UNIVERSIDAD TECNICA FEDERICO SANTA MARIA

Repositorio Digital USM **https://repositorio.usm.cl**

Tesis USM TESIS de Pregrado de acceso ABIERTO

2019-11

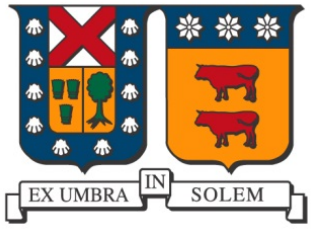
HERRAMIENTAS DE INTELIGENCIA ARTIFICIAL PARA DISEÑO DE BOTS CON ENFOQUE ESTRATÉGICO-TÁCTICO EN JUEGOS RTS

LIENLAF SOTO, ROBERTO GABRIEL

https://hdl.handle.net/11673/55806

Repositorio Digital USM, UNIVERSIDAD TECNICA FEDERICO SANTA MARIA

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



“HERRAMIENTAS DE INTELIGENCIA ARTIFICIAL PARA DISEÑO DE BOTS CON ENFOQUE ESTRATÉGICO-TÁCTICO EN JUEGOS RTS”

ROBERTO GABRIEL LIENLAF SOTO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Elizabeth Montero
Profesor Correferente: María Cristina Riff

Noviembre - 2019

DEDICATORIA

Dedicado a mi amigo Cristian Contreras Ferreira, que en paz descanse. Eras un amante de los juegos RTS y sé que te hubiera encantado ver este trabajo, aportar ideas y probar tus habilidades.

AGRADECIMIENTOS

Agradezco a mi familia por apoyarme todos estos años, incluso después de todas las situaciones complicadas que sucedieron. Agradezco además a todas las personas que me tuvieron paciencia, en especial a mis amigos que siguen a mi lado a pesar del trato que les doy.

RESUMEN

Resumen— Dentro del campo de inteligencia artificial es interesante la investigación correspondiente a los juegos RTS (estrategia en tiempo real) debido a su gran complejidad, tanto temporal como en la toma de decisiones. En este trabajo se plantea un sistema modular respecto a las distintas áreas de investigación de inteligencia artificial dentro de los juegos RTS, y se plantea el uso de algoritmos evolutivos, mapas de influencias, campos potenciales y algoritmos genéticos multi-objetivos para afrontar el problema de un diseño estratégico-táctico para diseñar un bot que sea capaz de competir en alto nivel.

Palabras Clave— Inteligencia artificial, Juegos RTS, Estrategia, Táctica, Algoritmos genéticos.

ABSTRACT

***Abstract*—** In the artificial intelligence field it's interesting the research about RTS games (Real-Time Strategy) because of its high complexity, time restrictions and decision making. In this work, we propose a modular system approach, focused on a variety of research topics about artificial intelligence on RTS games. The bot developed use a mix of evolutionary algorithms, influence maps, potential fields and NSGA-II as algorithms to resolve the problem of a strategic-tactic implementation as a bot design able to compete at a high level.

***Keywords*—** Artificial intelligence, RTS Games, Strategy, Tactics, Genetic Algorithms.

GLOSARIO

NSGA-II: Elitist Non-dominated Sorting Genetic Algorithm. Algoritmo genético con ordenamiento elitista sin dominancia.
RTS: Real-Time Strategy. Estrategia en tiempo real.

ÍNDICE DE CONTENIDOS

RESUMEN IV

ABSTRACT IV

GLOSARIO V

ÍNDICE DE FIGURAS VIII

ÍNDICE DE TABLAS VIII

ÍNDICE DE ALGORITMOS IX

INTRODUCCIÓN 1

CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA 3
1.1 Juegos RTS 4
1.2 Inteligencia Artificial 5
1.3 Características del problema 6

CAPÍTULO 2: MARCO CONCEPTUAL 10
2.1 Elementos a considerar en diseño y desarrollo de estrategias 10
2.1.1 Desarrollo de Inteligencia Artificial 11
2.1.2 Juegos RTS 12
2.1.3 Micro y macro control 12
2.1.4 Visión estratégica y táctica 13
2.2 Características del RTS abordado 14
2.2.1 Fases del juego 15
2.3 Estado del arte 15
2.3.1 Planificación en tiempo real ante adversidad 16
2.3.2 Toma de decisiones ante adversidad 24
2.3.3 Modelado de oponentes 27
2.3.4 Razonamiento espacial y temporal 28
2.3.5 Manejo de recursos 28
2.3.6 Colaboración 29
2.3.7 Pathfinding 29

3 Resumen 30

CAPÍTULO 4: PROPUESTA DE SOLUCIÓN 31
4.1 Modelo general 31
4.1.1 Arquitectura del modelo 32
4.2 Detalles del modelo 35

4.3 Implementación del modelo 39
4.3.1 Flujo del sistema 41
4.3.2 Ambiente de desarrollo 43
4.3.3 Enfoque Estratégico y Táctico 44
4.3.4 Implementación de los algoritmos 44

5 Resumen 60

CAPÍTULO 6: VALIDACIÓN DE LA SOLUCIÓN 61

CAPÍTULO 7: CONCLUSIONES 66

REFERENCIAS BIBLIOGRÁFICAS 69

ÍNDICE DE FIGURAS

1 Bases del jugador naranja y jugador rojo siendo atacadas por el jugador gris. 6

2 Escena de combate del jugador naranja contra el jugador azul. 7

3 Escenario de un combate caótico. 8

4 Enfoques estratégico-táctico versus micro-macro control 14

5 Estructura de un individuo. 19

6 Estructura de una formación. 25

7 Modelo del Sistema. 33

8 Modelo de Clases. 36

9 Clasificación de clases en áreas de investigación. 40

10 Ejemplo de un Orden de Construcción para la raza *Terran* 41

11 Flujo del sistema. 41

12 Ejemplo de grilla de celdas en el mapa del juego. 42

13 Ejemplo de campos potenciales afectando una unidad en particular. 43

14 Implementación enfocada en aspectos estratégico-tácticos. 45

15 Operador de mutación: *Insertar estructura*. 50

16 Operador de mutación: *Mover entidad*. 50

17 Selección de la siguiente generación 56

18 Mapa 'DesertKill'. 62

19 Boxplot. Distribución de individuos (población hijos) en generaciones. 64

20 Boxplot. Distribución de individuos finales (o padres) en generaciones 65

21 Evaluación de los campos potenciales a través de las generaciones. 65

ÍNDICE DE TABLAS

1 Arquitectura de la red neuronal 23

2 Ejemplo de *Goal* u Objetivo para la raza *Zerg* 40

3 Ejemplo de *Goal* u Objetivo para la raza *Terran* 46

4 Evolución de la evaluación de las tácticas 63

5 Uso de recursos del juego 63

ÍNDICE DE ALGORITMOS

1 enemies-threat-vector(M, U, E) 48

2 create-goal(U) 49

3 fast-nondominated-sort(P) 53

4 crowding-distance-assignment(I) 54

5 main-loop 55

6 crowded-comparison operator 55

INTRODUCCIÓN

En el área de los videojuegos el estilo de juegos de tipo *Real-Time Strategy (RTS)* o Estrategia en tiempo real ofrece una gran dificultad a la hora de desarrollar bots dentro de los mismos, pero también un ambienten interesante para la investigación de inteligencia artificial. El ambiente que envuelve las partidas de un juego RTS es sumamente complejo y difícil de modelar tanto para humanos como para máquinas, ya que una partida de juego de tipo RTS se basa en muchos tipos de agentes que se mueven e interactúan constantemente, lo que resulta en un enorme conjunto de dimensiones y estados. Esto sumado a la limitación temporal inherente en un juego en donde cada acción se realiza en tiempo real hace que muchos algoritmos y técnicas de inteligencia artificial no entreguen valor alguno al momento de ser utilizadas, por lo mismo, se ha buscado el desarrollo de bots que sean capaces de entregar una solución óptima y en poco tiempo.

Durante años, se han ido realizando investigaciones para abordar el tema. Se han ido utilizando algoritmos existentes, desarrollado nuevos algoritmos, utilizado análisis y ayuda de jugadores profesionales, desarrollado heurísticas y metaheurísticas, y en general, mezcla de todo lo anterior.

Dentro del desarrollo de estrategias para los bots. Hay distintos acercamientos que suelen guiar la construcción de los mismos, algunos enfocados en utilizar conocimiento de jugadores profesionales, algoritmos de búsqueda, reconocimiento de patrones, aprendizaje de maquina, etc.

En este documento se presentarán una serie de maneras de dividir el problema, lo que permite realizar distintos enfoques dependiendo el área que se aborde. Se describirán distintos bots y distintas técnicas utilizadas junto con el alcance de las mismas. Se presentará una solución, que incluye un modelo de un sistema que aborde varias de estas áreas y una implementación del mismo enfocado en algunas de estas, con el objetivo de optimizar los resultados[1] al momento de competir contra jugadores u otros bots ya existentes.

El documento se divide en capítulos, el Capítulo 1: Definición del problema, en donde se presenta el problema abordado, el ambiente del juego y un aborde a inteligencia artificial enfocada en juegos RTS. Seguidamente se presenta el Capítulo 2: Marco Conceptual, en donde se muestran y describen tanto las herramientas utilizadas, características del juego RTS abordado, enfoques de inteligencia artificial, así como distintos trabajos y soluciones ya hechas para el problema. Luego, en el Capítulo 4: Propuesta de Solución, se presenta la solución propuesta desarrollada, el modelo del sistema, la descripción del mismo enfocado a las problemas presentes para inteligencia artificial, y las áreas implementadas junto con las distintas metodologías utilizadas. A continuación, en el Capítulo 6: Validación de la Solución, se presentan las pruebas y resultados al utilizar la solución implementada. Finalmente, en el Capítulo 7: Conclusiones, se presentan de forma concisa los resultados, se muestran los

[1] Porcentaje de victorias

alcances y limitaciones de la solución propuesta, y se entregan recomendaciones junto al trabajo futuro.

CAPÍTULO 1
DEFINICIÓN DEL PROBLEMA

Un área que ha tomado bastante importancia en los últimos años es la de *Procedural Content Generation (PCG)*[2], especialmente enfocado en el área de videojuegos. PCG es la aplicación de técnicas y algoritmos para la creación de contenido en videojuegos utilizando información, limitada o indirecta, proveniente del usuario. PCG se refiere entonces a *software* computacional que puede generar o crear contenido en si mismo, ya sea solo o junto a la ayuda de uno o más personas, sean estas jugadores o diseñadores.[Shaker et al., 2016]

La palabra 'contenido' dentro de los videojuegos puede referirse a una variedad de aspectos del mismo, pero en general es cualquier característica que el juego 'contiene', ya sean las reglas, mapas, historia, personajes, etc. En particular en este estudio se adentrará en la rama de las estrategias, que en PCG se referiría a la generación de estrategias para bots (AI), específicamente para videojuegos de estrategia en tiempo real[3] (RTS).

Los juegos de estrategia en tiempo real (RTS) son esencialmente simulaciones militares simplificadas [Robertson and Watson, 2014]. En un juego RTS, un jugador controla indirectamente muchas unidades y edificios con el uso de órdenes desde una perspectiva superior. Estas órdenes se dan y ejecutan en tiempo real, las cuales permiten recolectar recursos, construir infraestructura y, en general, controlar y administrar cada unidad y recurso perteneciente al jugador.

La palabra *bot*, es un aféresis de la palabra robot, y normalmente se utiliza para referirse a un programa informático que efectúa automáticamente tareas repetitivas, o a alguna entidad virtual que simula, en alguna medida, comportamiento humano y que realiza tareas asignadas.

En el área de los videojuegos el estilo de juegos de tipo RTS ofrece una gran dificultad a la hora de desarrollar bots dentro de los mismos. El hecho de que este estilo de juegos sea uno de los más conocidos y jugados en el mundo ha provocado que gente externa les preste más atención, tanto desde aspectos competitivos (e-sports), económicos (patrocinadores) y/o de investigación (académicos).

El ambiente que envuelve las partidas de un juego RTS es sumamente complejo y difícil de modelar. Un juego de tipo RTS se basa en muchos tipos de agentes que se mueven e interactúan constantemente, lo que resulta en un enorme conjunto de dimensiones y estados. Esto se suma al limite temporal inherente del juego en donde cada acción se realiza en tiempo real (aproximadamente una acción cada 42 milisegundos)[Wu and Markham, 2017, sta,], lo que resulta en que muchos algoritmos y técnicas comunes de inteligencia artificial se vean nulificadas o simplemente no entreguen valor alguno al momento de ser utilizadas, por lo

[2]Generación Procedimental de Contenido
[3]Real Time Strategy

mismo, el desarrollo de bots que sean capaces de entregar una solución óptima, en muy poco tiempo y dentro de un espacio de soluciones enorme, se ha vuelto un tema de interés general.

Desde mediados de los años 2000 comenzó a aparecer un gran interés en el área de la Inteligencia Artificial enfocada en videojuegos luego del llamado realizado por Buro en [Buro, 2004], y debido a eso se han ido realizando distintas investigaciones para abordar el tema. Se han ido utilizando algoritmos existentes, desarrollado nuevos algoritmos, utilizado análisis y ayuda de jugadores profesionales, desarrollado heurísticas y metaheurísticas, y en general, mezcla de todo lo anterior.

Por otro lado, los juegos RTS tienen muchos aspectos que los hacen especiales y por lo que tienen su propia taxonomía, como lo son diferentes tipos de unidades, recursos y terrenos. Dichos aspectos se detallan más adelante, en la sección 2.1. Lo importante es que un juego RTS puede ser menos o más complejo dependiendo de la cantidad de variables y dimensiones que contenga.

1.1. Juegos RTS

En el área de videojuegos hay una gran variedad de juegos RTS con más o menos variables. Los casos más conocidos son las sagas de *Age of Empires*, *Warcraft*, *Starcraft*, *Command and Conquer* y *Total War*.

Un juego RTS, o de estrategia en tiempo real, trata de jugadores que administran y manipulan unidades, estructuras y recursos para cumplir objetivos o derrotar a los oponentes. Los jugadores tienen una vista aérea general de una zona del mapa en donde pueden seleccionar, revisar y manejar unidades, a la vez que se les permite trasladar dicha zona de visión hacia otras partes del mapa. Dentro de una partida, los jugadores entran a un mapa de juego (el cual puede ser especificado o aleatorio) y deben recolectar recursos para aumentar su cantidad de unidades y la calidad de las mismas. A los jugadores se les permite elegir previamente una raza o civilización distinta, las cuales poseen conjuntos específicos de tipos de unidades, estructuras, tecnologías y/o mecánicas, con sus características y estadísticas únicas para cada uno.

Para el caso particular de este documento, el análisis y la solución esta enfocada en el juego RTS de *Blizzard*[4]: *StarCraft*[5].

StarCraft es un juego RTS que lleva casi 20 años en la industria y que, para este caso, se considera un juego RTS Avanzado. *StarCraft* presenta un mayor nivel de dificultad que la mayoría de juegos RTS, y aún más dificultad que cualquier juego de mesa, dada la complejidad y la necesidad de acciones simultáneas inmediatas.

[4] `https://www.blizzard.com/`
[5] `https://starcraft.com/`

StarCraft es un juego sumamente complejo dentro de los RTS por tener dichas características adicionales que aumentan la cantidad de variables y la cantidad de dimensiones a analizar, por ejemplo[6]:

- Tres razas distintas: *Terran*, *Zerg* y *Protross*.
- Una gran variedad de distintas unidades con características particulares para cada raza.
- Dos tipos de recursos.
- Características adicionales a las unidades: armadura, energía, escudo.

En la imagen 1 se puede ver una situación cotidiana cualquiera dentro del juego en donde el jugador, en este caso las estructuras y unidades con colores naranja, debe administrar sus edificios, trabajadores y unidades militares para apoyar a su jugador aliado, de color rojo, que esta bajo ataque por las unidades del enemigo, de color gris. Se puede ver al jugador naranja (raza *Terran*) administrando su base. Lo que incluye recolectar recursos (en este caso minerales) en la parte izquierda de la imagen, entrenar unidades de combate en la parte central de la imagen y combatir contra las unidades del jugador gris en la parte superior derecha.

Otro ejemplo se puede ver en la imagen 2, en donde un ejército de unidades del jugador naranja (triángulo T.1) se enfrentan contra las unidades del jugador azul (triángulo T.2) en un área de alto impacto táctico como lo es un puente. Además, se puede observar en el mini-mapa del juego (cuadrado M.3) que hay otras unidades, bases de jugadores, enfrentamientos, etc. en varios lugares del mapa del juego a la vez. Por lo que los jugadores deben tomar múltiples decisiones simultáneamente por situación, en múltiples escenarios dentro del mapa y en el menor tiempo posible.

A medida que el juego avanza la cantidad de estructuras y unidades que debe controlar el jugador, así como las zonas distantes del mapa, van aumentando, lo que puede generar situaciones totalmente caóticas, especialmente si se trata de un enfrentamiento de múltiples jugadores divididos en múltiples facciones. Esto se puede ver en la figura 3, en donde la cantidad de acciones y decisiones que ocurren solamente en dicha batalla es tan grande que puede tomar varios segundos, e incluso minutos, analizar la situación que sucede en dicho escenario.

1.2. Inteligencia Artificial

Los videojuegos son un dominio ideal para explorar las capacidades de la Inteligencia Artificial (IA) ya que ofrece un ambiente restringido con un set de reglas fijo, en donde técnicas

[6]Otras cualidades son mencionadas en la sección 2.1

Figura 1: Bases del jugador naranja y jugador rojo siendo atacadas por el jugador gris.

y algoritmos pueden ser desarrollados y evaluados antes de ser aplicados a problemas más complejos en el mundo real[Schaeffer, 2001]. Por lo mismo, en los últimos años ha habido un creciente interés en la investigación basada en aplicación de IA para videojuegos.

En los juegos RTS se presentan simulaciones de combates, manejo y recolección de recursos, y toma de decisiones desde un punto de vista superior, en donde la perspectiva del jugador es global sobre el mapa y sus unidades. Al jugador se le permite realizar distintas acciones sobre estas unidades y el terreno del juego para lograr el objetivo propuesto y afrontar las distintas dificultades y problemas que encontrará. En las investigaciones son, virtualmente, estas mismas acciones las que se les permite realizar a los bots utilizados, por lo que es primordial conocerlas para conocer el limite de las acciones que se pueden realizar.

1.3. Características del problema

El diseño de estrategias óptimas en tiempo real sigue siendo algo sin resolver al día de hoy. Aunque hay varias implementaciones que consiguen muy buenos resultados, en general, los bots desarrollados aún no son capaces de derrotar a jugadores profesionales, a pesar de que los bots suelen tener ciertas ventajas inherentes como pueden ser: no tener que mover el mouse o la cámara a través del mapa. Esta situación se repite con los otros juegos RTS mencionados anteriormente. Dentro del área de juegos RTS, los bots no han sido capaces de derrotar de forma segura a los jugadores profesionales, cosa que no sucede en otros tipos de juegos, como lo son los juegos de mesa o juegos por turnos, en donde los bots hace años superaron a los mejores jugadores.

Es importante recalcar que el objetivo principal al hacer investigación y desarrollo de bots no es el de incrementar el entretenimiento ofrecido por el juego RTS (un aspecto posible de

Figura 2: Escena de combate del jugador naranja contra el jugador azul. Unidades del jugador naranja, en la zona superior izquierda (triángulo T.1) intentando defender su base ante el ataque del jugador azul, con sus unidades en la zona central derecha (triángulo T.2), el cual busca avanzar por un punto crítico del mapa como lo es un puente que separa ambas zonas.

PCG), sino el de avanzar en la creación de IA que asemeje comportamiento humano. Aunque de todas maneras el aumento de competitividad por parte de la IA puede ofrecer en sí un incremento en el disfrute del jugador.

Al momento de desarrollar algoritmos para la generación de estrategias hay que recordar que actualmente muchos bots comerciales hacen 'trampa' en contra del jugador, ya sea obteniendo más recursos de los debidos o teniendo información completa del mapa. Esto no permite un avance en el ámbito de cómo crear entidades inteligentes, por lo que es crítico no caer en tales prácticas si es posible[Buro, 2004].

Los puntos a abordar para PCG de estrategias para bots, que serán descritas más en profundidad en la sección 2.1.1, son en general los siguientes:

- Planificar en tiempo real
- Toma de decisiones bajo incertidumbre.
- Modelado de oponentes
- Razonamiento espacial y temporal.
- Manejo de recursos

Figura 3: Escenario de un combate caótico.
Cúmulos de unidades, de distintas razas: Unidades *Zerg* del jugador blanco en la parte inferior izquierda, unidades *Protoss* del jugador marrón en la superior izquierda, unidades *Terran* del jugador verde y púrpura abajo a la derecha, y otras unidades destruyendo una base en la parte superior derecha.

- Colaboración entre entidades
- *Pathfinding*[7].

Dichas áreas suelen ser abordadas utilizando algoritmos y *scripts* fijos en base a experiencia de jugadores profesionales, o en base a *machine learning*, dependiendo del enfoque del bot. Todas las áreas son de igual importancia, y se les pueden aplicar distintas técnicas y heurísticas para optimizar los resultados.

Entonces, el problema abordado se centra en el diseño de estrategias considerando estas áreas. En las cuales se puede utilizar tanto conocimiento de expertos para crear *scripts* como utilizar parámetros obtenidos a partir de resultados de algoritmos de inteligencia artificial.

Aunque la implementación y desarrollo de bots y el diseño de estrategias se realiza en casos particulares, el objetivo sigue siendo conseguir resolver problemas complejos en tiempo real.

Otras muchas formas de abordar, desarrollar y enfocar el problema son descritas detalladamente en [Ontanón et al., 2013].

Entonces, se presenta un problema de análisis, administración y toma de decisiones en múltiples escenarios. Dichos escenarios corresponden a situaciones variadas que suceden simultáneamente dentro de una partida de un juego RTS.

[7]Sistema que ayuda al jugador a mover unidades de un punto a otro evadiendo obstáculos de forma automática

Un juego RTS presenta el manejo de múltiples unidades con características distintas con el objetivo de cumplir ciertas condiciones de victoria o de derrotar a los enemigos. Los jugadores se enfrentan a otros jugadores con capacidades similares y objetivos opuestos, lo que conlleva a situaciones estratégicas y tácticas que deben ser resueltas.

Debido a las características de los juegos RTS, el manejo de las unidades se realizan en tiempo real y bajo información incompleta del estado del juego, agregando complejidad tanto al modelamiento como a la implementación de un sistema autónomo que sea capaz de jugar.

CAPÍTULO 2
MARCO CONCEPTUAL

Dentro del desarrollo de estrategias para los bots, hay distintos acercamientos que suelen guiar la construcción de los mismos. Algunos suelen utilizar varios algoritmos y parámetros preconstruidos[8] junto a algoritmos de inteligencia artificial. Otros en cambio, buscan que los bots utilicen puramente algoritmos que calculen sin utilizar parámetros de decisión previos. Los últimos combinan estos dos acercamientos, utilizando algoritmos con parámetros establecidos previamente, pero desarrollados en base a aprendizaje de máquinas.

