PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
Dirección de Investigación e Innovación
Programa IPre de Investigación en Pregrado

Ficha de información y autorización de informe[1]

Nombre alumno		**Nombre Mentor Responsable**	
Martín Alamos		Jorge Andres Baier	
Mail alumno		**Mail Mentor Responsable**	
malamos2@uc.cl		jabaier@ing.puc.cl	
Datos del curso			
Sigla		**Sección**	
IIC2613		1	
Periodo inscripción			
Año		**Semestre**	
2019		1	
Título investigación			
Regla de decisión para el problema de puzzle de 15			
Título del proyecto			
Regla de decisión para el problema de puzzle de 15			
Fecha inicio Investigación		**Fecha entrega Informe**	
1 de Agosto 2019		**7 de Septiembre 2020**	
¿Participó más de un alumno IPre en el desarrollo de esta investigación?[1]			
SI		**NO**	**X**
Los autores permiten la *eventual* publicación del material contenido en este informe en la Revista I3 de Investigación en pregrado de la Escuela de Ingeniería UC (i3.investigacion.ing.uc.cl).			
SI	**X**	**NO**	

Yo, **JORGE ANDRÉS BAIER**, con fecha **7 de Septiembre de 2020**, declaro que he leído y aprobado la información contenida en el siguiente documento.

Firma Mentor Principal

[1] Imprima y firme esta página. Suba una copia **por separado** escaneada en PDF al sitio web http://forms.investigacion.ing.uc.cl/index.php/191715?lang=es

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
Dirección de Investigación e Innovación
Programa IPre de Investigación en Pregrado

Regla de decisión para el puzzle de 15

Martin Alamos (malamos2@uc.cl)

Departamento de Computación, Ingeniería Civil Universidad Católica. 2020

Jorge Andres Baier (jabaier@ing.puc.cl)

Departamento de Computación, Ingeniería Civil Universidad Católica.

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
Dirección de Investigación e Innovación
Programa IPre de Investigación en Pregrado

Resumen

El puzzle de 15 es un rompecabezas deslizante que consiste de un tablero de 4x4 con fichas cuadradas numeradas en orden aleatorio y una ficha faltante. La ficha faltante permite al jugador mover las fichas adyacentes dejando un nuevo espacio libre. El objetivo es ordenar las piezas en la menor cantidad de movimientos.

Para resolver el problema existen diversos algoritmos computacionales. Para efectos de esta investigación se trabaja respecto a los más conocidos *A-Estrella(A*)* e *IDA-Estrella(IDA*)*. La investigación busca responder la pregunta ¿Existirá una variante de estos algoritmos que apoyado de inteligencia artificial pueda resolver el problema con mejores resultados en cuanto a la optimalidad, tiempo y/o memoria?

La investigación consiste en entrenar una red neuronal con soluciones del puzle para luego utilizarla como apoyo de los algoritmos de estudio y así hallar mejores resultados. Considerando un mejor resultado todo aquel que puede alcanzar la misma optimalidad (largo de la solución) de un problema con menos recursos (tiempo y/o memoria).

Finalmente se comparan los resultados con los algoritmos existentes de estudio en base a 76 problemas. Los resultados son positivos, se concluye que para una sub-optimalidad dada, un algoritmo similar a *A** con algunas modificaciones y apoyado de redes neuronales es capaz de encontrar mejores soluciones utilizando menos memoria.

Palabras claves*:* Puzzle de 15, Inteligencia Artificial, Heurística, Redes Neuronales, A-estrella.

1. Introducción

El puzzle de 15 es un rompecabezas deslizante que consiste de un tablero de 4x4 con fichas cuadradas numeradas en orden aleatorio y una ficha faltante. La ficha faltante permite al usuario mover las fichas adyacentes dejando un nuevo espacio libre. El objetivo es ordenar las piezas en la menor cantidad de movimientos.

