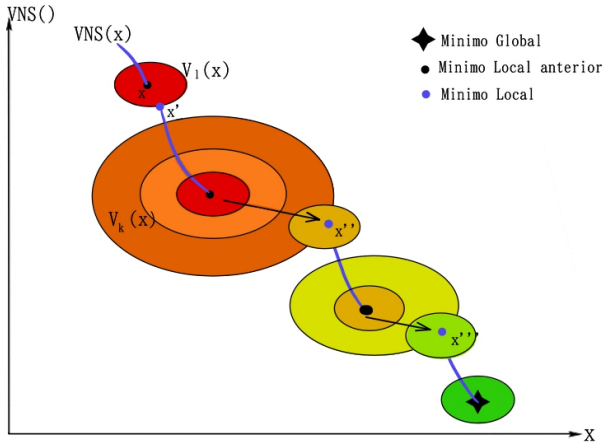

Figura 3.1: Diagrama VNS
Fuente: Elaboración Propia (2021)

El pseudocódigo del algoritmo propuesto se puede ver en el Algoritmo 3.2, donde las entradas son:

- S_C: segundos de corte, corresponden a un parámetro que indica cada cuántos segundos se desea hacer una comparación de los estados de las distintas ramas que corren en cierta iteración.
- N_k: árboles, se refiere a la cantidad de soluciones que se desea encontrar dentro de la vecindad para posteriormente recorrerlas en búsqueda de un óptimo local para la creación de la nueva vecindad.
- N_r: ramas, hace referencia a la cantidad de hilos que tiene cada uno de los árboles dentro de la vecindad actual.
- n: cantidad de unidades Objetivo que se desea crear.
- T: limite De Tiempo. Este es un parámetro creado por comodidad para realizar pruebas y búsquedas que requieran un tiempo muy elevado para ejecutarse, el cual es utilizado a modo de temporizador en el código, indicando la hora de término que se desea.

Los parámetros con subíndice *"K"* corresponden a vecindades que particularmente en la implementación serán conjuntos de *build-orders* y un *"estado"* asociado a la misma, el cual, es usado para crear puntos de inicio de nuevas vecindades, el criterio de detención para el método iterativo es haber encontrado una o varias *build-orders* que cumplan con tener la unidad objetivo y la cantidad indicada por el usuario. El algoritmo repite un ciclo donde a partir desde un "estado" dado, comienza a crear N_r cantidad de ramas (hilos) creados de forma aleatoria, hasta llegar a la unidad que se desea, respetando las restricciones referentes al árbol de tecnologías y a los costos de creación de cada unidad o edificio, los detalles de este proceso están detallados en las siguientes secciones. Finalmente, tras recorrer la K_{max} cantidad de vecindades se escoge la solución con el mayor *"score"* el cual está dado por la Ecuación 3.1:

$$score = \frac{(t_0 - t_{min})}{(t_{max} - t_{min})} \tag{3.1}$$

Donde t_0 corresponde al tiempo específico de la solución a ser comparada, t_{min} es el tiempo mínimo de todas las soluciones obtenidas dentro de la vecindad y t_{max} es el tiempo más alto de la misma vecindad.

ALGORITMO 3.1: Pseudocódigo VNS Básico

Entradas: S_c, A_k, N_r, n, U_o

Salida: BO (*Build-order* resultante).

Sea A_k, para k= 1, . . ., k_{max}un conjunto de estructuras
de la vecindad a usar en la búsqueda

$BO \leftarrow CrearBuildOrderInicial()$

$k \leftarrow CrearVecindadInicial(U_o, n, N_r, S_c)$

Define *criterio de paro*

mientras no *condición de parada*

Sea, $k \leftarrow 1$

mientras no $k = k_{max}$

genera un punto aleatorio (BO') en k-ésima vecindad de BO $(BO' \epsilon N_k(BO))$

aplica algún método de ***búsqueda local*** usando a BO' como punto inicial

(sea BO'' la solución encontrada)

si la solución obtenida BO''es mejor que la anterior (BO),

set $BO \leftarrow BO''$y $k \leftarrow 1$

sino $k \leftarrow k + 1$

fin

retornar (BO)

Figura 3.2: Pseudocódigo VNS Básico
Fuente: Elaboración Propia (2021)

3.1.1 Representación de Datos

El algoritmo implementado recibe dos archivos con extensión *".csv"* que contienen una representación de los datos de las unidades de la raza Zerg, los cuales son usados para crear Objetos de la clase "unidad" (Figura 3.3), la cual se encuentra detallada en la subsección 3.2.2.

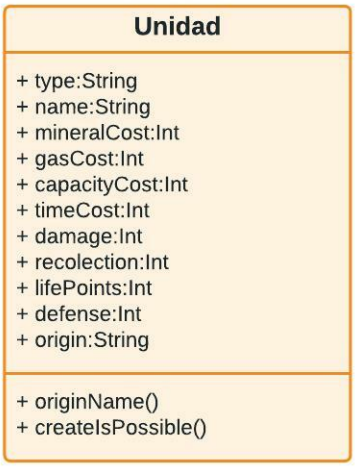

Figura 3.3: Clase Unidad
Fuente: Elaboración Propia (2021)

El primero de los archivos lleva el nombre de *"unidades.csv"* (Figura 3.4) y contiene todos los datos necesarios para llamar al constructor de la clase *Unidad*. Donde el *Type* corresponde al tipo de elemento, *'U'* en caso de ser una unidad y *'E'* en caso de ser un edificio, el nombre de la entidad, el costo de creación, sus valores de ataque, defensa y vida, además de la unidad de la cual evolucionan (*origen*).

Type	name	costoMineral	CostoGas	costoCapacidad	costoTiempo	damage	recoleccion	vida	defensa	origen
E	Hatchery	300	0	6	71	0	0	1500	1	ninguno
U	Overlord	100	0	8	18	0	0	200	0	Larva
U	Drone	50	0	1	12	5	0	40	0	Larva
E	Extractor	25	0	0	21	0	0	500	1	ninguno
E	Spawning Pool	200	0	0	46	0	0	1000	1	ninguno
E	Evolution Chamber	75	0	0	25	0	0	750	1	ninguno
E	Roach Warren	150	0	0	39	0	0	850	1	ninguno
U	Roach	75	25	2	19	18	0	145	2	Larva
U	Ravager	25	75	1	9	18	0	120	2	Roach
E	Baneling Nest	100	50	0	43	0	0	850	1	ninguno
E	Spine Crawler	100	0	0	36	25	0	300	2	ninguno
E	Spore Crawler	75	0	0	21	0	0	400	1	ninguno
U	Baneling	20	25	0.5	14	18	0	35	1	Zergling
E	Lair	150	100	0	57	0	0	2000	1	ninguno
U	Overseer	50	50	0	12	0	0	200	2	Overlord
E	Hydralisk Den	100	100	0	29	0	0	850	1	ninguno

Figura 3.4: Archivo unidades.csv
Fuente: Elaboración Propia (2021)

El segundo archivo llamado *tecnologías.csv* (Figura 3.5) contiene la información necesaria como el nombre de la tecnología, el costo de investigación, el nombre de la entidad donde se puede crear y el *costeTiempo* que corresponde al valor de los segundos que tarda en estar lista la mejora en una partida. Este valor debe ser multiplicado por el factor de velocidad del

juego visto en la subsección 2.2.3 para obtener el valor en segundos correspondiente al tipo de partida que se está evaluando.

tecnologia	costoMinera	CostoGas	costeTiempo	origen
Pneumatized Carapace	100	100	43	Hatchery
Burrow	100	100	71	Hatchery
Metabolic Boost	100	100	79	Spawning Pool
Adrenal Glands	200	200	93	Spawning Pool
Melee Attacks Level 1	100	100	114	Evolution Chamber
Melee Attacks Level 2	150	150	136	Evolution Chamber
Melee Attacks Level 3	200	200	157	Evolution Chamber
Missile Attacks Level 1	100	100	114	Evolution Chamber
Missile Attacks Level 2	150	150	136	Evolution Chamber
Missile Attacks Level 3	200	200	157	Evolution Chamber
Ground Carapace Level 1	150	150	114	Evolution Chamber
Ground Carapace Level 2	225	225	136	Evolution Chamber
Ground Carapace Level 3	300	300	157	Evolution Chamber
Glial Reconstitution	100	100	79	Roarch Warren
Tunneling Claws	100	100	79	Roarch Warren

Figura 3.5: Archivo tecnologías.csv
Fuente: Elaboración Propia (2021)

Una solución corresponde a una representación de estas entidades en el espacio euclidiano que satisfacen las restricciones de precedencia, ordenados linealmente de forma creciente por su tiempo de creación asociado, este conjunto de elementos es en la practica, una *build-order*. Para poder optimizarla haciendo uso del algoritmo *VNS*, se requiere de cuatro elementos fundamentales, una ”vecindad inicial”, un ”operador de mutación” y un ”operador de perturbación” y un ”selector de soluciones”.

3.1.2 Vecindad Inicial

La vecindad inicial del algoritmo VNS viene dada por el estado inicial de una partida claficatoria de Starcraft II, la cual, incluye:

- **1 Hatchery (Base principal)**
- **3 Larvas**
- **12 Drones**
- **0 Mineral**

- **0 Gas Vespeno**
- **1 Build-Order vacía**
- **1 Cola de construcción vacía**

A partir de este punto comienza a correr el tiempo y se genera *n* cantidad de árboles, que a su vez contienen *m* cantidad de ramas, hasta encontrar una solución que coincida con lo buscado por el usuario. Posteriormente, al encontrar la mejor solución dentro de la vecindad, se cambia de vecindad, generando una nueva a partir de la mejor solución encontrada en la vecindad anterior, pero truncada al estado que esta tenía a los *x* segundos, donde *x* es un número cualquiera mayor a cero entregado por el usuario como parámetro.

3.1.3 Operador de Perturbación

Los operadores de perturbación dentro del algoritmo *VNS* son los encargados de modificar las vecindades en cada iteración, la forma de hacerlo varía entre cada implementación, en este caso particular existe $O_P(vecindad1, vecindad2)$ que es el encargado de cambiar entre vecindades.

El método O_P es el operador encargado de modificar el estado inicial de la búsqueda de óptimos locales dentro de la vecindad, su funcionamiento consiste en tomar el estado de la mejor solución encontrada dentro de la vecindad actual (V_a) y truncarlo a los x segundos (x es un parámetro de entrada dado por el usuario), para con esto generar una nueva vecindad aleatoria, pero similar a V_a hasta los *x* primeros segundos.

OPERADOR DE PERTURBACIÓN $O_p(V_a, V_n)$

Entrada: V_a, V_n

Salida: *nuevoEstadoInicial*

$V_a = Mejor\ solución\ de\ la\ vecindad\ actual$ // $V_a\ \epsilon\ vecindad1$

$V_n = Mejor\ solución\ de\ la\ nueva\ vecindad$ // $V_n\ \epsilon\ vecindad2$

si $score(V_n) > score(V_a)$

$V_a \leftarrow V_n$ // $V_a\ toma\ el\ valor\ de\ V_n$

$nuevoEstadoInicial = truncar(V_a\ ,\ x)$ //$x = segundosenlosquesedeseatruncar$

Retornar $nuevoEstadoInicial$

Figura 3.6: Operador de Perturbación
Fuente: Elaboración Propia (2021)

3.1.4 Operador de Mutación

Los operadores de Mutación dentro del algoritmo *VNS* son los encargados de crear las ramas de los árboles dentro de la vecindad en cada iteración, la forma de hacerlo varía entre cada implementación, en este caso particular existe, $O_m(vecindad)$ que es el operador encargado de modificar cada árbol de la vecindad creando ramas aleatoriamente buscando alcanzar la unidad objetivo.

