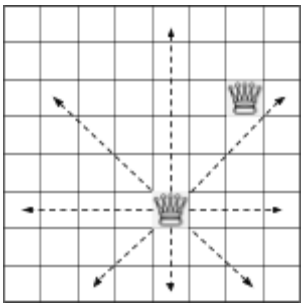
TEORÍA DE ALGORITMOS 1
Backtracking y branch and bound

Por: ING. VÍCTOR DANIEL PODBEREZSKI
vpodberezski@fi.uba.ar

1. Backtracking

Hasta el momento vimos una forma de resolver por fuerza bruta ciertos problemas. En problemas combinatorios buscamos una función de generación del espacio de soluciones. Probamos cada una de ellas para determinar si satisfacen las restricciones del problema y, en caso de un problema de optimización, nos quedamos con las soluciones que superan al resto. Luego presentamos una forma de resolver problemas expresando su espacio de soluciones como nodos de un grafo. Llamamos a este grafo de espacio de estados. Las posibles transiciones entre estados corresponden a ejes en el grafo. Una exploración exhaustiva de este espacio - siempre que el espacio de soluciones sea finito - nos asegura encontrar la solución del problema siempre que exista. Esta metodología la podemos utilizar tanto en problemas combinatorios como de exploración. Nos concentramos a continuación en los primeros.

Utilizaremos para ejemplificar el conocido problema de las 8 reinas. En este problema tenemos un tablero de ajedrez compuesto por una cuadrícula de 8 filas por 8 columnas. La pieza conocida como "reina" ubicada en un celda del tablero ataca a toda pieza que se encuentra en su misma fila, columna o diagonal. El desafío consiste en ubicar 8 reinas en el tablero de forma tal que ninguna de ellas ataque a otra.



En un problema combinatorio, en el que tenemos "n" elementos, podemos conformar diferentes posibles soluciones. En muchos de los casos, cada una de estas es posible expresarla como una tupla de como mucho t≤n elementos (x_1,x_2, ...x_{t-1}, x_t). Cada elemento x_i es seleccionado entre un conjunto finito de posibilidades. Existen un subconjunto de posibles soluciones que comienzan con los mismos "t-1" elementos iniciales. A su vez estos

forman parte de un conjunto de soluciones que inician con “t-2” mismo elementos. Esto nos permite establecer una jerarquía en el espacio de soluciones.

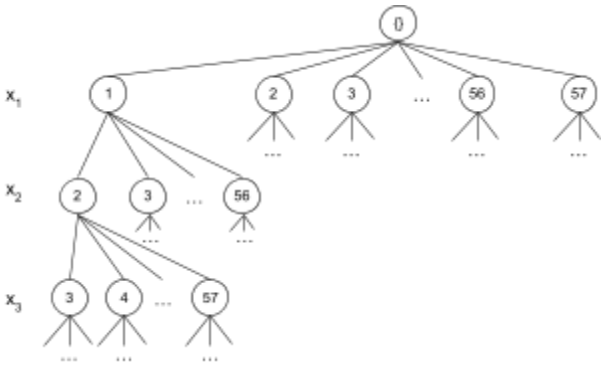
En el problema de la 8 reinas podemos expresar la solución como una tupla de 8 elementos donde cada elemento corresponde a una celda en la que está ubicada una reina en el tablero. Lo expresamos ordenados por número de celda de menor a mayor. Por ejemplo la tupla (2,12,24,27,37,47,49,62) corresponde a la posible solución mostrada en la imágen. Es fácil ver que esta no corresponde a una solución factible. Las reinas ubicadas en las celdas 2 y 47 se atacan entre sí. Podemos observar que la 7-tupla (2,12,24,27,37,47,49) puede ser el prefijo de otras 14 soluciones posibles. Planteada de la forma actual corresponde a buscar combinaciones de 64 elementos tomados de a 8. Que lo expresamos como $\binom{64}{8}$ *y totaliza 4.426.165.368 posibles soluciones a evaluar.*

Una solución alternativa y más eficiente considera que en una solución válida no puede existir más de una reina en la misma fila o columna. Por lo tanto una solución construida puede comenzar ubicando una reina en la primera fila en alguna de las 8 columnas. Luego una reina en la segunda fila en alguna de las 7 columnas no ocupadas por la reina de la primera fila. Se continúa por el resto de las filas hasta finalizar. La solución también es una 8-tupla, con valores diferentes entre sí entre 1 a 8. Si en la posición i se encuentra el valor xi implica que la reina se ubica en la fila i en la columna xi. En este caso se buscan como soluciones las permutaciones de 8 elementos. Lo expresamos como 8! y totaliza 40.320 posibles soluciones a evaluar Un número marcadamente menor a los de la propuesta de solución antes planteada. Continuaremos ejemplificando con el primer planteamiento. En otra sección retomaremos con más detalle un ejemplo de generación de permutaciones.

Gracias a esta jerarquía podemos expresar el grafo de estados del problema como un **árbol de estados**. La raíz representa habitualmente el estado en el que aún no se ha realizado elección de elemento alguno. Cada nodo corresponde a la adición de un elemento que se suma a los elementos de sus antecesores. Los nodos pueden representar tanto soluciones parciales como completas. Llamaremos **estados del problema** a todos los nodos del árbol. Dentro de ellos un subconjunto corresponde a los **estados solución.** En estos la tupla conformada por los elementos en su camino desde la raíz definen una

tupla en el espacio de soluciones. Son los estados cuyos elementos cumplen con las restricciones explícitas del problema. Finalmente, un subconjunto de estos corresponden a los **estados respuesta**. Corresponden a una solución válida al problema (cumple con sus restricciones implícitas). El árbol se ramifica hasta que no queden posibles elecciones por realizar.

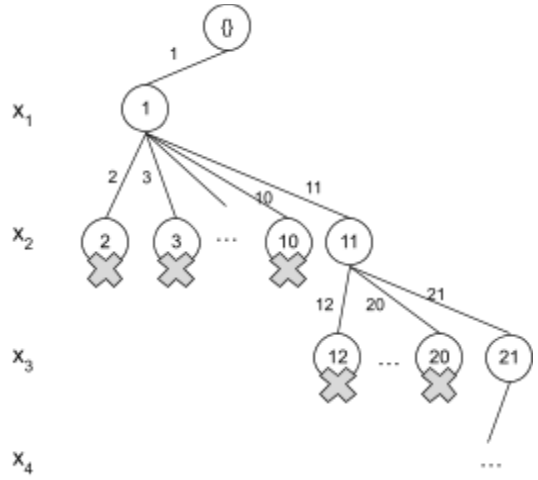
El problema de las 8 reinas planteado como la búsqueda de combinaciones se puede representar como árbol de estados con una raíz que representa el tablero vacío. Desde allí se pueden generar 57 nodos descendientes. Cada uno de ellos corresponde a ubicar la primera reina entre la primera y la celda 57 (emplazar en una celda posterior a esta no es posible dado que no entrarían las 7 reinas en las celdas restantes). Cada uno de estos nodos representan una tupla de un elemento (x_1) y son un estado del problema. Cada uno de estos nodos tendrán (64-x_1-7) descendientes con el emplazamiento de una segunda reina, estos representarán al estado del problema con una tupla de dos elementos (x_1,x_2). El 7 corresponde a la cantidad de reinas pendientes a ubicar (esta fórmula permite luego generalizar la generación de descendientes para niveles posteriores). La ramificación continuará hasta llegar al nivel 8 de profundidad del árbol correspondiente a las hojas del mismo. Estas representan tuplas de 8 elementos y son los estados solución. Algunos de ellos, que cumplan con las restricciones implícitas (no hayan reinas que se ataquen entre sí) serán los estados respuesta.



Ya definida la estructura del árbol de estados se debe determinar una manera de generarlo y recorrerlo. Utilizaremos la técnica de búsqueda por profundidad (Depth-First Search). Este algoritmo nos permite recorrer todo el árbol y analizar las posibles soluciones. La forma de recorrerlo, donde se adentra lo más que puede en la profundidad del árbol y retrocede cuando no quedan caminos por recorrer en la rama actual es lo que da nombre al método general: **Backtracking**. Se reconoce a D. H. Lehmer en la década del 50 como la persona que acuñó esta denominación. A R. J. Walker en los 60 por darle forma algorítmica. A S. Golomb y L. Baumert por demostrar su aplicabilidad en una gran variedad de situaciones.

