DOBLE GRADO EN
INGENIERÍA INFORMÁTICA Y MATEMÁTICAS

INTELIGENCIA ARTIFICIAL

PRÁCTICA 2
LOS EXTRAÑOS MUNDOS DE BELKAN

Autor
José Miguel González Cañadas

Profesor
Juan Luis Suárez Díaz

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS
INFORMÁTICA Y DE TELECOMUNICACIÓN

GRANADA, MES DE MAYO DE 2021

General

Si nos fijamos en cómo está estructurado un nodo en un comienzo, vemos que dispone de dos atributos,

```
struct nodo{
    estado st;
    list<Action> secuencia;
};
```

donde

```
struct estado {
    int fila;
    int columna;
    int orientacion;
};
```

Además, observemos en el algoritmo de búsqueda en profundidad proporcionado la manera en la que se inicializa el nodo *current* a *origen*, y la forma en la que se expande en general cada nodo,

```
nodo current;
current.st = origen;
current.secuencia.empty();
...
nodo hijo= current;
hijo.st.orientacion = (hijo.st.orientacion+1)%4;
if (Cerrados.find(hijo.st) == Cerrados.end()){
    hijo.secuencia.push_back(act);
    Abiertos.push(hijo);
}
```

Vemos que *secuencia* es copiado cada vez que queremos expandir un nodo, lo cual es tremendamente ineficiente. Para más *inri*, este atributo es copiado independientemente de que al final se termine expandiendo, por lo que estamos realizando copias inútiles frecuentemente (comprobamos si el nodo está en *Cerrados* después de haber copiado la secuencia).

Una solución inicial a este problema podría ser copiar únicamente en un principio los estados; si finalmente el nodo se llega a expandir, copiaremos entonces la secuencia correspondiente. No obstante, esto únicamente soluciona que ya no realicemos copias inútiles, pero las *útiles* se siguen realizando con normalidad. Esto implica que el algoritmo continúa siendo bastante ineficiente.

Para solucionar este incoveniente de manera definitiva, se decidió eliminar la estructura *nodo*, valiéndonos únicamente a partir de ahora de la estructura *estado*. Además de la fila, columna y orientación, ya no almacenará la secuencia de acciones que llevan hasta él, sino que únicamente contendrá un puntero al mejor padre, de forma que recontruir la secuencia de acciones será simplemente recorrer la secuencia de punteros que nos llevan desde el estado final al inicial.

El coste asociado a esta operación es meramente anecdótico en comparación a las soluciones propuestas con anterioridad.

```
struct estado {
  int fila;
  int columna;
  int orientacion;
  const estado * padre;
};
```

Niveles dos y tres

Para los niveles dos y tres hemos implementado un algoritmo A^* optimizado que nos permitirá llevar a cabo la búsqueda a los objetivos de una manera eficiente.

En primer lugar, la heurística empleada ha sido la distancia Manhattan media a los objetivos todavía no conseguidos (para el nivel tres) y la distancia Manhattan al objetivo (nivel dos).

```

int distManhattan(const estado origen, const estado obj){
    return abs(origen.fila-obj.fila) + abs(origen.columna-obj.
columna);
}

int calcularHeu(const estado & s, list<estado>::const_iterator it){
    int h=0,
        n=0;
    for (int i=0; i<3; i++, ++it){
        if (!s.obj[i]){
            h += distManhattan(s, *it);
            ++n;
        }
    }
    if(n!=0) h/=n;
    return h;
}
```

Como cabe esperar, la estructura *estado* ha sido extendida para estos niveles.

```
struct estado {
  int fila;
  int columna;
  int orientacion;
  bool zap; bool bik;
  bool obj[3];
  const estado * padre;
  int g_n; int h_n;
};
```

Por otro lado, tal y como viene explicado en la presentación de teoría relativa al Tema 3, el pseudocódigo que cabría esperar para este algoritmo sería el que sigue

```
Abiertos contiene el nodo inicial, Cerrados esta vacio
Comienza un ciclo que se repite hasta que se encuentra soluciin o hasta que Abiertos queda vacio
    Seleccionar el mejor nodo de Abiertos
    Si es un nodo objetivo terminar
    En otro caso se expande dicho nodo
    Para cada uno de los nodos sucesores
        Si esta en ABIERTOS insertarlo manteniendo la informacion del mejor padre
        Si esta en CERRADOS insertarlo manteniendo la informacion del mejor padre y actualizar la informacion de los descendientes
        En otro caso, insertarlo como un nodo nuevo
```