- **Vecindad:** Corresponde a una vecindad entregada como parámetro, es decir, un conjunto de variables que en su conjunto componen el *estado* de una vecindad.
- V**:** Entidad de tipo Vecindad.
- V_a**:** Corresponde a la vecindad que actualmente está siendo trabajada, en este caso $V = V_a$, ya que, V_a es una entidad de tipo Vecindad.
- **Completados**: Corresponde a un arreglo de *arboles*, los cuales todos deben formar parte de la vecindad trabajada.
- A_n**:** Corresponde a un Árbol cualquiera dentro de la vecindad V_a, el cual tras ser ramificado y haber encontrado una solución, es agregado a la lista de *Completados*

- **Cola:** Corresponde a la cola de construcción del árbol específico trabajado, es decir, almacena las unidades que ya fueron ordenadas a implementar, pero que aún no están listas, además esta lista debe ser diferente para cada árbol, ya que las soluciones de cada uno varían respecto a otro

El método O_m tiene un funcionamiento que consiste en tomar el estado inicial dado por el operador de perturbación, y generar *n* cantidad de caminos (ramas) de manera aleatoria, esto lo logra recorriendo un arreglo de unidades y edificios que representan los elementos del árbol de tecnologías, seleccionando uno al azar y, en caso de que pueda ser creado, lo añade a la cola de construcción. Este procedimiento se repite indefinidamente hasta que una o varias ramas alcancen el objetivo, al lograrlo, el árbol, al cual pertenece la rama se añade a otro arreglo para ser comparado con los demás árboles que lograron alcanzar el objetivo.

OPERADOR DE MUTACIÓN $O_m(V_a)$

Entrada: V_a

Salida: V_a

$V_a = Vecindad\ actual$

$completados = [A]$ // $completados\ \epsilon\ V_a$

$n_{max} = total\ de\ arboles$

$A_n = Árboles, para\ n = 1 \dots n_{max}$ // $A_n\ \epsilon\ V_a$ y

$cola = [elementos]$ // $cola\ \epsilon\ A_n$

para cada $A_n\ \epsilon\ V_a\ con\ n <\ n_{max}$:

- recorrer elementos del árbol de tecnologías
- escoger *elemento* aleatorio
- si *elemento* se puede crear:
  - añadir *elemento* a la *cola*
- si *elemento* es igual a *Objetivo*:
  - añadir A_n a *completados*

fin

Retornar V_a

Figura 3.7: Operador de Mutación
Fuente: Elaboración Propia (2021)

3.1.5 Criterios de Optimización

Para los criterios de optimización en un comienzo se consideró solamente *minimizar* el tiempo del *build-order*, pero durante la implementación, se decide incorporar también las variables de *Poder Ofensivo* y de *Poder Económico*, otorgándoles puntajes adicionales a los *build-orders* que cuenten con mayor ejercito o a los que tengan una capacidad de recolección de recursos lo suficientemente alta para ser consideradas como ”mejores” que las otras soluciones

de la vecindad en la que se encuentre. Estos puntajes son explicados en la subsección 3.1.6

3.1.6 Selección de Soluciones

Para la optimización y la convergencia de las soluciones se crearon dos funciones, $f1$ y $f2$, las cuales, buscan minimizar la desviación estándar de las representaciones de los objetos en el espacio euclidiano y por consiguiente reducir los tiempos de la *build-order*, el cual, es el parámetro que se busca minimizar para encaminar la solución hacia el óptimo, en otras palabras, crear selectores de soluciones, uno para las ramas y otro para los árboles de las vecindades.

$f1$ ó $scoreDeRamas()$ se encarga de asignar un puntaje de convergencia a cada rama, midiendo la *"distancia"* que hay desde el estado actual hasta la creación de las unidades objetivo, es decir, que tantos edificios, unidades o restricciones faltan por cumplir, para eso recorre la lista de elementos necesarios y el *build-order* existente hasta el momento y realiza una sumatoria de las unidades necesarias $(U_{necesaria})$, multiplicando el valor en segundos equivalente a 20 minutos (*1200*), el cual es el tiempo desde el que un *build-order* ya no se considera eficiente para esta implementación, menos el tiempo de creación de la unidad necesaria, lo cual da prioridad a los *build-orders* que creen las tecnologías necesarias antes que otras. Además, se le suma N_{uc} que es la cantidad de unidades de combates que contiene el *build-order* hasta ese momento, lo que se ve representado en la Ecuación 3.2.

$$f1 = N_{uc} + \sum_{(U=0)}^{n} U_{necesaria}(1200 - T_U) \tag{3.2}$$

$f2$ ó $scoreDeArboles()$ se encarga de asignar un puntaje a cada uno de los árboles pertenecientes a la vecindad objetivo del algoritmo, basándose en tres parámetros fundamentales, el *Tiempo*, el cual entrega un valor numérico que se normaliza para luego calcular el *score* asociado (3.1). El Poder Ofensivo (PO), correspondiente a la normalización de las sumatorias del daño de ataque de todas las unidades de combate (D_{uc}) en cada *build-order* de la vecindad y, por último, el Poder Económico (PE), que corresponde al puntaje normalizado de la cantidad de *minerales* sumado con la cantidad de *gas* recolectado por segundo en el momento final de cada *build-order* de la vecindad.

$$PO_0 = \sum_{(u=1)}^{n} D_{uc} \tag{3.3}$$

$$PO = \frac{(PO_0 - PO_{min})}{(PO_{max} - PO_{min})} \tag{3.4}$$

$$PE_0 = MineralPorSeg + GasPorSeg \tag{3.5}$$

$$PE = \frac{(PE_0 - PE_{min})}{(PE_{max} - PE_{min})} \tag{3.6}$$

$$f2 = Score + PO + PE \tag{3.7}$$

3.2 IMPLEMENTACIÓN

En esta etapa se procede a describir el desarrollo del proyecto basándose en la metodología utilizada, otorgando detalles del comportamiento del sistema y funcionamiento del mismo en cada etapa del proyecto.

3.2.1 Fases del Desarrollo

El diseño del algoritmo fue dividido en distintas fases, donde cada una de estas, usaba como base la fase anterior, pero sumándole ciertas funcionalidades, nuevos parámetros o mejoras del funcionamiento del algoritmo general como se muestra en la Figura 3.8.

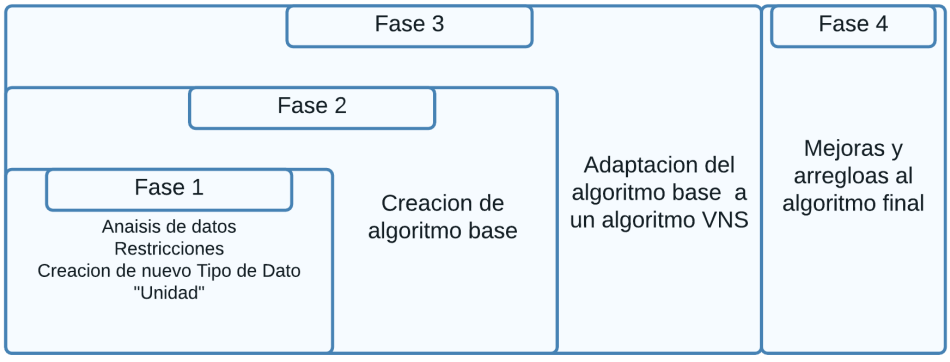

Figura 3.8: Fases de la implementación
Fuente: Elaboración Propia (2021)

En la **Fase 1** se realizó un análisis del problema y se investigó documentación asociada al juego, obteniendo datos importantes, como, por ejemplo, las unidades de la raza *Zerg*, sus edificios, sus tecnologías, restricciones de las unidades, valores o costos de creación o investigación, valores de ataque, defensa, etc. Y es con esta información que se decide crear el tipo de dato llamado *"Unidad"* (subsección 3.2.2) que contiene los elementos del árbol de tecnologías que más tarde se añaden a dos arreglos, uno que contiene solo unidades y el otro que contiene sólo construcciones. Se crea también un arreglo de restricciones para cada elemento siguiendo lo que indica el árbol de tecnologías de la raza *Zerg* (Figura 2.6).

La **Fase 2** se implementan funciones importantes para el funcionamiento del algoritmo, primero que nada se crean funciones para generar un "camino" desde el estado inicial hasta la unidad objetivo. Esta función lleva el nombre de *"unitWay (String:unitName, Int:amount)"*, y recibe como parámetro el nombre de la unidad en formato *String* y la cantidad de dicha unidad que se desea crear, recorre los arreglos de las unidades y restricciones, buscando el *"origen"* de cada una, además de los edificios y tecnologías que son necesarios para poder crear dichos elementos, hasta llegar al nodo inicial, es decir, el estado inicial de la vecindad. Para luego retornar un arreglo con los nombres de todos los elementos necesarios para crear la unidad objetivo, ordenados desde el más básico hasta el más complejo.

Otra función importante es "*itsPossible(String: unitName, Array: BuildOrder, Array: restrictions)*", la cual recibe como entrada el nombre de una unidad o edificio cualquiera que se desee crear, un arreglo que contiene listas con dos elementos dentro, el nombre de la unidad creada y el segundo en que esta se creó (*[Nombre, Segundos]*), y como tercer parámetro recibe la lista de restricciones creadas en la fase 1, para saber si es posible dicha unidad se buscan las restricciones y origen de esta, para seguidamente verificar si esos elementos existen ya creados en el *build-order*. En el caso de cumplirse todas las condiciones, retorna el valor booleano *True*, y en caso de que alguna condición no se cumpla, retorna *False*.

La **Fase 3** es la más compleja a nivel de lógica e implementación, ya que, es la que consumió casi todo el tiempo de programación, en esta fase se implementó una suerte de primer prototipo del algoritmo *VNS*, el cual trabaja la creación de vecindades, operadores de mutación, creación de ramas, restricciones del juego, tiempo de juego y múltiples consideraciones anexas a los datos previamente considerados, como el correcto uso de las *Queens* y distribución de los *Drones* en la extracción de los recursos disponibles.

En esta fase se crea una gran función llamada "*basicAlgorith(Int:threadSeconds, Int:trees , Int:statusTime, String:unitName, Estado:previousStatus)*", la que recibe como entradas la cantidad de segundos que cada rama va a crear, la cantidad de árboles que se desea crear y comparar dentro de cada vecindad, *statusTime* es un valor numérico que señala en qué segundo de cada árbol se desea hacer una copia del estado completo existente hasta ese momento,

con el fin de poder traspasar esa información a la siguiente vecindad a trabajar, también recibe como parámetro el nombre de la unidad objetivo, y por último un estado (*previousStatus*) el cual corresponde al estado desde el que se comienza a construir la nueva vecindad. Esta función será explicada más detalladamente en la subsección 3.2.4.

Por último, en la **Fase 4** se realizan mejoras al algoritmo implementado en la *fase 3* y se implementa realmente el algoritmo VNS, utilizando la función "*basicAlgorith ()*" como punto de inicio. La función final "*VNS (Int:threadSeconds, Int:neighborhood, Int:trees, Int:statusTime, Int:unitAmount, Int:timeLimit)*" es la que lleva la carga de comparar vecindades y elegir entre ellas la mejor solución, tras la normalización y obtención de sus respectivos *scores*, además también agrega las variables importantes a un arreglo para posteriormente escribir un archivo con extensión *".csv"* que es utilizado para gráficar el comportamiento del algoritmo y su convergencia.