Llegados a este punto es lícita la pregunta, si el agregado del árbol jerárquico no convierte a este método en menos eficiente que los métodos de generar y probar. En una inspección se puede observar que los estados del problema son en el mejor de los casos igual o mayor que la cantidad total de posibles soluciones del problema. Sin embargo existe una cuestión fundamental hasta ahora pasada por alto. Es posible en muchos casos simplemente evaluando un estado del árbol determinar que cualquier estado solución derivada de este sea incapaz de cumplir con las restricciones implícitas del problema. Llamaremos a esta **propiedad de corte**. En ese caso, podemos desistir de continuar explorando los descendientes de este nodo y regresar al estado anterior. Hablamos en este caso de una poda del árbol de estados. La aplicación de esta técnica en algunos problemas puede reducir considerablemente el espacio de soluciones a evaluar. Llamaremos **función límite** a la que analice la propiedad de corte.

Consideremos el recorrido del árbol de estados en el problema de las 8 reinas. Iniciamos en la raíz y buscamos el primer lugar donde ubicar la primera reina. Seleccionamos la posición 1. Ingresamos al nodos que representa el estado del problema mediante la tupla (1). Este estado es válido. Buscamos desde aquí el siguiente nodo y corresponde a ubicar la segunda reina en la casilla 2. En el nuevo nodo tenemos la tupla (1,2). Analizar la propiedad de corte en este caso corresponde a determinar si la reina agregada ataca a alguna de las previamente ubicadas.



Efectivamente se atacan entre sí (están en la misma fila). Por lo tanto se abandona la exploración de todos sus nodos descendientes, con la consecuente poda del árbol. Se regresa al nodo del estado (1). Se selecciona el siguiente estado posible a evaluar, la tupla (1,3). Que nuevamente al analizar la propiedad de corte nos indica que la reina en la posición 3 ataca a la que está en la 1. Este proceso se repite al intentar ubicar la reina en las posiciones de 4 a 10. Donde sí podremos ubicar la segunda reina es en la celda 11. Se continuará explorando desde allí para intentar ubicar la tercera reina. Se repetirá el proceso hasta encontrar una solución resultado o hasta terminar de recorrer todo el árbol y regresar a la raíz sin posibilidad de nuevas exploraciones. En la imagen se muestran las primeras exploraciones realizadas por el algoritmo

de backtracking para resolver este problema. Acompaña a cada eje el número que representa el orden en el que el nodo es visitado por primera vez. Es fácilmente observable la gran cantidad de poda realizada hasta el momento en el árbol.

A continuación se presenta un esquema general del funcionamiento de la metodología de backtracking mediante pseudocódigo. En este caso se busca hasta encontrar una solución válida.

Algoritmo genérico backtracking (búsqueda de una solución)

```
Backtrack (estadoActual):
        Si estadoActual es un estado resultado
                retornar estadoActual

        Si estadoActual supera la propiedad de corte
                Por cada posible estadoSucesor de estadoActual
                        resultado = Backtrack(estadoSucesor)

                        si resultado es un estado resultado
                                retornar resultado

        retornar vacio

Sea estadoInicial la raiz del arbol de estados
Backtrack(estadoInicial)
```

En las próximas secciones analizaremos diferentes problemas y los resolveremos mediante backtracking. El planteo, para mayor claridad se realiza de forma recursiva. Sin embargo, vale aclarar, que muchos de ellos se pueden expresar de forma iterativa utilizando estructuras de datos y una forma de exploración adecuadas.

2. Backtracking: Problema de la mochila

Recordemos el problema de la mochila. Su enunciado es el siguiente:

Problema de la Mochila

Contamos con una mochila con una capacidad de K kilos y queremos introducir dentro de ellas un subconjunto del conjunto E de “n” elementos con el objetivo de maximizar la

ganancia. Cada elemento i tiene un peso de k_i kilos y un valor de v_i. Los elementos no pueden ser divididos y el peso total seleccionado no puede superar la capacidad de la mochila.

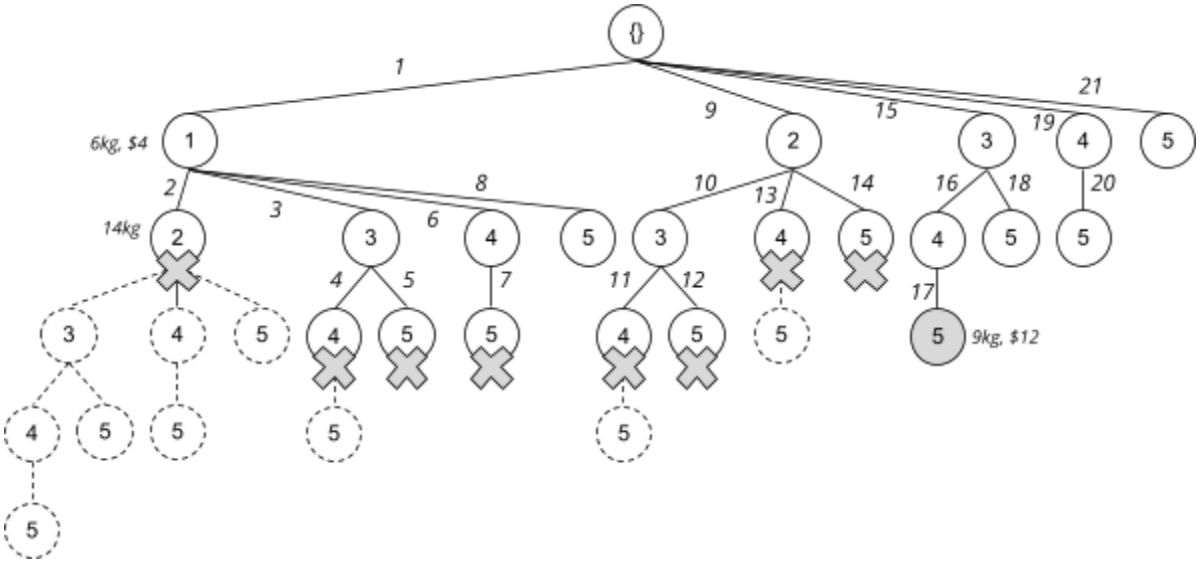
Corresponde a un problema de optimización cuyo espacio de soluciones son todas las combinaciones posibles de elementos que se pueden incluir en la mochila. Al resolverlo mediante generar y probar representamos ese espacio de soluciones como n-tuplas. Cada posición de la tupla representaba la inclusión o no de cierto elemento en la solución. Las restricciones explícitas corresponden a las combinaciones posibles de elementos. Las implícitas a evitar que lo incluido en la mochila no supere su capacidad. El criterio de optimización es la maximización de la suma de los valores de los elementos seleccionados.

Con el objetivo de resolver este problema mediante backtracking debemos pensar una manera de representar el árbol de estados en una manera jerárquica. Consideramos que los elementos a probar están ordenados arbitrariamente y lo podemos expresar como una lista de tamaño n. Una primera aproximación podría corresponder a una elaboración de tipo árbol binario. La raíz corresponde al estado de la mochila vacía. Cada nodo en el nivel "i" tiene dos descendientes. Por un lado si el elemento en la posición i se incluye y por el otro si no lo hace. Los nodos de inclusión corresponden a estados solución.

El árbol de estados presentado se puede compactar eliminando los estados de no inclusión. La raíz sigue representando la mochila vacía. Los descendientes de cada nodo corresponden a los posibles elementos a elegir posteriores al agregado en el mismo. De esta forma se arma lo que llamamos un árbol combinatorio. Todos los estados del problema corresponden a estados solución. Cada nodo estado se corresponde a tener en la mochila los elementos incluidos en los nodos del camino desde la raíz al mismo y adicionar alguno de los elementos posteriores al introducido en el nodo padre.

