Técnicas de diseño de algoritmos
Búsqueda exhaustiva

Luis Javier Rodríguez Fuentes
Amparo Varona Fernández

Departamento de Electricidad y Electrónica
Facultad de Ciencia y Tecnología, UPV/EHU
luisjavier.rodriguez@ehu.es
amparo.varona@ehu.es

OpenCourseWare 2015
Campus Virtual UPV/EHU

Búsqueda exhaustiva: Indice

Búsqueda exhaustiva: Introducción

- Muchos problemas de optimización sólo pueden resolverse de manera exacta explorando todas las posibles combinaciones de elementos y tratando de encontrar aquellas que sean solución, eligiendo entre ellas una cualquiera o aquella que optimice una determinada función objetivo.
- Ciertos problemas admiten estrategias de selección heurística de elementos que conducen a soluciones óptimas. Es lo que hacen los algoritmos voraces.
- Pero las estrategias voraces conducen, en la mayor parte de los casos, a soluciones subóptimas. Recuérdese el caso del problema de la mochila (discreto), o el problema del viajante.
- Aparece entonces la necesidad de **volver atrás**, es decir, deshacer decisiones previas que se ha demostrado que no conducían a una solución (por el contrario, los algoritmos voraces nunca deshacen una decisión previamente tomada).

Búsqueda exhaustiva: Introducción

- Los algoritmos que exploran todas las posibles combinaciones de elementos se conocen como **algoritmos de fuerza bruta** o algoritmos de búsqueda exhaustiva.
- A pesar de su nombre, estos algoritmos raramente exploran todas las posibles combinaciones de elementos, sino que expanden las combinaciones más prometedoras, de acuerdo a un cierto criterio relacionado con la función objetivo que se pretende optimizar.
- La técnica de búsqueda más sencilla, conocida como **backtracking** o **vuelta atrás**, opera de manera recursiva, explorando en profundidad el árbol virtual de combinaciones de elementos y evaluando cada nuevo nodo para comprobar si es terminal (es decir, si es una posible solución).
- En ciertos casos, si un nodo terminal es solución, el algoritmo termina, retornando la combinación de elementos correspondiente a dicho nodo.
- En otros casos, se busca **la mejor solución**, de modo que el algoritmo evalúa la función objetivo en el nodo terminal, comprueba si es mejor que el óptimo almacenado hasta la fecha y sigue buscando.

Backtracking: esquema de diseño

Los algoritmos de vuelta atrás operan como sigue:

- La solución se construye de manera incremental, añadiendo un nuevo elemento en cada paso.
- El proceso de construcción de la solución es recursivo y equivale a recorrer en profundidad un árbol virtual en el que ciertos nodos representan respuestas parciales (incompletas) y otros (los que cumplen ciertas restricciones) son respuestas completas, es decir, potenciales soluciones al problema.
- Si un nodo respuesta s es solución, hay tres posibilidades:
  (1) el algoritmo termina, retornando s;
  (2) el algoritmo añade s al conjunto de soluciones y continúa el proceso de búsqueda;
  (3) el algoritmo evalúa la función objetivo en dicho nodo, $f(s)$, actualiza la solución óptima s_{opt} y continúa el proceso de búsqueda.
- Si un nodo respuesta no es solución, el algoritmo continúa el proceso de búsqueda. Ello implica deshacer decisiones previas (esto es, volver a una llamada recursiva anterior) y tomar en su lugar otras decisiones, que conducen a otras ramas del árbol y finalmente a nuevos nodos respuesta, que serán evaluados, etc.

Backtracking: esquema de diseño

Backtracking rápido (encontrar una solución)

```
def backtracking(nodo):
    if EsSolucion(nodo):
        return nodo
    for v in Expandir(nodo):
        if EsFactible(v):
            s = backtracking(v)
            if s!=None:
                return s
    return None
```


Backtracking: esquema de diseño

Backtracking exhaustivo (encontrar todas las soluciones)

```
def backtracking(nodo):
    lista_soluciones = []
    if EsSolucion(nodo):
        lista_soluciones.append(nodo)
    for v in Expandir(nodo):
        if EsFactible(v):
            ls = backtracking(v)
            lista_soluciones.extend(ls)
    return lista_soluciones
```

Backtracking: esquema de diseño

Backtracking para optimización (encontrar la mejor solución)

```
def backtracking(nodo):
    if EsSolucion(nodo):
        mejor_sol = nodo
    else:
        mejor_sol = None
    for v in Expandir(nodo):
        if EsFactible(v):
            ms = backtracking(v)
            if mejor_sol==None or f(ms)>f(mejor_sol):
                mejor_sol = ms
    return mejor_sol
```


Backtracking: esquema de diseño

Elementos principales del esquema de diseño

- `nodo`: almacena el conjunto de elementos (o decisiones) que conforman una rama del árbol (virtual) de búsqueda. Corresponde a una respuesta parcial o completa al problema planteado.
- `Expandir(nodo)`: devuelve la lista de nodos que resulta al añadir un nuevo elemento al conjunto de elementos representado por `nodo`. Se supone que existe un número finito de elementos con el que expandir cada nodo (en caso contrario, esta estrategia no sería viable computacionalmente).
- `EsFactible(nodo)`: devuelve un booleano que indica si es posible construir una solución al problema a partir del conjunto de elementos representado por `nodo` (en muchas ocasiones, los elementos de `nodo` serán incompatibles con las restricciones impuestas a una solución).
- `EsSolucion(nodo)`: devuelve un booleano indicando si el conjunto de elementos representado por `nodo` es solución al problema planteado.
- `f(nodo)`: función objetivo. Sin pérdida de generalidad, se supone que la mejor solución es aquella que maximiza el valor de `f`.

