Algorítmica

Capítulo 5: Algoritmos para la Exploración de Grafos

Ejercicios prácticos

Objetivos de las prácticas

- Con esta práctica lo que se persigue es que el estudiante:
  1. Aprecie la potencia de los métodos Backtracking para resolver problemas, pero también comprenda sus limitaciones.
  2. Reconozca la importancia de disponer de buenas funciones de acotación así como de generación y ramificación de nodos.
  3. Compruebe la gran variedad de situaciones abordables con estas técnicas.
  4. Trabaje comprometidamente en equipo
  5. Aprenda a expresar en público las ventajas, inconvenientes y alternativas empleadas, para lograr la solución alcanzada

El problema del coloreo de un grafo

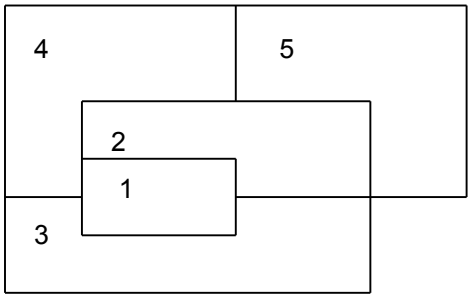
- Sea G un grafo y m un numero entero positivo. Queremos saber si los nodos de G pueden colorearse de tal forma que no haya dos vértices adyacentes que tengan el mismo color, y que solo se usen m colores para esa tarea.
- Este es el problema de la m-colorabilidad.
- El problema de optimización de la m-colorabilidad, pregunta por el menor numero m con el que el grafo G puede colorearse.
- Ese entero es el denominado Número Cromático del grafo.

El problema del coloreo de un grafo

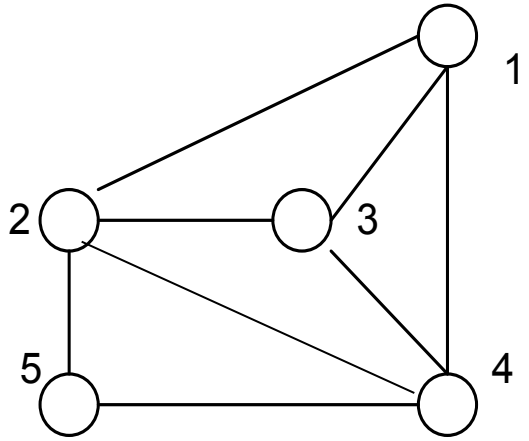
- Un grafo se llama plano si y solo si puede pintarse en un plano de modo que ningún par de aristas se corten entre si.
- Un caso especial famoso del problema de la m-colorabilidad es el problema de los cuatro colores para grafos planos que, dado un mapa cualquiera, consiste en saber si ese mapa podrá pintarse de manera que no haya dos zonas colindantes con el mismo color, y además pueda hacerse ese coloreo solo con cuatro colores.
- Este problema es fácilmente traducible a la nomenclatura de grafos

El problema del coloreo de un grafo

- El mapa



- puede traducirse en el siguiente grafo

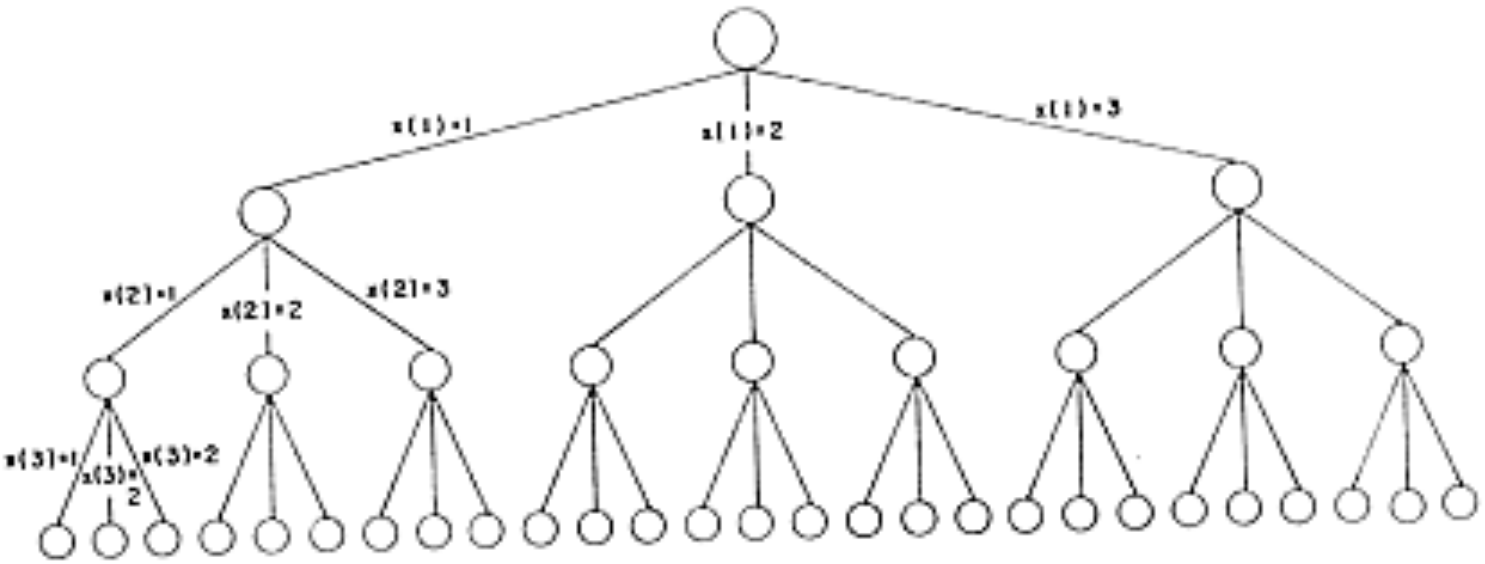


El problema del coloreo de un grafo

- Representamos el grafo por su matriz de adyacencia GRAFO(1:n, 1:n) siendo GRAFO(i,j) = true si (i,j) es una arista de G. En otro caso GRAFO(i,j) = false.
- Los colores se representan por los enteros 1, 2, ..., m
- Las soluciones vendrán dadas por n-tuplas (X(1),...,X(n), donde X(i) será el color del vértice i.
- Usando la formulación recursiva del procedimiento backtracking, puede construirse un algoritmo que trabaja en un tiempo $O(nm^n)$

El problema del coloreo de un grafo

- El espacio de estados subyacente es un árbol de grado m y altura n+1, en el que cada nodo en el nivel i tiene m hijos correspondientes a las m posibles asignaciones para X(i), $1 \leq i \leq n$, y donde los nodos en el nivel n+1 son nodos hoja.



