Tema 2. Búsqueda en Espacios de Estados

Algoritmos de Búsqueda Heuristica: otros algoritmos

Objetivos

1. Conocer los fundamentos de los algoritmos de búsqueda y el papel que juegan en la Inteligencia Artificial
2. Conocer el paradigma de Búsqueda en Espacios de Estados y los **algoritmos** básicos de búsqueda a ciegas y sobre todo **de búsqueda inteligente o heurística**
3. Saber cómo modelar problemas para resolverlos con Búsqueda en Espacios de Estados, en particular cómo introducir conocimiento específico del dominio del problema

Contenidos

1. Introducción
2. Espacios de búsqueda
3. Algoritmos de búsqueda no informada
4. **Algoritmos de búsqueda informada o heurística**
    1. Introducción
    2. El algoritmo A*: descripción y propiedades formales
    3. **Otros algoritmos**
        1. **Algoritmos ε-admisibles**
        2. **Algoritmos no admisibles**
        3. **Algoritmos de búsqueda heurística en árboles**
5. Técnicas de diseño de funciones heurísticas

2.4.3 Otros algoritmos de Búsqueda Heurística

- A* es normalmente la mejor opción para grafos con muchos caminos alternativos entre pares de nodos
  - Ejemplos: el 8-puzzle, cálculos de rutas óptimas, planes de actuación, ….
- Si el tamaño del problema es muy grande hay que renunciar a la admisibilidad
  - Una opción son los algoritmos *ε-admisibles*
  - O incluso algoritmos no admisibles sin cota del error
- Hay otras opciones que pueden ser más eficientes, particularmente cuando el espacio de búsqueda es un árbol
  - Algoritmos de Ramificación y Poda (*B&B*, Branch and Bound)
    - Combinan A* con búsqueda en árboles con cálculos de cotas superiores (con un algoritmo voraz). Si *f(n)* >= *cota_superior*, se descarta *n*
  - DF (Depth First) o backtracking parcialmente informado
    - Se ordena los estados sucesores (o las reglas en backtracking) con un heurístico
  - Iterative Deepening A* (IDA*)
    - Parecido a DF iterativa, pero los sucesivos límites de profundidad se fijan con valores de f() de ciertos nodos
  - Limited Discrepancy Search (LDS)
    - En cada iteración se recorre una parte del espacio de búsqueda aumentando en 1 el nivel de discrepancia heuristica (el nivel 0 es cuando se sigue una rama en la que cada nodo es el mejor entre sus hermanos según el heurístico, el nivel 1 es cuando se admite para cada rama como mucho una discrepancia, ….)

2.4.3.1. Algoritmos ε-admisibles

- Son algoritmos que encuentran, con seguridad, una solución cuyo coste no excede $(1+\varepsilon)C^*$. Ejemplos (en todos los casos, *h* debe ser admisible)
  - **Ponderación Estática (PEA*)**

    $f(n) = g(n) + (1+\varepsilon)h(n)$, $\varepsilon > 0$
  - **Ponderación dinámica**

    $f(n) = g(n) + h(n) + \varepsilon\,[1-d(n)/N]\,h(n)$, ($d(n)$ es la profundidad de n y N es la prof. máxima)
  - **Algoritmo Aε***

    FOCAL = $\{n \in$ **frontier** / $f(n) \leq (1+\varepsilon)\min(f(n'), n' \in$ **frontier**$)\}$

    Se ordena FOCAL con otro heurístico $h'(n)$ que puede no ser admisible
- Hay variantes que garantizan un error menor que ε con una alta probabilidad
  - **Bounded Cost A* (BCA*)**

    $f(n) = h(n)/(C-g(n))$, con $C \geq C^*$

2.4.3.2. Algoritmos no admisibles
Sin cota del error

- **Algoritmos voraces**
  - Una opción es utilizar un algoritmo A* prescindiendo de $g(n)$, es decir con $f(n) = h(n)$
- **En ocasiones se puede utilizar una función *h* no admisible pero que aproxime de forma razonable a *h**. Ejemplos**
  - El valor del heurístico *h* se puede obtener como el coste que proporciona un algoritmo voraz para una relajación del problema que representa el estado *n*
  - O se pueden utilizar distintas técnicas de aprendizaje automático, supervisado o no supervisado, por ejemplo técnicas de regresión utilizando como conjunto de entrenamiento los valores de $h^*(n)$ para un conjunto de estados resueltos previamente
  - También se puede modificar el cálculo de los heurísticos admisibles para ponderar al alza algunos términos que pueden ser demasiado optimistas: por ejemplo en el heurístico h_2 del 8-puzzle podemos contar 4 por las fichas que están a distancia 3, ó 5 por las que están a distancia 4.

2.4.3.3 Algoritmos de búsqueda heurística en árboles
Algoritmo IDA* (Iterative Deepening A*)

```
function IDA*-SEARCH(problem) returns solution-node or failure
    node = NODE(problem.INITIAL)
    limit = h(node)
    while (true) do
        t = IDA*-REC(node, problem, limit)
        if (t is a node) return t
        if (t  = ∞) return failure
        limit = t

function IDA*-REC (node, problem, limit) returns solution-node or a number
    if (IS-GOAL(node)) return node
    if (f(node) > limit)  return f(node)
    actions ← ACTION-LIST(node, problem)
    min = ∞
    while ( !IS-EMPTY(actions)) do
        action ← POP(actions)
        node-suc ←EXPAND(problem, node, action)
        t ← IDA*-REC (node-suc, problem, limit)
        if (t is a node) return t
        if (t  is a number) min = MIN(min, t)
    return min
```

- Es similar a una búsqueda iterativa en profundidad, pero el límite en cada iteración se establece a partir del valor de *f*() de ciertos nodos
  - Inicialmente, el límite es *f*(*inicial*) = *h*(inicial)
  - En las iteraciones sucesivas, el límite es el mínimo valor de *f*() de los nodos descartados en la iteración anterior
- IDA*-SEARCH(*problem*) devuelve un nodo solución o fallo si el problema no tiene solución
- IDA*-REC(*node, problem, limit*) comprueba si hay un nodo solución a partir de *node* con un valor de *f*() menor o igual que *limit*
  - Si encuentra solución devuelve el nodo solución encontrado
  - Si no encuentra solución devuelve un valor que es el nuevo limite para la siguiente iteración. Si el valor devuelto es ∞ entonces no hay solución a partir del *node*
- Propiedades de IDA*
  - Es admisible si *h*() es consistente
  - Reexpande muchos nodos en las iteraciones sucesivas, pero requiere muy poca memoria