CC371:3.- Practica Calificada

Backtracking & Branch and Bound

Brando Miguel Palacios Mogollon

bpalaciosm@uni.pe

FACULTAD DE CIENCIAS

Universidad Nacional de Ingenieria
Escuela Profecional de Ciencias de la Computacion

August 13, 2020

Overview

Pregunta 1

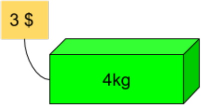
Pregunta 2

Motivación

Motivación

Solución: [1,0,1,1,0]

Si usa el
elemento 1



Enunciado

Considerando una mochila capaz de albergar MX de peso,sean los n elementos $(e_1, e_2, \cdots, e_n)$ con pesos $p_1, p_2, \cdots, p_n$ y beneficios $b_1, b_2, \cdots, b_n$. Se trata de encontrar qué combinación de esos elementos en la mochila que nos permita obtener la suma máxima de beneficios teniendo un limite de peso MX.

Análisis:

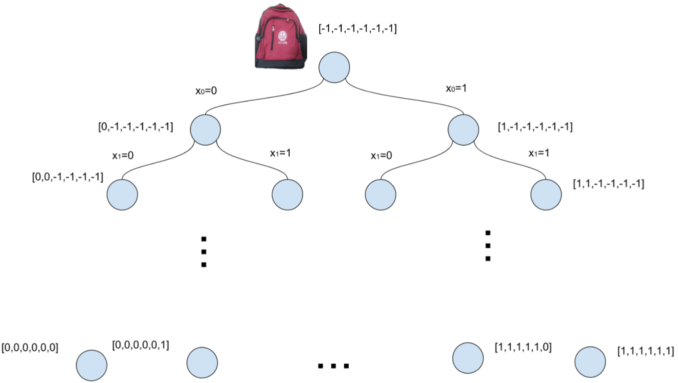
Podemos revisar esta combinación obteniendo como salida una tupla $x = (x_1, x_2, \cdots, c_n)$, con $x_i \in \{0, 1\}$, maximizan la función de beneficio que definiremos como:

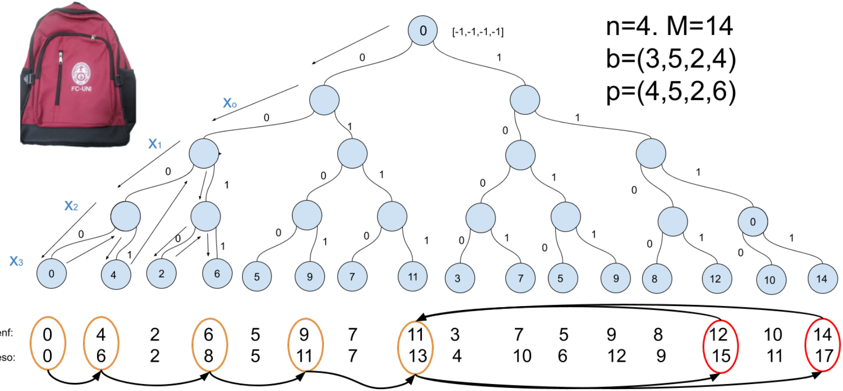
$$f(x) = \sum_{i=1}^{n} b_i x_i \tag{1}$$

Además, este no debe sobrepasar nuestra **cota** superior de un peso máximo MX:

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

por lo que tendremos una función que devolverá la tupla solución maximizando nuestra función objetivo (1) y una función que validara que dicha función cumple con la condición de (2).
Analizando la solución vemos que podemos describir el problema mediante un árbol de decisión de tipo binario, donde la profundidad sera el numero de elementos a elegir, el cual denotaremos como **etapa**.





FC-UNI
[-1,-1,-1,-1]
n=4. M=14
b=(3,5,2,4)
p=(4,5,2,6)
X_0
X_1
X_2
X_3
0 4 2 6 5 9 7 11 3 7 5 9 8 12 10 14
benf: 0 4 2 6 5 9 7 11 3 7 5 9 8 12 10 14
peso: 0 6 2 8 5 11 7 13 4 10 6 12 9 15 11 17

FACULTAD DE CIENCIAS

Variables: En un inicio se plantea tener valores constantes como el numero de elementos disponibles y el peso máximo de la mochila. Además la definición de cada uno de los elementos viene dada por una estructura que posee peso y beneficio.

```
const N ; // Numero de objetos
const MX; // Capacidad max de la mochila

TYPE Element = Elemento(Peso,beneficio);
Elementos = Array[1..n] of Elemento // Arreglo de objetos Elementos
solucion_actual, mochila_final = Array[1..n] of int;
```

