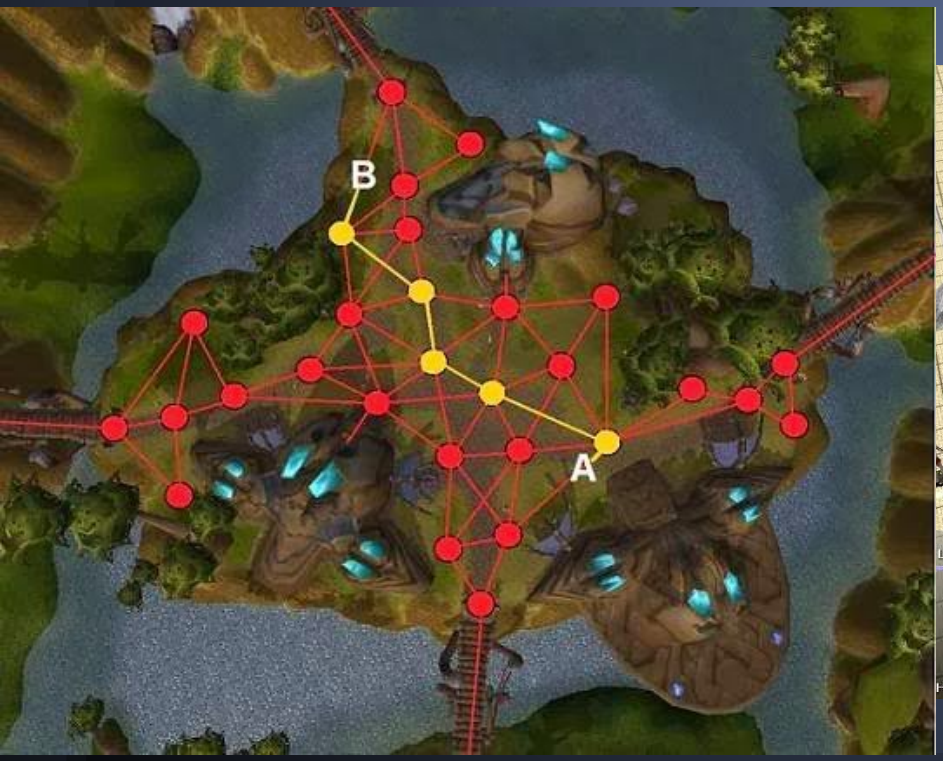
Inteligencia Artificial para Videojuegos

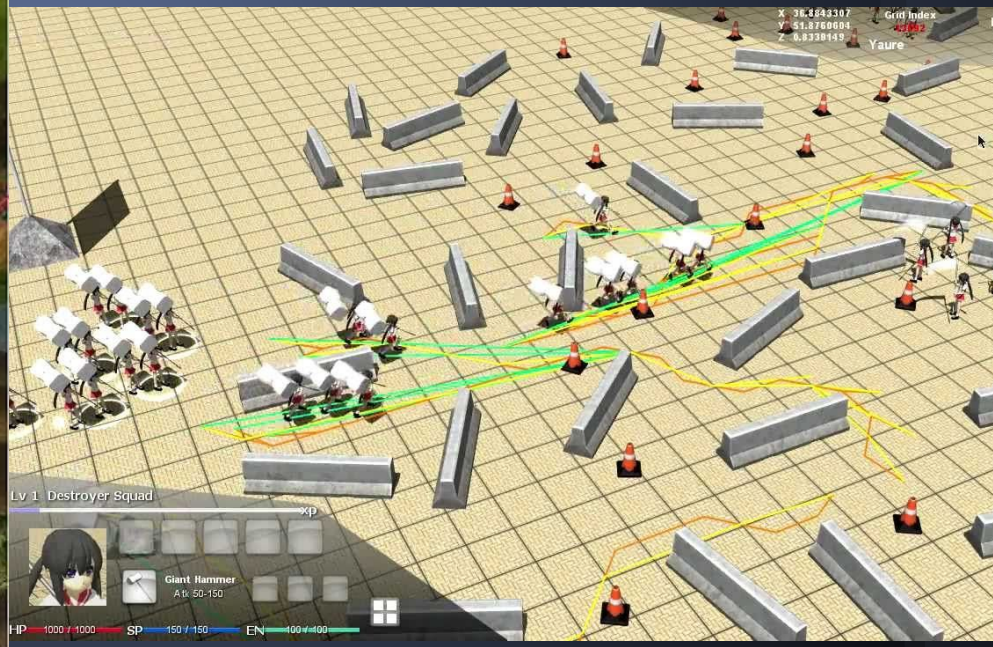
Navegación

Búsqueda de caminos usando estrategias informadas con heurísticas

Motivación

- Prácticamente desde los 90 *todo videojuego* donde se buscan caminos óptimos, usa A*





