‘2048 Solution’: Algoritmos eficientes para la resolución del juego 2048

Beltracchi Rodrigo Oscar, Dahl Juan Ricardo, Rizzalli Ayelén Analía.

Universidad Nacional del Centro de la Provincia de Buenos Aires

Resumen. “2048 Solution” fue desarrollado como proyecto final correspondiente a una materia de una carrera de Informática dictada en el segundo año de la misma. La materia aborda conceptos de análisis y diseño de algoritmos, en especial técnicas algorítmicas para resolver problemas de mediana escala. El objetivo de este proyecto fue implementar soluciones eficientes para el juego “2048”, haciendo énfasis en la aplicación de técnicas de diseño, analizando los comportamientos para cada solución, la complejidad temporal de los algoritmos implementados y el análisis empírico del tiempo de ejecución para cada una soluciones propuestas.
“2048 Solution” ha sido abordado desde las técnicas de “Backtracking”, “Búsquedas Heurísticas” y “Branch and bound”. Se implementó una interfaz gráfica que permite una visualización similar a la disponible para los dispositivos móviles, que permite ejecutar los distintos algoritmos y alcanzar resultados hasta la potencia de dos ´8192´, así como brindar al usuario la posibilidad de resolver el juego por su cuenta.

1 Introducción

El juego 2048 fue creado por un joven italiano llamado Gabriele Cirulli, quien se inspiró en dos juegos ya existentes: 1024! y Threes . El objetivo del juego es llegar a la potencia de dos más grande posible (2048 o incluso mayor). Para ello, se dispone de un tablero de 4x4 en el que se pueden realizar como máximo cuatro movimientos posibles (derecha, izquierda, arriba, abajo). Dependiendo el movimiento, si dos casillas son adyacentes e iguales, se combinarán en una única baldosa con su respectiva suma. Cabe aclarar que, en los movimientos ´derecha´ e ´izquierda´ se juntarán solo las casillas adyacentes respecto de las filas del tablero y, en los movimientos arriba y abajo harán lo propio con respecto a las columnas.

Además, cada vez que se realice un nuevo movimiento, y el tablero ya esté modificado con las colisiones correspondientes, aparece una ficha en una posición aleatoria, cuyo valor puede ser de 2 o 4. Se ubicará en cualquiera de los espacios libres del tablero y pasa a formar parte del mismo, pudiendo ser usada para futuras colisiones.

El juego finaliza de forma afortunada cuando una casilla contiene el valor 2048 - de ahí surge su nombre- pero puede continuar en busca de una potencia mayor. No obstante, si un jugador queda sin algún movimiento legal (es decir, ya no quedan

espacios vacíos y no existen baldosas adyacentes con el mismo valor), el juego termina.

A pesar de su fácil formulación es un problema duro de resolver. Como muchas soluciones para juegos, esta no es trivial por las distintas estrategias que se pueden plantear para su solución, pudiendo transformarse en un problema intratable. Es importante destacar que uno de los principales objetivos de la materia incluye aplicar buenas prácticas de programación aplicando técnicas de diseño, ejercitando en algunos casos con la resolución de distintos tipos de juegos, usando un enfoque metodológico de desarrollo no solo en los juegos sino en distintas aplicaciones tradicionales.

En las siguientes secciones se profundizará acerca de las características del juego, describiendo el desafío computacional que implica, haciendo un análisis detallado de los algoritmos implementados, para luego abordar con el diseño e implementación del proyecto, las soluciones encontradas y las conclusiones que se desprenden de su realización. El Anexo A contiene un diagrama de interacción entre clases/métodos más relevantes de la implementación y el Anexo B contiene el código de los algoritmos principales para resolver el juego.

2 Diseño de la solución. Descripción y Análisis de los Algoritmos

Como se mencionó anteriormente, el objetivo de este trabajo fue implementar distintos algoritmos basados en técnicas de diseño de algoritmos para la solución del "2048". Para llegar al diseño es necesario definir lo que significa el espacio de búsqueda particular para este juego. Luego, se hace un análisis exhaustivo de las distintas estrategias dentro del marco de las técnicas utilizadas para la implementación. A continuación se detallan cada una de ellas.