Función MochilaBusc: Es la función principal donde se presenta la forma del Backtraking, existiendo una condición de aceptación denotada por la función **valido** , además mientras la etapa no llegue al valor final ($etapa == N - 1$) esta realizar una función recursiva aumentado la etapa en 1, caso contrario actualizara la solución con la función **actualizarSol**.

```
funcion mochilabusc(int solucion[],int etapa,Elemento Elementos[],
                    int mochila_final[],int peso_final,int beneficio_final){
        int i=0;
        if(etapa>N-1) return 0;
        do{
                solucion[etapa]=i;
                if(etapa==N-1)
                        actualizarSol(solucion,Elementos,mochila_final,peso_final,beneficio_final);
                else
                        mochilabusc(solucion,etapa+1,Elementos,mochila_final,peso_final,beneficio_final)
                i++;
        }while(solucion[etapa]!=1);
        solucion[etapa]=-1
}
```

Función actualizarSol: Esta actualiza los datos del vector *mochila_final* dependiendo de las condiciones del problema, no excediendo el peso de la mochila y buscando teniendo un máximo de beneficio.

```
funcion actualizarSol(int solucion[],Elemento Elementos[],int mochila_final[],
                      int peso_final,int benef_final){

        int benef_total=0;
        int peso_total =0;

        for(int i=0;i<N;i++){
                benef_total+=(solucion[i])*(Elementos[i].beneficio);
                peso_total+=(solucion[i])*(Elementos[i].peso);
        }
        if(peso_total<MX){
                if(benef_total>benef_final){
                        for(int i=0;i<N;i++){
                                mochila_final[i]=solucion[i];
                        }
                        benef_final=benef_total;
                        peso_final=peso_total;
                }
        }
}
```

Función valido: Una función que valida que la solución actual no a superado el peso de la mochila. Esta puede servir como optimización podando los nodos que excedan el peso pero seria un problema de "Branch & Bound" por lo que lo usaremos implicitamente en la función **actualizarSol**.

```
funcion valido(int solucion[],int etapa,Elemento Elementos[]){
        int i=0:
        int  peso_parcial=0;
        while(i<=etapa){
        peso_parcial+= Elementos[i].pesos;
        i++
        }
        return peso_parcial<MX;
}
```

El coste del algoritmo de vuelta atrás depende del número de nodos recorridos. En sentido estricto, el coste del algoritmo no depende del tamaño del problema puesto a que éste es constante, pero si que podemos estimar el coste analizando cómo es el árbol que recorremos. De manera que el coste se puede estimar muy por encima como de 2^n. La función **actualizarSol** presenta bucles inscritos en condiciones por lo que por simple inspección podemos ver que siendo $H(n)$ el tiempo de complejidad de la función esta presente un orden lineal, es decir: $H(n) \in \mathcal{O}(n)$.
De esto podemos apreciar que la complejidad de nuestro algoritmo en la función **MochilaBusc**:$T(n) = k(T(n-1) + O(n))$ para el caso de $T(0) = 1$ dado que es la condición de salida.

$$T(n) = k^n(T(n) + O(n))$$

Para este caso sera binario por lo que $O(2^n)$

Tambien llamado Puzzle Gem, Puzzle Boss, Juego de los Quince, Mystic Square.

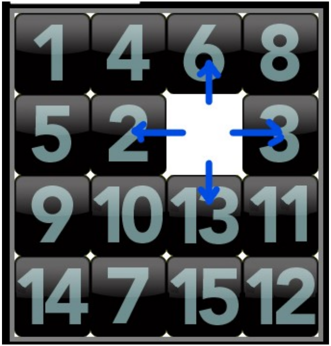
Figure 1: EE.UU. caricatura politica sobre la búsqueda de un candidato republicano a la presidencia en 1880

El juego fue idea por Sam Loyd en 1878. En el cual tenemos 16 casillas y 15 piezas.

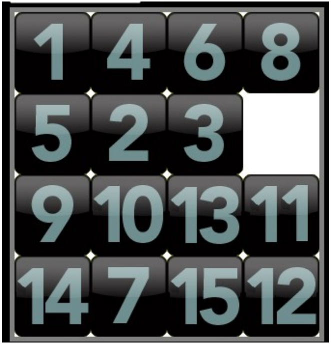
Figure 2: Puzzle de 15 de windows vista (2013)

Johnson Story (1879) demuestran que la mitad de las posiciones de salida para el n-puzzle son imposibles de resolver, sin importar cómo se hacen muchos movimientos.

Motivación



(a) Direcciones de espacio

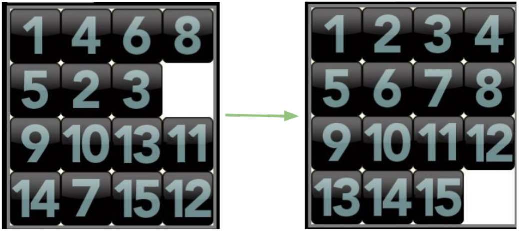


(b) Moviendo 3

Sea que se demuestra la paridad.
Si este cumple, buscar una secuencia de arreglos generara una complejidad $O(n^2)$ ademas, general cualquier secuencia, generara $O(n^3)$.
Sin embargo, buscar una secuencia más corta es NP-Completo.

