Backtracking

Prof. Rhadamés Carmona

Última revisión: 13/05/2019

Agenda

- Concepto
- Esquema general
- Mochila (Knapsack)
- Permutaciones, Combinaciones, Variaciones
- Rat in maze
- Sudoku
- 8 (N) reinas
- Suma de subconjuntos
- Coloración de grafos
- Ciclos hamiltonianos
- Ventajas / desventajas
- Ideas finales: alg. Probabilísticos, A*

Concepto

- Introducido por Lehmer en los 50’s. R.J. Walker dio una descripción algorítmica en 1960. Luego desarrollado por S. Golamb y L. Baumert.
- Es un algoritmo que explora sistemáticamente e incrementalmente el espacio de solución, considerando en cada paso todas las opciones, y abandona candidatos cuando no es posible completar una solución. Reduce la cantidad de trabajo en búsquedas exhaustivas.

Concepto

- El algoritmo va extendiendo una solución parcial abordando todos los caminos de solución posibles. Estas soluciones parciales forman un árbol potencial de búsqueda, que se va recorriendo top-down en “depth-first” order (DFS o en profundidad).
- Aquellas ramas en donde se sabe que no se llega a ninguna solución, son podadas, y así el árbol no crece indefinidamente.

Ejemplos

- Mochila: maximizar valor de los objetos llevados sujeto a un peso máximo
- Sudoku: colocar dígitos sin repetición en tablero
- 8 reinas: colocarlas sin que se coman
- Rat in maze (laberinto): la rata (1,1) puede llegar a la salida (n,n) de un laberinto de obstáculos?
- De un arreglo de n, generar P(n), V(n,m), C(n,m)
- M-coloring: se puede colorear un grafo con m colores sin que 2 nodos adyacentes tengan el mismo color?.

Esquema general

```
void bt(partial_solution, candidates)
{
    if (!is_valid( partial_solution)) return;
    if (is_solution(partial_solution))
        process_solution(partial_solution);
    /* else ?*/
    for each x in candidates
        bt(partial_solution⊕x, candidates – {x} );
}
```

Esquema general

- **is_valid**: es falso cuando no es una solución, pero más aún, cuando a partir de esa solución parcial NO se puede llegar a ninguna posible solución. Ejemplo: hay 7 reinas pero 2 de ellas se comen. Esta condición se puede también verificar antes de agregar un elemento a la solución para evita la llamada recursiva y cómputo en general. Definitivamente esto poda el árbol potencial de búsqueda.

Esquema general

- **is_solution**: es cierto cuando se puede considerar como una posible solución. Ejemplo, cuando el último elemento de partial_solution coincide con la salida del laberinto (ran in a maze), o cuando has colocado la 8va. reina en partial_solution y esta no se “come” con el resto.

Esquema general

- **bt(partial_solution⊕x, candidates – {x})**: se agrega x a la solución parcial, mientras se elimina x de los posibles candidatos. Note que esta operación está en el parámetro, por lo que no afecta el estado de partial_solution ni de candidates en el nivel de recursión actual. Si se hace directamente sobre las variables locales, hay que hacer roll-back después de la llamada recursiva.
- En ciertos problemas (e.g. generar variaciones con repetición) el candidato x no se excluye de candidates, pues puede estar más de una vez en la solución.

Esquema general

- **bt(partial_solution⊕x, candidates – {x})** puede ser rescrito como

```
If (is_valid(partial_solution + x))
{
        partial_solution += x;              // agregar alternativa
        candidates -= {x};                  // excluir, depende
        bt(partial_solution, candidates);
        candidates += {x};                  // incluir, depende
        partial_solution -= x;              // quitar alternativa
}
// a veces con un simple contador de “paso” se excluye
// automáticamente un candidato en futuros pasos recursivos
```

Esquema general

- **process_solution(c):** dependiendo del problema, es posible salir al encontrar la primera solución (usar un lógico global "found" para abortar la búsqueda en todos los niveles). Pero si se quiere la solución óptima, entonces se puede comparar esta solución con la mejor encontrada. Y si se quieren todas las soluciones, se puede guardar o imprimir la solución en este momento.

