MATEMÁTICA COMPUTACIONAL (CC 184)

INFORME DE TRABAJO FINAL

PROFESOR DEL CURSO:

Luis Martín Canaval Sánchez

INFORME PRESENTADO POR LOS ALUMNOS:

- Luis Gustavo Becerra Bisso U201915451
- Walter Jean Pierre Huapaya Chávez U201615183
- Stephano Heli Morales Linares U201912659

NOVIEMBRE 2020

INTRODUCCIÓN

El presente trabajo parcial se ha implementado en relación con el juego llamado Quoridor. El objetivo del juego es mover tu ficha a cualquier casilla del extremo opuesto del tablero, antes que tus rivales. Este juego se juega hasta con 4 personas y mínimo 2 personas. Para la implementación del juego se utilizará todo lo aprendido en lo que va del curso de Complejidad Algorítmica como Grafos, Backtracking y Notación Asintótica.

Por turnos, el jugador puede mover una ficha de forma horizontal o vertical, pero únicamente una casilla. Existe una forma de saltar a la ficha del otro jugador, que es cuando se encuentran las fichas de ambos jugadores una en frente de la otra. Asimismo, existen paredes que se dividen en cantidades iguales por cada persona que está jugando. Los jugadores pueden usar estas paredes para obstaculizar al rival, pero igualmente pierdes el turno. Para que se pueda utilizar el juego se necesita:

- Un tablero 9x9.
- 4 fichas y serán una por cada jugador.
- 20 bloques y serán divididos en partes iguales por cada jugador.

ESTADO DEL ARTE

Se han realizado numerosas investigaciones sobre algoritmos de búsqueda de grafo a nivel mundial. Si bien los estudios presentados a continuación no son los más recientes, se pueden aplicar a la perfección para el concepto del juego llamado Quoridor, ya que cumple la función de resolver laberintos en los cuales existen obstáculos.

Conceptos:

- **Breadth First Search** explora por igual en todas las direcciones. Este es un algoritmo increíblemente útil, no sólo para la búsqueda de rutas regulares, sino también para la generación de mapas de procedimiento, la búsqueda de rutas en campos de flujo, mapas de distancia y otros tipos de análisis de mapas.

- **El Algoritmo de Dijkstra** (también llamado Búsqueda de Costos Uniformes) nos permite priorizar qué caminos explorar. En lugar de explorar todos los caminos posibles por igual, favorece los caminos de menor costo. Podemos asignar costos más bajos para fomentar el movimiento en los caminos, costos más altos para evitar los bosques, costos más altos para desalentar el acercamiento a los enemigos, y más. Cuando los costos de movimiento varían, usamos esto en lugar de la primera búsqueda de la amplitud.

- **A*** es una modificación del Algoritmo de Dijkstra que está optimizado para un solo destino. El Algoritmo de Dijkstra puede encontrar caminos a todos los lugares; A* encuentra caminos a un lugar, o al más cercano de varios lugares. Da prioridad a los caminos que parecen conducir más cerca de un objetivo.

Shortest Path Searching for Road Network using A* Algorithm

Este documento de investigación tiene como objetivo encontrar la ruta más corta dentro de una red de carreteras. Para poder llevar a cabo su investigación, compararon dos algoritmos, Dijkstra y A*. Ambos algoritmos fueron sometidos a pruebas en búsqueda bidireccional, es decir, búsqueda del camino desde el punto final y del punto inicial, y unidireccional, es decir, sólo búsqueda desde el punto de partida. Además, se le añadió la condición de que, en ambos casos, puedan tener o no obstáculos.

