BattleTech

Jugador Inteligente

María Carrasco Rodríguez

Francisco Manuel Herrero Pérez

Ingeniería del Conocimiento

Curso 09/10

Índice general

1

Introducción

BattleTech es un juego de combates entre enormes máquinas de aspecto humanoide llamados *BattleMechs* (o más brevemente llamados Mechs).

Este juego fue diseñado por la compañía americana *FASA*, que debido al gran éxito obtenido, ha ido creciendo hasta el infinito con diversas aplicaciones. En España, lo primero que se comercializó fué la trilogía de novelas El sol y la espada, en la que *Michael A. Stackpole* desarrolla las líneas argumentales básicas del universo que sirve de fondo para el juego. Posteriormente, se editó el juego de tablero, la primera y segunda versiones de las reglas del juego de rol, seguidos de distintos accesorios (módulos, ampliaciones, figuras a escala, etc).

El juego trata del enfrentamiento por la supremacía entre cinco unidades políticas herederas de un antiguo gran imperio -la Liga Estelar-, que se extendía hasta los 300 años-luz de la Tierra. Tras el derrumbe de la Liga, comienzan las Guerras de Sucesión entre las Grandes Casas (las familias que rigen cada una de las unidades políticas). Estas guerras acaban por destruir los medios de producción de bienes, con lo que tras 300 años de guerra continuada, la economía se basa en el saqueo sistemático para mantener la capacidad bélica de los diversos ejércitos.

La acción principal de las novelas tiene lugar en el periodo comprendido entre el 3027 y el 3058, en el que Hanse Davion, Principe de la Federación de Soles, logra firmar una alianza con la Mancomunidad de Lira para someter al resto de naciones y reinstaurar la Liga Estelar. Esto da lugar a una serie de enfrentamientos políticos y militares a gran escala que dan un buen transfondo al juego de rol.

El juego -tanto en tablero como en rol- se basa en las unidades militares de las Grandes Casas, compuestas por robots de 20 metros de altura (Mechs), cada uno con más armas encima que pinchos tiene un puercoespín. Como la ciencia ha decaido a un nivel similar al del siglo XXII, estos están cayéndose a pedazos,

pero aún así son capaces de destrozar una ciudad sin esfuerzo. Aunque parezca un poco tonto eso de llevar un robot por un tablero disparando a otros robots, es divertidísimo. Si además lo incluyes en una campaña de un juego de rol, puedes acabar por olvidarte de los otros vicios.

1.1 Descripción del Problema

En la practica se ha de diseñar y desarrollar un prototipo de Jugador Inteligente.

Nuestro objetivo es crear un jugador inteligente que sea capaz de dirigir a los BattleMechs. Para ello debe decidir la acción a realizar en cada momento por ellos. Nuestro problema se restringe únicamente a controlar a los Mechs, aunque existan muchos otros tipos de vehículos en los combates - BattleMech, vehículo, Pelotón de infantería o Punta o Escuadra de armaduras de combate.

1.1.1 BattleMechs

Titanes humanoides de metal de 12 metros de altura cubiertos con láseres, cañones automáticos y docenas de otras armas letales; poder suficiente para arrasar manzanas enteras de edificios.

1.1.2 Mapas

En Battletech las partidas se juegan en mapas divididos en áreas de seis lados llamadas hexágonos, las cuales regulan el movimiento y el combate entre varias unidades. Los mapas pueden tener bosques, ríos, lagos, montañas y más.

1.1.3 Secuencia de Juego

Una partida de Battletech consiste en una serie de turnos. Durante cada turno todas las unidades en el mapa tienen la oportunidad de moverse y disparar sus armas. Cada turno consiste en varios segmentos de tiempo más pequeños, llamados fases.

Durante cada fase, los jugadores pueden escoger un tipo de acción, cada turno incluye las correspondientes fases, que se desarrollan en el orden siguiente:

1. Fase de Movimiento
2. Fase de Reacción
3. Fase de Ataque con Armas
4. Fase de Ataque Físicos
5. Fase de Final de Turno

Fase de Movimiento

El jugador que ha perdido la iniciativa mueve primero. El jugador que ganó la iniciativa mueve entonces. El movimiento se alterna entre los bandos hasta que todas las unidades se hayan movido.

Las unidades de Battletech cambian de posición y ubicación en el mapa realizando uno de los distintos movimientos. Durante la Fase de Movimiento de cada turno los jugadores deben escoger un modo de movimiento, para los Mechs las opciones son andar, correr o saltar.

Fase de Reacción

Fase posterior al movimiento que permite a los mechs de girar el torso. Sólo se permite un movimiento de giro, por lo que podremos girar a la derecha, a la izquierda o permanecer igual.

Fase de Ataque con Armas

Fase en la que los jugadores, por orden de iniciativa, pueden abrir fuego con la artilleria de sus mechs. El Mech elegirá las armas que disparara, dentro de las que dispone en su arsenal, y sobre qué enemigo usarlas.

Fase de Ataque Físicos

Fase en la que se presenta una nueva opción la opción de realizar daño físico a otros Mech, y es la última opción de hacerles daño dentro del turno en el que nos encontramos. El ataque físico consistirá en realizar un impacto físico, con algún garrote/arma o extremidad del robot.

Fase de Final de Turno

Fase en la que los jugadores pueden determinar acciones especiales como tirar algún garrote o munición al suelo o apagar o encender radiadores.

1.1.4 Objetivo

Nuestro objetivo principal en el juego será acabar con todos los jugadores que haya en el tablero, exceptuando al nuestro. Para ello, debemos destruir una a una

el resto de las unidades que participan en el juego.

1.2 Abstracción del Problema

Para resolver nuestro problema podemos ver al jugador inteligente como un *ente*, que percibe el universo a través de un conjunto entradas - que nos indican el estado actual del juego. Este ente debe ser capaz de escoger una acción a realizar.