3.2.2 Implementación Clase Unidad

Para representar el árbol de tecnologías Zerg se necesita un tipo de dato que sea capaz de almacenar diversas variables de importancia, para esto se optó por crear una clase llamada *Unidad*, la cual guarda las siguientes variables:

- **Type:** variable de tipo *char* que toma el valor 'E' en caso de ser una unidad de tipo edificio o 'U' en caso de ser una unidad común.
- **Name:** variable de tipo *String* que corresponde al nombre de la unidad en cuestión, pero en idioma inglés.
- **MineralCost:** variable de tipo *Int* que corresponde a la cantidad de minerales que cuesta construir o crear cierta unidad.
- **GasCost:** variable de tipo *Int* que corresponde a la cantidad de *gas vespeno* que cuesta construir o crear cierta unidad.
- **CapacityCost:** variable de tipo *Int* que corresponde a la cantidad de población que cierta unidad consume o aporta al total de población permitida.
- **TimeCost:** variable de tipo *Int* que corresponde a la cantidad de segundos que tarda construir o crear cierta unidad.
- **Damage:** variable de tipo *Int* que corresponde al valor de ataque que cierta unidad posee o puede infringir.

- **Recolection:** variable de tipo *Int* que corresponde a la cantidad de minerales o gas que cierta unidad puede recolectar (este valor es 0 para todas las unidades excepto los *Drones*)
- **LifePoints:** variable de tipo *Int* que corresponde a la cantidad de puntos de vida que posee cierta unidad.
- **Defense:** variable de tipo *Int* que corresponde al valor de defensa que cierta unidad posee o al daño que esta puede mitigar.
- **Origin:** variable de tipo *String* que corresponde al nombre de la unidad de la cual se crea o evoluciona la unidad en cuestión, por ejemplo, el origen de un *"Drone"* sería una *"Larva"*.

Además de las variables descritas, esta clase cuenta con dos funciones propias, *originName()*, la cual, entrega el nombre (*String*) de la unidad de la cual proviene, ya sea, *larva*, en caso de las unidades o el nombre de algún edificio especifico, por ejemplo el caso del *"Lair"*, el cual es un edificio que evoluciona a partir de la *"Hatchery"*. La clase también posee el método *createIsPossible()* el cual, es un método que verifica si la unidad se puede o no crear considerando la cantidad de recursos disponibles y el costo de creación de si misma, devolviendo *True* en caso de ser posible y *False* en caso contrario.

3.2.3 Implementación Algoritmo Base

La implementación del algoritmo *variable neighborhood search* se divide en dos partes importantes, primero el trabajo de los árboles y ramas que es considerado el algoritmo base y, por otro lado, el algoritmo *VNS* como tal, el cual, trabaja sobre lo anterior. A Continuación, se explicarán cada uno de los algoritmos y cómo se relacionan entre ellos.

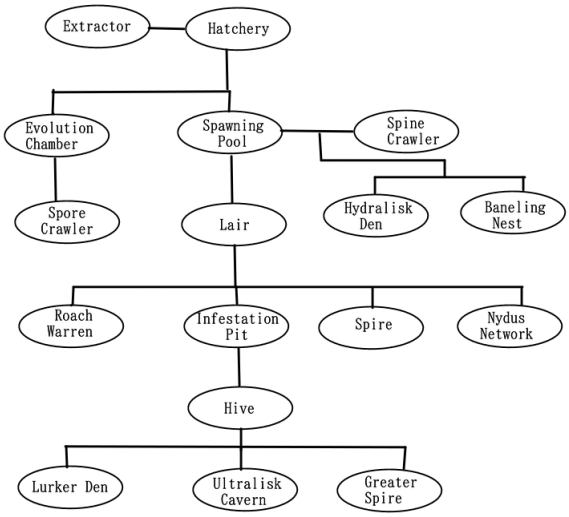

Figura 3.9: Ejemplo de diagrama de árbol
Fuente: Elaboración Propia (2021)

En la Figura 3.9 se muestra de manera gráfica cómo luce el árbol de tecnologías *Zerg* durante una de las iteraciones del algoritmo base. Si por ejemplo a este árbol se le indicara crear una unidad *Lurker*, el árbol resultante podría ser uno de los siguientes.

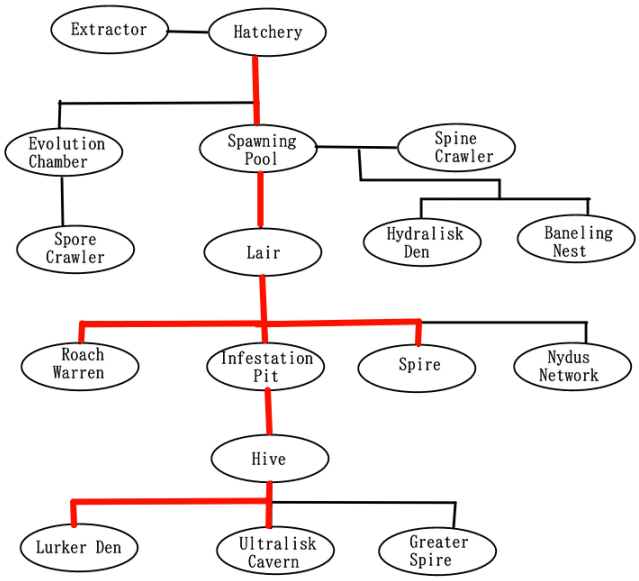

Figura 3.10: Ejemplo de árbol 1
Fuente: Elaboración Propia (2021)

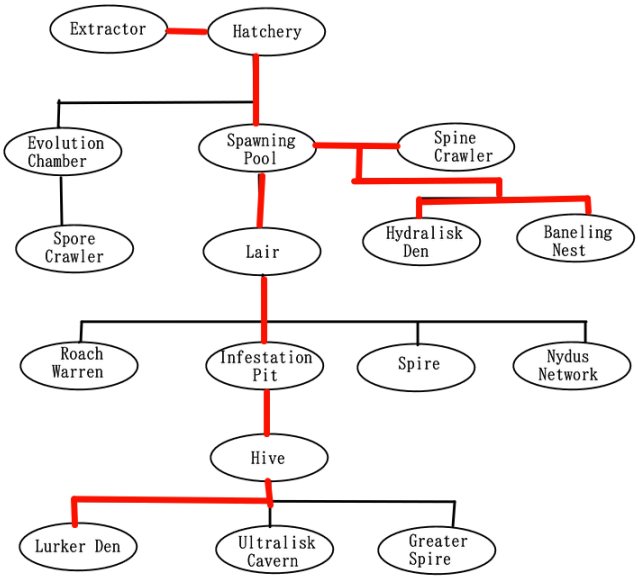

Figura 3.11: Ejemplo de árbol 2
Fuente: Elaboración Propia (2021)

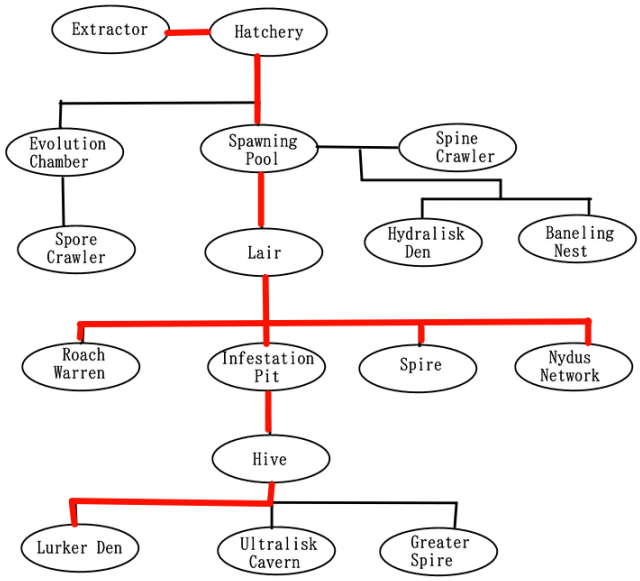

Figura 3.12: Ejemplo de árbol 3
Fuente: Elaboración Propia (2021)

Cada uno de estos Árboles cuenta con *n* ramas, y estas ramas en los casos que logran llegar al objetivo, son retornadas como posibles soluciones, las que fueron representadas

con líneas rojas en las figuras anteriores. Cada solución cuenta con *x* cantidad de unidades, y son comparadas entre ellas dentro de la vecindad, ya que los tres árboles en el ejemplo forman parte de la misma vecindad, como se muestra en la Figura 3.13.

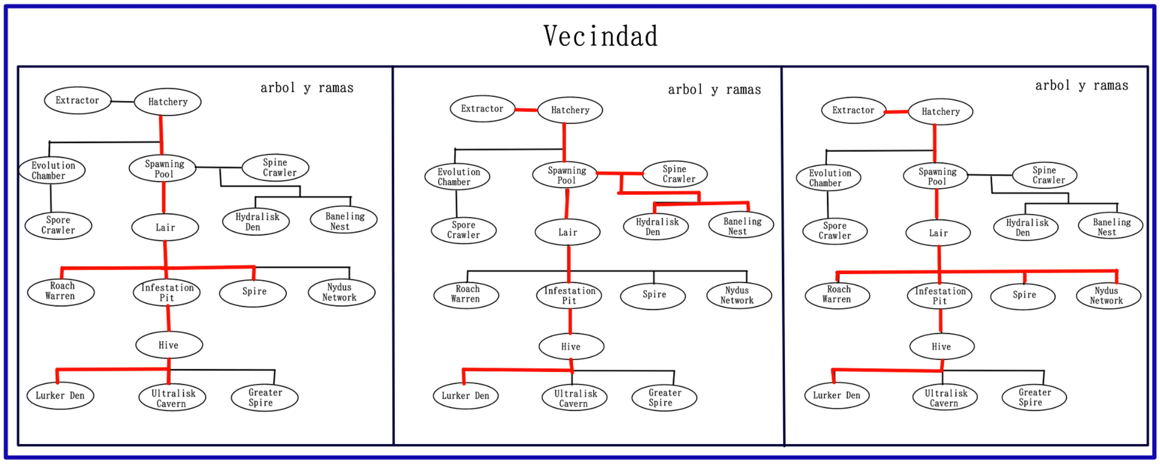

Figura 3.13: Ejemplo vecindad
Fuente: Elaboración Propia (2021)

El algoritmo base o $basicAlgorith()$ como fue nombrado en el código, se encarga de encaminar cada vecindad hacia un óptimo local, creando *n* caminos (ramas o hilos), desde el punto inicial dado hasta la unidad objetivo, el estado inicial es un conjunto de variables, el cual es usado para crear una nueva vecindad a partir de este punto. Los *estados* son constituidos por los siguientes elementos.

- **Build-order:** el estado del *build-order* en el momento de iniciar la vecindad
- **cantidad de unidades:** unidades creadas sin contar los drones (unidades recolectoras)
- **Cola de Construcción:** una lista con los elementos que estaban en cola al momento de iniciar la vecindad, es decir, en la primera vecindad se encuentra vacía, pero en las siguientes esta tendrá los elementos que se hayan estado creando en el instante que da inicio la nueva vecindad.
- **Minerales:** cantidad de mineral disponible en ese momento dado dentro de la vecindad.
- **Gas Vespeno:** cantidad de gas vespeno disponible en ese momento dado dentro de la vecindad.
- **Capacidad:** cantidad de población máxima permitida dentro de una vecindad
- **Segundos:** segundos transcurridos desde el inicio de la partida.