Contamos con una mochila de capacidad de 10kg y debemos seleccionar entre los siguiente elementos 1:(6kg, $4), 2:(8kg, $9), 3:(2kg, $2), 4:(4kg,$5), 5:(3kg,$5). El estado de la raíz del árbol corresponde a la mochila vacía y la posibilidad de elegir entre cualquiera de los elementos a ingresar en primer lugar. El algoritmo de backtracking comienza en la raíz. Puede seleccionar ingresar cualquiera de los 5 elementos. Comienza seleccionando el nodo que representa ingresar el elemento 1. En ese estado tendremos 6kg en la mochila y una ganancia de $4. Desde

allí podemos seleccionar de los elementos 2 al 5. Se selecciona el elemento 2 (y por lo tanto se llega en el árbol al estado que tiene el elemento 1 y 2 en la mochila). Este corresponde a tener 14ks y superar la capacidad de la máxima. Se debe desechar esta posible solución y todo las derivadas de esta. Se realiza la poda y se regresa al estado anterior. El proceso continúa hasta regresar a la raíz y no tener caminos inexplorados disponibles. En la imagen se muestra el árbol al finalizar la exploración. Los nodos en línea punteada son estados que se podan. Los nodos con la "X" son aquellos donde la evaluación de función límite supera la propiedad de corte. El número en cada eje es el orden en la exploración. El nodo gris indica al estado con la solución óptima. El camino desde la raíz representa los elementos en la mochila para lograrlo: 3,4 y 5.



El siguiente pseudocódigo sumariza el algoritmo de backtracking para resolver el problema de la mochila:

Problema de la mochila: Solución mediante backtracking

```
Sea elementos un vector de los elementos disponibles
Sea mochila una lista inicialmente vacía con los elementos seleccionados.
Sea maximaGanancia la cantidad máxima obtenida en la mochila
Sea maximaCombinacion los elementos seleccionados para obtener la máxima ganancia

Backtrack (mochila):
        Si mochila no supera la capacidad disponible
                Sea ganancia la suma de los elementos en la mochila
                si ganancia > maximaGanancia
                        maximaGanancia = ganancia
```

```
                    maximaCombinacion = mochila

        Si mochila tiene capacidad disponible
                Sea ultimoElemento el ultimo elemento añadido en la mochila
                Por cada elemento posterior a ultimoElemento en elementos
                        Agregar elemento a mochila
                        Backtrack(mochila)

                        Quitar elemento de mochila

mochila={}.
maximaGanancia=0
maximaCombinacion=mochila

Backtrack(mochila)
```

La complejidad de este algoritmo está dada por recorrer en el peor de los casos todo el espacio del problema definido por el árbol estados. Corresponde este proceso a La complejidad de este algoritmo está dada por el iterador de soluciones que se ejecutará $O(2^n)$. A esto se le debe sumar el trabajo a desarrollar en cada nodo. La mayoría de los procesos se pueden realizar aprovechando los cálculos anteriores: calcular la ganancia y el peso acumulado. El único proceso O(n) corresponde a intercambiar una nueva solución óptima encontrada por la previa.

3. Backtracking: Problema del viajante de comercio

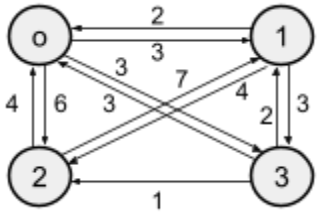
Regresamos al problema del viajante de comercio.

Problema del viajante de comercio
Contamos con un conjunto de “n” ciudades que visitar. Existen caminos que unen pares de ciudades. Cada camino inicia en una ciudad “x” y finaliza en la ciudad “y” tiene asociado un costo de tránsito de $w_{x,y}$. Partiendo desde una ciudad inicial y finalizando en la misma se quiere construir un circuito que visite cada ciudad una y solo una vez minimizando el costo total.

Al resolverlo por fuerza bruta se procedió a generar todas las permutaciones posibles. Sin embargo en muchas ocasiones, esto nos llevaba a evaluar en el peor de los casos una gran

cantidad de caminos inviables por estar construidos mediante tramos que no existían en la instancia del mismo.

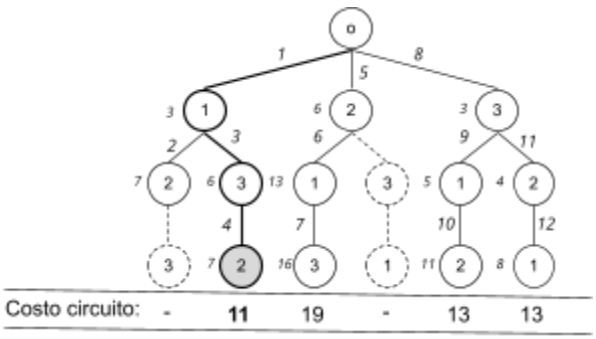
Usaremos un ejemplo sencillo para el resto de la explicación



Sea un viajante de comercio que se encuentra en la ciudad "o" y debe visitar las ciudades 1,2 y 3. Algunos de los caminos no están disponibles para unir entre ciudades. Conocemos los costos de traslado entre cada una de las ciudades conectadas. Se muestran en el grafo de la figura. Debe determinar el circuito a realizar para visitar una y solo una vez cada una regresando de donde partió.

Armaremos un árbol de estados que nos permita recorrer de una manera eficiente todos los circuitos posibles. En este caso la raíz representa el comienzo del viaje partiendo de la ciudad inicial. El camino desde la raíz hasta el nodo representa el recorrido realizado desde la ciudad inicial hasta el momento. Cada nodo representa la inclusión de una ciudad en el recorrido que no fue previamente visitada. Los estados descendientes del mismo son las posibles elecciones de próximas ciudades restringidas por las previamente visitadas. Al elegir una opción se debe sumar el costo del traslado. Las raíces del árbol serán los estados solución. Se cuenta con n! de estos estados correspondientes a las permutaciones posibles. Las hojas estarán a nivel n del árbol. En este caso la "poda" la podemos realizar al momento de generar los nodos descendientes. Evitamos crear un nodo si no existe un camino entre la ciudad en la que nos encontramos y una posible siguiente que aún no se ha visitado. Esto lo realizamos consultando si existe o no un camino que los una.

El viajante de comercio desde la ciudad "o" puede seleccionar cualquiera de las ciudades para iniciar el recorrido. Inicialmente selecciona la ciudad 1. Se crea el estado correspondiente al camino o,1. El mismo tiene un costo de 3. Desde la ciudad 1 puede seleccionar ir a la ciudad 2 o 3. Selecciona la ciudad 2 aumentando el costo del camino en 7 unidades con un total de 10. Podría, ahora, intentar ir a la ciudad 3. Pero no hay un camino disponible. En ese caso se recorta ese camino y todas sus posibles derivaciones. Lo único que le queda es ir a la ciudad "o" y cerrar el circuito. Sin embargo la



longitud del camino advierte que no se recorrieron todas las ciudades. Siendo que no puede seguir, realiza un backtrack y procede a continuar con la búsqueda en la ciudad 1. Allí selecciona ir a la ciudad 3 y aumentó el costo a 6. Desde la ciudad 3, de las ciudades no visitadas resta la 2. El camino que la une tiene un costo de 1. Ahora sí, desde la ciudad 2 puedo ir a la ciudad inicial con un costo adicional de 4. Se ha encontrado un circuito con costo total de 11. Esta puede no ser la menor disponible. Por lo que se debe, realizando backtracking, seguir recorriendo el árbol hasta agotar todos los caminos posibles. El árbol que se muestra presenta la exploración final de esta instancia del problema.