Pero, ¿Cómo escoger cual de entre todas las posibles acciones a realizar? La respuesta es bastante subjetiva, ya dependerá de la estrategia a seguir. Según la escogida, se obtendrá un resultado mejor o peor, pero también depende del caso concreto que estemos estudiando, ya que si cambiamos ciertos parámetros de nuestro universo, otra estrategia puede resultar más favorable.

A la hora de decidir qué estrategia seguir, se debe de tener en cuenta que tanto la **experiencia** adquirida tras años de juego y el **conocimiento** de las reglas del mismo, son puntos críticos a la hora de maxímizar nuestro rendimiento.

Nuestro ente se mueve en un espacio temporal discreto. El juego se divide por turnos y éstos a su vez, se ve dividido en segmentos más pequeños a los que llamamos fases. Cada fase se utiliza para realizar una acción determinada, y según la acción que queramos realizar, seguiremos una manera de razonar distinta dependiendo de los objetivos que queramos conseguir. Los *objetivos* últimos de nuestro ente podemos inferirlos de la definión del problema y son:

1. Sobrevivir hasta el final de la partida.
2. Eliminar al resto de entidades que participan en la partida.

A partir de un *conjunto de percepciones*, a través de las cuales conocemos el estado del mundo en el que nos encontramos, debemos ser capaces de decidir la *actitud* de nuestro ente en la situación. Esta actitud podrá ser ofensiva, atacante o neutral. Una vez sepamos el estado futuro que queremos alcanzar, estaremos en condiciones de decidir un *plan futuro de acción*.

También debemos tener en cuenta que nuestras acciones tienen una serie de consecuencias que producen un efeto en el ambiente. Así el estado en el que se encuentra el mundo cambia entre cada una de las fases. Es por ello que debemos actualizar las percepciones del mundo cada vez que nosotros u otor ente realiza una acción.

La naturaleza episódica del mundo en el que se encuentra nuesto ente hace una tarea fácil la división en fases del comportamiento del ente. Para cada una de las fases necesitamos tener en cuenta distintas características del mundo que percibimos, las cuales nos ayudarán a llegar a cumplir los objetivos.

También debemos tener en cuenta que las decisiones que tomemos en una fase afectarán al resto de las fases, por ello a la hora de tomar decisiones siempre debemos tener en mente como van a cambiar el mundo y como afectará al futuro.

2

Modelo Teórico

La inteligencia artificial, **IA**, es una disciplina relativamente nueva, la cual es considerada como una gran desconocida y una de las que más interés profano despierta. Pero, ¿qué es realmente la IA? Aunque existen muchas definiciones, podemos resumirlas en *"desarrollar sistemas que piensen y actúen racionalmente"*.

También hemos encontrado otras definiciones que resultan interesantes:

- *" The exciting new effort to make computers think ... machines with minds, in the full literal sense."* (Haugeland, 1985)

- *" The art of creating machines that perform functions that require intelligence shen permormed by people"* (kurzweil, 1990)

- *" The study of how to make computers do things at which, at the moment people are better"* (Rich and Knight, 1991)

En la IA se encuentra un paradigma conocido como *"paradigma de agentes"*, que aborda el desarrollo de entidades que puedan actuar de forma autómata y razonada. Si retomamos la definición dada anteriormente donde se consideraba a la IA como un medio para el desarrollo de sistemas que piensen y actúen racionalmente, podemos pensar que la IA, en su conjunto, trata realmente de construir precisamente dichas entidades autónomas e inteligentes.

> "Los agentes constituyen el próximo avance más significativo en el desarrollo de sistemas y pueden ser considerados como la nueva revolución en el software." Dr. Nicholas Jennings

2.1 Agente Inteligente

Un agente inteligente, es una entidad capaz de percibir su entorno, procesar tales percepciones y responder o actuar en su entorno de manera racional, es decir, de manera correcta y tendiendo a maximizar un resultado esperado.

En este contexto la racionalidad es la característica que posee una elección de ser correcta, más específicamente, de tender a maximizar un resultado esperado. Este concepto de racionalidad es más general y por ello más adecuado que inteligencia (la cual sugiere entendimiento) para describir el comportamiento de los agentes inteligentes. Por este motivo es mayor el consenso en llamarlos *agentes racionales*.

Los agentes inteligentes deben tener una serie de propiedades que exponemos a continuación:

- **Autonomía**

  Actúan por cuenta propia en nombre del usuario.

- **Inteligencia**

  Cerrada o adaptable al entorno (aprendizaje)

- **Reactividad – Proactividad**

  - Reactivo

    Actúa en función de los sucesos producidos en el entorno

  - Proactivo

    Toma la decisión de actuar antes de que se den los sucesos

- **Sociabilidad**

  Comunicación con: usuario/s, sistema/s, agentes/s...

- **Cooperación**

  Con otros agentes para realizar tareas de mayor complejidad

- **Movilidad**

  De un sistema a otro para acceder a recursos remotos o para reunirse con otros agentes

2.1.1 Clasificación

Es posible clasificar los agentes inteligentes en cinco categorías principales:

1. agentes reactivos.
2. agentes reactivos basados en modelo.
3. agentes basados en objetivos o metas.

4. agentes basados en utilidad.

5. agentes que aprenden.

6. agentes de consultas.

2.2 Ambientes

Existen diferentes tipos de ambientes:

- **Accesibles vs. no accesibles.** Un ambiente accesible es aquel en el cuál el agente puede obtener información completa, precisa y actualizada sobre el estado del ambiente. La mayoría de los ambientes moderadamente complejos (incluyendo el mundo físico e Internet) son inaccesibles. Mientras más accesible sea el ambiente es más sencillo construir agentes que funcionen sobre el.

- **Deterministas vs. no deterministas.** Un ambiente determinístico es aquel en el cuál cualquier acción tiene un único efecto garantizado No hay incertidumbre sobre el estado que resultará de realizar una acción. El mundo físico es no determinístico. Los ambientes no determinísticos son un mayor reto para los diseñadores de agentes.

