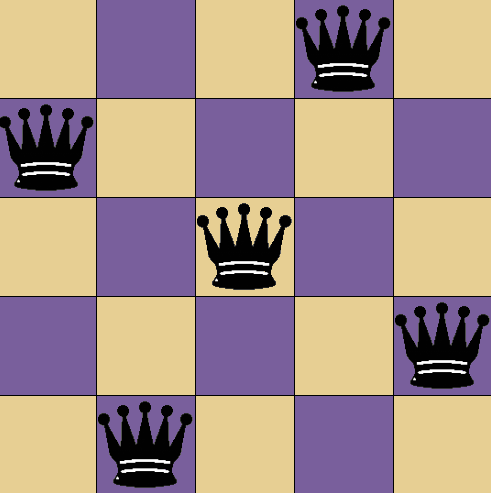
Estructura de datos y algoritmos



Técnica de diseño: **Ramificación y poda**

Ejemplo abordado: **N-Reinas**

Joaquín Molina

Maira Flores

2022

LCC, TUPW

Índice:

1. [*Descripción de la técnica de diseño y sus características - 3*](#_Descripción_de_la)
2. [*Descripción del problema - 4*](#_Descripción_del_problema)
3. [*Código fuente del algoritmo - 5*](#_Código_fuente_del)
4. [*Seguimiento del algoritmo - 6*](#_Seguimiento_del_algoritmo)
5. [*Conclusiones - 7*](#_Conclusiones)
6. [*Bibliografía 8*](#_Bibliografía)

# Descripción de la técnica de diseño

Esta técnica de diseño, cuyo nombre en castellano proviene del término inglés «**Branch and Bound**», se aplica normalmente para resolver problemas de optimización. Ramificación y poda, al igual que diseño **Vuelta atrás**, realiza una enumeración parcial del espacio de soluciones basándose en la generación de un árbol de expansión.

Una característica que posee es **la posibilidad de generar nodos siguiendo distintas estrategias**. El diseño Ramificación y Poda en su versión más sencilla puede seguir un **recorrido en achura** o **en profundidad**, o utilizando el cálculo de funciones de coste para seleccionar el nodo que en principio parece **más prometedor**.

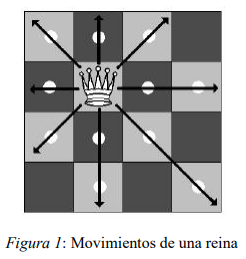
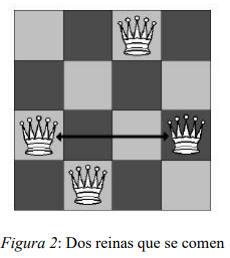
Definimos nodo **vivo** del árbol a un nodo con **posibilidades de ser ramificado**. Las búsquedas en amplitud utilizan una cola para almacenar los nodos de manera que se exploran en el orden que fueron creados. La **estrategia de mínimo coste** utiliza una «función de coste» para decidir en cada momento que nodo debe explorarse, con la esperanza de **alcanzar lo más rápidamente posible una solución más económica** que la mejor encontrada hasta el momento. Utilizaremos en este caso una estructura de montículo para almacenar los nodos ordenados por su coste.

En un algoritmo de Ramificación y Poda **se realizan 3 etapas**. La primera de ellas es la etapa de **Selección**, se encarga de extraer un nodo de entre el conjunto de los nodos vivos. En la segunda etapa, la **Ramificación**, **se construyen** los posibles nodos hijos del nodo seleccionado en el paso anterior. Por último, la etapa de **Poda**, en la que **se eliminan algunos de los nodos creados** en la etapa anterior. Esto contribuye a **disminuir de estos algoritmos la cantidad de posibilidades.** Ya que estos se basan en la exploración de un árbol de posibilidades.

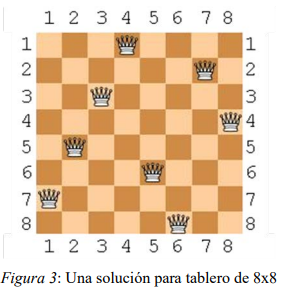
Para cada nodo dispondremos de una función de coste, que nos estime si el valor optimo es **candidato a ramificación**. Si la cota para un nodo deberá ser mejor que la solución real, o igual, **si es peor** que una solución ya obtenida por otra rama, **podemos podar** **esa rama** pues no es interesante ramificarla. Evidentemente no podemos realizar ninguna poda hasta que hayamos encontrado alguna solución, las funciones de coste han de ser crecientes respecto a la profundidad del árbol, es decir, si h es una función de coste entonces h(n) ≤ h(n’) para todo n’ nodo descendiente de n.