Por lo mismo, al abordar el problema se puede utilizar cualquiera de los enfoques mencionados, tanto para el juego en general, como utilizando las divisiones mencionadas en la sección 1.2. Pero, además, hay muchos más elementos a considerar a la hora de diseñar y desarrollar. Estos elementos se detallan a continuación.

2.1. Elementos a considerar en diseño y desarrollo de estrategias

Al momento de diseñar un bot para un juego RTS hay que abordar dos temáticas principales, los problemas generales de los juegos RTS y los problemas particulares de un juego RTS.

Cuando se desarrolla un bot de RTS, normalmente se aplica, practica y se diseña para una plataforma o juego en particular. Dependiendo el juego el bot tiene que enfrentarse a espacios de decisión y entornos más difíciles y complejos, esto debido al aumento de variables a considerar, e.g. distintos tipos de terrenos y elevaciones, mayor número de recursos, edificios y unidades, nuevos conceptos del juego (i.e. magia, poderes, etc.).

Por lo mismo, en este documento, se considerará un juego RTS básico (o estándar) que contenga:

- Unidades de combate, las cuales tienen cierto valor de ataque y ciertos puntos de vida.
- Edificios/Construcciones, los cuales construyen unidades de combate y trabajadores. Tienen ciertos puntos de vida y/o de armadura. Por lo menos debe haber un edificio que cree unidades de combate y uno que cree trabajadores.
- Trabajadores, unidades de recolección de recursos y construcción de edificios.
- Un recurso principal, necesario para construir edificios y crear unidades.
- Un tipo de terreno, en el cual las unidades se mueven y los edificios son construidos.

[8] Utilizando ayuda de jugadores profesionales

- Una sola raza o civilización.
- Mapa del juego descubierto.

Entonces, un juego RTS básico tiene por lo menos: un tipo de unidad de combate, un tipo de trabajador, dos tipos de edificios, un tipo de recurso y un mapa visible para los jugadores.

Cualquier juego que tenga más o menos de estas características se considerará un juego RTS avanzado o un juego RTS simplificado, respectivamente.

2.1.1. Desarrollo de Inteligencia Artificial

Debido a la amplia cantidad de dimensiones a considerar, al momento de diseñar el bot, se suele dividir el juego en varias tareas distintas. Cada una de estas áreas ofrece un mundo separado el cual resulta interesante trabajar. Al investigar sobre la generación de estrategias se suele abordar uno o más de estos aspectos [Buro and Furtak, 2004]:

- **Planificación en tiempo real:** El ambiente en los juegos RTS es uno adverso, el cual es dinámico, hostil e inteligente. Por lo mismo, ante la complejidad, es necesaria la creación de una abstracción del mundo del juego que permita realizar una búsqueda y que el resultado encontrado se pueda traducir en una solución.
- **Toma de decisiones ante adversidad:** En algunos juegos, los jugadores no son capaces de ver tanto las bases de los enemigos como los movimientos de unidades en todo momento. Por lo mismo, es necesaria la obtención de inteligencia y la toma de decisiones con poca o nula información. Esto incluye crear hipótesis y actuar acorde a ellas.
- **Modelado de oponentes:** El aprender cómo actúa normalmente un oponente en concreto o el poder clasificar a los oponentes dentro de ciertos patrones permite encontrar rápidamente debilidades y por lo tanto obtener ventaja.
- **Razonamiento espacial y temporal:** El entendimiento de relaciones temporales de las acciones y análisis del terreno es de suma importancia, especialmente porque los humanos suelen utilizar esta debilidad para vulnerar las estrategias de los bots.
- **Manejo de recursos:** La administración correcta de los recursos (materiales, unidades, recursos, etc.) es el punto central en el aspecto estratégico y por lo tanto vital al momento de generar y evaluar una estrategia.
- **Colaboración:** Aunque los juegos suelen ser 1 versus 1, hay situaciones en que varios grupos de entidades deben enfrentarse entre sí. Tanto la colaboración entre bots como la de bots con humanos agrega una nueva dimensión de dificultad a la generación de estrategias.

- ***Pathfinding*:** Se refiere al camino que deben recorrer las unidades para ir de un punto a otro. Muchos juegos RTS traen incluidos sistemas de *pathfinding* para ser utilizados por los jugadores, este es un tema de gran importancia en los RTS. La inclusión por defecto de estos sistemas en los juegos RTS refleja lo crítico que es su uso. Dependiendo del juego a abordar este puede tener mayor o menor importancia, e.g. consumo de combustible por parte de las unidades al moverse.

2.1.2. Juegos RTS

Dependiendo del juego RTS para el que se este diseñando el bot, hay una variedad aún más amplia de aspectos que se deben considerar. Algunos aspectos pueden ser, por ejemplo:

- Mayor tipo de recursos, unidades, edificios.
- Distintos tipos de terrenos.
- Distintos tipos de niveles del terreno.
- Habilidades especiales de las unidades.
- Distintos atributos de las unidades (e.g. defensa, velocidad, etc.).
- Mejoras tecnológicas dentro del juego.
- Visión de las unidades y construcciones.
- Mapa del terreno oculto (i.e. *Black Map*[9] y *Fog of War*[10]).
- Distintas razas o civilizaciones.

2.1.3. Micro y macro control

Por lo descrito en [Liquipedia, 2012], micro y macro control se refieren a dos tipos de técnicas que deben ser aplicadas al momento de jugar un juego RTS. Ambos tipos de control son cruciales a la hora de administrar las unidades y recursos. La teoría dice que se debe tener un balance entre ambos tipos de control para realizar una partida óptima. Micro control esta basado en un control local, especializado en eventos, en donde se busca controlar las unidades individualmente. Por otro lado, macro control esta basado en el manejo general del flujo del juego, esto se puede ver en la producción de unidades, administración de la economía y/o manejo de la expansión.

[9] Áreas no exploradas, de las cuales no se conoce qué contienen ni cuál es su topología

[10] Área que fue explorada en algún momento y se conoce su topología, pero de la cual no se tiene visión actual, y que por lo tanto, no se sabe qué unidades puede haber en el momento

2.1.4. Visión estratégica y táctica

La creación de un buen bot debe contemplar dos aspectos primordiales al momento de generar y aplicar una estrategia. La estrategia a utilizar debe contemplar las acciones desde un enfoque estratégico y un enfoque táctico. Estos dos enfoques son parecidos a los conceptos de micro y macro control, pero son distintos en el área de aplicación que tienen. Micro y macro control se refieren a la forma de controlar y administrar unidades y recursos, mientras que la visión estratégica y táctica va más enfocado hacia el diseño a abordar. Un enfoque estratégico esta enfocado a la implementación de cierta idea o plan, mientras que uno táctico va hacia el cómo realizar dicho plan.

Comúnmente, las técnicas utilizadas para manejar la complejidad en los juegos RTS se basan en utilizar abstracciones que a menudo entregan buenas formas de tomar decisiones estratégicas, pero puede afectar la calidad de las decisiones tácticas debido a la perdida de detalle. Contrariamente, métodos que se enfocan en el manejo de subgrupos de unidades pueden entregar buen rendimiento táctico, pero un pobre planeamiento estratégico al no coordinar de buena manera dichos grupos o no seguir un plan mayor que guíe la partida [Barriga et al., 2017]. Entonces, al igual que con los tipos de control, debería buscarse un balance entre ambos tipos de visión. Esta dualidad es pobremente abordada en la literatura ya que implica un análisis doble, el cual suele conllevar mayor tiempo de cálculo.

Es importante aclarar que los aspectos estratégico-táctico y el micro-macro control son perpendiculares entre sí, dándose la combinación de ambos ejes y resultando finalmente en cuatro aspectos a tener en cuenta: Micro estrategia, macro estrategia, micro táctica y macro táctica.

Para aclarar la diferencia entre estas cuatro clasificaciones, utilizaremos como ejemplo la idea, acciones y consecuencias de realizar un plan militar de sitiar una base enemiga. Para lograr esto podemos utilizar como estrategia el de rodear con tropas militares dicha base enemiga y combatir cualquier unidad enemiga que intente escapar de dicho encierro. La parte estratégica de dicho plan vendría dado, primeramente, por el plan en si: rodear la base, esto corresponde al aspecto macro estratégico (ámbito general de nuestra estrategia). Luego, el plan se puede ajustar para estacionar una mayor cantidad de tropas en alguna zona en específico. Esto corresponde a un control más fino sobre la ya dada estrategia: aspecto micro estratégico. Por otro lado, podemos movilizar las tropas en cada zona, si se hace que las tropas en las zonas se organicen de manera de resguardar puntos críticos del terreno, como puentes y caminos, se estaría manipulando de manera general (macro) el aspecto táctico del plan: macro táctica. Finalmente, el combate de cada grupo de tropas contra el enemigo se puede realizar de maneras particulares, ejerciendo pequeñas órdenes sobre las unidades, o sea un micro control sobre la táctica. Estos cuatro enfoques se pueden ver contrastados en la figura 4.

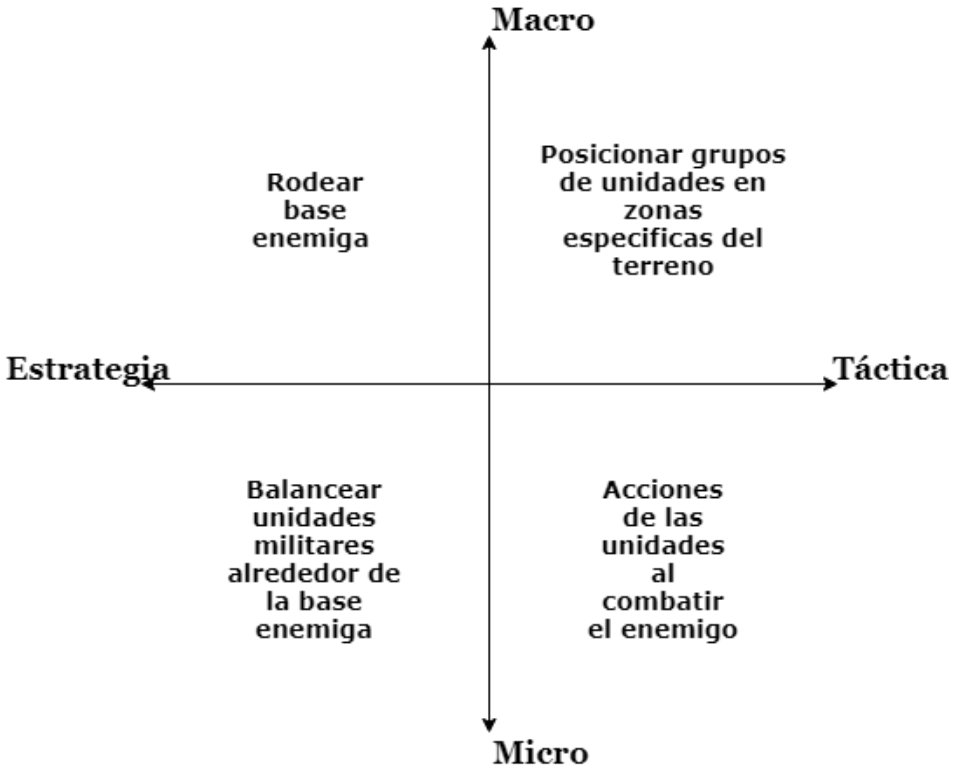


Figura 4: Enfoques estratégico-táctico versus micro-macro control

2.2. Características del RTS abordado

StarCraft es un juego sumamente complejo dentro de los RTS por tener las características adicionales descritas en la sección 2.1.2 que aumentan la cantidad de variables y la cantidad de dimensiones a analizar, en particular se caracteriza por tener:

- Tres razas distintas: *Protoss*, *Terran* y *Zerg*.
- Una gran variedad de distintas unidades con características particulares para cada raza.
- Dos tipos de recursos: *Mineral* y *Gas Vespeno*.
- Características adicionales a las unidades: armadura, energía, escudo.
- Dos tipos de niveles de terreno: tierra y aire.
- Propiedades del terreno dependiendo de la elevación.

Además, debido a las características del juego, el uso de macro y micro control tiene un fuerte impacto en el rendimiento general del jugador.

2.2.1. Fases del juego

Así como en la mayoría de juegos de estrategia (tanto RTS, como juegos por turnos o juegos de mesa), las partidas se pueden dividir en tres fases dependiendo de la duración actual de la partida. Dichas fases suelen denominarse como: *early game*, *mid game* y *late game*.

La fase de *early game* suele ser aproximadamente los primeros 7 a 11 minutos de juego, y está marcado por un avance lento, enfocado en recolectar y acumular recursos, expandir la base y armar un ejército. Normalmente, el *early game* de un jugador termina cuando el ejército armado está listo para atacar. Comparativamente con otros juegos se le puede llamar la 'Apertura'.

El *early game* suele ser la mayor fase de estudio, ya que es en esta fase donde las optimizaciones y los errores suelen afectar más el rendimiento del jugador y la estrategia a utilizar.

La fase de *mid game* suele comenzar entre los 7 y 12 minutos, y está marcada por el inicio del combate entre los jugadores, suele durar entre 5 y 10 minutos y suele ser la fase en donde se define la partida. A su vez suele ser la fase más interesante al momento de investigar algoritmos tácticos y la adaptabilidad de los bots.

La fase de *late game* está marcada por la poca cantidad de recursos disponibles o la alta dificultad y riesgo de obtener nuevos recursos. Los jugadores en este punto suelen tener todas las tecnologías desarrolladas y una estrategia consolidada en la partida, con poca posibilidad de modificación.

2.3. Estado del arte

Debido a la gran variedad de dimensiones, conceptos y enfoques posibles, el modelado, diseño y construcción de los bots en juegos RTS se puede hacer de muchas maneras en busca de lograr mejores resultados. En esta sección se describen distintos bots desarrollados en los últimos años, tanto bots dedicados a *StarCraft* como bots dedicados a otros juegos RTS, pero que proveen interesantes resultados y aplicaciones.

Los distintos enfoques en los que los algoritmos y métodos se centran son principalmente los definidos en la Sección 2.1.1. Cada algoritmo o método puede abordar uno o varios de estos enfoques simultáneamente. A continuación se mencionan algunos ejemplos de conocidas técnicas aplicadas a cada uno.

2.3.1. Planificación en tiempo real ante adversidad

Debido al complejo ambiente presentado en los juegos RTS, es necesario crear abstracciones del mismo para poder realizar búsqueda de soluciones.

En [Weber et al., 2010] se presenta una aplicación de *Goal-Driven Autonomy* para razonar sobre objetivos generales, se presenta un modelo que utiliza agentes que 'razonan' sobre sus objetivos, con lo que logran adaptarse a eventos no anticipados. Eso resulta en una separación entre la elección de un objetivo y la ejecución del mismo. Se presenta una implementación de *Goal-Driven Automony* con planes reactivos y su aplicación en *Starcraft*.

El agente desarrollado detecta *discrepancias* en el estado del juego. Estas discrepancias corresponden a acciones realizadas por el oponente que contradicen las expectaciones del bot. Dichas discrepancias pueden ser ataques recibidos del oponente, el oponente construye o crea unidades nuevas, ocurre un cambio en la cantidad de poder de combate de los jugadores, etc.

Las discrepancias detectadas por el bot se traducen en *explicaciones* para modelar las acciones actuales del oponente. Las explicaciones pueden ser por ejemplo: El oponente está construyendo unidades aéreas, el oponente se esta expandiendo, el oponente tiene una ventaja en fuerzas de combate, etc.

Luego, el bot toma estas explicaciones y genera distintos objetivos (goals) los cuales serán traducidos luego en acciones. Ejemplos de estos objetivos son: Ejecutar estrategia, atacar con todas las unidades, expandir base, entrenar nuevos trabajadores, etc. Finalmente, el bot ejecuta las acciones que correspondan para el/los objetivos propuestos.

Por otro lado, una técnica bastante utilizada es la de Optimización de Órdenes de Construcción, en donde se crea un Orden de Construcción que corresponde a una serie de unidades, estructuras y tecnologías de forma ordenada para su construcción y desarrollo. Dicho orden de construcción corresponde a un plan estratégico que el bot utilizara para guiarse.

En [Churchill and Buro, 2011] se utiliza un algoritmo de Branch-and-bound para realizar una búsqueda de orden de construcción. El algoritmo parte de un estado inicial y realiza una búsqueda recursiva depth-first con cierto objetivo dado, dicho objetivo corresponde a un *Orden de construcción*. El orden de construcción esta dado por una secuencia de acciones, unidades y edificios, que tiene como objetivo final crear cierta cantidad de unidades y estructuras basándose en los limites de recursos y las dependencias del árbol de tecnologías.

Luego, en [Churchill and Buro, 2012] se plantea el agregar al modelo anterior técnicas de dividir y conquistar al momento de crear IA para juegos RTS. Para enfrentar dicho problema se presenta UAlbertaBot como arquitectura para solucionarlo, entregando un sistema que divide las distintas áreas del bot, enfocando cada una en algún aspecto distinto de la ejecución, por ejemplo, un modulo del bot para la estrategia y otra para la táctica.

En [Fjell and Møllersen, 2012] los autores presentan un método para construir un modelo basado en estrategias, utiliza el formato de órdenes de construcción, pero basado en conocimiento experto. Se busca como objetivo el reconocer la estrategia del oponente y seleccionar una estrategia acorde capaz de vencerla. El bot razona sobre las unidades presentes para modelar de cierta manera su estrategia y luego escoge una estrategia acorde para contrarrestarla. El método incorpora pesos en el orden y el tiempo de cada estructura con el objetivo de hacer un reconocimiento avanzado.

Siguiendo la idea de realizar búsquedas de un orden de construcción en [García-Sánchez et al., 2015] se plantea un *framework* llamado *StarCraftGP*, el cual intenta crear una estrategia completa para el juego. Específicamente, se centra en la construcción de estrategias para la raza *Zerg*, y tiene un enfoque de nivel estratégico.

La implementación utiliza el *framework* como un generador automático de estrategias, usando Programación Genética (GP[11]). Este método utiliza representaciones de árboles o grafos lineales. Su funcionamiento se basa en generar programas que resuelvan cierta tarea dada. Esta técnica puede producir combinaciones de condiciones y acciones que potencialmente difieren de lo que un humano podría diseñar, haciendo posible que se puedan obtener bots desde cero, sin que se les añada conocimiento humano. Este método se diferencia de otros modelos de Algoritmos genéticos que se basan en la optimización de parámetros y restricciones.

La implementación realizada se compila en *OpprimoBot*, y se permite que sea este mismo quien se encargue del nivel táctico. La función de *fitness* utilizada para dirigir el proceso evolutivo esta basada en una serie de partidas contra oponentes seleccionados y los resultados son analizados y convertidos en métricas numéricas. Dos son las funciones que se prueban:

- Victoria: El *score* final entregado por *StarCraft* al terminar una partida.
- Reporte: Una métrica más compleja, la cual apunta a separar el éxito militar del desarrollo económico.

Por otro lado, cada nodo usado en la representación interna (genoma) corresponde a *macros*[12] con un set de parámetros. Los operadores pueden modificar los parámetros de dichas macros e incluso la estructura del genoma, agregando o removiendo tanto nodos como subárboles. Como el genoma puede ser ajustado de distintas maneras, las soluciones obtenidas pueden resultar muy complejas. Para el caso de *StarCraft*, dichas soluciones pueden ser convertidas en conjuntos de reglas o estados. Una ventaja de la Programación genética es que las soluciones propuestas pueden alejarse mucho de la intuición humana y aun así ser competentes.

En otros tipos de juegos, los bots creados a partir de GP obtienen algunas veces mejores

[11] Genetic Programming
[12] Conjunto de instrucciones

resultados que soluciones provenientes de otras técnicas, incluso derrotando a jugadores profesionales [Sipper et al., 2007].

El funcionamiento, entonces, se trata de utilizar un conjunto de restricciones codificadas para generar el código de un bot. Cada bot cumple el rol de un individuo de la población. Cada bot se enfrenta a un conjunto de oponentes seleccionados en una serie de partidas para obtener un valor de *fitness* que será utilizado como influencia en su probabilidad de reproducción, creando nuevas soluciones derivadas y permitiendo que sobreviva a través de las generaciones.

Cada bot contiene dos métodos principales. Primero, el constructor, que define el plan inicial (un Orden de construcción) y el ejército a construir. El segundo método es llamado en cada *Frame* y se encarga de computar las acciones, ya sean crear unidades, crear nuevos ejércitos, modificar el Orden de construcción, etc. basándose en la situación actual. Todo centrado en evolucionar la visión estratégica, dejando la táctica constante a cargo de *OpprimoBot*.

En la implementación, la estructura de un individuo está dividida en *Section*, *Subsection* y *Macro*. Una *Section* contiene muchas *Subsection* y aparece exactamente una vez en el genoma. Una *Subsection* puede aparecer múltiples veces y contiene muchas *Macros*, pero las *Subsections* tienen que seguir el orden estricto en que son declaradas. Finalmente, una *Macro* representa una instrucción individual o un bloque de código, y las *Macro* pueden aparecer en cualquier orden dentro de una *Subsection*. Esta estructura jerárquica permite que cualquier individuo compile sin errores.

Los individuos al tener dos métodos, poseen dos *Sections*, llamadas *initialization* y *computeActions*. Ver figura 5.

Los operadores de mutación utilizados se basan en cruzar y mutar la estructura de los individuos. Se pueden dividir dos individuos en uno o dos puntos y combinarlos. Se pueden agregar, remover o cambiar una instancia de una *Subsection*. Además, se pueden agregar, remover o cambiar instancias de una *Macro*. Finalmente, se pueden cambiar uno o todos los parámetros correspondientes a una instancia de una *Macro*.

La eficiencia de una estrategia de *StarCraft* es fácilmente medida al jugar partidas contra distintos oponentes. Comúnmente, se utilizan las victorias como principal medida para mejorar. Sin embargo, una evaluación global confiable requiere un mayor número de evaluaciones. No sólo hay que considerar que algunos planes son más efectivos en mapas específicos, sino que además algunas estrategias sirven como estrategias que contrarrestan otras, y además, el juego utiliza algunas elementos aleatorios[13] que pueden afectar el desarrollo de una partida. Por lo mismo, utilizar información extra puede permitir guiar de mejor manera la evolución.

Para manejar aquello, se utilizan las dos funciones de *fitness* ya mencionadas. La función de

[13]Una unidad que ataca desde terreno menos elevado hacia una unidad enemiga en terreno más elevado, tiene aproximadamente un 47 % de fallar el ataque.