En la actualidad existen algoritmos muy populares como ***A-ESTRELLA*** *(A*)* e ***IDA-ESTRELLA*** *(IDA*)* que se utilizan para resolver un sin fin de problemas, uno de ellos, el puzle de 15. Originalmente, *A** fue diseñado como un algoritmo genérico para recorrer grafos transversalmente [5]. Su uso más común es la búsqueda de la ruta más óptima (*pathfinding*), que aplica tanto a mapas de vida real como en videojuegos. Otras aplicaciones de este algoritmo son el TSP[1], búsqueda de caminos en tiempo real[2], navegación sobre grafos y juegos de puzzle[3].

Como se presenta en distintos artículos de hace mucho tiempo, por ejemplo, "Generalized best-first search strategies and the optimality of A*" de Dechter. 1985 [4], A* es un algoritmo que encuentra siempre la solución óptima a un problema con un costo alto de memoria ya que debe almacenar todos los estados visitados

En la presente investigación se busca mejorar el rendimiento de A* con el uso de una red neuronal, creando un nuevo algoritmo mejor en cuanto a tiempo de ejecución y memoria. Para el desarrollo de dicho algoritmo se utiliza el puzle de 15 como un ejemplo de los problemas que se pueden resolver. El alcance de la investigación tiene impacto en la resolución de problemas de grafos y árboles de decisión, aportando un avance en el uso de la inteligencia artificial para la resolución de estos problemas.

2. Experimentación

En primer lugar se utilizó el algoritmo IDA-estrella para encontrar las soluciones óptimas a 30.367 problemas distintos creados al azar. Dichas soluciones se escribieron en un archivo indicando el estado del tablero en cada instante y el movimiento necesario para seguir el camino óptimo a la solución. Se escribió un total de 1.510.673 estados del tablero distintos con sus respectivos movimientos. El 90% de los estados se utilizó para entrenar una red neuronal que recibía un estado del tablero como *input* y como *output* retornaba una probabilidad sobre el mejor movimiento a realizar (izquierda, abajo, arriba o derecha). El 10% restante se utilizó para evaluar el rendimiento de la red, validando así su funcionamiento. A continuación se despliega el flujo de entrenamiento hasta predicción.

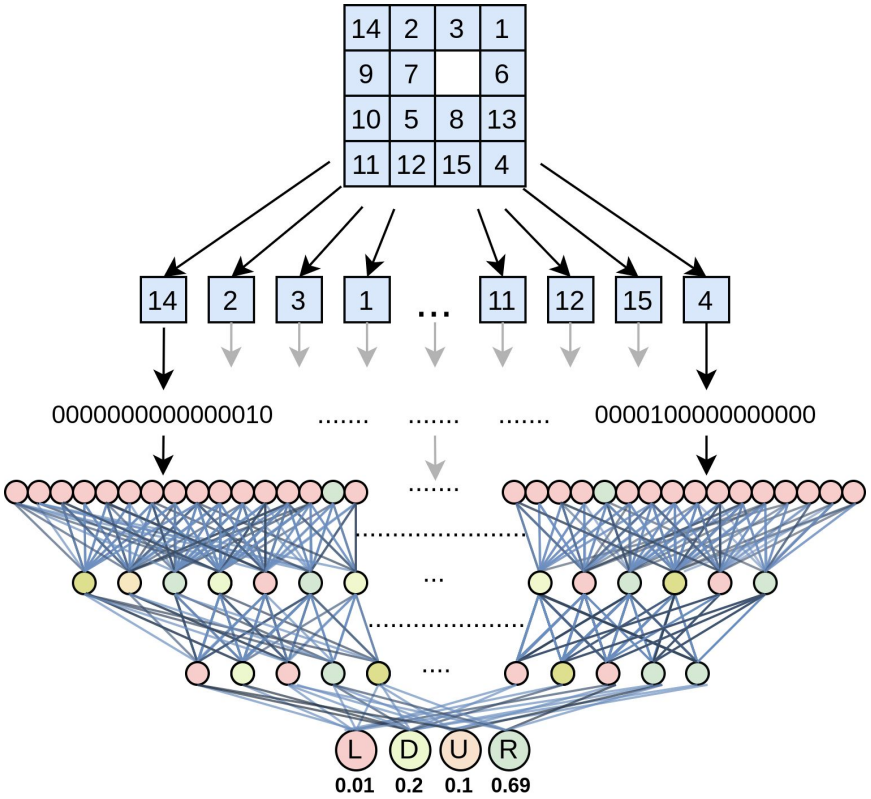

Figura 1.- Representación del flujo de entrenamiento de la red neuronal.