Motivación

- ¡La información es poder!
- Se puede resolver un problema *cuando se sabe algo* sobre la solución
  - Como mínimo, *reconocer la llegada* a un nodo destino (= estrategias no informadas)
  - Pero lo ideal es *poder estimar la cercanía* del nodo actual al nodo destino (= estrategias informadas)

Puntos clave

- Hitos históricos
- Algoritmo A*
  - Pseudocódigo
- Heurísticas
  - Pseudocódigo
- Mejoras al algoritmo A*
  - Búsqueda jerárquica de caminos
  - Búsqueda parcial de caminos
  - Búsqueda de caminos en tiempo contínuo

Hitos históricos

- A* se publicó en 1968
- Posiblemente en los años 80 *ya se utilizó* en videojuegos

https://github.com/luzbel/vigasocosdl-la-abadia-del-crimen/blob/master/abadia.asm

```
; ---------- algoritmo de alto nivel para la búsqueda de caminos entre 2 posiciones ----------

; algoritmo de búsqueda de caminos entre 2 puntos
; iy apunta a los datos del personaje que busca a otro
; ix apunta a la posición del personaje/objeto que se busca
098A: 3E FE       ld   a,$FE
098C: 32 B6 2D    ld   ($2DB6),a        ; indica que no se ha podido buscar un camino
098F: 3E 00       ld   a,$00            ; modificado desde el bucle principal del juego con la animación de guillermo
0991: E6 01       and  $01
0993: C0          ret  nz                       ; si está en la mitad de la animación, sale

0994: 3A A9 2D    ld   a,($2DA9)        ; si en esta iteración ya se ha encontrado un camino, sale (sólo se busca un camino por iteración)
0997: A7          and  a                        ; si ya se ha encontrado un camino, sale
0998: C0          ret  nz

0999: 3E 76       ld   a,$76
099B: 32 A4 48    ld   ($48A4),a        ; indica que hay que buscar una posición con el bit 6 en el algoritmo de búsqueda de caminos
099E: AF          xor  a
099F: 32 B6 2D    ld   ($2DB6),a        ; indica que de momento no se ha encontrado un camino
09A2: FD 7E 04    ld   a,(iy+$04)       ; obtiene la altura del personaje que busca a otro
09A5: CD 73 24    call $2473            ; dependiendo de la altura, devuelve la altura base de la planta en b
09A8: 58          ld   e,b                      ; e = altura base de la planta del personaje que busca a otro
09A9: DD 7E 02    ld   a,(ix+$02)       ; obtiene la altura del personaje buscado
09AC: E6 3F       and  $3F
09AE: CD 73 24    call $2473            ; dependiendo de la altura, devuelve la altura base de la planta en b

09B1: 7B          ld   a,e                      ; a = altura base de la planta del personaje que busca a otro
09B2: 21 CD 05    ld   hl,$05CD         ; apunta a tabla con las conexiones de las habitaciones (planta baja)
09B5: A7          and  a
09B6: 28 0A       jr   z,$09C2          ; si el personaje que busca a otro está en la planta baja, salta
09B8: 21 7D 06    ld   hl,$067D         ; apunta a tabla con las conexiones de las habitaciones (primera planta)
09BB: FE 08       cp   $08
09BD: 28 03       jr   z,$09C2          ; si el personaje que busca a otro está en la primera planta, salta
09BF: 21 85 06    ld   hl,$0685         ; apunta a tabla con las conexiones de las habitaciones (segunda planta)

09C2: 22 0A 44    ld   ($440A),hl       ; guarda la dirección de la tabla
09C5: B8          cp   b
09C6: 28 6F       jr   z,$0A37          ; si están en la misma planta, salta

; aquí llega si los personajes no están en la misma planta
09C8: 3E 10       ld   a,$10
09CA: 38 02       jr   c,$09CE          ; si el personaje que busca a otro está en una planta inferior al personaje de destino, a = 0x10
09CC: 3E 20       ld   a,$20            ; en otro caso, a = 0x20

09CE: 4F          ld   c,a                      ; c = indicador de si hay que subir o bajar planta
```