El problema del coloreo de un grafo

Algoritmo M-Color (k)

```
while (true)
   SiguienteValor(k)
   if (color[k] = 0) then break                    (1)
   if (k = n)
     then print este coloreo                       (2)
     else M-Color (k + 1)                          (3)
endWhile
```

(1) {no hay mas colores para k}
(2) {se encontró un coloreo valido para todos los nodos}
(3) {intenta colorear el siguiente nodo}

El problema del coloreo de un grafo

Algoritmo SiguienteValor (k)

{Devuelve los posibles colores de X(k) dado que X(1) hasta X(k-1) ya han sido coloreados}

```
while (true)
 color[k] = (color[k] + 1) mod (n + 1)
 if (color[k] = 0) then return                    (1)
 for i = 1 to n+1
   if (conec[i,k] and color[i] = color[k])
     then break
 endfor
 if (i = n+1) return                              (2)
endWhile
```

(1) no hay mas colores para probar

(2) Se ha encontrado un nuevo color (ningun nodo colisiona)

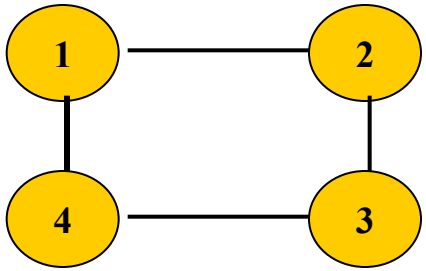
Eficiencia del algoritmo

- El número de nodos internos en el espacio de estados es $\sum_{i=1..n-1} m^i$
- En cada nodo interno SiguienteValor invierte O(nm) en determinar el hijo corespondiente a un coloreo legal.
- El tiempo total esta acotado por