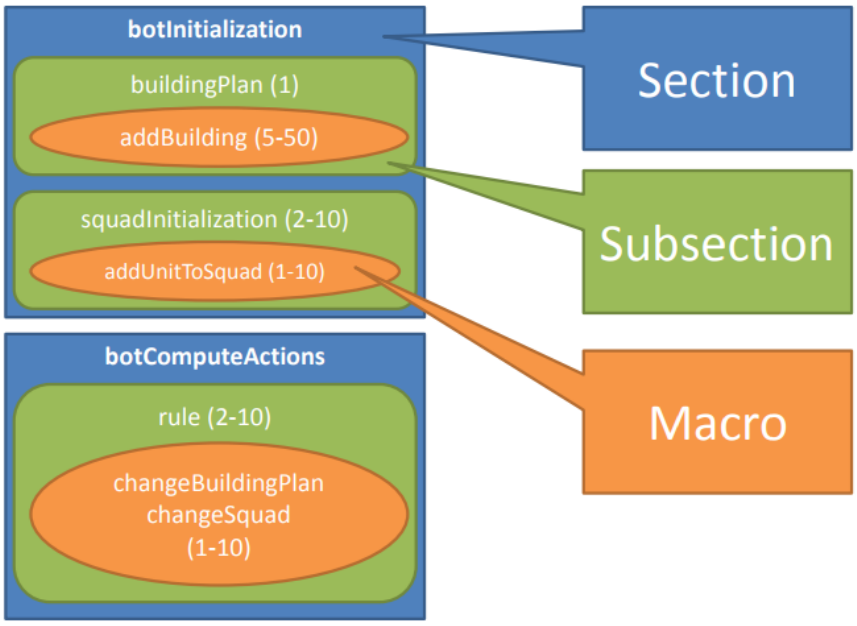


Figura 5: Estructura de un individuo.
Cada *Section* aparece una vez; cada *Subsection* puede aparecer varias veces, pero siempre en el mismo orden; *Macros* pueden aparecer cualquier cantidad de veces y en cualquier orden.

Victoria utiliza el *score* de *StarCraft* para evaluar los individuos. Mientras que la función de Reporte utiliza muchas métricas internas del juego, en particular:

- Victoria Militar: Cada vez que el individuo es capaz de derrotar al oponente dentro de un tiempo.
- Victoria Económica: Si el tiempo se acabó, se compara el puntaje por unidades y estructuras.
- Destrucción Relativa: El puntaje promedio de unidades y estructuras destruidas.
- Economía Relativa: El puntaje promedio de unidades y estructuras construidas.

Siguiendo la línea de búsqueda de órdenes de construcción, en [Wu and Markham, 2017] se plantea un algoritmo de Programación Dinámica, dicha propuesta se basa en la teoría de que entre más pronto se realice un ataque, menores serán las defensas del enemigo. En él se propone que el *Early Game* en *StarCraft* es una fase crítica y busca encontrar de la manera más rápida el orden de construcción óptimo para la formación rápida de un ejército. El método de programación dinámica planteado entrega menores tiempos de computación y además mejores resultados. El algoritmo está hecho directamente para la raza *Terran*, pero se pueden realizar los cambios pertinentes para adaptarlo a las otras dos razas.

La representación del estado del juego se hace en un vector de enteros, mientras que cada elemento del vector representa el número de cada tipo de unidad.

$$S = (\textit{Worker}, \textit{Marine}, \textit{Medic}, \textit{Firebat}, \textit{Centre}, \textit{Barrack}, \textit{Bunker}, \textit{Depot}, \textit{Academic})$$

Además, se define un vector $r_{mine}(x)$ que representa el ratio de recolección de minerales cuando hay x Trabajadores/*Workers*.

Luego, se presentan distintas restricciones para la aplicación del algoritmo:

- Restricciones de recursos: Una función $T_{build}(t)$ que indica el tiempo de construcción de la unidad tipo t. Se definen las funciones $C_{mine}(S)$ y $C_{gas}(S)$ que indican los recursos (mineral y gas) necesarios para construir las unidades del estado S.
- Restricciones de tiempo: Se definen las funciones $T_{mine}(S)$ y $T_{gas}(S)$ que indican el tiempo necesario para recolectar ambos tipos de recursos para construir las unidades del estado S.
- Restricciones de población[14]: En la raza *Terran*, un *Depot* provee 8 espacios de población. Por cada unidad de tipo t, una unidad requiere $Supply(t)$ unidades de espacios de población. Se define la restricción entonces como:

$$S(Depot) \times 8 \geq \sum_{\forall s \in S, s \neq Depot} Supply(s) \times S(s)$$

- Restricciones de precondiciones: Varias unidades de tipo t tienen como requerimiento que el jugador tenga la unidad de tipo p. Entonces, se define la función estática $Requires(t)$ que representa la unidad de tipo p necesaria para construir la unidad de tipo t. Se define $RequiereConstraint(S)$ como $\forall t \in S \Rightarrow Require(t) \in S$.
- Restricción de concurrencia: En cualquier momento de tiempo, un edificio sólo puede construir una unidad a la vez.
- Ecuación de transición de estado: Se define la función $F(S)$ que representa el mínimo tiempo requerido para lograr el estado S.

$$F(S) = \min_{\forall s \subseteq S} \max(F(S - s) + T_{build}(s), T_{mine}(s), T_{gas}(s))$$

Sujeto a:
$F(S) \leq T_{max}$
$SupplyConstraint(S)$
$RequireConstraint(S)$
$ConcurrentConstraint(S, s)$

[14]Valor que restringe la cantidad de unidades que se pueden tener en el juego

- El estado previo se almacena en $Prev(S)$. La diferencia entre estados corresponde a las unidades que se tienen que construir en dicho tiempo.

La complejidad del método es $O(nm)$, donde n es el número total de estados y m el número promedio de variaciones de estado por estado.

Tomando otros acercamiento, en [Gaudl et al., 2013] se presenta un enfoque hacia comportamientos, en donde se detalla el desarrollo iterativo de un bot basándose en un Diseño orientado en Comportamientos (Behaviour Oriented Design). El sistema utiliza un Plan de acciones (también llamado Plan reactivo), en donde se priorizan ciertas condiciones y acciones correspondientes como respuesta a que se cumplan dichas condiciones. Para determinar las prioridades entre los módulos se usa como mecanismo de selección de acciones: POSH[15]. Con este modelo el bot es capaz de jugar la raza *Zerg* en *Starcraft*.

La arquitectura del sistema tiene dos partes centrales: Las *librerías de comportamientos* que contienen primitivas que son llamadas por el selector de acciones cuando son requeridas. Y el *Selector de acciones POSH* que consiste en planes con cierta jerarquía de acciones y ciertos disparadores correspondientes. Los planes se componen de *elementos guía*, *competencias* y *patrones de acciones*. Los elementos guía corresponden a acciones o planificaciones generales que conducirán las acciones a tomar, algunos ejemplos son: Crear constructores, construir nueva base, defender base o construir ejército. Por otro lado, las competencias corresponden a atributos qué se deben obtener para dicho elemento guía. Ejemplos de competencias son: Obtener cierta cantidad de recursos para 'Crear constructores' y construir ciertos edificios para 'Construir ejército'. Finalmente, los patrones de acciones son acciones a realizar para cumplir las condiciones.

Por otro lado, Tavares plantea en [Tavares et al., 2016] un enfoque en donde clasifica distintas estrategias dependiendo qué tan buenas son contra otras estrategias, de forma de tener una colección de estrategias capaces de contrarrestar las estrategias enemigas. Con este plan reactivo se presenta un diseño que prioriza la selección de estrategias por sobre la construcción de estrategias. Debido a que los oponentes pueden seleccionar a su vez estrategias que contrarresten a las estrategias elegidas por el bot, se genera este enfrentamiento de selección al cual denominan Metajuego de Selección de Estrategias. Dicho metajuego corresponden a un juego de piedra-papel-tijera de estrategias con respecto al oponente.

Un método totalmente distinto a los anteriores se presenta en [Barriga et al., 2017], en donde se plantea un algoritmo de redes neuronales que combina la visión estratégica y la visón táctica, y en el que se observa un mejor ratio de victorias que algoritmos que se enfocan independientemente en dichas visiones.

El algoritmo logra disminuir el tiempo utilizado en la construcción de la estrategia, y así asignar mayor tiempo de cálculo para búsqueda táctica.

[15]Parallel-rooted Ordered Slip-stack Hierarchical

El algoritmo se implementa y se aplica en μRTS.

La implementación propone entrenar una Red Neuronal Convolucional (CNN[16]) para predecir la salida de *Puppet Search*, dejando la mayoría del tiempo libre para el uso de un algoritmo de búsqueda táctica. *Puppet Search* es un algoritmo de búsqueda estratégica que usa abstracciones.

El uso de *Deep Learning*, y específicamente CNN ha logrado buenos resultados para problemas con espacios de decisión enormes. Pero, se asume total visibilidad del ambiente y fallas de comunicación entre los agentes.

Por otro lado, *Puppet Search* es un algoritmo de búsqueda que ha entregado buenos resultados en grandes escenarios de RTS. Utiliza rápidos *scripts* con una pequeña cantidad de puntos de elección. Estos *scripts* son normalmente estrategias escritas directamente en el código, y los puntos de selección dependen de las restricciones de tiempo. Estos puntos de selección se pasan a algún algoritmo del estilo *adversarial look-ahead* como Alpha-beta o Monte Carlo. Luego, el algoritmo utiliza un *Forward Model* para examinar los resultados de las distintas combinaciones de elección y decidir el mejor curso de acción. Este algoritmo suele entregar no tan buenos resultados en escenarios pequeños, debido al pobre y estático nivel táctico.

La red utilizada esta compuesta por dos capas convolucionales seguidas por dos capas adicionales 'totalmente' conectadas. El tamaño del escenario en donde se ejecuta afecta enormemente el tamaño del problema, especialmente en las dos capas adicionales conectadas. Para enfrentar este problema, se diseña una Red Totalmente Convolucional (FCN[17]) que sólo consiste de capas convolucionales intermedias. Esta estructura tiene dos redes: *Evaluation Network* y *Policy Network*, las cuales solo difieren en la primera y última capa, como se muestra en la Tabla 1.

La primera capa de la *Policy Network* tiene un plano adicional, que indica qué estrategia del jugador se está computando. La capa final de la *Evaluation Network* tiene dos salidas, indicando si el ganador es el jugador 1 o el jugador 2. Por otro lado, la *Policy Network* tiene cuatro salidas, indicando una de cuatro posibles acciones. Esta estructura permite que el número de parámetros de la red no aumente cuando el tamaño del mapa incrementa.

El uso de *Puppet Search* requiere un *Forward Model* para examinar el resultado de distintas acciones. La mayoría de juegos RTS no tienen un *Forward Model* o un simulador más que el juego en si mismo, lo cual puede resultar muy lento. Por lo mismo, el uso de la *Policy Network* para seleccionar *scripts* durante el transcurso del juego permite evadir la necesidad del *Forward Model*. El uso de este algoritmo permite reducir el tiempo requerido por *frame* a 3ms en comparación con los 100ms de los algoritmos de búsqueda, lo que permite utilizar el tiempo ahorrado en algoritmos tácticos.

[16]Convolutional Neural Network
[17]Fully Convolutional Network

Tabla 1: Arquitectura de la red neuronal

Evaluation Network	Policy Network
Input 128x128, 25 planes	Input 128x128, 26 planes
2x2 conv, 32 filters, pad 1, stride 1, LReLU Dropout 0.2	
3x3 conv, 32 filters, pad 0, stride 2, LReLU Dropout 0.2	
2x2 conv, 48 filters, pad 1, stride 1, LReLU Dropout 0.2	
3x3 conv, 48 filters, pad 0, stride 2, LReLU Dropout 0.2	
2x2 conv, 64 filters, pad 1, stride 1, LReLU Dropout 0.2	
3x3 conv, 64 filters, pad 0, stride 2, LReLU Dropout 0.2	
1x1 conv, 64 filters, pad 0, stride 1, LReLU	
1x1 conv, 2 filters pad 0, stride 1, LReLU	1x1 con, 4 filters pad 0, stride 1, LReLU
Global averaging over 16x16 planes	
2-way softmax	4-way softmax

La implementación del algoritmo combina la FCN para decisiones estratégicas y un algoritmo *Adversarial look-ahead* para la táctica. En este caso la componente estratégica maneja el Macro control militar, económico y de producción, mientras que la componente táctica maneja el Micro control durante el combate.

El algoritmo construye una visión del estado del juego basándose sólo en las unidades que están cerca de unidades enemigas. En caso de que este estado limitado esté vacío, entonces toda la computación se asigna a la componente estratégica, en caso contrario los recursos se comparten. Finalmente, el algoritmo estratégico computa las acciones para todas las unidades, seguido por el algoritmo táctico que computa para las unidades en el estado limitado. En caso de que ambos algoritmos presenten acciones para una misma unidad, las acciones del algoritmo táctico priorizan.

Además, en [Naves and Lopes, 2012], mencionado en la sección 2.3.4, se crea un plan estratégico al utilizar las técnicas aplicadas para *Maximizar la recolección de recursos* y *Optimizar el desarrollo de un ejército*.

2.3.2. Toma de decisiones ante adversidad

En algunos juegos, los jugadores no son capaces de ver las unidades enemigas ni el terreno sin haberlo explorado previamente. Por lo mismo, es necesaria tanto la obtención de inteligencia como la toma de decisiones bajo poca o nula información. Esto incluye crear hipótesis y actuar acorde a ellas.

Un acercamiento común al momento de enfrentarse a este problema es el de los Modelos Bayesianos. Ejemplos de ellos es [Synnaeve and Bessiere, 2011b] en donde se utiliza un Modelo Bayesiano que utiliza la información alrededor de las unidades como si fueran sensores en un robot, basando la decisión de qué movimiento hacer en tal situación. Este movimiento puede incluir el moverse, atacar o utilizar habilidades. Además, recibe una señal de una jerarquía superior que indica si debe huir o pelear.

En [Synnaeve and Bessiere, 2012] se describe un Modelo Bayesiano generativo de ataques tácticos, el cual sirve tanto para predecir ataques como para tomar decisiones tácticas. Este modelo detecta las unidades enemigas, guarda dicho información y la utiliza con conocimiento previo para predecir los movimientos futuros, esto permite manejar incertidumbre en las posiciones de las unidades enemigas y el árbol de tecnologías desarrollado.

Otro acercamiento con Modelos Bayesianos es el presentado en [Parra and Garrido, 2012] en donde se propone un modelo que toma la información previa desde un jugador humano, y luego la utiliza para imitar su comportamiento en la toma de decisiones y en el micro manejo de las unidades de combate. El bot utiliza conocimiento proveniente de las partidas de una persona y luego intenta imitar dicho comportamiento aplicando redes bayesianas. Este acercamiento ayuda a enriquecer la experiencia de juego del jugador sujeto, ya que crea un modelo que puede predecir las acciones del mismo jugador, lo que posteriormente puede utilizarse para sistemas de inteligencia artificial que provean soporte a cada jugador.

Por otro lado, otros acercamientos se han hecho utilizando otros métodos, como lo son algoritmos de búsqueda. Entre ellos se destacan algoritmos de enjambre (*Swarm Intelligence*), Monte-Carlo y Algoritmos genéticos.

En [Lin and Ting, 2011] Lin y Ting plantean una táctica basada en formaciones emergentes, utilizando Algoritmos genéticos para abordar el problema táctico en *Starcraft*. El algoritmo genético parametriza las formaciones que pueden tomar escuadrones de unidades, y luego utiliza un algoritmo genético para evolucionar dichos parámetros. Finalmente el bot utiliza dichos parámetros para armar formaciones con la que contrarrestar ciertas unidades. En la figura 6 se puede observar una formación táctica siguiendo parámetros que indican que una unidad debe estar detrás de una unidad vecina dada y esta debe escoger lado izquierdo o derecho, con lo que se logran estas formaciones diagonales.

En el caso de Monte-Carlo, en [Zhe et al., 2012] se presenta una aplicación de Monte-carlo planning (MCPlan) capaz de modelar oponentes a la vez que controla las unidades, esto enfocado en *StarCraft*. En el modelo se utiliza MCPlan para calcular y controlar los movimientos

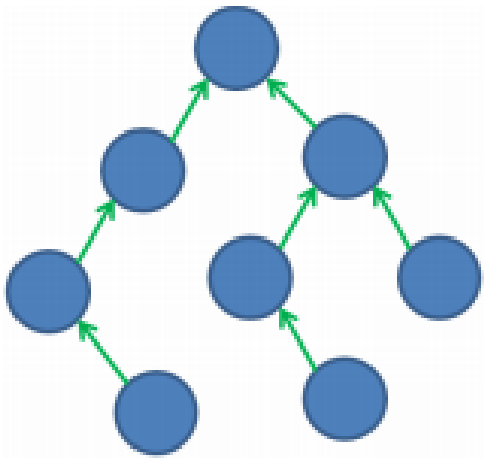


Figura 6: Estructura de una formación.
Estructura con dos reglas simples: 1) estar detrás de una unidad vecina dada y 2) estar un poco hacia la derecha o izquierda.

dados a cada unidad, además se aplica un algoritmo $\epsilon-$ greedy para modelar oponentes en una simulación para mejorar el rendimiento de MCPlan ayudándolo a decidir el mejor curso de acción.

Otro método usado es el de *Swarm Intelligence*[18]. En [Gonzalez and Garrido, 2011] se propone un método que trata a las unidades como si fueran miembros de un enjambre, el método se basa entonces en utilizar inteligencia de enjambre para controlar un enjambre que corresponde a las unidades militares. Este método que controla unidades del mismo tipo y provee una forma de que dichas unidades alcancen una posición adecuada para atacar a un grupo enemigo que a su vez esta conformado por unidades del mismo tipo.

Además, en [Tavares et al., 2014] se utiliza inteligencia de enjambre para modelar agentes que resuelven problemas de asignación de tareas, en donde a cada agente se le asignan tipos de tareas específicas para ser ejecutadas con fin de lograr cierto objetivo dado, como puede ser el 'recolectar mineral' con el fin de crear cierta estructura o desarrollar alguna tecnología. Se aplica un algoritmo de asignación de tareas basado en Inteligencia de enjambre y se ajustan los parámetros del mismo utilizando un algoritmo genético. El algoritmo utilizado se llama *Swarm-GAP* que es la aplicación de inteligencia de enjambre a un modelo de *Extended Generalized Allocation Problem (E-GAP)*, las tareas a asignar incluyen la construcción de edificios, recolección de recursos, entrenamiento de unidades, etc.

Un enfoque distinto que permite tomar decisiones respecto al movimiento de unidades proviene desde la física y es aplicado en robótica, este es el de utilizar Campos Potenciales. Los campos potenciales asignan fuerzas de atracción y repulsión a posiciones del mapa para

[18]Inteligencia de enjambre

guiar el movimiento de los entes. En [Hagelbäck, 2012] se utiliza una combinación de A* y Campos potenciales (*Potential Fields*) para manejar de mejor forma el aspecto dinámico del micro control de unidades, permitiendo desplazar las unidades a través del mapa evadiendo o persiguiendo enemigos de forma de llegar a un objetivo dado.

Otro acercamiento con campos potenciales es presentado en [Rathe and Svendsen, 2012] en donde se deja el control de las unidades a los campos potenciales, los cuales son ajustados mediante el uso de Algoritmos evolutivos multi objetivos, específicamente *Nondominated Sorting Genetic Algorithm* (NSGA-II). El modelo planteado controla las unidades posicionando distintos tipos de campos potenciales en el mapa. Los pesos de los campos potenciales que se usan al calcular son optimizados mediando NSGA-II. Los entes que entregaran fuerza a los campos son constantes, como rampas, murallas, depósitos de mineral, etc. Y variables, como edificios y unidades. Los parámetros que optimiza NSGA-II varían el peso que se le da a ciertos atributos al calcular los campos potenciales, dichos atributos incluyen el daño de ataque, la armadura, la vida restante, la posición actual, entre otros.

En [Louis and Liu, 2018] también se utiliza NSGA-II para optimizar los parámetros de los campos potenciales. Dichos campos potenciales utilizan primeramente un Mapa de influencia en donde se divide el mapa en un grilla, y luego a cada grilla se le calcula la influencia de cada unidad en cierto radio. Luego junto a este mapa de influencias y a distintos campos potenciales, los cuales consideran el tiempo de enfriamiento del arma, la vida porcentual restante y la distancia a cada unidad para calcular el objetivo a perseguir y el movimiento adecuado para lograrlo.

Otra forma de atacar el problema de la toma de decisiones es mediante el uso de redes neuronales artificiales. Un ejemplo de ello es [Wender and Watson, 2012] en donde se presenta un modelo que utiliza Reinforcement Learning para el micro manejo de las unidades de combate en *Starcraft*, en donde la red neuronal se entrena para controlar los movimientos y acciones de las unidades en el combate. Para confrontar el problema innato de recompensa no inmediata de *Starcraft* se utilizan aplicaciones de Q-Learning y Sarsa algorithm.

Finalmente, otros algoritmos enfocados para la toma de decisiones ante adversidad son: [Barriga et al., 2017], discutido en la sección 2.3.1, en donde se utiliza *Adversarial Look-ahead* junto a la FCN para tomar las decisiones; [Weber et al., 2010], en donde se utiliza una *Goal-driven Autonomy*.

Por otro lado, en [Churchill and Buro, 2012] se plantea la arquitectura de UAlbertaBot que utiliza un script de *Attack weakest unit in range* para las decisiones tácticas de combate. Y en [Alburg et al., 2014] se utiliza un Modelo Bayesiano para predecir las unidades oponentes y asistir la toma de decisiones.

2.3.3. Modelado de oponentes

El aprender cómo actúan los oponentes es vital para poder clasificarlo dentro de ciertos criterios y patrones, lo que permite detectar debilidades y desventajas.

Para modelar oponentes un enfoque bastante utilizado es el de aplicar modelos bayesianos. Algunos ejemplos son [Synnaeve and Bessiere, 2011a], [Oen, 2012] y [Jónsson et al., 2012] en donde se presentan algoritmos con un modelo bayesiano para predecir la estrategia inicial[19] de los oponentes basándose en información recolectada mediante exploración, que contrasta edificios y unidades descubiertas con marcas de tiempo [20] provenientes de información previa. Además, este modelo puede predecir el árbol tecnológico que esta desarrollando el oponente.

En [Hostetler et al., 2012] se plantea las dificultades y costes de explorar con el objetivo de obtener información. Por lo mismo, se presenta un Modelo Bayesiano Dinámico para inferir aspectos no observados durante el juego utilizando una combinación de un modelo generativo de cómo las estrategias se relacionan con objetos observables y un framework que incorpora evidencia obtenida mediante la exploración del mapa.