Mochila (Knapsack)

- Dados N objetos, con un valor y un peso cada uno, llenar una mochila de objetos sin sobrepasar un determinado peso MAX.
- **is_valid**: la suma de los pesos de los objetos dentro de la mochila no pueden sobrepasar MAX.
- **is_solution**: cualquier mochila no vacía es una posible solución.

Mochila (Knapsack)

- **Partial_solution**: una lista de objetos, con la suma de sus pesos y la suma de sus valores.
- **Candidates**: todos los objetos remanentes que no se han colocado en la mochila.

```
struct element {
    public:
        int value, weight;
}
```

Mochila (Knapsack)

```
class solution : public list<element> {
    public:
        int svalue=0, sweight=0;
};

solution knapsack(vector<element> &candidates, int max) {
    solution current, best;
    sort_by_weight(candidates);// poda: a[i] no cabe➔ a[j] tampoco, j>i
    mochila(0, current, candidates, best);
    return best;
}
```

Mochila (Knapsack)

```
void  knapsack (int step, solution &s, vector<element>
    &candidates, solution &best) {
    // if (c.sweight > MAX) return;        innecesario
    if (/*c.sweight <= MAX && */s.svalue > best.svalue) best = s;
    for (int i=step; i<candidates.size(); i++) {
      const element &x = candidates[i];
      if (s.sweight + x.weight > MAX) break;// poda por "sorting"
      s.push_back(x); s.sweight +=x.weight; s.svalue+= x.value;
      knapsack (step+1, s, candidates, best);
      s.pop_back(); s.sweight -=x.weight; s.svalue-= x.value;
    }
}
```

Permutaciones

```
void bt(sol, candidates)
{
  if (!is_valid( sol )) return;
  if (is_solution( sol ))
    process_solution( sol );
  /* else ?*/
  for each x in candidates
      bt(partial_solution⊕x, candidates – {x} );
}
```

0 1 2 3 1 0 2 3 2 1 0 3 3 1 2 0
0 1 3 2 1 0 3 2 2 1 3 0 3 1 0 2
0 2 1 3 1 2 0 3 2 0 1 3 3 2 1 0
0 2 3 1 1 2 0 3 2 0 3 1 3 2 0 1
0 3 2 1 1 3 2 0 2 3 0 1 3 0 2 1
0 3 1 2 1 3 0 2 2 3 1 0 3 0 1 2

```
//P no hace falta. Se puede usar el mismo A
void permute(int step, int A[], int P[], int n)
{

  if (step==n)
     print(P,n);
  else
    for (int i=step; i<n; i++)
    {
              P[step] = A[i];
              std:: swap(A[step], A[i]);
              permute(step+1, A, P, n);
              std::swap(A[step], A[i]);
    }
}
```

Variaciones

```
void bt(sol, candidates)
{
  if (!is_valid( sol )) return;
  if (is_solution( sol ))
    process_solution( sol );
  /* else ?*/
  for each x in candidates
      bt(partial_solution⊕x, candidates – {x} );
}
```

La única diferencia es que se imprime al llegar a m elementos en V. Cambiando m por n, obtenemos de nuevo las permutaciones.

```
void variations(int step, int pos, int A[], int V[],
int n, int m)
{

  if (pos==m)
     print(V,m);
  else
    for (int i=step; i<n; i++)
    {
              V[pos] = A[i];
              std:: swap(A[step], A[i]);
              variations (step+1, pos+1,A, V, n,m);
              std::swap(A[step], A[i]);
    }
}
```

Combinaciones

```
void bt(sol, candidates)
{
  if (!is_valid( sol )) return;
  if (is_solution( sol ))
    process_solution( sol );
  /* else ?*/
  for each x in candidates
      bt(partial_solution⊕x, candidates – {x} );
}
```