$$\sum_{i=1..n-1} m^i n = n(m^{n+1}-1)/(m-1) = O(nm^n)$$

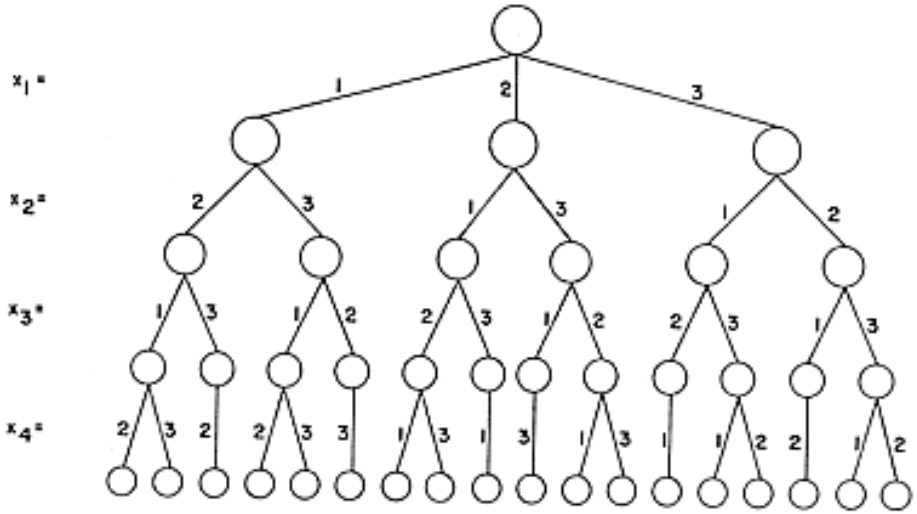
Ejemplo de coloreo

Si consideramos el siguiente grafo

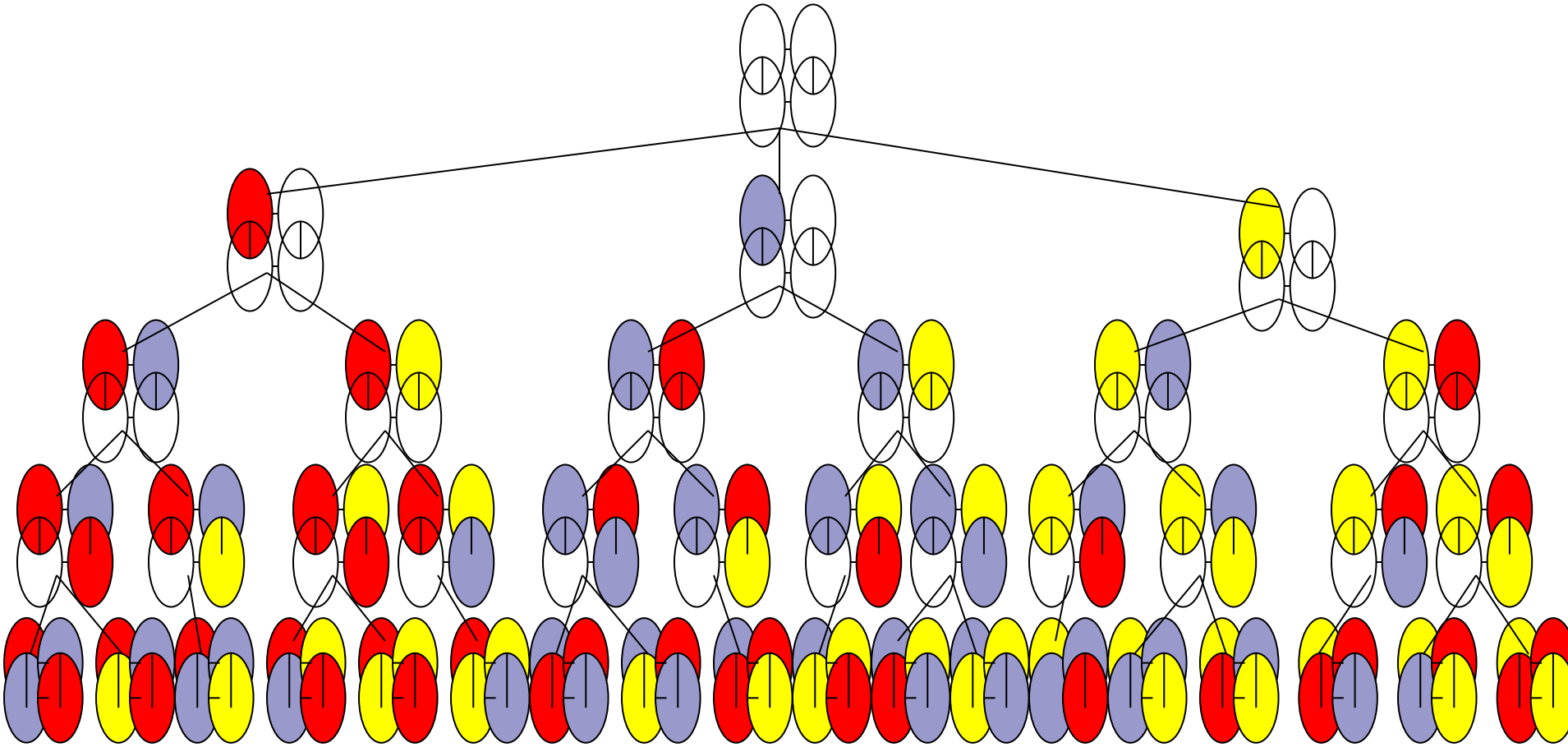


Cada camino a una hoja representa un coloreo usando a lo mas 3 colores

El arbol que genera M-Color es



Otra representación del ejemplo



Laberintos y Backtracking

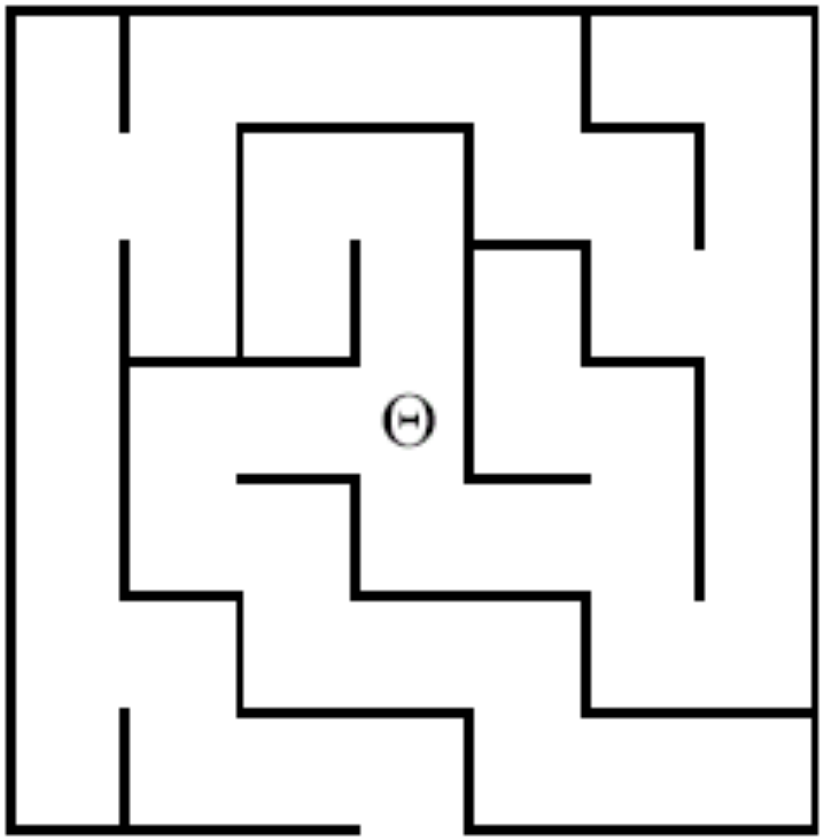
Este mosaico representa un laberinto, y esta en la Catedral de Chartres. Antes de estar alli, ya se conocia en Creta mil años antes.

Tambien es conocido en otras culturas.

Un laberinto puede modelarse como una serie de nodos.

En cada nodo hay que tomar una decision que
nos conduce a otros nodos.

Un laberinto sencillo

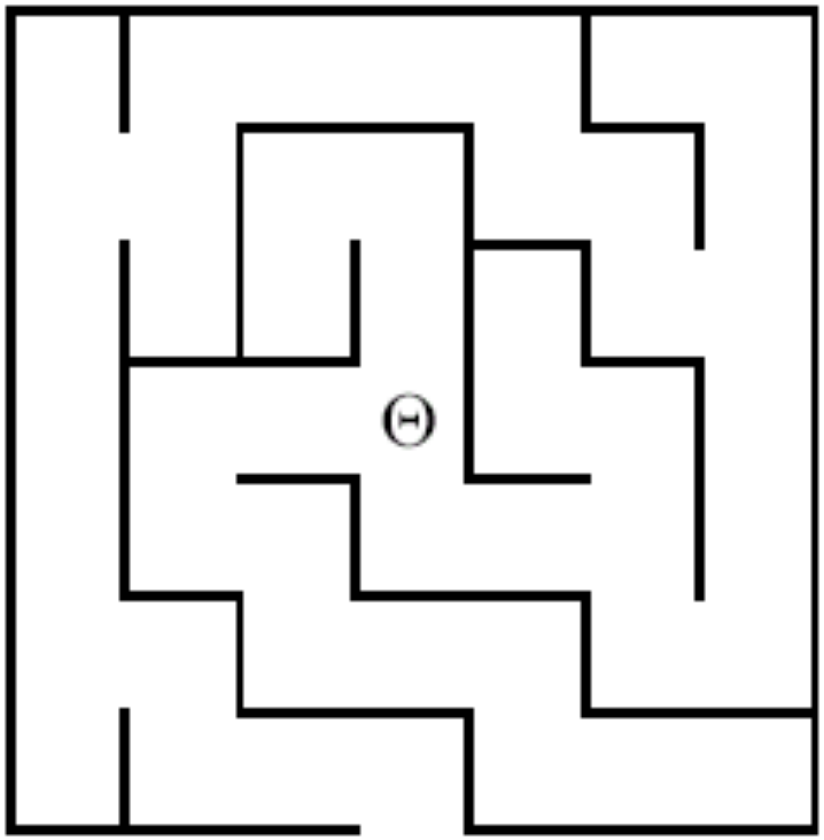


Buscar en el laberinto hasta encontrar una salida. Si no se encuentra una salida, informar de ello

Algoritmo Backtracking Modificado

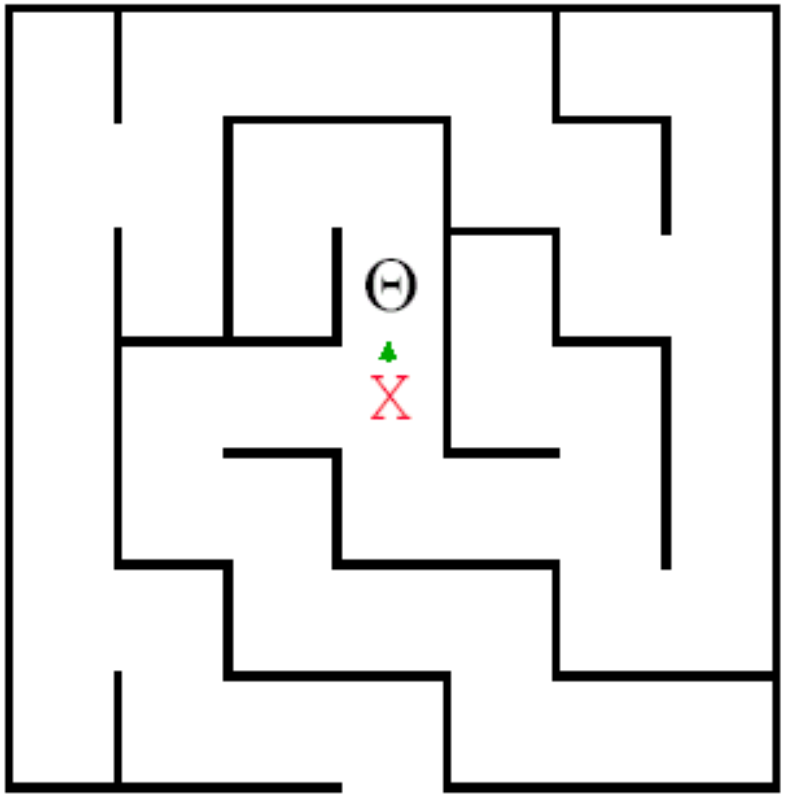
- Si la posición actual esta fuera, devolver TRUE para indicar que hemos encontrado una solución.
  Si la posición actual esta marcada, devolver FALSE para indicar que este camino ya ha sido explorado.
  Marcar la posición actual.
  For (cada una de las 4 direcciones posibles **)**
  { **Si (** Esta direccion no esta bloqueada por un muro **)**
  { Moverse un paso en la dirección indicada desde la posición actual.
  Intentar resolver el laberinto desde ahí haciendo una llamada recursiva.
  Si esta llamada prueba que el laberinto es resoluble, devolver TRUE para indicar este hecho.
  }
  }
  Quitar la marca a la posición actual.
  Devolver FALSE para indicar que ninguna de las 4 direcciones lleva a una solución

