18 de noviembre de 2022

Estructura de Datos y Algoritmos

Ramificación y poda (puzzle)

Licenciatura en Ciencias de la Computación

Nicolas Quiroga - Martin gonzalez - NICOLAS LAHOZ

INDICE

Tabla de contenido

[Ramificación y poda 2](#_Toc119575428)

[Algunas consideraciones y características a tener en cuenta sobre esta técnica 2](#_Toc119575429)

[Problemática tratada 3](#_Toc119575430)

[Código Fuente 4](#_Toc119575431)

[Seguimiento del algoritmo 6](#_Toc119575432)

[Conclusiones 7](#_Toc119575433)

[Bibliografía 8](#_Toc119575434)

# Ramificación y poda

La técnica de diseños de algoritmos ramificación y poda/acotación (RyP), es una variante de la técnica de BackTracking (BT) notablemente mejorado, donde los algoritmos de RyP se aplican para resolver asuntos o problemas de optimización. Esta técnica se suele interpretar como un árbol de soluciones, donde cada rama conduce a una posible solución siguiente a la actual. Esta técnica, utiliza cotas para podar (eliminar o descartar) aquellas ramas del árbol que no conducen a la solución óptima. Para ello calcula en cada nodo una cota del posible valor de aquellas soluciones alcanzables desde ése. Si la cota muestra que cualquiera de estas soluciones tiene que ser necesariamente peor que la mejor solución hallada hasta el momento no necesitamos seguir explorando por esa rama del árbol, lo que permite realizar el proceso de poda.

Lo que le da valor a esta técnica es la posibilidad de disponer de distintas estrategias de exploración del árbol y de acotar la búsqueda de la solución, que en definitiva se traduce en eficiencia. La dificultad está en encontrar una buena función de coste para el problema, buena en el sentido de que garantice la poda y que su cálculo no sea muy costoso.

Un concepto importante para entender el algoritmo de ramificación y poda es el de nodo vivo, también llamado estado. Un nodo vivo del árbol de expansión es un nodo con posibilidades de ser ramificado, es decir, un nodo que no ha sido podado. En cambio un nodo muerto, es un nodo del que no van a generarse más hijos por algunas de las siguientes razones: ya se han generado todos sus hijos, no es factible o no es prometedor (un nodo es prometedor si la información que tenemos de ese nodo indica que expandiéndolo se puede conseguir una solución mejor que la solución en curso), pero en cualquier instante del algoritmo pueden existir muchos nodos vivos y muchos nodos muertos pero solo existe un nodo en expansión que es aquel del que se están generando sus hijos en ese instante. Para determinar en cada momento qué nodo va a ser expandido y dependiendo de la estrategia de búsqueda seleccionada, necesitaremos almacenar todos los nodos vivos/estados en alguna estructura que podamos recorrer.

Por otro lado, en un algoritmo de Ramificación y Poda básico, se realizan tres etapas:

La primera de ellas, denominada de Selección, se encarga de extraer un nodo de entre el conjunto de los nodos vivos. La forma de escogerlo va a depender directamente de la estrategia de búsqueda que decidamos para el algoritmo.

En la segunda etapa, la Ramificación, se construyen los posibles nodos hijos del nodo seleccionado en el paso anterior.

Finalmente se realiza la tercera etapa, la Poda, en la que se eliminan algunos de los nodos creados en la etapa anterior. Esto contribuye a disminuir en lo posible el espacio de búsqueda y así atenuar la complejidad de estos algoritmos basados en la exploración de un árbol de posibilidades. Aquellos nodos no podados pasan a formar parte del conjunto de nodos vivos, y se comienza de nuevo por el proceso de selección.