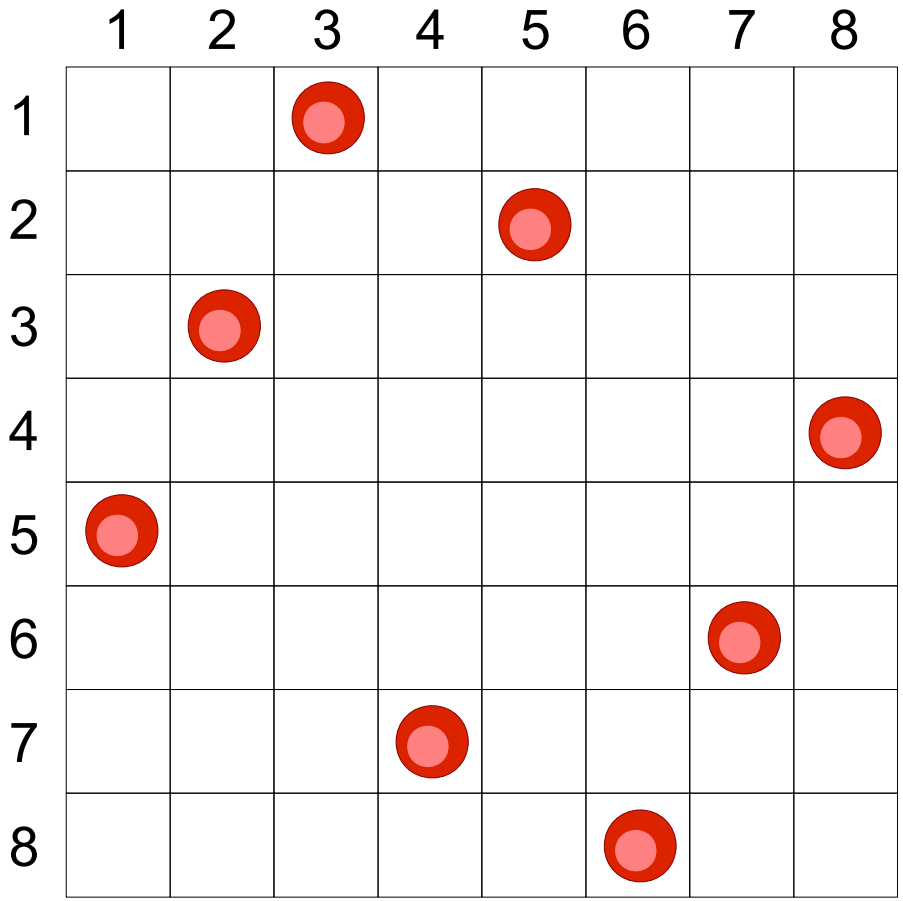
El problema de las n reinas

Enunciado: Considerese un tablero de ajedrez de $n \times n$ casillas (típicamente, $n = 8$). Se trata de colocar n reinas en el tablero sin que se amenacen entre sí. Basta con encontrar una solución (esquema de backtracking *rápido*).

- Un nodo consistirá en un vector c de tamaño $k \leq n$, tal que $c[j]$ será la columna que ocupa la reina en la fila j, $\forall j \in [1, k]$.
- La estrategia de construcción de la solución (función `Expandir()`) consistirá en expandir el vector c poniendo una reina en la siguiente fila del tablero.
- La función `EsFactible()` se encarga de comprobar que cada nueva reina se coloca de modo que no amenaza a las anteriores, evitando las columnas y diagonales ya cubiertas (esto reduce enormemente el coste).
- Por otra parte, un nodo será solución si $k = n$.
- Como el esquema es un backtracking rápido, el algoritmo termina en cuanto encuentra una solución.

El problema de las *n* reinas

Una de las soluciones para un tablero de 8×8



El problema de las *n* reinas

Implementación en lenguaje Python

```
def EsFactible(k,j,c):
    for i in range(k):
        if c[i]==j or (i-c[i])==(k-j) or (i+c[i])==(k+j):
            return False
    return True

def reinas(c,n):
    k = len(c)
    if k == n:
       return c
    for j in range(n):
       if EsFactible(k,j,c):
           v = c[:] + [j]
           s = reinas(v,n)
           if s != None:
               return s
    return None
```

Si se generasen todas las posiciones, la complejidad sería del orden de n^n.
Pero, ¿cuál es el coste en la práctica? Por ejemplo, para $n = 8$, $n^n = 16{,}777{,}216$, pero son necesarias tan sólo 114 llamadas para encontrar una solución.

El problema de la suma de conjuntos de enteros

Enunciado: Considerese un vector $v = (v_1, v_2, \ldots, v_n)$ con n enteros positivos distintos. El problema de la suma consiste en encontrar todos los subconjuntos de v cuya suma sea exactamente M.