Un laberinto sencillo



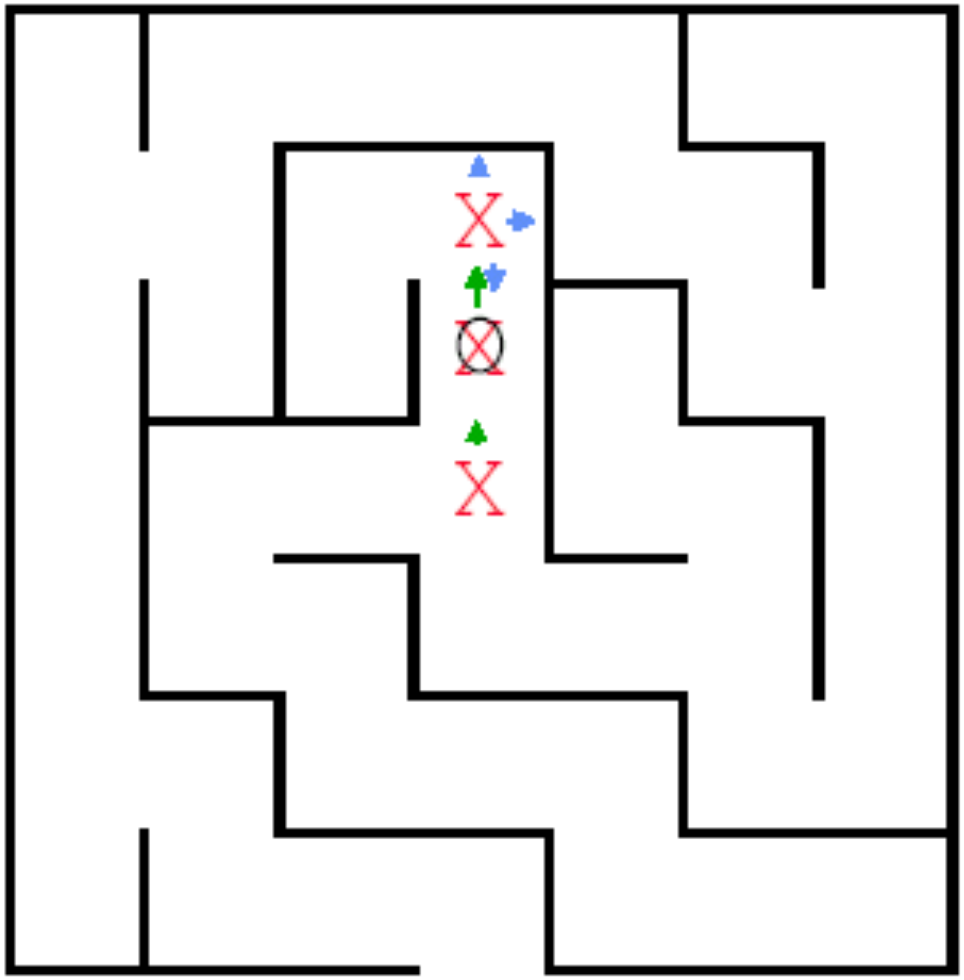
Aplicar el anterior algoritmo desde esa posición en este laberinto

Backtracking en Acción



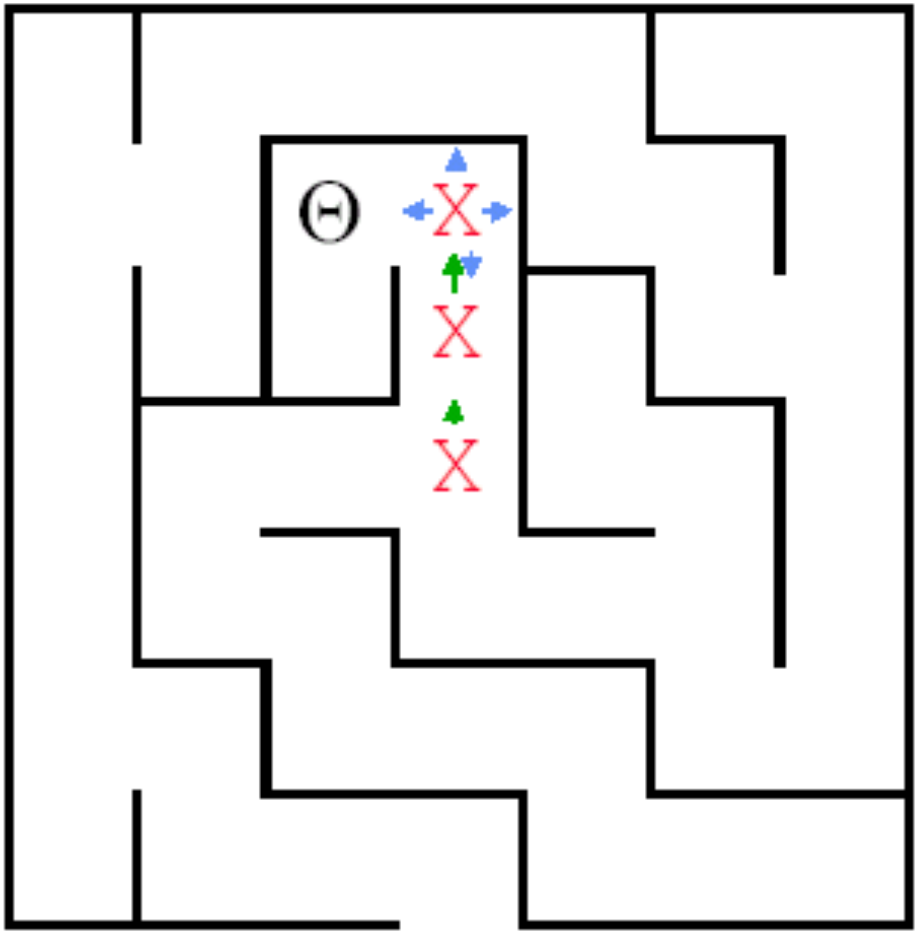
- La parte crucial del algoritmo es el lazo FOR que nos lleva hacia las posibles alternativas que hay en un punto concreto.
- Aquí nos movemos hacia el norte.

Backtracking en Acción



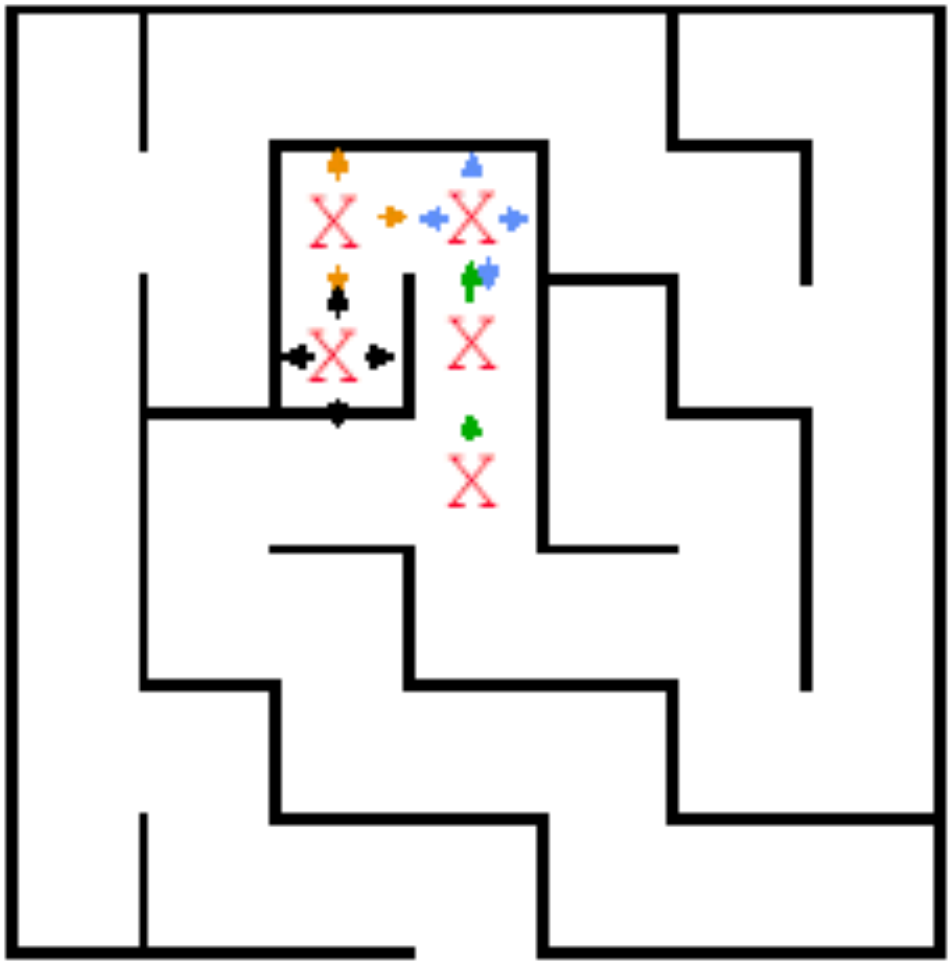
- Aquí nos movemos hacia el Norte de nuevo, pero ahora la dirección Norte esta bloqueada por un muro. El Este también esta bloqueado, por lo que intentamos el Sur.
- Esa acción descubre que ese punto esta marcado, de modo que volvemos atrás.