En [Alburg et al., 2014] se presenta un bot de *StarCraft* capaz de predecir el estado del árbol tecnológico del oponente utilizando una Red Bayesiana (modelo bayesiano). Además, modela cómo estas predicciones ayudan a asistir al bot en la toma de decisiones estratégicas. Se utilizan las redes bayesianas para obtener predicciones y luego el agente encargado de la composición de las unidades se genera dinámicamente basado en las predicciones y las observaciones de las unidades oponentes.

Modelos alternativos a los modelos bayesianos que se han propuestos son los de Modelos ocultos de Markov y programación utilizando *Answer set programmin*(ASP). Ejemplos de estos son [Dereszynski et al., 2011] en donde se estudia el problema de aprender modelos probabilísticos de comportamientos en estrategias de alto nivel. Los modelos se aprenden automáticamente desde un conjunto de datos previos, y se apunta a capturar los estados y puntos de decisiones críticos que emergen de dichos juegos. El acercamiento centrado en datos propuesto no esta limitado o presenta prejuicios ante estrategias preconcebidas. El modelo de comportamientos se basa en modelos ocultos de Markov y soporta una variedad de usos en el diseño de IAs, los que incluyen modelos de aprendizaje para predicciones de las acciones futuras de los jugadores o identificar estrategias nuevas que no se encuentran en los datos previos.

Por otra parte, en [Stanescu and Čertickỳ, 2014] se presenta un método para predecir la combinación de unidades producidas más probable por un oponente durante cierto periodo de tiempo. Se utiliza Answer Set Programming (ASP) como paradigma ya que su semántica se adapta bien al razonamiento bajo incertidumbre e información incompleta.

[19] También llamado *Opening*
[20] *Timestamp*

Otros ejemplos son [Synnaeve and Bessiere, 2012], mencionado en la Sección 2.3.2, en donde también se utiliza un modelo bayesiano para la predicción de los ataques oponentes. Y [Fjell and Møllersen, 2012], mencionado en la sección 2.3.1, en donde se modela la estrategia del oponente haciendo un mapeo comparativo con información previa. Parecido a los métodos de modelos bayesianos.

2.3.4. Razonamiento espacial y temporal

El entendimiento de las relaciones temporales de las acciones y el análisis del terreno es primordial a la hora de desarrollar agentes inteligentes.

Un acercamiento al razonamiento espacial es el de analizar el terreno y las regiones que lo componen. En [Perkins, 2010] se presenta un algoritmo que particiona el terreno en un conjunto de polígonos y computa los *choke points* o cuellos de botella que se generan entre regiones adyacentes. Esta representación se utiliza como componente en agentes de IA para razonar en relación al terreno, planear múltiples rutas de ataque y hacer otras decisiones tácticas.

Por otro lado, en [Weber et al., 2011] se propone otro acercamiento para razonamiento temporal y espacial, en donde se presenta un modelo para predecir la posiciones de las unidades enemigas basándose en las posiciones en donde se detectaron. El modelo utiliza un acercamiento basado en partículas en donde se representa las estimaciones como un problema de optimización, y automáticamente se aprenden los parámetros para el modelo de partículas desde repeticiones de partidas de expertos. El modelo sigue las unidades del oponente y provee condiciones para activar comportamientos tácticos en las unidades del bot.

Algunos algoritmos tratados en secciones anteriores también tienen un enfoque de razonamiento espacial o temporal. Uno de esos casos es [Weber et al., 2010] de la sección 2.3.1, donde *Goal-Driven Autonomy* tiene un razonamiento temporal implícito.

2.3.5. Manejo de recursos

En el área competitiva, la administración correcta y óptima de los recursos disponibles, tanto materiales y recursos naturales, como las unidades y el tiempo, es el punto central en el aspecto estratégico y, por lo tanto, vital al momento de generar y evaluar distintas estrategias.

En [Naves and Lopes, 2012] se plantea que los juegos RTS se caracterizan por dos fases. La primera se enfoca en la recolección de recursos y el desarrollo de un ejercito. La segunda fase trata de utilizar los recursos producidos para batallar contra los oponentes. Se presenta un enfoque hacia la primera fase, en donde se trabaja sobre la elección de objetivos a lograr durante el juego, en donde se busca maximizar la producción de recursos basado en planeamiento y búsqueda estocástica.

Por otro lado, como manejo de las estructuras, en [Certicky, 2013] se presenta la técnica utilizada por jugadores profesionales de *StarCraft* llamada *Wall-in*, la cual corresponde a una forma de posicionar las estructuras del jugador de forma de obstaculizar al oponente para que este logre entrar a la base propia. Certicky aborda este problema y presenta una solución declarativa utilizando el paradigma de Answer Set Programming (ASP).

Un modelo más enfocado en los recursos económicos fue presentado en [Weber, 2018], en donde se revisa un método utilizado en una de las IA participantes en el torneo de *StarCraft*; CUNYbot. CUNYbot toma decisiones estratégicas usando un modelo económico de pocas dimensiones que tradicionalmente escribe el comportamiento de países. CUNYbot utiliza un algoritmo genético para optimizar los parámetros del modelo económico. Además, con esto permite predecir ciertos aspectos de la estrategia del oponente.

Otros métodos que presentan acercamientos sobre manejo de recursos son, por ejemplo, [Churchill and Buro, 2011], mencionado en la sección 2.3.1, que tiene una parte que se encarga de manejo de recursos, en donde se utiliza *Resource Optimization* como objetivo para mejorar la estrategia.

2.3.6. Colaboración

En su mayoría las partidas en los juegos RTS son del estilo 1vs1, pero hay situaciones en que un mayor grupo de entidades deben enfrentarse entre sí, ya sea solas o formando alianzas. Esta colaboración agrega una nueva dimensión a la dificultar al momento de generar estrategias y manejar tácticas.

Aunque la colaboración entre bot y bot o entre bot y humano no es un ámbito muy investigado, en [Magnusson and Balsasubramaniyan, 2012] se presenta un estudio en donde se desarrolla un bot con comunicación y habilidades de control. Se estudia la interacción del bot con participantes humanos y se investiga qué comportamientos les gusta o disgusta.

Los estudios respecto a la colaboración son prácticamente nulos, esto debido a que se requiere toda una estructura o sistema por detrás para apoyar la idea de un ente que colabore (con humanos u otros bots), i.e. se requiere un sistema que abarque todas las otras áreas de investigación y además acepte algún protocolo de comunicación.

2.3.7. Pathfinding

La búsqueda de caminos es un punto vital y transversal a la hora de aplicar y ejecutar cualquier estrategia o táctica. Muchas veces este aspecto se deja de lado en el desarrollo de bots para juegos RTS debido a que existe la posibilidad de utilizar los algoritmos de *pathfinding* innatos del juego. Pero, dependiendo del juego este puede tener mayor o menor importancia.

El método de [Synnaeve and Bessiere, 2011b], mencionado en la Sección 2.3.2, aborda el problema de cómo mover grupos de unidades tanto fuera como dentro de un combate para lograr que no choquen o se molesten entre sí. Utiliza un Modelo bayesiano para tomar la decisión de hacia donde mover cada unidad.

Por otro lado, en [Hagelbäck, 2012] se utiliza *Potential Fields* con A* para movilizar las unidades a través del mapa.

Resumen

Dentro de los juegos RTS existe una gran variedad de características y de complejidad que diferencian a los distintos juegos. En este trabajo se estudia un juego RTS de complejidad relativa media: *StarCraft*, el cual posee un conjunto interesante de propiedades a abordar al momento de implementar bots y algoritmos de inteligencia artificial.

La investigación de inteligencia artificial en los juegos RTS se ha clasificado en distintas áreas, descritas en la sección 2.1.1, usando herramientas como branch-and-bound, programación genética y redes neuronales para la planificación en tiempo real, algoritmos genéticos y Monte-carlo para toma de decisiones ante adversidad, algoritmos de búsqueda para manejo de recursos, etc. Por lo mismo, se han realizado distintas investigaciones sobre cada área, ya sea por separado o en un subconjunto de las mismas. Para el caso de este trabajo (como se describirá en detalle en la sección 4.3) es importante destacar las áreas de planificación en tiempo real y de toma de decisiones ante adversidad, ya que ambas áreas son las que afectan en mayor medida a los aspectos estratégicos y tácticos del bot. En particular, es importante tener en mente la optimización de un orden de construcción presentado en [Churchill and Buro, 2011] y [Churchill and Buro, 2012], y la optimización de campos potenciales presentada en [Louis and Liu, 2018]. Estos algoritmos servirán como base e inspiración para los algoritmos propuestos en el Capitulo 4.

CAPÍTULO 4
PROPUESTA DE SOLUCIÓN

En este documento se propone una arquitectura que permite una estructura modular con respecto a los algoritmos de Inteligencia Artificial que se pueden utilizar. Se utilizan distintas clases que obtienen información y utilizan los algoritmos para crear soluciones para las distintas áreas mencionadas en la Sección 2.1.1. Primeramente se explica el modelo general y la arquitectura del mismo en la sección 4.1, dando a conocer las distintas entidades que interactúan dentro del sistema y una breve lista de sus funciones. Luego, en la sección 4.2 se muestra en mayor detalle las funciones de cada entidad y cómo interactúan y se comunican entre ellas.

Este trabajo se centra particularmente en la estrategia y la táctica envuelta en la planificación de la estrategia general y las acciones de las unidades. Por esto, la implementación, detallada en la sección 4.3 esta centrada en solo un par de áreas de investigación, específicamente Planificación en tiempo real y Toma de decisiones ante adversidad. Se detalla cómo cada parte del modelo ataca cada área y el por qué estas no se desarrollan para esta implementación. Luego, en la sección 4.3.1 se presentan las estructuras encargadas de los aspectos estratégicos y tácticos, así como los algoritmos que las generan, presentando el flujo de información de estos. En la sección 4.3.2 se presenta BWAPI como herramienta de desarrollo en el juego. Finalmente, se describen los algoritmos utilizados y a continuación la implementación de cada uno de ellos en el sistema desarrollado.

4.1. Modelo general

El modelo general del sistema busca modularizar las partes del bot con el objetivo de separarlas y lograr tratar cada área de investigación de forma separada. El sistema utiliza seis entidades principales distintas para abarcar las siete áreas mencionadas previamente en la sección 2.1.1. Estas entidades realizan distintas funciones y se comunican entre sí. Cada una puede poseer distintos algoritmos que se encarguen de manejar la información y realizar las tareas correspondientes a dichas áreas. Además, se utiliza una jerarquía de entidades sobre las unidades manipulables del juego, separando las labores de administración de la base con las labores de movimientos militares y combate.

4.1.1. Arquitectura del modelo

La arquitectura general del sistema (Ver figura 7)[21] esta dividida en distintos módulos que se comunican entre sí. Cada uno encargándose de un área distinta del juego. Primeramente, en la parte central esta **General** que se encarga de actualizar las unidades, juntar la información y controlar el resto de ámbitos del sistema. Asigna las unidades a las distintas partes del sistema correspondiente a su objetivo y función.

Por un lado esta **Commander**, que lidera **Leader**s y **Explorer**s, el cual se encarga del aspecto militar. Asigna las unidades tipo *Militar* a el/los *Leader* y tipo *Scout* a el/los *Explorer*. El *Commander* puede crear o destruir *Leaders* o *Explorers* de acuerdo a las necesidades del momento.

En contraste, para la administración de la base, recolección y manejo de recursos, construcción de unidades y estructuras, y desarrollo de tecnologías, se encarga **BaseManager**. El cual utiliza información estratégica y de planificación para realizar las tareas utilizando unidades tipo *Worker*.

Planner es el encargado de conocer la estrategia y la planificación de la base del momento y crear un plan de construcción con las posiciones correspondientes de cada estructura para lograr un objetivo en concreto; un ejemplo de esto puede ser el de realizar un *Wall-in*[22].

La comunicación con otros jugadores esta a cargo de la entidad **Messenger**, la cual envía y recibe mensajes, y los traduce como información hacia *General* para su uso.

Analyst se encarga de analizar las unidades oponentes que se detectan y modelar la estrategia y táctica que planea/utiliza el oponente.

Tactician y **Strategist** son los agentes principales en este trabajo, estos utilizan algoritmos para generar y actualizar los distintos planes, estrategias y tácticas, tanto micro como macro, que utilizan las demás entidades del sistema. La investigación presentada en este documento se centra fuertemente en estas dos entidades y en los algoritmos utilizados por ellas.

En la figura 7 se observa la interacción de las entidades y algunos aspectos del juego. A continuación se menciona de forma general el comportamiento de cada una:

- **General**
    - Recibe la información completa desde las entidades y maneja la información del juego.

[21] Los diagramas presentados en la sección 3 fueron desarrollados con la herramienta de diagramas *Visual Paradigm*

[22] Construir edificios dentro de la base de tal manera de utilizarlos como una muralla, siendo esta no su función primaria.

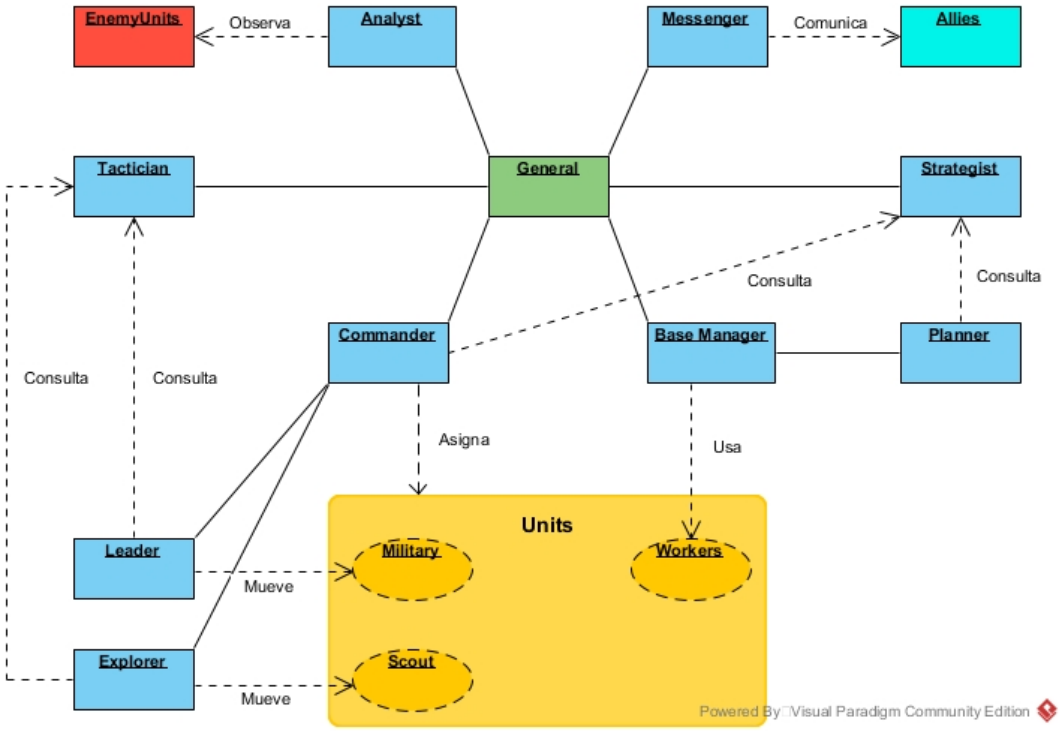


Figura 7: Modelo del Sistema.
Fuente: Elaboración propia.

- Controla las unidades dirigidas por el bot y las asigna a Commander y BaseManager.
- Envía y recibe información de Messenger para comunicarse con otros jugadores.
- Entrega información relevante a Strategist, Tactician y Analyst.

- **Messenger**
  - Envía y recibe información a otros jugadores.
  - Compila y envía información a General.
- **Analyst**
  - Analiza las unidades enemigas.
  - Predice la estrategia y el Árbol de tecnologías enemigo.
- **Commander**
  - Recibe el control de las unidades de tipo Militar y Scout desde General.
  - Consulta las Órdenes de posicionamiento a Strategist.
  - Asigna unidades *Scout* a Explorer.

- Crea Leaders y les asigna unidades *Militar*.
- Envía órdenes a Leaders y Explorers.

- **Leader**
  - Ejecuta órdenes provenientes de Commander.
  - Consulta Tactician.
  - Maneja unidades *Military* asignadas.
- **Explorer**
  - Ejecuta órdenes provenientes de Commander.
  - Consulta Tactician.
  - Maneja unidades *Scout* asignadas.
- **BaseManager**
  - Consulta Planner.
  - Utiliza *Workers* para construir y recolectar recursos.
  - Crea unidades y construye edificios.
- **Strategist**
  - Recibe información desde General.
  - Utiliza algoritmos para generar un Orden de construcción (macro estrategia).
  - Utiliza algoritmos para generar Planificación de base (micro estrategia).
  - Utiliza algoritmos para generar Órdenes de posicionamiento de las unidades (macro estrategia).
- **Tactician**
  - Recibe información desde General
  - Utiliza algoritmos para generar Órdenes de exploración (macro táctica).
  - Utiliza algoritmos para generar Órdenes de movimiento para las unidades (macro táctica).
  - Utiliza algoritmos para generar Órdenes de combate para las unidades (micro táctica).
- **Planner**
  - Consulta el Orden de construcción y la Planificación de base a Strategist.
  - Utiliza algoritmos para determinar las posiciones de los edificios a construir.

Para los casos de *Strategist* y *Tactician* es importante aclarar el significado general de las representaciones generadas por los algoritmos. Para el caso de *Strategist*, un Orden de construcción corresponde a una secuencia de unidades, estructuras y tecnologías a desarrollar (se verá un ejemplo más adelante en la sección 4.3). Planificación de base corresponde a una idea general del posicionamiento de las estructuras a construir, i.e. construir los edificios en posiciones que tengan un impacto estratégico, y que no sea en posiciones arbitrarias. Órdenes de posicionamiento corresponde a regiones del mapa del juego que deben ser ocupadas por las unidades, i.e. que las unidades se posicionen en zonas del mapa que permitan una ventaja estratégica o táctica, y que no estén en posiciones arbitrarias sin mayor impacto.

Por su parte, para el caso de *Tactician*, las Órdenes de exploración corresponden a zonas del mapa que deben ser exploradas o que deben ser vigiladas para obtener información relevante. Las Órdenes de movimiento corresponden al macro control de un grupo de unidades militares con el fin de ocupar las zonas designadas por las órdenes de posicionamiento de una forma tácticamente óptima. Finalmente, las Órdenes de combate corresponden a las acciones de micro control que ejecutarán las unidades a la hora del combate.

4.2. Detalles del modelo

La parte principal del sistema es *General*, que se encarga de manejar la información y comunicarse con todo el resto de partes del sistema. *Tactician* y *Strategist* utilizan algoritmos para crear estructuras que representan estrategias y tácticas militares que se encargan de dirigir las decisiones del bot. *Commander* recibe las unidades militares y las unidades de exploración, luego las asigna a un grupo de uno o más *Leaders* y un grupo de uno o más *Explorers*. *BaseManager* recibe los trabajadores y realiza las instrucciones provenientes de *Planner*. *Analyst* observa las unidades enemigas, recopila información y analiza el comportamiento de los oponentes. *Messenger* se comunica con el resto de jugadores.

En la figura 8 se puede observar las relaciones entre las entidades, los atributos que manejan y las funciones que realizan. A continuación se detalla y describe el comportamiento general de cada entidad y su funcionamiento e interacción dentro del sistema:

- **General**
  - *UnitsOwned* es la lista de unidades controladas por el jugador en el instante dado.
  - *Update()* se encarga de tomar la información del instante actual y actualizar la información correspondiente a las unidades que permanecen con vida, actualizando *UnitsOwned*.
  - *AssignUnits()* toma las unidades correspondientes de *UnitsOwned* y pasa el control a *Commander* y *BaseManager* según corresponda por tipo de unidad.
  - *SendInfo()* envía actualizaciones de la información del juego e información de *Analyst* a *Strategist* y *Tactician*.

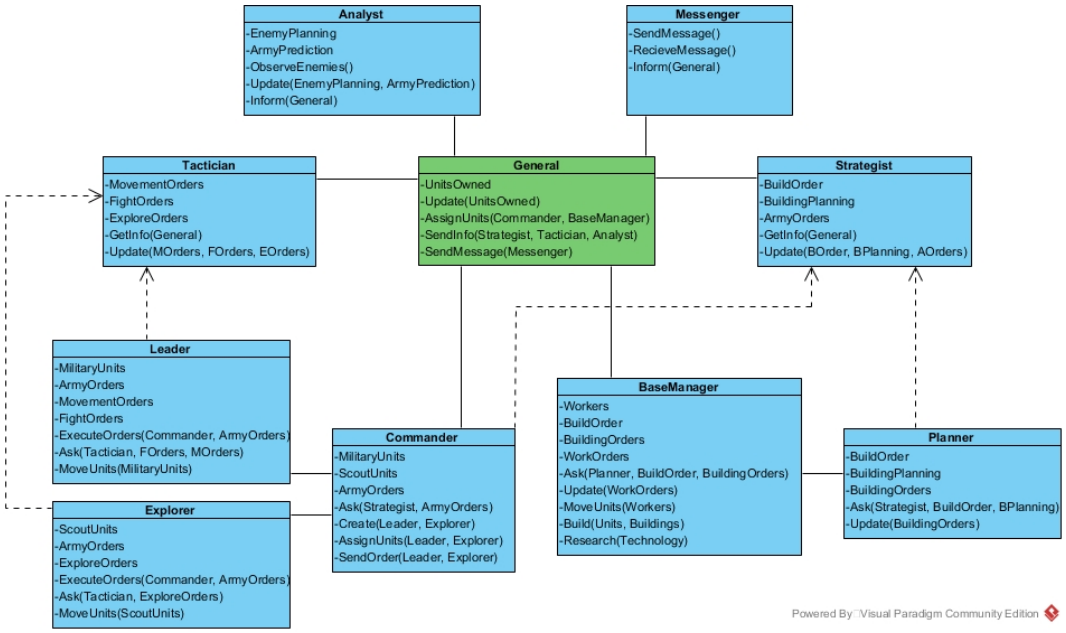


Figura 8: Modelo de Clases.
Fuente: Elaboración propia.

- *SendMessage()* envía información a *Messenger* para ser transmitida a otros jugadores.

- **Messenger**
  - *SendMessage()* y *RecieveMessage()* se encargan de la comunicación con otros jugadores.
  - *Inform()* compila la información relevante recibida y se la transmite a *General*.

- **Analyst**
  - *EnemyPlanning* es la predicción de cuál árbol tecnológico el oponente esta desarrollando.
  - *ArmyPrediction* es la predicción de la posición de las unidades enemigas observadas.
  - *ObserveEnemies()* observa las apariciones de las unidades enemigas y guarda dicha información.
  - *Update()* utiliza algoritmos para modelar el comportamiento de los oponentes y actualiza *EnemyPlanning* y *ArmyPrediction* según la información obtenida.
  - *Inform()* envía la información, *EnemyPlanning* y *ArmyPrediction* a *General*.