La siguiente sintetiza en pseudocódigo el funcionamiento de esta propuesta:

Problema del viajante: Solución mediante backtracking

```
Sea C[x,y] el costo de ir de la ciudad x a la y
Sea A[y] la lista de ciudades adyacentes de la ciudad y.
Sea camino el recorrido realizada hasta el momento
Sea minimoCosto el costo del camino minimo encontrado
Sea minimoCamino el circuito de menor costo encontrado

Backtrack (camino):
        Sea x la ultima ciudad visitada.
        Si longitud del camino es n
                Si la ciudad o se encuentra en A[x]
                        Agregar al final de camino la ciudad o
                        Sea costoCamino la suma de los costos de camino
                        Si costoCamino < minimoCosto
                                minimoCosto = costoCamino
                                minimoCamino=camino
                        Remover o de camino
        Sino
                Por cada ciudad y en A[x] no visitada previamente excepto “o”
                        Agregar y a camino
                        Backtrack(camino)
                        Quitar  y de camino

camino={o}.
minimoCosto=infinito
minimoCamino={}

Backtrack(camino)
```

La complejidad de este algoritmo está dada por recorrer en el peor de los casos todo el espacio del problema definido por el árbol estados. Se puede ver que este trabajo en el peor de los casos se ejecuta más de n!. Por lo tanto resulta posiblemente peor que el de generar y probar si todas las ciudades están unidas con todas (no existe una poda posible). El proceso de calcular el costo acumulado hasta el momento en cada nodo se puede hacer en O(1) utilizando el costo del padre. Las complejidades de iterar lista de ciudades adyacentes si se mantiene, determinar si una ciudad ya fue visitada y copiar en caso de obtener una solución óptima para resguardarlo la podemos hacer en el peor de los casos en O(n). Siendo posible cada una de estas consultas realizarlas en O(1) con las estructuras adecuadas y utilizando mayor almacenamiento.

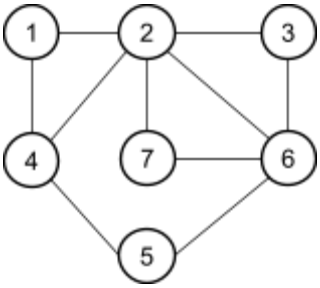
4. Backtracking: K coloreo de grafos

El problema de coloreo de grafos tuvo como origen un problema relacionado: El coloreo de mapas. La necesidad de trazar mapas políticos, donde diversas entidades se coloreaban con diferentes colores nos lleva al siglo XIX. En una reunión de la Sociedad Matemática de Londres en 1879 se establece para la comunidad el problema. Los mapas se podían representar como grafos donde los nodos eran las diferentes zonas y los ejes establecen la existencia de límites entre estas. Dados los postulados establecidos, los grafos estudiados eran particulares: planares. Los grafos planares son aquellos que se pueden dibujar en el plano sin que sus ejes se intersecten entre sí. El problema más general, en el que se puede analizar cualquier grafo es con el que trabajaremos a continuación. Su enunciado es el siguiente:

K Coloreo de grafos
Contamos con un grafo G=(V,E) con “n” vértices y “m” ejes. Queremos asignarles no más de K colores a sus vértices de modo que para cualquier par de vértices adyacentes no compartan el mismo color.

Podemos resolver este problema mediante generar y probar realizando una variación de las n-tuplas. En vez de representar las posibles soluciones con un vector binario de n posiciones, será un vector “k-ario”. Se tendrán en total k^n posibles coloraciones. Entre ellas

muchas inválidas (por no cumplir la condición de adyacencia) y otras tantas que sí. Se conoce como número cromático de un grafo a la cantidad total de formas diferentes de colorear un determinado grafo. Para el análisis que realizamos nos alcanza con hallar una coloración determinada.



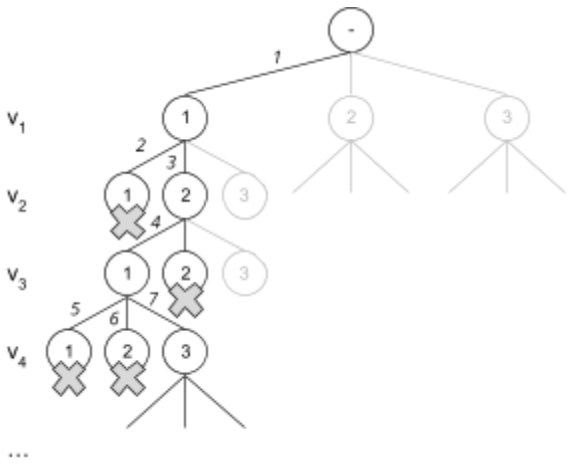
Queremos saber si podemos colorear los vértices del grafo de la imagen con 3 colores. Este contiene 7 vértices y 10 ejes. Solucionarlo mediante generar y probar requiere 7^{10} soluciones posibles. Esto corresponde a un total de 282.475.249. Una cantidad considerable. Ninguna solución válida podrá tener por ejemplo los vértices 7 y 6 con el mismo color. De igual manera los vértices 2, 3 y 6 deben tener colores diferentes entre sí.

Para resolver el problema mediante backtracking primero determinaremos cómo representar el árbol de estados. En primer lugar estableceremos un orden de 1 a "n" para los vértices del grafo. Llamaremos v_i al vértice en la posición i. Propondremos un árbol k-ario de profundidad máxima "n". El nodo raíz corresponde al grafo sin ningún color en sus ejes. Desde allí se desprenden k nodos hijos. Cada uno de estos corresponde a asignarle un color diferente al vértice v_1. Desde cada uno de ellos se desprenden nuevamente k nodos representando los colores asignados al vértice v_2. Así continúa hasta llegar a la profundidad "n". Estos nodos corresponden al espacio de solución y pueden existir un máximo de k^n. El camino desde la raíz a estos determina la asignación de un color para cada vértice. Se puede verificar si el coloreo es válido si por cada vértice no comparte el mismo color con sus adyacentes. En caso de encontrar una asignación válida se retorna y finaliza la exploración.

Es posible aplicar un mecanismo de poda en cada estado del problema. Al asignar un color a un nodo determinado se puede observar si alguno de sus adyacentes tienen el mismo. En ese caso, cualquier asignación posterior será inválida como solución. Se abandona la exploración en los descendientes del mismo y se realiza backtrack para regresar al nodo padre. Desde allí continúa la exploración.

En el grafo del ejemplo nos interesa encontrar un posible coloreo. Iniciamos asignando el color "1" al vértice "1". Desde allí podemos asignar 3 colores al vértice 2. Comenzamos coloreando con el color "1" y al verificar la validez notamos que los vértices 1 y 2 son adyacentes y no pueden

tomar el mismo color. No seguimos analizando asignaciones desde allí. Realizamos backtrack y ahora coloreamos el vértices 2 con el color "2". Como es una coloración válida, continuamos con el vértice 3. En ese caso seleccionamos el color "1" que resulta en un coloreo válido. De esa forma continuamos hasta llegar a una coloración válida de 3 colores para los 7 vértices o informamos que no existe.



El siguiente pseudocódigo corresponde a una posible solución siguiendo la idea planteada:

K coloreo de grafos: Solución mediante backtracking

```
Sea C[x] el color asignado al vértice x
Sea A[x] la lista de vértices adyacentes del vértice x.

Backtrack (nroVertice):

        Por cada color disponible
                C[nroVertice]=color

                Sea coloreoValido = true
                Por cada vertice x en A[nroVertice]
                        Si C[nroVertice] == C[x]
                                coloreoValido =false

                Si (coloreoValido y nroVertice==n)
                        retornar true

                Si (coloreoValido y nroVertice<n)
                        Si Backtrack(nroVertice+1)
                                retornar true
                Quitar color a C[nroVertice]
        Retornar false

vertice=1
Si Backtrack(vertice)
```

```
        Imprimir C[x] para todo vertice x del grafo
Sino
        Imprimir 'no hay un coloreo posible'
```

La complejidad de este algoritmo está dada por recorrer en el peor de los casos todas los estados del árbol de estados: $O(k^n)$. Además el trabajo realizado en cada estado está definido por la iteración por cada uno de los colores posibles para colorear el vértice actual O(k) y por cada uno de estas coloraciones verificar si no hay otro vértice adyacente con ese mismo color en O(n). Sumando todos los procesos tenemos una complejidad temporal de $O(nk^{n+1})$.

Con pocas modificaciones en la lógica se puede lograr conseguir todas las coloraciones posibles (y por lo tanto obtener el número cromático de la instancia).

5. Branch and Bound

El método conocido como **ramificación y poda** (Branch and Bound) se lo suele definir como una variante de Backtracking más sofisticada para problemas de optimización. Al igual que este último en última instancia recorre todo el espacio de estados del problema. Sin embargo presenta diferencias a la hora de definir la forma de exploración del árbol de estados y a la metodología de poda del mismo.