2.1 El espacio de búsqueda

Un aspecto común estudiado en cada una de las técnicas implementadas es la organización de los datos de forma tal que sea posible lograr disponer todos los escenarios del juego en una única estructura. Luego, cada algoritmo tendrá la tarea de realizar un recorrido eficiente en busca de la solución. De este modo, el espacio de búsqueda se describió como un árbol cuya raíz es el tablero inicial. Cada nodo del árbol posee como máximo cuatro hijos (movimientos posibles). Desarrollando cada uno de los nodos con sus respectivos hijos se genera un árbol que contiene todas las combinaciones posibles del juego. La Figura 1 ilustra este espacio de forma parcial.

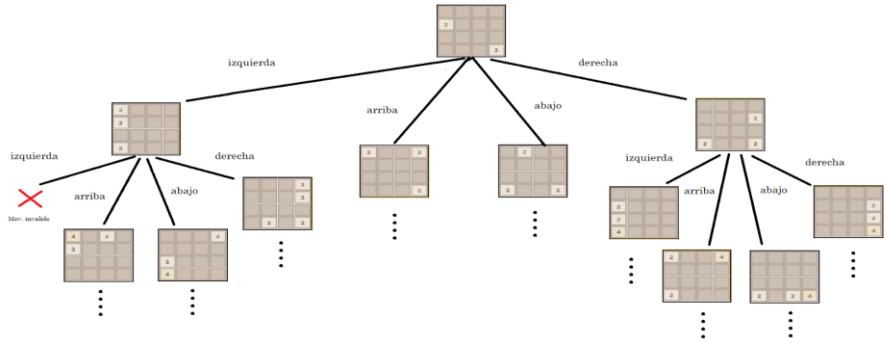


Fig.1. Espacio parcial de búsqueda para '2048'

Un ejemplo para el árbol es un nodo que se genera a partir de la ejecución de un movimiento hacia arriba, como muestra la figura 2.

```
#include "marriba.h"

MArriba::MArriba(Tablero* t): Movimiento (t){
setNombre(10);
}

void MArriba::mover(){
    cam= false;
    Ficha* aux;
    Ficha* actual;
    int k=00;
    for (int i=0; i<4;i++){
        k= 0+i;
        aux= juego->getFicha(k);
        for (int j=10; j<40 ; j+=10){
            actual = juego->getFicha(j+i);
            if (!actual->esVacia() && !actual->seUso()){
                unir (actual, aux);
                if (actual->esVacia()){
                juego->borrarFicha(i+j);
                }
                aux=actual;
                k= i+j;
            }
        }
    }
    correr();
}
void MArriba::correr(){
    int aux=0;
    for (int i=0; i<4;i++){
        for(int j=00; j<40; j+=10){
            if (!juego->esVacio(i,j)){
                if (aux!=j){
                   juego->cargarFicha(juego->getFicha(i+j),aux+i);
                   juego->borrarFicha(i+j);
                   cam=true;
                 }
                aux+=10;}

            }
        aux=0;
        }
    addNew();
}
```

Fig.2. Métodos del movimiento "arriba".

2.2. Backtracking

'Backtracking' o 'vuelta atrás' es una técnica usada para resolver problemas cuya solución involucra el recorrido de un gran espacio de búsqueda [5] que esta determinado por el pseudo-código de la figura 3. La idea es, sistemáticamente, ir explorando y eliminando posibilidades. El espacio de búsqueda en este tipo de circunstancias se modela en forma de árbol evitando así, posibles repeticiones y ciclos. Determinar si un nodo es solución, diseñar una poda acorde y determinar los hijos de cada estado son los aspectos a definir antes de utilizar dicho algoritmo. En general, la solución puede expresarse como una tupla $(x_1, x_2,\ldots, x_n)$, donde los x_i son elegidos de un conjunto finito S_i.

La poda es una función que permite averiguar si una secuencia de decisiones, la solución actual en curso, cumple las restricciones propias del problema.