# Algunas consideraciones y características a tener en cuenta sobre esta técnica

El algoritmo de Ramificación y Poda es una técnica basada en el recorrido de un árbol que contiene soluciones en las cuales:

* Se realiza un recorrido sistemático en un árbol de soluciones
* El recorrido no tiene por qué ser necesariamente en profundidad, sino que seguirá una estrategia de ramificación, guiada por estimaciones del beneficio que se realizaran para cada nodo.
* Se utilizan técnicas de poda para eliminar nodos que no lleven a la solución óptima, donde esta se realiza estimando en cada nodo las cotas de beneficio que se pueden obtener a partir del mismo.

Además, una característica que le hace diferente al diseño de BackTracking es la posibilidad de generar nodos siguientes distintas estrategias, ya que el BT realiza la generación de descendientes de una manera sistemática y de la misma forma para todos los problemas, haciendo un recorrido en profundidad del árbol que representa el espacio de soluciones, es decir que este explora todo el árbol de posibles soluciones.

En cambio, en el diseño Ramificación y Poda las posibles estrategias de ramificación y exploración del árbol de estados a emplear son las siguientes:

* Estrategia FIFO (first in first out): La lista de nodos vivos es una cola y, por lo tanto, el recorrido del árbol se realiza en anchura.
* Estrategia LIFO (last in first out): La lista de nodos vivos es una pila y, por lo tanto, el recorrido del árbol se realiza en profundidad.
* Estrategia LC (least cost): Se selecciona de toda la lista de nodos vivos aquél con el que se obtenga un mayor beneficio (o menor coste) para explorar a continuación.

También es importante tener en cuenta que la ventaja adicional que poseen estos algoritmos: la posibilidad de ejecutarlos en paralelo. Debido a que disponen de un conjunto de nodos vivos sobre el que se efectúan las tres etapas del algoritmo antes mencionadas, nada impide tener más de un proceso trabajando sobre este conjunto, extrayendo nodos, expandiéndolos y realizando la poda. El disponer de algoritmos paralelizables es muy importante en muchas aplicaciones en las que es necesario abordar los problemas de forma paralela para resolverlos en tiempos razonables, debido a su complejidad.

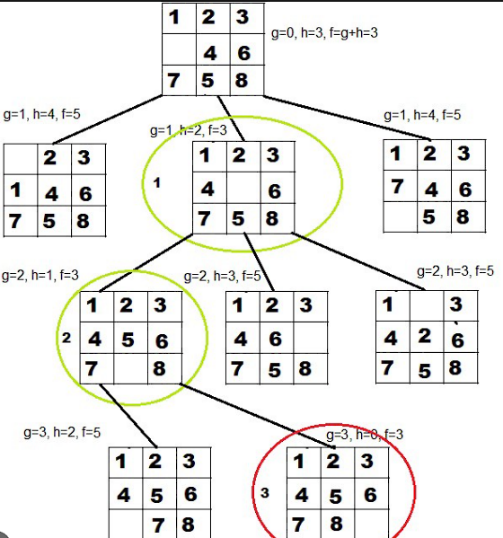
Pero sus requerimientos de memoria son mayores que los de los algoritmos BackTracking. Ya que no se puede disponer de una estructura global en donde ir construyendo la solución, puesto que el proceso de construcción deja de ser tan “ordenado” como antes. Ahora se necesita que cada nodo sea autónomo, en el sentido que ha de contener toda la información necesaria para realizar los procesos de bifurcación y poda, y para reconstruir la solución encontrada hasta ese momento.

# Problemática tratada

Este juego es una generalización del Puzzle-15 ideado por Sam Loyd en 1878. Disponemos de un tablero con n2 casillas y de n2 –1 piezas numeradas del uno al n2 –1. Dada una ordenación inicial de todas las piezas en el tablero, queda sólo una casilla vacía, a la que denominaremos “hueco”. Nuestro objetivo es transformar, mediante movimientos legales de la fichas, dicha disposición inicial de las piezas en una disposición final ordenada, en donde en la casilla [i,j] se encuentra la pieza numerada (i–1)\*n + j y en la casilla [n,n] se encuentra el hueco. Los únicos movimientos permitidos son los de las piezas adyacentes (horizontal y verticalmente) al hueco, que pueden ocuparlo; al hacerlo, dejan el hueco en la posición en donde se encontraba la pieza antes del movimiento. Otra forma de abordar el problema es considerar que lo que se mueve es el hueco, pudiendo hacerlo hacia arriba, abajo, izquierda o derecha (siempre sin salirse del tablero). Al moverse, su casilla es ocupada por la pieza que ocupaba la casilla a donde se ha “movido” el hueco.