- **Episódicos vs. no episódicos.** En un ambiente episódico, el funcionamiento de un agente depende de un número discreto de episodios, sin que halla relación entre el funcionamiento del agente en diversos escenarios. Los ambientes episódicos son más simples desde la perspectiva del desarrollador del agente porque el agente puede decidir que acción tomar basándose sólo en el episodio actual - no necesita reaccionar sobre las interacciones entre el episodio actual y los futuros.

- **Estáticos vs. dinámicos.** Si el abiente cambia mientras un agente toma una acción a seguir, entonces se dice que el ambiente es “dinámico”.

- **Discretos vs. continuos.** Un ambiente es discreto si tiene un número fijo y finito de acciones y percepciones.6:

  - ajedrez → discreto.
  - conductor → contínuo.

  Si existe una cantidad limitada de percepciones y acciones distintas y discernibles, se dice que el ambiente es discreto. Si no es posible enumerarlos, entonces es un ambiente continuo.

3

Justificación de la Idoneidad del Modelo Teórico

Nuestro **jugador inteligente** se corresponde con un **agente racional**, y por lo tanto debe poseer algunas de las características que éste posee:

- **Sociable.** Interacciona con otros agentes por *paso de mensajes*. De forma que pueden conocer el estdo entre ellos.
- **Reactivo.** El jugador podrá responder a cambios en el entorno en el que se encuentra situado. Estos cambios se le indican al jugador mediante los ficheros, que en cada fase de la partida el jugador debe leer para obtener toda la información necesaria.
- **Pro-activo.** El jugador debe ser capaz de intentar cumplir sus planes u objetivos. Su objetivo final será destruir a el resto de los jugadores.
- **Autonomía.** Nuestro agente actuará sin intervención de ningún usuario externo a nuestro programa. Por tanto tiene control total sobre sus acciones y

estados internos.

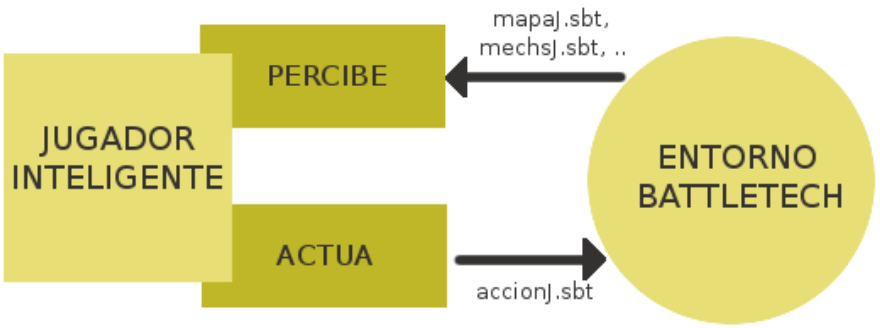


Figura 3.1: Esquema de funcionamiento.

Hemos de diferenciar nuestro sistema de los sistemas Multi-Agente (SMA). Éstos son grupos de agentes que interaccionan entre sí para conseguir objetivos comunes, pero en nuestro entorno los distintos jugadores no se comunican entre sí - no tienen conducta social-. Si tuvieramos esto en cuenta, aumentaría la complejidad en el desarrollo.

De entre todas las posibles clasificaciones de agentes inteligentes, el nuestro se acomoda mas rigurosamente al agente *basado en metas*. Estas ayudan a decidir las acciones correctas en cada momento. En nuestro juego, el agente o jugador escoge un objetivo, en base al cual toma sus decisiones. Por tanto, no basta con conocer el entorno, sino además es necesario determinar las acciones a seguir que permitan alcanzar la meta. Elegir las acciones correctas varia en complejidad.

- *Búsqueda*
- *Planificación*

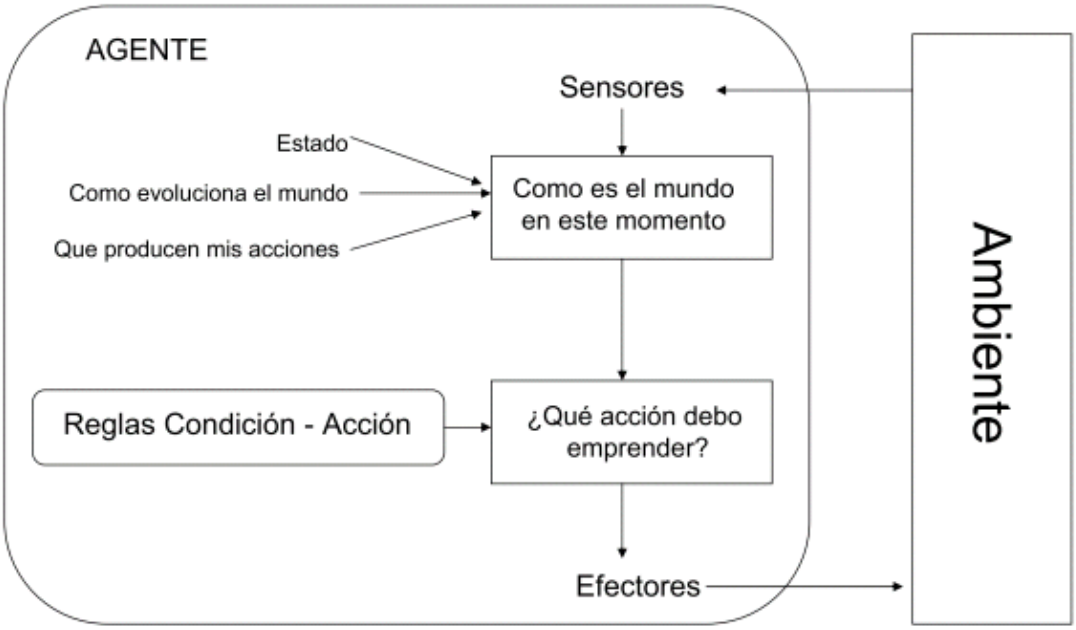


Figura 3.2: Agente basado en metas.