Backtracking en Acción



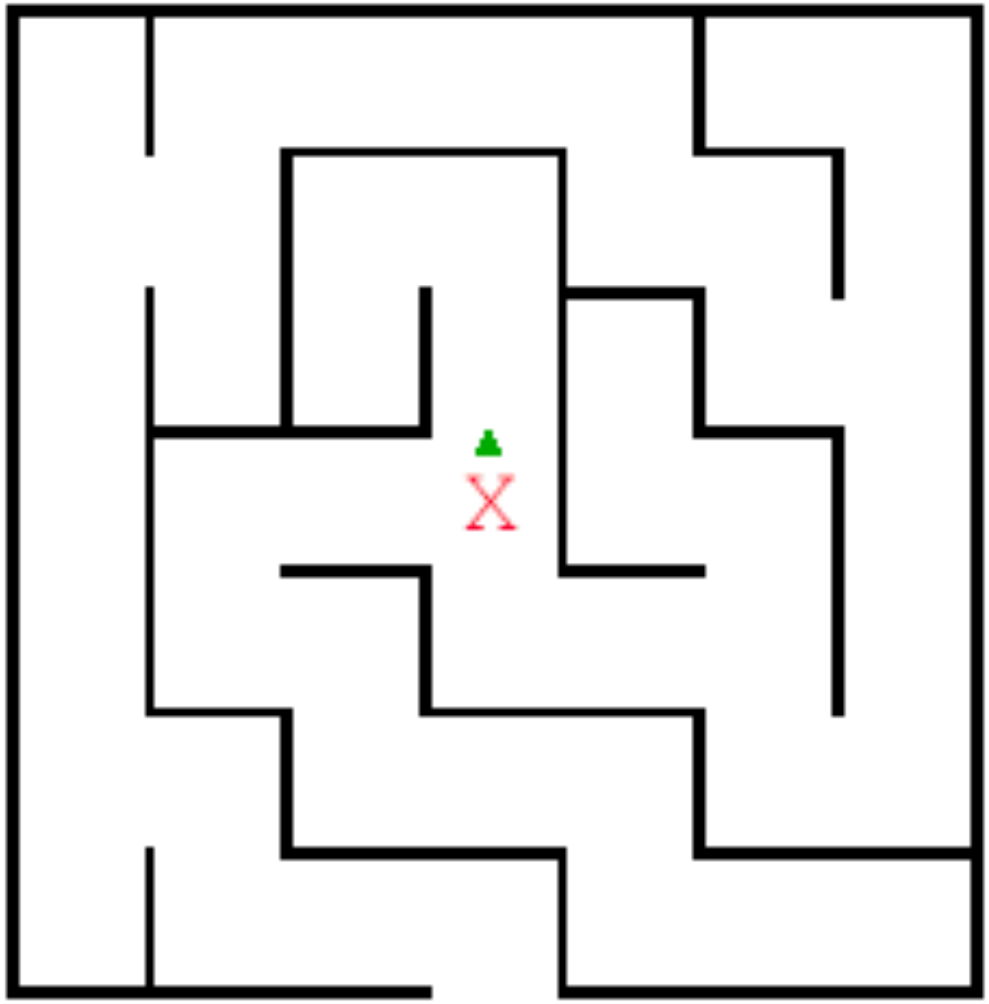
Por tanto el siguiente movimiento que podemos hacer es hacia el Oeste

Backtracking en Acción



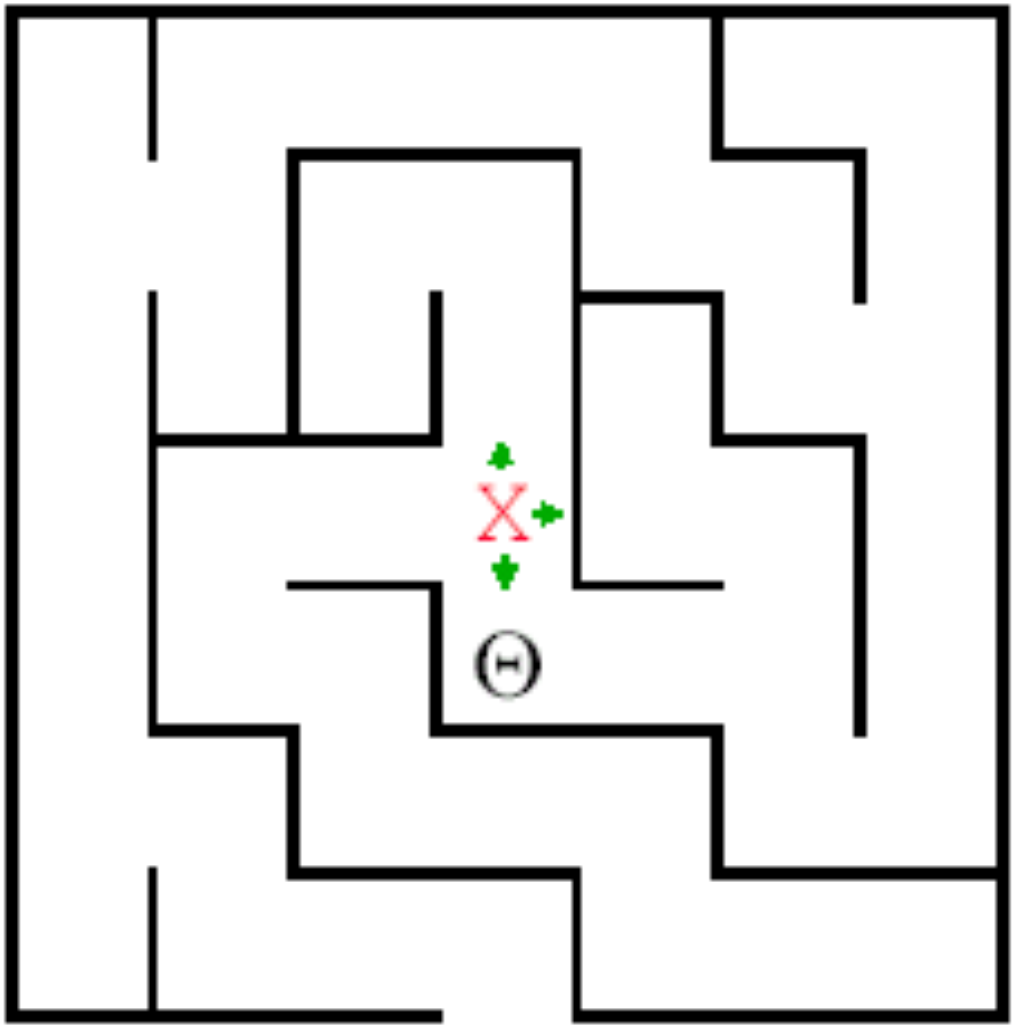
Este camino llega a un nodo (final) muerto .
¡Por tanto es el momento de hacer un backtrack!