- Las soluciones tendrán forma de tupla: $x = (x_1, x_2, \ldots, x_n)$, con $x_i \in \{0, 1\}$, y deberán cumplir: $\sum_{i=1}^{n} x_i v_i = M$
- Para resolver este problema, se aplicará un backtracking exhaustivo, que irá considerando todas las posibilidades (incluir/no incluir cada elemento en la solución: $x_i = 1$ y $x_i = 0$) a partir de una tupla vacía, hasta llegar a una tupla de tamaño n, que será evaluada y, si suma M, añadida al conjunto de soluciones.
- Dada una respuesta parcial $(x_1, x_2, \ldots, x_{k-1})$, y una decisión local x_k, con $k \leq n$, la función `EsFactible()` comprobará que no se ha sobrepasado y que aún es posible alcanzar la suma M:

$$\sum_{i=1}^{k-1} x_i v_i + x_k v_k \leq M$$

$$\sum_{i=1}^{k-1} x_i v_i + x_k v_k + \sum_{i=k+1}^{n} v_i \geq M$$

El problema de la suma de conjuntos de enteros

Implementación en lenguaje Python

```
def precompute_table(v):
    global t          # t[k] = sum_{i=k+1-->len(v)-1} v[i]
    t=[0]
    for i in range(len(v)-1,0,-1):
        t.insert(0,t[0]+v[i])

def suma_enteros(x,v,suma,M):
    global t          # t[k] = sum_{i=k+1-->len(v)-1} v[i]
    k = len(x)
    n = len(v)
    if k == n:        # es solucion: suma = M
        return [x]
    ls = []
    # if EsFactible x[k]=1
    if suma+v[k] <= M and suma+v[k]+t[k] >= M:
        ls = ls + suma_enteros(x[:]+[1],v,suma+v[k],M)
    # if EsFactible x[k]=0
    if suma+t[k] >= M:
        ls = ls + suma_enteros(x[:]+[0],v,suma,M)
    return ls
```

El problema de la mochila (discreto)

Enunciado: Considerese una mochila capaz de albergar un peso máximo M, y n elementos con pesos $p_1, p_2, \ldots, p_n$ y beneficios $b_1, b_2, \ldots, b_n$. Se trata de encontrar qué combinación de elementos, representada mediante la tupla $x = (x_1, x_2, \ldots, x_n)$, con $x_i \in \{0, 1\}$, maximiza el beneficio:

$$f(x) = \sum_{i=1}^{n} b_i x_i \tag{1}$$

sin sobrepasar el peso máximo M:

$$\sum_{i=1}^{n} p_i x_i \leq M \tag{2}$$

- En este caso, se aplicará un esquema de backtracking para optimización, que explorará todas las posibles soluciones a partir de una tupla prefijo de tamaño k, con un peso restante r, y devolverá la tupla que maximiza la función objetivo (1).
- La función `EsFactible()` simplemente comprobará la condición (2).

El problema de la mochila (discreto)

Implementación en lenguaje Python

```
def mochila_d_bt(x,r,p,b):
    k = len(x)
    n = len(p)
    if k == n:
        return x,0
    max_b = 0
    mejor_sol = x[:] + [0]*(n-k)
    for i in range(k,n):
        if p[i] <= r:
            x_new = x[:] + [0]*(i-k) + [1]
            ms,b_ms = mochila_d_bt(x_new,r-p[i],p,b)
            if b[i] + b_ms > max_b:
                max_b = b[i] + b_ms
                mejor_sol = ms
    return mejor_sol,max_b
```

El problema del viajante

Enunciado: Dado un grafo G no dirigido, ponderado, conexo y completo, y un vértice v_0, encontrar el *ciclo hamiltoniano* de coste mínimo que comienza y termina en v_0.

- Al igual que el problema de la mochila (discreto), se trata de un problema de optimización. Si el grafo tiene n vértices, la solución consistirá en una secuencia de vértices de longitud $n+1$: $x_1^{n+1} = (x_1, x_2, \ldots, x_n, x_{n+1})$ donde $x_i \neq x_j \ \forall i \neq j$, con la excepción de los vértices inicial y final: $x_{n+1} = x_1 = v_0$. Dicha solución tendrá asociado un coste $C(x_1^{n+1}) = \sum_{i=1}^{n} c(x_i, x_{i+1})$.
- La solución se irá construyendo de forma incremental, a partir de un recorrido parcial de k vértices, $x_1^k = (x_1, \ldots, x_k)$, con un coste acumulado $C(x_1^k)$. Cada decisión local consistirá en elegir el siguiente vértice x_{k+1} del recorrido. La función `EsFactible()` simplemente comprobará que x_{k+1} no está en x_1^k.
- Después de cada decisión, se llamará recursivamente al mismo algoritmo, con la secuencia x_1^{k+1} y el coste acumulado $C(x_1^{k+1})$. Esta llamada retornará la mejor solución alcanzable con dicho prefijo, así como su coste. De todas las soluciones obtenidas de esta forma, el algoritmo retornará la de menor coste.
- Por último, si $k = n$, el algoritmo completará el ciclo hamiltoniano haciendo $x_{n+1} = v_0$ y $C(x_1^{n+1}) = C(x_1^n) + c(x_n, v_0)$, y retornará el par $(x_1^{n+1}, C(x_1^{n+1}))$.

El problema del viajante

Implementación en lenguaje Python

```
# n: numero de vertices del grafo
# c: costes de los arcos: c[(i,j)] = c[(j,i)] (grafo no dirigido)
# x: prefijo de longitud k (inicialmente, x=[v0])
# C: coste acumulado de x (inicialmente, C=0)
def viajante_bt(x,C,n,c):
    k = len(x)
    if k == n: # x es solucion
        return x[:]+[x[0]], C+c[(x[n-1],x[0])]
    min_c=None
    for v in range(n):
        if v not in x: # v es factible
            new_x = x[:] + [v]
            new_C = C + c[(x[k-1],v)]
            sol,c_sol = viajante_bt(new_x,new_C,n,c)
            if min_c == None or c_sol < min_c:
                min_c = c_sol
                mejor_sol = sol
    return mejor_sol, min_c
```


Juegos

- Un juego establece unas reglas según las cuales dos o más jugadores actúan en un entorno complejo para alcanzar un objetivo.
- El desarrollo del juego puede implicar tanto la colaboración como la competición entre jugadores o grupos de jugadores.
- En ciertos juegos, los objetivos de los jugadores no son mutuamente excluyentes, lo que significa que varios jugadores podrían alcanzarlos y no habría un único ganador.
- En otros juegos, sin embargo, todos los jugadores tienen el mismo objetivo y sólo uno de ellos puede alcanzarlo. En estos casos, independientemente de la estrategia puntual de cada uno de ellos, todos compiten contra todos.
- En esta sección consideraremos sólo este último tipo de juegos, más en concreto **juegos entre dos jugadores que actúan alternativamente para maximizar sus opciones de alcanzar el objetivo**. En este esquema se incluyen juegos como el 3 en raya, el 4 en raya, el go, el ajedrez, las damas, etc.