3.1 Ambientes

Las características de nuestro ambiente son:

- **Accesibles.** En nuestro caso se trata de un ambiente accesible ya que el agente tiene acceso al estado total del ambiente. Mediante los ficheros de mapa obtenemos toda la información necesaria.
- **No deterministas.** Los estados futuros son una función probabilística del estado actual y las acciones del agente.
- **Episódicos.** Es un ambiente episódico, la experiencia del agente se divide en "episodios"- o fases-. Cada episodio consta de un agente que percibe y actúa.

- **Estáticos.** En nuestro caso tratamos con ambientes *estáticos*, puesto que no se tiene que observar y pensar al mismo tiempo. Esto quiere decir que en cada fase tenemos un ambiente y para una cada agente, mientras decide su acción a realizar no cambiará. Aunque si cambia en una misma fase de un agente a otro.

- **Discretos.** Al existir una cantidad limitada de percepciones y acciones distintas y discernibles podemos decir que el ambiente es discreto, lo cual nos facilita el trabajo.

Programa de Ambientes.

Un simulador toma como entrada uno o más agentes y dispone de lo necesario para proporcionar las percepciones correctas una y otra vez a cada agente y así recibir como respuesta una acción.

El simulador procede a actualizar al ambiente tomando como base las acciones, y posiblemente otros procesos dinámicos del ambiente que no se consideran como agentes.

Los agentes se diseñan para que funcionen dentro de un conjunto de ambientes diversos. Para poder medir el desempeño de un agente es necesario contar con un simulador que seleccione ambientes particulares en los que se pueda probar al agente.

4

Descripción de la Solución

4.1 Módulo de Movimiento

4.1.1 Búsqueda de caminos

Uno de los mayores desafíos en el diseño de **Inteligencia Artificial** realista en juegos de ordenador es el movimiento del agente. Las estrategias de busqueda de caminos o *pathfinding* son empleados como centro de los sistemas de movimiento.

Las estrategias de pathfinding debe encontrar un camino desde cualquier coordenada del mundo hasta otra. Dados los puntos origen y destino, encuentran intermedios que formen un camino a nuestro destino. Para esto debemos tomar algún tipo de estructura de datos para guiarnos en el movimiento. Esto nos lleva inevitablemente a utilizar recursos de CPU, especialmente cuando buscamos un camino que no existe.

De entre todos los algoritmos que se usan actualmente, el más conocido y extensamente usado es el algoritmo A estrella **A***.

Veamos una comparación de tres tipos distintos de algoritmos.

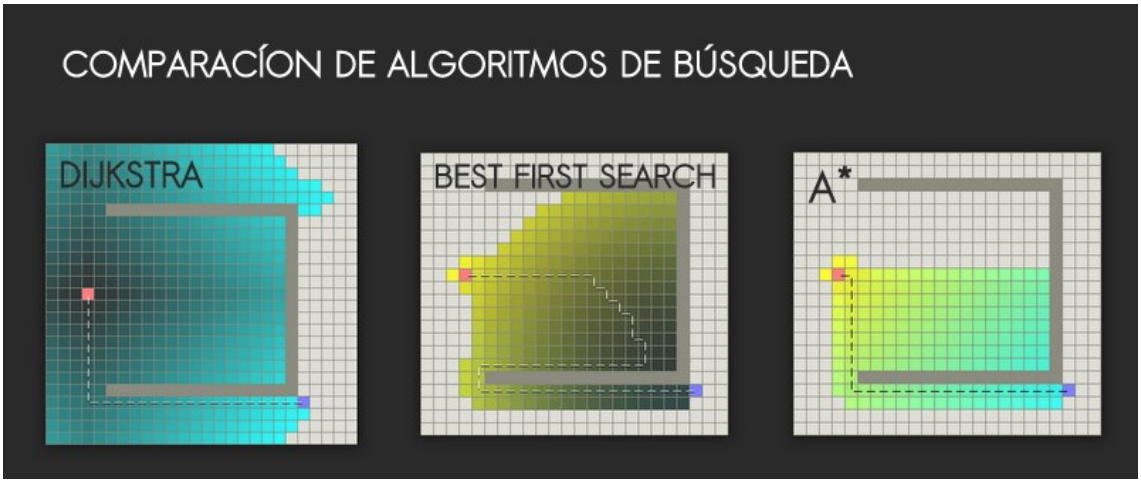


Figura 4.1: Comparación de algoritmos de búsqueda de caminos.

1. **Djkstra**. Este algoritmo empieza visitando los vértices del grafo en el punto de partida. Luego va reiteradamente examinando los vértices mas cercanos que aún no hayan sido examinados. Se expando desde el nodo inicial hasta alcanzar el destino. Pero aunque esté garantizado encontrar una solución óptima, comprueba demasiadas casillas, por lo que hace un gasto de recursos enorme.
   Para una implementación simple, tenemos un tiempo de ejecución $O(n^2)$

2. **Best First Search**. Es un algoritmo *Greedy* que trabaja de una forma simmilar al algoritmo de Dijkstra, aunque este presenta una "heurística" de como de lejos está nuestro objetivo. Aunque no nos garantice encontrar la solución óptima, si puede encontrar una solución apróximada en un tiempo mucho

menor. El mayor inconveniente con este algoritmo es que intenta moverse hacia el objetivo aunque no sea el camino correcto -tal y como muestra la figura 4.1.1. Esto es debido a que sólo tiene en cuenta el coste para llegar al objetivo, e ignora el coste del camino que lleva hasta ese momento. Entonces intentará seguir aunque el camino sea muy largo.
El tiempo de ejecución es $O(n)$.

3. **A***. Este algoritmo fue desarrollado en 1968 para combinar enfoques heurísticos como en *Best First Searh* y enfoques formales como ocurre en *Dijkstra*. Aunque A* este enfocado construido sobre la heurística -y aunque esta no proporciona ninguna garantía-, A* puede garantizar el camino más corto.

4.1.2 Algoritmo A*

El algoritmo de búsqueda A* es un tipo de algoritmo de búsqueda en grafos. Se basa en encontrar, siempre y cuando se cumplan ciertas condiciones, el camino de menor coste entre un nodo origen y uno objetivo.

