UNIVERSIDAD
DE GRANADA

Técnicas de los Sistemas Inteligentes
grado en ingeniería informática

PRÁCTICA 1

Técnicas de Búsqueda

Autores
Vladislav Nikolov Vasilev
Carlos Núñez Molina

Rama
Computación y Sistemas Inteligentes

Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación

Curso 2018-2019

Índice

1. Descripción General de la Solución

El aspecto fundamental de la práctica es cómo elegir qué gemas coger y en qué orden. Hay en total 23 gemas por nivel, de las cuales solo se necesitan coger 9. Si se hacen los cálculos, hay $\binom{23}{9} \cdot 9! = 296541907200$ combinaciones posibles. Este número es inabarcable para el A*, sin importar la estrategia usada, por lo que no podemos usarlo para que resuelva el nivel desde cero: hace falta simplificar el problema.

Para reducir el número de posibilidades se ha usado una estrategia de *clustering*, técnica de aprendizaje no supervisado. La heurística detrás de esto es la siguiente: si nos encontramos en un *cluster* (grupo) de gemas, al estar estas gemas todas juntas, generalmente será una buena idea (un buen plan) coger todas las gemas del *cluster* antes de irse a otro. Por tanto, hemos transformado el problema de qué gemas coger y en qué orden al problema de qué *clusters* de gemas coger y en qué orden. Como el número de *clusters* es mucho menor que el de gemas, este problema sí que es abordable. Para generar los *clusters* se ha usado un algoritmo llamado *DBSCAN*. Su funcionamiento (implementado) es el siguiente: se va iterando por todas las gemas del nivel; si esa gema no pertenece a un *cluster* y no hay otra gema de algún *cluster* cerca suya se crea un nuevo *cluster* y se asigna a él; después se ve qué otras gemas sin *cluster* están cerca de ésta y se asignan al mismo *cluster*. Una gema está cerca de otra si su distancia Manhattan es menor o igual a un parámetro ε del método. En la práctica se ha usado $\varepsilon = 3$, que es el que genera mejores *clusters*. Para elegir el *tour* (camino) a través de los *clusters* se ha usado un simple algoritmo de *Branch&Bound*, que devuelve un camino a través de *clusters* de forma que en total se consigan el número de gemas necesarias para abandonar el nivel, siendo el camino elegido en función de la distancia entre los *clusters* y la "dificultad" de cada *cluster* (el número de rocas, muros y enemigos en el *cluster* y cómo de alejadas están sus gemas).

De esta forma, esta es la estrategia fundamental usada en la resolución de la práctica: agrupar las gemas en *clusters* e ir yendo de un *cluster* a otro hasta tener 9 gemas, en cuyo caso se planifica para abandonar el nivel.

La integración del comportamiento reactivo y deliberativo, a grandes rasgos y en pseudocódigo, es la siguiente:

Algorithm 1 Integración del comportamiento reactivo-deliberativo (I)

```
1: procedure ACT()
2:     if primer_turno then                ▷ Esto se hace en el constructor
3:         crearClusetrsYCircuitos()
4:         cluster_actual ← 0
5:         buscarPlan(cluster_actual)
```

Algorithm 2 Integración del comportamiento reactivo-deliberativo (II)

```
6:      end if
7:              ▷ Plan creado cuando el jugador puede o va a morir en próx. turnos
8:      if plan_no_morir.isEmpty() then
9:          return plan_no_morir.first()
10:     end if
11:     if num_gems ≥ 9 then                    ▷ Dirigirse a la salida si se puede salir
12:         buscarPlanAbandonarNivel()
13:     end if
14:     if busqueda_no_terminada then    ▷ Búsqueda puede tardar múlt. turnos
15:         seguirBuscandoPlan()
16:     end if
17:     if busqueda_terminada and camino_no_encontrado then
18:         hay_que_replanificar ← true
19:         if num_gems < 9 then
20:             removeCluster(cluster_actual)  ▷ Eliminar cluster y recrear circuito
21:             crearClusterYCircuito()              ▷ porque el cluster es inaccesible
22:         end if
23:     end if
24:     if hay_que_replanificar then
25:         buscarPlan()
26:     end if
27:     if busqueda_terminada then
28:         if plan_vacio then
29:             cluster_actual ← cluster_actual + 1
30:             buscarPlan(cluster_actual)
31:         else
32:             accion ← plan.first()
33:         end if
34:     end if
35:                         ▷ Parte reactiva: ver si ejecutar la acción del plan o no
36:     if enemigos_cercanos or muerte_por_roca then
37:         crearPlanNoMorir()
38:         return plan_no_morir.first()
39:     end if
40:     if jugador_choca_con_roca_cayendo then
41:         return quedarse_quieto
42:     end if
43:     return accion
44: end procedure
```