Búsqueda minimax

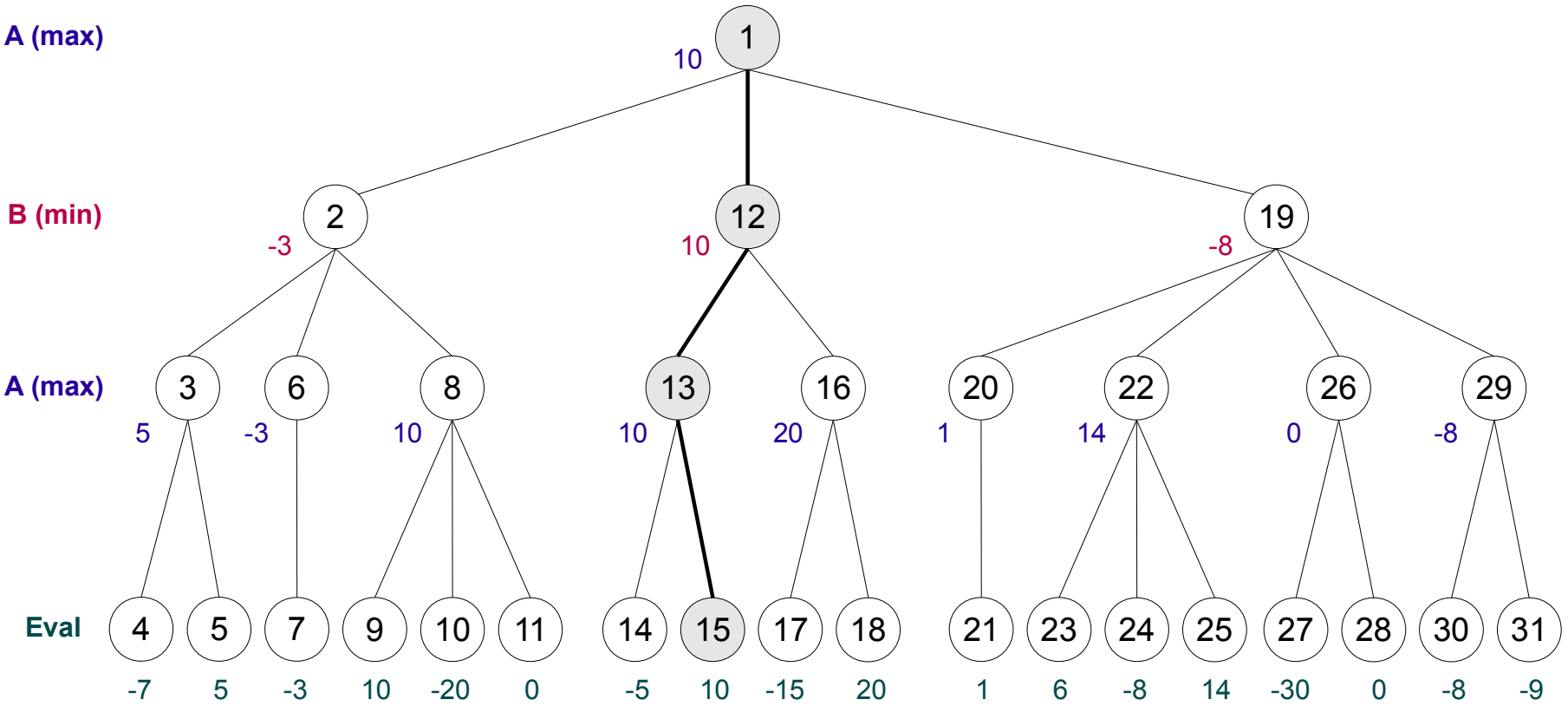
- El problema se podría plantear así: dada una posición de partida, en la que el jugador A debe realizar un movimiento, **¿cuál es el movimiento que maximiza sus opciones de ganar?**
- Debe tenerse en cuenta que:
  (1) No es posible expandir completamente el árbol de movimientos de uno y otro jugador, porque es demasiado grande o incluso infinito (se podrían producir bucles en la secuencia de jugadas); así pues, **el análisis de las jugadas debe limitarse a un cierto nivel de profundidad** k.
  (2) **Cada posición** s **se evalúa mediante una función** $f(s)$, que toma valores positivos para posiciones de ventaja del jugador A, valores negativos para posiciones de ventaja del jugador B y 0 para posiciones neutras.
- La **técnica de análisis (búsqueda minimax)** consistirá en expandir el árbol hasta una profundidad k, evaluar las posiciones obtenidas en dicho nivel y aplicar una estrategia de selección de jugadas **que maximiza (jugador A) y minimiza (jugador B) alternativamente el valor de la función de evaluación** al propagarlo hacia arriba en el árbol.