http://cubeman.org/arcade-source/pacman.asm

```
        ;; 4e66 last state coin inputs shifted left by 1
        ;; 4e6b #coins per #credits
        ;; 4e6c #left over coins (partial credits)
        ;; 4e6d #credits per #coins
        ;; 4e6e #credits
        ;; 4e6f #lives per game

        ;; 4e80-4e83 P1 score
        ;; 4e84-4e87 P2 score
        ;; 4e88-4e8b High score

        ;; 4370 #players (0=1, 1=2)

       Starting address: 0
  Ending address: 16383
    Output file: (none)
Pass 1 of 1
0000  f3        di                      ; Disable interrupts
0001  3e3f      ld      a,#3f
0003  ed47      ld      i,a             ; Interrupt page = 0x3f
0005  c30b23    jp      #230b           ; Run startup tests

        ;; Fill "hl" to "hl+b" with "a"
0008  77        ld      (hl),a
0009  23        inc     hl
000a  10fc      djnz    #0008           ; (-4)
000c  c9        ret
000d  c30e07    jp      #070e

        ;; hl = hl + a, (hl) -> a
0010  85        add     a,l
```

Pac-Man (1980)?

La Abadía del Crimen (1986)?

Hitos históricos

- Fue en los años 90 cuando su uso se extendió tanto
- Es la época de los juegos RTS como Warcraft (1994-...) o Age of Empires (1996-)

Algoritmo A* (A-Estrella)

A-STAR

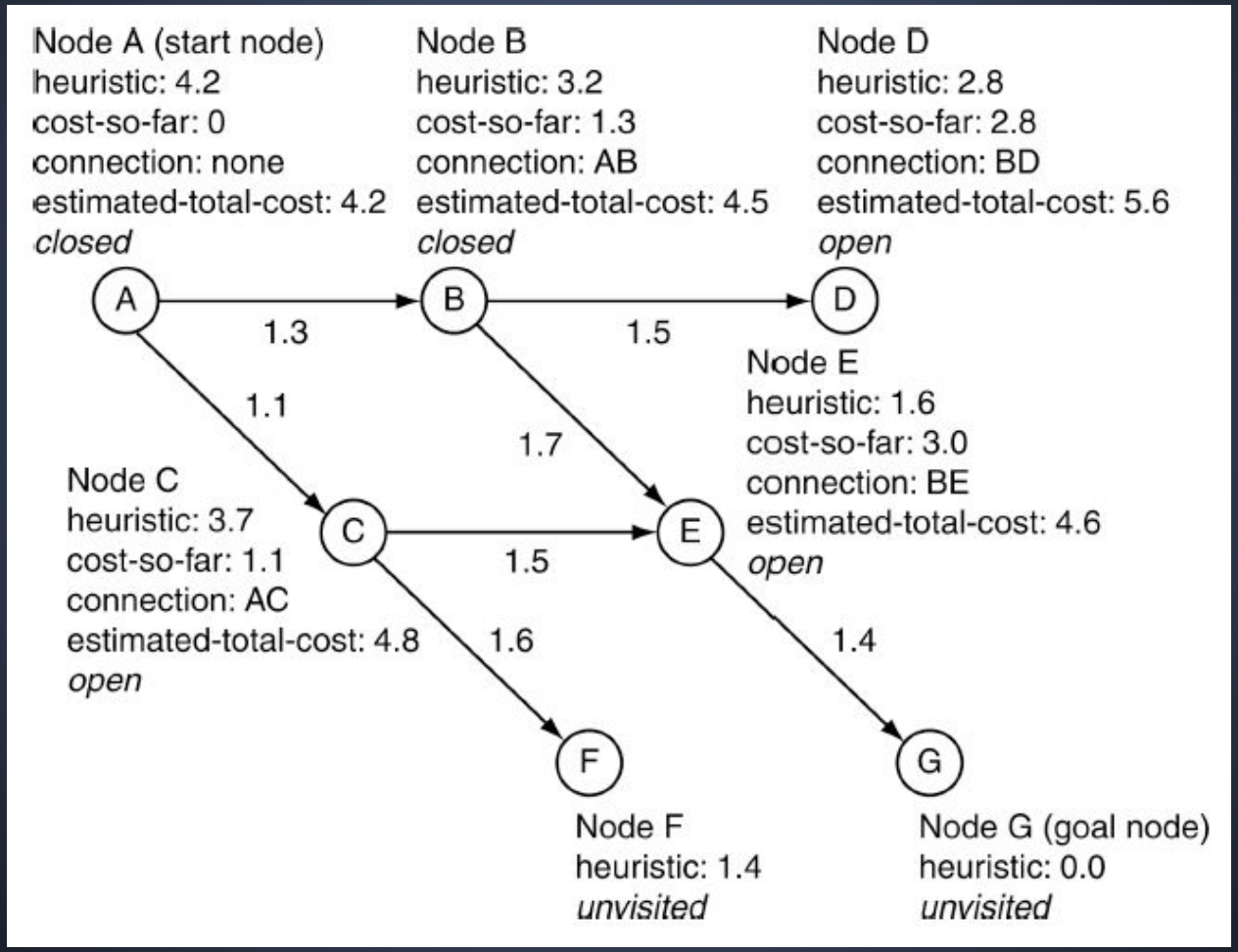
- Es sencillo, completo, óptimo, tiene *eficiencia óptima* y posibles variantes
  - Busca un camino mínimo *entre A y B*, en un grafo pesado no negativo y dirigido, no como Dijkstra
  - Como otros algoritmos, es útil para resolver muchos otros problemas, no sólo Navegación
  - No se interesa por el nodo que acumula menor coste hasta el momento, sino el que acumula poco coste hasta el momento y estima también poco coste hasta el nodo destino
  - Esa *estimación* la da una función heurística, de la que depende *mucho* la eficiencia (*si es mala, mejor Dijkstra*)