Por último, determinar correctamente los hijos de cada estado es una forma sistemática y organizada de generar y recorrer el espacio de búsqueda. Para adaptar el problema a dicha estrategia, es necesario definir correctamente las características mencionadas anteriormente.

En una primera instancia, la tupla $(x_1, x_2,\ldots, x_n)$ que comprende la solución del problema está compuesta por cada tablero que nos conlleva al resultado esperado con su respectivo orden.

En lo que respecta a la función hijos, está dada siempre por cuatro nodos -sin incluir la poda- los cuales son los escenarios posibles luego de aplicar alguno de los movimientos permitidos (derecha, izquierda, arriba, abajo). Los mismos se perciben fácilmente una vez que se diseñó el espacio de búsqueda [4][7].

Habitualmente, cuando se usa la técnica de "Bactracking" es importante identificar cada uno de los hijos pero, sin importar el orden en el cual van a ser ejecutados en tiempo de ejecución. En otras palabras, no existen prioridades entre ellos e incluso, este caso pareciera pertenecer a este tipo de problemas. Sin embargo, luego de estudiar minuciosamente algunas tácticas para ganar el juego, se observó que la búsqueda del algoritmo puede mejorar o empeorar de acuerdo a la disposición de los hijos.

Una estrategia de las recién mencionadas para resolver el juego consiste en elegir una esquina tratando en todo momento de tener el número de mayor potencia en el rincón elegido. Se eligió el rincón superior izquierdo, por lo que los hijos se generan de la siguiente forma: izquierda, arriba, abajo, derecha. Repitiendo estos pasos se provoca la llamada "escalera" en la que cada casilla contiene una potencia menor de modo que al "chocar" las más pequeñas, en una secuencia de pasos aumenta la potencia mayor. Así, se debe evitar movimientos que "desordenen" dicha disposición. En este caso mover hacia abajo pondría en riesgo la partida por lo que se lo colocó en última instancia en la lista de movimientos. A su vez, la poda que caracteriza esta implementación es determinada estrictamente solo por las reglas del juego. Específicamente la poda se define por si es factible o no realizar el movimiento en cuestión.

```
int back (Tablero copia, int&elegido, int profundidad){
    if la profundidad es mayor o igual a cero{
        for el = todos posibles movimientos{
            b= copia;
            realizo movimiento sobre b;
            if no es movimiento invalido
                heu = evauluacion del tablero b en la heuristica;
                heu += valor de back en nivel siguiente (b,el,profundidad-1)
                if heu es mayor a la evaluacion anterior
                    modifico eleccion
                    valor anterior = heu
```

Fig.3. Pseudo-código de generación del espacio de búsqueda.

Esta técnica reduce el espacio de búsqueda mediante la implementación de la heurística A que evalúa los tableros generados en cada nodo, permitiendo realizar una mejor elección de la jugada a seguir y evitando también la generación de nodos sin cambio.

Este algoritmo para darle un valor a la jugada se basó en una estrategia que consiste en ponderar cada tablero nuevo generado por el algoritmo de Backtracking según algunas características detalladas a continuación, para así poder comparar diferentes estados y elegir el movimiento que genere el mejor de ellos. La heurística elegida para evaluar los tableros se puede expresar mediante la siguiente fórmula:

Heur = Puntaje + (Lugares disponibles * Ficha Alta) + Matriz de Valores.

- Puntaje: cuanto mayor sea el puntaje en un movimiento, quiere decir que juntamos una ficha alta.
- Lugares disponibles en el tablero: ya que va a resultar ser un número relativamente bajo en comparación al valor del puntaje, lo multiplicamos por el valor de la ficha más alta. Esto permite que nos ocupemos por el valor que estamos buscando formar, ya que si aumenta la ficha de mayor valor, ese tablero tendrá más peso en la elección de un movimiento,
- Suma de los valores de las fichas según la Matriz de ponderación (véase Fig. 4.) (que nos permitirá tener los mayores valores en las esquinas del tablero o en los bordes, de esta manera evitaremos que la ficha alta se rodee de valores bajos que no son directamente necesarios para aumentarla).

