Inteligencia Artificial para Videojuegos

Navegación

Resolución de problemas en el espacio de estados

Motivación

- El problema más habitual es *buscar un camino*… no cualquiera, sino el *más corto*
  - El de *menor coste*, en tiempo y distancia
  - El agente expresa así *inteligencia*, y una navegación de lo más *creíble* (en general)

- ¡Es un concepto potente, que se puede aplicar a *muchos problemas*!

Puntos clave

- Hitos históricos
- Problemas en el espacio de estados
  - Ejemplo
- Algoritmo de Dijkstra
  - Explicación
  - Pseudocódigo
  - Estructuras de datos
  - Debilidades

Hitos históricos

- El ajedrez es uno de los primeros dominios estudiados, sobre todo en años 50 y 60
  - Juego de información completa
  - Aunque a día de hoy sabemos que "sólo" la fuerza bruta no sirve y se usan heurísticas

Problemas en el espacio de estados

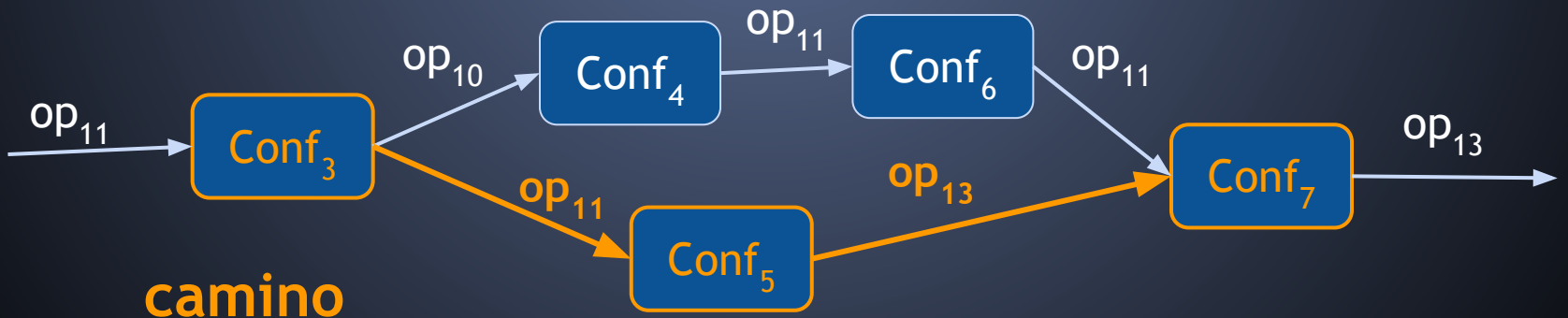
- La clave está en definir el problema así
  - Configuraciones posibles, empezando por la *inicial*
  - Operadores posibles, con una función para conocer los *aplicables* a una determinada configuración
  - Modelo de transición, función que devuelve la configuración resultante de aplicar un cierto operador a una configuración concreta
  - Prueba de objetivo, función que indica si una cierta configuración es *objetivo* (pueden serlo varias)
  - PATH COST *Coste del camino**, función que asigna un valor numérico a cada *camino* (suma de los costes *no negativos* de cada paso/transición)

STEP COST

* Es interesante tenerla, aunque la IA no la use como utilidad

Problemas en el espacio de estados

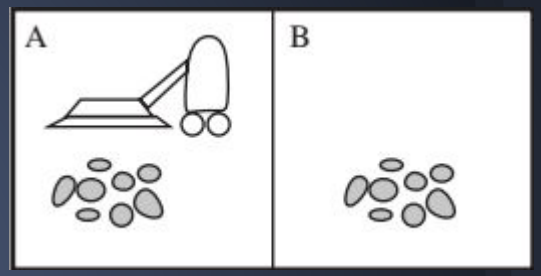
- En dominios de trabajo, ¿qué es un *camino*?
  - La *configuración inicial*, los *operadores* y el *modelo de transición* definen implícitamente un grafo ponderado no-negativo dirigido con todas las configuraciones alcanzables (espacio de estados)
  - Un camino en *este espacio* es *cualquier secuencia de configuraciones conectadas por operadores*