A=0,1,2,3 n=4, m=3, C(4,3) = 4!/(3!1!) = 4

0 1 2
0 1 3
0 2 3
1 2 3

```
void combine(int step, int pos, int A[], int C[],
int n, int m)
{

  if (pos==m)
     print(C, m);
  else
    for (int i=step; i<n; i++)
    {
              C[pos] = A[i];
              combine (step+1, pos+1,A, C, n, m);
    }
}
```

Sudoku

3		6	5		8	4		
5	2							
	8	7					3	1
		3		1			8	
9			8	6	3			5
	5			9		6		
1	3					2	5	
							7	4
		5	2		6	3		

Every colum, row and 3x3 square should contain all digits 1..9

Sudoku

- Intentamos rellenar los dígitos uno por uno. Siempre que encontremos que el dígito actual no puede llevar a una solución, lo eliminamos (retrocedemos) y probamos el siguiente dígito. Esto es mejor que el enfoque ingenuo (generando todas las combinaciones posibles de dígitos y luego probando cada combinación una por una) ya que elimina un conjunto de permutaciones cada vez que retrocede.

Sudoku

- **Partial_solution**: el mismo tablero, con información adicional, como la cantidad de posiciones libres restantes.
- **Candidates**: los 9 dígitos posibles.
- En cada nivel de recursión, se selecciona la primera posición libre del tablero UNICAMENTE. Allí se hace recursión con los valores 1..9, pasando las posiciones libres restantes.

Sudoku

- **is_solution**: en partial_solution se puede llevar un contador de posiciones libres restantes. Cuando una solución parcial haya llenado todos los “huecos” (libres==0), entonces, retorna verdad.
- **is_valid**: debe chequear que el dígito a colocar no aparezca en su fila, columna ni sub matriz de 3x3.

Sudoku

```
void sudoku(int A[9][9], list<pair<int,int>> &libres, bool &found) {
  if (libres.size()==0)
     print(A); found = true;
  else  {
    int i = libres.begin()->first;
    int j = libres.begin()->second;          // esta posición se trata en este nivel
    libres.pop_front();                      // y no se repone
    for (int digit=1; digit<=9 && !found; digit++)   {
              A[i][j] = digit; // agregar alternativa a la solución
              if (horizOK(A, i) && vertOK(A,j) && subMatrixOK(A,i,j))
                        sudoku(A, libres, found);
              A[i][j] = 0;  // deshacer alternativa; es innecesario en este problema
    }
}
```

Las posiciones libres deben tener un dígito inválido en el tablero, e.g 0 o -1.

Se debe llenar la lista de posiciones libres previamente, found = false.

Complejidad = 9^n, con n = cantidad de posiciones libres.

Como n es a lo sumo 81, entonces queda O(1). What?.

Rat in maze

- La rata arranca en la posición (0,0).
- El mapa o laberinto es un arreglo 2D de posiciones libres (false) y obstáculos (true).
- La salida está en la posición (n-1,n-1).
- Podemos obtener cualquier camino a la salida, todos los caminos o el mejor camino.
- En este ejercicio vamos a imprimir todos los caminos. Para el óptimo hacer algo similar a mochila (lleva traza del mejor encontrado).

Rat in maze

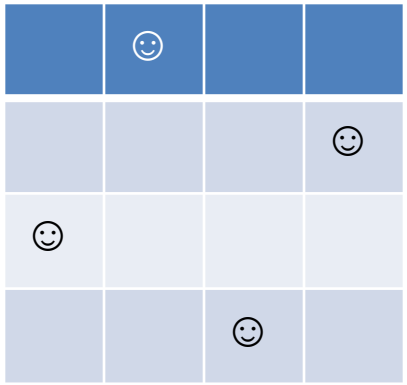
- is_valid: una alternativa es válida si la posición a donde va la rata no está bloqueada.
- is_solution: si la rata está en la salida, es decir, en la posición (n-1,n-1).
- partial_solution: la solución la vamos a ir llevando en una lista de posiciones, junto al tablero mismo.
- candidates: las 4 posiciones en “false” alrrededor de la rata en el nivel actual.