Algoritmo A* (A-Estrella)

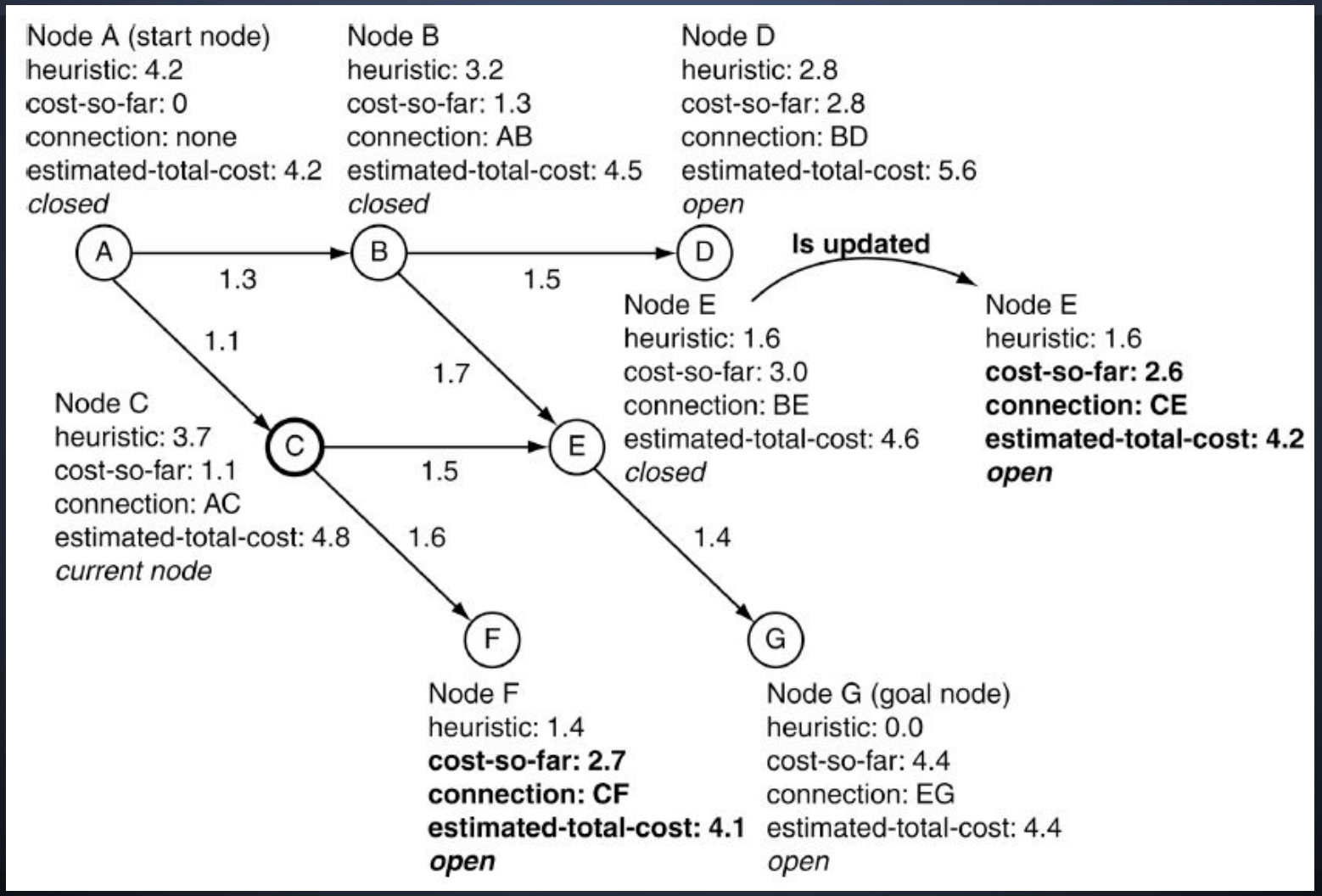
- Se trata de un algoritmo iterativo
  - En cada iteración se exploran las conexiones del nodo actual y los registros sus *nodos hijo* se guarda el coste hasta el momento, como en Dijkstra
  - Pero además a ese coste real *se le suma* la estimación de la función heurística y se guarda el resultado, la estimación del coste total del mejor camino *origen-destino*, que cruza por ese nodo