La matriz de ponderación, es una estrategia que implementamos, para que las fichas con grandes puntajes se sitúen en los bordes, cada ficha con su valor asociado se multiplica por el lugar en el que se encuentra, y de esa forma calculamos un valor para el tablero. Principalmente queremos tener las fichas más grandes en las esquinas o en los bordes, sí un tablero posee un 512 en el centro solo va a tener ese valor, porque se multiplica por uno, en cambio sí está situado en la esquina, se multiplica el valor por 10, o sea 5120 puntos solo por esa ficha. Luego de calcular todos los valores, los sumamos y ese es el valor asociado al tablero.

10	2	2	10
2	1	1	2
2	1	1	2
10	2	2	10

Fig.4 Matriz de Ponderación

2.3 El algoritmo A*

A* se trata de un algoritmo heurístico [4] que usa el conocimiento del dominio para adaptar el solucionador y, de esta manera, éste sea más potente y consiga llegar a la solución con mayor rapidez. Se dice que estas técnicas utilizan el conocimiento para avanzar buscando la solución al problema ya que una de sus principales características es que se hará uso de una función de evaluación heurística, mediante la cual se etiquetarán los diferentes nodos del espacio de búsqueda y servirá para determinar la probabilidad de dichos nodos de pertenecer al camino óptimo. Es de vital importancia encontrar una heurística que aproxime lo mejor posible cada situación del problema ya que dependiendo de su performance, la técnica puede desempeñarse de mejor o peor manera. Por otra parte, debe ser eficiente en términos de complejidad debido a su uso reiterado. La función de evaluación está compuesta a su vez, por la suma de otras dos funciones. Es decir:

$$f(n) = h(n) + g(n)$$

Donde h(n) es una función heurística que expresa que tan lejos se encuentra el nodo para llegar a la solución. Este conocimiento permite calcular la distancia entre el estado obtenido y el estado final. En cuanto a g(n) es una función simple que lleva la distancia desde la raíz del árbol de búsqueda hasta el estado en cuestión. La suma de h y g le otorgan el valor final de la función de evaluación. Luego, a medida que se van procesando estados se los mantiene ordenado crecientemente de modo tal que el primero de ellos será el más prometedor por el momento.

La finalidad de la técnica es realizar una búsqueda “guiada” por lo que su razón es establecer una función de evaluación que identifique lo mejor posible cada uno de los escenarios a medida que van siendo procesados, por lo que es válido pensar que el crecimiento del árbol se hace inyectando conocimiento [6]. Por este motivo, y para conocer aún mejor las ventajas y limitaciones de A*, se implementaron dos funciones heurísticas alternativas de manera tal que cada una de ellas produciría un orden de prioridad distinto y, por ende, un recorrido diferente del espacio de búsqueda seleccionado.

Este algoritmo pondera los posibles estados basados en la hipótesis (heurística B) de que a la hora de jugarlo, las mayorías de las estrategias utilizadas es combinar lo más posible e intentar utilizar la menor cantidad de casillas vacías. La relación con los pasos dados hasta el momento solo sirve para que ambos calificativos del escenario

(casillas y puntuación) adquieran en la mayoría de los casos valores cercanos. Por ejemplo, el puntaje del juego es directamente proporcional con el número de pasos, es decir a medida que aumentan los movimientos lo hace también la puntuación y por el lado de las casillas vacías siempre es un valor pequeño que ronda en el intervalo (0,14) aproximadamente. De este modo, relacionando los pasos con ambos criterios hace que siempre tengan valores cercanos y que uno de ellos no sea mucho mayor que el otro lo que quitaría mucha importancia, generando que casi ni impacte a la hora de "guiar" el recorrido. Se puede expresar mediante la siguiente fórmula:

$$h(n)=\text{casillas vacías}*\text{pasos} + (\text{Puntos} \div \text{pasos})$$

Donde casillas vacías corresponde al número de casilleros vacíos del tablero en cuestión, pasos es la cantidad de movimientos necesarios para llegar a esa situación y, por último los puntos son la puntuación que presenta dicho escenario.