- **Recolectores de mineral:** cantidad de Drones que se encuentran extrayendo mineral en ese instante.
- **Recolectores de Gas:** cantidad de Drones que se encuentran extrayendo gas vespeno en ese instante.
- **Cantidad de Drones:** cantidad de Drones totales existentes en ese momento dentro de la vecindad
- **Estado Guardado:** estado de la vecindad a los x segundos transcurridos. Esta variable es usada para que en caso de que la vecindad actual, entregue la mejor solución dentro de la vecindad actual, se pueda crear una nueva a partir de dicho estado almacenado.
- **Elementos necesarios:** es una lista con todos los elementos que son necesarios para llegar al objetivo del *build-Order*.
- **Lista de Queens:** es una lista con la energía de cada unidad *'Queen'* creada dentro de la vecindad, para saber en qué momentos estas alcanzan las 25 unidades de energía necesarios para ”inyectar” la base y así generar una nueva larva.
- **Cantidad de larvas:** son la cantidad de larvas existentes dentro de la vecindad, para poder verificar si una unidad se puede crear o no en cierto instante.

A partir de este *“Estado”* se comienzan a generar las ramificaciones, que son generadas de forma aleatoria nodo por nodo y segundo a segundo, pero, sujetos a ciertos porcentajes de probabilidad en cada una de las elecciones posibles, los cuales son:

- Probabilidad de elegir crear una unidad recolectora →16.67%
- Probabilidad de elegir crear una unidad cualquiera →16.67%
- Probabilidad de elegir crear un edificio →16.67%
- Probabilidad de elegir crear una unidad priorizando los necesarios →16.67%
- Probabilidad de elegir crear un edificio priorizando los necesarios → 16.67%
- Probabilidad de esperar (no construir) durante un segundo →16.67%

Estas probabilidades fijadas arbitrariamente no son fijas del todo, varias de ellas van ajustándose a distintos momentos o situaciones dentro de la partida, por ejemplo, la probabilidad de elegir una unidad recolectora *(Drone)* disminuye a un 10% en los momentos donde se supera la cantidad de *drones* máximos capaces de recolectar, es decir, 18 recolectores de mineral y 6 recolectores de gas por cada base *(Hatchery)* existente y a su vez se incrementan las

probabilidades de crear un edificio pasando de 16.67% →23.3%” , ya que eso quiere decir que existe un *Drone* disponible para convertirse en una construcción.

Al alcanzar el máximo de población permitida, se añade un *Overlord* u *Overseer* a la lista de elementos necesarios, para de esta forma intentar aumentar las posibilidades de que se cree esta unidad y así elevar el limite de población permitido, pero en caso de crearse un edificio que aumente la población o dicho *Overlord*, se vuelve a quitar de la lista para volver al estado de prioridades anterior.

El 16,67% de probabilidades de ”no construir” se implementó, ya que al comenzar una partida, los recursos son muy bajos como para crear unidades costosas y eso ocasionaba que crearan muchos *Drones*, debido a su bajo costo. Esta nueva opción, sumada a las opciones de priorizar elementos necesarios, genera mejoras sustanciales en la eficiencia de la distribución de recursos y en los tiempos de búsqueda de soluciones, sin dejar de lado la posibilidad de seguir creando *build-orders* 100% aleatorias.

El funcionamiento de la prioridad de elementos toma en cuenta los elementos de la lista *“necesarios”* existente en cada árbol de la vecindad. Cada rama de cada árbol cuenta con una copia de esta lista, llamada *“necesariosTemporal”*, la cual se va modificando, dependiendo de las circunstancias, por ejemplo, añadiendo *Overlords* o *Extractores* en casos donde se alcanza el máximo de población o cuando se crea una nueva *Hatchery* respectivamente. Cabe destacar que la existencia de los *Extractores* dentro de la lista de necesarios es solo en casos donde la unidad objetivo, edificio objetivo o cualquiera de las entidades necesarias para alcanzar el objetivo, necesiten *gas vespeno*; en caso contrario el *Extractor* existe solo en la lista de edificios del árbol de tecnologías, haciendo que crearlo sea solo cuestión de azar.

En casos donde se alcanza la unidad objetivo, pero la cantidad buscada es mayor a uno, se añade a la lista de *“necesariosTemporal”* la cantidad faltante de esta unidad, para así intentar aumentar las probabilidades de crearla, sin perder la aleatoriedad del algoritmo. Por ultimo cabe mencionar, que después de cierto punto, cuando la cantidad de recursos lo permite, existe la probabilidad de un 5,55% de crear unidades dobles. En el juego se pueden crear tantas unidades como *Larvas* disponibles haya, pero por temas de orden visual, manejo de probabilidades, y aleatoriedad del algoritmo se impuso como máximo dos unidades por segundo de juego.

3.2.4 Adaptación a VNS

Para convertir el algoritmo base en un verdadero *algoritmo VNS*, se requieren cuatro funciones indispensables.

1. Evaluar las soluciones obtenidas por cada árbol.

2. Un sistema de decisión basado en puntajes para cada árbol de la vecindad.
3. Generar nuevas vecindades dependiendo de las soluciones obtenidas en la vecindad actual.
4. Crear una condición de parada.

La **evaluación de las soluciones** de cada árbol utiliza el tiempo y las unidades de la *build-order* para con ellas poder obtener los valores que en la implementación son llamados *"poderEconomico"* y *"poderOfensivo"*. El tiempo se obtiene buscando el último elemento de la *build-order*, obteniendo su tiempo de creación en segundos. Para la obtención del *"poderEconomico"* se revisa el estado final de la solución del árbol que se está evaluando, donde se indica la cantidad de *Drones* que se encuentran extrayendo minerales y *gas vespeno* en dicho momento, con estos valores multiplicados por la tasa de recolección correspondiente por segundo se obtiene un puntaje. En cuanto al *"poderOfensivo"*, éste se calcula recorriendo la *build-order* y sumando el valor de ataque de cada unidad existente, Ya teniendo estos tres valores, se procede a normalizar cada uno por separado, usando para ello las funciones de puntajes expuestas en la subsección 3.1.6:

- **Score** = ecuación 3.1
- **PoderOfensivo** = ecuación 3.4
- **PoderEconomico** = ecuación 3.6

Estos puntajes normalizados con valores entre 0 y 1, luego son sumados para obtener el puntaje de final del árbol.

El **sistema de decisión** basado en puntajes, utiliza los *scores* antes mencionados y selecciona el *build-order* que tenga el puntaje más alto, este *build-order* seleccionado es truncado a los *n* segundos indicados por el usuario como parámetro de entrada al algoritmo, realizando una copia del estado de la vecindad dueña de la solución en ese momento, pasando ese estado como parámetro al *basicAlgorith*, el cual comienza a crear una nueva búsqueda de la unidad objetivo generando así una **nueva vecindad**.

Al terminar cada iteración se almacena el *build-order* obtenido en una lista externa que almacena todas las soluciones encontradas para posteriormente compararlas y medir la convergencia de estas, esperando encontrar en ella una aproximación a un óptimo global, lo cual, sigue siendo imposible de comprobar, pero mediante la implementación de esta metaheurística se espera al menos aproximarse a un óptimo.

La **condición de detención** del algoritmo ocurre cuando tras n cantidad de iteraciones del algoritmo completo se alcanza un punto donde los tiempos, los *build-orders* y los

distintos $scores$ convergen a un mismo *estado*, es decir, llegan a la misma solución, catalogándola como la mejor aproximación al objetivo buscado.

3.3 PROTOTIPOS IMPLEMENTADOS

Durante el tiempo de implementación se realizan tres prototipos, cada uno de estos prototipos implementa distintas etapas, y cada uno engloba al prototipo anterior dentro de su funcionamiento, generando una jerarquía muy marcada entre cada uno de estos.

En el **prototipo 1** se utiliza la clase creada para la representación de unidades, las restricciones y los "caminos" o secuencias de nodos que se deben seguir para completar cierta unidad, para crear un primer ejemplo de cómo luciría una solución.

El **prototipo 2** contiene el *basicAlgorith()* (Figura 3.2), y es acompañado de ciertas pruebas que se explican detalladamente en el capítulo 4.

El **prototipo 3** contiene el algoritmo **VNS** implementado usando como base el prototipo 2 desarrollado con anterioridad.

3.4 PROTOTIPO 1: CONCEPTO

Prototipo llevado a cabo durante el mes de mayo, tomando aproximadamente dos semanas de tiempo de implementación, tras haber completado la etapa de análisis y creación de la clase *Unidad*. Este prototipo fue el encargado de buscar caminos dentro del árbol de tecnologías.

Este prototipo fue probado de forma manual, indicando a la función implementada que encontrara cada uno de los elementos del árbol de tecnologías y verificando en el juego si dicho elemento era posible crearse siguiendo los pasos descritos y revisando si alguno de los pasos previos a la unidad objetivo requería de *gas vespeno*, para agregar el *extractor* (edificio necesario para la extracción de gas) a las restricciones, ya que, en el árbol de tecnologías esta construcción, no es una restricción directa de las unidades que usan *gas vespeno* para ser creadas.

3.5 PROTOTIPO 2: FUNCIONAL

El prototipo 2 o prototipo funcional, implementado durante todo el mes junio y parte de julio tomando aproximadamente seis a siete semanas, cuenta con una complejidad mucho más

elevada a nivel de código en comparación con otros prototipos que forman parte del proyecto, ya que cuenta con 26 funciones internas, que se encargan de efectuar los cálculos repetitivos simples, como el cálculo de los minerales extraídos por segundo, cantidad de larvas existentes, energía de las *Queens*, verificar la población límite, restar los recursos al momento de crear una unidad, etc. y también funciones con un nivel de lógica más elevada como el cálculo de los *scores* en las ramas de cada árbol de la vecindad trabajada o también la distribución de los *Drones* que se dedican a la extracción de Mineral y *gas vespeno.*

3.6 PROTOTIPO 3: FINAL

El prototipo 3 o final tomó cerca de dos semanas, llevado a cabo en el mes de julio y comienzos de agosto, cuenta solo con tres funciones, la función de puntaje que utiliza la normalización del *score*, *potencia Económica* y *potencia Ofensiva* para asignar un puntaje a cada solución encontrada y retornar la mejor solución. La función de cambio de vecindad, que se encarga de traspasar el estado de la mejor solución al algoritmo base para que comience desde ahí una nueva vecindad y, por último, la función *VNS* que engloba las dos anteriores y encierra el algoritmo base en un ciclo iterativo que se repite *n* cantidad de veces hasta que llega a un momento donde las soluciones entregadas tienen una similitud o convergencia tan alta (sobre el 97%) que son consideradas soluciones del algoritmo.

3.7 DISEÑO EXPERIMENTAL

El proceso evaluativo del código y sus soluciones se llevó a cabo con pruebas y comparaciones bajo ciertos estándares. Se obtienen estadísticas de distintas variables de importancia, como *potencia ofensiva*, *potencia económica*, *unidades creadas*, *tiempos* de *build-orders* y *tiempos de ejecución*, también se usan los tiempos obtenidos en la creación de distintas unidades y edificios, para realizar comparaciones con otros *build-orders* ya existentes y ver si las creadas a partir del algoritmo superan a las ya conocidas.

