Métodos de búsqueda
Informados y no informados

Docentes:

- Rodrigo Ramele
- Juliana Gambini
- Juan Santos
- Paula Oseroff
- Eugenia Piñeiro
- Santiago Reyes

Alumnos:

- Apablaza, Matías (59714)
- Beade, Gonzalo (61223)
- D'Agostino, Leonardo Agustín (60335)

Fecha:

- Primer Cuatrimestre 2022

Comentarios al algoritmo original

Sean **A** un árbol, **F** un conjunto de nodos frontera de **A** y **Ex** el conjunto de nodos explorados.

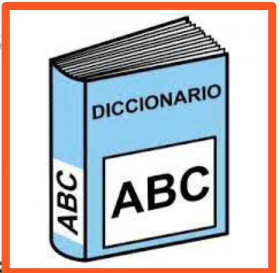
Algoritmo de búsqueda

1. Crear **A**, **F** y **Ex** inicialmente vacíos
2. Insertar el nodo raíz $\mathbf{n}_0$ en **A** y **F**.
3. Mientras **F** no esté vacía

aer el primer nodo **n** de **F**

no está en **Ex**, agregarlo

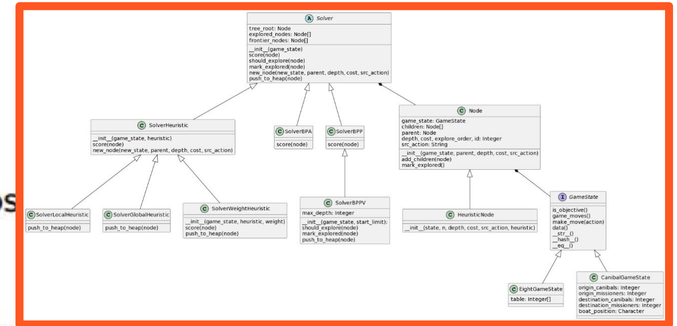
está etiquetado con un es



Devolver la solución, formada por los arcos

la raíz $\mathbf{n}_0$ y el nodo **n** en **A**

Termina Algoritmo.