Después de realizado los experimentos, se obtuvieron los siguientes datos:

➢ Sin obstáculos:

Algorithm	Condition	Length	Time	Operation
A star	One-Direction	38	3.8190	553
Dijkstra	One-Direction	38	14.8760	1933
A star	Bi-direction	38	0.5920	348
Dijkstra	Bi-direction l	38	8.2140	1618

➢ Con obstáculos:

Algorithm	Condition	Length	Time	Operation
A star	One direction	38	5.095	361
Dijkstra	One direction	38	12.1030	1601
A star	Bi-direction	38	3.8220	550
Dijkstra	Bi-directional	38	6.1370	1233

Como se puede observar, el algoritmo A* obtuvo un mejor resultado en todas las pruebas a las que fueron sometidos ambos algoritmos. Por lo que se concluyó que el algoritmo A* es mucho más eficiente en la búsqueda de la ruta más corta que el algoritmo de Dijkstra.

- **DFS:** Algoritmo que se utiliza para poder recorrer todos los nodos de un grafo o árbol de manera ordenada. Se centra en ir lo más profundo dentro de un árbol para poder encontrar la solución en lugar de expandirlo horizontalmente como lo hace el BFS.

METODOLOGÍA

Luego de las investigaciones, se buscaron algoritmos relacionados para poder realizar el juego de Quoridor. Los pasos que haremos en las semanas restantes antes de la entrega final son:

- Se asignarán tareas en la plataforma de trello para poder tener un control de las actividades realizadas.
- Se hará uso del repositorio de Github, en donde cada miembro del equipo subirá avances del desarrollo de su algoritmo.
- Durante la semana 6, se deberá tener implementado los algoritmos a utilizar por cada miembro dentro del repositorio de Github, solo para encontrar el camino más corto hacia un punto con obstáculos. Además, se deberá identificar el espacio de búsqueda para cada algoritmo. También, se definirá los diferentes tamaños de entrada de datos y definir las métricas que usarán para evaluar la eficiencia de los algoritmos a usar.
- Para la parte métricas se deberá experimentar con las mismas condiciones para los diferentes algoritmos a utilizar, con un mínimo de 5 experimentos para cada uno.
- Se terminará de implementar todas las reglas del juego Quoridor para cumplir con la rúbrica del trabajo.

EXPERIMENTOS Y RESULTADOS

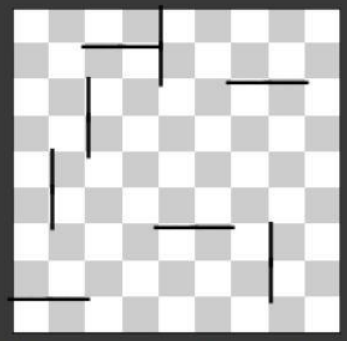
El algoritmo 1:
Se encuentra el camino más corto gracias al algoritmo BFS, que intenta visitar todo el tablero hasta encontrar el primer espacio del otro lado. Después, se agarra las posiciones de la posición final y sacas sus raíces padres. Y así, guardar todas las posiciones en un arreglo y utilizarlos.

```
def camino(arr, root, x, y):
    if root[x][y] == None:
        return arr
    else:
        x, y = root[x][y]
        camino(arr, root, x, y)
        aux = [x, y]
        arr.append(aux)
        return arr

    return arr
```

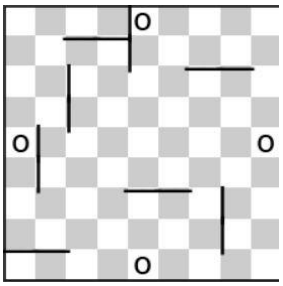
```python
def Algoritmo1(posx, posy, g):
    n = 9

    visitado = [[False for i in range(n)] for j in range(n)]
    aux = []

    dx = [1, -1, 0, 0]
    dy = [0, 0, 1, -1]

    def valid(x, y, dx, dy, g):
        if (x >= 0) and (x < n) and (y >= 0) and (y < n) and (g[x][y] != 2) and (not visitado[x][y]):
            if dx == 1:
                if lineas_h[y] != x - 1:
                    return True
                else:
                    return False
            if dy == 1:
                if lineas_v[x] != y - 1:
                    return True
                else:
                    return False
            if dx == -1:
                if lineas_h[y] != x:
                    return True
                else:
                    return False
            if dy == -1:
                if lineas_v[x] != y:
                    return True
                else:
                    return False
        else:
            return False


    def BFS(fila, columna, g):
        aux.append((fila, columna))
        visitado[fila][columna] = True
        distancia = [[False for i in range(n)] for j in range(n)]
        root = [[None for i in range(n)] for j in range(n)]

        distancia[fila][columna] = 0

        while len(aux) > 0:
            x, y = aux.pop(0)

            for i in range(4):
                nx = x + dx[i]
                ny = y + dy[i]
                if valid(nx, ny, dx[i], dy[i], g):
                    visitado[nx][ny] = True
                    aux.append((nx, ny))
                    root[nx][ny] = x, y
                    distancia[nx][ny] = distancia[x][y] + 1
                    if g[nx][ny] == 1:
                        return root, nx, ny

    raiz, x, y = BFS(posx, posy, g)
    return raiz, x, y
```