También se lleva a cabo pruebas en un escenario real (pruebas en el juego) por jugadores de distintos niveles, desde principiante hasta jugadores experimentados, los cuales prueban los *build-orders* obtenidos, enfrentando a la IA del juego y a otros jugadores para poder obtener el porcentaje de victorias al utilizar los *build-orders* del algoritmo, porcentaje que luego es comparado con las estadísticas del jugador previas al uso de estas, es decir, mientras utilizaban

build-orders preexistentes o aleatorias, todo esto con el fin de calcular y verificar si realmente existe una mejora significativa en los resultados, probando así, si se consigue una mejora en la probabilidad de victoria del jugador que usa los *build-orders* creados por el *algoritmo VNS*.

3.7.1 Parametrización

En una primera etapa y previo a las pruebas, se parametrizó el algoritmo propuesto utilizando el paquete *IRACE* de *R* (Lopez-Ibañez & Stüzle, 2016). *IRACE* es un método de configuración automática de algoritmos de optimización que busca la mejor configuración de parámetros dado un conjunto de instancias del problema, para esto implementa una carrera iterativa que es una extensión del procedimiento *F-race* (Birattari & Varrentrapp, 2002) y aplica pruebas estadísticas al final de cada iteración (Lopez-Ibañez & Stüzle, 2016). *IRACE* toma como entradas una definición del espacio de parámetros del algoritmo a ajustar, un conjunto de instancias del problema, y las opciones de configuración. Luego busca en el espacio de parámetros configuraciones con mejor rendimiento, ejecutando el algoritmo objetivo con distintas configuraciones e instancias, evaluando los resultados y aplicando una prueba estadística para seleccionar las mejores configuraciones, repitiendo este proceso hasta que una condición de término definida es alcanzada.

Para evaluar el algoritmo propuesto, el espacio de los parámetros se definió con los intervalos que se ven en la Tabla 3.1.

Tabla 3.1: Espacio de parámetros

Parámetros	**Sigla**	**Rangos**
Vecindades	v	[20 a 2000]
Árboles	tr	[20 a 500]
Ramas	rs	[20 a 500]
Ratio de Perturbación	pr	[0,0 a 1,0]
Ratio de Mutación	mr	[0,0 a 1,0]
rango de Mutación	r	[0,0 a 1,0]
Probabilidad de Mutación	pmr	[0,0 a 1,0]

Fuente: elaboración propia (2021)

3.7.2 Evaluación

Posterior a la parametrización de las entradas del algoritmo se procede a realizar pruebas con los resultados que este entrega. Se realizaron rutinas de búsquedas para cada entidad del árbol de tecnologías *Zerg*, pero solo se muestran los ejemplos de los mejores resultados obtenidos, la búsqueda de un *Lurker* y la búsqueda de 20 *Roach*.

3.7.2.1 Convergencia del algoritmo

Las pruebas de convergencia miden dos conceptos muy importantes del algoritmo, como lo son, el tiempo del *build-order* y la convergencia de las soluciones.

El tiempo del *build-order* debe cumplir con la condición de estar dentro de un tiempo prudente, concretamente el tiempo máximo es de 20 minutos, es decir 1.200 segundos, lo que corresponde a cinco minutos por sobre el promedio de una partida clasificatoria *1vs1* estándar de *Starcraft II*, cualquier solución encontrada sobre este límite de tiempo es automáticamente desechada, ya que no aportaría suficiente valor al jugador. Por otro lado, las soluciones deben *converger en una cierta solución*, para corroborar que el estado encontrado corresponde a una aproximación a óptimo global, cabe recordar que *"óptimo global"* en la notación de *VNS* hace referencia al óptimo local de todas las vecindades revisadas y no hay forma de comprobar si corresponde realmente a un óptimo global dado el problema *RCPSP* (sección 2.1).

Otra de las pruebas de convergencia y que va de la mano con la anterior es la de **convergencia de tiempos** del *build-order*, es decir, comprobar que los tiempos de las soluciones tiendan a un mismo resultado (segundo), lo que indica que se ha encontrado un óptimo local, a la vez este debe ser un valor razonable y similar o mejor que los *build-orders* ya existentes, ya que lo primordial dentro del objetivo inicial es aumentar las probabilidades de victoria de los jugadores, eso requiere que además de ser un buen *build-order*, sea lo más acotado posible en cuestiones de tiempo, para poder tener más posibilidad de atacar primero a su enemigo y ser capaz de ganarle en combate.

3.7.2.2 Comparación con build-orders oficiales

Con el fin de verificar si los tiempos resultantes de las soluciones del algoritmo *VNS* son acordes a la realidad, verificar si sus ordenes son posibles de seguir por un jugador y, por ultimo, verificar si aportan una ventaja real al momento de facilitar la rápida creación de unidades, es que se realizan comparaciones con *build-orders* preexistentes (que se pueden encontrar en la *web*), ya que estas son usadas día a día por jugadores reales, cuentan con una eficacia comprobada en partidas y son un buen comparador a la hora de evaluar los tiempos de las soluciones del algoritmo.

Estas comparaciones consideran los siguientes puntos:

- **Tiempos totales:** tiempo que tarda en crear la ultima unidad del *build-order*.
- **Diferencias de tiempos:** tiempos del *build-order* obtenido como solución, menos el tiempo de un *build-order* similar preexistente.
- **Diferencias de ejercito:** diferencia en la cantidad de unidades totales de ambos *build-order*.

3.7.2.3 Aplicación en juego

Finalmente, se realizan pruebas en un ambiente competitivo, para las cuales se buscan jugadores de distintos niveles de experiencia, un jugador *unRanked*, el cual, cuente con básica o nula experiencia en el videojuego, un jugador que cuente con experiencia de nivel medio y un jugador experimentado para así marcar su desempeño intentando seguir la *build-order* obtenida como solución del algoritmo, obtener sus comentarios y realizar ajustes en caso de ser necesario.

Estas pruebas se realizan en un ambiente controlado, jugando contra la *IA* en velocidad *Faster*, con un total de tres intentos para cada jugador, se toman los tiempos estimados y reales de cada intento para cada jugador y se evalúan los resultados.

CAPÍTULO 4. RESULTADOS

4.1 PARAMETRIZACIÓN

Tras el proceso de parametrización de IRACE, se obtienen los resultados mostrados en la Figura 4.1

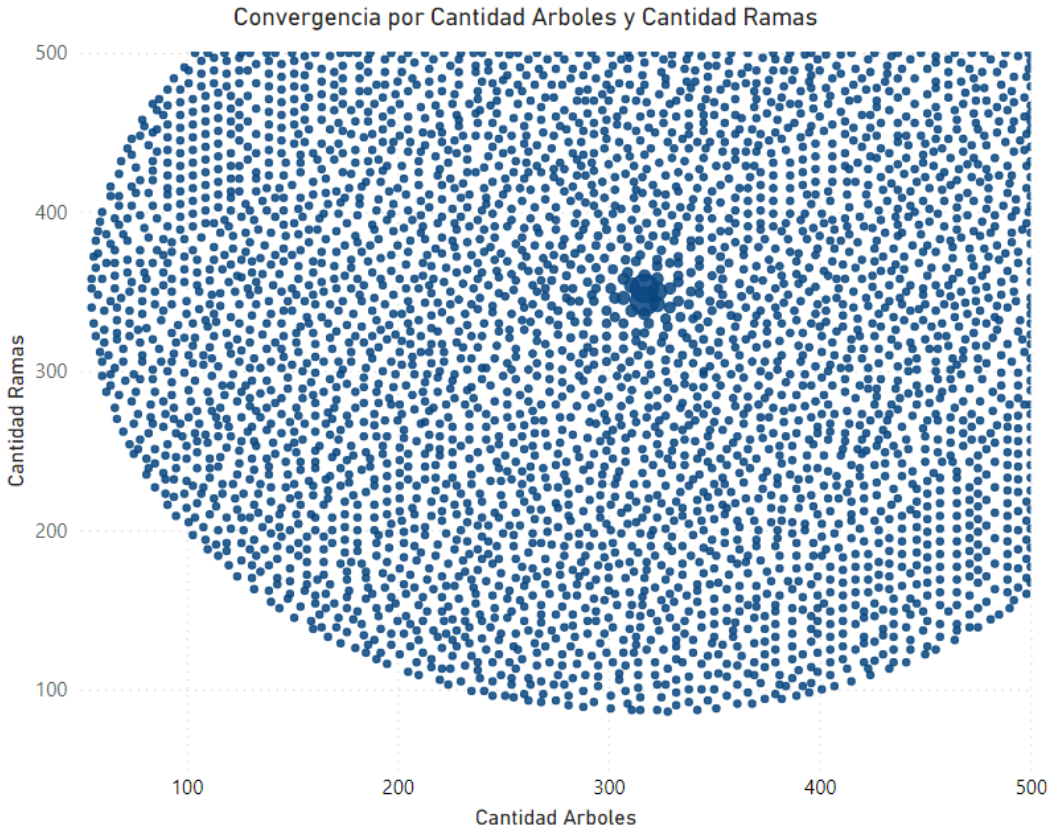

Figura 4.1: Convergencia de Arboles vs Ramas
Fuente: Elaboración Propia (2021)

La parametrización se ejecuta con 10 iteraciones para cada unidad buscada y las búsquedas utilizadas fueron, *1-Lurker, 5-Lurker, 1-Roach y 20-Roachs, 1-Queen*, al terminar la ejecución se extrajeron los resultados para generar la gráfica de la Figura 4.1, se realiza un pequeño filtro, eliminando las soluciones con convergencia muy baja (menor a 0,15) y se muestran solo las combinaciones de parámetros que retornaron buenas aproximaciones a óptimos, con convergencia sobre el 0,75. Se aprecia que los valores entre los 298 y 330 árboles, con 320 a 360 ramas son los que tienen mayor cantidad de aproximaciones y cuentan con una convergencia sobre el 0,97 en la mayoría de los casos.

Tabla 4.1: Resultados parametrización

Parámetros	Rangos	Convergencia	Valor elegido
Vecindades	1147,1321 ó 1553	0.99	No opcional
Arboles por Vecindad	Entre 313 y 321	0.97	317 Arboles
Ramas por Árbol	Entre 345 y 351	0.98	348 Ramas

Fuente: elaboración propia (2021)

Los valores elegidos se seleccionaron eligiendo el punto medio de los intervalos con mejor convergencia para cada uno de los parámetros trabajados, excepto en el caso del número de vecindades, ya que ese es un valor no opcional dentro del algoritmo, es decir, no se puede elegir la cantidad de vecindades que se generan, sino que el algoritmo por sí solo converge a medida que genera *n* número de vecindades.

4.2 TIEMPOS DE EJECUCIÓN

Tabla 4.2: Resultados Tiempo de Ejecución vs Unidad

Unidad Objetivo	Cantidad de unidades	Tiempo de ejecución	Mejor tiempo build-order
Lurker	1	10:13 (min)	03:58 (min)
Lurker	5	10:29 (min)	06:05 (min)
Roach	1	06:54 (min)	03:25 (min)
Roach	20	14:37 (min)	05:58 (min)
Queen	1	01:34 (min)	02:05 (min)

Fuente: Elaboración Propia (2021)

En la Tabla 4.2 se puede ver como existe una relación directa entre el *nivel* de la unidad buscada y el tiempo que tarda el algoritmo en converger a una solución, esto se debe a que mientras mas compleja es la unidad objetivo, mas caminos existen para llegar a ella, es por esto que el caso de la *Queen* tardó solo 01:34 minutos en encontrar la solución, en cambio tarda 10:13 minutos en encontrar un *Lurker*. Otro punto importante es la cantidad de unidades buscada, ya que por ejemplo en la búsqueda de *20 Roachs*, tardó 14:37 minutos, mientras que en buscar solo 1 tarda casi la mitad, esto se debe a que una vez que encuentra el elemento buscado, prioriza la creación de esa unidad, por lo que en ese estado, los árboles de la vecindad tendrían *scores* muy similares, lo que vuelve lenta la convergencia hacia una sola solución.