- **Commander**

- *MilitaryUnits* corresponde al grupo de unidades de tipo *Military* asignadas por *General*.
- *ScoutUnits* corresponde al grupo de unidades de tipo *Scout* asignadas por *General*.
- *ArmyOrders* corresponde a las posiciones en donde se deben situar los grupos de unidades militares (distintos *Leaders*) en el mapa del juego.
- *Ask()* actualiza *ArmyOrders* desde *Strategist*.
- *Create()* crea *Leaders* o *Explorers* según se necesite.
- *AssignUnits()* asigna las unidades correspondientes a cada *Leader* y *Explorer*.
- *SendOrder()* envía las posiciones (*ArmyOrders*) que cada *Leader* y *Explorer* debe tomar.

- **Leader**
  - *MilitaryUnits* corresponde al grupo de unidades asignadas.
  - *ArmyOrders* corresponde a la posición general del mapa en donde se deben posicionar las unidades asignadas.
  - *MovementOrders* corresponde al movimiento general del grupo de unidades de *MilitaryUnits*.
  - *FightOrders* corresponde a las instrucciones para calcular los movimientos y decisiones de cada unidad asignada (perteneciente a *MilitaryUnits*).
  - *ExecuteOrders()* pregunta a *Commander* para actualizar *ArmyOrders*.
  - *Ask()* pregunta a *Tactician* para actualizar *MovementOrders* y *FightOrders*.
  - *MoveUnits* mueve las unidades siguiendo las instrucciones de *ArmyOrders*, *MovementOrders* y *FightOrders*.

- **Explorer**
  - *ScoutUnits* corresponde al grupo de unidades de exploración asignadas.
  - *ArmyOrders* corresponde a la posición general en donde se debe explorar.
  - *ExploreOrders* corresponde a las instrucciones para explorar la zona correspondiente.
  - *ExecuteOrders()* pregunta a *Commander* para actualizar *ArmyOrders*.
  - *Ask()* pregunta a *Tactician* para actualizar *ExploreOrders*.
  - *MoveUnits* mueve las unidades siguiendo las instrucciones de *ArmyOrders* y *ExploreOrders*.

- **BaseManager**
  - *Workers* corresponde al grupo de unidades tipo trabajador asignadas.

- *BuildOrder* detalla el orden en que se construirán las unidades y las estructuras.
- *BuildingOrders* detalla las posiciones en donde se deben construir las estructuras.
- *WorkOrders* detalla las órdenes a seguir por las unidades de *Workers* asignadas para lograr los objetivos de *BuildOrder* y *BuildingOrders*.
- *Ask()* consulta a *Planner* para actualizar *BuildOrder* y *BuildingOrders*.
- *Update()* utiliza algoritmos de manejo de recursos para optimizar las decisiones y actualiza *WorkOrders* según los planes de *BuildOrder* y *BuildingPlanning*.
- *MoveUnits()* mueve y asigna tareas a los *Workers* siguiendo las instrucciones de *WorkOrders*.
- *Build()* construye estructuras y recluta unidades según corresponda.
- *Research()* desarrolla el árbol de técnologías según corresponda.

- **Strategist**
  - *BuildOrder* representa el orden en que se deben construir las estructuras, reclutar unidades y desarrollar tecnologías.
  - *BuildingPlanning* representa el posicionamiento general de la planificación de la base.
  - *ArmyOrders* representa las posiciones en las que se deben mover las unidades de tipo *Militar* y *Scout*.
  - *GetInfo()* recibe actualizaciones de información desde *General*.
  - *Update()* utiliza algoritmos para actualizar *BuildOrder*, *BuildingPlanning* y *ArmyOrders* dependiendo de la información recibida.
- **Tactician**
  - *MovementOrders* representa los movimientos generales que debe tomar un grupo de unidades.
  - *FightOrders* representa los movimientos y acciones particulares que utilizarán cada unidad en los grupos controlados por los *Leaders*.
  - *ExploreOrders* representa los movimientos y acciones que utilizarán los *Explorers* para mover y explorar.
  - *GetInfo()* recibe actualizaciones de información desde *General*.
  - *Update()* utiliza algoritmos para actualizar *FightOrders* y *ExploreOrders* dependiendo de la información recibida.
- **Planner**
  - *BuildOrder* representa el orden en que se deben construir las estructuras, reclutar unidades y desarrollar tecnologías.
  - *BuildingPlanning* representa las posiciones generales donde irán las estructuras.

- *BuildingOrders* representa las posiciones particulares en donde se deben construir cada estructura.
- *Ask()* consulta información a *Strategist*.
- *Update()* utiliza algoritmos de manejo de recursos y planificación para actualizar *BuildingOrders* dependiendo de la información recibida.

Las entidades utilizan distintos tipos de algoritmos. Más precisamente *Analyst* utiliza algoritmos de modelado de oponentes para calcular *EnemyPlanning* y algoritmos de razonamiento espacial y temporal para calcular *ArmyPrediction*. Por otro lado, *Planner* utiliza algoritmos de manejo de recursos y de planificación espacial para generar el mapa de posiciones de *BuildingOrders*. Además, *BaseManager* utiliza algoritmos de manejo de recursos para calcular las acciones a realizar en *WorkOrders*. Finalmente, el grueso del sistema se realiza en *Tactician* y *Strategist*, que utilizan distintos algoritmos para calcular las decisiones tácticas y estratégicas (a micro y macro control), siendo sus representaciones: *MovementOrders* (órdenes de movimiento), *FightOrders* (órdenes de combate) y *ExploreOrders* (órdenes de exploración), y *BuildOrder* (orden de construcción), *BuildingPlanning* (planificación de base) y *ArmyOrders* (órdenes de posicionamiento), respectivamente.

4.3. Implementación del modelo

En el ámbito desarrollado en este trabajo se enfocó en la implementación de las partes **General**, **Strategist**, **Tactician**, **Commander**, **Leader**, **BaseManager** y **Planner**. Las entidades restantes: **Messenger**, **Analyst** y **Explorer** se dejan como desarrollo para trabajo futuro.

La elección va acorde a investigar y desarrollar principalmente los ámbitos de:

- **Planificación en tiempo real**
- **Toma de decisiones ante adversidad**

Pero, además abarcan levemente las áreas de **Modelado de oponentes**, **Razonamiento espacial y temporal**, **Manejo de recursos** y **Pathfinding**, debido a que se necesita aunque sea un mínimo desarrollo de ellas para que el sistema funcione. Con respecto a la área restante, **Colaboración**, esta se omite ya que se evaluarán solamente partidas y mapas de 1 versus 1 y se omitirá la comunicación entre oponentes.

En la figura 9 se puede observar cómo las partes del sistema atacan cada área de investigación.

El sistema desarrollado posee primordialmente tres partes fundamentales para atacar los problemas de Planificación en tiempo real y de Toma de decisiones ante adversidad. Estas

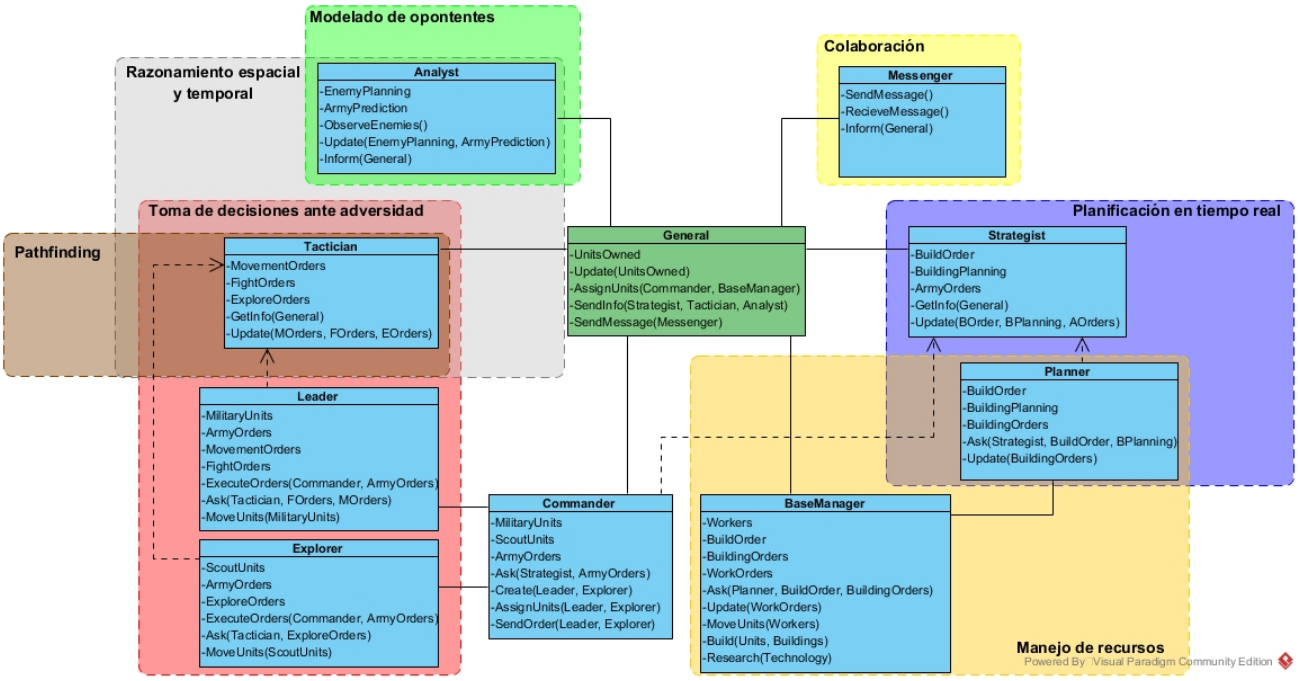


Figura 9: Clasificación de clases en áreas de investigación.
Fuente: Elaboración propia.

partes toman aspectos estratégicos o tácticos, y de micro o macro control. Dichas partes corresponden a:

1. La creación de un *Military Goal* u **Objetivo** militar de una estrategia, el cual corresponde al conjunto de unidades que el bot intentará entrenar para combatir al oponente. Ver Tabla 2.

Tabla 2: Ejemplo de *Goal* u Objetivo para la raza *Zerg*

Tipo de unidad	Cantidad
Zerling	12
Hydralisk	0
Mutalisk	5
Ultralisk	2

2. La creación de un *Build Order* u **Orden de Construcción** (directamente mencionado en la sección 4.1.1), el cual dicta una secuencia de unidades, estructuras o tecnologías que serán entrenadas, construidas o desarrolladas en orden. Ver Figura 10

3. La implementación de *Potential Fields* o Campos Potenciales que funcionarán como tácticas, y que guiarán los movimientos de las unidades de combate. Específicamente

Figura 10: Ejemplo de un Orden de Construcción para la raza *Terran*

SCV	SCV	SCV	Barracks	Marine	SCV	Marine	Factory	SupplyDepot	...	Tank

se utiliza una implementación similar a la utilizada en [Louis and Liu, 2018]. La cual es explicada en más detalle en la sección 4.3. Estas estructuras corresponden a las órdenes de posicionamiento, órdenes de movimiento y órdenes de combate mencionadas anteriormente en la sección 4.1.1.

4.3.1. Flujo del sistema

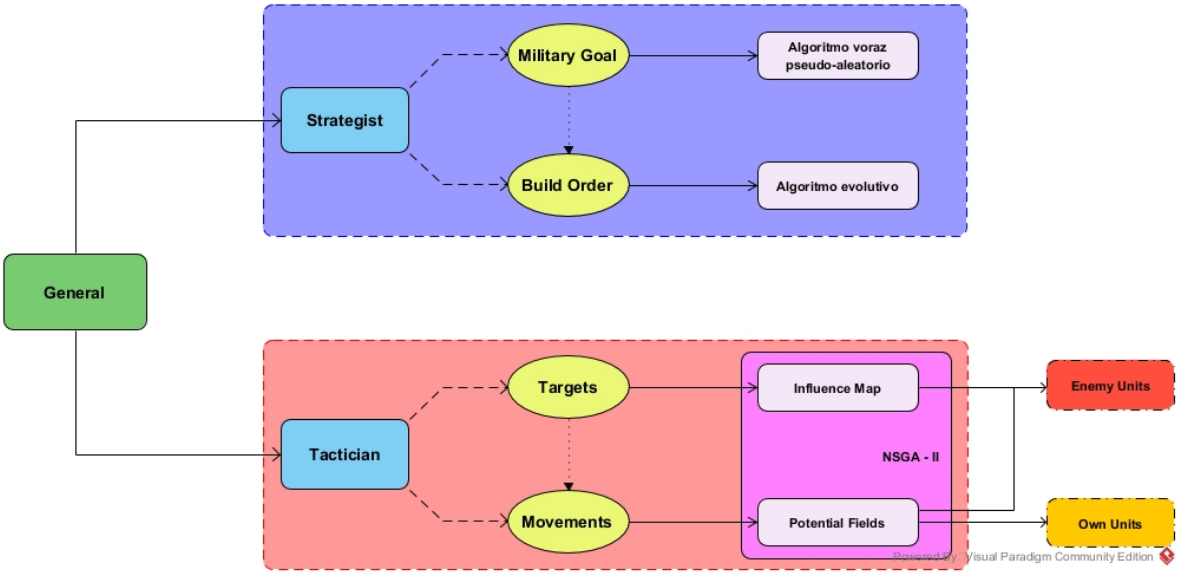


Figura 11: Flujo del sistema.
Fuente: Elaboración propia.

El sistema genera ciertas estructuras para encargarse de los aspectos estratégicos y tácticos. Estas estructuras son generadas por algoritmos utilizados por las entidades Strategist y Tactician. En la figura 11 se puede apreciar las áreas de Planificación en tiempo real como la zona azul y Toma de decisiones ante adversidad en rojo.

Strategist utiliza dos estructuras, *Military Goal* u objetivo, el cual es construido con un algoritmo voraz. Este sirve como entrada para *Build Order* o estrategia, en donde se aplica un pre-procesamiento para tratar los prerrequisitos de las unidades y luego se utiliza un algoritmo evolutivo para buscar una solución óptima. La estrategia final obtenida es usada por *Planner* y *BaseManager* para construir la base y entrenar unidades.

Tactician por su parte maneja dos estructuras, *Targets* y *Movements*. Para la primera se calcula un mapa de influencias sobre las celdas. Dichas celdas corresponden a una grilla sobre el mapa del juego, ver figura 12, y sus valores se calculan dependiendo de las unidades enemigas. Estos valores pasan como entrada a los campos potenciales junto a información de las unidades enemigas y unidades propias. Los campos potenciales se traducen a *Movements* que dicta el movimiento de las unidades. Los campos potenciales y el mapa de influencias tienen parámetros que se optimizan previamente con un algoritmo genético multi-objetivo (NSGA-II). Estos campos potenciales son utilizados posteriormente por Leaders para tomar las decisiones de movimiento de las unidades militares. Las fuerzas de los campos potenciales se pueden ver ejemplificados en la figura 13 en donde la unidad del centro escoge su movimiento dependiendo de los campos generados por el resto de unidades.

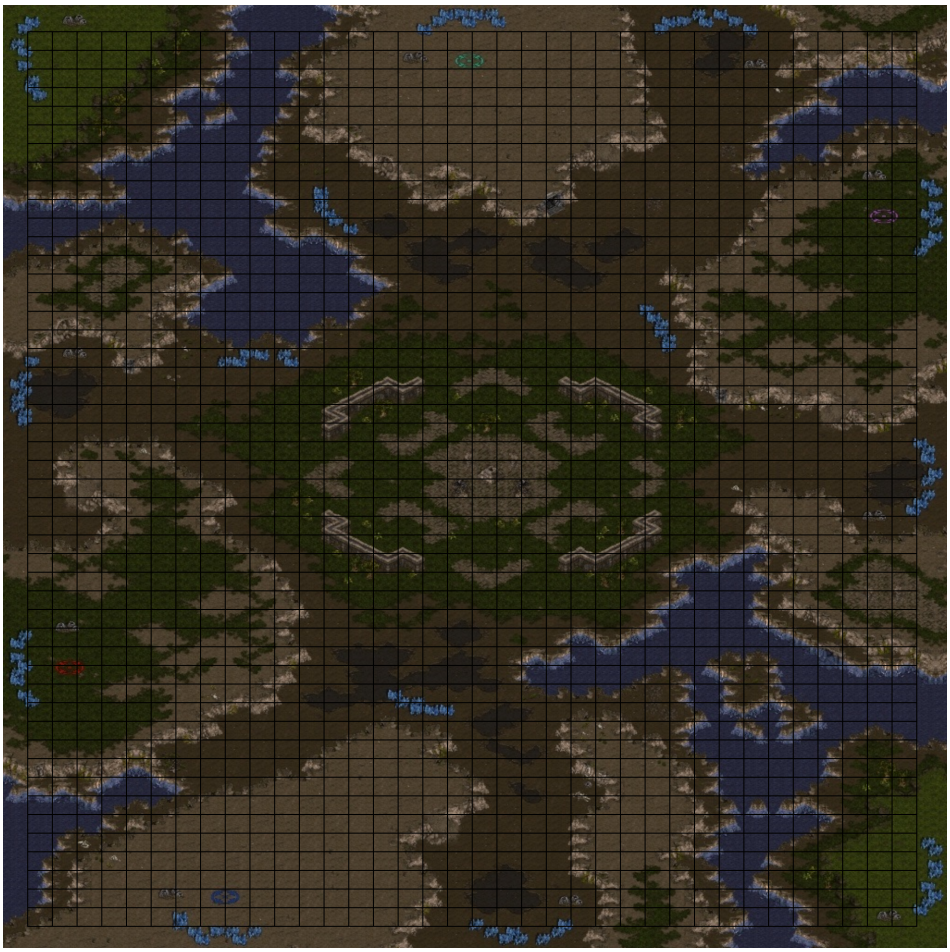


Figura 12: Ejemplo de grilla de celdas en el mapa del juego.
Fuente: Elaboración propia.

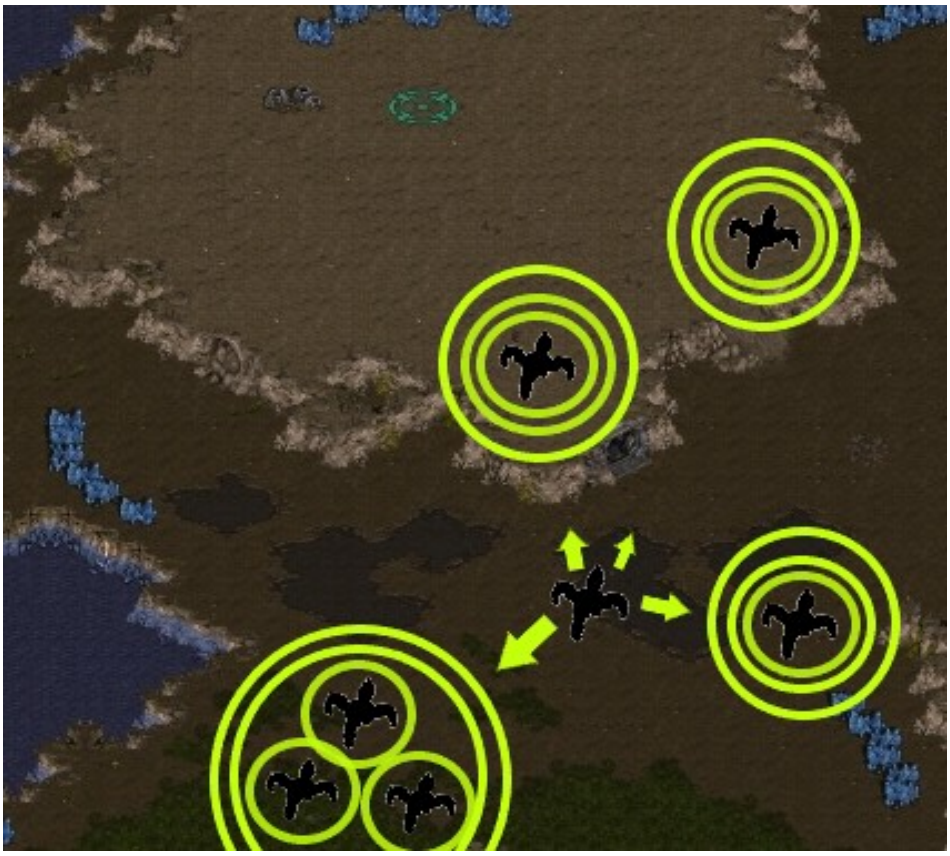


Figura 13: Ejemplo de campos potenciales afectando una unidad en particular.
Fuente: Elaboración propia.

4.3.2. Ambiente de desarrollo

El bot será desarrollado para ser capaz de jugar partidas en *Starcraft*, cuyas características son mencionadas en la sección 1.1 y abordadas más a fondo en la sección 2.2.

El juego posee tres razas distintas: *Zerg*, *Protoss* y *Terran*, las cuales poseen características distintas en su estilo de juego y tanto unidades como estructuras y tecnologías distintas. Los recursos utilizables son comunes a las tres razas, dichos recursos corresponden a recursos naturales: *Mineral* y *Gas Vespeno*, y a recursos de población como lo es la cantidad de *población soportada* (Soporte de población). Cada unidad, estructura y tecnología dentro del juego tiene un costo de Mineral y Gas, pero las unidades además tienen un costo de soporte.

En *Starcraft* una de las características más importante y que entrega mayor dificultad es la presencia del Mapa negro y de la Niebla de guerra, que impiden el conocimiento total del terreno sin previa exploración y la información correspondiente a la cantidad y posición de las unidades enemigas.

Para la implementación se utilizó la API desarrollada para *Starcraft*: **BWAPI**[23], la cual permite obtener una gran cantidad de datos, información y el estado mismo del juego en cada fotograma. Esta API permite el desarrollo de un bot en C/C++, Java y Python. Posteriormente el bot desarrollado se inyecta en el juego mediante el uso de *ChaosLauncher* el cual es un software que permite abrir una partida de *Starcraft* pero habilitando la opción de inyectar cualquier elemento extra al juego.

4.3.3. Enfoque Estratégico y Táctico

Primeramente hay que destacar que hay un aspecto estratégico inherente de los juegos RTS, este es que el objetivo final de una partida es ganar, ya sea derrotando al oponente o logrando objetivos específicos del mapa. Esta característica innata se debe considerar que es la guía final para cualquier estrategia o táctica a considerar.

El aspecto estratégico del bot se abarca en el Objetivo y el Orden de construcción declarados. El objetivo cubre netamente la parte macro de la estrategia, ya que dicta el conjunto de unidades que serán nuestra mayor herramienta para derrotar al oponente. El orden de construcción complementa el objetivo en el sentido que se basa en los pasos a seguir para llegar a tal punto, por lo mismo este también corresponde a un aspecto macro de la estrategia.