Más concretamente, veamos qué debemos hacer si nos encontramos con un nodo repetido en cerrados

```
Proceso recursivo para un nodo n en cerrados

p(n) padre anterior de n
p_c(n) padre actual

caso 1: p_c(n) no es mejor que p(n). Fin
caso 2: p_c(n) es mejor que p(n)
    actualizar g(n)  al  coste actual del camino
    actualizar el padre de n a p_c(n). Seguir recursion para cada uno de sus hijos que esten en Cerrados
Si n tiene sucesor en Abiertos
    Si el coste actual del camino hasta el sucesor mejora el coste anterior del sucesor
        actualizar el coste del sucesor
        actualizar el padre del sucesor
    Fin
```

No obstante, nosotros no tendremos que llevar a cabo la recursión en cerrados aprovechando que la heurística implementada es monótona (localmente admisible). Esto nos garantiza que el primer cerrado que se genera va a tener siempre el menor coste; esto es, no habrá que actualizarlo pues nunca un nuevo estado va a ser mejor.

Por otro lado, no controlaremos que se añadan abiertos repetidos a la cola con prioridad con el objetivo de realizar un menor número de búsquedas dentro de la ella (eficiencia lineal), pero evitaremos expandir abiertos que ya se hayan expandido alguna vez aprovechando una mayor eficiencia en la búsqueda en *Cerrados*, que es un set (eficiencia *O(log(n))*).

Sin embargo, aunque la búsqueda en *Cerrados* tiene un coste asociado de *O(log(n))* tal y como hemos señalado anteriormente, decidimos implementar *Cerrados* basándonos en un *unordered_set*. En este contenedor de la *STL*, al igual

que en set, la búsqueda se realiza por clave (*class Key*), siendo ésta única. La diferencia principal es la asociación de un valor *hash* a cada objeto. La búsqueda será tanto más rápida si conseguimos asociar un hash distinto a elementos distintos (inyectividad del *hash*).

```
    template < class Key,
        class Hash = hash<Key>,
        class Pred = equal_to<Key>,
        class Alloc = allocator<Key>
        > class unordered_set;
```

Con tal fin, el hash resultante (*class Hash*) se obtiene a partir de algunos de los atributos que conforma un estado. En un dato tipo *size_t* (*unsigned long* en nuestro caso) almacenamos contiguamente los distintos valores de mediante dos operaciones lógicas *shift left* y *or*. Con el primer operador nos desplazamos un determinado número de posiciones hacia la izquierda para almacenar contiguamente los distintos atributos, mientras que el segundo nos permite almacenar el valor exacto de cada uno de ellos (recordemos que al crear el hash se inicializan todas las posiciones a cero).

```
    struct Hasher {
    public:
        size_t operator()(const estado&a) const{
            size_t hash = a.fila;
            hash |= a.columna << 8;
            hash |= a.orientacion << 16;
            hash |= a.bik << 18;
            hash |= a.zap << 19;
            for (int i=0; i<3; i++){
                hash |= a.obj[i] << (20+i);
            }
            return hash;
        }
};
```

Nótese que fila y columna deberán tener un valor inferior a 256 (no contemplamos la posibilidad de trabajar con mapas de dimensión mayor); para la orientación necesitamos dos bits (0,1,2 ó 3), mientras que para el resto únicamente uno (true or false).

Class Pred deberá devolver *true* si los dos estados pasados como argumento coinciden.

```
    struct ComparaEstados {
        bool operator()(const estado& a, const estado& n) const {
            return (a.fila == n.fila && a.columna == n.columna && a.
    orientacion == n.orientacion &&
            a.bik == n.bik && a.zap == n.zap && a.obj[0] == n.obj[0] &&
     a.obj[1] == n.obj[1] && a.obj[2] == n.obj[2]);
        }
    };
```

Por último, *Abiertos* se organizará según la suma del coste y de la heurística de cada estado

```
    struct ComparaNodosSegunCosteYHeuristica {
        bool operator()(const estado& st1, const estado& st2) const {
            return st1.g_n + st1.h_n > st2.g_n + st2.h_n;
        }
    };
```