Rat in maze

```
void rat (int i, int j, int A[n][n], list<pair<int,int>> &path) {
   if (i==n-1 && j==n-1)
          process_solution(A, path);
   else   {
             pair<int,int> candidates[4] = { {i-1,j}, {i+1,j}, {i,j-1}, {i,j+1} }; // simplificado
             for (each x in candidates)   {
                          int a = x.first;
                          int b = x.second;
                          if (a>=0 && a<n && b>=0 && b<n && A[a][b]==false) {
                                       A[a][b] = true;
                                       path.push_back(make_pair(a,b));
                                       rat(a,b,A,path);
                                       path.pop_back();
                                       A[a][b] = false;
                          }
             }
    }
}
```

8 reinas

- 8 reinas para 8x8
- N reinas para NxN
- Consiste en colocar las N reinas en un tablero NxN sin que las reinas se “alcancen” en movimientos horizontales, verticales o diagonales.
- Para N<4 esto no es posible.



8 reinas

- Las peores soluciones: (a) obtener todas las permutaciones de un arreglo de 64 casillas con 8 casillas marcadas en 1, y 56 marcadas en 0. Luego probar cada permutación. (b) Generar todas las combinaciones C(64,8) y chequear la condición de que no se ataquen por comb. (c) Buscar las posiciones libres (revisando las 64 posibilidades) en cada iteración.

8 reinas

- Optimización 1: en el nivel x de recursión, considerar las 8 posiciones de la fila x únicamente.
- Como hay 8 filas, tenemos 8^8 verificaciones en todo el algoritmos (aprox. 16M).
- Podemos marcar las columnas y diagonales visitadas para reducir la “verificación” a un simple “if A[i][j] está libre”, pero el recorrido sigue siendo 8^8.

8 reinas

```
void reinas(int step, int &A[8][8])
{
        if (step==8)
          procesarSolucion(A);
        else
          for (int j=0; j<8; j++) if (A[step][j] == 0)
          {
                marcarColumna(j,1);             // A[step..7,j]=1
                marcarDiagonales(step,j,1);
                A[step][j] = 2;
                reinas(step+1);
                marcarColumna(j,0);             // A[step..7,j]=0
                marcarDiagonales(step,j,0);
          }
}
```

8 reinas

- Optimización 2: generar todas las permutaciones de 8 números (01234567). Son 8!, osea, 40.320.
- Se toma cada permutación, y los números se colocan uno en cada fila. Note que las reinas no se ven ni en filas ni en columnas, por lo que solo hay que verificar diagonales.
- La siguiente optimización es generar la permutación en backtracking y no continuar la recursión cuando 2 reinas se ven en diagonal.

N reinas

```
void permute(int step, int A[], int n) {
  if (step==n)
     print(A,n);          // (0,A[0]), (1,A[1]), (2,A[2]), …, (n-1, A[n-1])
  else
    for (int i=step; i<n; i++)  {
              bool valid = true;
              for (int k=0; k<step; k++)
                        if (sameDiagonal(k, A[k], step, A[i]))  {
                                  valid = false; break;
                        }
              if (valid) {
                        std:: swap(A[step], A[i]);
                        permute(step+1, A, n);
                        std::swap(A[step], A[i]);
              }
    }
}
```

main: permute(0, A=[0,1,2,3,4,5,6,7], 8); // sameDiagonal should be O(1)

Suma de subconjuntos

- Dado un conjunto de enteros, existe un subconjunto cuya suma sea X?.
- NP-completo.
- Se pude resolver con backtracking, o programación dinámica.

