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).

1.1 Descripción del Problema

En la practica se ha de diseñar y desarrollar un prototipo de Ju-

gador Inteligente.

1.2 Abstracción del Problema

La inteligencia artificia, **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

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*.

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

- **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.

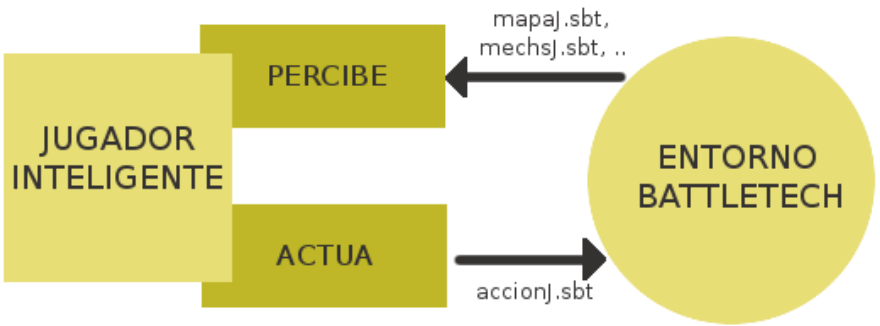


Figura 1.1: Esquema de funcionamiento.

Hemos de diferenciar nuestro sistema de los sistemas Multi-Agente (SMA). Éstos son grupos de agentes que interaccionan entre sí para conseguir objeti-

vos 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.

Nuestro agente es *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*

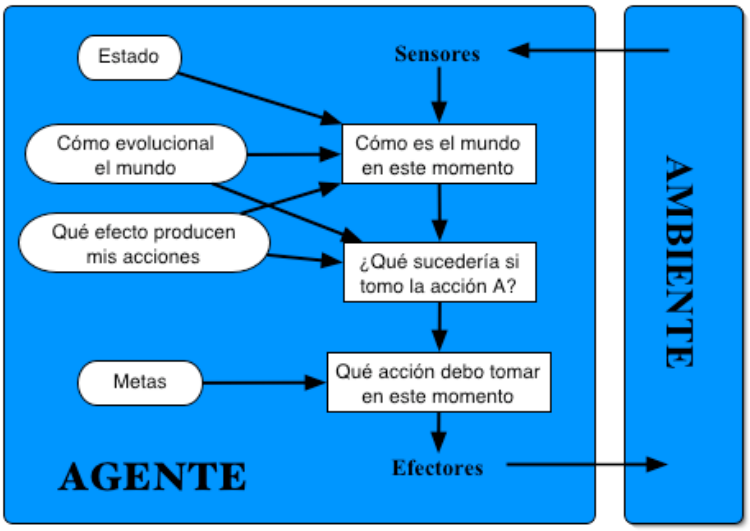


Figura 1.2: Agente basado en metas.

Ambientes

Existen diferentes tipos de ambientes:

- **Accesibles y no accesibles.** En nuestro caso se trata de un ambiente accesible, ya que el agente tiene acceso al estado total del ambiente.
- **Deterministas y no deterministas.** Depende de si el estado siguiente se determina a partir del estado y las acciones elegidas por el agente.

- **Episódicos y no episódicos.** En ambientes episódicos, como es nuestro caso, la experiencia del agente se divide en "episodios"- o fases-. Cada episodio consta de un agente que percibe y actúa.
- **Estáticos y dinámicos.** Si el abiente cambia mientras un agente toma una acción a seguir, entonces se dice que el ambiente es "dinámico". Pero en nuestro caso tratamos con ambientes *estáticos*, puesto que no se tiene que observar y pensar al mismo tiempo.
- **Discretos y continuos.** 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. Nuestro problema abarca ambienes discretos, 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.

2

Modelo Teórico

3

Descripción de la Solución

3.1 Movimiento

3.1.1 Lógica del movimiento

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.

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

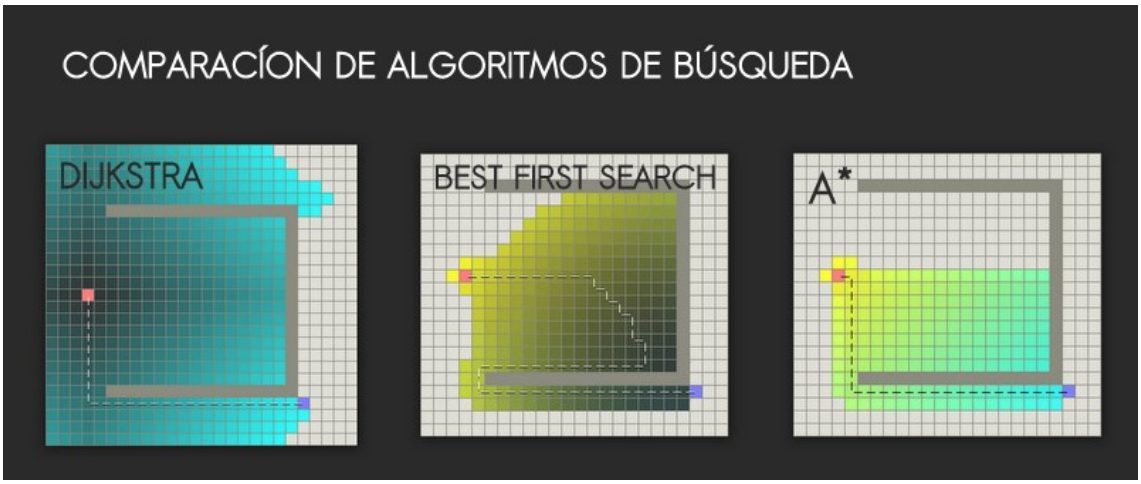


Figura 3.1: Comparación de algoritmos de búsqueda de caminos.

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 3.1.1. 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.

3.1.2 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.

3.2 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

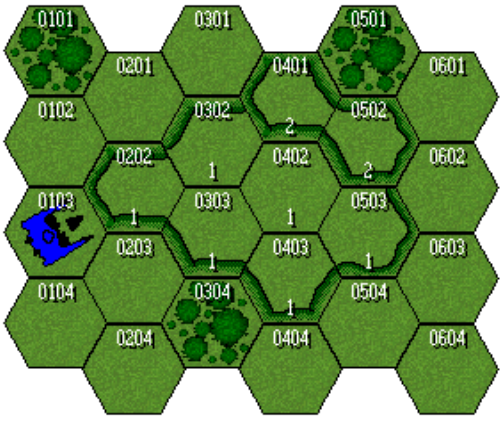
tu grafo ni como representes o como calcules el coste de moverte de un lugar a otro. Deja a gusto del programador especificar la información mediante el paso de funciones al constructor.

3.2.1 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.

3.2.2 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 sobreestimación 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.

3.2.3 Implementando la lista abierta

Un aspecto interesante de implementación para mejorar la optimalidad de nuestro programa es cómo implementar la lista de nodos abiertos. Re-

cordamos 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).

3.3 Reacción

3.4 Ataque con Armas

3.5 Ataque Físico

3.6 Final de Turno

Esta fase nos brinda la posibilidad de apagar o encender radiadores, soltar garrote o expulsar municiones.

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]

[9]

[10]