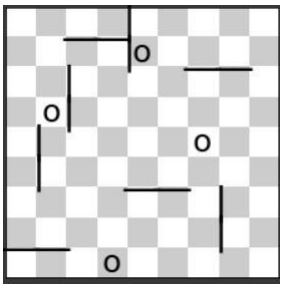
```
root1, x1, y1 = Algoritmo1(0, 4, g1)
arr = camino(arr, root1, x1, y1)
```

```
Numero de barreras que quieres del 1 al 8:
8
```



```
Elige cuantos jugadores quieres del 1 al 4:
4
Duration: 0:00:00.000171
Duration: 0:00:00.000207
Duration: 0:00:00.000262
Duration: 0:00:00.000189
```





Ganador bot top

```python
def muros(pos_x, pos_y, g):
    direccion = 0
    ruta = []
    ruta = Algoritmo1(pos_x, pos_y, g)
    x1 = ruta[1][0]
    y1 = ruta[1][1]
    x2 = 0
    y2 = 0
    direc, direc2 = x1 - pos_x, y1 - pos_y
    if direc == 0:
        if direc2 == 1:
            direccion = 1
        else:
            direccion = 3
    else:
        if direc == 1:
            direccion = 2
        else:
            direccion = 4

    if direccion == 2:
        num = rd.randint(1, 2)
        if num == 1: x2, y2 = x1, y1 + 1
        else: x2, y2 = x1, y1 - 1
    if direccion == 4:
        num = rd.randint(1, 2)
        if num == 1: x2, y2 = x1, y1 - 1
        else: x2, y2 = x1, y1 + 1
    if direccion == 1:
        num = rd.randint(1, 2)
        if num == 1: x2, y2 = x1 + 1, y1
        else: x2, y2 = x1 - 1, y1
    if direccion == 3:
        num = rd.randint(1, 2)
        if num == 1: x2, y2 = x1 - 1, y1
        else: x2, y2 = x1 + 1, y1

    if x2 >= 9 or x2 < 0 or y2 >= 9 or y2 < 0:
        return False

    for i in range(len(pos_muros)):
        if (pos_muros[i][0] == x1 and pos_muros[i][1] == y1 and pos_muros[i][2] == direccion) or (pos_muros[i][0] == x2 and pos_muros[i][1] == y2 and pos_muros[i][2] == direccion):
            return False

    pos_muros.append((x1, y1, direccion))
    pos_muros.append((x2, y2, direccion))
```

```python
301
302     pos_muros.append((x1, y1, direccion))
303     pos_muros.append((x2, y2, direccion))
304
305     if not Conexo(pos_x, pos_y, g):
306         pos_muros.pop(-1)
307         pos_muros.pop(-1)
308         return False
309
310     return True
311
```

Estos son los resultados de la duración de 4 bots con el mismo algoritmo, pero en diferente posición: arriba, abajo, derecha e izquierda. En un tablero con 8 barreras posicionadas de manera al azar.

Los datos de entrada de este algoritmo son: número de barreras del 1 al 8 y número de jugadores del 1 al 4.