Suma de subconjuntos (todas las soluciones)

```
void  sumsubset(int step, vector<int>&A, vector<int> &S,
                        int sum, int x) {
    if (sum==x && S.size() > 0)
          saveSolution(S);
    for (int i=step; i<A.size(); i++) {
        S.push_back(A[i]);
        sumsubset(step+1, A, S, sum+A[i], x);
        S.pop_back();
    }
}
```

Dado S y x: vector<int> S; sumsubset(0, A, S, 0, x);

Suma de subconjuntos (Primera solución)

```
void  sumsubset(int step, vector<int>&A, vector<int> &S,
                        int sum, int x, bool &found) {
    if (sum==x && S.size() > 0) {
          processSolution(S); found = true; }
    for (int i=step; i<A.size() && !found; i++) {
        S.push_back(A[i]);
        sumsubset(step+1, A, S, sum+A[i], x);
        S.pop_back();
    }
}
```

Dado A y x: `bool found=false; vector<int> S;`

`sumsubset(0, A, S, 0, x, found);`

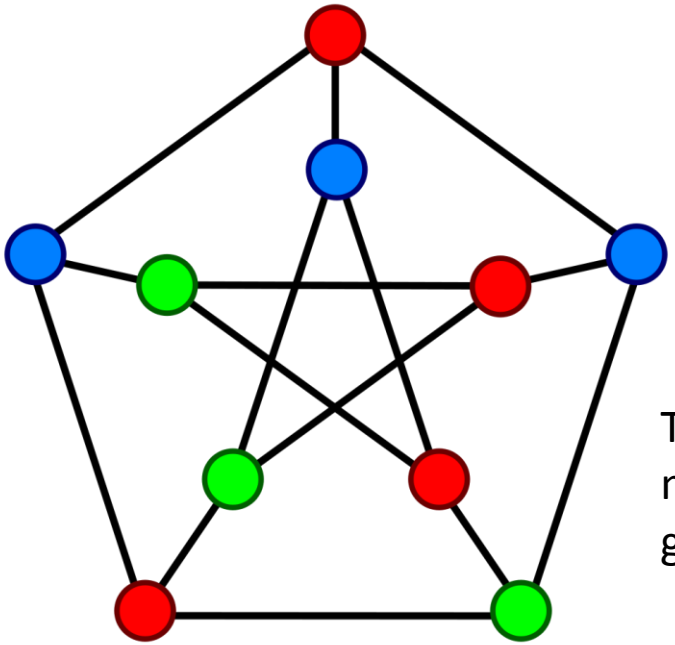
Suma de subconjuntos (Solución más pequeña)

```
void  sumsubset(int step, vector<int>&A, vector<int> &S,
     vector<int> &best, int sum, int x) {

    if (best.size() > 0 && S.size() >= best.size()) return;

    if (sum==x && S.size() > 0) best = S;

    else for (int i=step; i<A.size(); i++) {

         S.push_back(A[i]);

         sumsubset(step+1, A, S, sum+A[i], x);

         S.pop_back();

    }

}
```

Dado S y x: vector<int> S, best; sumsubset(0, A, S, best, 0, x);

Coloración de grafos

- **Coloración** de los vértices de un **grafo** tal que ningún vértice adyacente comparta el mismo color.



Tres colores es el mínimo para este grafo

Coloración de grafos

- La pregunta es, se puede colorear el grafo G(V,E) con k colores?.
- La solución inicial por backtracking es probar cada uno de los k colores (k alternativas) al vértice step-ésimo, en el paso “step”.
- Al darle el color, la “alternativa” es válida si todos sus adyacentes tienen otro color, o no han sido coloreados aún.
- $O(n^k)$, con n=|V|, k = #colores

Coloración de grafos

- Como nos importa responder “sí o no”, entonces con encontrar la primera solución basta.
- Esa primera solución se puede encontrar más rápidamente si ordenamos los vértices ascendentemente por grado.