* Coste = Coste de la transición de $Conf_3$ a $Conf_5$ + de $Conf_5$ a $Conf_7$

Ejemplo

- El “juego” de la aspiradora
  - Un mundo con 2 casillas, A y B
    - Puede o no “aparecer” *polvo*
  - Un *robot-aspiradora* como NPC (con su IA)
    - Sabe *en qué casilla está* y *si tiene polvo*
    - Puede *aspirar*, moverse a la *izquierda* o a la *derecha*

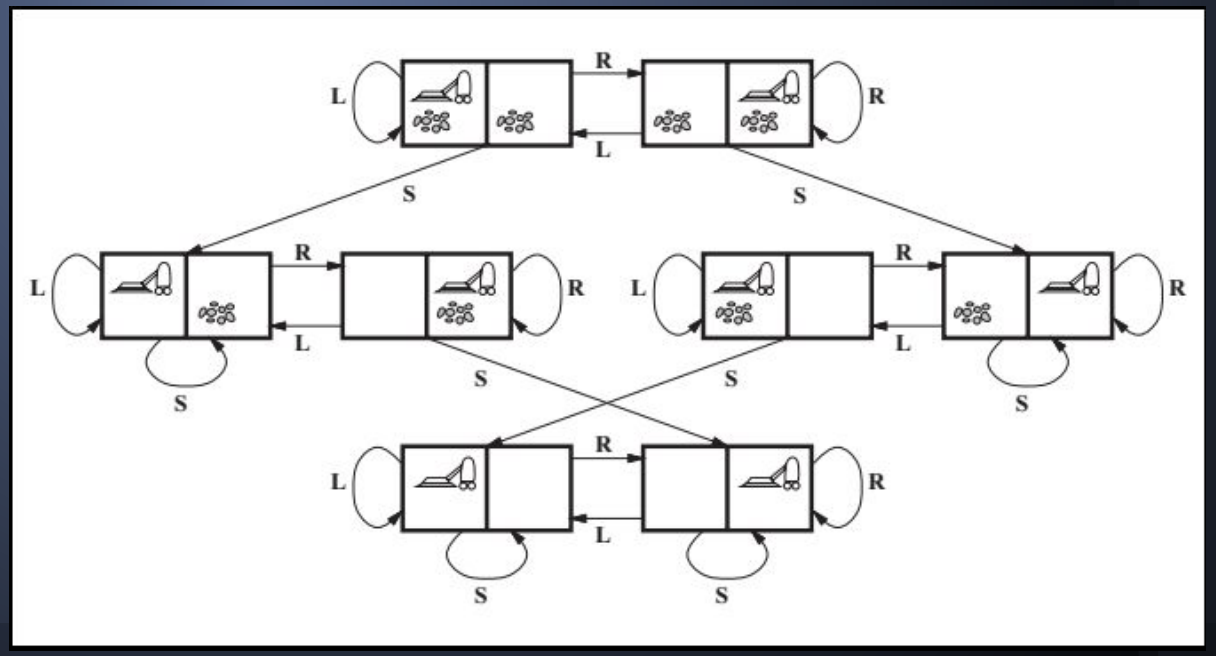


Ejemplo

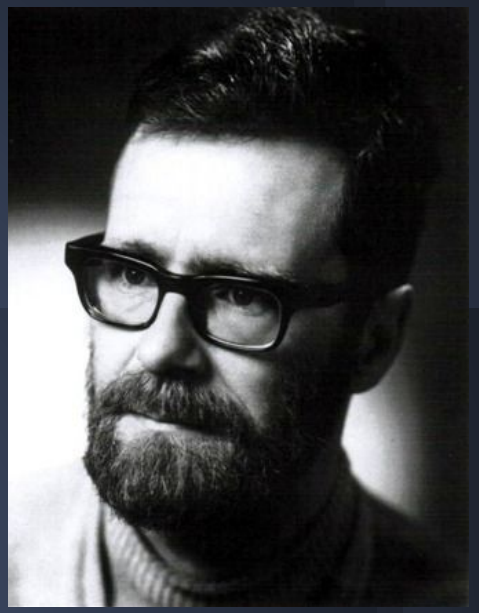
- Para formular el dominio, hay que definir muy bien el problema
  - Configuraciones, si hay N casillas, hay N configuraciones posibles del robot x 2^N del polvo
    - Para N=2, son 8 configuraciones distintas
    - La *inicial* será el robot en A, y polvo en A y B
  - Operadores hay 3, *izquierda*, *derecha* y *aspirar*
  - Modelo de transición, es obvio (moverse hacia fuera o aspirar sin polvo *no tiene efecto alguno*)
  - Prueba de objetivo, comprobar que no hay polvo en ninguna de las N casillas (este es nuestro concepto de “tener la casa limpia”)

Ejemplo

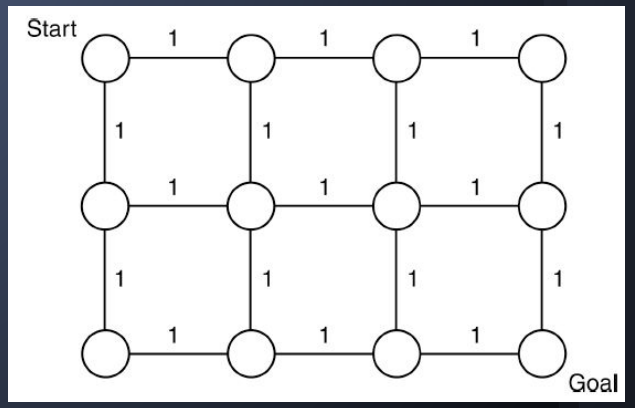
- *Coste del camino*, es habitual considerar que cada transición cuesta 1 y así el coste del camino coincidirá con el número de *pasos* del mismo
- En este ejemplo hasta es posible visualizar el espacio de búsqueda *completo*
  - A menudo suelen ser inabarcables, ¡incluso *infinitos*!