$$f(nodo) = g(nodo) + h(nodo)$$

Algoritmo A* (A-Estrella)



Algoritmo A* (A-Estrella)



Pseudocódigo

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

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

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

Pseudocódigo

```
23      open += startRecord
24      closed = new PathfindingList()
25
26      # Iterate through processing each node.
27      while length(open) > 0:
28          # Find the smallest element in the open list (using the
29          # estimatedTotalCost).
30          current = open.smallestElement()
31
32          # If it is the goal node, then terminate.
33          if current.node == goal:
34              break
35
36          # Otherwise get its outgoing connections.
37          connections = graph.getConnections(current)
38
39          # Loop through each connection in turn.
40          for connection in connections:
41              # Get the cost estimate for the end node.
42              endNode = connection.getToNode()
43              endNodeCost = current.costSoFar + connection.getCost()
44
45              # If the node is closed we may have to skip, or remove it
46              # from the closed list.
47              if closed.contains(endNode):
48                  # Here we find the record in the closed list
49                  # corresponding to the endNode.
50                  endNodeRecord = closed.find(endNode)
```

Pseudocódigo

```
50                endNodeRecord = closed.find(endNode)
51
52                # If we didn't find a shorter route, skip.
53                if endNodeRecord.costSoFar <= endNodeCost:
54                    continue
55
56                # Otherwise remove it from the closed list.
57                closed -= endNodeRecord
58
59                # We can use the node's old cost values to calculate
60                # its heuristic without calling the possibly expensive
61                # heuristic function.
62                endNodeHeuristic = endNodeRecord.estimatedTotalCost -
63                                   endNodeRecord.costSoFar
64
65            # Skip if the node is open and we've not found a better
66            # route.
67            else if open.contains(endNode):
68                # Here we find the record in the open list
69                # corresponding to the endNode.
```

Pseudocódigo

```
70                  endNodeRecord = open.find(endNode)
71
72                  # If our route is no better, then skip.
73                  if endNodeRecord.costSoFar <= endNodeCost:
74                      continue
75
76                  # Again, we can calculate its heuristic.
77                  endNodeHeuristic = endNodeRecord.cost -
78                                     endNodeRecord.costSoFar
79
80              # Otherwise we know we've got an unvisited node, so make a
81              # record for it.
82              else:
83                  endNodeRecord = new NodeRecord( )
84                  endNodeRecord.node = endNode
85
86                  # We'll need to calculate the heuristic value using
87                  # the function, since we don't have an existing record
88                  # to use.
89                  endNodeHeuristic = heuristic.estimate(endNode)
90
91              # We're here if we need to update the node. Update the
92              # cost, estimate and connection.
93              endNodeRecord.cost = endNodeCost
94              endNodeRecord.connection = connection
95              endNodeRecord.estimatedTotalCost = endNodeCost +
                    endNodeHeuristic
```

Pseudocódigo

```
97              # And add it to the open list.
98              if not open.contains(endNode):
99                  open += endNodeRecord
100
101         # We've finished looking at the connections for the current
102         # node, so add it to the closed list and remove it from the
103         # open list.
104         open -= current
105         closed += current
106
107     # We're here if we've either found the goal, or if we've no more
108     # nodes to search, find which.
109     if current.node != goal:
110         # We've run out of nodes without finding the goal, so there's
111         # no solution.
112         return null
113
114     else:
115         # Compile the list of connections in the path.
116         path = []
117
118         # Work back along the path, accumulating connections.
119         while current.node != start:
120             path += current.connection
121             current = current.connection.getFromNode( )
122
123         # Reverse the path, and return it.
124         return reverse(path)
```

Tiempo = O(l*m)

Espacio ≤ O(l*m)

donde *n* son los nodos con coste estimado total menor que el destino, y *m* la media de conexiones salientes de un nodo