Coloración de grafos

Estructuras de datos

```
struct vertex {
        int ID;
        int color = -1;          // uncolored
        bool operator < (const vertex &x) const {
                return ady.size() < x.ady.size();
        }
        vector<int> ady;     // ady.size() es el grado
};

typedef vector<vertex> graph;
```

O simplemente…

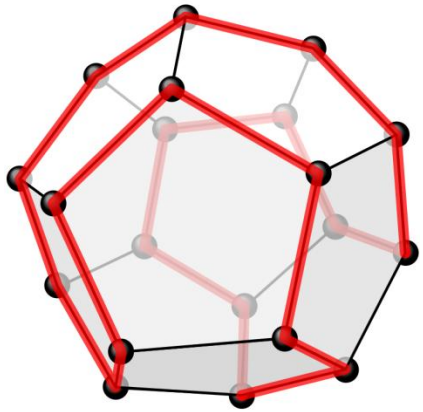
```
class graph : public vector<vertex>
{
};
```

Coloración de grafos

```
void coloring(int step, graph &g, int k, bool &done) {
        if (step==g.size())                          // all vertices considered
                done= true;
        for (int c=0; c<k && !done; c++) {           // for each color
            bool valid = true;
            for (int i=0; i<g[step].ady.size(); i++) // check valid option c
                if (g[g[step].ady[i]].color == c) {
                        valid = false;
                        break;
                }
            if (valid) {
                    g[step].color = c;
                    coloring (step+1, g, k, done);
                    g.v[step].color = -1;
            }
        }
}
```

Ciclo Hamiltoniano

- Un C.H. en un grafo conexo consiste en seleccionar las aristas de un grafo de manera de generar un ciclo, visitando todos los vértices solo una vez.
- Si removemos una arista, entonces es un camino Hamiltoniano.



Ciclo Hamiltoniano

Estructuras de datos

```
struct vertex
{
        int ID;
        bool visited;
        vector<int> ady;
};

typedef vector<vertex> graph;
```

O simplemente…

```
class graph : public vector<vertex>
{
};
```

Ciclo Hamiltoniano

```
void ham(int step, int index, graph &g, int *loop, bool &done) {
        if (step==g.size())      // all vertices were considered
                done=true;// (loop[step] ==loop[0]);
        else
                for (int i=0; i<g[step].ady.size() && !done; i++) {
                        int j = g[step].ady[i];
                        if (g[j].visited == false) {
                                g[j].visited  = true;
                                loop[step]      = index;
                                loop[step+1] = j;
                                ham(step+1, j, g, loop, done);
                                g[j].visited = false;
                        }
                }
}
```

Ventajas

- Garantía: encuentra la solución, si existe.
- Es sencillo de implementar en general, código corto, y las soluciones suele seguir un mismo esquema.
- Explora incrementalmente todas las soluciones posibles, y descarta tempranamente candidatos que no llevarán a la solución.
- Obtiene todas las soluciones posibles.

Desventajas

- No hay memorización, por lo que se puede recorrer un subárbol potencial de búsqueda muchas veces.
- Suele ser de naturaleza “exponencial”, lo cual lo hace impráctico para problemas grandes.
- Se necesitan podas inteligentes y dependientes del problema para reducir su tiempo de ejecución.
- Consume mucho espacio de pila, en problemas medianos o grandes.
- Mientras más alternativas por estado es más lento.

Ideas finales

- Para problemas con muchas soluciones (e.g. N reinas) se puede explorar solo algunas posiciones aleatorias en cada nivel de recursión en vez de todas (algoritmo probabilístico).
- Para problemas en donde las diferentes alternativas tienen heurísticamente distina posibilidad de estar en la solución, las alternativas podrían explorarse en cierto orden (A*) para encontrar la primera solución rápidamente (seguramente una buena solución).

Tarea

- Implementar las dos soluciones planteadas para N reinas, e incluir luego un approach probabilístico para la primera solución: considerar aleatoriamente solo el 30% de las posiciones de cada fila. Comparar tiempo de respuesta para distintos valores de N.
- Con qué probabilidad se encuentra una solución al explorar el 30% de los elementos de cada fila?. Obtenga esta cifra empíricamente para cada N, probando con N=8, N=16, N=32, N=64.