Búsqueda minimax

- Como el análisis es parcial (sólo alcanza profundidad k) y la función de evaluación proporciona tan sólo una medida aproximada de la ventaja relativa de cada jugador en una posición, no podemos asegurar que la estrategia sea óptima.
- Se dice, por ello, que la búsqueda minimax es una **estrategia heurística**.
- En todo caso, si la profundidad es lo bastante grande y la función de evaluación está bien definida, esta estrategia permitirá elegir una de las mejores jugadas posibles.

Búsqueda minimax

Ejemplo de árbol minimax (profundidad 3):



Búsqueda minimax: ajedrez

Ajedrez: Juegan blancas (max)

```
def blancas(posicion,k):
    if k == 0:
        return Eval(posicion)
    lista_posiciones = mueven_blancas(posicion)
    if len(lista_posiciones) == 0:
        return Eval(posicion)
    max_v = None
    for p in lista_posiciones:
        v = negras(p,k-1)
        if max_v == None or v > max_v:
            max_v = v
    return max_v
```

Búsqueda minimax: ajedrez

Ajedrez: Juegan negras (min)

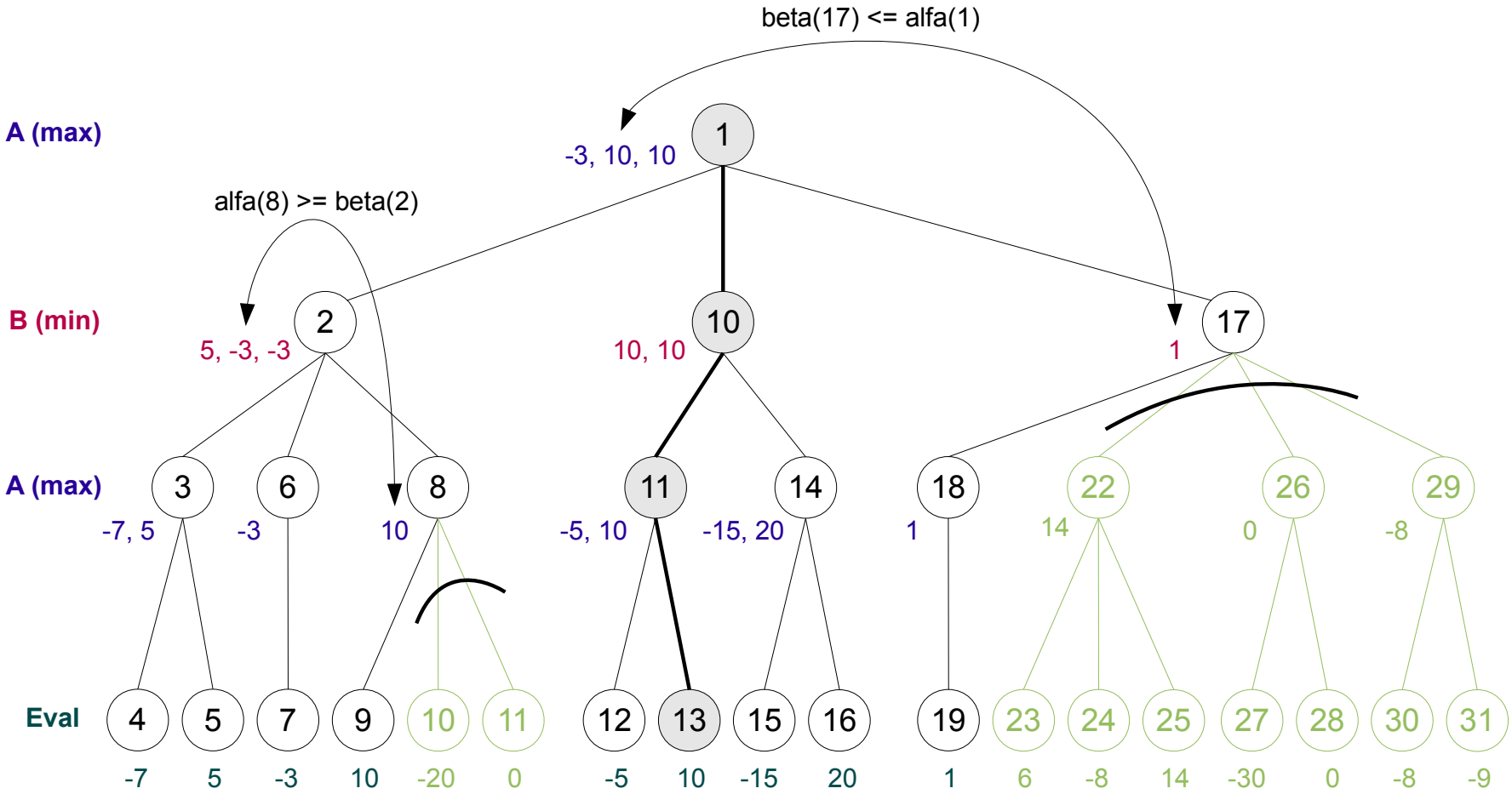
```
def negras(posicion,k):
    if k == 0:
        return Eval(posicion)
    lista_posiciones = mueven_negras(posicion)
    if len(lista_posiciones) == 0:
        return Eval(posicion)
    min_v = None
    for p in lista_posiciones:
        v = blancas(p,k-1)
        if min_v == None or v < min_v:
            min_v = v
    return min_v
```