Participación

tiny.cc/IAV

- ¿Qué es coste total estimado de un nodo X?
  A. Estimación del coste *origen-destino*, pasando por X
  B. Coste real de *origen* a *destino*, pasando por X
  C. Estimación *origen-X*, más coste real *X-destino*
  D. Estimación *X-destino*, más coste real de *origen-X*
- Desarrolla tu respuesta (en texto libre)

Heurísticas

- Función heurística:

```
1  class Heuristic:
2      # An estimated cost to reach the goal from the given node.
3      function estimate(node: Node) -> float
```

o más concretamente:

```
1  class Heuristic:
2      # Stores the goal node that this heuristic is estimating for.
3      goalNode: Node
4
5      # Estimated cost to reach the stored goal from the given node.
6      function estimate(fromNode: Node) -> float:
7          return estimate(fromNode, goalNode)
8
9      # Estimated cost to move between any two nodes.
10     function estimate(fromNode: Node, toNode: Node) -> float
```

*Así se usa luego: `pathfindAStar(graph, start, end, new Heuristic(end))`

Heurísticas

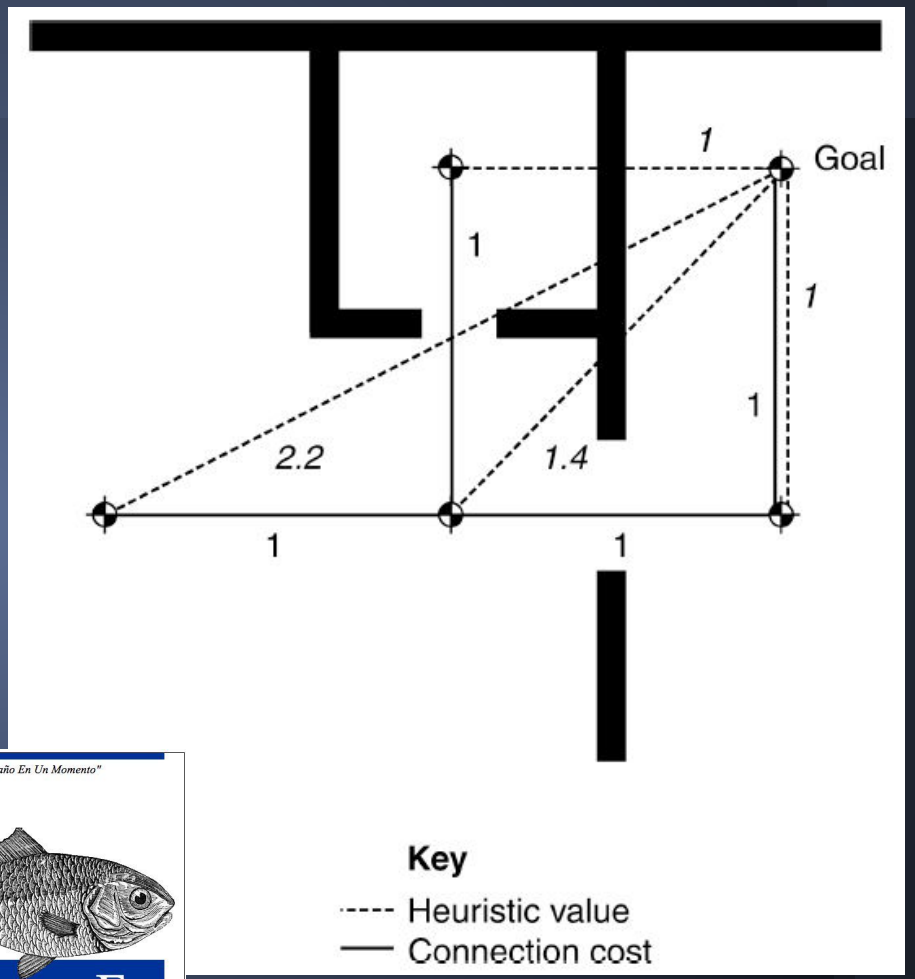
- El algoritmo de Dijkstra puede verse como un caso particular del A*, donde la función heurística siempre *devuelve 0*
- Es completo y óptimo sólo si la heurística es *admisible* (*consistente*, al ser sobre grafos)
  - Admisible = No *sobreestima* el coste óptimo real hasta el nodo destino
    h(nodo) ≤ coste óptimo hasta el destino
    con lo que f(nodo) tampoco *sobreestimará* el coste óptimo real de la solución (camino completo)