# Descripción del problema

Se trata de colocar **N reinas** en un tablero de NxN de tal forma que no se **puedan comer** entre sí. En **la figura 1** se muestran **los movimientos posibles de una reina** en un tablero de 4x4 (movimientos horizontales, verticales y diagonales), mientras que la figura 2 muestra **un ejemplo de dos reinas que pueden comerse**.



Un ejemplo de una posible solución para **un tablero de 8x8** sería el siguiente:



# Código fuente del algoritmo

import numpy as np

N=4

class Nodo:

    \_\_arreglo = None #arreglo posee la reina

    \_\_dimension = None #dimension del arreglo

    \_\_pos = None #coordenadas reina

    \_\_ult=0 #ulitma reina

    \_\_cota=999

    \_\_sig=None

    def \_\_init\_\_ (self, cant, x=None, y=None):

        self.\_\_dimension = cant

        self.\_\_arreglo = np.full(cant, -1)

        self.\_\_pos=[]

        self.\_\_sig=None

        self.\_\_cota=999

        if x != None and y != None:

            self.\_\_pos.append(x)

            self.\_\_arreglo[x]=y

    def setCota(self,cota):

        self.\_\_cota=cota

    def getCota(self):

        return self.\_\_cota

    def setArray(self,a):

        self.\_\_arreglo=a

    def getArray(self):

        return self.\_\_arreglo

    def setSig(self,elemento):

        self.\_\_sig=elemento

    def setReina(self,x,y):

        self.\_\_pos.append(x)

        self.\_\_arreglo[x]=y

        self.\_\_ult+=1

    def setReinas(self,a):

        self.\_\_pos=a

    def getReinas(self):

        return self.\_\_pos

    def getX(self):

        if self.\_\_pos != None:

            return self.\_\_pos[self.\_\_ult]

    def getY(self):

        if self.\_\_pos!= None:

            return self.\_\_arreglo[self.\_\_pos[self.\_\_ult]]

    def choca(self,reina): #trae solo una reina

        choca=0

        if len(reina.getReinas()) >= 1:

            for x in self.\_\_pos:

                if abs(x+self.\_\_arreglo[x]) != abs (reina.getX()+reina.getY()) and abs(x-self.\_\_arreglo[x]) != abs(reina.getX()-reina.getY()):#diagonales

                            if x != reina.getX() and self.\_\_arreglo[x] != reina.getY():#vertical y horizontal

                                pass

                            else:

                                choca+=1

                else:

                    choca +=1

            return choca

        else:

            return choca

class Grafo: #grafo conexo

    \_\_raiz=None

    \_\_LNV=None #LISTA DE NODOS VIVOS

    def \_\_init\_\_(self):

        self.\_\_LNV=[]

        self.\_\_raiz=None

    def Buscar(self, elemento):

        bandera = True

        i = 0

        while bandera and i < len(self.\_\_vertices):

            if self.\_\_vertices[i] == elemento:

                bandera = False

            else:

                i += 1

        if bandera == False:

            return i

        else:

            return None

    def CrearNodoInicial(self):

        new=Nodo(N,0,0)

        new1=Nodo(N,0,1)

        nuevo=Nodo(N,0,2)

        new3=Nodo(N,0,3)

        self.\_\_LNV.append(new)

        self.\_\_LNV.append(new1)

        self.\_\_LNV.append(nuevo)

        self.\_\_LNV.append(new3)

    def Genera(self,nivel):

        lista=[]

        i=0

        for i in range(N):

                new=Nodo(N,i,nivel)

                lista.append(new)

        return lista

    def BuscarHijo(self,nivel):

        lista=self.Genera(nivel)

        print(lista)

        return lista

    def BuscarLNV(self,elemento):

        i=0

        while(i<len(self.\_\_LNV) and elemento!=self.\_\_LNV[i]):

            i+=1

        if i<len(self.\_\_LNV) and elemento==self.\_\_LNV[i]:

            return i

        else:

            print("No se encontro el elemento en la LNV")

    def Arbol(self):