Backtracking en Acción



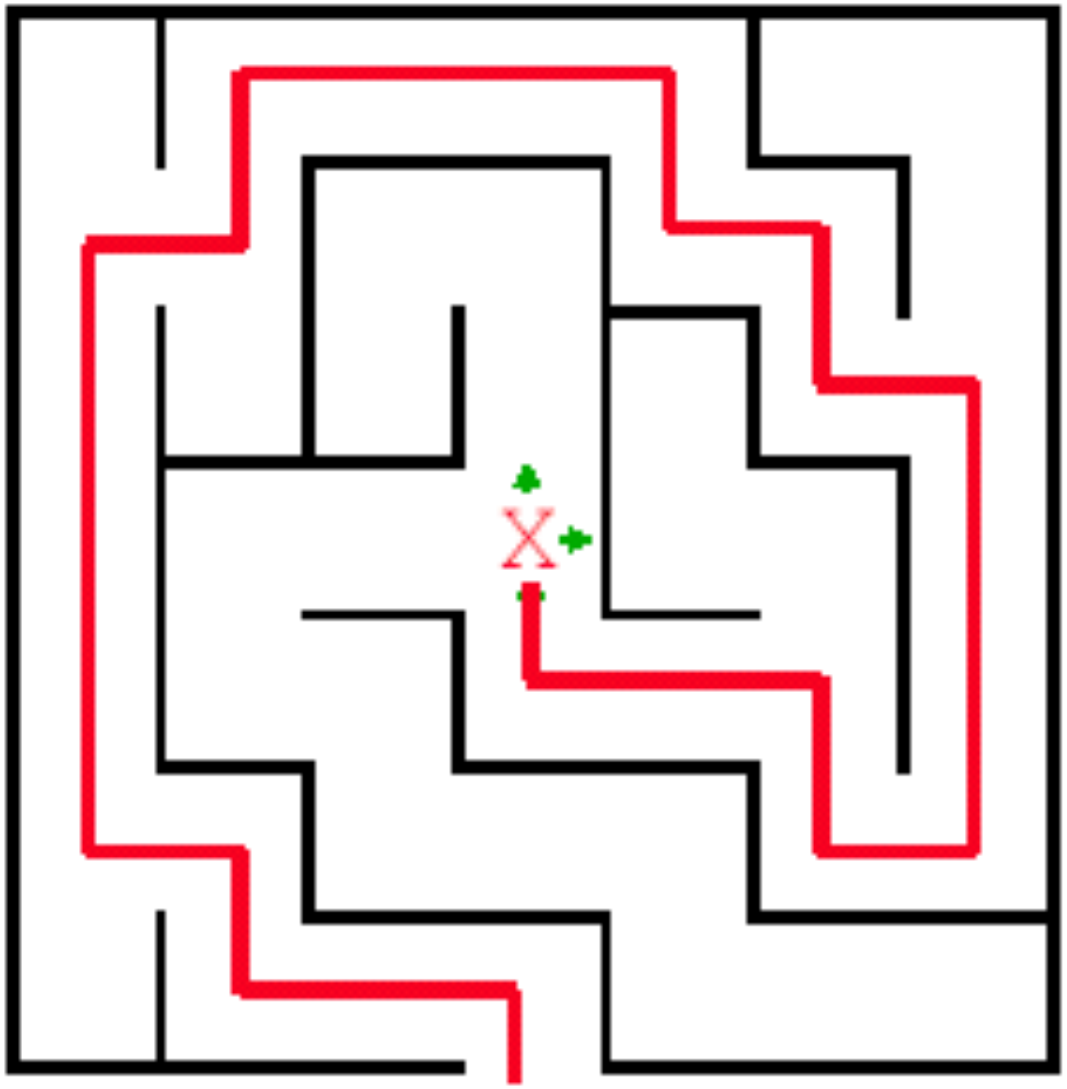
Se realizan sucesivas llamadas recursivas hasta volvernos a encontrar aquí

Backtracking en Acción



Intentamos ahora el Sur

Primer camino que se encuentra



Laberinto

- El problema consiste en encontrar la salida de un laberinto. Más concretamente, supondremos que el laberinto se representa mediante una matriz bidimensional de tamaño n×n. Cada posición almacena un valor 0 si la casilla es transitable y cualquier otro valor si la casilla no es transitable.
- Los movimientos permitidos son a casillas adyacentes de la misma fila o la misma columna. Podemos suponer que las casillas de entrada y salida del laberinto son (0,0) y (n − 1,n − 1) respectivamente.
- Por tanto el problema consiste en, dada una matriz que representa el laberinto, encontrar si existe un camino para ir desde la entrada hasta la salida.
- Diseñar e implementar un algoritmo backtracking para resolver el problema.
- Modificar el algoritmo para que encuentre el camino más corto. Realizar un estudio empírico de la eficiencia de los algoritmos.

Laberinto

- La representación de una solución parcial será una matriz (como el propio laberinto) donde iremos indicando en cada casilla el orden en que ha sido visitada (en vez de la forma más habitual en este tipo de algoritmos que consistiría en un vector donde en la posición k-ésima indicaríamos las coordenadas de la casilla que visitamos en la iteración k).
- Dada una posición, generaremos los (hasta 4) posibles movimientos, comprobando que son posibles (no nos salimos de los límites del laberinto ni atravesamos una casilla no transitable, ni visitamos una casilla ya visitada).
- Si llegamos a la posición (n − 1,n − 1) entonces hemos encontrado un camino y terminamos; sino continuamos recursivamente haciendo otra iteración. Cuando no se pueda continuar desde alguna posición, se produce la vuelta atrás.

Laberinto

- El algoritmo quedaría de la siguiente forma:

```
laberinto(k) {
mov=0; //con valores de 1 a 4 para indicar los movimientos (arriba, abajo, izqda y dcha)
while (!exito && (mov<4)) { //mientras no haya encontrado el camino y
                            //no haya agotado los movimientos posibles
   fila=fila+muevefil[mov];
   columna=columna+muevecol[mov];
   if ((0<=fila) && (fila<=n-1) && (0<=columna) && (columna<=n-1) && (lab[fila,columna]==0)) {
                                                                    //si es un movimiento válido
      lab[fila,columna]=k+1; //se mueve uno a esa casilla, en la iteración k-esima
      if ((fila==n-1) && (columna==n-1)) exito=true;
      else {
            laberinto(k+1);
            if (!exito) lab[fila,columna]=0; //si no hay exito hacemos la vuelta atrás
           }
    }
    fila=fila-muevefil[mov];
    columna=columna-muevecol[mov];
    mov++;
 }
 }
```

Laberinto

- Los movimientos permitidos se codifican en el array muevefil y muevecol antes de llamar a laberinto, de la siguiente manera:

```
muevefil[0]=1; muevecol[0]=0; //abajo
muevefil[1]=0; muevecol[1]=1; //derecha
muevefil[2]=0; muevecol[2]=-1; //izquierda
muevefil[3]=-1; muevecol[3]=0; // arriba
```

- El orden en que se intentan los movimientos puede ser importante. En este caso como queremos ir de la casilla (0,0) a la $(n-1,n-1)$, parece razonable intentar primero ir hacia abajo y hacia la derecha.
- La llamada a laberinto se haría de la siguiente manera:

```
exito = false;
fila=0;
columna=0;
laberinto(1);
```