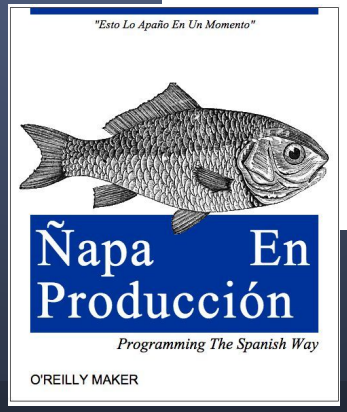
Heurísticas

- Consistente/Monótona = Nunca estima '*a mejor*'
  h(nodo) ≤ coste de expandirlo + h(nodo hijo)
  con lo que f(nodo) sigue un camino *no decreciente*
- Consistencia → Admisibilidad *¡Atención!*
- A* es también óptimamente eficiente, porque expande los *menos nodos posibles (el menor "relleno" posible)*
  - Cuanto más *precisa* sea la estimación que nos da, menor será el coste del algoritmo (menos nodos)

Heurísticas

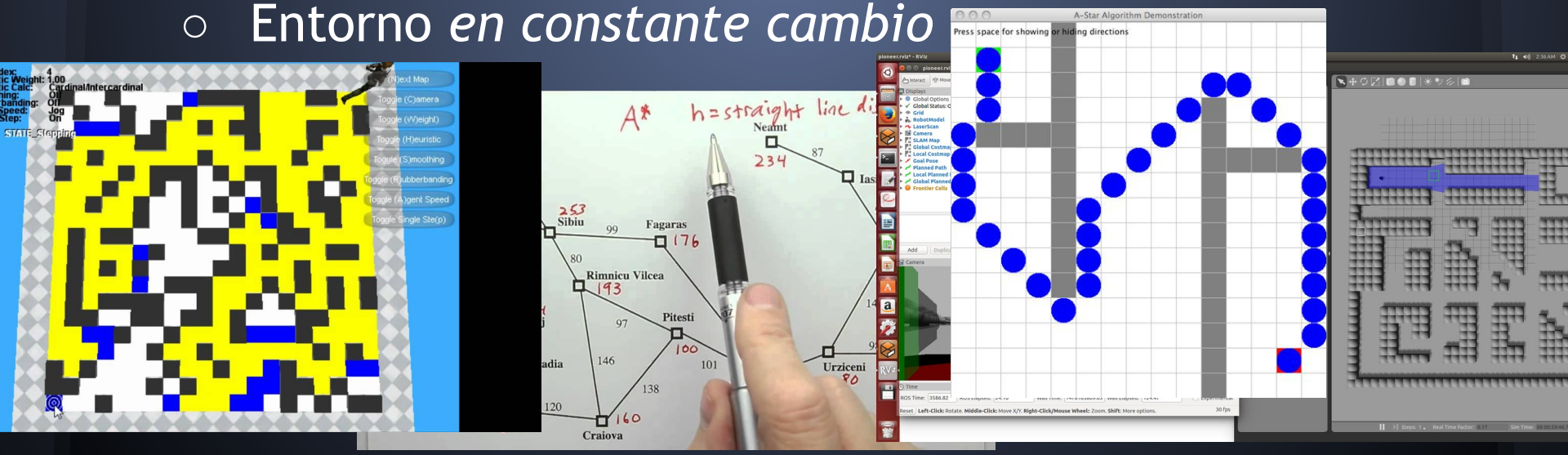
- Heurísticas típicas son la distancia Euclídea o Manhattan
- A veces nos apaña una heurística que no es admisible pero sí es muy *precisa*
  - No encontrará caminos óptimos, *¡pero casi!*





Mejoras al algoritmo A*

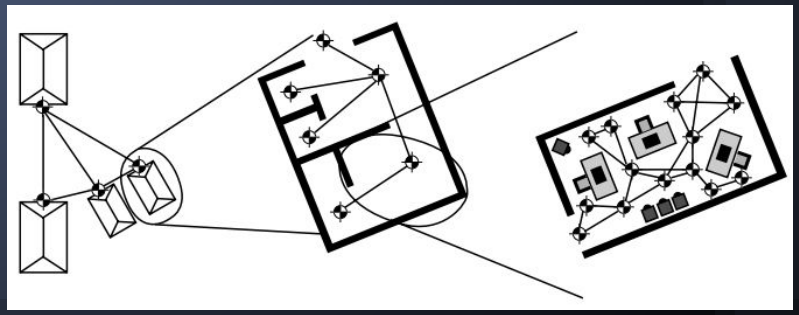
- A* es muy eficiente con buena heurística
  - Decenas de miles de nodos, con un código simple
- ¿Pero qué pasa cuando *surgen problemas*?
  - Millones de nodos, como en un MMOG
  - Cientos de agentes a la vez, como en un RTS
  - Entorno *en constante cambio*