Nuestro objetivo es encontrar el camino más corto entre dos puntos superando obstáculos (ya que en caso de que no hubiera obstáculos, es la línea recta). Esta técnica muy usada en videojuegos de estrategia y, en general en todos los videojuegos donde se trata la inteligencia artificial. Por ello decidimos incorporarla a nuestra práctica.

La mayor ventaja de este algoritmo con respecto a otros es que tiene en cuenta tanto el valor heurístico de los nodos como el coste real del recorrido. Así, el

algoritmo A* utiliza una función de evaluación:

$$f(n) = g(n) + h'(n)$$

Donde:

- **h'(n)** Valor heurístico del nodo a evaluar desde el actual n hasta el final.
- **g(n)** Coste real del camino recorrido para llegar a dicho nodo, n.

A* mantiene dos estructuras de datos auxiliares:

- **Abiertos**. Cola de prioridad, ordenada por el valor f(n) de cada nodo. (Lista de los nodos que necesitan ser comprobados)
- **Cerrados**. Guarda la información de los nodos que ya han sido visitados.

En cada paso del algoritmo se expande el nodo que esté primero en abiertos, y en caso de que no sea un nodo objetivo, calcula la f(n) de todos sus hijos, los inserta en abiertos, y pasa el nodo evaluado a cerrados.

4.1.3 Implementación

Algoritmo: *A*. A estrella*
Input: *start*: Node de comienzo; *goal* Nodo final
Output: Lista con nodos que forman el camino (si existe).

```
closed_set = {}

start_node = start
start_node.g_cost = 0
start_node.f_cost = compute_f_cost(start_node, goal)

open_set = PriorityQueueSet()
open_set.add(start_node)

while len(open_set) > 0:
curr_node = open_set.pop_smallest()

if curr_node.coord == goal:
return reconstruct_path(curr_node)

closed_set[curr_node] = curr_node

for succ_coord in successors(curr_node.coord):
succ_node = succ_coord
succ_node.g_cost = compute_g_cost(curr_node, succ_node)
succ_node.f_cost = compute_f_cost(succ_node, goal)

if succ_node in closed_set:
continue

if open_set.add(succ_node):
succ_node.pred = curr_node

return []
```

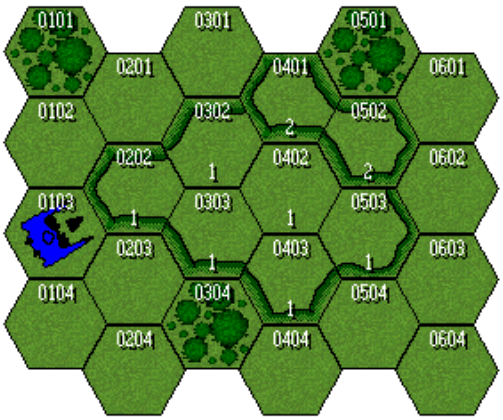
La implementación es genérica y no se basa en ningún juego en concreto. **PathFinder** es una clase genérica que no tiene en cuenta como representes tu grafo ni como representes o como calcules el coste de moverte de un lugar a otro. Deja a gusto del programador especificar la información mediante el paso de funciones al constructor.

Pathfinder

Esta clase implementa nuestro algorimo A*. Veamos ahora de que se compone.

Sucesores o Hijos

¿Cómo sabe *PathFinder* la forma de nuestro grafo? Sólo nos basta con especificarlo en la función **successors**. Esta función especifica los sucesores de un node, que son aquellos nodos a los que se puede llegar desde el nodo inicial en un solo paso.



En nuestro ejemplo, los sucesores del nodo *0103*, donde se encuentra nuestro

personaje, son: *0102, 0202, 0203, 0104*

Ateniéndonos a las restricciones de *BattleTech*, vemos que tenemos que tener en cuenta varias cosas:

- La diferencia de altura entre dos casillas colindantes no puede mayor a dos.
- Si corremos, la profundidad del agua tiene que ser menor a uno.
- En el caso de que andemos o corramos, debemos tener en cuenta si la casilla esta incendiada. En tal caso el fuego puede provocar un sobre calentamiento muy peligroso.
- El el movimiento hacia atrás el máximo desnivel permitido es uno.

Para comprobar todos estos casos hacemos uso de la función **checkCell(i,d, mov)**, donde **i** es la casilla desde la que nos queremos mover, **d** es la casilla a la que nos queremos mover y **mov** es el tipo de movimiento a realizar.

Función de evaluación

Función de coste

PathFinder también conoce el coste de cada movimiento ya que se lo estamos diciendo en la función **move_cost**. Esta función nos dice el coste real para movernos de un nodo a otro.

Para implementar esta función hemos tenido en cuenta varios factores:

- *El tipo de movimiento.* Andar, correr o saltar. Cada tipo de movimiento tiene sus propias restricciones, las cuales hemos de tener en cuenta.

- *Los objetos.* Dependiendo del tipo de objeto(escombros, edificios ...) que se encuentre en una casilla, será más o menos costoso - o incluso imposible - el paso a través de ella.
- *El nivel o profundiad.* El máximo desnivel posible andando y corriendo es 2, mientras que saltando es un desnivel no mayor a los PM_{salto}.
- *Tipo de terreno.* Cada tipo de terreno lleva asociado un coste distinto.
- *Cambio de encaramiento.* Cambiar nuestro encaramiento también tiene un coste asociado, por lo tanto, debemos tener en cuenta el número de veces que giramos para poder desplazarnos a la casilla deseada.

Función heurística

La última función que pasamos como argumento a *PathFinder* es **heuristic_to_goal**. Tal y como su nombre indica es la heurística o coste del movimiento estimado para ir de un nodo origen a un nodo destino.