La red neuronal constaba de 256 (16x16) neuronas de *input* (16 por cada cuadrante del tablero), 2 capas intermedias de (64 y 16 neuronas), y luego una última capa de de *output* de 4 neuronas. Esta última indicaba el movimiento recomendado por la red neuronal (izquierda, abajo, derecha o arriba).

Una vez entrenada la red neuronal se llevó a cabo la búsqueda de un algoritmo mixto que combinaba A-estrella y las recomendaciones sugeridas por la red neuronal entrenada. Dicho algoritmo queda abierto a cualquier otra implementación. En primer lugar, el algoritmo desarrollado priorizaba las recomendaciones de la red neuronal, y en base a pruebas con ejemplos aleatorios, se aumentó o disminuyó dichas preferencias según los resultados. En el anexo 2 se puede encontrar un pseudocódigo de dicho algoritmo mixto.

Determinada la estructura definitiva del algoritmo mixto se procedió a hacer las pruebas finales, que consistían en evaluar el nuevo algoritmo respecto de los algoritmos de estudio en base a 76 problemas de *benchmark* entregados por el profesor a cargo. Dado que el algoritmo creado ocupa una red neuronal como base de apoyo, este no puede garantizar que la solución es la más óptima ya que la red neuronal hace inferencias de acuerdo a lo aprendido y no se puede dar certeza de las conclusiones. Dicho lo anterior, el nuevo algoritmo no es comparable con los algoritmos A* e IDA*, dado que estos si se enfocan completamente en la solución más óptima (corta), por lo tanto, se procede a utilizar los algoritmos similares W-A* y W-IDA*, qué son las modificaciones más comunes de los algoritmos originales para buscar soluciones sub óptimas[6].

El trabajo realizado y eficiencia del algoritmo se puede medir según la cantidad de nodos que expande y visita. Cada expansión significa procesar el estado, calcular la heurística y luego nuevas expansión hasta llegar al objetivo. El objetivo del experimento es reducir el número de expansiones para una optimalidad dada.

3. Resultados y discusión

El primer resultado obtenido fue la creación de la red neuronal. Para la creación de dicha red neuronal se utilizaron 1.360.673 ejemplos de estados del tablero y su respectivo movimiento óptimo en la fase de entrenamiento y luego 150.000 para validar el funcionamiento.

En el siguiente gráfico se muestra como fue el rendimiento de la red neuronal a medida que se fueron sumando ejemplos en el entrenamiento.