Para realizar la poda, se mantienen los mecanismos incluidos por Backtracking: Desestimar la exploración de ramas al momento de determinar que no cumplen la propiedad de corte. Además se incorpora una nueva evaluación relacionada con la imposibilidad de mejorar la mejor solución encontrada hasta el momento continuando la exploración por los descendientes de la misma. Para eso es indispensable contar inicialmente con un valor inicial de una posible solución factible que se actualiza cada vez que se encuentra una mejor. Poder estimar para cada nodo del árbol el valor límite que se podría conseguir recorriendo cada uno de sus descendientes. Llamaremos a esta **función costo**. Al evaluar un nodo se compara el valor de su función costo con el valor de la mejor solución factible encontrada. En caso de que no logre superarla, se realiza la poda. La determinación de la función costo es intrínseca al problema y es una cuestión a no desdeñar. Seleccionar una equivocada puede llevar a podar equivocadamente porciones del árbol que contenga la

solución óptima. O en sentido inverso, puede obviar la poda de ramas cuya exploración no brinden mejora a lo ya encontrado.

En cuanto a la manera de recorrer el árbol se busca privilegiar la exploración de las ramas más promisorias. Para lograrlo, existen diferentes propuestas. La manera tradicional es conocida como **depth-first branch-and-bound** (DFBB)[1] [2]. En esta partimos de un nodo y expandimos todos los nodos descendientes. Se evalúan, podando según el criterio de poda y entre los restantes seleccionando el más promisorio aún no explorado. Esta elección corresponde al uso de la función costo que se debe determinar según el problema a resolver. El nodo seleccionado pasa a ser el nodo activo. Si el nodo no cuenta con descendientes por recorrer (sea por que ya se agotaron en exploraciones previas, se podaron o simplemente no tiene) se realiza el proceso de Backtrack y se regresa a su nodo padre. El procedimiento se repite hasta finalizar el recorrido de todos los caminos posibles y encontrar la solución óptima.

A continuación se muestra un esquema general de Branch and bound utilizando DFBB:

Algoritmo genérico Branch and Bound (DFBB)

```
Backtrack (estadoActual):
        Sea descendientes los nodos descendientes de estadoActual
        Por cada posible estadoDescendiente de estadoActual
                Si estadoDescendiente supera la propiedad de corte
                        Calcular fc funcion costo de estadoDescendiente
                        Agregar estadoDescendiente a descendientes con fc

        Mientras existan estados descendientes no explorados
                Sea estadoProximo el estado en descendientes aún no analizado de mayor fc

                Si el fc de estadoProximo es mayor a la mejor solución obtenida
                        Si estadoProximo es un estado solución y es superior a la mejor
                                mejor = estadoProximo

                        Backtrack(estadoProximo)
```

[1] Lawler, E. L. and Woods, D. 1966. Branch-and-bound methods: A survey. Operations Research 14.
[2] Kumar, Vipin 1987. Branch-and-bound search. In Shapiro, Stuart C., editor 1987, Encyclopaedia of Artificial Intelligence: Vol2. John Wiley and Sons, Inc., New York. 1000-1004.

```
Sea estadoInicial la raiz del arbol de estados
Backtrack(estadoInicial)
```

Una manera alternativa de recorrer el árbol es la conocida como **best-first search** (BeFS)[3]. En este caso el procedimiento de expansión de nodos es similar al anterior. Lo que se modifica es la elección del próximo a explorar. Deja de ser únicamente entre los descendientes del nodo actual, sino entre todos los nodos expandidos aún no explorados. Corresponde a una elección global en contraposición de una local. Como ventaja frente a la anterior se puede establecer que los nodos más promisorios son evaluados con prioridad. La desventaja es la necesidad de generar en una estructura jerárquica todos los nodos expandidos. Lo que equivale a un uso más intensivo de la memoria. La estructura a utilizar puede ser una cola de prioridad por ejemplo.
A continuación se muestra un esquema general de Branch and bound utilizando BeFS:

Algoritmo genérico branch and bound (BeFS)

```
Sea estadosDisponibles los estados en el árbol expandidos por analizar

Sea estadoInicial la raiz del arbol de estados
Calcular fc funcion costo de estadoInicial
Agregar estadoInicial con fc a estadosDisponibles

Mientras queden estados estos en estadosDisponibles
        Sea estadoActual el estado de mayor fc en estadosDisponibles

        Si estadoActual supera la propiedad de corte
                Si fc de estadoActual es mayor a la mejor solución obtenida
                        Si estadoActual es un estado solución
                                mejor = estadoProximo

                        Por cada posible estadoSucesor de estadoActual
                                Calcular fc funcion costo de estadoInicial
                                Agregar estadoSucesor con fc a estadosDisponibles
```

[3] Pearl, J. Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley, 1984. p. 48.

Una análisis más detallado de estos métodos y otros puede leerse en Depthfirst vs best-first search [4] entre otras publicaciones [5].

A continuación desarrollaremos algunos ejemplos de la aplicación de branch and bound.

6. Branch and Bound: Problema de la mochila

Regresamos al problema de la mochila que ya analizamos mediante backtracking. Vamos a, en este caso para resolverlo, modificar la forma que representamos el árbol de estados. Para cada elemento "i" calculamos su valor por unidad u_i: el cociente entre su valor v_i y su peso k_i. Al comparar dos elementos por su valor por unidad observamos que a mayor proporción, significa que una misma cantidad de peso ofrece más ganancia. Por lo tanto, el orden de evaluación de los elementos en el árbol estará dado por este valor de forma decreciente.

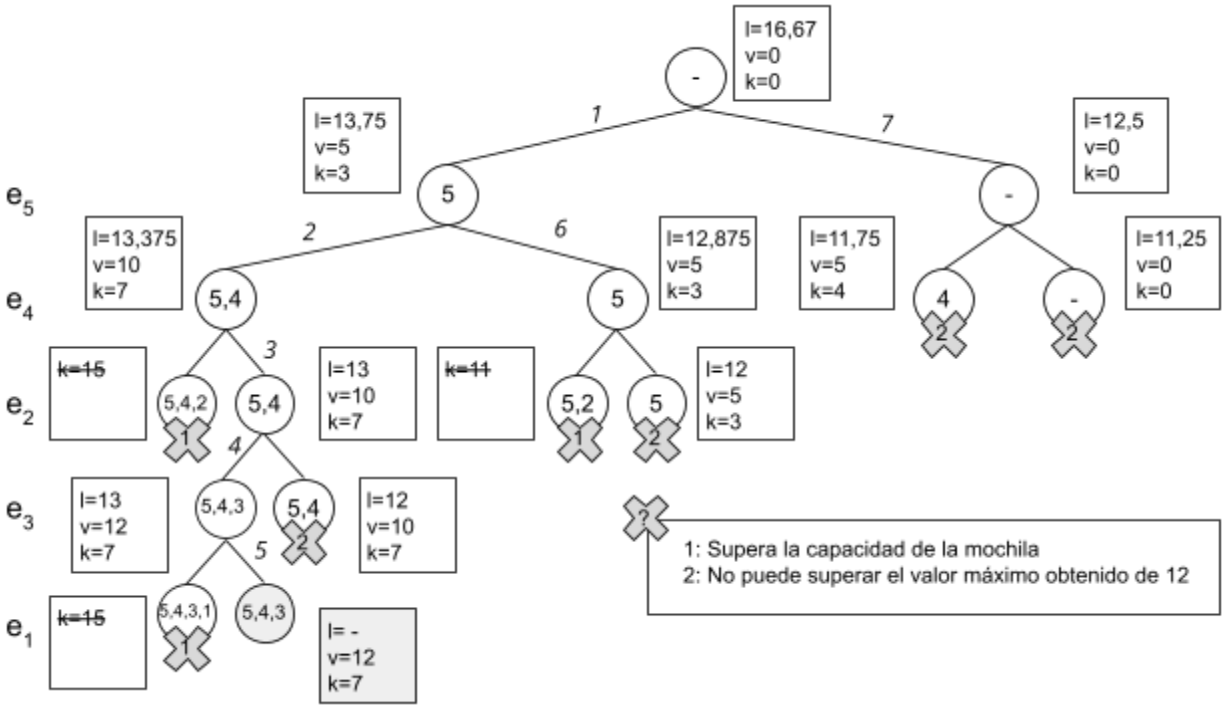
Al analizar el problema mediante backtracking se brindaron dos modelos de árbol. El primero, se desestimó por una alternativa más compacta. En este caso, volveremos al desechado: Un árbol binario donde cada nivel corresponde a determinar si el elemento i se incluye o no dentro de la mochila. La raíz corresponde a la mochila vacía. Cada nodo en el nivel "i" tiene dos descendientes. Si tenemos n elementos la profundidad máxima corresponderá a n. En cada nodo tendremos como información el peso cargado en la mochila para ese estado al que llamaremos w y el valor v de ganancia obtenido.