Efectuaremos la expansión de estados mediante el método *generarSucesor*, que expande el estado que se le proporciona como argumento.

```

estado ComportamientoJugador::generarSucesor(const estado &padre,
    Action value, int level) {
    estado sucesor;
    switch(value){
        case actTURN_R:
            sucesor = EstadoTurnR(padre);
            sucesor.g_n = padre.g_n;
            sucesor.g_n += level==4 ? coste_bateriaN4(padre, padre.zap,
     padre.bik, actTURN_R) : coste_bateria(padre, padre.zap, padre.bik,
     actTURN_R);
        break;
        case actTURN_L:
            sucesor = EstadoTurnL(padre);
            sucesor.g_n = padre.g_n;
            sucesor.g_n += level==4 ? coste_bateriaN4(padre, padre.zap,
     padre.bik, actTURN_L) : coste_bateria(padre, padre.zap, padre.bik,
     actTURN_L);
        break;
        case actFORWARD:
            sucesor = EstadoForward(padre);
            sucesor.g_n = padre.g_n;
            sucesor.g_n += level==4 ? coste_bateriaN4(padre, padre.zap,
     padre.bik, actFORWARD) : coste_bateria(padre, padre.zap, padre.bik
    , actFORWARD);
        break;
    }
    sucesor.padre=&padre;
    return sucesor;
}
```

Nivel cuatro

Como se ha visto hasta ahora, la eficiencia en los algoritmos de búsqueda ha sido la cuestión predominante durante el desarrollo de la práctica. Aprovecharemos entonces el algoritmo A^x para un único objetivo desarrollado en el nivel dos.

A continuación, introduciremos describiremos la actuación del agente durante la ejecución de este nivel con comentarios a cada paso.

En primer lugar, llevaremos la cuenta de los instantes de ejecución con una variable de estado *instante*, que incrementaremos en una unidad por cada iteración (acción realizada por el agente). Inicializamos los objetivos si la lista de objetivos está vacía (este if será true una única vez, al inicio, y ya no lo será más) y actualizamos los objetivos completados por el estado actual.

```
    instante++;
    if (obj.empty()) {
        calcularObjetivos(sensores);
    }
    ActualizarObjs(actual, obj.begin());
```

A continuación, procedemos a actualizar el mapa ahora que conocemos más información del mapa, así como el bikini y las zapatillas del estado actual, ahora que conocemos nuestra posición y orientación.

```
    actualizarMapa(sensores);
    actualizarBikZap(actual);
```

Comprobamos si nos encontramos en una casilla de recarga. Si lo estamos, nos esperamos hasta recargar la batería al nivel establecido según las condiciones de la ejecución: 500 si el instante de ejecución es superior a 2700, 2900 en caso contrario.

```

    if (mapaResultado[actual.fila][actual.columna] == 'X') {
        yendo_a_recargar = false;
        int esperarBateria;
        esperarBateria = (instante > 2700 ? 500 : 2900);
        if (sensores.bateria < esperarBateria) {
            return Action::actIDLE;
        }
    }
```

En lo sucesivos iremos debatiendo en qué condiciones el agente debe recalcular el plan, y cuál ha de ser el objetivo a alcanzar trás la ejecución de dicho plan según ciertos parámetros. En un principio, el plan ha de actualizarse si hemos actualizado el mapa recientemente. Por otro lado, como es lógico, también habrá de calcularse si todos los objetivos propuestos ya han sido conseguidos.

```

    bool calcular_plan = false;
    if (mapaResultado != oldMapaResultado)
        calcular_plan = true;
    oldMapaResultado = mapaResultado;

    if (actual.obj[0] and
        actual.obj[1] and
        actual.obj[2])
```

```
    {
        calcularObjetivos(sensores);
        for (int i = 0; i < 3; i++)
            actual.obj[i] = false;
        ActualizarObjs(actual, obj.begin());
        calcular_plan = true;
    }
```

Ahora buscamos la casilla de recarga más cercana según distancia Manhattan y debatimos sobre la posibilidad de visitarla. Bajo esta premisa, concluimos que una buena solución puede ser la de visitarla en el caso en que se encuentre a distancia menor o igual que cuatro y la batería esté por debajo de 700. Si estuviera por debajo de 500, debemos acudir a recargar sin importar el resto de condiciones de la ejecución.

No obstante, antes de acudir a recargar comprobamos primero que la casilla de recarga ha sido encontrada; en caso contrario, podría suceder que un mapa no tuviera ninguna casilla de recarga y el agente se quedara pillado intentando calcular un plan hacia ningún sitio.

En cualquier caso, si decidimos ir a recargar por cualquiera de los motivos debemos poner *yendo_a_recargar* a *true*.

```
    estado recarga;
    bool recarga_encontrada = recargaMasCercana(actual, recarga,
    recargas.begin());
    if (recarga_encontrada and
        distManhattan(actual, recarga) <= 4 and
        sensores.bateria < 700)
    {
        yendo_a_recargar = true;
        calcular_plan = true;
    }

    if (sensores.bateria < 500 and recarga_encontrada) {
        yendo_a_recargar = true;
        calcular_plan = true;
    }
```

Por último nos centramos en la búsqueda del objetivo más cercano en distancia Manhattan. Emplearemos varias formas de dirigirnos hacia el objetivo, dependiendo de las condiciones de ejecución en la que nos encontremos: por A^x, optimizando el coste hacia el objetivo, o por Anchura, optimizando el número de acciones (instantes de ejecución).