2.4 Branch and Bound

Branch & Bound o ramificación y poda es una técnica que se basa en podar a partir de cotas predictivas sobre los nodos del árbol del espacio de búsqueda. La poda permite eliminar ciertas zonas del espacio de búsqueda. Un nodo en curso se poda cuando existe información del mismo que, pese a cumplir las restricciones y ser "prolongable", no conducirá a la solución resultado. Así, se logra una búsqueda informada en la que los nodos poco prometedores no se expanden [6].

La poda se basa en una función de evaluación que debe ser efectiva y "barata" en términos de eficiencia ya que, en caso contrario podría generar una búsqueda relativamente mala. De acuerdo a la situación de cada uno de los nodos, se los puede caracterizar como Nodo Vivo, Muerto o en Expansión. El nodo en expansión es aquel que más promete dentro de todos los nodos vivos, es decir, aquel que la información que brinda su cota nos dice que es el que por el momento, nos llevará a una mejor solución. Los nodos vivos se mantienen en una estructura lineal (listas, pilas, filas) o colas de prioridad (heap). El valor que el nodo vivo "promete" es la clave del ordenamiento en la cola de prioridad.

Para llegar al objetivo que persigue el juego 2048 se realizó una pequeña adaptación a la técnica ya que la misma se emplea usualmente en problemas de optimización. La modificación a la técnica fue suponer de antemano el valor de la cota solución de modo que los nodos serán catalogados como vivos o muertos de acuerdo a la comparación con dicha cota, es decir, aquellos nodos que sean menos prometedores que la cota solución no se insertarán en la lista de nodos vivos.

En el momento de la implementación, luego de diseñar el espacio de búsqueda, se definió una cota que caracterice lo mejor posible cada escenario del juego. Esta función de evaluación se basó en la siguiente fórmula:

$$\text{Cota} = \text{valor real} + \text{valor esperado}$$

El valor real de la cota ideada se obtiene de acuerdo a la mayor potencia de dos que posee el tablero. Dicha elección se basó en la hipótesis de que este número, en

cierto modo representa estimativamente el camino hecho hasta el momento y que tan lejos se ha llegado en vistas a la solución.

Se implementó una cota solución como base de comparación con los estados en curso. La misma, sigue el formato de cota ya expresado considerando sólo el valor real ya que, de este modo el valor esperado podría verse como nulo.

En lo que concierne al valor esperado, se forma de acuerdo a dos aspectos del escenario a calcular. En una primera instancia, se consideró la cantidad de casillas vacías. Para relacionarlo con el valor real, es decir, la mayor potencia del tablero; el valor esperado era el resultado de la relación matemática $2^{\text{nro casillas vacías}}$. Así, la cota nos orientaría al resultado posible de expandir el nodo en cuestión. Se consideró identificar a las casillas vacías del tablero como el aspecto que determinase el futuro del juego al expandir dicho nodo por el motivo de que mientras mayores sean los casilleros vacíos existentes en el tablero, mayores serán las probabilidades de conseguir una mejor performance.

Sin embargo, en el análisis del comportamiento de dicha cota se pudo percibir ciertas falencias a tal punto de que en muchas situaciones se podan, no sólo nodos con pocas posibilidades de ganar, sino que también muchos otros que son relativamente buenos en la situación en la que están (véase Fig. 5).

Fig.5. Escenario de un caso drástico

Siguiendo con la expresión propuesta de una cota de la técnica “Branch & Bound” en el escenario propuesto en la imagen se corresponde la casilla 1024 con el valor real, es decir, 1024 identificaría el trayecto que la solución parcial ofrece. Sumado a esto, son ocho las casillas vacías del tablero por lo que, nuestra cota predictiva espera obtener como máximo otra puntuación de 28 (256).