Utilizaremos la función para el costo una función que llamaremos "l". Dado un nodo a una profundidad del árbol i, veremos cual es el elemento aun no evaluado de mayor valor por unidad. Dado el ordenamiento planteado, corresponde simplemente al elemento i+1. Obtenemos cuál es el espacio disponible en la mochila (K-k). Ahora supondremos que en el mejor de los casos todos ese espacio se llenará con el elemento i+1 (u otros posteriores con mismo valor por unidad). Sumando el valor de la ganancia obtenida con la mayor ganancia hipotética posible tenemos un tope a la ganancia de cualquier estado del árbol

[4] Vempaty, N.R., V. Kumar, and R.E. Korf, "Depthfirst vs best-first search," Proc. 9-th National Conf. on AI, AAAI-91, CA, July, 1991, pp.434-40.
[5] Weixiong Zhang and Richard E. Korf. 1993. Depth-first vs. best-first search: new results. In Proceedings of the eleventh national conference on Artificial intelligence (AAAI'93). AAAI Press, 769–775.

que descienda del nodo. Si este valor es inferior a la máxima ganancia encontrada, podemos realizar la poda. Además entre todos los nodos a expandir se seleccionará a aquel con mayor valor. La expresión de la función corresponde a $l = v + (K - k) * u_{i+1}$. Para recorrer el árbol utilizaremos depth-first branch-and-bound.

*Contamos con una mochila de capacidad de 10kg y debemos seleccionar entre los siguiente elementos 1:(6kg, $4), 2:(8kg, $9), 3:(2kg, $2), 4:(4kg,$5), 5:(3kg,$5). Calculamos el valor por unidad de cada uno de ellos: u_1=1/3, u_2=9/8, u_3=1, u_4=5/4, u_5=5/3. Ordenados de forma descendente nos quedará: (5,4,2,3,1). La raíz del árbol de estados corresponde a la mochila vacía. Además corresponde, hasta el momento, a la solución factible de mayor ganancia (cero). Podemos calcular el valor l de la raíz como 0 + 10*5/3 = 16,67 (redondeado). Este valor corresponde a la máxima ganancia teórica que podríamos tener en la mochila. Desde la raíz tenemos dos opciones: incluir o no el elemento 5. En caso de incluirlo tendremos 7kg de disponibilidad en la mochila y 5 de ganancia. El elemento posterior con mayor valor por unidad es el 4. Por lo tanto podemos calcular el l del nodo como 5 + 7*5/4 =13,75. La contraparte, no seleccionar el elemento 5, mantiene la mochila vacía y su valor l lo podemos calcular como 10*5/4 = 12,5. Elegimos el nodo de la inclusión por tener mayor valor en la función l, superar la máxima ganancia previa y no superar la capacidad de la mochila. En este nodo se evalúa incluir o no el elemento 4. Aplicando los mismos criterios. La rama se poda si no puede superar el máximo encontrado o supera el límite de la mochila. Si no hay nodos descendientes se realiza backtrack, para explorar nodos aún no analizados. Se muestra en la imagen la exploración del árbol para esta instancia. La rama a la izquierda corresponde a incluir al elemento. El número en el eje muestra el orden de exploración de los nodos. La tabla en cada nodo incluye la máxima ganancia teórica posible, el peso en la mochila acumulado y la ganancia obtenida. Se puede observar que la mejor ganancia corresponde a seleccionar a los elementos 5,4 y 3. En cada nodo podado se referencia cuál fue el motivo. Cada nodo puede podarse en el mismo momento de crearse o al regresar y establecer que no puede superar al máximo obtenido al momento.*



El siguiente corresponde al pseudocódigo del algoritmo:

Problema de la mochila: Branch and Bound (DFBB)

```
Backtrack (mochila, nro):
        Sea elemento el elemento en la posicion nro en elementos.

        Sea mochilaAmpliada = mochila ∪ elemento
        Sea descendientes los nodos descendientes de estadoActual

        Sea pesoActual el peso de los elementos en la mochila
        Sea gananciaActual la suma de los valores de los elementos en mochilaAmpliada

        Sea pesoAmpliado el peso de los elementos en la mochilaAmpiada
        Sea gananciaAmpliada la suma de los valores de los elementos en mochilaAmpliada
        Si nro==n
                Si pesoActual <= K y gananciaActual > mejorGanancia
                        mejorMochila = mochila
                        mejorGanancia = gananciaActual
                Si pesoAmpliado <= K y gananciaAmpliada > mejorGanancia
                        mejorMochila = mochilaAmpliada
                        mejorGanancia = gananciaAmpliada
        sino
```

```
        Sea elementoSig el elemento en la posicion nro+1 en elementos.
        Sea valorUnSig el valor por unidad de elementoSig

        Sea CotaActual = gananciaActual + (K -  pesoActual ) * valorUnSig
        Sea CotaAmpliada = gananciaAmpliada + (K -  pesoAmpliado ) * valorUnSig
        Sea opciones los estados descendientes posibles.

        Si  pesoActual <= K y CotaActual > mejorGanancia
                Agregar mochila a descendientes con CotaActual

        Si  pesoActual <= K y CotaActual > mejorGanancia
                Agregar mochilaAmpliada a descendientes con CotaAmpliada

        Mientras existan mochilas en descendientes no explorados
                Sea mochilaDesc en descendientes aún no analizada con mayor cota

                Si cota > mejorGanancia
                        Backtrack(mochilaDesc,nro+1)

Sea mochila la mochila inicialmente vacia
Sea K la capacidad de la mochila
Sea elementos el listado de n elementos ordenados segun su valor por unidad
Sea nroelemento la posicion en el listado de elemento a evaluar
Sea mejorMochila la combinacion de la mochila que da mayor ganancia encontrada
Sea mejorGanancia la combinacion de la mochila que da mayor ganancia encontrada

nroelemento =1
mejorResultado = {}
mejorGanancia = 0
Backtrack(mochila,nroelemento)
```

La complejidad fundamental de este algoritmo está dada por la necesidad en el peor de los casos de explorar todos los estados del problema. Al ser un árbol binario de profundidad n, tendremos un total de 2^n estados. El trabajo dentro de cada problema se podría acotar a O(n). Lo que nos lleva a una complejidad final de $O(n2^n)$. Hay formas alternativas de elaborar los subproblemas. El seleccionado y desarrollado si bien no es el más eficiente, presenta una lógica clara. También se puede resolver de forma iterativa utilizando una estructura de pila para los estados a explorar. En ese caso se debe almacenar por cada estado una copia de los elementos en la mochila y el próximo elemento a evaluar.

7. Branch and Bound: Problema del viajante de comercio

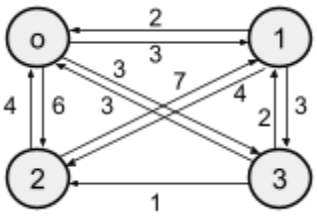
Regresamos al problema del viajante de comercio que ya analizamos mediante backtracking. Es un problema de optimización donde buscamos minimizar el costo del circuito entre "n" ciudades. Es decir que buscamos obtener un ciclo hamiltoniano de menor costo posible para una instancia particular.