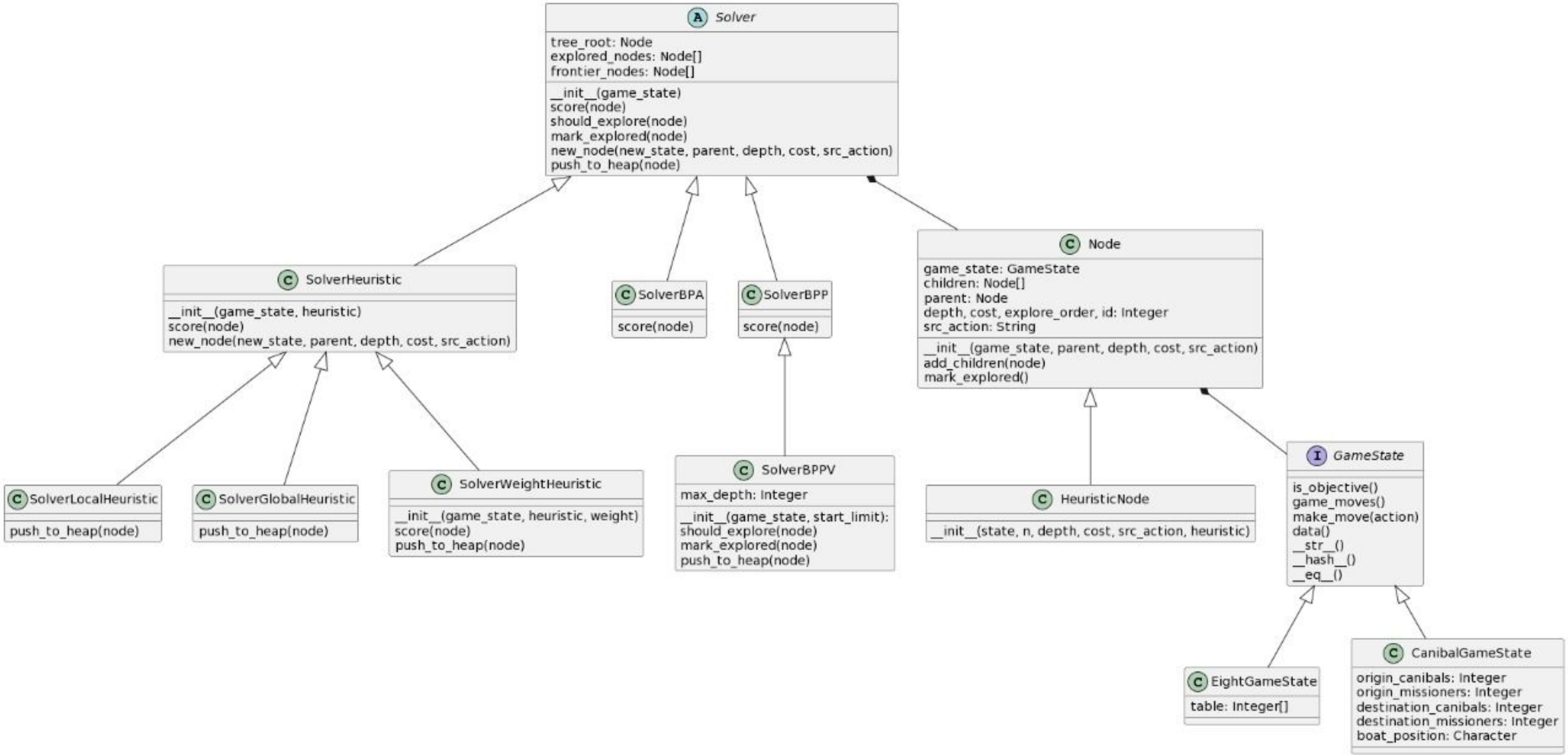


Expandir el nodo **n**, guardando los sucesores de **n** en **A** y, en **F**, si es que no están en **Ex**.

Reordenar **F** según el método de búsqueda.

4. Termina Algoritmo sin haber encontrado una solución.

Diagrama de clases



M

GameState abstracto

```python
class GameState():

  """ Devuelve verdadero si el estado es objetivo/ganador"""
  @property
  def isobjective(self):
    raise 'Not implemented exception'

  """
  Devuelve un conjunto de strings representando movimientos posibles
  según las reglas del juego
  """
  @property
  def game_moves(self):
    raise 'Not implemented exception'
```

```python
"""
Ejecuta un movimiento en el juego. De ser válido, devuelve un nuevo estado con dicho movimiento ejecutado
"""
def make_move(self, m: str):
  raise 'Not implemented exception'

"""
Todo juego debe poder mostrarse en pantalla, aunque sea una representación muy básica
"""
def __str__(self):
  return 'Empty Game!'

"""
Todo juego debe poder devolver un conjunto de pares con informacion del estado interno del mismo
"""
@property
def data(self):
  raise 'Not implemented exception'
```

Clase Node

```python
class Node:

    LAST_ID = 0
    LAST_EXPLORED_ORDER = 0

    def __init__(self, game_state: GameState, parent, depth, cost, src_action):
        self.game_state = game_state
        self.children = []
        self.parent = parent
        self.depth = depth
        self.cost = cost
        self.id = Node.LAST_ID
        self.explore_order = -1
        self.src_action = src_action
        Node.LAST_ID += 1

    def add(self, node):
      self.children.append(node)

    def mark_explored(self):
      self.explore_order = Node.LAST_EXPLORED_ORDER
      Node.LAST_EXPLORED_ORDER += 1
```

```python
@property
def state(self):
  return self.game_state

def __hash__(self):
  return hash(self.game_state)

def __eq__(self, other):
  return type(other) is Node and self.game_state == other.game_state
```