Búsqueda minimax con poda alfa-beta

- En la estrategia minimax, el valor de un nodo maximizador (jugador A) irá tomando valores cada vez mayores. Así, si en un momento dado dicho valor supera el del nodo minimizador (jugador B) del que es hijo, no tiene sentido seguir explorándolo, porque ya nunca será elegido por el jugador B.
- La observación es simétrica con respecto a un nodo minimizador, que irá tomando valores cada vez más pequeños. Si su valor llega a ser menor que el del nodo maximizador del que es hijo, ya nunca será elegido por el jugador A y no tiene sentido seguir explorándolo.
- Así pues, ciertas ramas podrán podarse de acuerdo a sendas cotas: alfa (que nunca decrece) para los nodos MAX y beta (que nunca crece) para los nodos MIN. Ambas cotas se van actualizando durante el proceso de búsqueda.
- La **poda alfa-beta** se concreta en las dos reglas siguientes:
  - En un nodo MIN, la búsqueda termina en cuanto su valor beta sea menor o igual que el valor alfa de su antecesor MAX.
  - En un nodo MAX, la búsqueda termina en cuanto su valor alfa sea mayor o igual que el valor beta de su antecesor MIN.
- La poda alfa-beta está asociada a procesos de búsqueda en profundidad, ya que al menos una parte del árbol de búsqueda debe ser expandida hasta la máxima profundidad, pues los valores de alfa y beta están basados en las evaluaciones de nodos en ese nivel.

Búsqueda minimax con poda alfa-beta

Ejemplo de árbol minimax con poda alfa-beta (profundidad 3):



Búsqueda minimax con poda alfa-beta: ajedrez

Ajedrez: Juegan blancas (max)

```
def blancas(posicion,k,beta):
    if k == 0:
        return Eval(posicion)
    lista_posiciones = mueven_blancas(posicion)
    if len(lista_posiciones) == 0:
        return Eval(posicion)
    alfa = None
    for p in lista_posiciones:
        v = negras(p,k-1,alfa)
        if alfa == None or v > alfa:
            alfa = v
        if beta != None and alfa >= beta:
            break
    return alfa
```

Búsqueda minimax con poda alfa-beta: ajedrez

Ajedrez: Juegan negras (min)

```
def negras(posicion,k,alfa):
    if k == 0:
        return Eval(posicion)
    lista_posiciones = mueven_negras(posicion)
    if len(lista_posiciones) == 0:
        return Eval(posicion)
    beta = None
    for p in lista_posiciones:
        v = blancas(p,k-1,beta)
        if beta == None or v < beta:
            beta = v
        if alfa != None and beta <= alfa:
            break
    return beta
```

Ramificación y poda: ideas generales

- Una **búsqueda en anchura** recorre los nodos de un grafo por niveles a partir de un nodo raíz: en primer lugar, los nodos conectados directamente con el nodo raíz (nivel 1); a continuación, los nodos conectados con el nodo raíz a través de un único nodo intermedio (nivel 2), etc.
- La estrategia de **ramificación y poda** generaliza la búsqueda en anchura manteniendo una estructura de nodos vivos (esto es, nodos creados pero no expandidos aún, que pueden estar en distintos niveles del árbol) y recorriéndola en un cierto orden *inteligente* (que no es ni en profundidad ni en anchura).
- Para ello, cada nodo vivo v es etiquetado con una estimación heurística de la probabilidad de encontrar una solución a partir de la solución parcial que dicho nodo representa. En problemas de optimización, se asocia a cada nodo una cota optimista del valor máximo que la función objetivo puede alcanzar al expandir dicho nodo: $f_{max}(v)$.
- Los nodos se ordenan entonces de acuerdo a dicho valor, lo que permite **explorar en primer lugar los nodos más prometedores**. Si lo que se busca es una solución cualquiera, esta estrategia reduce el número de nodos que es necesario expandir para encontrarla y, por tanto, el coste de la búsqueda.
- En **problemas de optimización**, a medida que van encontrándose soluciones completas y se dispone del valor de la función objetivo para la solución óptima, $f(s_{opt})$, es posible eliminar de la lista de nodos vivos (esto es, **podar**) aquellos nodos para los que $f_{max}(v) \leq f(s_{opt})$.

Ramificación y poda: esquema de resolución

Ramificación y poda para optimización

```
# f(v) retorna una cota optimista de la funcion objetivo si v no es solucion
# f(v) retorna el valor exacto de la funcion objetivo si v es solucion
def ramificacion_y_poda(raiz):
    fopt = None
    lista = [raiz]
    while len(lista)>0:       # mientras queden nodos vivos
        nodo = lista.pop()    # extraemos el mas prometedor
        if EsSolucion(nodo): # si es solucion, actualizamos fopt y podamos
            if fopt==None or f(nodo)>fopt:
                fopt = f(nodo)
                mejor_solucion = nodo
                while len(lista)>0 and f(lista[0])<=fopt:
                    lista.pop(0)
        else:                 # si no es solucion, lo expandimos
            for v in Expandir(nodo):
                if fopt==None or f(v)>fopt:  # si EsFactible(v)
                    i=0
                    while i<len(lista) and f(lista[i])<=f(v):
                        i=i+1
                    lista.insert(i,v) # se introduce (en orden) en la lista
    return mejor_solucion
```

El problema de la mochila (discreto)

- El problema de la mochila (discreto) podrá resolverse de manera más eficiente si se descartan aquellos nodos que no puedan superar el beneficio aportado por la mejor solución hasta ese momento.
- El planteamiento general del algoritmo será un poco distinto al presentado en el esquema general, porque se empleará una búsqueda en profundidad.
- La diferencia, con respecto al backtracking, está en el uso del **máximo global de la función objetivo** (el beneficio total, que va actualizándose a medida que avanza la búsqueda), para **podar aquellos nodos que no puedan superarlo**.
- Dado un nodo v, representado por un vector de k decisiones, con un peso restante disponible m, la **resolución del problema de la mochila (continuo)** para las $n - k$ decisiones restantes, con un peso límite m, permite obtener una **cota superior del beneficio** alcanzable a partir del nodo v.
- Para ello, sin pérdida de generalidad, se supone que los vectores de pesos y beneficios entán ordenados de mayor a menor ratio beneficio/peso.