Para representar el árbol de estados utilizaremos la misma estructura que presentamos cuando resolvimos este problema mediante backtracking. La raíz corresponde al comienzo del ciclo partiendo de la ciudad inicial. Desde allí se pueden visitar un subconjunto de ciudades restringidas por aquellas en las que existe un camino que las una. Un nodo a profundidad "i" del árbol corresponde a un camino que comienza en la ciudad inicial y pasa por i-1 ciudades intermedias. Esas ciudades no se repiten. Desde la ciudad en la posición i-1 se puede seleccionar una próxima ciudad aún no visitada y que tenga un camino que arribe de la misma. Los estados solución corresponden a hojas del árbol ubicadas en profundidad n. Todos los estados solución que descienden de un nodo de profundidad "i" tienen al menos las primeras i-1 ciudades iguales en su recorrido.

En este caso utilizaremos la estrategia de best-first para recorrer el árbol. Para eso y para la poda debemos definir una función costo. En este caso, al ser un problema de minimización queremos desestimar la exploración de los descendientes de un nodo si la mejor solución encontrada hasta el momento o "mejorActual" es menor a la mejor solución posible que se puede obtener desde un estado solución accesible desde el mismo. Además entre todos los nodos sin explorar queremos seleccionar a aquel más promisorio: aquel que tiene más posibilidad de superar el mínimo camino factible encontrado. Un criterio simple sería simplemente podar si el costo acumulado supera la mejor solución obtenida. Sin embargo, se puede acotar mejor. Para hacerlo revisaremos en primer lugar las características del problema.

Podremos representar un circuito C como una lista de ciudades visitadas en orden: $x_1,x_2,...x_{n-1}$. Donde X_1 es la primera ciudad visitada desde la ciudad de origen "o" y x_n es la ciudad previa antes de regresar a "o". En este caso es costo total del camino corresponde a $Costo(C) = w_{o,x_1} + \left(\sum_{j=1}^{n-1} w_{x_j,x_{j+1}}\right) + w_{x_{n-1,o}}$. Podemos separar el circuito en dos partes C_1=

$x_1,...,xi$ y $C_2 = x_{i+1},...,x_{n-1}$. La primera corresponde en ir del origen a la ciudad intermedia "i" y la segunda regresar desde "i" por las ciudades aún no visitadas al origen. En este caso podemos expresar el costo del circuito como $Costo(C) = Costo(C_1) + Costo(C_2) = \left[w_{o,x_1} + \left(\sum_{j=1}^{i-1} w_{x_j x_{j+1}}\right)\right] + \left[\left(\sum_{j=i}^{n-1} w_{x_j x_{j+1}}\right) + w_{x_{n-1,o}}\right]$.

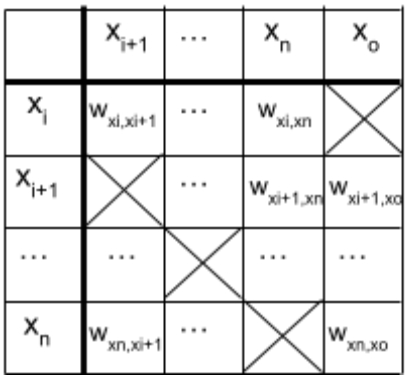


Repasamos la instancia de ejemplo que presentamos para este problema en backtracking. Contamos con un total de 4 ciudades. La ciudad de partida es "o". Un posible ciclo corresponde a o,3,1,2,o. El costo de este corresponde a $w_{o,3} + w_{3,1} + w_{1,2} + w_{2,o} = 3 + 2 + 4 + 4 = 13$. Si partimos el ciclo en la ciudad 3. Tendremos que C_1 corresponde a o,3 y C_2 a 1,2,o. Sus costos respectivos son 3 y 10.

La llegada a un nodo a profundidad "i" del árbol de estados corresponde a la concreción del circuito parcial C_1. Las ciudades visitadas, así como el costo acumulado lo podemos calcular sin errores. Si llamamos X al conjunto de todas las ciudades a recorrer, podemos calcular el subconjunto Y ⊆ X como aquellas ciudades aún no visitadas al llegar a ese nodo. Existirán posiblemente varias alternativas para generar el camino C_2. En el caso extremo todas las permutaciones posibles de las ciudades en Y. Cada una de estas con un costo diferente. Nos interesa poder medir el menor de los costos posibles al que llamaremos MinCosto(C_Y). Si $MejorActual \leq Costo(C_1) + MinCosto(C_y)$, sabemos que no hay un circuito mejor al existente que comience con el camino C_1.

El árbol de estados de la instancia del problema comienza por la ciudad o. Desde este se puede expandir en segunda instancia 3 nodos. Corresponde a elegir cual corresponde a la próxima ciudad a visitar. Supongamos que nos encontramos analizando el nodo correspondiente a, desde el origen, visitar la ciudad 3. El costo de C1 es 3. El conjunto Y de las ciudades a visitar son 1 y 2 (para luego regresar a "o"). Por tratarse de sólo 2 ciudades, existen 2! permutaciones posibles: 1,2 y 2,1. El costo de ellos (regresando o la ciudad inicial) en ambos casos es 13. Por lo tanto MinCosto(C_y)=13.

¿Cómo calculamos MinCosto(C_Y)? La dificultad reside en que su obtención implica un cálculo combinatorio (las permutaciones de las ciudades en Y). Por lo tanto no es eficiente. Reemplazamos este valor por uno cuyo cálculo sea más sencillo aunque puede no ser



	x_{i+1}	...	x_n	x_o
x_i	$w_{xi,xi+1}$	...	$w_{xi,xn}$	
x_{i+1}		...	$w_{xi+1,xn}$	$w_{xi+1,xo}$
...	...		...	...
x_n	$w_{xn,xi+1}$	...		$w_{xn,xo}$

exacto. Supongamos por un momento que todas las ciudades en Y están comunicadas entre sí y además comunicadas con la última ciudad xi de C_1. Podemos expresar los valores en una grilla donde la columna representa la ciudad de origen y la fila la de destino. En la celda se puede obtener el costo de ir desde la ciudad en fila a la ciudad en la columna. Un camino representado en la grilla corresponde a la utilización de una celda por fila y que esas celdas no coincidan en la columna con otra seleccionada en diferente fila. La suma de los valores de las celdas seleccionadas corresponde al costo del camino.

Podemos sugerir tomar por cada fila el menor peso disponible. Tomares como precaución no seleccionar para x_i la ciudad de origen (a menos que sea la única posibilidad). De hacerlo estaremos cerrando prematuramente el ciclo. Llamaremos a este valor MinCosto(Y) y se puede calcular como $min\left\{w_{x_i,z}\ /z \in Y\right\} + \sum_{y \in Y} min\left\{w_{y,z}\ /z \in Y \cup \{o\}\right\}$

Las celdas seleccionadas podrían no corresponder a un camino válido (Puede incluir 2 o más celdas de la misma columna). No obstante nos servirá de valor límite. La sumatoria de los pesos van a ser iguales o menores al costo del menor camino mínimo válido posible en C_y. Expresamos esa desigualdad como $MinCosto(C_y) \geq MinCosto(Y)$. Esta función nos servirá de función costo para determinar qué nodo explorar primero y cuales podar. Si $MejorActual \leq Costo(C_1) + MinCosto(Y)$ entonces no hay posibilidad que circuito válido que se pueda construir en esa rama del árbol de estado mejore la mejor solución encontrada hasta el momento. Calcular MinCosto(Y) tiene una complejidad de $O(n^2)$ que es radicalmente inferior a calcular las permutaciones que es O(n!).

Continuando con el ejemplo anterior, tenemos que C1=o,3 con costo 3, Y={1,2}. Podemos armar la grilla de 2x2. Las filas para 3, 1 y 2. Las columnas para 1,2 y "o". El mínimo costo de la fila del elemento "3" es 1. Para las filas de "1" y "2" son respectivamente 2 y 4. Por lo tanto MinCosto(Y)=7. Que como se puede observar no corresponde a un ciclo

	1	2	o
3	2	1	
1		4	2
2	7		4

válido (utiliza 2 veces la columna del elemento "o"). Pero que nos puede servir como cota inferior. Sumando este valor al costo de C1 llegamos a 10. Corresponde al valor a considerar en la poda para este nodo y al utilizado para determinar si es más promisorio que otros. Que el valor sea menor al real nos asegura que no podemos de forma equivocada una rama que podría tener dentro el óptimo.