- Si queremos un algoritmo para encontrar el camino mas corto, no pararemos cuando encontremos una solución, sino que seguimos la exploración.
- Se puede realizar una poda para no explorar soluciones parciales que ya tengan una longitud mayor que la mejor hallada hasta el momento

Laberinto: Algoritmo para encontrar el camino mas corto

```
laberintomin(k) {
mov=0;
while (mov<4) { //mientras no haya agotado los movimientos posibles
   fila=fila+muevefil[mov];
   columna=columna+muevecol[mov];
   if ((0<=fila) && (fila<=n-1) && (0<=columna) && (columna<=n-1) && (lab[fila][columna]==0)) {
                                                                    //si es un movimiento válido
      lab[fila][columna]=k+1; //se mueve uno a esa casilla, en la iteración k-esima
      if ((fila==n-1) && (columna==n-1)) {
         minimo=k; //almacenamos el camino mas corto hasta ahora
         for (i = 0; i < n; i++)
          for (j = 0; j < n; j++)
            minlab[i][j]=lab[i][j];
      }
      else if (k<=minimo) laberintomin(k+1); //aqui se hace una poda si la solucion parcial
                                             //ya es peor que la actual
      lab[fila][columna]=0;
   }
   fila=fila-muevefil[mov];
   columna=columna-muevecol[mov];
   mov++;
}
}
```

- La variable mínimo almacena la longitud del mejor camino encontrado hasta ahora. Si la longitud del camino que se esta explorando actualmente llega a superar ese valor, entonces no es necesario continuar por ese camino puesto que no podra ser el mas corto.

División en dos equipos

- Se desea dividir un conjunto de n personas para formar dos equipos que competirán entre sí. Cada persona tiene un cierto nivel de competición, que viene representado por una puntuación (un valor numérico entero).
- Con el objeto de que los dos equipos tengan una capacidad de competición similar, se pretende construir los equipos de forma que la suma de las puntuaciones de sus miembros sea la misma.
- Diseñar e implementar un algoritmo backtracking para resolver, si es posible, este problema.
- Mejorarlo usando alguna técnica de poda. Realizar un estudio empírico de la eficiencia de los algoritmos.

División en dos equipos

- Representamos el conjunto de personas como un vector p[i], i = 0, …,n-1, donde p[i] es la puntuación del individuo i-ésimo.
- Las posibles soluciones del problema se representan mediante otro vector v de tamaño n, donde v[i] puede tomar los valores 0 ó 1, indicando que la i-ésima persona se asigna al equipo 1 o al equipo 2, respectivamente.
- Para que el problema pueda tener solución y el conjunto de puntuaciones pueda dividirse en dos subconjuntos que sumen lo mismo, es obviamente necesario que la suma de todas las n puntuaciones,

$$\sum_{i=1}^{n} p[i]$$

  sea par (lo cual supondremos).
- El algoritmo procederá de la forma habitual: en cada etapa k se intentará asignar la k-ésima persona a uno de los dos grupos posibles.
- Al llegar a la última persona, si la suma de las puntuaciones de ambos grupos es igual, tenemos la solución; en caso contrario se hace la vuelta atrás.

División en dos equipos

```
formaequipos(k) {

for (i=0; i<2; i++) {
  v[k]=i;
  suma[i]=suma[i]+p[k];
  if (k<n-1) formaequipos(k+1);
  else if (suma[0]==suma[1]) {
          exito = true;
          devolver v; //obtenida una solucion
       }
  suma[i]=suma[i]-p[k];
}
}
```

- La llamada inicial al procedimiento sería,

  exito = false;

  formaequipos(0);

  if (!exito) print "no hay solucion";

División en dos equipos

- En este problema es posible hacerlo algo mejor imponiendo una restricción en forma de cota que posibilite podar nodos del árbol de expansión que sepamos que no conducen a la solución.
- Así, en el momento en que la suma de los elementos de un subconjunto sobrepase la mitad de la suma total, podemos dejar de explorar esa rama.

```
formaequipospoda(k) {

for (i=0; i<2; i++) {
  v[k]=i;
  suma[i]=suma[i]+p[k];
  if (suma[i]<= sumatotal/2) //si no es así se poda
     if (k<n-1) formaequipospoda(k+1);
     else if (suma[0]==suma[1]) {
             exito = true;
             devolver v; //obtenida una solucion
          }
  suma[i]=suma[i]-p[k];
}
}
```

Cena de gala

- Se va a celebrar una cena de gala a la que asistirán n invitados. Todos se van a sentar alrededor de una única gran mesa rectangular, de forma que cada invitado tendrá sentados junto a él a otros dos comensales (uno a su izquierda y otro a su derecha).
- En función de las características de cada invitado (por ejemplo categoría o puesto, lugar de procedencia,...) existen unas normas de protocolo que indican el nivel de conveniencia de que dos invitados se sienten en lugares contiguos (supondremos que dicho nivel es un número entero entre 0 y 100).
- El nivel de conveniencia total de una asignación de invitados a su puesto en la mesa es la suma de todos los niveles de conveniencia de cada invitado con cada uno de los dos invitados sentados a su lado.
- Se desea sentar a los invitados de forma que el nivel de conveniencia global sea lo mayor posible.
- Diseñar e implementar un algoritmo backtracking para resolver este problema y realizar un estudio empírico de su eficiencia.

Cena de gala: Planteamiento

- Numeraremos los asientos de la mesa como 0, 1 hasta n-1 (entendiendo que números contiguos representan asientos contiguos y que las posiciones 0 y n - 1 también son contiguas).
- También representaremos a cada invitado como un número entre 0 y n - 1. Un caso del problema será una matriz c, de tamaño n × n de enteros entre 0 y 100, donde c[i,j] contiene el nivel de conveniencia de sentar juntos a los invitados i y j.
- No es necesario que la matriz sea simétrica, aunque lo supondremos por simplicidad. Una posible solución del problema se modelizará como un vector a de tamaño n, donde a[i] es el invitado que es asignado al asiento i-esimo.

Cena de gala: Algoritmo

- El algoritmo procederá de la forma habitual: en cada etapa k se intentará asignar al k-ésimo asiento a una persona valida (es decir, que todavía no se haya sentado). El proceso continuará mientras siga habiendo personas sin sentar.
- Cuando no queden mas personas por sentar, el algoritmo calcula el nivel de conveniencia de esa solución, actualiza el mejor valor hallado si es el caso, y hace una vuelta atrás, seleccionando otra persona válida para sentar en la posición que le precede en el nivel anterior.
- Para evitar explorar soluciones que realmente son iguales aunque no lo parecen (por ejemplo 01234, 40123, 34012,...), se puede fijar un invitado cualquiera a un asiento cualquiera (por ejemplo el invitado 0 se sentará siempre en el asiento 0) y explorar las posibles asignaciones al resto de asientos.