Como ejemplo de un aspecto micro de la estrategia se puede mencionar el *BuildingPlanning* y *ExploreOrders* mencionados en la sección 4.2, el primero corresponde al posible posicionamiento que se puede dar a las estructuras a construir dadas por el orden de construcción, mientras que el segundo corresponde a las áreas o regiones a explorar. Este aspecto sale del área de aplicación de este trabajo ya que en su mayoría corresponden a las áreas de Manejo de recursos y Modelado de oponentes.

El aspecto táctico del bot se abarca en el Mapa de influencias y los Campos potenciales que se juntan para formar las fuerzas que guiarán los movimientos de las unidades militares. Esta mezcla de fuerzas junta el aspecto macro y micro de la táctica, pero para aclarar un poco estos dos ámbitos, se puede pensar el mapa de influencias como un aspecto macro, ya que guía de forma general hacia donde se deben mover las unidades. Mientras que los campos potenciales corresponden a la micro táctica, ya que mueven más detalladamente las acciones de las unidades al momento del posible combate. Un esquema se puede ver en la figura 14.

4.3.4. Implementación de los algoritmos

- Algoritmo de inicialización aleatoria factible para la creación de un objetivo inicial

[23] `https://bwapi.github.io/`

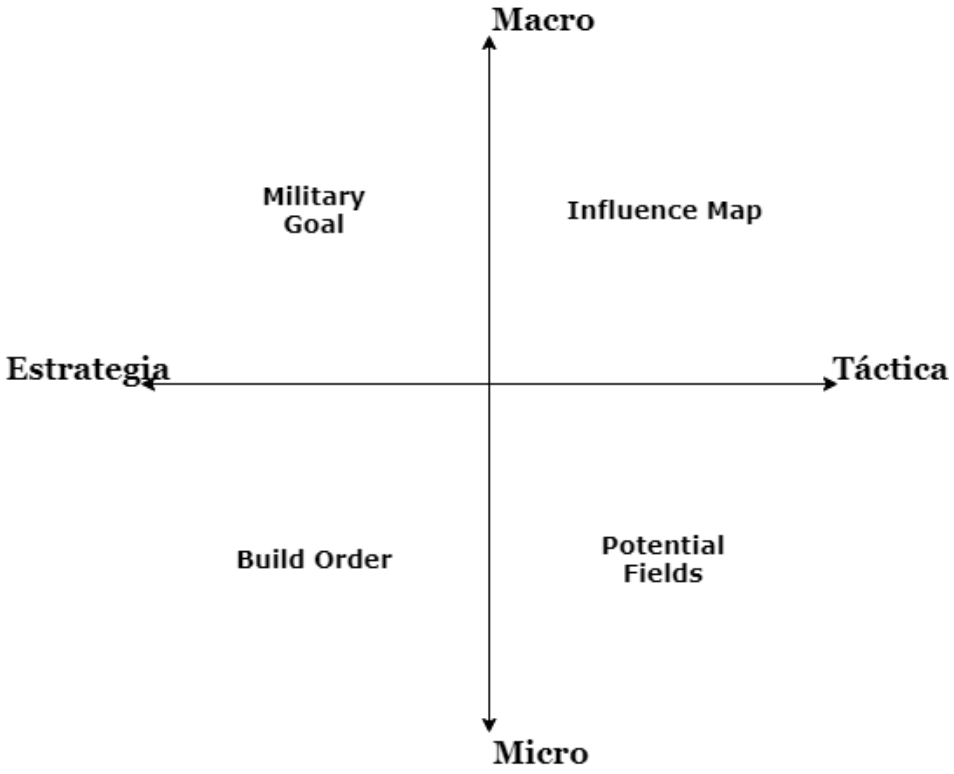


Figura 14: Implementación enfocada en aspectos estratégico-tácticos.
Fuente: Elaboración propia.

Un algoritmo de inicialización aleatoria factible es un algoritmo constructivo que crea una solución seleccionando miembros aleatorios de una lista de candidatos.

Inicialmente se tiene el conjunto de candidatos, un conjunto solución vacío, un conjunto de restricciones y un criterio de término. El algoritmo selecciona aleatoriamente algún candidato, si este no rompe alguna de las restricciones se agrega a la solución. Se repite este procedimiento iterativamente hasta que se cumple el criterio de término.

Primeramente al iniciar una partida no hay mucha información que se pueda obtener del enemigo como para predecir qué estrategia usará ni que movimientos hará para ejecutarla. Por lo mismo, es común que al inicio de una partida se comience con una estrategia definida sin pensar mucho en lo que este haciendo el enemigo. Sin embargo, no hay que olvidar que existen muchas estrategias y movimientos que el enemigo puede realizar que pueden salir de estrategias comunes u ortodoxas, para esto se puede implementar al bot ya sea conocimiento previo o un algoritmo de modelado de oponentes, lo cual esta fuera del área de aplicación de este trabajo.

Para generar una estrategia inicial primero el bot genera un objetivo inicial, el cual corresponde a un grupo de unidades que intentará crear para combatir, este objetivo luego guiará la creación de la estrategia inicial, proceso que será descrito más abajo.

La estructura de un objetivo toma la forma de un conjunto de pares de unidad-cantidad,

un ejemplo de esto se puede ver en la tabla 3.

Para crear este objetivo inicial se utiliza un algoritmo de inicialización aleatorio factible que toma la lista de unidades de combate disponibles e iterativamente agrega una unidad aleatoria al conjunto de unidades. El algoritmo comienza con un conjunto de unidades vacío y con una cantidad de recursos de Soporte de Población (mencionado en la sección 4.3.2). El algoritmo va seleccionando unidades aleatoriamente mientras va consumiendo el soporte de población disponible, cuando este se acaba, entonces las unidades del conjunto se cuentan y se resumen en un conjunto de pares unidad-cantidad como los mostrados en la tabla 3. El algoritmo voraz tiene como conjunto de

Tabla 3: Ejemplo de *Goal* u Objetivo para la raza *Terran*

Tipo de unidad	Cantidad
Marine	12
Medic	4
Firebat	2
Ghost	0
Vulture	0
Siege Tank	4
Goliath	0
Wraith	5
Battlecruiser	2

candidatos los tipos de unidades militares disponibles.

- Algoritmo voraz pseudo-aleatorio para actualización del objetivo

  Un algoritmo voraz es un algoritmo de búsqueda que sigue una heurística que consiste en elegir la mejor opción en cada paso. Es un algoritmo iterativo que construye paso a paso una solución.

  El algoritmo consta de:

  - Lista de candidatos disponibles
  - Conjunto inicial vacío
  - Función miope o de selección
  - Función objetivo

  La lista de candidatos es el conjunto de posibles elecciones en cada paso. El conjunto inicial representa la solución construida, a esta se le agrega un candidato iterativamente. La función de selección o función miope selecciona el candidato más óptimo a cada paso. Y la función objetivo, que evalúa la solución y es la que busca minimizar o maximizar.

  Un algoritmo voraz pseudo-aleatorio se diferencia de un algoritmo voraz en que al momento de elegir un candidato no elige directamente el mejor, sino que tiene una

lista de posibles candidatos, la cual suele ser de menor tamaño que la lista total de candidatos, y elige un candidato aleatoriamente de estos posibles.

Cuando la partida lleva más tiempo, el oponente tiene la capacidad de empezar a crear estructuras y unidades, lo que refleja su estrategia. Esta estrategia se estima de las unidades militares entrenadas por el oponente. Utilizando esta información se busca crear un objetivo y estrategia que logren contrarrestar dichas unidades.

Primeramente, el algoritmo necesita ciertos parámetros que contrasten las distintas fuerzas de cada unidad y que represente de cierta manera el poder de una unidad frente a otra. Para esto se crea una matriz de valores entre unidades, la que se llamará *Matriz de Unidad versus Unidad*.

- Matriz de Unidad versus Unidad

  La matriz se define como la matriz M cuadrada simétrica de $n \times n$, en donde cada fila i y columna j representa los valores de cierta unidad frente al resto de unidades. El valor v_{ij} representa el valor de poder de la unidad i frente a la unidad j.

  Las unidades dentro de *Starcraft* pueden ser de tipo Volador o tipo Terrestre, por lo que a su vez las unidades de combate tienen valores ofensivos hacia Tierra y hacia Aire. Por esto, el valor de la casilla v_{ij} se debe calcular dependiendo de los tipos (volador o terrestre) de la unidad i y de la unidad j.

  Para calcular un valor representativo del poder de una unidad i de tipo t_i sobre otra unidad j, se utiliza el 'daño porcentual infligido en el tiempo': $\frac{d_i^{t_i}}{w_i^{t_i} h_j}$, siendo $d_i^{t_i}$ el daño del arma de la unidad i de tipo t_i, $w_i^{t_i}$ el tiempo de enfriamiento o de recarga de dicha arma, y h_j los puntos de vida máximos de la unidad enemiga j. Estos parámetros se consideran con el objetivo de no sesgar hacia unidades de alto daño pero lenta recarga y de considerar dicho daño con respecto a la vida de la unidad oponente.

  Para evitar que los valores beneficien a unidades propias con mucho daño y pocos puntos de vida, adicionalmente se considera el daño porcentual en el tiempo infligido por el enemigo j sobre la unidad i: $\frac{d_j^{t_j}}{w_j^{t_j} h_i}$.

  Además, se considera el coste de soporte de población de la unidad i para evitar que el algoritmo no tienda a elegir siempre unidades con mucho valor de poder pero que son costosas y lentas de investigar y/o entrenar. Para esto se define $supply_i$ como la cantidad de soporte de población necesario para entrenar i.

  Finalmente, se define el valor de poder de una unidad i de tipo t_i frente a una unidad j de tipo t_j como:

$$v_{ij} = \frac{d_i^{t_i}}{supply_i \cdot w_i^{t_i} \cdot h_j} - \frac{d_j^{t_j}}{supply_j \cdot w_j^{t_j} \cdot h_i} \tag{1}$$

Por otro lado, se considera que algunas unidades tienen la característica de ser Invisibles o Detectoras. Las unidades invisibles no pueden ser vistas normalmente, mientras que las unidades detectores permiten ver a las unidades invisibles dentro de cierto radio. Este nuevo valor I_{ij} corresponde al máximo valor v_{ij} de la matriz si i es detector y j invisible. El valor I_{ij} entonces se define como:

$$I_{ij} = \begin{cases} max\{V\} & \text{si } i \text{ detector y } j \text{ invisible} \\ min\{V\} & \text{en otro caso} \end{cases} \quad (2)$$

Finalmente, el valor de la casilla v_{ij} de la matriz de unidad versus unidad se define como:

$$v_{ij} = \begin{cases} I_{ij} & \text{si } i \text{ detector} \\ v_{ij} & \text{en otro caso} \end{cases} \quad (3)$$

El algoritmo considera la lista de unidades del oponente y la contrasta con la matriz de unidades versus unidades. El algoritmo utiliza una lista de largo $|U|$, siendo U el conjunto de unidades de combate de la raza del jugador (bot). Luego, recorre la lista de unidades del oponente E y por cada unidad e de dicha lista suma los valores v_{eu} de la matriz M tales que $u \in U$. Ver Algoritmo 1.

Algorithm 1 enemies-threat-vector(M, U, E)

```
S = |U|
for all e ∈ E do
    for i = 1 to |M| do
        if i ∈ U then
            S ← S + M_e
return S
```

Luego, la función miope del algoritmo voraz ordena la lista U en orden descendente y crea un subconjunto de candidatos de largo $lc = \frac{|U|}{2}$. De este grupo de candidatos selecciona uno aleatoriamente y lo agrega a la solución S.

Como restricción se utiliza una cantidad de soporte de población máxima $Supply_{max}$. El algoritmo puede agregar unidades a la solución S mientras la cantidad de soporte de población de la solución no sobrepase $Supply_{max}$. Se define el soporte de población de una unidad i como $supply_i$. Entonces el algoritmo en cada iteración busca seleccionar unidades pseudo-aleatorias que contrarresten las unidades actuales del enemigo mientras consume el soporte de población disponible. Ver Algoritmo 2.

La solución construida por el algoritmo voraz corresponde al nuevo conjunto de unidades militares que el bot utilizará como objetivo.

- Algoritmo evolutivo para la creación de un orden de construcción

Algorithm 2 create-goal(U)

1: $S = \emptyset$
2: **repeat**
3: $\quad c = random()\ \%lc$
4: $\quad S \leftarrow S \cup U[c]$
5: $\quad Supply_{max} = Supply_{max} - supply_i$
6: **until** $Supply_{max} < 0$
7: **return** S

Un algoritmo evolutivo es un algoritmo de búsqueda que toma una población de soluciones y realiza transformaciones sobre la misma para explorar y buscar mejores soluciones.

Un algoritmo evolutivo se compone de:

- Una representación de las soluciones, llamado cromosoma.
- Conjunto de soluciones, llamado población.
- Método de selección de padres.
- Operadores de mutación y cruzamiento.
- Método de selección de la siguiente generación.
- Posible elitismo.
- Función de evaluación.

El algoritmo comienza con una población de cromosomas inicial, a esta se le aplica el método de selección para generar una población de padres. Esta población de padres se le aplica los operadores de mutación y cruzamiento para generar una población de hijos. La población de hijos se evalúa utilizando la función de evaluación y se utiliza la selección de la siguiente generación para elegir a la siguiente generación y repetir el proceso. El algoritmo puede o no tener un método de elitismo que mantiene los mejores individuos para que sobrevivan a través de las generaciones.

Este algoritmo recibe como entrada un objetivo, o conjunto de unidades militares, a entrenar. Calculado previamente por el algoritmo de inicialización o el algoritmo voraz pseudo-aleatorio. Este objetivo se transforma de una tabla de pares unidad-cantidad (Tabla 2 y 3) en una lista secuencial de construcción (ver Figura 10).

Al objetivo de entrada se le realiza un pre-procesamiento correspondiente a la adición del árbol de prerrequisitos de dichas unidades militares. Cada unidad militar del juego tiene un edificio o unidad que es capaz de entrenarla, esta unidad o estructura previa se le llama prerrequisito. Las unidades, estructuras y tecnologías que son prerrequisitos de otras unidades a su vez tienen prerrequisitos, esto permite construir un árbol de prerrequisitos. Dicho árbol de prerrequisitos se agrega al objetivo, se consideran los prerrequisitos como restricciones, tanto del pre-procesamiento como restricciones de la estrategia solución, y se manejan de manera de no romper la secuencialidad

de construcción. A esta nueva lista secuencial de construcción se le eliminan repeticiones de unidades y estructuras debidas a la cantidad del mismo tipo de unidad. Esta lista pasa a ser nuestra primera estrategia básica.

Con esta estrategia inicial iniciamos nuestra población, en la primera generación todos los individuos son iguales (primera estrategia) y se les aplican operadores de mutación a través de las generaciones.

Los operadores de mutación están pensados para no romper las restricciones del árbol de prerrequisitos. Dichos operadores son:

- Insertar trabajador:

  Se agrega una unidad de tipo trabajador en cualquier de la lista.

- Insertar estructura

  Se selecciona una estructura presente en la estrategia y se inserta una estructura del mismo tipo en una posición posterior a la estructura original. Un ejemplo se puede ver en la Figura 15, en donde si se selecciona la estructura *Barracks* de la posición 3, se puede insertar otra *Barracks* en una posición aleatoria entre la posición 4 y el final de la lista.

Figura 15: Operador de mutación: *Insertar estructura*.

1:SCV	2:SCV	**3:Barracks**	4:SCV	5:SCV	6:Marine	7:Factory	...

- Mover entidad de posición

  Se selecciona una entidad, ya sea unidad, estructura o tecnología, y se mueve a una nueva posición aleatoria entre la posición de su prerrequisito más cercano y la entidad más cercana que la necesite como prerrequisito. Un ejemplo se puede ver en la Figura 16. Si se selecciona *Barracks* de la posición 3, esta se puede mover desde la posición 3 hasta la posición 6, ya que *SCV* de la posición 2 es prerrequisito de *Barracks* y a su vez *Marine* de la posición 6 tiene como prerrequisito *Barracks*.

Figura 16: Operador de mutación: *Mover entidad*.

1:SCV	2:SCV	**3:Barracks**	4:SCV	5:SCV	6:Marine	7:Factory	...

De la población inicial de una generación se selecciona la misma población como padres de la siguiente generación. Cada individuo tiene una probabilidad de sufrir una mutación, esta selecciona al azar de entre los operadores de mutación. Luego de que cada individuo pase por este proceso, la población resultante es la población de hijos.

A cada individuo de la población de hijos se le calcula el soporte de población necesario para entrenar todas las unidades del objetivo y los trabajadores agregados, y se agregan las estructuras correspondientes.

Una vez generada la población de hijos, esta se evalúa utilizando la función de evaluación. La función de evaluación F se puede ver en la ecuación 7, y esta compuesta por la función militar M que representa el valor militar de una solución (estrategia) S, calor que corresponde a la suma de los valores militares de las unidades militares u de dicha estrategia aportados en el tiempo. El valor militar de una unidad militar u (unidad militar que proviene inicialmente del objetivo dado) viene dado por su valor de poder hacia las unidades enemigas, dicho valor se calcula tomando los valores de la matriz de unidad versus unidad y sumándolo por cada unidad enemiga. Este proceso de valor de poder fue realizado previamente en el algoritmo voraz y se puede ver en el algoritmo 1. El valor militar de la unidad u decae en el tiempo para promover tener poder militar lo más pronto posible. Para el tiempo t se utiliza la cantidad de fotogramas que deberían transcurrir desde el inicio de la estrategia hasta completar el entrenamiento de la unidad, tomando en consideración tanto la creación de estructuras y unidades previas como la recolección de los recursos para las mismas. El valor militar de una unidad u viene dado por la ecuación 4, en donde y_u corresponde al valor correspondiente a la suma de los valores correspondientes a la unidad u en la matriz de unidad versus unidad, y t_u el tiempo (en frames) en donde se crea la unidad u para la estrategia S. Mientras que la función de puntaje militar M se puede ver en la ecuación 5. El valor k es un factor que aumenta o disminuye el peso del decaimiento en el tiempo.

Por otro lado, se utiliza una evaluación económica E en el tiempo, ecuación 6, para incentivar la creación de trabajadores y el desarrollo tanto de estructuras como de tecnologías. La función $p(t)$ corresponde al poder militar estimado a obtener en el tiempo para una cierta cantidad de trabajadores, $r(w)$ es la cantidad de recursos recolectados por frame por una cantidad w de trabajadores, p y c son el poder y el costo promedio de una unidad, T es el tiempo en donde se termina de construir y desarrollar lo dictado por la estrategia S.

$$m(u) = \int_0^\infty (y_u \cdot e^{-kt_u}) dt \tag{4}$$

$$M(S) = \sum_{u \in S} (y_u \cdot k \cdot e^{-kt_u}) \tag{5}$$

$$\begin{aligned} E(S) &= \int_T^\infty \left(\frac{dp(t)}{dt} (t - T) \cdot e^{-kt} \right) dt \\ E(S) &= r(w) \cdot \frac{p}{c} \cdot \frac{e^{kT}}{k^2} \end{aligned} \tag{6}$$

$$F(S) = M(S) + E(S) \tag{7}$$

Finalmente, la población de hijos se combina con la población de padres y los mejores pasan a la siguiente generación. Este proceso iterativo se repite por una cantidad dada de generaciones.

- Algoritmo para movimiento de unidades militares

  Para calcular los movimientos y acciones de las unidades se utiliza un algoritmo genético multiobjetivo (NSGA-II) que evoluciona un conjunto de campos potenciales.

  El algoritmo genético multiobjetivo NSGA-II [Deb et al., 2002], al igual que el algoritmo genético, utiliza una población de individuos (o soluciones) que evoluciona a través de generaciones. A diferencia de un algoritmo genético normal, en un algoritmo multiobjetivo cada individuo tiene una evaluación por cada objetivo.

  Para mantener el elitismo dentro de la población en un algoritmo genético estándar, las soluciones se ordenan con respecto a su evaluación. En el caso de NSGA-II las soluciones tienen valores por objetivo, por lo que al comparar dos individuos, uno puede ser mejor en cierto objetivo pero peor en otro.

  Para mantener el elitismo en NSGA-II se integra el concepto de *dominancia*. Un individuo domina a otro si para cada objetivo, este tiene un valor menor (o mayor, dependiendo si se busca minimizar o maximizar). Este concepto implica la existencia de un subconjunto de individuos que dominan de alguna forma al resto, pero que no se dominan entre sí. Este subconjunto es llamado *frente*. Si se quitan las soluciones de este frente del conjunto total de individuos, es posible calcular nuevamente un subconjunto de individuos que dominan al resto, ya que las soluciones que los dominaban no están presentes. Esto implica la existencia de distintos frentes dentro de la población, lo que permite ordenarla.

  Para realizar el ordenamiento de los individuos con respecto a los objetivos, o sea generar los distintos frentes, NSGA-II utiliza un algoritmo llamado *Fast nondominated sorting*. Este algoritmo calcula para cada individuo dos entidades. 1) Contador de dominancia n_p, el cual corresponde a la cantidad de soluciones que dominan a la solución p. Y 2) Conjunto de dominados S_p que corresponde al conjunto de soluciones que p domina.

  Todas las soluciones en el primer frente tendrán como contador de dominancia $n_p = 0$. Por cada solución con $n_p = 0$ se visita cada miembro $q \in S_p$ y se reduce su contador de dominancia n_q en 1. Si ocurre que $n_q = 0$, entonces se agrega a una lista separada Q. Estos miembros de la lista Q corresponden al segundo frente. El proceso se repite con los miembros de Q para obtener el tercer frente, y así hasta que todos los frentes han sido identificados. El pseudocódigo de este algoritmo se puede ver en el Algoritmo 3

  Con los frentes calculados, NSGA-II selecciona una población de padres y realiza los operadores de mutación y cruzamiento para generar una población de hijos. Estas dos poblaciones se juntan en un conjunto de individuos y se utiliza *Fast Nondominated Sorting* para calcular sus frentes y ordenarlos. De esta nueva población combinada y ordenada (población del doble de tamaño de la original) se seleccionan los individuos que pasarán a la siguiente generación. NSGA-II busca mantener la mayor cantidad de diversidad entre los individuos, por lo que al momento de seleccionar estos, se presenta un concepto de *densidad*, que corresponde a una métrica de densidad de soluciones

Algorithm 3 fast-nondominated-sort(P)

1: **for all** $p \in P$ **do**
2: $S_p = \emptyset$
3: $n_p = 0$
4: **for all** q ∈P **do**
5: **if** $p < q$ **then**
6: $S_p = S_p \cup \{q\}$
7: **else if** $q < p$ **then**
8: $n_p = n_p + 1$
9: **if** $n_p = 0$ **then**
10: $p_{rank} = 1$
11: $F_1 = F_1 \cup \{p\}$
12: $i = 1$
13: **while** $F_i \neq \emptyset$ **do**
14: $Q = \emptyset$
15: **for all** $p \in F_i$ **do**
16: **for all** $q \in S_p$ **do**
17: $n_q = n_q - 1$
18: **if** $n_q = 0$ **then**
19: $q_{rank} = i + 1$
20: $Q = Q \cup \{q\}$
21: $i = i + 1$
22: $F_i = Q$

alrededor de una solución en particular. Esta densidad se calcula en factor a las evaluaciones de cada objetivo.