Hay muchas formas de realizar esta función, la primera que utilizamos es la **distancia manhatan**.

$$manhatan(x, y, x', y') = Vx + Vy$$

Siendo,

$$Vx = |x' - x|$$

$$Vy = |y' - y|$$

Esta heurística no es completamente precisa, ya que nos ofrece una sobreesti-

mación del resultado correcto. Esto se debe a que la conectividad entre celdas es diferente en un mapa hexagonal que en uno rectangular, aunque la representación que hemos usado del mapa pueda invitar al error, ya que usamos una matriz cuadrada.

Finalmente, tras realizar varios experimentos con la función *manhattan*, nos dimos cuenta de que conducía a resultados erróneos en un número elevado de ocasiones. Así, encontramos la función que nos da la distancia exacta para un tablero hexagonal:

$$hexagonal\_distance(x, y, x', y') = Vx + max\{0, Vy - (Vx/2) - factor\}$$

Siendo:

$$factor = \begin{cases} 0 & \text{if } mod(Vy, 2) \neq 0 \\ mod(x - 1, 2) & \text{if } y < y' \\ mod(x' - 1, 2) & \text{otherwise} \end{cases}$$

De esta forma obtenemos la distancia real entre dos casillas de un tablero hexagonal. Esto nos permite una mayor precisión en nuestros cálculos.

Implementando la lista abierta

Un aspecto interesante de implementación para mejorar la optimalidad de nuestro programa es cómo implementar la lista de nodos abiertos. Recordamos que A* usa la lista abierta para realizar un seguimiento de los nodos que todavía tiene que vistar. Esta quizás sea la estructura de datos más importante para el algoritmo, y su correcta implementación es no trivial.

Aunque al principio empezamos con una implementación ineficiente de la lista abierta, al decidir optimizar su implementación con una estructura de datos más acertado mejoró cien veces su velocidad. Una de las mejores fuentes de inspiración fueron las notas de Amit.5

Combinando las colas de prioridad y las estructuras de datos tipo *set*, se consigue una cola de prioridad en la que sus componentes están garantizados a ser únicos.

Esto nos proporciona una complejidad O(1) para pruebas de pertenencia, y =(log N) para la extracción del menor elemento. La inserción en cambio es mas complicada. Cuando un elemento no existe, se añade en O(log N). En cambio, cuando ya existe, su prioridad se comprueba con el nuevo elemento en O(1). Si la prioridad del nuevo elemento es menor, se actualiza en la cola. Esto lleva un tiempo O(N).

4.1.4 Lógica del movimiento

Todo esto de cómo encontrar el camino óptimo está muy bien, y además nos ayuda en gran parte a decidir como movernos, pero ¿Cómo hacerlo?

Nos basamos en conseguir una meta, una vez observado el ambiente, tendremos que decidir a qué casilla nos vamos a mover. Lo primero que vamos ha hacer será fijarnos un **objetivo enemigo**, y a partir de ahí seguiremos razonando según las observaciones que hagamos del mundo. Decidimos que nuestro objetivo sería el jugador más cercano al nuestro. Esto es así ya que según nuestra experiencia

adquirida en el juego, vimos que era la mejor opción.

Una vez hemos fijado un objetivo enemigo, debemos determinar la **posición objetivo**. Uno de los factores más importantes a la hora de elegir la posición a la que queremos movernos es saber si tenemos iniciativa o no en la fase actual. Si tenemos iniciativa - esto quiere decir que nos movemos antes que nuestro enemigo -, intentaremos elegir una posición estrategica, no demasiado cercana, a la que le resulte más complicado dispararnos. Si en cambio no tenemos la iniciativa, vamos a intentar acercarnos lo más posible a nuestro enemigo, y además queremos quedar encarados de forma que nuestro enemigo quede dentro de linea de visión.

También debemos tener en cuenta la distancia a la que esté nuestro objetivo, ya que todo esto influirá también en nuestra decisión.

Pero otra pregunta que surge inevitablemente una vez que hemos decidido la posición futura es, ¿Cómo nos vamos a mover?. Para responder utilizamos una serie de reglas de producción que basamos en heurísticas muy simples:

- Si tenemos puntos suficientes para llegar andando, andamos.
- Si tenemos puntos suficientes para llegar corriendo, corremos.
- Si tenemos puntos suficientes para llegar saltando, saltamos.

Si ninguna de estas reglas se cumplen, decidiremos intentar llegar a la posición más cercana a la posición objetivo andando. Damos una prioridad al movimiento de andar ya que es el movimiento más benévolo. Genera menos calor y nos deja con un disparo más limpio.

¿Como llegar ahora a esa posición objetivo? Si ya estamos en la posición que queremos, nos quedamos *inmóviles*. Si podemos llegar andando, entonces andamos. En caso contrario, si podemos llegar al destino corriendo, entonces corremos.

Si podemos llegar de un salto, entonces saltamos. La última decisión es si no podemos llegar de ninguna de las tres formas, entonces calcularemos una semi-ruta en la que nos moveremos andando.

Mech en suelo.

Otra situación muy común durante una partida de BattleTech es que los Mechs caigan al suelo. Aunque esto puede suceder por varios motivos, siendo uno de ellos la decisión propia, en nuestro caso esta última no se da. Esto es debido a que pensamos que no nos compensa realizar esta acción y por lo tanto no favorece para logar los objetivos marcados.

Por tanto, nuestro agente siempre intentará permanecer de pie. Nuestro comportamiento es entonces muy sencillo en estas situaciones:

1. Si nuestro jugador esta en el suelo y tiene puntos suficientes para poder levantarse, entonces nos levantamos antes de intentar realizar cualquier otro movimiento:
$$PM(andar) \geq 2$$
2. En caso contrario no realizar ninguna acción.

Aunque según la documentación del juego existe una regla especial que nos dice que al quedarnos $PM(andar) == 1$, tendrémos una oportunidad especial para hacer un intento de levantarnos, la experiencia con la interfaz de nuestro juego nos ha revelado muchos problemas. Así que, decidimos quitar esta opción.

4.2 Módulo de Reacción

El módulo de reacción es inmediatamente posterior al módulo de movimiento. En esta fase se presenta la posibilidad de corregir el encaramiento del torso para

mejorar la estrategia de ataque - mejorando el ángulo de disparo, tenemer a mas enemigos en línea de visión, proteger partes débiles de nuestro mech, etc. -.