Los tiempos de ejecución del algoritmo no son relevantes para la correcta evaluación de soluciones, ya que este tiempo transcurre en un ambiente externo al juego y se debe contar con el resultado de la ejecución del código antes de realizar las pruebas que se muestran en la Sección 4.3, esto se debe a que la ejecución del algoritmo tarda mucho más que una partida por lo que su uso en el juego debe ser de forma posterior a su ejecución.

4.3 PRUEBAS DE CONVERGENCIA

Figura 4.2: Prueba de convergencia: diferencia de score vs vecindades
Fuente: Elaboración Propia (2021)

- En el experimento se efectuaron 309 repeticiones antes de encontrar la primera diferencia con valor igual a 0.
- La unidad deseada → *Lurker*
- Cantidad de *Lurker* deseados → 5.
- La cantidad de árboles es de 317.
- La cantidad de ramas de cada árbol es de 348.

- Para cada rama se efectúa una comparación cada 10 segundos en el juego. Velocidad del juego = *Faster (tipo Clasificatoria)*

El gráfico de la Figura 4.2 muestra la diferencia de *scores* entre óptimos locales de las vecindades contiguas, donde se nota una clara y consistente pendiente descendente, la cual, logra llegar muy cerca del valor cero, lo que indica que todas las vecindades consultadas se aproximan hacia un mismo estado, con la misma cantidad de unidades, obreros y la cantidad requerida de unidades objetivo, además de tener una muy buena convergencia promedio (cercana a 0,98) de todas los intentos realizados. Por lo que la prueba de convergencia se considera **Superada**

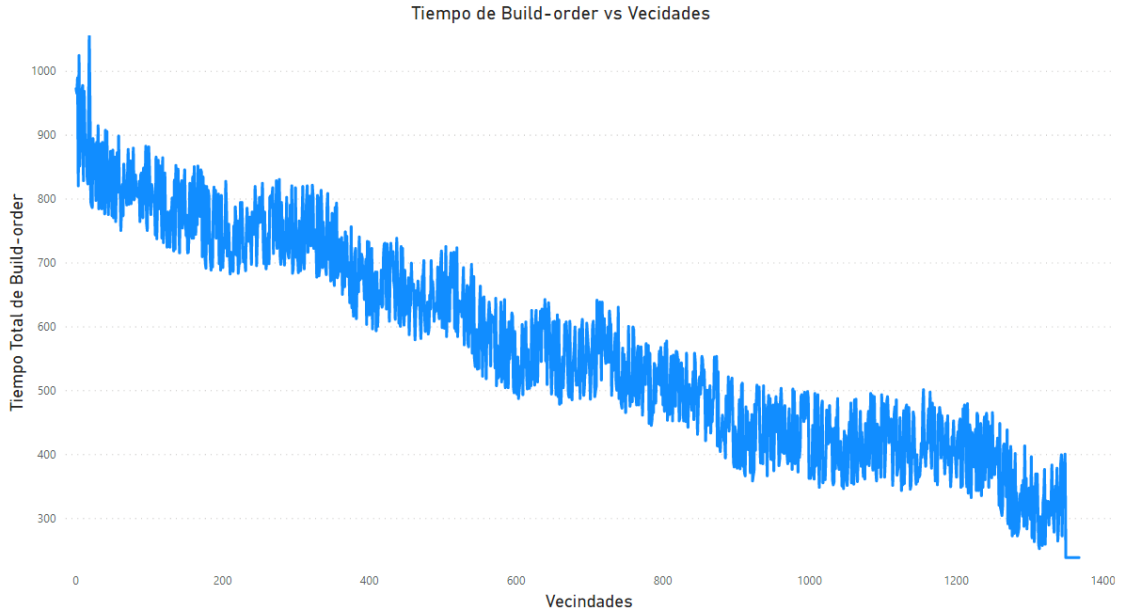

Figura 4.3: Prueba de convergencia: tiempos finales vs vecindades (20-Roach)
Fuente: Elaboración Propia (2021)

En la Figura 4.3 se puede ver como en el mismo caso de estudio (*1-Lurker*) el tiempo de las *build-orders* obtenidas en cada vecindad convergen en un solo punto, pero con un descenso muy agitado y variante, esto se debe a que el ”camino” para llegar a esta unidad es muy variable, y la diversidad de soluciones encontradas era muy alta, pero se puede ver como a medida que compara cientos de vecindades, logra converger hacia un mismo punto, que corresponde a 238 segundos, es decir, 3:58 minutos, que es el tiempo en completar la búsqueda del *lurker*. Por lo que la prueba de convergencia se considera **Superada**

4.4 PRUEBAS RENDIMIENTO

Dado que existen dos algoritmos funcionando dentro de la implementación, se vuelve necesario responder la pregunta ¿existe una mejora real al utilizar ambos algoritmos, o bastaría con solo el *basicAlgorith()*?. Para responder esta pregunta se realizan cuatro pruebas con búsquedas diferentes (*1-Lurker, 5-Lurkers, 1-Roach y 20-Roachs*) y se comparan los tiempos de *build-order* obtenido como solución en cada iteración.

Cada prueba cuenta con once repeticiones y se destaca con color amarillo el mejor tiempo obtenido con el *basicAlgorith* y en color verde el mejor resultado del algoritmo *VNS*. A simple vista se puede aprecias que los tiempos obtenidos con el algoritmo *VNS* son mejores y tienen una dispersión menor, pero eso no basta dar por terminada la prueba, en este caso es necesario aplicar una prueba estadística para comprobar que los resultados son significativamente mejores, en esta ocasión se elije el método de *Wilcoxon*, el cual se realiza sobre la *Prueba 4* (Tabla 4.6), ya que presenta el *build-order* más extenso y es usado para las pruebas con jugadores reales en la Sección 4.5.

Tabla 4.3: Prueba 1 - Búsqueda de 1 Lurker

Busqueda	Iteración	basicAlgorith (min)	VNS (min)	basicAlgorith (seg)	VNS (seg)
1 Lurker	1	5:34	4:10	334	250
1 Lurker	2	6:03	4:32	363	272
1 Lurker	3	5:10	4:05	310	245
1 Lurker	4	4:46	4:08	286	248
1 Lurker	5	5:02	3:58	302	238
1 Lurker	6	5:54	3:59	354	239
1 Lurker	7	4:16	4:25	256	265
1 Lurker	8	5:31	4:01	331	241
1 Lurker	9	6:11	4:14	371	254
1 Lurker	10	5:24	4:11	324	251
1 Lurker	11	5:39	4:23	339	263

Tabla 4.4: Prueba 2 - Búsqueda de 5 Lurkers

Busqueda	Iteración	basicAlgorith (min)	VNS (min)	basicAlgorith (seg)	VNS (seg)
5 Lurker	1	7:34	6:21	454	381
5 Lurker	2	7:13	6:16	433	376
5 Lurker	3	6:29	6:11	389	371
5 Lurker	4	7:02	6:19	422	379
5 Lurker	5	7:54	6:40	474	400
5 Lurker	6	6:14	6:24	374	384
5 Lurker	7	7:16	6:27	436	387
5 Lurker	8	7:21	6:09	441	369
5 Lurker	9	7:56	6:05	476	365
5 Lurker	10	6:44	6:22	404	382
5 Lurker	11	6:33	6:15	393	375

Fuente: Elaboración Propia (2021)

Tabla 4.5: Prueba 3 - Búsqueda de 1 Roach

Busqueda	Iteración	basicAlgorith (min)	VNS (min)	basicAlgorith (seg)	VNS (seg)
1 Roach	1	4:34	3:36	274	216
1 Roach	2	3:36	3:41	216	221
1 Roach	3	4:18	3:29	258	209
1 Roach	4	4:02	3:31	242	211
1 Roach	5	4:40	3:28	280	208
1 Roach	6	3:51	3:37	231	217
1 Roach	7	3:59	3:35	239	215
1 Roach	8	4:17	3:35	257	215
1 Roach	9	5:11	3:42	311	222
1 Roach	10	4:43	3:25	283	205
1 Roach	11	3:45	3:50	225	230

Fuente: Elaboración Propia (2021)

Tabla 4.6: Prueba 4 - Búsqueda de 20 Roachs

Busqueda	Iteración	basicAlgorith (min)	VNS (min)	basicAlgorith (seg)	VNS (seg)
20 Roach	1	6:52	6:12	412	372
20 Roach	2	7:15	5:58	435	358
20 Roach	3	7:04	6:20	424	380
20 Roach	4	7:01	6:03	421	363
20 Roach	5	6:23	6:08	383	368
20 Roach	6	6:45	6:16	405	376
20 Roach	7	7:30	6:14	450	374
20 Roach	8	7:11	6:13	431	373
20 Roach	9	7:56	6:01	476	361
20 Roach	10	6:06	6:05	366	365
20 Roach	11	6:59	6:00	419	360

Fuente: Elaboración Propia (2021)

4.4.1 Test de Wilcoxon

Para la prueba estadística de *Wilcoxon* primero se debe plantear la hipótesis:

- $H_0 : Me_d = 0$
- $H_1 : Me_d \neq 0$

H_0 plantea que las medianas de los resultados de las pruebas son iguales a cero, lo que indica que no existe una mejora significativa a nivel de tiempos entre el algoritmo base y el algoritmo final, mientras que la H_1, nos dice que efectivamente existe una mejora significativa, por lo que el algoritmo *VNS* aporta un valor suficiente a la implementación.

El siguiente paso es calcular el estadístico de prueba W, donde W es el valor menor entre w^+ y w^-:

- w^+ : suma de los rangos con signo positivo
- w^- : suma de los rangos con signo negativo

Los rangos mencionados corresponden a valores numéricos calculados a partir del valor absoluto de la diferencias entre los tiempos del $basicAlgortih$ y los del *VNS* ordenados de forma creciente, como se muestra en la Tabla 4.7

Tabla 4.7: Tiempos con diferencias y rangos

Busqueda	Iteración	basicAlgorith (seg)	VNS (seg)	diferencia	abs(diferencia)	Rangos
20 Roach	10	366	365	1	1	1
20 Roach	5	383	368	15	15	2
20 Roach	6	405	376	29	29	3
20 Roach	1	412	372	40	40	4
20 Roach	3	424	380	44	44	5
20 Roach	4	421	363	58	58	6,5
20 Roach	8	431	373	58	58	6,5
20 Roach	11	419	360	59	59	8
20 Roach	7	450	374	76	76	9
20 Roach	2	435	358	77	77	10
20 Roach	9	476	361	115	115	11

Fuente: Elaboración Propia (2021)

Al sumar los rangos y calcular W se obtienen los siguientes resultados:

- $w^+ = 66$
- $w^- = 0$
- $W = 0$

Tras obtener el valor de W se debe calcular el valor estadístico Z, el que se obtiene a partir de la Ecuación 4.1, donde n es la cantidad de elementos a evaluar ($n = 11$).

$$Z = \frac{W - \frac{n(n+1)}{4}}{\sqrt{\frac{n(n+1)(2n+1)}{12}}} \approx -2,07469 \tag{4.1}$$

Con este valor y utilizando un índice de confianza $\alpha = 0,05$, se calcula el punto critico de la tabla de distribución normal estadística como se ve en la Ecuación 4.2. Con este punto critico, ya se puede comprobar o descartar la hipótesis nula (H_0), pero para una mayor certeza también se calcula el *p-valor* haciendo uso de una función de *Python* y obteniendo como resultado lo que se ve en la Ecuación 4.3.

$$Z_{1-\frac{\alpha}{2}} = 1,96 \tag{4.2}$$

$$p - valor = 0,019007534 \tag{4.3}$$

Finalmente estos valores son menores al valor de significancia estadística, por lo que la decisión es que *existe evidencia estadística suficiente para rechazar la hipótesis nula* H_0, **Comprobando** así, que el algoritmo *VNS* aporta una mejora significativa en términos de tiempo en la implementación y funcionamiento del código.