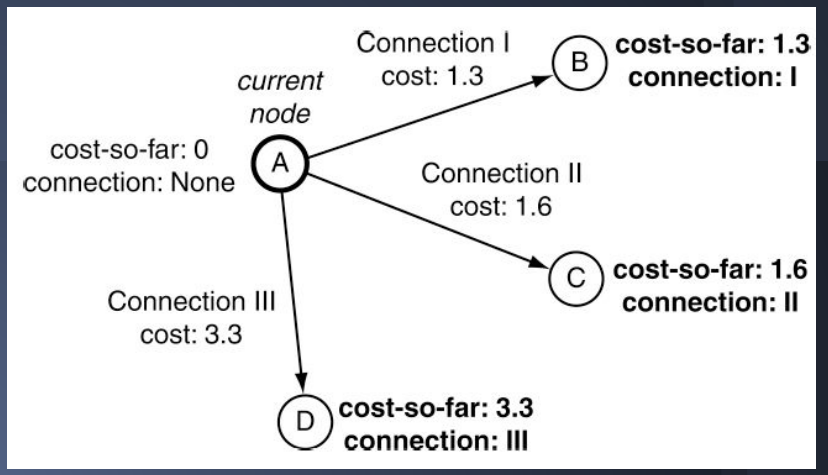
Algoritmo de Dijkstra



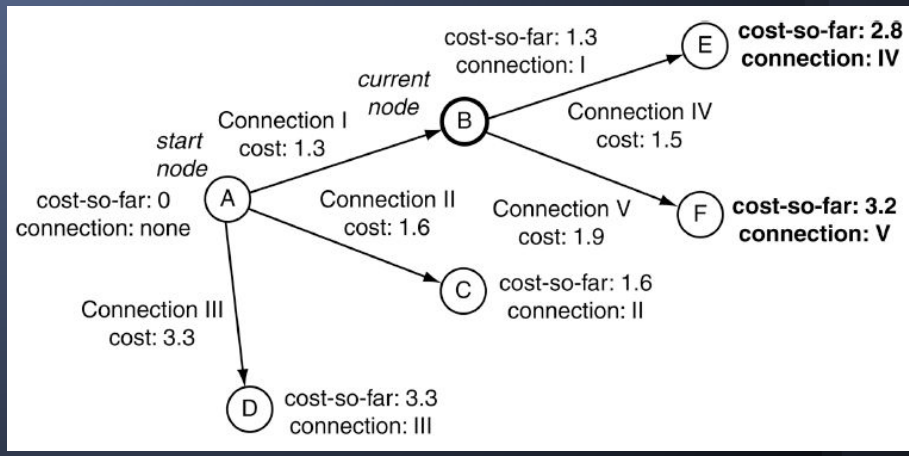
- Busca todos los caminos más cortos desde un nodo inicial a *todos los nodos destino*
  - Se usa en Táctica y Estrategia, y es la base de "fuerza bruta" de otros algoritmos más prácticos que se usan en Navegación, que sólo buscan el camino más corto entre 2 nodos
  - Si existen varios caminos mínimos entre dos nodos, nos *da igual* el que nos devuelva el algoritmo



Explicación



- Recibe un destino y *explora*, dejando rastro para luego reconstruir el camino más corto (solución)
- En cada iteración del algoritmo