Búsqueda jerárquica de caminos

HIERARCHICAL PATHFINDING

- Simplifica la búsqueda trabajando a varios niveles (de nuevo, *agrupando* nodos)
  - Aunque A* puede buscar caminos de *decenas de miles de nodos en un sólo frame*, y es fácilmente interrumpible, sigue teniendo sus límites
  - Las agrupaciones se conectan usando heurísticas
    - Distancia mínima, máxima o mínima promedio
  - Los caminos a *niveles distintos* al actual se pueden guardar en memoria, y los caminos de partes prefabricadas del entorno se reutilizan



Búsqueda parcial de caminos

PARTIAL PATHFINDING

- A* puede tener varios destinos, aunque suele *evitarse* porque no funciona muy bien
  - Mejor se decide seguir uno y se usa el A* normal
- Si el entorno cambia mucho, no replanificar
  - Se busca el camino *incrementalmente* con algoritmos como D* (A* dinámico)
- Además de variantes restringidas en memoria como IDA*, MA* o SMA*, hay otras que gastan mucha porque se guardan partes reutilizables de la búsqueda

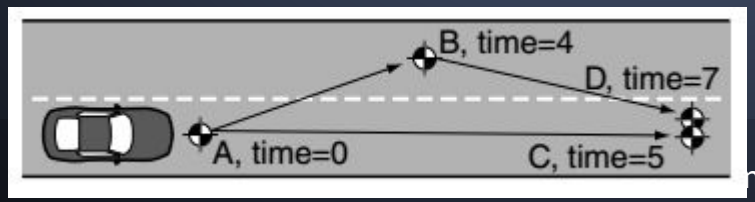
  LIFELONG PLANNING A-STAR SEARCH (LPA*)
  - A veces en una flota de planificadores disponibles

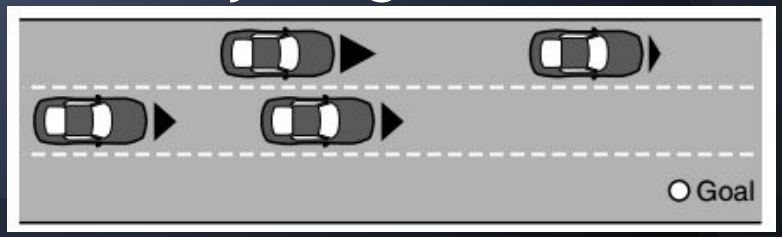
    POOLING PLANNERS

Búsqueda de caminos en tiempo contínuo

CONTINUOS TIME PATHFINDING

- D* funciona mejor *cuanto menos y más concretos* sean los cambios en el entorno
  - ¿Y si todo el entorno está *en cambio continuo*?
  - Ej. Búsqueda de caminos para vehículos
- Buscar en tiempo contínuo supone dejar de considerar *las posiciones* como nodos
  - Ahora serán los *estados del entorno* junto al *tiempo* que predecimos tardar en alcanzar dichos estados
  - Primero se crea este grafo dinámico y luego se usa *SMA* u otro* para resolverlo





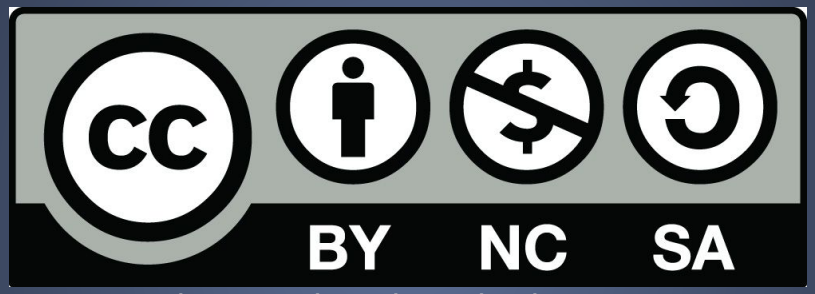
Resumen

- El algoritmo A* es completo, óptimo, y muy eficiente si se usa una buena heurística
- La heurística debe ser admisible y, en grafos, además debe ser consistente
- El algoritmo A* admite mejoras, como la búsqueda jerárquica de caminos, que agrupa nodos y trabaja a varios niveles
- También hay variantes para búsqueda parcial de caminos y búsqueda de caminos en tiempo contínuo

Más información

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

Críticas, dudas, sugerencias...



Federico Peinado (2019-2020)

www.federicopeinado.es