Es posible resolver el problema mediante Ramificación y Poda utilizando dos funciones de coste diferentes: a) La primera calcula el número de piezas que están en una posición distinta de la que les corresponde en la disposición final. b) La segunda se basa en la suma de las distancias de Manhattan desde la posición de cada pieza a su posición en la disposición final. La distancia de Manhattan entre dos puntos del plano de coordenadas (x1,y1) y (x2,y2) viene dada por la expresión |x1 – x2| + |y1 – y2|.

# Código Fuente

Para resolver la problemática tratada, decidimos utilizar el Algoritmo, A\* , donde es un algoritmo de búsqueda inteligente o informada que busca el camino más corto desde un estado inicial al estado meta a través de un espacio de problema, usando una heurística óptima (en nuestro caso consideramos la distancia de manhattan y fuera de lugar). Este algoritmo utiliza una función de evaluación f(n) = g(n) + h'(n), donde h'(n) representa el valor heurístico del nodo a evaluar desde el actual, n, hasta el final, y g(n), el costo real del camino recorrido para llegar a dicho nodo, n. A\* mantiene dos estructuras de datos auxiliares, que podemos denominar abiertos, implementado como una [cola de prioridad](https://www.ecured.cu/index.php?title=Cola_de_prioridad&action=edit&redlink=1) ordenada por el valor f(n) de cada nodo, y cerrados, donde se guarda la información de los nodos que ya han sido visitados. En cada paso del algoritmo, se expande el nodo que esté primero en abiertos, y en caso de que no sea un nodo objetivo, calcula la f(n) de todos sus hijos, los inserta en abiertos, y pasa el nodo evaluado a cerrados. El algoritmo es una combinación entre búsquedas del tipo [primero en anchura](https://www.ecured.cu/index.php?title=Primero_en_anchura&action=edit&redlink=1) con [primero en profundidad](https://www.ecured.cu/index.php?title=Primero_en_profundidad&action=edit&redlink=1): mientras que h'(n) tiende a primero en profundidad, g(n) tiende a primero en anchura. De este modo, se cambia de camino de búsqueda cada vez que existen nodos más prometedores.

**class** **Estrategia**(ABC):

"""

Propiedades de performance de las diferentes estrategias para resolver el puzzle

"""

num\_nodos\_expandidos = **0** #cuántos nodos en el árbol de búsqueda necesitan expandirse los algoritmos para resolver el rompecabezas

solucion = None # la secuencia de operaciones para resolver el rompecabezas

**@abstractmethod**

**def** **resolver\_puzzle**(self):

"""

El algoritmo para resolver el puzzle

:return: Lista con objetos de puzzle

"""

**raise** NotImplemented

**class** **AEstrella**(Estrategia):

HALLAR\_DISTANCIA\_MANHATTAN, HALLAR\_FUERADELUGAR = 'hallar\_distancia\_de\_manhattan', 'Hallar\_fuera\_de\_Lugar'

CONSTANTES\_HEURISTICAS = [HALLAR\_DISTANCIA\_MANHATTAN, HALLAR\_FUERADELUGAR]

#funciones que se pueden seleccionar para calcular el valor heuristico

hallar\_funciones = {

HALLAR\_DISTANCIA\_MANHATTAN: Puzzle.hallar\_distancia\_de\_manhattan.\_\_name\_\_,

HALLAR\_FUERADELUGAR: Puzzle.Hallar\_fuera\_de\_Lugar.\_\_name\_\_

}

**def** **\_\_init\_\_**(self, puzzle\_inicial, heuristic=None):

"""