  - Considera un nodo como actual, con todas sus conexiones, guardando referencias a todos los nodos a los que se llega (y el coste de llegar), para siguientes iteraciones

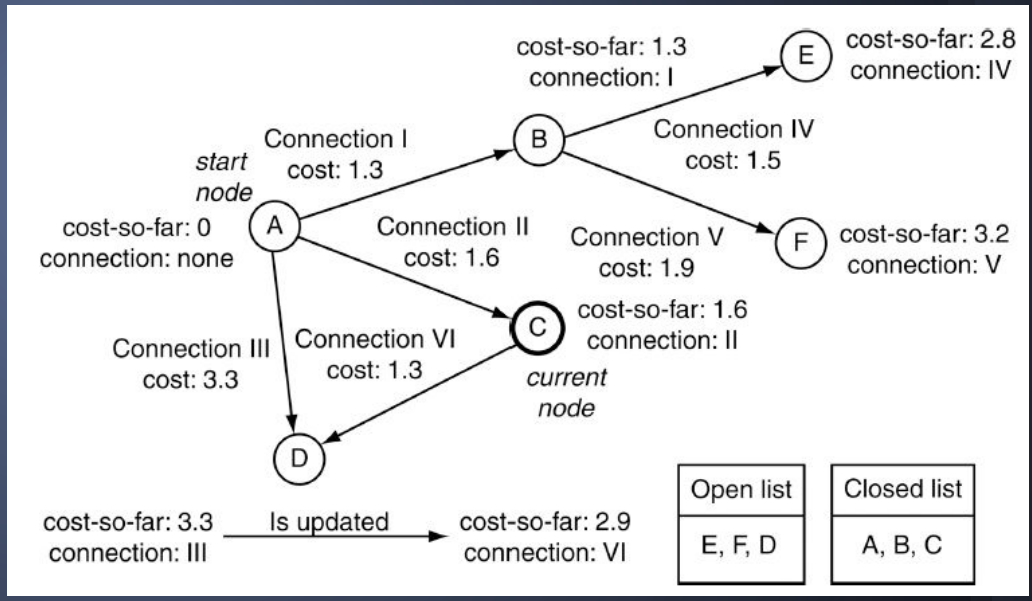


Explicación

- Usa 2 listas, una para nodos abiertos y otra para nodos cerrados
  - La lista abierta guarda todas las referencias a los nodos que ha visitado, pero sobre los que no ha iterado aún. Empieza con el nodo inicial (coste 0)
  - La lista cerrada contiene las referencias a los nodos sobre los que ya se ha iterado. Empieza vacía (∅)
  - Los nodos que no están en ninguna de las dos listas son nodos no visitados del grafo
- En cada iteración, se elige el nodo abierto con menor "coste hasta el momento"
  - Y tras procesarlo, se manda a la lista cerrada