Para estimar la densidad de las soluciones alrededor de una solución particular se calcula la distancia promedio de dos puntos, a cualquier lado de la solución, a lo largo de cada uno de los objetivos. Esta cantidad $i_{distance}$ sirve como un estimado del perímetro del cuboide que forman las soluciones vecinas más cercanas al individuo en particular. Esta distancia se le llama distancia de hacinamiento.

Para calcular la distancia de hacinamiento primeramente es necesario ordenar los individuos en orden ascendente para cada objetivo. Para las soluciones con mayor y menor valores en cada objetivo se les asigna una distancia infinita. El resto de soluciones intermedias se les asigna un valor correspondiente a la diferencia normalizada absoluta de los valores de las soluciones adyacentes para cierto objetivo. Este cálculo se repite para todos los objetivos. Este algoritmo se denomina *Crowding distance assignment* y se puede ver el pseudocódigo en Algoritmo 4

Algorithm 4 crowding-distance-assignment(I)

```
l = |I|
for all i do
    I[i]_distance = 0
for all m ∈ M do
    I = sort(I, m)
    I[1] = I[l] = ∞
    for i = 2 to (l − 1) do
        I[i]_distance = I[i]_distance + (I[i + 1].m − I[i − 1].m)/(f_m^max − f_m^min)
```

Siendo M la lista de objetivos y f_m^{max} y f_m^{min} valores máximos y mínimos del objetivo m.

El algoritmo NSGA-II entonces para cada generación selecciona una población de padres desde la generación actual, realiza operadores de cruzamiento y mutación, evalúa esta nueva población de hijos, junta la población de hijos con la generación anterior y a esta combinación la ordena y selecciona los miembros de la siguiente generación, repitiendo el proceso hasta que se cumplan los criterios de término. Esta iteración se puede ver en el Algoritmo 4.

Para comparar el orden de dos soluciones se define el orden parcial $\leq_n$ como 'i es menor que j si el frente de i es menor al frente de j o si i y j pertenecen al mismo frente pero $i_{distance}$ es mayor a $j_{distance}$'. Esto se puede ver en el Algoritmo 6.

Finalmente el proceso completo de selección de una nueva generación se puede ver en la Figura 17.

Para el movimiento de las unidades militares se utiliza calcula un mapa de influencias y una red de campos potenciales que guían la dirección de que seguirán las unidades. Estas dos estructuras utilizan ciertos parámetros que se optimizan utilizando una Algoritmo genético multiobjetivo (NSGA-II).

Algorithm 5 main-loop

1: $R_t = P_t \cup Q_t$
2: $F =$ fast-nondominated-sort(R)
3: $P_{t+1} = \emptyset$
4: $i = 1$
5: **repeat**
6: crowding-distance-assignment(F_i)
7: $P_{t+1} = P_{t+1} \cup F_i$
8: $i = i + 1$
9: **until** $|P_{t+1}| + |F_i| \leq N$
10: Sort($F_i, \leq_n$)
11: $P_{t+1} = P_{t+1} \cup F_i[1 : (N - |P_{t+1}|)]$
12: $Q_{t+1} =$ make-new-population(P_{t+1})
13: $t = t + 1$

Algorithm 6 crowded-comparison operator

1: **if** $(i_{rank} < j_{rank})$ or $((i_{rank} = j_{rank})$ and $(i_{distance} > j_{distance}))$ **then**
2: **return** $i \leq_n j$

Para construir el mapa de influencias y los campos potenciales se toman en consideración la posición de las unidades enemigas en el mapa.

- Mapa de influencias

  Un mapa de influencias toma un mapa o terreno y lo divide en distintas zonas o regiones a las que se les calculará un valor llamado influencia.

  El algoritmo para calcular un mapa de influencias se compone de:

  - Una representación continua o discreta de cualquier dimensionalidad, llamado mapa.
  - Método de división que separa el mapa en áreas disjuntas, llamadas regiones.
  - Función de influencia inicial.
  - Conjunto de entidades internos o externos a las regiones.
  - Conjunto de restricciones.
  - Función de evaluación.

  El algoritmo primeramente divide el mapa en las distintas regiones. Luego, para cada región se calcula la suma de las influencias de las entidades que afectan sobre ella, respetando las restricciones impuestas, para calcular el valor de influencia total de dicha región. La influencia que emana cada entidad corresponde a un valor de influencia inicial, y esta disminuye con respecto a la distancia de la entidad a la región específica.

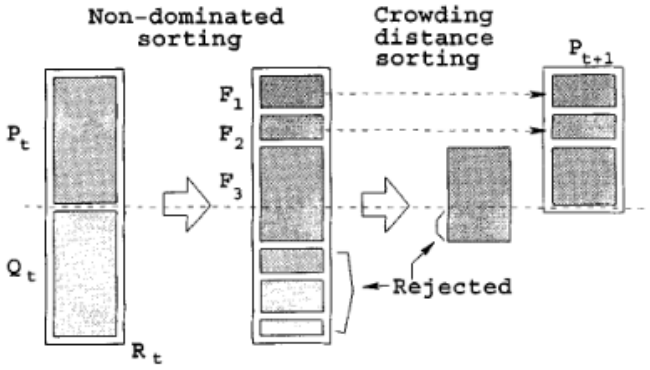


Figura 17: Selección de la siguiente generación
Fuente: 'A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II' [Deb et al., 2002].

La magnitud de una influencia es de la forma:

$$I_t = \sum_{e \in R} (I_o - (d_e(I_o I_f))) \tag{8}$$

Siendo I_t la influencia total de una región. R el conjunto de entidades que cumplen las restricciones. I_o la influencia inicial. d_e la distancia de la entidad a la región. Y $I_o I_f$ el factor de decrecimiento por distancia, el cual es una porción de la influencia inicial.

El mapa de influencias utiliza el mapa del juego como el mapa completo de influencias. Luego, se divide el mapa del juego en una grilla cuadrada. BWAPI permite obtener esta grilla directamente, en donde cada celda es un cuadrado de 32x32 píxeles.

Como entidades se consideran todas las unidades enemigas conocidas y se utiliza como influencia la distancia de dichas unidades hacia cada celda. Como restricción se utiliza un radio r de celdas alrededor de la celda que se quiera calcular su valor de influencia.

El valor de influencia inicial de una celda se compone de una razón porcentual de los puntos de vida y una razón porcentual del tiempo de enfriamiento del arma de todas las unidades en ella.

El valor de influencia inicial se puede ver en la ecuación 9, en donde R_h corresponde a la razón entre los puntos de vida actuales de la unidad y los puntos de vida totales de la misma. R_w corresponde a la razón del tiempo de enfriamiento o tiempo de recarga restante con el tiempo total de enfriamiento. w_1 y w_2 corresponden a valores de peso, los cuales son optimizados por el algoritmo genético.

$$I_o = w_1 R_h + w_2 R_w \tag{9}$$

El valor total de influencia de una celda se puede ver en la ecuación 10, en donde R es el conjunto de unidades en un radio r de la celda. I_o es la influencia inicial, d_u

es la distancia desde la celda a la unidad, y I_f es un factor de decrecimiento por unidad de distancia. r y I_f son parámetros que son optimizados por el algoritmo genético.

$$I_t = \sum_{u \in R} (I_o - (d_u(I_o I_f))) \tag{10}$$

- Campos potenciales

  Los campos potenciales representan fuerzas de atracción y repulsión que sirven para guiar el movimiento y trayectoria de entidades en una zona dada.

  Los campos potenciales se componen de:

  - Una zona donde se mueven las entidades y donde están presentes los campos.
  - Puntos de interés fijos o variables dentro del campo que generan fuerzas.
  - Parámetros que indican la fuerza de cada campo.
  - Fuerzas de atracción y repulsión correspondientes a cada campo.

  Para calcular las fuerzas de los campos potenciales el algoritmo calcula para cada punto de interés su fuerza de atracción y/o repulsión utilizando los parámetros dados. Luego, estas fuerzas interactúan con las entidades que se moverán a través de la zona.

  La fuerza de un campo potencial es de la forma:

  $$F = cd^e \tag{11}$$

  En donde c y e son constantes, mientras que d es la variable que dicta la fuerza, generalmente distancia.

  El algoritmo de campos potenciales utiliza el mapa del juego como mapa en donde estarán presentes los campos potenciales. La posición de los campos en el mapa corresponden a una celda de 1x1 píxeles, dicha celda corresponde a la posición de las unidades del juego, tanto enemigas como del bot.

  El campo potencial total es una suma de cuatro tipos de campos potenciales de la forma $f = cd^e$. El campo potencial que dicta el movimiento de las unidades se puede ver en la ecuación 12, en donde T corresponde al campo potencial dictado por el mapa de influencias, D una suma de campos potenciales dictado por la distancia de las unidades, H una suma de campos potenciales dictado por el porcentaje de puntos de vida restantes de las unidades y W una suma de campos potenciales dictado por el porcentaje de tiempo de enfriamiento restante. Cada campo potencial se calcula para el movimiento de una unidad específica, por lo que cada campo se calcula iterativamente sobre el conjunto total de unidades propias.

  $$F = T + D + H + W \tag{12}$$

Los campos potenciales D, H y W están compuestos por un vector dirección dado por la diferencia normalizada entre el vector posición p de la unidad actual y la posición de otra unidad u. Este vector se puede ver en la ecuación 13.

$$\hat{n} = \frac{u - p}{|u - p|} \tag{13}$$

La suma de campos potenciales D tiene una magnitud dada por la distancia entre la unidad actual y otra unidad u. En la ecuación 14 la primera suma es sobre las unidades propias F, siendo d la magnitud de distancia entre la unidad actual y la unidad u. La segunda suma es sobre las unidades enemigas E. c_1, c_2, c_3, c_4, e_1, e_2, e_3 y e_4 son parámetros constantes que optimiza el algoritmo genético.

$$D = \sum_{u \in F} (\hat{n} c_1 d_u^{e_1} - \hat{n} c_2 d_u^{e_2}) + \sum_{u \in E} (\hat{n} c_3 d_u^{e_3} - \hat{n} c_4 d_u^{e_4}) \tag{14}$$

La suma de campos potenciales H tiene una magnitud dada por el porcentaje de puntos de vida restantes de la unidad u. En la ecuación 15, al igual que en D la primera suma es sobre las unidades propias F mientras que la segunda suma sobre las enemigas E, siendo h el porcentaje de puntos de vida restantes de la unidad u. c_5, c_6, c_7, c_8, e_5, e_6, e_7 y e_8 son parámetros constantes que optimiza el algoritmo genético.

$$H = \sum_{u \in F} (\hat{n} c_5 h_u^{e_5} - \hat{n} c_6 h_u^{e_6}) + \sum_{u \in E} (\hat{n} c_7 h_u^{e_7} - \hat{n} c_8 h_u^{e_8}) \tag{15}$$

La suma de campos potenciales W tiene una magnitud dada por el porcentaje de tiempo de enfriamiento restante del arma de la unidad u. En la ecuación W, al igual que en los dos casos anteriores, la primera suma es sobre unidades propias F y la segunda sobre las unidades enemigas E. w es el porcentaje de tiempo de enfriamiento restante, mientras que c_9, c_{10}, c_{11}, c_{12}, e_9, e_{10}, e_{11} y e_{12} son parámetros constantes que optimiza el algoritmo genético.

$$W = \sum_{u \in F} (\hat{n} c_9 w_u^{e_9} - \hat{n} c_{10} w_u^{e_{10}}) + \sum_{u \in E} (\hat{n} c_{11} w_u^{e_{11}} - \hat{n} c_{12} w_u^{e_{12}}) \tag{16}$$

La suma de campos potenciales T tiene como vector de dirección normalizado $\hat{m}$, el cual es la diferencia entre el vector de dirección de la unidad actual con el vector dirección de las celdas de interés del mapa de influencias. Una celda de interés es una celda del mapa de influencias que contiene unidades enemigas. En la ecuación 17, I es el conjunto de celdas de interés, d_c es la distancia entre la unidad actual y la celda c. c_{13} y e_{13} son parámetros que optimiza el algoritmo genético.

$$T = \sum_{c \in I} \hat{m} c_{13} d_c^{e_{13}} \tag{17}$$

El algoritmo de movimiento utiliza el mapa de influencias para seleccionar celdas de interés, estas celdas identifican posibles áreas objetivos a las cuales mover las unidades y atacar. Este algoritmo se encarga en parte del aspecto macro de la táctica.

Los campos potenciales guían el movimiento de las unidades de manera más fina. Utilizando campos potenciales para cada unidad enemiga y cada unidad propia. Considerando la distancia de las unidades, el porcentaje de vida restante de las unidades y el tiempo de enfriamiento del arma de las unidades.

Los parámetros del mapa de influencias, w_1, w_2, r y I_f, y los parámetros de los campos potenciales, c_1 a c_{13} y e_1 a e_{13}, son optimizados por un algoritmo genético multiobjetivo (NSGA-II).

Para NSGA-II se utiliza como representación un vector binario, en donde cada parámetro se traduce en un número de bits de dicho vector.

- Para los parámetros w_1, w_2 y I_f se utilizan 5 bits para cada uno, lo que denota valores entre $[0, 31]$ y se traducen al rango $[0, 1]$.
- Para el parámetro de radio r, se utilizan 3 bits, lo que da un rango de radio de $[0, 7]$ celdas.
- Para los parámetros c_1 a c_{13} se utilizan 12 bits para cada uno, lo que entrega valores del rango $[0, 4095]$ y se traducen al rango $[-10239, 10240]$.
- Para los parámetros e_1 a e_{13} se utilizan 4 bits para cada uno, lo que se traduce a valores del rango $[-7, 8]$

Los parámetros del mapa de influencias utilizan 18 bits, mientras los de los campos potenciales ocupan 208 bits. En total la representación es un vector binario de largo $18 + 208 = 226$ bits.

El algoritmo utiliza dos objetivos: maximizar el daño infligido y minimizar el daño recibido. El algoritmo intenta maximizar ambos objetivos, por lo que se realiza un ajuste, ver ecuación 18, en donde D_i es el daño infligido por las unidades propias U y D_r es el daño infligido por las unidades enemigas (o recibido por las propias), ambos daños normalizados para ser valores en el rango $[0, 1]$. Para esto utilizamos los puntos de vida porcentuales de las unidades, buscando maximizar los puntos de vida porcentuales restantes al final de la partida de las unidades propias y minimizar los puntos de vida porcentuales restantes de las unidades enemigas. Para esto se calcula los puntos de vida porcentuales restantes de las unidades propias y de las unidades enemigas, ver ecuación 19, en donde U es el conjunto total de unidades propias, tanto vivas como muertas, mientras que E es el conjunto total de unidades enemigas, vivas o muertas. h^c son los puntos de vida actuales al final de la partida y h_t los puntos de vida máximos de la unidad.

$$fitness = [(\sum D_i), (1 - \sum D_r)] \tag{18}$$

$$fitness = \left[\left(\frac{\sum_{u \in U} h_u^c}{\sum_{u \in U} h_u^t} \right), \left(1 - \frac{\sum_{e \in E} h_e^c}{\sum_{e \in E} h_e^t} \right) \right] \tag{19}$$

La población inicial se crea aleatoriamente. En cada generación se se selecciona la población de padres con un torneo binario utilizando los individuos de la generación actual. Se utiliza como operador de cruzamiento una recombinación en dos puntos con probabilidad 0.9, y como operador de mutación un cambio de bit con probabilidad 0.05, intentando imitar y conseguir el buen comportamiento del algoritmo de movimiento de unidades presentado en [Louis and Liu, 2018].

La población de hijos obtenida después del cruzamiento y la mutación se combina con la población actual y se calculan los frentes para ordenar la población combinada para luego escoger los mejores individuos, los cuales pasan a la siguiente generación.

Resumen

En este trabajo se propone un modelo de un sistema con un enfoque en la modularización de las áreas de investigación. En particular se tiene la implementación de varias de las partes del sistema, enfocada en dos áreas principales de la investigación de inteligencia artificial: Planificación en tiempo real y Toma de decisiones ante adversidad, pero abarcando levemente otras áreas necesarias como lo son el modelado de oponentes, razonamiento espacial y temporal, manejo de recursos y pathfinding.

Las dos áreas abordadas son importantes debido a su impacto primordial en los aspectos estratégicos y tácticos del comportamiento del bot. Siendo los algoritmos implementados para éstos enfocados en los ámbitos de micro-macro control estratégico y táctico.

Entonces, la implementación consta de un área estratégica (*Strategist*) que se centra principalmente en la planificación en tiempo real. Utilizando un objetivo militar como macro estrategia y un orden de construcción como micro estrategia. Por su parte, el área táctica (*Tactician*) se centra en la toma de decisiones ante adversidad, utilizando un mapa de influencias como macro táctica y campos potenciales como micro táctica. Ambas áreas con su respectiva optimización: la táctica con una optimización hecha previamente y la estratégica con optimizaciones hechas en tiempo real durante la partida.

CAPÍTULO 6
VALIDACIÓN DE LA SOLUCIÓN

La implementación hecha se desarrolló con la herramienta BWAPI 4.4.0, utilizando como plataforma StarCraft 1.16.1. Mientras que las pruebas se realizaron en un procesador Intel Core i5-6200U.

Las partidas simultaneas corresponden a enfrentamientos de 1vs1 en contra de la inteligencia artificial inherente de StarCraft. En el uso de la API es importante destacar que se habilitó la opción *CompleteMapInformation*. Esta opción permite al bot ignorar la niebla de guerra y el mapa negro, obteniendo virtualmente conocimiento de todo los movimientos de unidades en el mapa. Se utilizó esta opción para suplir el área de Modelado de oponentes, la cual está fuera del área de aplicación de este trabajo. Además, para las pruebas solo se utilizaron dos mapas distintos específicos: *DesertKill* y *Showdown*. Finalmente, se utilizaron las opciones *setLocalSpeed* y *setFrameSkip* para acelerar las partidas ejecutadas en la API y disminuir el tiempo de las simulaciones; cada partida completa dura entre 5 y 10 segundos. En las partidas el bot solamente utilizó la raza *Protoss*.

La pruebas de la implementación se dividen en dos fases. La primera fase de pruebas fue la de optimizar los parámetros del algoritmo de campos potenciales mediante NSGA-II. Para este proceso de utilizó un parámetro $k = 0{,}00035$ como ajuste para el algoritmo de generación de estrategias. Este valor de k se consiguió a base de prueba y error, y su función es la de generar estrategias balanceadas, i.e. estrategias que no estén sesgadas hacia estrategias de early-game (primeros 5-7 minutos de juego) o de late-game (15+ minutos). A su vez se utiliza $lc = 3$ como largo de la lista de candidatos para el algoritmo voraz.

Para el algoritmo NSGA-II se utilizó una población de 50 individuos, a través de 25 generaciones. Cada individuo corresponde a un gen de 226 bits, descrito en la sección 4.3.4. Los operadores genéticos corresponden a un cruzamiento en dos puntos con una probabilidad de 80 % y una mutación de cambio de bit con una probabilidad de 5 %.

En la tabla 4 se puede ver el promedio de las evaluaciones de los individuos de la población en cada generación. La columna correspondiente al valor defensivo corresponde solo a valores nulos, esto representa que durante el entrenamiento se perdieron todas o casi todas las partidas, i.e. el bot no sobrevivió con ninguna unidad, por lo que todos los puntos de vida fueron recibidos como daño.

Se puede ver en la tabla que el bot va mejorando su forma de utilizar la táctica, y con una mayor cantidad de generaciones se pueden obtener resultados prometedores.

Una vez se obtienen mejores parámetros para las tácticas, se pasa a la segunda fase de pruebas, en donde el sistema puede utilizar tácticas optimizadas previamente por él mismo para jugar las partidas. Dentro de esta fase pasa a tomar mayor importancia la generación de

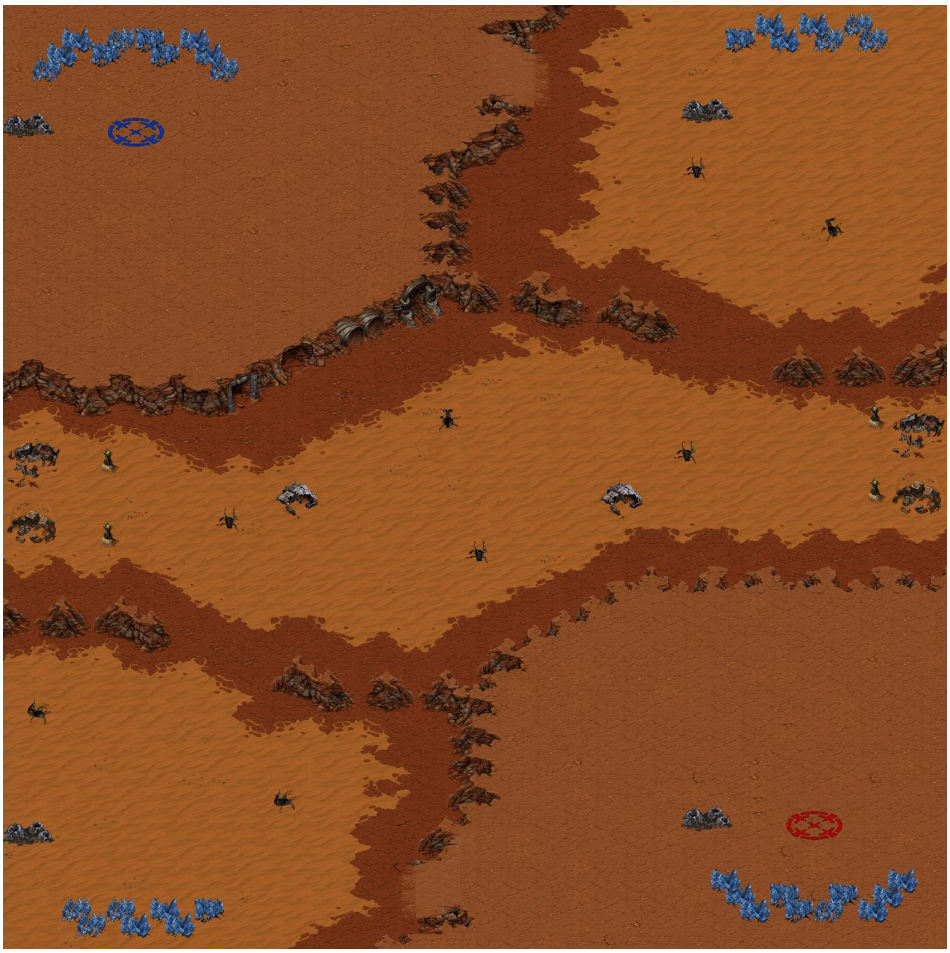


Figura 18: Mapa 'DesertKill'.
Mapa utilizado ampliamente en la optimización de los campos potenciales. La zona superior izquierda e inferior derecha sirven como puntos de inicio de ambos jugadores.

estrategias, y por lo tanto, los parámetros de dicho algoritmo.