:param puzzle\_inicial: Puzzle

:param heuristic: 'distancia de manhattan' (default) or 'fuera de lugar'

"""

self.start = puzzle\_inicial

self.hallar\_funcion = self.hallar\_funciones[self.HALLAR\_DISTANCIA\_MANHATTAN]

**if** heuristic:

**if** heuristic **not** **in** self.CONSTANTES\_HEURISTICAS:

**raise** **RuntimeError**(f'Nombre de funcion heuristica invalido. Debe ser {self.CONSTANTES\_HEURISTICAS}')

self.hallar\_funcion = self.hallar\_funciones[heuristic]

**def** **\_\_str\_\_**(self):

**return** 'Estrategia con algoritmo A\*'

**def** **\_calcular\_nueva\_heuristica**(self, movimiento, nodo\_final):

"""

Heuristica que calcula que tan bueno es el movimiento actual

:param movimiento: Puzzle

:param nodo\_final: Puzzle

:return: valor heuristico (entero)

"""

**return** getattr(movimiento, self.hallar\_funcion)() - getattr(nodo\_final, self.hallar\_funcion)()

**def** **resolver\_puzzle**(self):

# Cada sublista en la cola es un camino a ser explorado y es el primer elemento del camino

# es el total heuristico (int) para el camino

cola = [[getattr(self.start, self.hallar\_funcion)(), self.start]]

camino = [] # el camino actual que queremos explorar

expandido = [] # Posiciones ya exploradas

num\_nodos\_expandidos = **0** # contador usado para el analisis de la performance

**while** cola:

# encuentra que camino en la cola tiene el menor valor heuristico

i = **0**

**for** j **in** range(**1**, len(cola)):

**if** cola[i][**0**] > cola[j][**0**]: # minimum

i = j

camino = cola[i] # toma el camino con el menor valor heuristico

cola = cola[:i] + cola[i + **1**:] # elimina el camino de la cola

nodo\_final = camino[-**1**] # La ultima posicion en el camino que estamos explorando

**if** nodo\_final.posicion == nodo\_final.POSICION\_FINAL\_PUZZLE: # la ultima posicion de nuestro camino es la ultima posicion

**break**

**if** nodo\_final.posicion **in** expandido: # evitar loop infinito

**continue**

**for** movimiento **in** nodo\_final.get\_movimientos(): # loop a traves de todos los movimientos posibles para la posicion actual

**if** movimiento.posicion **in** expandido:

**continue**

# agrega el camino con la nueva posicion y sus heuristicas al final de la cola

nuevo\_camino = [camino[**0**] + self.\_calcular\_nueva\_heuristica(movimiento, nodo\_final)] + camino[**1**:] + [movimiento]

cola.append(nuevo\_camino)

expandido.append(nodo\_final.posicion)

num\_nodos\_expandidos += **1** # incrementa el contador de rendimiento

# setea los valores de la clase base

self.num\_nodos\_expandidos = num\_nodos\_expandidos

self.solucion = camino[**1**:]

# Seguimiento del algoritmo

Para resolver el Puzzle nuestra función “resolver\_puzzle”, cuenta con los siguientes atributos a utilizar en la función:

* Listas:
  + Cola (Contiene el valor entero de la heurística a utilizar que puede ser la distancia de manhattan como fuera de lugar, y el otro valor que almacena es una estructura de tipo Puzzle)
  + Camino (Será el actual que queremos explorar y este nos proporcionará los movimientos más óptimos para alcanzar el objetivo del juego)
  + Expandido (Mantiene el seguimiento de las posiciones ya exploradas)
* Entero:
  + Num\_nodos\_expandidos (Contador usado para el análisis del rendimiento del algoritmo)

Luego iteramos con un while mientras la cola no quede vacía, seguido a esto realizamos una iteración con un for para obtener el camino en la cola que tiene el menor coste y es actualizada la variable de camino con un nuevo camino de menor costo heurístico. Se elimina este camino de la cola ya que fue almacenado en la variable camino y colocamos en el nodo final la última posición en el camino que estamos explorando.