4.5 PRUEBAS EN JUEGO

Tabla 4.8: Build-Order 20-Roachs

Minuto	Orden		Minuto	Orden		Minuto	Orden
00:01	Drone		03:10	Drone		04:25	Hatchery
00:13	Overlord		03:11	Drone		04:39	Queen
00:19	Drone		03:12	Drone		04:49	Extractor
00:34	Drone		03:19	Drone		04:51	Extractor
00:35	Drone		03:20	Drone (x2)		05:10	Drone (x2)
00:49	Hatchery		03:21	Drone (x2)		05:11	Drone (x2)
00:54	Drone		02:22	Drone (x2)		05:12	Drone (x2)
00:57	Drone		03:28	Overlord		05:13	Overlord (x2)
01:17	Extractor		03:36	Overlord		05:14	Overlord (x2)
01:20	Spawning Pool		03:37	Overlord		05:15	Overlord (x2)
01:35	Drone		03:43	Quenn		05:16	Overlord (x2)
01:43	Overlord		03:51	Overlord		05:33	Drone
02:08	Queen		03:52	Overlord		05:34	Roach (x2)
02:09	Queen		04:11	Roach Warren		05:35	Roach (x2)
02:13	Zergling		04:13	Lair		05:38	Roach (x2)
02:17	Drone		04:14	Drone (x2)		05:39	Roach (x2)
02:23	Drone		04:15	Drone (x2)		05:40	Roach (x2)
02:24	Drone		04:16	Drone (x2)		05:53	Roach (x2)
02:42	Hatchery		04:17	Drone (x2)		05:54	Roach (x2)
02:49	Drone		04:18	Drone (x2)		05:55	Roach (x2)
02:50	Drone (x2)		04:19	Drone (x2)		05:56	Roach (x2)
02:51	Drone		04:20	Drone (x2)		05:57	Roach (x2)
02:57	Overlord		04:21	Drone (x2)		05:58	Roach (x2)

Fuente: elaboración propia (2021)

Pruebas realizadas por tres distintos jugadores de distintos niveles de experiencia del juego, intentando seguir el *build-order* obtenido para crear 20 *Roach* en 05:58 minutos (Tabla 4.8).

Tabla 4.9: Resultados en Juego

Jugador	**Nivel jugador**	**Tiempo VNS**	**intento 1**	**intento 2**	**intento 3**
SteffanUwU	unRanked	05:58	14:43	11:22	09:43
Fardess	unRanked/exPlatino	05:58	08:48	07:29	06:58
LurkasKnight	Diamante	05:58	06:53	06:33	06:07

Fuente: elaboración propia (2021)

Cada jugador tuvo tres intentos para seguir el build-order, y los tiempos obtenidos son mostrados en la Tabla 5.1. El jugador LurkasKnight además de obtener el mejor resultado, señala que “el algoritmo es muy bueno, pero al no considerar el movimiento de las unidades o del jugador, al momento de crear la segunda y la tercera *Hatchery* se genera un retraso, porque el *Drone* no alcanza a llegar a dicha ubicación para el segundo en el que se indica”, también menciona que existe un detalle con la cantidad Larvas porque algunas quedaban ahí durante unos segundos sin que se crearan nuevas unidades y eso no es 100% eficiente.

Con estos comentarios se decidió realizar un cambio al algoritmo retrasando los tiempos de las *Hatcherys* en dos segundos para que el jugador tenga un poco más de espacio para ubicar a la unidad en el lugar necesario, en cuanto a la Larva que queda pendiente se opta por dejar sin cambios, debido a que cualquier cambio podría afectar de forma significativa el porcentaje designado para la aleatoriedad del algoritmo.

4.6 COMPARACIÓN BUILD-ORDERS

Tabla 4.10: Comparaciones de Build-orders

Build-order	Tiempo build VNS	Tiempo build Web	Diferencia	Diferencia ejercito
1-Lurker	03:58	04:30	-32 seg	-16 unidades
1-Roach	03:25	03:05	+20 seg	+4 unidades
20-Roachs	05:58	05:00	+58 seg	+10 unidades

Fuente: elaboración propia (2021)

Tras este análisis se obtienen ciertas características del algoritmo, ya que se aprecia que en la búsqueda de unidades complejas (*Lurker*) el algoritmo *VNS* tarda menos, pero tiene deficiencias en cuanto a ejercito, mientras que, en la búsqueda de unidades mas simples como las *Roachs* este logra tener tiempos mas elevados, pero teniendo mas unidades de combate que una *build-order* típica, y la única forma de medir eficazmente este tipo de diferencias sería mediante pruebas practicas, con enfrentamientos de jugadores de un nivel similar.

CAPÍTULO 5. CONCLUSIONES

5.1 ANÁLISIS DE RESULTADOS

Tabla 5.1: Resumen Pruebas

Prueba	Resultado esperado	Resultado Obtenido	Evaluación
Convergencia	Las soluciones obtenidas del algoritmo dentro de una vecindad convergen hacia un mismo punto, el cual, debe corresponder a la mejor solución dentro de la vecindad.	Las soluciones obtenidas convergen claramente dentro de cada vecindad, aunque la desviación estándar y varianza entre las soluciones es muy alta en algunos casos, pero siempre convergen a un buen punto	Aceptado
Aproximación a óptimos	Las soluciones obtenidas por el algoritmo deben converger hacia un punto, el cual debe corresponder a una aproximación de un buen *build-order* para llegar a una cierta unidad.	Las soluciones obtenidas logran aproximaciones a ciertos puntos, pero dado que es imposible saber con certeza si es o no un óptimo global. Se considera como óptimo global el mejor elemento entre todas las vecindades creadas	Aceptado
Tiempos en el juego	Los tiempos indicados como solución deben apegarse a la realidad, haciendo posible que los jugadores puedan seguir los pasos indicados por la solución.	Las soluciones obtenidas logran converger para cada unidad que se probó, obteniendo tiempos consistentes con la realidad, similar a *build-orders* ya existentes, o incluso más bajos.	Aceptado

Fuente: Elaboración Propia (2021)

5.2 CUMPLIMIENTO DE OBJETIVOS

A continuación, se presentan los cumplimientos y resultados de los objetivos planteados en el Capítulo 1.5 Comenzando con los objetivos específicos y finalizando con el objetivo general.

5.2.1 Objetivos específicos

1. El objetivo de organizar los distintos conjuntos de datos del árbol de tecnología de la raza *Zerg* en un formato que puede ser usado en la construcción del algoritmo, se considera **Cumplido**, ya que, se crea una Clase llamada Unidad, la cual logra almacenar todas las características necesarias de los elementos del árbol de tecnologías *Zerg*
2. El objetivo de construir funciones para manejar y manipular los tipos de datos creados (operadores de perturbación y de mutación), se considera **Cumplido**, ya que se crean satisfactoriamente operadores de perturbación y operadores de mutación para el manejo de las vecindades del algoritmo *VNS*.
3. El objetivo de implementar el algoritmo de *Variable Neighborhood Search* para la minimización de conjuntos de vecindades, se considera **Cumplido**, ya que se logra implementar tanto el algoritmo base basado en metaheurísticas como el algoritmo *VNS* basado en metaheurísticas de solución única.
4. El objetivo de evaluar el resultado del algoritmo implementado en términos de eficiencia, optimización de tiempo, convergencia de soluciones y pruebas prácticas en el juego, se considera como **90% cumplido**, ya que se logran realizar la mayor parte de las pruebas de forma satisfactoria, pero quedan pendientes pruebas a futuro en ambiente profesional, por jugadores de diferentes ligas y con distintos niveles de destreza lo que constituyen la prueba Tasa de victorias.

5.2.2 Objetivo general

El objetivo general de proveer una solución utilizando el algoritmo de búsqueda multi-objetivo *VNS*, que sea capaz de encontrar aproximaciones de buenos *build-orders* para

la raza *Zerg* en *Starcraft II*, además de probar su eficacia en partidas reales en términos de tiempo requerido para completar dichos *build-orders*, se considera como **Cumplido**, ya que se logra encontrar buenas aproximaciones que aunque puedan ser mejoradas, ya cumplen con lo estipulado en esta memoria, y sin duda ayudan a cualquier jugador que quiera buscar nuevas estrategias a la hora jugar *Starcraft II.*

5.2.3 Alcances

Tabla 5.2: Resumen Alcances

Alcances	Esperado	Estado	Acotaciones
Algoritmo VNS	Implementar un algoritmo metaheurístico VNS, para la búsqueda de aproximaciones a buenas build-orders	LOGRADO	Ninguna
Recursos	La implementación debe considerar parámetros como, los recursos existentes y velocidad de extracción de recursos	LOGRADO	La velocidad de extracciones de recursos corresponde a un juego en velocidad normal (no partidas clasificatorias)
Costos	La implementación debe considerar parámetros como, costes de creación, tanto como minerales, gas vespeno y población requerida	LOGRADO	Los tiempos de creación de las unidades corresponden a las de un juego en velocidad normal(no partidas clasificatorias)
Restricciones	La implementación debe considerar restricciones tales como, la dependencia del árbol de tecnologías Zerg, disponibilidad de larvas y población limite	LOGRADO	Ninguna
Visualización	La implementación debe considerar la visualización de la *build-order* por parte del jugador, y debe contar con un formato de fácil lectura.	LOGRADO	Cumple con ser de fácil lectura, pero se espera en un futuro implementar una interfaz especial para su visualización, con temática acorde al videojuego.

Fuente: Elaboración Propia (2021)

5.2.4 Limitaciones

En esta sección se muestran tablas resumen de las limitaciones consideradas dentro de esta memoria, además de ciertas acotaciones especificas.

Tabla 5.3: Resumen de Limitaciones

Limitación	Esperado	Considerado	Acotaciones
Movimientos del usuario	La velocidad de los movimientos y ordenes del usuario quedan fuera del alcance de la solución	No Considerado	Ninguna
Mapas	Debido a la gran variedad de mapas se hace imposible su consideración dentro del código implementado	No Considerado	Ninguna
Movimientos de unidades	Las decisiones que toma el jugador como, ataques, espiar, encerronas, invasiones, etc. quedan totalmente fuera del alcance de la implementación	Considerado	Posterior a las pruebas prácticas en el juego, se toman en consideración las sugerencias hechas por uno de los jugadores, añadiendo ajustes a los tiempos del algoritmo.

Fuente: Elaboración Propia (2021)

5.2.5 Veredicto: Pregunta de investigación

Tras el análisis de resultados, ya es posible darle respuesta a la pregunta de investigación planteada en la sección 1.3, la que plantea la pregunta "*¿Cómo encontrar buenas órdenes de construcción que permitan conseguir determinada entidad, más rápido que la aplicación de órdenes de construcción aleatorias para jugadores que utilicen la raza Zerg?*", pues la respuesta puede no ser 100% concluyente, ya que, el algoritmo *VNS* no siempre entrega soluciones mas *rápidas*, pero si puede entregarnos *build-orders* que se encuentran a un nivel similar a las usadas por profesionales, y donde quizás algunas de estas logre superar a las ya conocidas.