Las posibilidades que se nos presentan ahora son tres:

```
change = {0: 'Igual', 1: 'Derecha', 2:'Izquierda'}
```

Las cuales nos indican lois tres posibles cambios que podemos realizar:

1. **Igual**. Indica que no vamos a realizar ninguna cambio. Dejamos nuestro encaramiento intacto para la siguientes fases.
2. **Derecha**. Giramos el mech para posicionar su torso una cara a Derecha.
3. **Izquierda**.Giramos el mech para posicionar su torso una cara a Izquierda.

4.3 Módulo de Ataque con Armas

4.3.1 Lógica del ataque con armas

El ataque con armas es una de las fases más importantes del juego. El objetivo principal del Mech en la fase de ataque con armas será elegir a que enemigos y con que armas desea atacar. Para poder realizar esta fase inteligentemente, dotaremos al Mech de la capacidad para analizar los distintos condicionantes que intervienen en el ataque. Antes de adentrarnos en que tipos de condicionantes, es necesario definir la estrategia general que realizarán nuestros Mechs. La estrategía será maximizar el daño sobre el atacado minimizando los daños en el atacante, es decir, atacaremos al Mech objetivo con todas las armas posibles mientras no se supere

un valor umbral de daño en forma de calor en el atacante. Por tanto queda claro que los condicionantes que debe determinar el Mech serán:

1. Se realiza ataque.
    a) Localización del Mech objetivo.
    b) Elección de armas a usar en el ataque.
2. No se realiza ataque.

Se realiza un ataque

En primer lugar es necesario saber si el Mech desea atacar o prefiero no realizar ningún tipo de ataque. Para discriminar entre las dos actuaciones nos basamos en calcular que enemigos están dentro de nuestra línea de visión, pero con la práctica, hemos visto que el programa *LDVyC.exe* usado para calcular si un enemigo está en la línea de visión no funciona como esperabamos, es decir, el programa indica que un Mech está en linea de visión incluso cuando nuestro enemigo está a nuestra espalda. Por tanto, además de que esté en línea de visión, comprobamos que nuestro torso esté en un ángulo correcto para el ataque. Si existe al menos un Mech que cumpla estos requisitos el ataque se realizará.

Localización del Mech objetivo

Una vez seleccionados los Mechs que están dentro de nuestra línea de visión, elegimos cual es el que está más próximo a nosotros, calculando la distancia entre nuestra casilla en el tablero y las casillas de los enemigos a los que podemos atacar. Determinamos así el Mech objetivo sobre el que recaerán todo el poder de nuestras armas.

Elección de armas a usar en el ataque

Para la resolución de este problema usaremos el enfoque teórico del algoritmo de la mochila que es del tipo algoritmo voraz. Lo algoritmos voraces son aquellos, que para resolver un determinado problema, sigue una metaheurística consistente en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima.

Algoritmo de la mochila

Figura 4.2: Algoritmo de la mochila

"El problema de la mochila consiste en llenar una mochila con n objetos. Cada objeto i tiene un peso determinado ci siempre positivo y una utilidad o valor asociado, también positivo, bi. Se ha de considerar además que la mochila tiene una capacidad limitada P, por tanto, se han de escoger aquellos objetos xi que maximicen la utilidad de quien llena la mochila sin exceder su capacidad".

Cada arma tiene un determinado valor equivalente a la distancia que es capaz de alcanzar en un disparo, por tanto bajo el enunciado del algoritmo de la mochila, esos n objetos a introducir en la mochila serán las armas que tiene el Mech que va a realizar el ataque, las cuales sean capaces de alcanzar al Mech objetivo. Una vez realizada esta discriminación en las armas a usar, determinaremos el peso de cada arma ci, que será la temperatura que genera en el arma sobre el Mech atacante al ser disparada. El parámetro bi de calidad de cada arma n responderá a la siguiente expresión:

$$bi(n) = \frac{n.getDano()}{n.getCalor()}$$

La calidad de un arma será el resultado de dividir el daño que produce dicha arma en el atacado entre el calor que produce el arma en el atacante. Una vez ordenadas las armas en función de su calidad, debemos determinar la capacidad de la mochila P. Denominaremos a P como el umbral máximo de temperatura que puede soportar un Mech después de realizar un ataque.

Determinación del valor umbral de temperatura

Debe quedar claro que lanzar todas las armas posibles a nuestro enemigo no es una buena estrategia, ya que a cada disparo, la temperatura de nuestro Mech puede ascender de tal forma que incluso quede desactivado. Por tanto debemos calcular cuanta temperatura puede soportar el Mech en la realización del ataque. Sabemos que si un Mech pasa de 15º al acabar el turno, recibirá un punto de daño y si sobrepasa 26º recibirá dos. Por tanto podemos determinar el umbral de temperatura soportable de la siguiente forma:

- Si la temperatura del Mech es menor que 10º el umbral será 12º.
- Si la temperatura del Mech está comprendida entre 10º y 15º el umbral

será 9º.

- Si la temperatura del Mech está comprendida entre 16º y 20º el umbral será 6º.
- Si la temperatura del Mech está comprendida entre 20º y 26º el umbral será 4º.
- Si la temperatura del Mech es mayor que 26º el umbral será 2º

Determinado P (umbral de temperatura) ya tenemos las armas que usaremos en el ataque dentro de nuestra *mochila*.

No se realiza ataque

Si no hay ningún enemigo dentro de nuestra línea de visión no realizaremos ningún ataque, quedando como única posibilidad recoger un garrote, sí está alojado en nuestra casilla, para usarlo más tarde en un posible ataque físico.

4.4 Módulo de Ataque Físico

Al igual que el ataque físico el Mech deberá tener en cuenta unos condicionantes para calcular la estrategia a seguir durante la fase de ataques físicos:

1. Se realiza ataque.
   a) Localización del Mech objetivo.
   b) Elección de armas a usar en el ataque.
2. No se realiza ataque.

