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).

2

Descripción del Problema

3
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 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

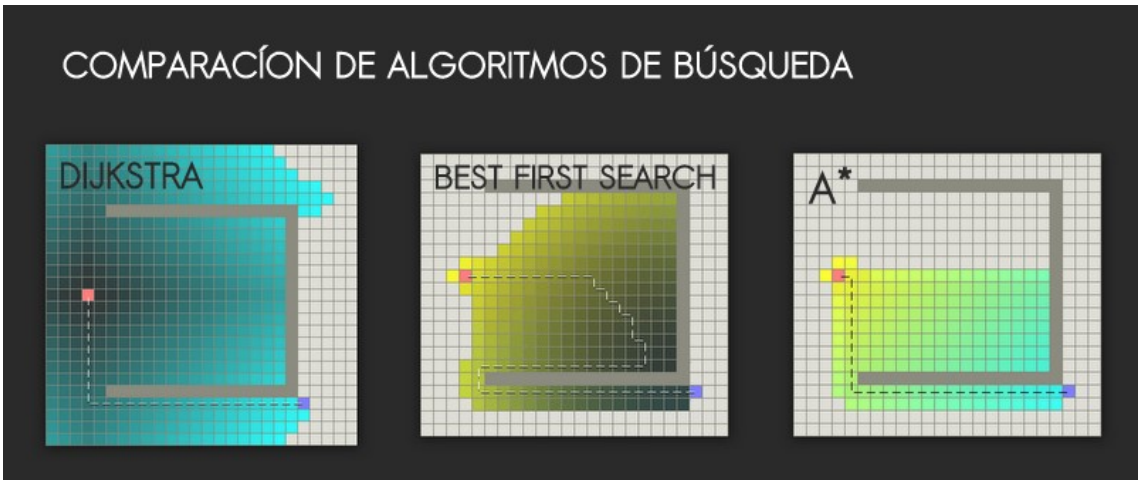


Figura 3.1: Comparación de algoritmos de búsqueda de caminos.

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. 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 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.

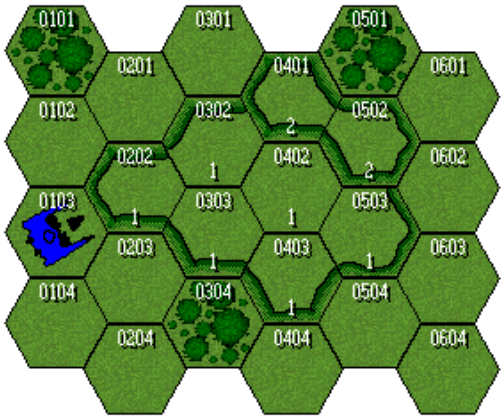
3.2 Implementación

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

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.

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**.

$$manhatan(x, y, x', y') = |x' - x| + |y' - y|$$

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 usada del mapa invite al error.

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. 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

Reacción

5

Ataque con Armas

6

Ataque Físico

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

[7]

[8]

[9]