Viendo la cota como un único resultado, el tablero es representado por el puntaje 1280 y como consecuencia, sería podado por la cota solución. A simple vista se puede percibir que este escenario es uno de los mejores que se puede presentar en el juego antes de ser ganado y, el algoritmo diseñado lo descartaría.

Esta situación nos da pie para suponer que se debería mejorar la función evaluadora analizada hasta el momento. De este modo, se buscó complementar no solo las casillas vacías sino que se pensó también que muchas veces poder realizar combinaciones es de gran ayuda e incluso, se puede decir que de esta forma a su vez aumentando dicho número como consecuencia se disminuye las chances de perder el juego. Así, se evitan ciertos problemas como el del escenario anterior y la cota adopta

un resultado más significativo para luego comparar. La nueva cota frente a la misma situación permitiría que el algoritmo considere ese estado como prometedor y llegue a un final esperado en el juego.

3. La Implementación

Uno de los objetivos principales de la materia para la cual este trabajo fue realizado este proyecto es aplicar un estilo de programación donde se proporcione una clara separación de responsabilidades, permitiendo que la lógica de la solución opere independientemente de la visualización y de los datos. De esta manera se ha trabajado poniendo el mayor énfasis y cantidad de tiempo de trabajo en la lógica de los algoritmos, es decir en la aplicación de los conocimientos recibidos en cuanto a las técnicas de diseño para luego diseñar la visualización del juego [8].

Para comprobar los comportamientos de los diferentes algoritmos estudiados y visualizar si fueron o no los esperados se necesitó de una interfaz gráfica que permita al usuario interactuar y conocer dichos resultados. Por este motivo, se puso énfasis en elaborar una visualización sencilla tomando como guía la ofrecida por el juego primitivo.

El almacenamiento de los datos propios del tablero en la memoria es por medio de una matriz de 4x4. De esta manera se logra reproducir la situación del juego separando claramente las responsabilidades mencionadas recientemente (figura 6).

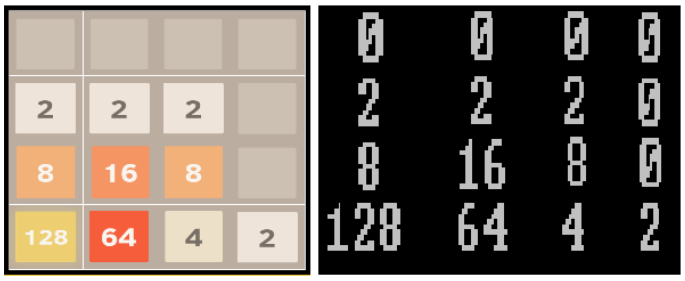


Fig.6- Relación Visual-almacenamiento interno

La aplicación fue desarrollada utilizando el entorno de desarrollo Qt Creator [1] principalmente por su compatibilidad con el lenguaje estudiado en la cátedra (C++) [2] y a su vez, por su gran variedad de bibliotecas y documentación que le permiten al programador simplificar la construcción de la interfaz.

Para crear el juego solo se implementó una única clase de tipo *Qt Designer Form: MainWindow*. Provee todos los métodos y slots necesarios para jugar el juego de modo manual como así también, para mostrar las soluciones de los distintos algoritmos ya detallados. Para llevar a cabo la aplicación se hizo uso de diferentes Buttons, Label y Widgets que ofrece Qt Creator. La base de dicha clase es representar

visualmente un determinado tablero por lo que, para ello se hizo uso de un objeto QGridLayout por medio del cual mediante imágenes se representa cada casilla del escenario (véase Fig. 6).

Mediante diferentes Buttons se produce la interacción del usuario pudiendo jugar en modo manual, elegir un algoritmo, mostrar la solución que ofrece dicho algoritmo, o volver al menú inicial en donde se eligen las variaciones que ofrece la aplicación.