En particular, iremos con Anchura a por el objetivo según el siguiente criterio.

(I) Si tenemos bikini

- Si la distancia al objetivo más cercano es menor o igual que 16

- Si el instante de ejecución es posterior a 2000
  - Si el nivel de batería es superior a 2000

(II) Si no tenemos bikini

- Si tenemos zapatillas
  - Si la distancia al objetivo más cercano es menor o igual que 8
    - Si el instante de ejecución es posterior a 1750
      - Si el nivel de batería es superior a 2200
- Si no tenemos zapatillas
  - Si la distancia al objetivo más cercano es menor o igual que 12
    - Si el instante de ejecución es posterior a 2500
      - Si el nivel de batería es superior a 2500

Si decidimos ir con Anchura a por el objetivo, ponemos la variable de estado *ir_con_anchura* a *true*. Por último, calculamos el nuevo plan si procede hacia la recarga más cercana o, por el contrario, hacia el objetivo más cercano, ya sea optimizando la carga de batería o el número de acciones, según sea el caso tal y como acabamos de estudiar.

```
1
2  objetivo_actual = objetivoMasCercanoNoConseguido(actual, obj.begin());
3
4  if (calcular_plan or plan.empty()) {
5      plan.clear();
6      bool found_a_plan;
7      if (yendo_a_recargar) {
8          found_a_plan = pathFinding_AEstrella(actual, recarga, plan, 4);
9      } else if (ir_con_anchura){
10         found_a_plan = pathFinding_Anchura(actual, objetivo_actual,
       plan);
11     } else {
12         found_a_plan = pathFinding_AEstrella(actual, objetivo_actual,
       plan, 4);
13     }
14     VisualizaPlan(actual, plan);
15 }
```

Para finalizar, cabe destacar que hemos ido modificando los costes según el instante de ejecución en el que nos encontramos. Así, si el instante de ejecución es suficientemente alto, podemos reducir el coste asociado a viajar por el agua o por el bosque bajo ciertas condiciones. De igual forma, a partir de cierto instante de ejecución es conveniente dejar de explorar nuevo mapa e ir por rutas conocidas, por lo que podemos subir el coste asociado a esto último.

Como aclaración, los costes expresados no tienen por qué ser óptimos para cada algoritmo; dependerá de la implemetación que se haya realizado en los

niveles anteriores y de la política de actuación que siga el agente en este nivel, entre otras cosas. Por ello, podría suceder que si aplicamos estos costes a otra solución, el resultado obtenido no sea lo bueno que cabría esperar.

```
int ComportamientoJugador::coste_bateriaN4(estado est, bool zap, bool
    bik, Action act){
    char casilla = mapaResultado[est.fila][est.columna];
    double t = (double)instante / 3000;

    if (casilla == 'T') {
        return 2;
    } else if (casilla == '?') {
        if (bik){
            return 15*t + (1-t);
        } else if (zap){
            return 20*t + (1-t);
        } else {
            return 25 *t + (1-t);
        }
    } else {
        switch(act){
        case actFORWARD:
            if (casilla == 'A'){
                return (bik ? t + (1-t)*10 : t*150 + (1-t)*200);
            } else if (casilla == 'B'){
                return (zap ? t + (1-t)*15 : t*40 + (1-t)*100);
            }
        break;
        default:
            if (casilla == 'A'){
                return (bik ? t + (1-t)*5 : t*100 + (1-t)*500);
            } else if (casilla == 'B'){
                return (zap ? 1 : t + (1-t)*3);
            }
        break;
        }
    }
    return 1; }
```

```
[RESUMEN]
P1 mapa30: 110, Nivel: alto (>=100)
P2 mapa50: 67, Nivel: alto (>=60)
P3 mapa75: 40, Nivel: alto (>=40)
P4 mapa100: 29, Nivel: alto (>=25)
P5 pinkworld: 36, Nivel: alto (>=30)
P6 islas: 21, Nivel: alto (>=15)
P7 marymonte: 28, Nivel: alto (>=25)
P8 medieval: 30, Nivel: alto (>=30)
```

Figura 1: Resumen de los objetivos conseguidos por el agente en los distintos mapas propuestos en la práctica