Clase abstracta Solver (1/4)

```
class Solver():

    '''
      La funcion score es una funcion que
      - dado un nodo representante de un estado del juego -
      permite  saber su puntaje. Mientras menor sea el puntaje, mejor.
      El algoritmo selecciona al que menor valor de score tiene.
      Si se quiere seleccionar al que mayor valor de score tenga, se lo puede multiplicar por -1
    '''
    def score(self, node):
      raise 'Not implemented exception'

    def __init__(self, game_state: GameState):
      self.initial_state = game_state
      self.root = self.new_node(self.initial_state, None, 0, 0)

    @property
    def initial_node(self):
      return self.root
```

Clase abstracta Solver (2/4)

```
def __iter__(self):
  # self.iter_done = False
  self.frontier = []
  self.explored = set()
  heappush(self.frontier, (self.score(self.root), 0, self.root))
  return self
```

Clase abstracta Solver (3/4)

```
def __next__(self):

  self.check_frontier()
  n = heappop(self.frontier)[-1]
  self.mark_explored(n)

  if n.state.isobjective:
    return n, True

  game_moves = n.game_state.game_moves
  for m in game_moves.keys():
    new_state = n.game_state.make_move(m)
    if new_state is not None:
      node = self.new_node(new_state, n, n.depth + 1, n.cost + game_moves[m])
      if self.should_explore(node):
        n.add(node)
        self.push_to_heap(node)

  # print(n.game_state)
  return n, False
```

Clase abstracta Solver (4/4)

```
def should_explore(self, node):
  return node.game_state not in self.explored

def new_node(self, new_state, parent, depth, cost):
  return Node(new_state, parent, parent.depth+1 if parent != None else 0, cost)

def check_frontier(self):
  if len(self.frontier) == 0:
    raise StopIteration

def push_to_heap(self, node):
    heappush(self.frontier, (self.score(node), node.id, node)) # Le agrego el ID para romper desempates

def mark_explored(self, n):
   n.mark_explored()
   self.explored.add(n.game_state)
```

D

Métodos No Informados

```
class SolverBPA(Solver):
  def score(self, node):
    return node.depth
```

```
class SolverBPP(Solver):
  def score(self, node):
    return -node.depth
```

Métodos No Informados

```python
class SolverBPPV(SolverBPP):

  def __init__(self, game_state: GameState, max_depth: int):
    super().__init__(game_state)
    self.max_depth = max_depth

  def mark_explored(self, n):
    n.mark_explored()
    self.explored[n.game_state] = min

  def __iter__(self):
    x =  super().__iter__()
    self.explored = {}
    return x

  def should_explore(self, node):
    return node.game_state not in self.explored.keys() or self.explored[node.game_state] > node.depth

  def push_to_heap(self, node):
    heappush(self.frontier, ( int(node.depth / (self.max_depth + 1)), self.score(node), node.id, node))
```

Sea R C { n / n es un nodo }
xRy <-> coc(x.depth, max_depth+1) = coc(y.depth, max_depth+1)
R es de equivalencia.

Métodos informados

```
class SolverHeuristic(Solver):

  def __init__(self, game_state: GameState, heuristic):
    self.h = heuristic
    super().__init__(game_state)

  def score(self, node):
    return node.heuristic

  def new_node(self, new_state, parent, depth, cost, src_action):
    n = self.HeuristicNode(new_state, parent, parent.depth+1 if parent != None else 0, cost, src_action, self.h(new_state))
    return n

  class HeuristicNode(Node):

    def __init__(self, state, n, depth, cost, src_action, heuristic):
      super().__init__(state, n, depth, cost, src_action)
      self.heuristic = heuristic
```