Cena de gala: Algoritmo

```
sienta(k) {

for (j=1; j<n; j++) {
   a[k]=j;
   if factible(k)
     if (k<n-1) sienta(k+1);
     else {
           c=calculaconveniencia();
            if (c>mejorconveniencia) {
                              mejorconveniencia=c;
                              solucion=a;
                             }
           }
 }
 }
```

- La función calculaconveniencia simplemente suma los niveles de conveniencia entre invitados sentados juntos:

```
calculaconveniencia() {
suma=0;
for (i=1; i<n-1; i++)
    suma=suma+c[a[i],a[i-1]]+c[a[i],a[i+1]]; // desde 1 hasta n-2
suma=suma+c[a[0],a[n-1]]+c[a[0],a[1]]; //se añade los que están al lado de 0
suma=suma+c[a[n-1],a[n-2]]+c[a[n-1],a[0]]; //se añade los que están al lado de n-1
return suma;
}
```

Cena de gala: Algoritmo

- La comprobación de factibilidad de la solución parcial hallada hasta el momento simplemente veriíca que el invitado sentado en la posición k no haya sido asignado anteriormente (no esté ya sentado):

```
factible(k) {

for (i=1; i<k; i++)
   if (a[k]==a[i]) return false;
return true;
}
```

- La llamada inicial al procedimiento sería

```
mejorconveniencia=0;
a[0]=0;
sienta(1);
devolver solucion;
```

Transporte de mercancías

- Una empresa dispone de n centros de fabricación/distribución (cada uno situado en una localización diferente) y necesita abastecer n puntos de venta, situados en diferentes ciudades.
- La distancia entre el centro de distribución i y el punto de venta j es d(i, j), i,j = 1,...,n.
- La empresa desea abastecer cada punto de venta desde un único centro de distribución, y desea que la suma de las distancias entre cada centro de distribución y su punto de venta asignado sea lo más pequeñaa posible.
- Diseñar e implementar un algoritmo backtracking que resuelva este problema.
- Mejorarlo usando alguna técnica de poda. Realizar un estudio empírico de la eficiencia de los algoritmos.

Transporte de mercancías

- Suponemos una matriz d[i, j]; i,j = 0,…,n-1, donde d[i, j] es la distancia entre el i-ésimo centro de distribución y el j-ésimo punto de venta.
- Las posibles soluciones se representan mediante un vector v de tamaño n, donde v[i] puede tomar los valores 0, 1,…,n-1, e indica el punto de venta que se asigna al i-ésimo centro de distribución.
- Entonces en cada etapa k se intenta asignar al k-ésimo centro de distribución cada uno de los n puntos de venta, comprobando que la asignación sea factible, es decir que ese punto de venta no haya sido aún asignado a ningún centro.
- Una vez procesado el ultimo centro, se calcula la distancia total y se compara con la mejor distancia encontrada hasta ahora, reemplazándola si es menor.

Transporte de mercancías: Algoritmo

- El algoritmo quedaría así

```
asigna(k) {

for (j=0; j<n; j++) {
   v[k]=j;
   if factible(k)
     if (k<n-1) asigna(k+1);
     else {
          c=calculadistancia();
          if (c<mejordistancia) {
                                mejordistancia=c;
                                solucion=v;
                               }
          }
}
}
```

Transporte de mercancías: Algoritmo

- La función calculadistancia suma las distancias entre los centros y puntos asociados (1)
- La comprobación de factibilidad de la solución parcial hallada hasta el momento simplemente verifica que el punto de venta recién asignado no haya sido asignado anteriormente (2)
- La llamada inicial al procedimiento es (3)

```
calculadistancia() {

suma=0;
for (i=0; i<n; i++)
   suma=suma+d[i,v[i]];
return suma;
}
```

(1)

```
factible(k) {

for (i=0; i<k; i++)
   if (v[k]==v[i]) return false;
return true;
}
```

(2)

```
mejordistancia=infinito;
asigna(0);
devolver solucion;
```

(3)

Transporte de mercancías: Mejora

- Es posible mejorar algo mejor imponiendo una restricción en forma de cota que posibilite podar nodos del árbol de expansión que sepamos que no conducen a la solución óptima.
- Así, si vamos calculando el valor de la suma de distancias de las asignaciones hechas hasta el momento, cuando dicha suma sobrepase la mejor suma de distancias hallada hasta ahora, podemos dejar de explorar esa rama (en la llamada inicial hay que inicializar sumadistancias=0).

```
asignapoda(k) {

for (j=0; j<n; j++) {
   v[k]=j;
   sumadistancias=sumadistancias+d[k,j];
   if (factible(k) && (sumadistancias<=mejordistancia)) //si no se poda
      if (k<n-1) asignapoda(k+1);
      else {
            c=calculadistancia();
            if (c<mejordistancia) {
                              mejordistancia=c;
                              solucion=v;
                             }
          }
   sumadistancias=sumadistancias-d[k,j];
}
}
```

Estación de ITV

- Una estación de ITV consta de m líneas de inspección de vehículos iguales.
- Hay un total de n vehículos que necesitan inspección.
- En función de sus características, cada vehículo tardará en ser inspeccionado un tiempo t_i, i = 1, ... , n.
- Se desea encontrar la manera de atender a los n vehículos y acabar en el menor tiempo posible.
- Diseñar e implementar un algoritmo backtracking que determine cómo asignar los vehículos a las líneas.
- Mejorarlo usando alguna técnica de poda. Realizar un estudio empírico de la eficiencia de los algoritmos.

Estación de ITV: Planteamiento

- Sea el vector t[i], i = 0, ... , n−1, dando t[i] el tiempo de inspección del vehículo i-ésimo.
- Las posibles soluciones del problema se representarán mediante otro vector de enteros v de tamaño n, donde v[i] puede tomar los valores 0,1, ... , m−1, indicando la línea de inspección a la que se asigna el i-ésimo vehículo.
- Se acabará en el momento en que se termine de atender al último vehículo.
- Por tanto el tiempo empleado por una asignación será el máximo de la suma de los tiempos empleados en los vehículos asignados a cada línea, y la solución óptima será aquella asignación que minimice ese tiempo máximo.

Estación de ITV: Algoritmo

- El algoritmo procederá de la forma habitual: en cada etapa k se intentará asignar el k-ésimo vehículo a cada una de las m líneas de inspección. Al llegar al último vehículo, se calcula el tiempo requerido y se compara con el mejor tiempo encontrado hasta ahora, reemplazándolo si es menor.

```
asignalinea(k) {

for (j=0; j<m; j++) {
   v[k]=j;
   sumatiempos[j]=sumatiempos[j]+t[k];
   if (k<n-1) asignalinea(k+1);
   else {
         t=calculatiempo();
         if (t<mejortiempo) {
                              mejortiempo=t;
                              solucion=v;
                             }
        }
   sumatiempos[j]=sumatiempos[j]-t[k];
}
}
```

Estación de ITV : Algoritmo

- La función calculatiempo calcula el tiempo máximo empleado entre todas las líneas:

```
calculatiempo() {

max=sumatiempos[0];
for (j=1; j<m; j++)
   if (sumatiempos[j]>max) max=sumatiempos[j];
return max;
}
```

- La llamada inicial al procedimiento es

```
mejortiempo=infinito;
asignalinea(0);
devolver solucion;
```

Estación de ITV : Mejora

- En este problema es posible hacerlo algo mejor imponiendo una restricción en forma de cota que posibilite podar nodos del árbol de expansión que sepamos que no conducen a la solución. Así, en el momento en que la suma de los tiempos asignados a una línea sobrepase el mejor tiempo hallado hasta ahora, podemos dejar de explorar esa rama

```
asignalineapoda(k) {

for (j=0; j<m; j++) {
   v[k]=j;
   sumatiempos[j]=sumatiempos[j]+tiempo[k];
   if (k<n-1) {
      if (sumatiempos[j] <= mejortiempo) asignalineapoda(k+1); //si no se poda
   }
   else {
         t=calculatiempo();
         if (t<mejortiempo) {
                              mejortiempo=t;
                              solucion=v;
                             }
        }
   sumatiempos[j]=sumatiempos[j]-tiempo[k];
}
}
```