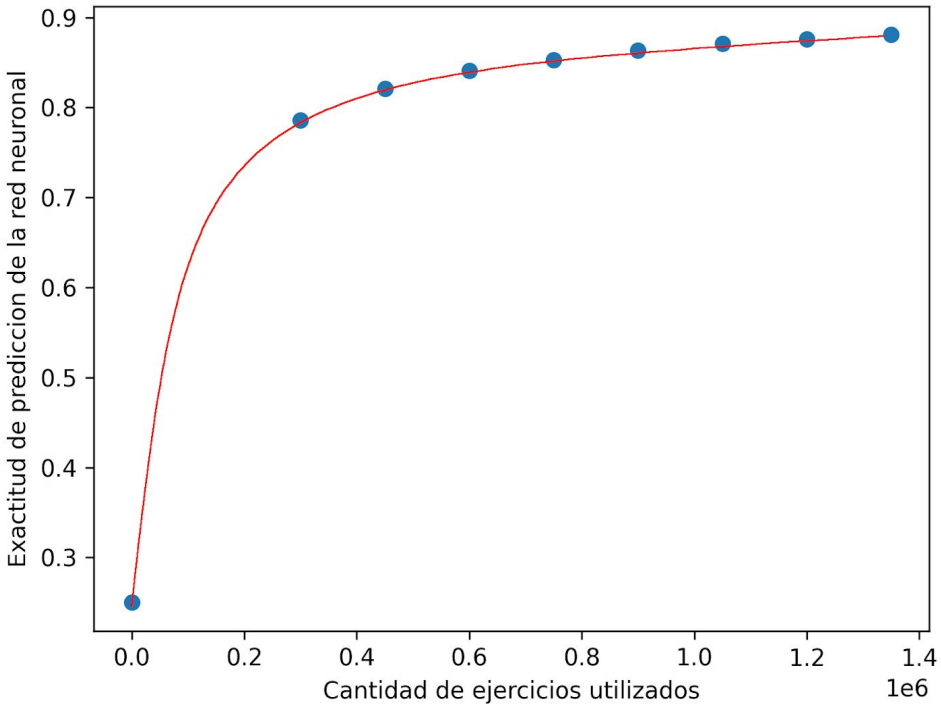

Figura 2.- Aprendizaje de la red neuronal a medida que se aumentan los casos de estudio.

Se determinó que 1.360.673 ejemplos es un número suficiente de entrenamiento debido a la curva de aprendizaje. Cabe destacar que a mayor casos de ejemplo mejor será el desempeño por lo que es una decisión que se puede profundizar.

Una vez implementada la red neuronal se procedió a crear un algoritmo mixto que combina A* y las predicciones de la red neuronal, llamamos a este algoritmo *NN*. Finalmente, con el *NN* funcional se hicieron pruebas en base a 76 ejemplos. El resultado se comparó con los algoritmos WIDA*, donde *W* es una variable entre 1 e infinito que hace referencia al peso (*weight*) de la heurística. A medida que crece el peso (*w)* las soluciones son menos óptimas a cambio de utilizar menos recursos (tiempo y memoria). A continuación se despliega el desempeño de WIDA-*w* con distintos pesos (*w*) y *NN*.