M

Métodos informados

```
class SolverLocalHeuristic(SolverHeuristic):

  def __init__(self, game_state: GameState, heuristic):
    super().__init__(game_state, heuristic)

  def push_to_heap(self, node):
    heappush(self.frontier, ( -node.depth, self.score(node), node.id, node))
```

```
class SolverGlobalHeuristic(SolverHeuristic):

  def __init__(self, game_state: GameState, heuristic):
    super().__init__(game_state, heuristic)

  def push_to_heap(self, node):
    heappush(self.frontier, (self.score(node), node.id, node))
```

Métodos informados

```
class SolverWeightHeuristic(SolverHeuristic):

  def __init__(self, game_state: GameState, heuristic, w):
    super().__init__(game_state, heuristic)
    self.w = w

  def score(self, node):
    return node.heuristic * self.w + node.cost * (1-self.w)

  def push_to_heap(self, node):
    heappush(self.frontier, (self.score(node), node.id, node))
```

Heurística #1 - Correctness (admisible)

$$Sea\ neq : \mathbb{N}^2 \rightarrow \{0,1\}\ /\ neq(x,y) = \begin{cases} 0 & x = y \\ 1 & sino \end{cases}$$

$$h : S \rightarrow \mathbb{N} / h(A) = \sum_{i=1}^{8} neq(A[i-1],\ i)$$

correctness (

8	2	6
5	1	4
7		3

) = 7

Heurística #1 - Correctness (admisible)

1) correctness_heuristic(state) == 0 sii todos los números están en su lugar.

$$h(A) = 0 \iff \sum_{i=1}^{8} neq(A[i-1],\ i) = 0 \iff neq(A[i-1],\ i) = 0\ \forall i \iff A[i-1] = i\ \forall i$$

2) es **admisible** pues nunca sobreestima el costo de llegar a una solución.

La demostración es directa. Como hay n números incorrectamente posicionados, tienen que ser movidos por el 0 al menos una vez. Por lo tanto, la cantidad de movimientos no puede ser menor que n.

D

Heurística #2 - Manhattan (admisible)

```
def manhattan_heuristic(state):
  table = state.data
  distance_sum = 0

  for position in table.keys():
    number = table[position]
    if number == 0:
      continue

    x1 = (int(position)-1) % 3
    y1 = int((int(position)) / 3)

    x2 = (number-1) % 3
    y2 = int((number-1) / 3)

    distance_sum += abs(x2-x1) + abs(y2-y1)

  return distance_sum
```

manhattan (

8	2	6
5	1	4
7		3

) = 11

Heurística #2 - Manhattan (admisible)

1) manhattan(state) == 0 sii todos los números están en su lugar.

La demostración es casi idéntica a la de correctness. Igualar sumandos a 0 y obtener xi'=xi y yi'=yi

2) manhattan es **admisible** pues nunca sobreestima el costo de llegar a una solución.

La demostración es directa. Un número mal puesto N tiene que recorrer como mínimo la distancia manhattan al lugar donde corresponde.

Heurística #3 - Manhattan^2 (no admisible)

```python
def manhattan_squared_heuristic(state):
  table = state.data
  distance_sum = 0

  for position in table.keys():
    number = table[position]
    if number == 0:
      continue

    x1 = (int(position)-1) % 3
    y1 = int((int(position)) / 3)

    x2 = (number-1) % 3
    y2 = int((number-1) / 3)

    distance_sum += (abs(x2-x1) + abs(y2-y1))**2

  return distance_sum
```

manhattan^2 (

8	2	6
5	1	4
7		3

) = 23

Heurística #3 - Manhattan^2 (no admisible)

1) manhattan^2(state) == 0 sii todos los números están en su lugar.