Enunciado

Enunciado



Configuración Inicial   Configuración Objetivo

Análisis:

- La solución de fuerza bruta:
  - Hay $16! \approx 20,9 \times 10^12$ configuraciones, aunque **sólo (?) la mitad pueden alcanzarse**
- El mejor forma para resolverlo es: Algoritmo de busqueda A* $(O(ln(n)))$

Configuraciones alcanzables

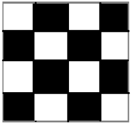
Teorema

Sea una configuración existente, la configuración objetivo es **alcanzable** si para esta configuración:

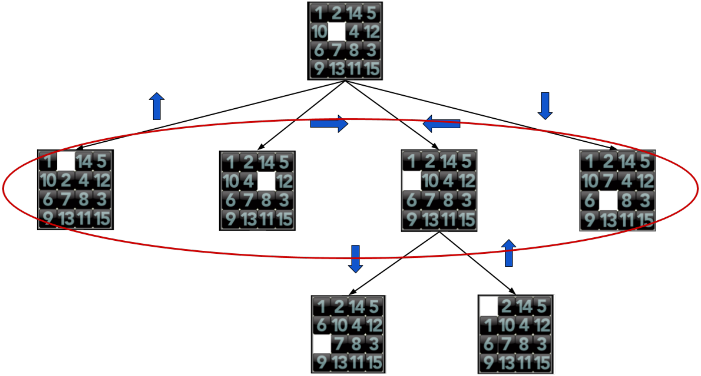
$$\sum_{i=1}^{16} m(i) + x \quad es \ par$$

Donde:

- $pos(i)$: Posición de la ficha i, definiendo $pos(16)$ a casilla vacia.
- $m(i)$: Número de fichas $j < i$ tales que $pos(j) > pos(i)$.
- x: Sea $x = 1$ si la casilla vacia se encuentra en alguna casilla negra, caso contrario $x = 0$.



El problema original de Loyd, no es alcanzable.



- Recordando que los hijos de cada nodo serán los movimientos de la casilla vacia.
- Sera posible podar los nodos que retornen a la condición anterior, además debe cumplir con las reglas del juego (bordes).

Descripción de árbol: profundidad

- Se sigue el análisis tradicional en base a los movimientos realizados
- Similar al backtraking, obtendrá como ultimo nodo a una posible solución.

FACULTAD DE CIENCIAS

- Buscamos obtener el ordenamiento final, donde la casilla $[i, j]$ se encuentran enumeradas por $(i - 1) * n + j$ y en la casilla $[n, n]$ el valor vacio.
- El valor vacio sera el que realiza los movimientos de forma $(\uparrow, \rightarrow, \leftarrow, \downarrow)$, al realizar el movimiento su casilla es reemplazada por el valor reemplazado.
- Conociendo la complejidad tan alta de problema, emplearemos una estrategia LC, conocida como estrategia del tipo no ciega. para ellos definiremos una **función de coste** para la selección de cada nodo del árbol.

Función de estimación

$$\hat{c}(x) = f(x) + \hat{g}(x)$$

Donde:
$f(x)$= longitud del camino de la raiz (configuración inicial) a x.
$\hat{g}(x)$= número de casillas no vacias que no están en su sitio objetivo (configuración objetivo).

La estrategia empleara la siguiente idea:

- Se crea una cola de nodos vivos x con sus $\hat{c}(x)$.
- Se calcula $c(1)$: nodo inicial.
- Se genera sus hijos se añaden a la cola.
- Se elige al minimo, se elimina de la cola y se generan sus hijos. Repitiendo hasta que se indique.

FACULTAD DE CIENCIAS

```
algoritmo mínimoCoste(ent x0: nodo)
variables c:cola; {cola con prioridades <x,coste(x)>}
          éxito: bool; xcurso,x:nodo
principio
  si esSolución(x0) entonces escribir(x0)
  sino
    crearColaVacía(c); {cola de nodos vivos}
    añadir(c,<x0,coste(x0)>); éxito:=falso;
    mientrasQue not éxito and not esVacía(c) hacer
      xcurso:=min(c); {nodo en expansión}
      eliminar(c,xcurso);
      mientrasQue not éxito and hay_otro_hijo_x_de_xcurso hacer
        si esSolución(x) entonces escribir(x); éxito:=verdadero
        sino añadir(c,<x,coste(x)>) fsi
      fmientrasQue
    fmientrasQue
    si not éxito entonces escribir("No hay solución") fsi
  fsi
fin
```

- La complejidad se vendrá dada por $O(f(n))$
- Lamentablemente, la complejidad del tiempo en el peor de los casos sigue siendo exponencial.
- Ramifica por cada iteración, búsqueda de opciones disponibles por iteración.
- Sea b el factor de ramificación para n=16 (15-puzzle), entonces $O(b^n)$