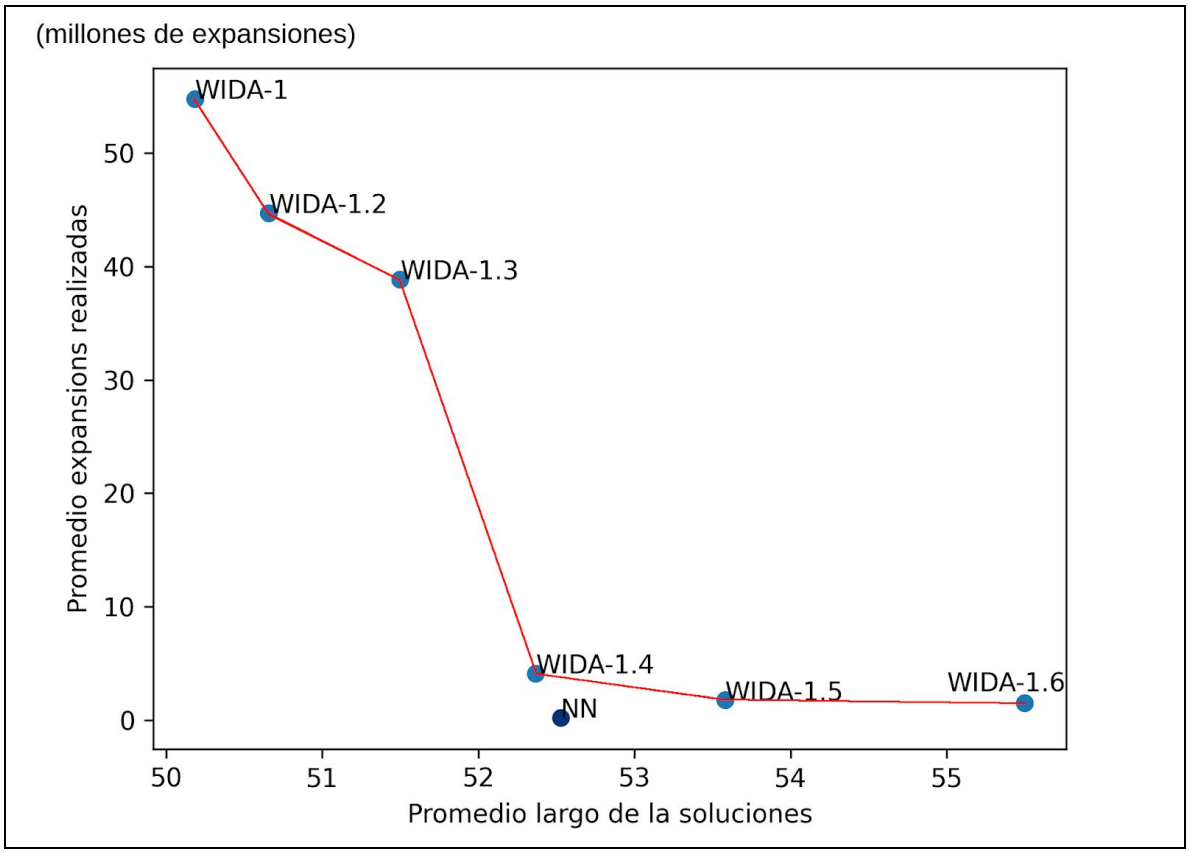

Figura 3.- Comparación del costo de procesar soluciones para el algoritmo *W-IDA* y *NN*

Como se puede apreciar en la Figura 3, el algoritmo creado (*NN*) encuentra soluciones de largo promedio 52,5 utilizando tan solo 0,18 millones de expansiones, mientras que WIDA-1.5 encuentra soluciones de largo mayor (53,6) utilizando 1,7 millones de expansiones (más de 9 veces el número de expansiones realizadas por *NN*). Para más detalle del gráfico se puede apreciar la información en el tabla del anexo 1.

4. Conclusiones

El algoritmo mixto creado no da al 100% con lo que se buscaba en un origen ya que se sacrificó la optimalidad de la solución y por alcances de la investigación no se pudo evaluar la variable tiempo en la comparación de resultados. Esto se debe a que la red neuronal corre en código Python y utiliza librerías externas que hacen el proceso de recomendación mucho más lento.

Lo que sí se logró, es que encontró con un algoritmo apoyado de redes neuronales que dada una optimalidad deseada, encuentra soluciones que realiza menos expansiones y por ende ocupan menos memoria y procesamiento que el conocido algoritmo WIDA*, respondiendo afirmativamente nuestra hipótesis. Estos resultados son de gran importancia dado que se puede extrapolar a muchos otros problemas de grafos y árboles de decisión, además que la solución que se encontró utilizaron ciertas variables que dan para profundizar.

La investigación actual abrió dos otras posibles investigaciones, en primer lugar es el estudio de las variables del nuevo algoritmo. Las variables en las que se puede profundizar son a grandes rasgos dos, el entrenamiento de la red neuronal y el algoritmo mixto creado.

En segundo lugar se abre la opción de investigar respecto de la implementación de este nuevo algoritmo en otros problemas de grafos y árboles de decisión.

Glosario

A-ESTRELLA: Algoritmo de inteligencia artificial para encontrar soluciones óptimas en un grafo en base a búsqueda BFS.

IDA-ESTRELLA: Algoritmo de inteligencia artificial para encontrar soluciones óptimas en un grafo en base a búsqueda DFS.

Referencias

[1] O. Vinyals, M. Fortunato, and N. Jaitly: Pointer networks, in Advances in Neural Information Processing Systems. pp. 2692–2700. (2015).