Explicación

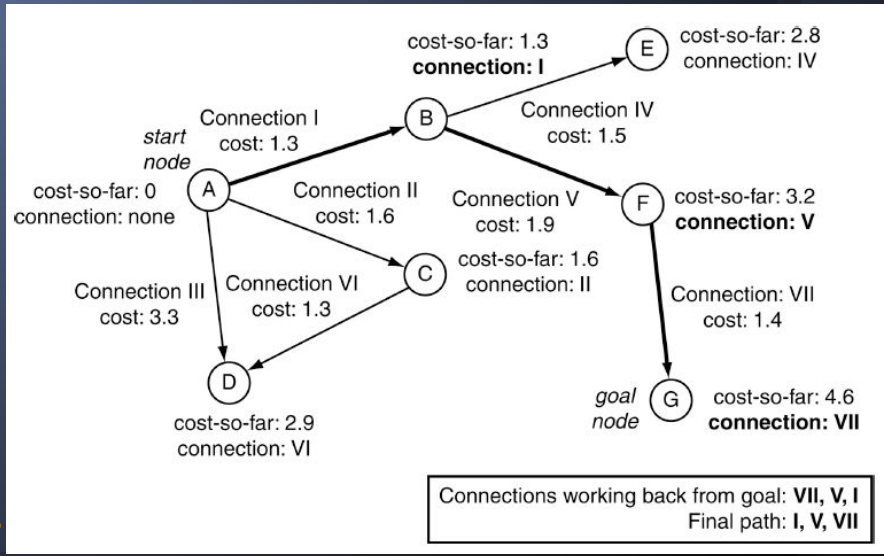
- Al llegar a un nodo ya visitado
  - Si está *cerrado*, lo podemos ignorar (no ocurrirá)
  - Si está *abierto*, calculamos el coste hasta el momento y sólo si es menor al que tiene el nodo encontrado, sobreescribimos su conexión y su coste… se añadiría a la lista abierta, eliminándola de la cerrada si hace falta



Explicación

* ¡Ojo, no se termina hasta que no se selecciona de la lista abierta!

- Esto se termina al vaciar la *lista abierta*
  - Se han considerado todos los nodos alcanzables del grafo desde el nodo inicial y están en *lista cerrada*
  - Lo correcto es terminar* *sólo* cuando el nodo objetivo es el más pequeño de la lista de nodos abiertos (menor "coste hasta el momento")
- Se reconstruye la solución (paso a paso)
  - Al final son las conexiones de atrás al inicio, y luego se *invierte*



Participación

tiny.cc/IAV

- ¿Qué guarda la lista abierta de Dijkstra?
  A. Todos los nodos iterados
  B. Todos los nodos visitados pero no iterados
  C. Todos los nodos no visitados
  D. Todos los nodos no visitados ni iterados
- Desarrolla tu respuesta (en texto libre)

Pseudocódigo

- Versión para buscar *sólo un camino*

```
function pathfindDijkstra(graph: Graph,
                          start: Node,
                          end: Node) -> Connection[]:
    # This structure is used to keep track of the information we need
    # for each node.
    class NodeRecord:
        node: Node
        connection: Connection
        costSoFar: float

    # Initialize the record for the start node.
    startRecord = new NodeRecord()
    startRecord.node = start
    startRecord.connection = null
    startRecord.costSoFar = 0

    # Initialize the open and closed lists.
    open = new PathfindingList()
    open += startRecord
    closed = new PathfindingList()

    # Iterate through processing each node.
    while length(open) > 0:
        # Find the smallest element in the open list.
        current: NodeRecord = open.smallestElement()

        # If it is the goal node, then terminate.
        if current.node == goal:
            break
```