Se realiza ataque

En primer lugar, al igual que en el ataque con armas, debemos conocer cuales son los posibles objetivos de dicho ataque. Para poder realizar un ataque físico es condición indispensable que el Mech atacado esté dentro de nuestra línea de visión, en un ángulo correcto para realizar el ataque, en una casilla adyacente y además que el nivel de su casilla sea igual a la nuestra. Si existe al menos un Mech enemigo que cumpla esos requisitos respecto a nuestro Mech se podrá realizar el ataque.

Localización del Mech objetivo

A partir del conjunto de Mech objetivos que cumplan la condición para ser atacados, localizaremos el más cercano a nosotros y será el objetivo final de nuestro ataque físico.

Elección de las armas a usar en el ataque

Determinado finalmente el Mech objetivo, necesitamos saber con que lo atacaremos. En el caso del ataque físico las armas a usar contra el objetivo serán nuestras extremidades, patadas y puñetazos, o un garrote recogido en anteriores fases. Si disponemos del garrote, simplemente atacaremos con él, si no, procederemos a usar las extremidades. Para determinar que extremidades podemos usar en el ataque debemos identificar cuales son las armas que usamos en el anterior ataque con armas, ya que si se ha usado un arma alojada en el brazo derecho, dicha extremidad ya no podrá ser usada en el ataque. Por tanto, usando de nuevo el módulo de ataques con armas, volveremos a estimar que armas fueron usada y en que extremidad están alojadas. Las extremidades que no fueron usadas en el

ataque con armas podrán ser utilizadas en el ataque físico. Si usamos los brazos debemos tener en cuenta que sólo se podrán usar los puños contra nuestro enemigo si no está tumbado en el suelo y si nuestro torso está en el ángulo correcto. De igual manera, si usamos las piernas, el ángulo de ellas deberá ser correcto en relación al enemigo. Queda aclarar que si podemos usar las dos piernas, eligiremos una cualquiera, ya que no es posible dar una patada con dos piernas a la vez.

No se realiza ataque

Si no se cumplen los requisitos para realizar un ataque físico el Mech no hará nada, pasando así a su fase de fin de turno.

4.5 Final de Turno

En esta fase es de utlidad para estar preparado para el siguiente turno de forma idónea. Podemos arrojar al suelo útiles de nuestro Mech, tales como munición o un garrote.

Además podremos apagar o encender radiadores, si disponemos de ellos, en función de nuestra temperatura.

Nuestro jugador inteligente arrojará el garrote si lo posee y si ha perdido alguna extremidad, buscará munición de las armas contenidas en las extremidades perdidas, arrojándolas al exterior.

5

Conclusión

Esta memoria relata el largo proceso de diseño y creación de un **agente inteligente**. Para logarlo, nos hemos basado en los fundamentos teóricos que nos han explicado en la asignatura, y haciendo elecciones de entre todas las posibilidades ofertadas, hemos llegado a desarrollar una arquitectura que nos ha permitido dotar a nuestro jugador de inteligencia.

Primero realizamos un proceso de búsqueda de información que nos resultase necesaria para entende el proceso que ibamos a llevar acabo en la práctica, más tarde, nos dimos cuenta de que la necesidad de aprender a jugar de forma mas "avanzada", para poder llegar a entender la lógica de cada una de las fases del juego. Después de mucho tiempo y ayuda, conseguimos entender qué debiamos hacer en el juego y cómo hacerlo. Así al conseguir una visión abstracta del conjunto, ya empezamos a la programación de la práctica.

Una decisión muy importante fue la el lenguaje de programación a utilizar. Después de considerar muchas opciones, decidimos que la mejor opción sería utilizar **Python**. Python es un lenguaje que todo el mundo debería conocer. Su

sintaxis simple, clara y sencilla; el tipado dinámico, el gestor de memoria, la gran cantidad de librerías disponibles y la potencia del lenguaje, entre otros, hacen que desarrollar una aplicación en Python sea sencillo, muy rápido y, lo que es más importante, divertido. La sintaxis de Python es tan sencilla y cercana al lenguaje natural que los programas elaborados en Python parecen pseudocódigo. Por este motivo se trata además de uno de los mejores lenguajes para comenzar a programar. La ventaja de los lenguajes compilados es que su ejecución es más rápida. Sin embargo los lenguajes interpretados son más flexibles y más portables. Gracias a la eficiencia de nuestro código, y a la complejidad no muy elevada del mismo - cosa que hemos tenido muy en cuenta a la hora de programar -, los requisitos de velocidad de ejecución de la práctica se cumplen, por lo que no hemos tenido problemas con la velocidad.

Otra característica importante es que el intérprete de Python está disponible en multitud de plataformas (UNIX, Solaris, Linux, DOS, Windows, OS/2, Mac OS, etc.) por lo que si no utilizamos librerías específicas de cada plataforma nuestro programa podrá correr en todos estos sistemas sin grandes cambios. Además, con el programa py2exe, podremos generar ejecutables para funcionar en windows casi sin ningún problema. [1]

[1] En nuestro caso, sí tuvimos problemas, por lo que la práctica no funciona correctamente en las aulas de prácticas.

Bibliografía

[1] González Duque, R., 2003. *"PYTHON para todos."* 1^{st}ed.

[2] Gutschmidt, Tom, 2003. *"Game Programming with Python, Lua, and Ruby."* Premier Press

[3] Weixiong Zhang, 1999. *"State-Space Search. Algorithms, Complexity, Extensions and Applications."* Springer-Verlag: NY

[4] Pilgrim, Mark, 2004. *"Dive into Python."* Apress.

[5] Amit Patel
`http://theory.stanford.edu/~amitp/GameProgramming/`

[6] Russell, S. & Norvig, P. *"Artificial Intelligence: A Modern Approach"* 3^{rd}ed.

[7] `http://www.policyalmanac.org/games/aStarTutorial.htm`

[8] `http://www.solaris7.com/`

[9] `http://code.google.com/p/smart-player/`