El problema de la mochila (discreto)

Función auxiliar: devuelve el máximo beneficio que se puede alcanzar a partir de un prefijo de la solución de longitud `k` con un peso restante disponible `m`, suponiendo que es posible introducir fracciones de objetos en la mochila.

```
# algoritmo voraz para el problema de la mochila (continuo)
# p y b ordenados de mayor a menor ratio b[i]/p[i]
def cota_beneficio(k,m,p,b):
    n=len(p)
    p_ac = 0.0
    b_ac = 0.0
    for i in range(k,n):
        if p_ac + p[i] <= m:
            p_ac = p_ac + p[i]
            b_ac = b_ac + b[i]
        else:
            factor = (m-p_ac)/p[i]
            b_ac = b_ac+b[i]*factor
            break
    return b_ac
```

El problema de la mochila (discreto)

Implementación en lenguaje Python:

```
# x: respuesta parcial, con peso y beneficio acumulados p_x y b_x
# b_max: maximo beneficio alcanzado hasta el momento
# p y b ordenados de mayor a menor ratio b[i]/p[i]
def mochila_rp(x,p_x,b_x,b_max,p,b,M):
    ms,b_ms = None,b_max # ms: mejor solucion, b_ms: beneficio de ms
    k = len(x)
    n = len(p)
    if k == n:
        if b_x > b_ms:
            ms,b_ms = x,b_x
    else:
        for d in [0,1]: # ramificacion: se abren dos posibles ramas
            # poda en funcion de M y b_ms (que se va actualizando)
            cota_b = cota_beneficio(k+1,M-p_x-d*p[k],p,b)
            if p_x+d*p[k] <= M and b_x+d*b[k]+cota_b > b_ms:
                y = x[:]+[d]                    # y: expansion de x
                p_y = p_x+d*p[k]                # peso de y
                b_y = b_x+d*b[k]                # beneficio de y
                s,b_s = mochila_rp(y,p_y,b_y,b_ms,p,b,M)
                if b_s > b_ms:
                    ms,b_ms = s,b_s
    return ms,b_ms
```

El problema de la asignación de tareas

Enunciado: Dados n agentes y n tareas y una matriz de costes c de tamaño $n \times n$, tal que $c(a, t)$ es el coste de que el agente a lleve a cabo la tarea t, el problema consiste en **asignar una tarea a cada agente de manera que el coste total acumulado sea mínimo**. Cada agente realizará una sola tarea y todas las tareas deben ser realizadas.

- Existen $n!$ combinaciones distintas, demasiadas incluso para valores moderados de n.
- Se aplicará una estrategia de ramificación y poda, a partir de dos valores de referencia:
  1. *cma*: el **coste mínimo absoluto** alcanzable
  2. *cms*: el **coste de la mejor solución** obtenida hasta el momento
- Si la búsqueda encuentra una solución cuyo coste iguala *cma*, la búsqueda habrá finalizado, porque no es posible encontrar soluciones mejores.
- El valor de *cms* se va actualizando a medida que avanza la búsqueda y se utiliza para podar nodos v cuyo coste mínimo alcanzable $c_{min}(v)$ sea igual o superior a *cms*. Nótese que cada vez que se actualiza *cms* es necesario volver a revisar los nodos vivos para ver cuáles de ellos deben ser podados.
- Por otra parte, el **nodo más prometedor** (el nodo a expandir a continuación) será **aquel que tenga un valor de c_{min} más pequeño**.

El problema de la asignación de tareas: Ejemplo

Agentes	Tareas 1	2	3	4
A	11	12	18	40
B	14	15	13	22
C	11	17	19	23
D	17	14	20	18

(1) $cma = 54$
- Suma de los mínimos de fila $\rightarrow$ 49
- Suma de los mínimos de columna $\rightarrow$ 54

Nos quedamos con el máximo de ambos, ya que todos los agentes deben realizar una tarea y todas las tareas deben ser realizadas.