Pseudocódigo

```
31          # Otherwise get its outgoing connections.
32          connections = graph.getConnections(current)
33
34          # Loop through each connection in turn.
35          for connection in connections:
36              # Get the cost estimate for the end node.
37              endNode = connection.getToNode()
38              endNodeCost = current.costSoFar + connection.getCost()
39
40              # Skip if the node is closed.
41              if closed.contains(endNode):
42                  continue
43
44              # .. or if it is open and we've found a worse route.
45              else if open.contains(endNode):
46                  # Here we find the record in the open list
47                  # corresponding to the endNode.
48                  endNodeRecord = open.find(endNode)
49                  if endNodeRecord.cost <= endNodeCost:
50                      continue
```

Pseudocódigo

```
52              # Otherwise we know we've got an unvisited node, so make a
53              # record for it.
54              else:
55                  endNodeRecord = new NodeRecord()
56                  endNodeRecord.node = endNode
57
58              # We're here if we need to update the node. Update the
59              # cost and connection.
60              endNodeRecord.cost = endNodeCost
61              endNodeRecord.connection = connection
62
63              # And add it to the open list.
64              if not open.contains(endNode):
65                  open += endNodeRecord
66
67          # We've finished looking at the connections for  the current
68          # node, so add it to the closed list and remove it from the
69          # open list.
70          open -= current
71          closed += current
72
73      # We're here if we've either found the goal, or if we've no more
74      # nodes to search, find which.
75      if current.node != goal:
76          # We've run out of nodes without finding the goal, so there's
77          # no solution.
78          return null
79
80      else:
81          # Compile the list of connections in the path.
82          path = []
```

Pseudocódigo

```
84          # Work back along the path, accumulating connections.
85          while current.node != start:
86              path += current.connection
87              current = current.connection.getFromNode( )
88
89          # Reverse the path, and return it.
90          return reverse(path)
```

* La lista contiene estructuras *NodeRecord*, con estos métodos:
smallestElement, devuelve la estructura con el menos coste hasta el momento
contains indica si la lista contiene el NodeRecord de ese nodo
find devuelve el NodeRecord de ese nodo

* **reverse** da la vuelta a la lista

Estructuras de datos

- Se usa una lista simple para recopilar la solución, cuya implementación da igual
- Las listas abierta y cerrada son críticas, tienen 4 operaciones que optimizar en la implementación
  - Añadir un nodo (+=)
  - Eliminar un nodo (-=)
  - Encontrar el nodo más pequeño (*smallestElement*)
  - Encontrar un nodo en particular (*contains* y *find*)

Estructuras de datos

- Al implementar *Connection*, procura evitar hacer cualquier tipo de procesamiento