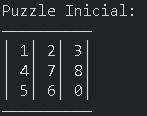
Posteriormente, si la posición del nodo final es igual a la posición de cómo debería de quedar resuelto el puzzle, la iteración es detenida y significa que hemos resuelto el juego. Si este if no se cumple, la ejecución continua y se evalúa que la posición del nodo final no se encuentre en los ya explorados para evitar un loop infinito por lo cual se utiliza la sentencia “continue” que será utilizada cuando en un bucle queramos obviar una parte del código. Una vez se ejecute esta sentencia, se omitirá todo el código existente después de ella y se volverá a ejecutar otra iteración del ciclo (hasta que termine).

Si este if no se cumple se realiza un loop para obtener todos los movimientos posibles para la posición actual, y se verifica que alguno de estos movimientos no haya sido explorado, si es así, se omite el resto del código y se pasa a la siguiente iteración. Luego se agrega el camino con la nueva posición y sus heurísticas al final de la cola, se incrementa en 1 el contador de rendimiento y si finaliza la iteración del while, se almacenan el numero de nodos expandidos para verificar el rendimiento del algoritmo y la solución que muestra el camino mas optimo.



# Conclusiones

En problemas de complejidad pequeña (número reducido de movimientos para resolver el problema) la heurística más eficaz es la de Manhattan, ya que es capaz de resolver el problema con menor consumo de memoria (utiliza menos nodos). A medida que la que la complejidad del problema va aumentando se va comprobando como la utilización de nodos por parte de las diferentes heurísticas va aumentando en grandes cantidades. Y debido a que el algoritmo A estrella es una combinación de primero en amplitud y en profundidad, además de considerar una cola de prioridad con los costos, puede llegar hacerlo muy eficiente dependiendo de la heurística utilizada y la construcción del algoritmo. De manera que si por ejemplo, tomamos el mismo tablero o puzzle con la heurística de la distancia de manhattan como con la de fuera de lugar, obtendríamos los siguientes resultados:



Utilizando la distancia de manhattan se obtiene lo siguiente:



El número de movimientos realizados para resolver el puzzle es de 18

En cambio, utilizando la heurística fuera de lugar, tenemos:



Y el numero de movimientos realizados es de 28, para el mismo juego.

Finalmente podemos observar, que para este algoritmo aplicado, la mejor herustica utilizada es la distancia de manhattan, ya que se requiere un menor numero de movimientos realizados para resolver el puzzle y también es menor la cantidad de nodos expandidos.

# Bibliografía

* <http://www.lcc.uma.es/~av/Libro/CAP7.pdf>
* <https://www.studocu.com/es-mx/document/instituto-tecnologico-superior-de-chicontepec/algebra-lineal/algoritmo-de-8-puzzle-apuntes-1/16745480>
* <https://webdiis.unizar.es/asignaturas/EDA/ea/slides/6-Ramificacion%20y%20acotacion.pdf>
* <https://www.geeksforgeeks.org/check-instance-15-puzzle-solvable/>
* <http://www.itnuevolaredo.edu.mx/takeyas/apuntes/Inteligencia%20Artificial/Apuntes/tareas_alumnos/A-Star/A-Star(2005-II-A).pdf>
* [https://www.ecured.cu/Algoritmo\_de\_B%C3%BAsqueda\_Heur%C3%ADstica\_A\*#.C2.BFC.C3.B3mo\_funciona\_A.2A.3F](https://www.ecured.cu/Algoritmo_de_B%C3%BAsqueda_Heur%C3%ADstica_A*#.C2.BFC.C3.B3mo_funciona_A.2A.3F)
* <https://www.infor.uva.es/~calonso/IAI/TrabajoAlumnos/Algoritmo_A.pdf>