La demostración es casi análoga a la de manhattan. Igualar sumandos a , despejar el cuadrado y obtener xi'=xi y yi'=yi

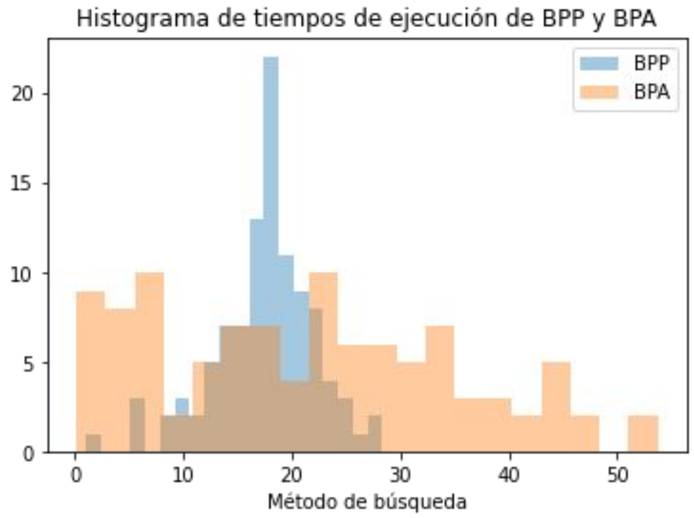
2) manhattan^2 **no** es admisible.

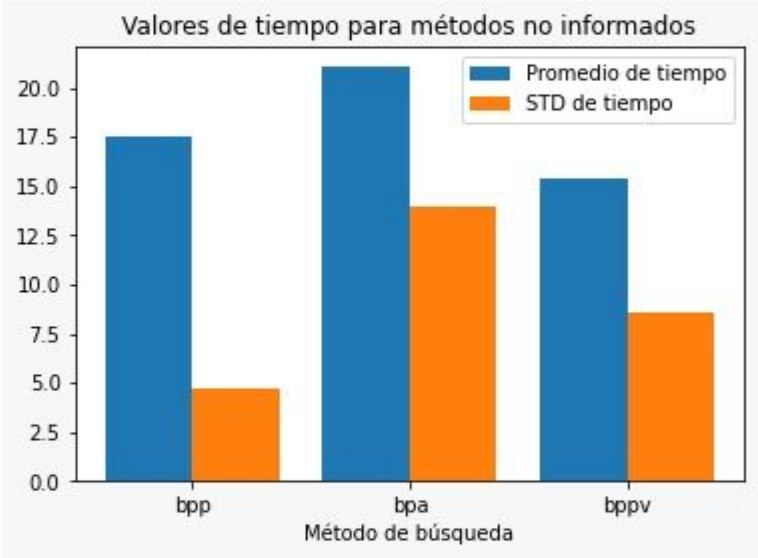
1	2	3
5	6	8
4		7

El contraejemplo que se muestra tiene valor de heurística 8, pero puede resolverse en 7 movimientos. Hacer circular a la casilla blanca en sentido antihorario por la segunda y tercera fila.