El espacio de búsqueda depende del tablero que salga. Sin embargo, sería máximo 81 veces que el algoritmo busca un camino con BFS si no encuentra alguno hasta una posición del lado contrario.

En la imagen se puede ver la duración que encuentra un camino cada bot.

Además, se logró implementar la lógica de colocación de muros de manera automática por parte de los bots.

Dijkstra:

Se utilizó el algoritmo Dijkstra para resolver el problema de camino más corto. Si bien el algoritmo no se llegó a implementar con el uso del tablero, es capaz de encontrar el camino más corto para una matriz de adyacencia.

```python
import sys
import time

class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    def printSolution(self, dist):
        print ("Distancias desde origen")
        for node in range(self.V):
            print (node, "-", dist[node])

    def minDistance(self, dist, shortPathSet):

        min = sys.maxsize

        for v in range(self.V):
            if dist[v] < min and shortPathSet[v] == False:
                min = dist[v]
                min_index = v
        return min_index
```

```python
    def dijkstra(self, initPoint):

        dist = [sys.maxsize] * self.V
        dist[initPoint] = 0
        shortPathSet = [False] * self.V

        for _ in range(self.V):

            u = self.minDistance(dist, shortPathSet)
            shortPathSet[u] = True

            for v in range(self.V):
                if self.graph[u][v] > 0 and shortPathSet[v] == False and dist[v] > dist[u] + self.graph[u][v]:
                    dist[v] = dist[u] + self.graph[u][v]

        self.printSolution(dist)

start_time = time.time()
g = Graph(9)
g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
           [4, 0, 8, 0, 0, 0, 0, 11, 0],
           [0, 8, 0, 7, 0, 4, 0, 0, 2],
           [0, 0, 7, 0, 9, 14, 0, 0, 0],
           [0, 0, 0, 9, 0, 10, 0, 0, 0],
           [0, 0, 4, 14, 10, 0, 2, 0, 0],
           [0, 0, 0, 0, 0, 2, 0, 1, 6],
           [8, 11, 0, 0, 0, 0, 1, 0, 7],
           [0, 0, 2, 0, 0, 0, 6, 7, 0]];

g.dijkstra(0)
print("Tiempo de ejecución:  %s segundos" % (time.time() - start_time))
```

```
Distancias desde origen
0 - 0
1 - 4
2 - 10
3 - 3
4 - 9
5 - 6
6 - 4
7 - 5
8 - 10
Tiempo de ejecución:  0.011934041976928711 segundos
```

A*

Se utilizó el A* porque es uno de los mejores algoritmos cuando se trata de buscar caminos mínimos y rápidos. Lamentablemente este código no se pudo implementar con el tablero del juego, pero se busca el camino más rápido.

```python
import time
class Node():

    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position

        self.g = 0
        self.h = 0
        self.f = 0

    def __eq__(self, other):
        return self.position == other.position


def astar(table, start, end):

    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0

    open_list = []
    closed_list = []

    open_list.append(start_node)

    while len(open_list) > 0:

        # Tener nodo que es esta usando
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index
```

```
open list.pop(current index)
closed list.append(current node)


if cur rent_node == end_node :

    current = current_uode

        path.apperd(current.position)

    returu path[::—i] # Returns el camimo de reverse


children = [ ]
for new position in [(8,—1), (8, 1), (—1, 8), (1, 8), ( 1,—1), (—1, 1), (1,—1), (1, 1)]: # Adjacent squares


    nodegosition = (current_node.position[0] + newgosition[0], current_node.position[1} * new_position[1])


    if node position[0j > (ten(table) 1) or uode position[0] < 0 or uode position[1] > (len(table[lem(table)-1)) -1) or node position[1] < 0:


    # No hay Carrera?
    if table[node position[0]][node positional]] != 0:



    mew node = Node(curreut mode, uode position)


    cfiildreu.appemQ(uew no0e)



    for child in children:

        # Esta en la closed list
        for closed_child in closed_list:
            if child == closed_child:
                continue

        S Valores para g, h, f
        child.g = current_node.gi 1
        child.h = ((child.position[B]   end_uode.position[0]) ** z) + ((child.position[1]   end_node.position[i]) '* 2)
        child.f = child.g   child.h

        O Esta en open list
        for open_node in open_list:
            if child == open node and child.g > open node.g:
                continue


        open list.append(child)
```

```
start_time = time.time()
def main():

    table = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 1, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 1, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 1, 1, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 1, 0, 0, 0, 1, 0, 1, 0, 0],
            [0, 0, 0, 0, 1, 0, 1, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 1, 0, 1, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

    start = (0, 3)
    end = (8, 9)

    path = astar(table, start, end)
    print(path)
    print("--- %s seconds ---" % (time.time() - start_time))

if __name__ == '__main__':
    main()
```

```
[(0, 3), (1, 2), (2, 1), (3, 1), (4, 2), (5, 3), (5, 4), (6, 5), (7, 5), (8, 6), (8, 7), (9, 8), (8, 9)]
--- 0.091913223266601562 seconds ---
```