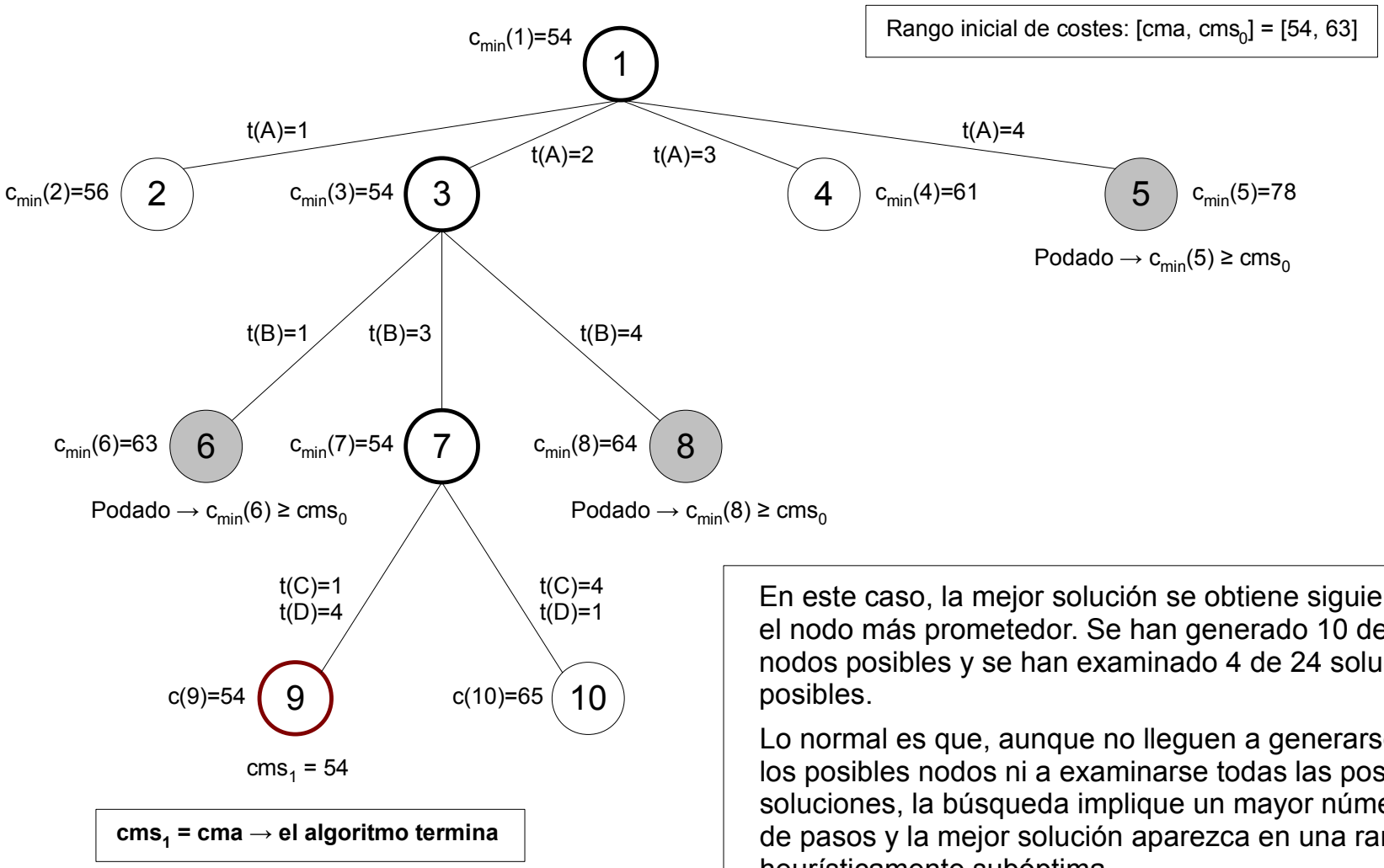
(2) Inicialización de $cms = 63$
- Diagonal principal $\rightarrow$ 63
- Diagonal opuesta $\rightarrow$ 87

Nos quedamos con la solución de coste mínimo.

(3) Inicialmente, el rango de costes es $[54, 63]$.

(4) El coste mínimo alcanzable a partir de un nodo v, $c_{min}(v)$, se calcula del mismo modo que cma:
- para los agentes y tareas que quedan por asignar en v, se suman los mínimos de fila por un lado: $s_{minf}(v)$, y los de columna por otro: $s_{minc}(v)$;
- $c_{min}(v)$ = coste de las asignaciones en v + máximo$(s_{minf}(v), s_{minc}(v))$.

El problema de la asignación de tareas: Ejemplo



En este caso, la mejor solución se obtiene siguiendo el nodo más prometedor. Se han generado 10 de 41 nodos posibles y se han examinado 4 de 24 soluciones posibles.

Lo normal es que, aunque no lleguen a generarse todos los posibles nodos ni a examinarse todas las posibles soluciones, la búsqueda implique un mayor número de pasos y la mejor solución aparezca en una rama heurísticamente subóptima.

El problema de la asignación de tareas

Implementación en lenguaje Python:

```
def asignar(c):
    n = len(c)
    ms, cms = init_ms(c)
    cma = compute_cma([],c)
    if cma<cms:
        nodo_raiz = [[],cma] # nodo raiz: [tupla vacia, cma global]
        lista = [nodo_raiz]  # lista de nodos vivos
    while len(lista)>0:      # mientras queden nodos vivos
        x, cma = lista.pop() # extraemos el nodo mas prometedor
        tareas = [t for t in range(n) if t not in x]
        for t in tareas:     # ramificacion
            x_new = x[:]+[t]
            cma_new = compute_cma(x_new,c)
            if cma_new<cms:  # nodo podado si cma_new >= cms
                if len(x_new)==n: # x_new es solucion
                    ms, cms = x_new, cma_new
                    # cms actualizado --> poda de nodos con cma>=cms
                    while len(lista)>0 and lista[0][1]>=cms:
                        lista.pop(0)
                else: # insercion en orden (de mayor a menor cma)
                    i=len(lista)-1
                    while i>=0 and lista[i][1]<cma_new:
                        i=i-1
                    lista.insert(i+1,[x_new,cma_new])
    return ms, cms
```

El problema de la asignación de tareas

Funciones auxiliares:

```
def init_ms(c):
    n = len(c)
    cdiag1 = 0
    cdiag2 = 0
    for i in range(n):
        cdiag1 += c[i][i]
        cdiag2 += c[i][n-i-1]
    if cdiag1<=cdiag2:
        return [i for i in range(n)], cdiag1
    else:
        return [n-i-1 for i in range(n)], cdiag2

def compute_cma(x,c):
    n = len(c)
    k = len(x)
    coste_x = 0
    for i in range(k):
        coste_x += c[i][x[i]]
    sum_minf = sum(min(c[i][j] for j in range(n) if j not in x) for i in range(k,n))
    sum_minc = sum(min(c[i][j] for i in range(k,n)) for j in range(n) if j not in x)
    return coste_x + max(sum_minf,sum_minc)
```