5.2.6 Veredicto: Hipótesis

Para dar por concluido el uso de la metodología, se debe responder la hipótesis planteada en la subsección 1.6.2, la que menciona que *"el uso del Algoritmo Variable neighborhood search sobre el problema RCPSP de optimización de build-orders en starcraft II entrega buenas aproximaciones a óptimos globales"*. Tras analizar las pruebas y probar las soluciones de forma practica la hipótesis es aceptada, ya que, aunque en algunos casos las soluciones tarden un poco mas que las *build-orders* conocidas, estas cuentan con un mayor poder ofensivo, lo que ayudaría teóricamente a mejorar el rendimiento de los jugadores, aumentando con ello sus posibilidades de victoria, que a final de cuentas es el objetivo de todo jugador.

5.3 FUTURO DEL PROYECTO Y EXPERIENCIA DEL DESARROLLO

El código implementado logra cumplir con casi la totalidad de lo esperado, solo quedando pendiente las pruebas practicas con jugadores reales para la medición de la mejora de la tasa de victorias. Por lo que a futuro se desea completar esta batería de pruebas y realizar un análisis estadístico del cambio del desempeño de los jugadores involucrados, acompañado de un estudio más profundo del juego, como mecánicas específicas y tomando en cuenta consejo de jugadores con mayor experiencia, para realizar futuras mejoras al algoritmo e intentar, de esta forma, mejorar los *build-orders* obtenidos mediante el algoritmo *VNS*, en caso de ser posible.

El uso de tecnologías o mejoras dentro del algoritmo también será implementado, ya que algunas de estas aumentarían el valor de *Poder Ofensivo*, pudiendo lograr que algunos óptimos locales cambien drásticamente, y quizás así encontrando nuevos puntos de convergencia de vecindades.

También se ha negociado con el *Streamer* y jugador profesional mexicano *JimRising* la posibilidad de que pruebe algunas de las *build-orders* obtenidos con este algoritmo, para ver su rendimiento en un ámbito profesional y de mas alto nivel que lo probado durante esta experiencia.

Por ultimo, se planea realizar una interfaz gráfica para mostrar las soluciones del código implementado, la cual, tenga una temática similar y acorde a la raza *Zerg*, pudiendo así facilitar el uso por parte de usuarios con menos cercanía a este tipo de tecnologías.

5.4 REFLEXIONES PERSONALES

Durante el transcurso de la carrera como estudiante, uno enfrenta diferentes tipos de dificultades tanto académicas como personales, que nos ponen a pruebas como estudiante y más como persona, ya sea con diferentes tipos de proyectos, trabajos, presentaciones, etc. La elección de tema y la realización del mismo viene a ser la demostración de todas las habilidades aprendidas durante la carrera estudiantil. La implementación del algoritmo *variable neighborhood search* dejó en evidencia algunas de estas habilidades, como por ejemplo, el manejo del paradigma imperativo, el paradigma orientado a objetos, manejos de distintos tipos de datos, etc. además de esto fue un verdadero agrado poder crear una herramienta que en un futuro pueda ayudar a muchos jugadores, que le agrada este tipo de juegos tanto como a mi. Es por esto que espero poder seguir trabajando en este proyecto y agregar todas las mejoras que considere necesarias.

GLOSARIO

1. **Árbol**: se le llama de esta forma a los grafos existentes dentro de las vecindades,los cuales corresponden a una representación del árbol de tecnologías *Zerg*.
2. **Árbol de tecnología**: es un diagrama de árbol evolutivo que simula el avance de la tecnología en los juegos de estrategia histórica de una manera determinista.
3. **Biomateria Zerg**: es una red de suministro constituida por venas, conductos, órganos y líquidos vitales que dotan de sustento a las construcciones *Zerg* para poder funcionar, es más que nada un límite dentro de la superficie del terreno de juego.
4. **Bot**: *software* de *inteligencia artificial* que cuenta con la capacidad de jugar partidas del videojuego en cuestión. Su *inteligencia artificial* puede estar basada en diversos algoritmos y métodos de aprendizaje diferentes.
5. **Build-Order**: es una especie de guía, la cual indica el objeto y momento específico se debe construir una entidad para poder obtener cierta unidad específica dentro del árbol de tecnología de una raza en concreto.
6. **Convergencia**: este concepto se utiliza para referirse a los casos donde múltiples elementos o soluciones se encaminan hacia un mismo punto, suele medirse con valores entre 0 y 1, aunque en ciertas partes de la implementación se encuentra entre 0 y 3.
7. **Convergencia Prematura**: en el contexto de un algoritmo genético la convergencia prematura ocurre cuando hay un mal planteamiento del problema, ocasionando que los individuos de la población solución evolucionen de nula o mala manera, terminando en una solución errónea.
8. **Drone**: unidad obrera de la raza Zerg, evoluciona desde una larva y puede convertirse en múltiples estructuras.
9. **Fuerza Económica**: o Potencia Económica, es un parámetro arbitrario calculado a partir de la suma de los recursos actuales con la cantidad de recursos obtenidos por segundo en cierto instante de tiempo.
10. **Hatchery**: estructura inicial de la raza Zerg, cumple el rol de base principal y tiene la función de generar larvas, investigar ciertas tecnologías, evolucionar en otros edificios más avanzados, etc.
11. **Meta-game**: del inglés ”Most Efficient Tactic Available” (táctica más eficiente disponible), por lo que una definición de meta-game podría ser: elegir la opción más eficiente dentro de

un abanico de posibilidades. Dentro de los juegos competitivos, donde se lucha por saber quién es el mejor, suele existir un número de elecciones total (100 personajes, 1.000 cartas, 5 armas, 10 *build-orders*, etc.), pero no por ello todas están al mismo nivel de importancia.

12. **NP-Hard**: categoría de problemas de decisión para los cuales no existe un algoritmo de complejidad polinomial capaz de resolverlos y a la vez tan o más difícil que el problema más difícil de la categoría *NP (Nondeterministic polynomial time)*.

13. **Parametrización**: en el contexto de este informe, se refiere al proceso de buscar las entradas del algoritmo que entreguen las soluciones mejor evaluadas dentro de un software, que este caso, es IRACE, el cual, es muy usado en la actualidad para problemas de esta índole.

14. **Potencia de combate**: es un parámetro arbitrario calculado a partir de la suma del valor de ataque de las unidades ofensivas, multiplicado por la cantidad de estas, y dividiéndola por la cantidad de población utilizada.

15. **Rama**: se le llama así al “camino” que recorre cada solución perteneciente a un cierto árbol, que a su vez forma parte de una vecindad, también se le puede llamar hilo o hebra.

16. **Software**: programa o conjunto de los componentes lógicos necesarios que permiten realizar distintas tareas específicas.

17. **Vecindad**: es un concepto propio de las metaheurísticas de población que, específicamente en el algoritmo *VNS*, se refiere al conjunto de soluciones que comparten cierto punto de origen común, que es llamado estado inicial, cada vecindad contiene múltiples árboles que a su vez tienen gran cantidad de ramas.

18. **Zerg**: una de las tres razas del videojuego Starcraft II, la cual cuenta con una estética similar a la de la película *Alien*, y con unas mecánicas de juego similares a la de una colmena de insectos

REFERENCIAS BIBLIOGRÁFICAS

Artigues, C. n. (2021). The resource-constrained project. Recuperado el 15 de mayo del 2021, de.

Birattari, S. T.-P. L., M., & Varrentrapp, K. (2002). A racing algorithm for configuring metaheuristics.

Churchill, B. M., D., & Kelly, R. (2019). Robust continuous build-order optimization in starcraft. Conference on Computatonal Intelligence and Games, CIG. London, Reino unido: IEEE .

Goncharov, E. N. (2019). Variable neighborhood search for the resource constrained project scheduling problem. Recuperado de Part of the Communications in Computer and Information Science book series (CCIS, volume 1090).

Huertos, A. A. (2019). www.computerhoy.com. Recuperado el 24 de mayo del 2021, de https://computerhoy.com/listas/gaming/20-mejores-juegos-estrategia-pc-370293.

Justesen, N. (2017). Continual online evolutionary planning for in-game build order adaptation in starcraft. Recuperado de GECCO 2017 - Proceedings of the 2017 Genetic and Evolutionary Computation Conference, (pp. 187-194). Berlin, Germany.

Kowalczuk, C. C. (2019). revisar. Recuperado de 24th International Conference on Methods and Models in Automation and Robotics., (pp. 105-110). Miedzyzdroje, Poland.

LaTeX (2019). A document preparation system. Recuperado de 2019-05-18, de https://www.latex-project.org/.

Lopez-Ibañez, D.-L. J. C. L. P. B. M., M, & Stüzle, T. (2016). The irace packag. Recuperado de Iterated racing for automatic algorithm configuration. Operations Research Perspectives. sciencedirect, 3, 43 -58.

LucidChart (2021). Flowchart symbols and notation. 27 de Septiembre del 2020, de https://www.lucidchart.com/pages/flowchart-symbols-meaning-explained.

Microsoft (2021). Powerbi. Recuperado el 15 agosto del 2021, de https://docs.microsoft.com/es-es/power-bi/fundamentals/power-bi-overview.

Mladenovic, N., & Hansen, P. (1997). Variable neighborhood search. Recuperado el 17 febrero del 2021, de Computers and Operations Research, 24, 1097-1100. https://doi.org/10.1016/S0305-0548(97)00031-2.

Overleaf (2021). overleaf.com. Recuperado el 17 febrero del 2021, de https://www.overleaf.com/about.

PiousFlea (2010). Scientifically measuring mining speed. Recuperado el 15 de septiemnbre del 2021, de https://tl.net/forum/sc2-strategy/140055-scientifically-measuring-mining-speed.

Popper, K. R. (1934). Lógica de investigación científica. Recuperado de Australia: Tecnos.

Practicodeporte ((2019, octubre 30)). www.efe.com. Recuperado el 20 mayo del 2021, de https://www.efe.com/efe/espana/practicodeporte/starcraft-ii-sc2-regresa-al-intel-extreme-masters-de-katowice-2020/50000944-4099063#:~:text=de%20Katowice%202020-,StarCraft%20II%20(SC2)%20regresa%20al%20Intel%20Extreme%20Masters%20de%20Katowice,premio%20tot.

Takino, H., & Hoki, K. (2015). Human-like build-order management in starcraft to win against specific opponent's strategies. Recuperado de Proceedings - 3rd International Conference on Applied Computing and Information Technology and 2nd International Conference on Computational Science and Intelligence, ACIT-CSI 2015 (pp. 97-102). Okayama, Japan.: Article number 7336040.

Talbi, E. (2009). Metaheuristics from design to implementation. Lille, France: Wiley, Jhon Wiley and Son.

Villalobos-Cid, D.-M., M., & Inostroza-Ponta, M. (2018). Performance comparison of multi-objective local search strategies to infer phylogenetic trees. Recuperado de IEEE Congress on Evolutionary Computation, (pp. 1-8)..

Vinyals, O., & Horgan, D. (2019). Grandmaster level in starcraft ii using multi-agent reinforcement learning. Recuperado de Nature, Volume 575, Issue 7782, 350-354.

Volz, P. M., V., & Bonde, M. (2019). Towards embodied starcraft ii winner prediction. Recuperado de Communications in Computer and Information Science volume 1017 (pp. 3-22). Stockholm, Sweden.: 7th Workshop on Computer Games, CGW 2018, held in conjunction with the 27th International Conference on Artificial Intelligence, IJCAI 2018.