Unificando todos los pasos establecidos, podemos expresar el pseudocódigo del algoritmo como:

Algoritmo genérico branch and bound (BeFS)

```
Sea C[x,y] el costo de ir de la ciudad x a la y
Sea A[y] la lista de ciudades adyacentes de la ciudad y.
Sea camino el recorrido realizada hasta el momento
Sea minimoCosto el costo del camino minimo encontrado
Sea minimoCamino el circuito de menor costo encontrado

camino={o}.
minimoCosto=infinito
minimoCamino={}

Definir fc=0 para costo de camino
Agregar camino con fc a caminosDisponibles

Mientras queden caminos estos en caminosDisponibles
        Sea caminoActual el camino de mayor fc en caminosDisponibles

        Sea x la ultima ciudad visitada en caminoActual.
        Si longitud del camino es n
                Si la ciudad o se encuentra en A[x]
                        Agregar al final de caminoActual la ciudad o
                        Sea costoCamino la suma de los costos de caminoActual
                        Si costoCamino < minimoCosto
                                minimoCosto = costoCamino
                                minimoCamino=camino

        Sino
                Por cada ciudad y en A[x] no visitada previamente excepto "o"
                        Sea caminoAmpliado = caminoActual+{y}
                        Determinar Y ciudades aun no visitadas en caminoAmpliado
                        Sea costoCamino la suma de los costos de caminoAmpliado
```

```
        Calcular fc=costoCamino +MinCosto(x,Y)
        Si fc < minimoCosto
                Agregar caminoAmpliado  con fc a caminosDisponibles
```

La complejidad final de este algoritmo está principalmente establecida por explorar en el peor de los casos todos los estados del árbol de estados en O(n!). En cada subproblema se requiere realizar procesos O(n) como determinar cuales son las ciudades aún no incluidas en el camino, calcular el costo actual, entro otros. Además el calcular la función límite tiene una complejidad $O(n^2)$. Por último, en este caso se está utilizando una cola de prioridad para mantener cada uno de los estados del árbol de estados que deben ser analizadas. La complejidad de esta estructura está dada en O(AlogA) con A la cantidad de elementos incluidos. En este caso A=n!. Por lo que podríamos unificar y hablar de una complejidad final de $O(n! (n^2 + \log n!)) = O(n! \log n!))$=. Un valor que crece de forma acelerada a medida que crece ”n”.

8. Branch and Bound: Clique máximo

Trabajamos con el problema de clique buscando su existencia en un grafo. Limitamos la búsqueda a uno de tamaño K. Es lícito y necesario en situaciones específicas conocer cuál es el clique de mayor tamaño. Entendemos el tamaño como su conformación por la mayor cantidad de vértices. Por ese motivo reformulamos el enunciado de nuestro problema a:

Problema del clique máximo
Dado un grafo no direccionado G=(V,E) con V su conjunto de vértices y E el de sus aristas. Queremos obtener el clique de tamaño mayor dentro de él.

Para la resolución por “generar y probar” debemos probar todo conjunto posible de vértices y determinar si corresponde a un clique. En caso afirmativo determinar si su tamaño es mayor al de todos los hallados con anterioridad. Al finalizar el proceso tendremos el clique máximo.

Dado el planteo, el espacio de soluciones del problema es similar al problema de la mochila. Así como probamos cada combinación de elementos en la mochila, en este caso

probamos cada posible combinación de vértices para determinar si corresponde a un clique y su tamaño. Ordenaremos arbitrariamente los vértices del grafo para tener una jerarquía en su evaluación. Llamaremos a este orden $V=[v_1,v_2, \dots , v_n]$. Lo utilizaremos a la hora de determinar los próximos nodos a generar en el árbol de estados.

Construiremos un árbol combinatorio. La raíz corresponde a no seleccionar ningún vértice del grafo (la ausencia de clique). Los descendientes de cada nodo corresponden a los posibles vértices a elegir posteriores a los ya agregados (teniendo n cuenta el ordenamiento de vértices planteado) para intentar conformar un clique de mayor tamaño. Todos los estados del problema corresponden a estados solución.

Si analizamos un estado del árbol, vemos que los vértices seleccionados del grafo al momento se pueden representar como una lista ordenada según el momento de inserción. Por ejemplo $[a_1, a_2, \dots a_i]$ corresponde al estado en un nodo de profundidad "i" en el que se agregó el vértice del grafo a_i. En su nodo padre se insertó a_{i-1} y sucesivamente. Esta jerarquía en la inserción es útil para generar una función de corte. Basta con que $[a_1, a_2, \dots a_{i-1}]$ no sea un clique para saber que $[a_1, a_2, \dots a_i]$ tampoco lo es. Esto nos permite podar el espacio de exploración y evitar analizar sin sentido descendientes del árbol sin posibilidad de brindar soluciones.

Con lo establecido hasta el momento podemos plantear un algoritmo de Backtracking para su resolución. Recorremos mediante DFS el árbol de estados hasta terminar de explorar todos sus nodos. En esa instancia retornamos la solución encontrada: los nodos que conforman el mayor clique encontrado y su cardinalidad. No obstante queremos llegar a un algoritmo de Branch And Bound. Para eso nos está faltando mínimamente agregar una función costo que nos ayude a achicar aún más el espacio de exploración. Analizaremos varias opciones que se pueden incluir.

Supongamos que hasta el momento encontramos un clique máximo $M=[m_1, m_2, \dots m_s]$ que contiene "s" vértices. Nos encontramos analizando un nodo del árbol de estados que al momento corresponde a un clique $A=[a_1, a_2, \dots a_i]$ con $i<s$ vértices del grafo. Podemos ver cuales son los vértices que faltan probar. Estos serán aquellos posteriores a a_i en el orden V. Llamamos a este subconjunto P y su cantidad de elementos p. Si $p+i<s$ entonces sabemos que es imposible que explorando los descendientes del nodo encontremos un clique de mayor tamaño al previamente encontrado.

Regresando al clique A en el nodo analizado, podemos ver que cada vértice del grafo a_x incluido tiene un grado. Es decir una cantidad de vértices adyacentes que llamamos grado(a_x). Un clique que contiene el vértice a_x no puede de ninguna manera superar el tamaño dado por el grado(a_x). Si del clique A se obtiene el vértice de menor grado, tendremos otro criterio para determinar si continuamos explorando o no los nodos descendientes del nodo. Formalmente: $min\{grado(a_x) / a_x \in A\} < s$

Por último podemos vincular ambos conceptos anteriores para generar una función límite de mejores prestaciones. Cada vértices a_x en el Clique A tiene el conjunto adj(a_x) de sus vértices adyacentes. Si realizamos la intersección de todos los adj(a_x) de A tendremos los vértices que todos comparten de vecinos. Tienen que estar los de A (sino sería absurdo que sea un clique) y eventualmente podrán existir otros vértices adicionales. En el caso extremo todos estos adicionales pueden formar parte de un clique. Matemáticamente podemos expresar el concepto como $B = \bigcap_{a_x \in A} adj(a_x)$. Si ahora realizamos una intersección de C=$B \cap P$ tendremos a aquellos vértices que faltan probar que podrían llegar a formar cliques de tamaño mayor con A. Llamamos c a la cantidad de elementos en C.

El valor c más el tamaño del clique A es un límite del tamaño de clique al que se podría llegar profundizando la exploración del nodo. Por lo tanto si $c + i < s$ podamos los descendientes del nodo. En caso contrario expandimos y seguimos explorando aunque solo utilizando los elementos de C como descendientes.

Es importante aclarar que el proceso de intersección no es necesario realizarlo desde el inicio al analizar un nodo, sino que se pueden utilizar los cálculos del nodo padre adaptándose a medida que se incorporan nuevos vértices.

Solo resta seleccionar el criterio del próximo nodo a analizar para tener completo el algoritmo de Branch and bound. Puede ser Best-First Search , depth-first branch-and-bound o cualquier otro que se establezca.