El algoritmo de generación de estrategias esta fuertemente sujeto a los parámetros que se le den y a la cantidad de funcionalidades y características que se le entregue al bot. Esto se ve fuertemente reflejado en la tabla 5, en donde al terminar la partida el bot aun tiene una gran cantidad de recursos para gastar, pero que no fueron utilizados eficientemente.

La cantidad de recursos sobrantes menor que se puede observar con un lc más grande corresponde a que el bot divide mucho su atención en intentar entrenar más variedad de unidades, lo que conlleva más recursos utilizados en estructuras, más tiempo en preparación y, finalmente, el bot esta menos preparado para defenderse y pierde en menor tiempo. Por otro lado, la cantidad menor de recursos sobrantes con k mayores representa que la estrategia se enfoca mucho en lo militar, entrenando pocos trabajadores para recolectar recursos mientras que gasta cada mineral en unidades militares. Al contrario, con un k menor, la cantidad

Tabla 4: Evolución de la evaluación de las tácticas

Generación	Evaluación Ofensiva (%)	Evaluación Defensiva (%)
1	1.365	0
2	1.409	0
3	1.418	0
4	2.634	0
5	3.186	0
10	4.069	0
12	5.049	0
15	6.799	0
19	8.160	0
20	8.354	0
25	9.098	0
35	9.588	0

Evolución a través de las generaciones de las evaluaciones defensivas y ofensivas del algoritmo de campos potenciales. La evaluación ofensiva corresponde al porcentaje de vida que se daño al enemigo, mientras que la defensiva es el porcentaje de la vida propia recibida como daño.

Tabla 5: Uso de recursos del juego

k	lc	Recursos
0.01	**3**	112,1
0.005	**3**	114,8
0.001	**3**	108,5
0.0005	**3**	198,5
0.0001	**3**	1776,1
0.00035	1	502,2
0.00035	2	471,6
0.00035	3	451
0.00035	4	412,9
0.00035	5	377,3

Cantidad de recursos promedio sobrantes del bot finalizada la partida. k corresponde al parámetro k de las ecuaciones 5 y 6. Y lc corresponde a lc del algoritmo 2.

de recursos sin utilizar se incrementa en gran manera, esto se debe a que el bot se enfoca mucho en entrenar trabajadores y recolectar recursos, dejando de lado lo militar, lo que lleva a acumular grandes cantidades.

En la figura 19 podemos observar cómo se distribuyen las poblaciones de individuos hijos por generaciones. Los individuos pueden tener evaluaciones muy variadas, habiendo individuos con evaluaciones casi nulas, mientras que otros tienen evaluaciones, relativamente, muy buenas. Esto se puede contrastar con los individuos finales de cada generación (o padres de la siguiente), en donde la calidad de los individuos va mejorando notoriamente a través de las generaciones, con cambios más o menos drásticos dependiendo del comportamiento de la población de hijos.

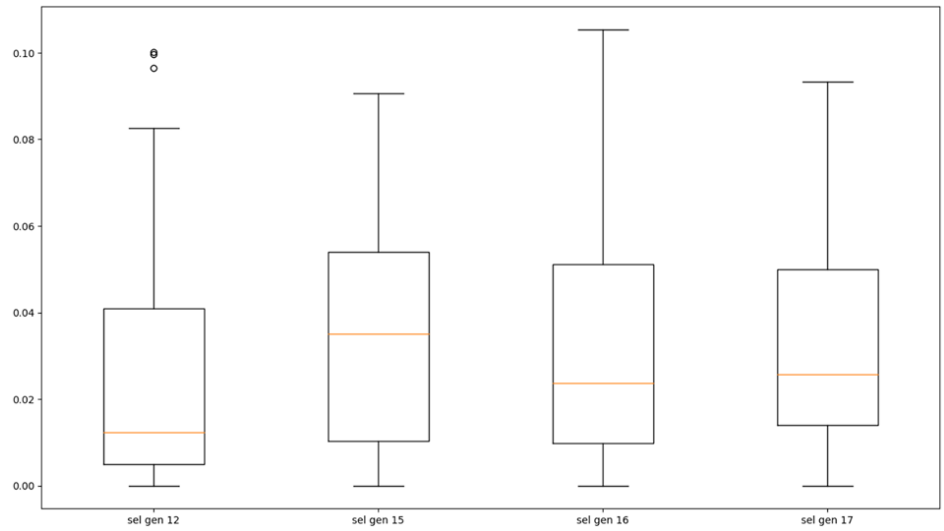


Figura 19: Boxplot. Distribución de individuos (población hijos) en generaciones.
Fuente: Elaboración propia.

En conclusión, aunque los resultados son aun preliminares, descontando los posibles problemas generados por falta de implementaciones suplementarias para mejorar características adyacentes a los algoritmos hechos, el comportamiento del algoritmo de campos potenciales junto a NSGA-II resulta satisfactorio. En la figura 21 se puede apreciar cómo aumenta su eficiencia a través de las generaciones. Por otro lado, el algoritmo generador de estrategias desde el inicio estaba pensado para generar soluciones que no necesariamente sean las más óptimas para cada situación, por lo que su resultado es el esperado. Sin embargo, los mismos resultados muestran que es necesario acompañar tal algoritmo con algún otro algoritmo o característica, y el algoritmo solo en sí no basta para abarcar toda el área de interés de la planificación en tiempo real.

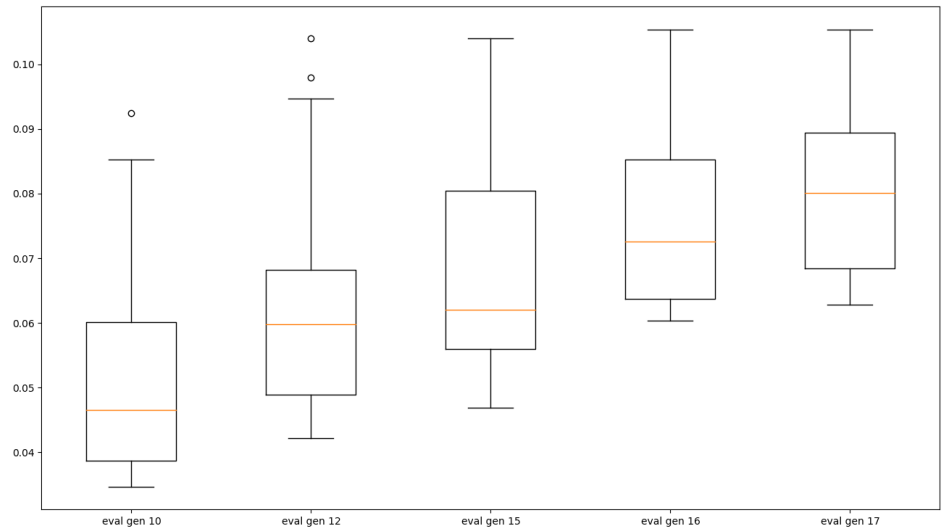


Figura 20: Boxplot. Distribución de individuos finales (o padres) en generaciones
Fuente: Elaboración propia.

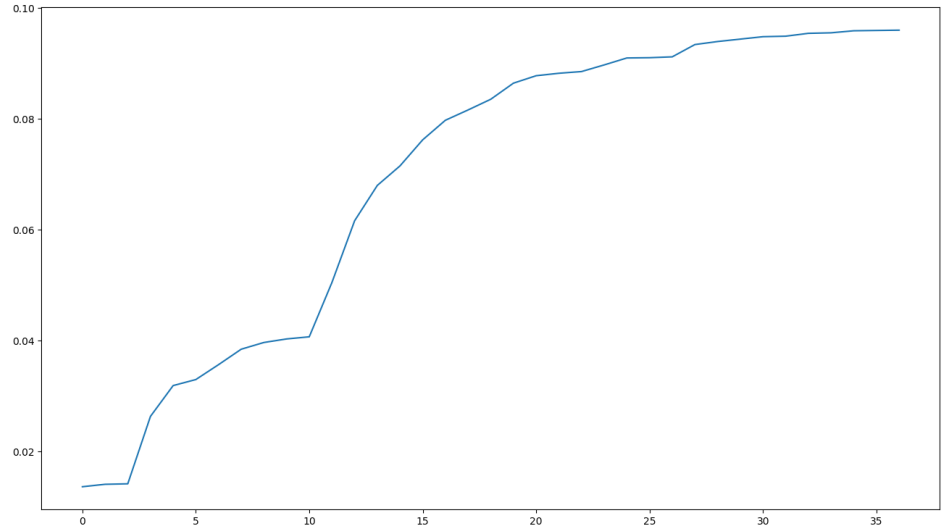


Figura 21: Evaluación de los campos potenciales a través de las generaciones.
Fuente: Elaboración propia.

CAPÍTULO 7
CONCLUSIONES

Los juegos RTS presentan un interesante y complejo ambiente para el desarrollo e investigación de inteligencia artificial. Las características de información parcial, ambiente adversario, movimiento por distintos terrenos y la limitación temporal de toma de decisiones presenta un escenario de gran dificultad tanto para jugadores como para investigadores y para bots.

El hecho de que el ambiente del juego sea de combate militar lleva a las conocidas ideas de las estrategias y tácticas militares, pero enfocándose hacia una simulación computacional. De la misma forma que en el mundo real, podemos modelar los problemas de combate de los juegos RTS dentro de sus aspectos estratégicos y tácticos, a la vez que podemos clasificarlos aun más mediante el control macro y micro que nos permite la esencia del juego en sí.

El trabajo de inteligencia artificial e investigación que se ha desarrollado en juegos RTS nos permite detectar distintas áreas a las cuales enfocar nuestra atención y nuestro trabajo. En este documento se propone un sistema con un enfoque modular (dentro de lo posible), que permite detectar y atacar cada área de manera particular. Específicamente, se presenta una implementación centrada en las áreas de planificación en tiempo real y toma de decisiones ante adversidad. Pero todas estas áreas tienen su esfuerzo e investigación previa por parte de la comunidad. Distintos tipos de herramientas y técnicas se han utilizado para cada una, como lo son branch-and-bound, programación genética y redes neuronales para la planificación en tiempo real, o algoritmos genéticos y de Monte-carlo para la toma de decisiones, etc. En esta investigación son de gran importancia optimización de un orden de construcción de Churchill y Buro [Churchill and Buro, 2011] [Churchill and Buro, 2012], y la optimización de campos potenciales de Luois y Lui [Louis and Liu, 2018]. Ambos sirvieron como base y como inspiración para el trabajo presentado.

En el sistema presentado se destacan las dos áreas ya mencionadas, las cuales se tratan con dos entidades llamadas *Strategist* y *Tactician*. Ambas entidades utilizan distintos algoritmos de construcción y de reparación para generar soluciones respectivas. Por una parte, la parte estratégica utiliza un objetivo militar y un orden de construcción, siendo estos una lista de tipos de unidades a construir y la cantidad de la misma, y una secuencia de unidades, estructuras y tecnologías a crear en un orden especifico, respectivamente. Además se utiliza un algoritmo de inicialización aleatoria factible que genera un objetivo militar aleatorio; un algoritmo voraz pseudo-aleatorio, para actualizar el objetivo militar en pos de contrarrestar las unidades enemigas; y un algoritmo evolutivo, que tiene como entrada el objetivo militar y genera un orden de construcción u estrategia para luego optimizarlo. Por otra parte, la táctica utiliza un mapa de influencias y campos potenciales. El mapa de influencias secciona el mapa del juego y asigna valores a dichas casillas dependiendo de las unidades enemigas. A su vez, los campos potenciales toman parámetros y variables del juego, y las utiliza para darle valores de fuerza a las unidades, tanto enemigas como propias. Los parámetros de los

campos potenciales, incluyendo el campo potencial atribuido al mapa de influencias, son optimizados mediante un algoritmo genético multi-objetivo: NSGA-II. El algoritmo optimiza en una fase previa los parámetros, los cuales pueden ser ocupados posteriormente en cualquier partida.

Los resultados obtenidos, aunque prometedores, se ven afectados por la falta de desarrollo en el resto de áreas de investigación, las cuales aunque están fuera del área de trabajo siguen siendo partes primordiales del sistema, y las implementaciones leves de dichas áreas afectan en mayor o menor medida al rendimiento del bot. Los problemas se pueden ver, por ejemplo, en cómo el bot no es capaz de administrar eficientemente sus recursos, acumulando minerales si razón aparente, no teniendo la capacidad de invertirlos en tecnologías o unidades militares. Lo que fuerza situaciones en donde el ejército del bot esta en desventaja, cuantitativa y/o cualitativa, frente al enemigo, a pesar de haber tenido el mismo tiempo para desarrollarse. De todas formas, el algoritmo de campos potenciales, junto a su optimización por NSGA-II, se muestra prometedor. Esto se puede ver en cómo la evaluación ofensiva seguía incrementando a través de las generaciones. Por otro lado, el algoritmo de generación de estrategias es difícil de ver su real eficacia al considerar todos los factores, pero cumple el objetivo de generar estrategias de forma rápida y que no sean fácilmente predichas por los enemigos.

Finalmente, como trabajo futuro quedan varias ideas abiertas. Primeramente, referente al sistema queda abierta la posibilidad de implementar cualquiera de las áreas no abordadas, tanto por separado como en conjunto, ya sea utilizando los algoritmos propuestos o utilizando nuevos. Así mismo, se pueden realizar mejoras a la arquitectura del sistema, con el objetivo de lograr una mejor modularización de las entidades con respecto a las áreas de investigación.

Respecto a los algoritmos implementados, una idea que surgió, pero no se implementó, fue la de utilizar NSGA-II para realizar una optimización previa a las estrategias (parecido a lo hecho en la táctica), utilizando la evaluación militar y la evaluación económica como objetivos independientes. Lo que permitiría obtener un frente de pareto, en el cual las soluciones representen estrategias enfocadas hacia el early o el late game. Lo que permitiría consecuentemente, el cambiar en tiempo real la meta estrategia para adaptarse a la situación. Además, se puede realizar optimización hacia los parámetros utilizados.

Para el caso de los algoritmos de la táctica, se tenía pensado el agregar un nuevo campo potencial a la ecuación, el cual se compondría de las casillas del mapa que no se pueden cruzar, i.e. obstáculos. Con esto el algoritmo de campos potenciales se podría utilizar completamente para moverse por el mapa, y posteriormente adaptar o agregar un algoritmo de pathfinding. Además, sería interesante cambiar los objetivos del NSGA-II, o modifican cómo estos se calculan. Por ejemplo, en muchos casos la partida se resuelve sólo con el resultado de las batallas entre unidades militares, y por lo tanto puede ser beneficioso no considerar el daño hecho a las estructuras o a los trabajadores dentro de la evaluación.

En cualquier caso, debido al tamaño del sistema, hay mucho trabajo que se hace para mejorar

o probar nuevos conceptos.

REFERENCIAS BIBLIOGRÁFICAS

[sta,] Frame rate. `http://www.starcraftai.com/wiki/Frame_Rate`. Accessed: 01-12-2017.

[Alburg et al., 2014] Alburg, H., Brynfors, F., Minges, F., Mattsson, B. P., and Svensson, J. (2014). Making and acting on predictions in starcraft: Brood war. *Bachelor of Science Thesis. University of Gothenburg*.

[Barriga et al., 2017] Barriga, N., Stanescu, M., and Buro, M. (2017). Combining strategic learning and tactical search in real-time strategy games. *arXiv preprint arXiv:1709.03480*.

[Buro, 2004] Buro, M. (2004). Call for ai research in rts games. pages 139–142.

[Buro and Furtak, 2004] Buro, M. and Furtak, T. M. (2004). Rts games and real-time ai research. 6370.

[Certicky, 2013] Certicky, M. (2013). Implementing a wall-in building placement in starcraft with declarative programming. *arXiv preprint arXiv:1306.4460*.

[Churchill and Buro, 2011] Churchill, D. and Buro, M. (2011). Build order optimization in starcraft. In *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

[Churchill and Buro, 2012] Churchill, D. and Buro, M. (2012). Incorporating search algorithms into rts game agents. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*.

[Deb et al., 2002] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197.

[Dereszynski et al., 2011] Dereszynski, E., Hostetler, J., Fern, A., Dietterich, T., Hoang, T.-T., and Udarbe, M. (2011). Learning probabilistic behavior models in real-time strategy games. In *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

[Fjell and Møllersen, 2012] Fjell, M. S. and Møllersen, S. V. (2012). Opponent modeling and strategic reasoning in the real-time strategy game starcraft. Master's thesis, Institutt for datateknikk og informasjonsvitenskap.

[García-Sánchez et al., 2015] García-Sánchez, P., Tonda, A., Mora, A. M., Squillero, G., and Merelo, J. (2015). Towards automatic starcraft strategy generation using genetic programming. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 284–291. IEEE.

[Gaudl et al., 2013] Gaudl, S. E., Davies, S., and Bryson, J. J. (2013). Behaviour oriented design for real-time-strategy games. In *FDG*, pages 198–205.

[Gonzalez and Garrido, 2011] Gonzalez, I. and Garrido, L. (2011). Spatial distribution through swarm behavior on a military group in the starcraft video game. In *2011 10th Mexican International Conference on Artificial Intelligence*, pages 77–82. IEEE.

[Hagelbäck, 2012] Hagelbäck, J. (2012). Potential-field based navigation in starcraft. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 388–393. IEEE.

[Hostetler et al., 2012] Hostetler, J., Dereszynski, E. W., Dietterich, T. G., and Fern, A. (2012). Inferring strategies from limited reconnaissance in real-time strategy games. *arXiv preprint arXiv:1210.4880*.

[Jónsson et al., 2012] Jónsson, B. et al. (2012). *Representing uncertainty in rts games*. PhD thesis.

[Lin and Ting, 2011] Lin, C.-S. and Ting, C.-K. (2011). Emergent tactical formation using genetic algorithm in real-time strategy games. In *2011 International Conference on Technologies and Applications of Artificial Intelligence*, pages 325–330. IEEE.

[Liquipedia, 2012] Liquipedia (2012). Micro and macro. `http://wiki.teamliquid.net/starcraft/Micro_and_Macro`. Accessed: 01-12-2017.

[Louis and Liu, 2018] Louis, S. J. and Liu, S. (2018). Multi-objective evolution for 3d rts micro. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE.

[Magnusson and Balsasubramaniyan, 2012] Magnusson, M. and Balsasubramaniyan, S. (2012). A communicating and controllable teammate bot for rts games.

[Naves and Lopes, 2012] Naves, T. F. and Lopes, C. R. (2012). Maximization of the resource production in rts games through stochastic search and planning. In *2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2241–2246. IEEE.

[Oen, 2012] Oen, R. E. (2012). Aspire adaptive strategy prediction in a rts environment. Master's thesis, The University of Bergen.

[Ontanón et al., 2013] Ontanón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., and Preuss, M. (2013). A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games*, 5(4):293–311.

[Parra and Garrido, 2012] Parra, R. and Garrido, L. (2012). Bayesian networks for micromanagement decision imitation in the rts game starcraft. In *Mexican International Conference on Artificial Intelligence*, pages 433–443. Springer.

[Perkins, 2010] Perkins, L. (2010). Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition. In *Sixth Artificial Intelligence and Interactive Digital Entertainment Conference*.

[Rathe and Svendsen, 2012] Rathe, E. A. and Svendsen, J. B. (2012). Micromanagement in starcraft using potential fields tuned with a multi-objective genetic algorithm. Master's thesis, Institutt for datateknikk og informasjonsvitenskap.

[Robertson and Watson, 2014] Robertson, G. and Watson, I. (2014). A review of real-time strategy game ai. *Ai Magazine*, 35(4):75–104.

[Schaeffer, 2001] Schaeffer, J. (2001). A gamut of games. *AI Magazine*, 22(3):29–29.

[Shaker et al., 2016] Shaker, N., Togelius, J., and Nelson, M. J. (2016). *Procedural content generation in games*. Springer.

[Sipper et al., 2007] Sipper, M., Azaria, Y., Hauptman, A., and Shichel, Y. (2007). Designing an evolutionary strategizing machine for game playing and beyond. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 37(4):583–593.

[Stanescu and Čertickỳ, 2014] Stanescu, M. and Čertickỳ, M. (2014). Predicting opponent's production in real-time strategy games with answer set programming. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(1):89–94.

[Synnaeve and Bessiere, 2011a] Synnaeve, G. and Bessiere, P. (2011a). A bayesian model for opening prediction in rts games with application to starcraft. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pages 281–288. IEEE.

[Synnaeve and Bessiere, 2011b] Synnaeve, G. and Bessiere, P. (2011b). A bayesian model for rts units control applied to starcraft. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pages 190–196. IEEE.

[Synnaeve and Bessiere, 2012] Synnaeve, G. and Bessiere, P. (2012). A bayesian tactician.

[Tavares et al., 2016] Tavares, A., Azpurua, H., Santos, A., and Chaimowicz, L. (2016). Rock, paper, starcraft: Strategy selection in real-time strategy games. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*.

[Tavares et al., 2014] Tavares, A. R., Azpúrua, H., and Chaimowicz, L. (2014). Evolving swarm intelligence for task allocation in a real time strategy game. In *2014 Brazilian Symposium on Computer Games and Digital Entertainment*, pages 99–108. IEEE.

[Weber et al., 2010] Weber, B. G., Mateas, M., and Jhala, A. (2010). Applying goal-driven autonomy to starcraft. In *Sixth Artificial Intelligence and Interactive Digital Entertainment Conference*.

[Weber et al., 2011] Weber, B. G., Mateas, M., and Jhala, A. (2011). A particle model for state estimation in real-time strategy games. In *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

[Weber, 2018] Weber, B. S. (2018). Standard economic models in nonstandard settings–starcraft: Brood war. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE.

[Wender and Watson, 2012] Wender, S. and Watson, I. (2012). Applying reinforcement learning to small scale combat in the real-time strategy game starcraft: Broodwar. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 402–408. IEEE.

[Wu and Markham, 2017] Wu, L. and Markham, A. (2017). Evolutionary machine learning for rts game starcraft. In *AAAI*, pages 5007–5008.

[Zhe et al., 2012] Zhe, W., Nguyen, K. Q., Thawonmas, R., and Rinaldo, F. (2012). Using monte-carlo planning for micro-management in starcraft. In *Proc. of the 4th Annual Asian GAME-on Conference on Simulation and AI in Computer Games (GAMEon ASIA 2012)*, pages 33–35.