CONCLUSIONES

Se concluye que mediante las pruebas de los 3 algoritmos se puede buscar el camino más corto para que se llegue de un punto al otro. Se presentó un pequeño problema a la hora de la presentación en tablero de dos algoritmos, pero si buscan el camino más corto para llegar de una posición a otra.

Los algoritmos también se programaron en Python y se utilizaron los diferentes métodos usados en clase y se investigó sobre las partes del código que no habían sido enseñadas para una mejor implementación, funcionamiento y tener un código sin errores.

Si bien sólo se pudo experimentar con el algoritmo de BFS con el uso del tablero, sabemos que por información encontrada en línea que para este tipo de grafos donde todos tienen un mismo valor, el algoritmo BFS es igual de eficiente que Dijkstra. Tal y como se puede apreciar en la página web de Stanford[1], Dijkstra es mejor en los casos donde se tienen en cuenta otros factores como, por ejemplo, el peso que tiene pasar por cada uno de los recuadros.

Se encontraron problemas al tratar de implementar al 100% el algoritmo de A* por lo que se decidió cambiar dicho algoritmo a otro con el que estemos más familiarizados.

Además, se encontraron dificultades al intentar implementar los muros y las diferentes reglas necesarias para que el juego esté completo, tales como, la implementación de los muros en los 3 algoritmos, si bien se pudo implementar para los algoritmos BFS y DFS, no se logró realizarlo en el algoritmo de Dijkstra, también, se presentaron dificultados en la validación de los saltos de casillas cuando los jugadores se encontraban en casillas contiguas. Con un poco más de tiempo se podría lograr cubrir al 100% las reglas del juego.

[1] https://cs.stanford.edu/people/abisee/tutorial/dijkstra.html

BIBLIOGRAFÍA

Autere, A. (2005, noviembre). *EXTENSIONS AND APPLICATIONS OF THE A* ALGORITHM*. Helsinki University of Technology Laboratory for Theoretical Computer Science. http://lib.tkk.fi/Diss/2005/isbn9512279487/isbn9512279487.pdf

Sharma, S, & Pal, B (2015, 7 julio). Shortest Path Searching for Road Network using A* Algorithm. International Journal of Computer Science and Mobile Computing. https://ijcsmc.com/docs/papers/July2015/V4I7201599a23.pdf

Liu, X, & Gong, D (2011, abril). *A Comparative Study of A-star Algorithms for Search and rescue in Perfect Maze*. Research Gate. https://www.researchgate.net/publication/238009053_A_comparative_study_of_A-star_algorithms_for_search_and_rescue_in_perfect_maze

Red Blob Games: Introduction to A*. (2014, 26 mayo). Red Blob Games. Recuperado de: https://www.redblobgames.com/pathfinding/a-star/introduction.html [9/19/2020]

SAILOR Tutorial: Graph Search Algorithms. (s. f.). https://www.stanford.edu/. Recuperado 2 de octubre de 2020, de https://cs.stanford.edu/people/abisee/tutorial/dijkstra.html