[2] F. Muñoz, M. Fadic, C. Hernández, and J. A. Baier: A neural network for decision making in real-time heuristic search. in SOCS 2018. (2018).

[3] A. Graves, G. Wayne et al.: Hybrid computing using a neural network with dynamic external memory. Nature, vol. 538, no. 7626, p. 471. (2016).

[4] Dechter, Rina; Judea Pearl: Generalized best-first search strategies and the optimality of A*. Journal of the ACM. 32 (3): 505–536. doi:10.1145/3828.3830. S2CID 2092415. (1985).

[5] Peter E. Hart, Nils J. Nilsson, Bertram Raphael: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Trans. Syst. Sci. Cybern. 4(2): 100-107. (1968).

[6] Ira Pohl: Heuristic Search Viewed as Path Finding in a Graph. Artif. Intell. 1(3): 193-204. (1970).

Anexos

1. Tabla de datos

Tabla 1.- Desempeño promedio de los algoritmos

Algoritmo	**Promedio largo de las soluciones por problema**	**Promedio de expansiones realizado por problema (en millones)**
NN	52,52	0,18
WIDA-1	50,18	54,76
WIDA-1.2	50,67	44,69
WIDA-1.3	51,50	38,84
WIDA-1.4	52,37	4,11
WIDA-1.5	53,58	1,74
WIDA-1.6	55,50	1,49

2. Pseudocódigo del algoritmo mixto creado

```python
def searchSolution():
    goal = Node(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
    neuronal_network = NeuronalNetwork("nn.h5")  # importa red neuronal

    # binary heap de nodos ordenados por valor f
    preferred_heap = BinaryHeap() # binary heap de nodos con preferencia según la red neuronal
    normal_heap = BinaryHeap()  # binary heap de nodos sin preferencia según la red neuronal

    # se crea el nodo del estado inicial, por ejemplo:
    initial_node = Node(8, 14, 2, 3, 4, 11, 15, 0, 1, 9, 10, 12, 13, 5, 6, 7)

    # se calcula la heuristica desde el punto inicial
    heuristic = calculateHeuristic(initial_node)
    initial_node.g = 0
    initial_node.f = initial_node.g + heuristic # valor de f siempre es igual a g + h

    states_visited = []  # lista de estados ya visitados
    normal_heap.insert(initial_node)

    current = 0
    # mientras exista de donde sacar un nodo
    while not (normal_heap.isEmpty() and preferred_heap.isEmpty()):
        # iteramos 1 a 1 de entre los heaps para sacar un nodo
        if current == 0:
            heap = normal_heap
        else:
            heap = preferred_heap
        current = (current + 1) % 2  # alternamos current entre 0 y 1
        node = heap.extract()  # retiramos el nodo con menor valor f
        if not node:
            continue  # volver al while
        if node == goal:
            return node  # retornamos el objetivo y salimos del while

        # obtenemos los estados posibles con un movimiento
        node_successors = node.getSuccessors()
        for successor in node_successors:

            is_new = successor in states_visited  # primera vez que visita el estado?
            is_shorter_path = False
            if is_new:
                successor.g = node.g + 1
                states_visited.insert(successor)
            elif successor.g < node.g + 1: # encontramos una ruta más barata para un estado?
                is_shorter_path = True

            if is_new or is_shorter_path:
                successor.f = successor.g + calculateHeuristic(successor)
                # si no hay hijo del nodo o encontramos un mejor hijo
                if (not node.child or successor.f < node.child.f):
                    node.child = successor  # dejamos registro del movimiento
                prediction_posibility = neuronal_network.predictNode(node, successor)
                if prediction_posibility > 0.5:
                    # inserta y ordena el binaryHeap segun valor f = g + h
                    preferred_heap.insert(successor)
                elif prediction_posibility > 0.1:
                    # inserta y ordena el binaryHeap segun valor f = g + h
                    normal_heap.insert(successor)

    return initial_node
```