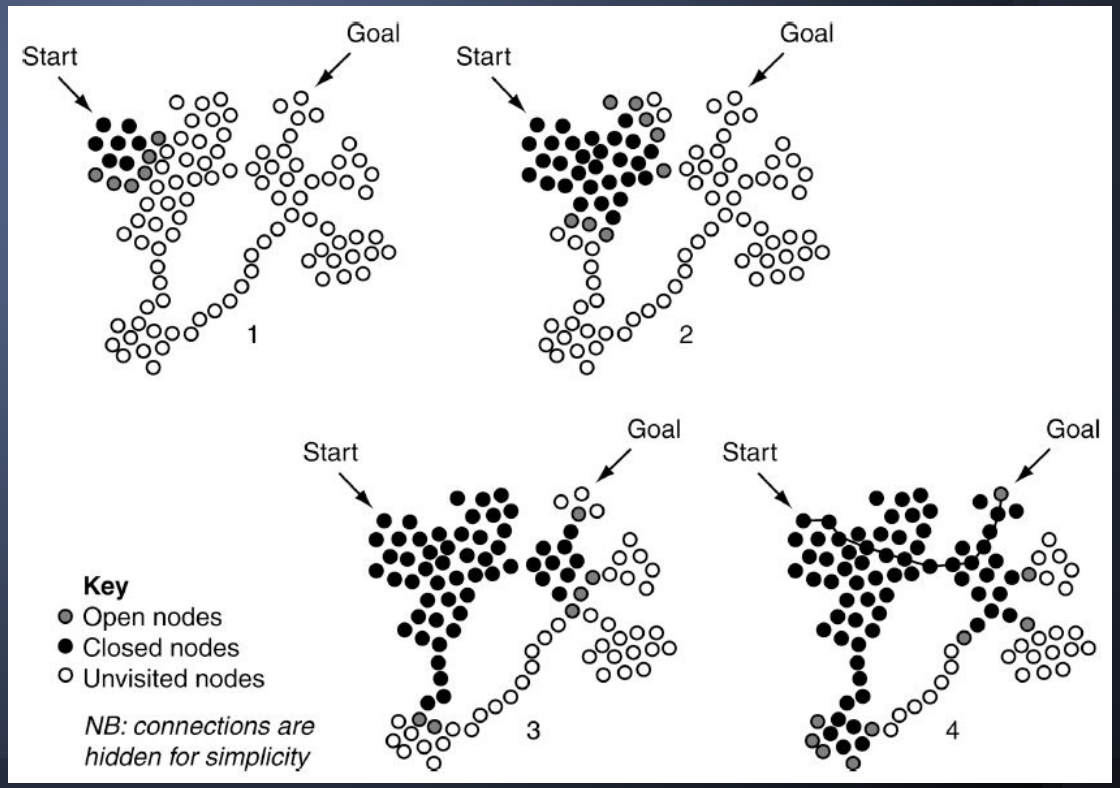
```
1  class Connection:
2      cost: float
3      fromNode: Node
4      toNode: Node
5
6      function getCost( ) -> float:
7          return cost
8
9      function getFromNode( ) -> Node:
10         return fromNode
11
12     function getToNode( ) -> Node:
13         return toNode
```

* Estos valores se *precalculan* en la generación: cuando convertimos la geometría del entorno a nodos

Tiempo = O(n*m)
Espacio ≤ O(n*m)
donde *n* son los nodos que están más cerca que el destino y *m* la media de conexiones salientes de un nodo

Debilidades

- Sólo es práctico para encontrar el camino más corto a *todos* los posibles destinos
  - Fíjate que la frontera de la búsqueda son los nodos en la lista abierta



* La línea negra marca la solución (camino más corto)

* Los nodos visitados que no se usan en la solución, son el **relleno**… generalmente demasiado relleno

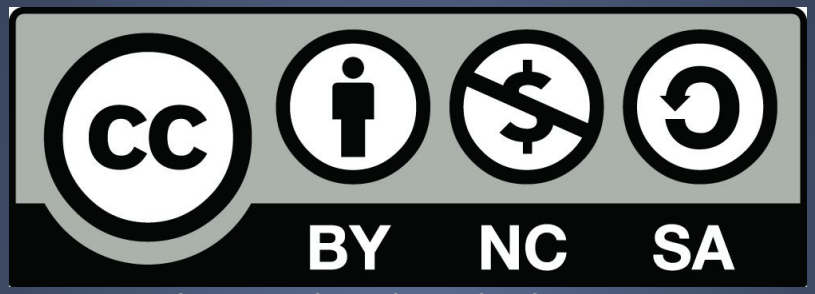
Resumen

- Como hito histórico en la resolución de problemas en el espacio de estados tenemos al Ajedrez
- El “juego de la aspiradora” es un ejemplo sencillo de esta forma de modelar
- Explicamos el algoritmo de Dijkstra, dando el pseudocódigo recomendado y comentarios a la estructura de datos
- Su debilidad es que no es práctico para dar con el camino más corto a un único destino

Más información

- Millington, I.: Artificial Intelligence for Games. CRC Press, 3rd Edition (2019)

Críticas, dudas, sugerencias...



Federico Peinado (2019-2020)

www.federicopeinado.es