        k=0

        for nivel in range(1,4):

            for j in self.\_\_LNV:

                    listah=[]

                    print("-------------")

                    print("nivel:",nivel)

                    print("padre:",j.getX(),j.getY())

                    listaHijos=self.BuscarHijo(nivel)

                    listah.append(self.Asociar(listaHijos,j))

            self.\_\_LNV.pop(self.BuscarLNV(j))

            self.\_\_LNV+=listah

    def Asociar(self,hijos,padre):

        for i in hijos:

            print("hijo;x y:",i.getX(),i.getY(),"colision:",padre.choca(i))

            if i.choca(padre)==0 and i.choca(padre)<=padre.getCota():

                print("nodos vivos",i.getX(),i.getY())

                padre.setReina(i.getX(),i.getY())

                a=Nodo(N)

                nuevo=self.Copiar(a,i)

                return padre

    def Copiar(self,A,B):

        A.setArray(B.getArray())

        A.setReinas(B.getReinas)

    def Mostrar(self):

        for i in self.\_\_LNV:

            print("lista de nodos vivos----------")

            print("(",i.getX(),",",i.getY(),")")

            print("---------------")

'''

repetir

 elegir el nodo mas prometedor como nodo\_E;

 generar todos sus hijos;

 matar el nodo\_E;

 para cada hijo hacer

    si colision(hijo) > colision(mejor\_solucion\_en\_curso)

        entonces se mata

    sino

        si no es solucion

            entonces se pasa a la lista\_de\_nodos\_vivos

        sino

            {es solucion: el coste no es estimado sino real}

            es la mejor\_solucion\_en\_curso y se revisa la lista\_de\_nodos\_vivos,

            eliminando los que prometen algo peor

        fsi

    fsi

 fpara

hasta que la lista esta vacia

 '''

if \_\_name\_\_ == '\_\_main\_\_':

    grafo=Grafo()

    grafo.CrearNodoInicial()

    grafo.Arbol()

# Seguimiento del algoritmo

import numpy as np

N=4

class Nodo:

    \_\_arreglo = None #arreglo posee la reina

    \_\_dimension = None #dimension del arreglo

    \_\_pos = None #coordenadas reina

    \_\_ult=0 #ulitma reina

    \_\_cota=999

    \_\_sig=None

    def \_\_init\_\_ (self, cant, x=None, y=None):

        self.\_\_dimension = cant

        self.\_\_arreglo = np.full(cant, -1)

        self.\_\_pos=[]

        self.\_\_sig=None

        self.\_\_cota=999

        if x != None and y != None:

            self.\_\_pos.append(x)

            self.\_\_arreglo[x]=y

    def setCota(self,cota):

        self.\_\_cota=cota

    def getCota(self):

        return self.\_\_cota

    def setArray(self,a):

        self.\_\_arreglo=a

    def getArray(self):

        return self.\_\_arreglo

    def setSig(self,elemento):

        self.\_\_sig=elemento

    def setReina(self,x,y):

        self.\_\_pos.append(x)

        self.\_\_arreglo[x]=y

        self.\_\_ult+=1

    def setReinas(self,a):

        self.\_\_pos=a

    def getReinas(self):

        return self.\_\_pos

    def getX(self):

        if self.\_\_pos != None:

            return self.\_\_pos[self.\_\_ult]

    def getY(self):

        if self.\_\_pos!= None:

            return self.\_\_arreglo[self.\_\_pos[self.\_\_ult]]

    def choca(self,reina): #trae solo una reina

        choca=0

        if len(reina.getReinas()) >= 1:

            for x in self.\_\_pos:

                if abs(x+self.\_\_arreglo[x]) != abs (reina.getX()+reina.getY()) and abs(x-self.\_\_arreglo[x]) != abs(reina.getX()-reina.getY()):#diagonales

                            if x != reina.getX() and self.\_\_arreglo[x] != reina.getY():#vertical y horizontal

                                pass

                            else:

                                choca+=1

                else:

                    choca +=1

            return choca

        else:

            return choca

class Grafo: #grafo conexo

    \_\_raiz=None

    \_\_LNV=None #LISTA DE NODOS VIVOS

    def \_\_init\_\_(self):

        self.\_\_LNV=[]

        self.\_\_raiz=None

    def Buscar(self, elemento):

        bandera = True

        i = 0

        while bandera and i < len(self.\_\_vertices):

            if self.\_\_vertices[i] == elemento:

                bandera = False

            else:

                i += 1

        if bandera == False:

            return i

        else:

            return None

    def CrearNodoInicial(self):

        new=Nodo(N,0,0)

        new1=Nodo(N,0,1)

        nuevo=Nodo(N,0,2)

        new3=Nodo(N,0,3)

        self.\_\_LNV.append(new)

        self.\_\_LNV.append(new1)

        self.\_\_LNV.append(nuevo)

        self.\_\_LNV.append(new3)

    def Genera(self,nivel):

        lista=[]

        i=0

        for i in range(N):

                new=Nodo(N,i,nivel)

                lista.append(new)

        return lista

    def BuscarHijo(self,nivel):

        lista=self.Genera(nivel)

        print(lista)

        return lista

    def BuscarLNV(self,elemento):

        i=0

        while(i<len(self.\_\_LNV) and elemento!=self.\_\_LNV[i]):

            i+=1

        if i<len(self.\_\_LNV) and elemento==self.\_\_LNV[i]:

            return i

        else:

            print("No se encontro el elemento en la LNV")

    def Arbol(self):

        k=0

        for nivel in range(1,4):

            for j in self.\_\_LNV:

                    listah=[]

                    print("-------------")

                    print("nivel:",nivel)

                    print("padre:",j.getX(),j.getY())

                    listaHijos=self.BuscarHijo(nivel)

                    listah.append(self.Asociar(listaHijos,j))

            self.\_\_LNV.pop(self.BuscarLNV(j))

            self.\_\_LNV+=listah

    def Asociar(self,hijos,padre):

        for i in hijos:

            print("hijo;x y:",i.getX(),i.getY(),"colision:",padre.choca(i))

            if i.choca(padre)==0 and i.choca(padre)<=padre.getCota():

                print("nodos vivos",i.getX(),i.getY())

                padre.setReina(i.getX(),i.getY())

                a=Nodo(N)

                nuevo=self.Copiar(a,i)

                return padre

    def Copiar(self,A,B):

        A.setArray(B.getArray())

        A.setReinas(B.getReinas)

    def Mostrar(self):

        for i in self.\_\_LNV:

            print("lista de nodos vivos----------")

            print("(",i.getX(),",",i.getY(),")")

            print("---------------")

'''

repetir

 elegir el nodo mas prometedor como nodo\_E;

 generar todos sus hijos;

 matar el nodo\_E;

 para cada hijo hacer

    si colision(hijo) > colision(mejor\_solucion\_en\_curso)

        entonces se mata

    sino

        si no es solucion

            entonces se pasa a la lista\_de\_nodos\_vivos

        sino

            {es solucion: el coste no es estimado sino real}

            es la mejor\_solucion\_en\_curso y se revisa la lista\_de\_nodos\_vivos,

            eliminando los que prometen algo peor

        fsi

    fsi

 fpara

hasta que la lista esta vacia

 '''

if \_\_name\_\_ == '\_\_main\_\_':

    grafo=Grafo()

    grafo.CrearNodoInicial()

    grafo.Arbol()

# Conclusiones

El algoritmo de **Ramificación y Poda** es una variante que muestra una **mejora notable** frente al ya conocido “**Backtracking**”, aunque, Si bien Ramificación y Poda es un algoritmo mejorado, **no es recomendable** utilizarlo en escenarios **donde el tiempo y memorias son escasos**, donde reinan los algoritmos como “**Algoritmos Ávidos**”.

# Bibliografía

Guerequeta, R.; Vallecillo, A. **Técnicas de diseño de algoritmos.** Servicio de Publicaciones de la Universidad de Málaga. 2° Ed. 2000. <http://www.lcc.uma.es/~av/Libro/indice.html>

A.A.E.D TEMA 7 **Ramificación y Poda**: <http://dis.um.es/~ginesgm/files/doc/tema7-2.pdf>

Andreas Spading, Pablo Itaim Ananias, **Problema de las n-reinas**. Valparaíso, 01 de julio del 2005 <https://www.cs.buap.mx/~zacarias/FZF/nreinas3.pdf>

Simona Bernardi, **Ramificación y Poda**, Presentación adaptada de la original de J. Campos. Universidad de Zaragoza. <https://webdiis.unizar.es/~jcampos/ab/material/6-BranchBound.pdf>