2. Comportamiento Reactivo

3. Comportamiento Deliberativo

Para la realización del comportamiento deliberativo se han implementado tres versiones del A*: una que permite ir de una posición inicial a una final, una que permite ir de una posición inicial a una final recogiendo las gemas de un *cluster* y una parecida a la anterior pero sin posición final. En las tres, aparte de las listas que utiliza el algoritmo, se ha añadido una lista de explorados que contiene los nodos visitados y los expandidos para poder hacer una consulta rápida de qué nuevos nodos expandir y cuáles no. La lista de nodos cerrados no se revisita, ya que la optimalidad no es lo más importante en este caso al estar trabajando a nivel de *cluster* y no de gemas. Como se valora mucho la eficiencia en el tiempo, se ha modificado el A* para que las búsquedas se puedan ejecutar en varios turnos, guardando la información internamente.

Se han usado 2 heurísticas distintas, una para el A* que busca un camino desde una casilla inicial a otra final y otra para el A* que busca un camino que coja todas las gemas de un *cluster*.

- **Heurística camino, *getHeuristicDistance***: Obtiene la longitud del camino (número de casillas de separación) que une ambas casillas, pudiendo atravesar rocas pero no muros. Si la casilla inicial y final difieren en su posición x y su posición y, aumenta en 1 la distancia (ya que el agente tendrá que girar una vez como mínimo para llegar al destino). Debido a que puede atravesar las rocas (a diferencia del agente), esta es una heurística obtenida mediante un modelo relajado, con lo que es admisible y monótona.

- **Heurística gemas, *getHeuristicGems***: Crea un grafo donde las n gemas dadas son los n nodos y escoge los $n-1$ lados más cortos de este grafo. El coste del lado entre la gema a y b se corresponde con el valor de la distancia entre a y b, medida usando $getHeuristicDistance$.

  Esta heurística devuelve la suma de la distancia de la casilla inicial a la gema más cercana a esta, más la suma de los $n-1$ lados más cortos del grafo mencionado anteriormente, más la distancia de ir de la casilla final a su gema más cercana. La heurística sería admisible y monótona si se usara así, pero hemos decidido multiplicar esta suma por $\alpha = 2$, con lo que deja de ser admisible y monótona, a cambio de aproximar mejor la longitud real del camino, lo que hace que el A* tenga que explorar menos estados. Aunque no consiga la solución óptima de esta forma, hemos hecho pruebas y, de media, este camino no tiene más de 5 casillas de diferencia con el óptimo.

A continuación se procede a mostrar el pseudocódigo de la primera versión del A*, sobre la que se comentarán brevemente las otras:

Algorithm 3 Versión del A* para ir de un inicio a un final

```
1: procedure BUSCARPLAN(inicio, fin, casillas_ignorar)
2:     plan ← ε
3:     Inicializar variables y listas
4:     while not encontrado and not lista_abiertos.empty() and not timeout
   do
5:         Comprobar si se ha producido timeout
6:         nodo ← lista_abiertos.getRemoveFirst()
7:         if nodo.posicion() = fin then
8:             encontrado ← true
9:         else vecinos ← obtenerVecinos(nodo.posicion())
10:            for each vecino ∈ vecinos do
11:                if posicionValida(vecino) and vecino ∉ casillas_ignorar then
12:                    Generar costes e informacion para el siguiente nodo
13:                    if vecino produce caída de roca then
14:                        Simular caída de rocas y generar nueva información
15:                    end if
16:                    if vecino ∉ lista_explorados then
17:                        lista_abiertos.add(siguiente nodo)
18:                        lista_explorados.add(siguiente nodo)
19:                    end if
20:                end if
21:            end for
22:        end if
23:        lista_cerrados.addFirst(nodo)
24:    end while
25:    if timeout then
26:        Guardar información
27:        return
28:    end if
29:    if encontrado then
30:        plan ← procesarPlan(lista_cerrados.getFirst())
31:    end if
32:    return plan
33: end procedure
```

Sobre esta implementación de la primera versión se tienen que realizar muy pocas modidicaciones para llegar a las otras versiones. En el caso de la segunda, hay que pasarle una lista de gemas que coger, comprobar en la línea 7 también si se han cogido todas las gemas y usar una u otra heurística al generar un nuevo nodo. En el caso de la tercer versión, respecto a la anterior, no hay que pasarle una posición final y en la línea 7 solo comprobar si se han cogido todas las gemas de la lista.