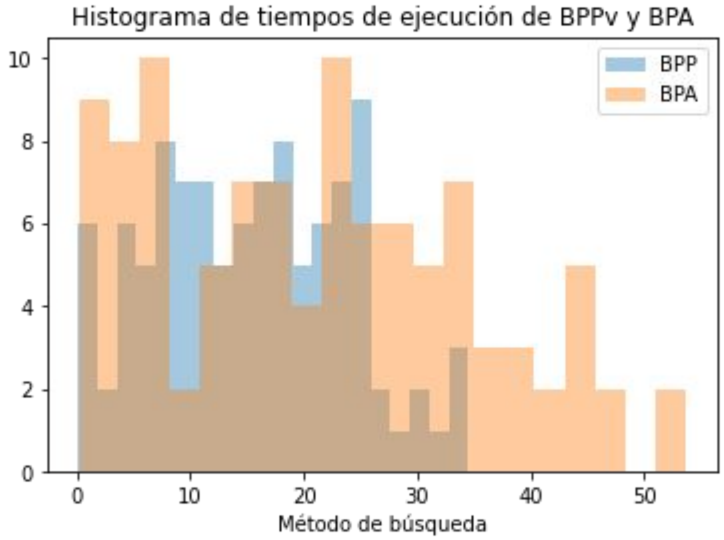
Gráficos - MNI

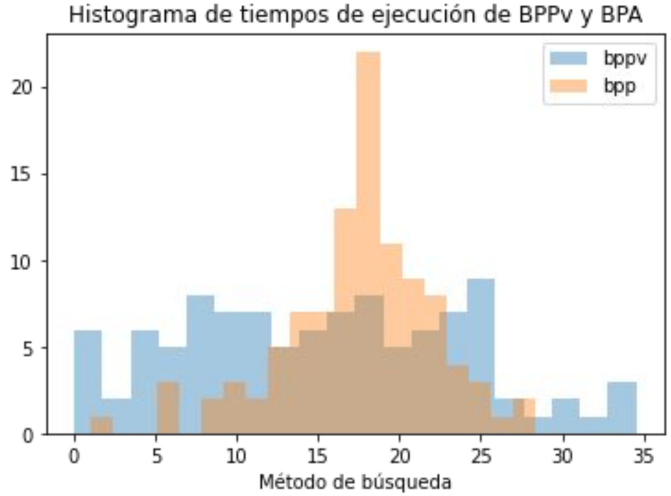
Corrimos 103 tableros distintos con todos los métodos de búsqueda.
En términos de cantidad óptima de movimientos, el menor costo que apareció fue 11, el mayor 28 y el promedio fue 22.



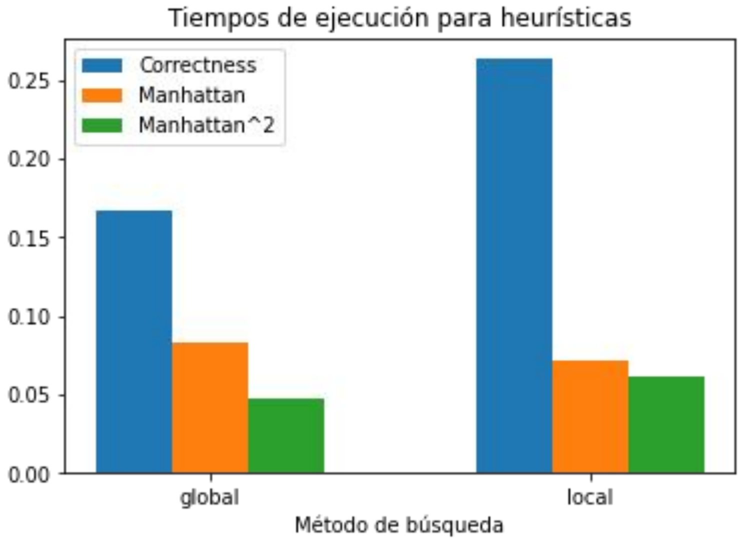


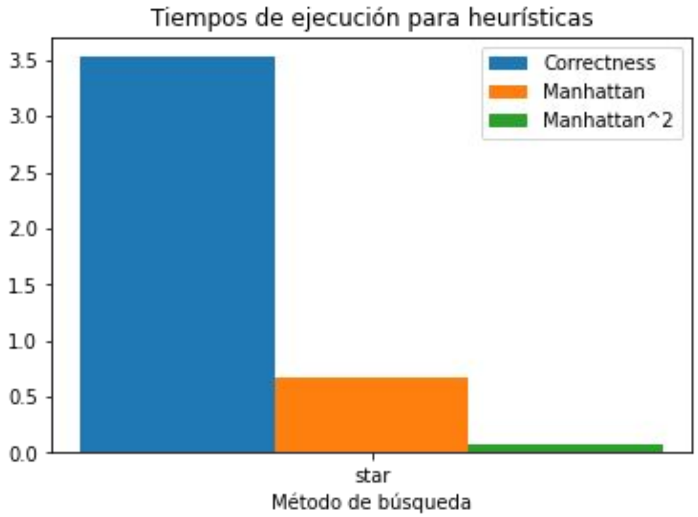
Gráficos - MNI





Gráficos - MI





D

Demo