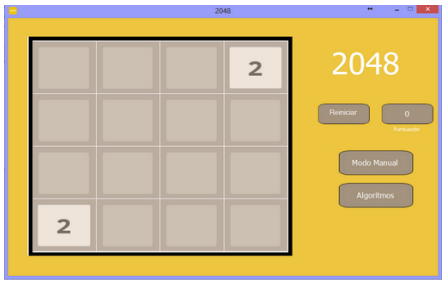


Fig.7. Menú inicial del juego

4. Comparación con otras Implementaciones

A pesar de lo popular de este juego, las implementaciones que se pueden encontrar del mismo no son tantas. Una implementación publicada recientemente en Internet está basada en Inteligencia Artificial [7]. La misma se refiere a lo que denomina Clustering Score realizando una medición de los valores dispersos a lo largo del tablero. Es importante destacar los resultados a los que arriba el autor obteniendo las siguientes conclusiones: "*Como era de esperar la precisión (también conocido como el porcentaje de juegos que se ganó) del algoritmo en gran medida depende de la profundidad de búsqueda que utilizamos. Cuanto mayor sea la profundidad de la búsqueda, mayor será la precisión y cuanto más tiempo se requiere para funcionar En las pruebas hechas, una búsqueda con la profundidad 3 tiene una duración inferior a 0,05 segundos, pero le da un 20 % de posibilidades de ganar, una profundidad de 5 dura aproximadamente 1 segundo, pero le da un 40 % de posibilidades de ganar y finalmente una profundidad de 7 dura 27 a 28 segundos y supone en torno a un 70-80 % de posibilidades de ganar".*

La implementación anterior no se encontraba publicada cuando se comenzó a desarrollar este proyecto, sólo se había encontrado material donde se sugerían estrategias para conseguir ganar el juego. Metodológicamente no hay implementaciones que puedan ser usadas como base para comparar este trabajo. Si observamos los valores alcanzados por nuestra implementación, vemos que supera ampliamente lo anterior, no sólo la performance de los algoritmos implementados sino también que las posibilidades de ganar el juego alcanzan el 100% y que se ha llegado hasta la potencia de 2 de 8192. La tabla que se muestra a continuación (véase Tabla 1) indica el comportamiento empírico de un par de los algoritmos implementados, pudiendo de esta forma asegurar que los resultados fueron mejor que los esperados antes de comenzar el proyecto, en la etapa de estudio de distintas estrategias.

Tabla 1. Comportamiento empírico con objetivo puesto en 8192. La misma muestra la cantidad de pasos o movimientos de las soluciones de prueba junto con la cantidad de estados (tableros) generados por el algoritmo para encontrar la solución.

8192			
Backtracking (Sin heurística)		**A***	
Estados	**Pasos**	**Estados**	**Pasos**
248757	4671	203481	4707
87781	4257	151180	4245
198358	4267	138256	4288
895468	4648	167313	4368
1282779	4688	102076	4173
84106	4530	286570	4292
256811	4305	239592	4396
182515	4215	121805	4195
98512	4395	210567	4389
197637	4436	135690	4480

En promedio, 32423 son los escenarios generados mientras que en la búsqueda guiada necesita 34750. Si bien la media no siempre representa de la mejor manera el conjunto de resultados, el mismo nos sirve para poder comparar ambos recorridos de forma más homogénea. Este dato, nos permite percibir más detalladamente que la búsqueda en profundidad se ve muy afectada por valores extremos ya que la mejor y la peor solución -considerando estados visitados del árbol- se alejan mucho del promedio lo que da cuenta de valores dispersos y, que en parte el azar juega un papel más importante en este algoritmo.

Por el lado de la búsqueda guiada, en cambio, si bien en promedio necesita más estados, en cierto modo el conjunto de prueba cuenta con resultados más parejos.

Cabe aclarar que, el mismo análisis empírico también fue realizado para los comportamientos de ambos algoritmos con el objetivo puesto en 2048 y 4096. En comparación, se percibió que la búsqueda en profundidad a medida que se aumenta la potencia de dos para ganar el juego, si bien responde, cada vez la relación entre estados y pasos baja en gran medida. Sin embargo, pareciera que la búsqueda guiada también lo hace pero no tan notoriamente.

Los resultados obtenidos para el backtracking en conjunto con la heurística A se pueden calificar como buenos solo para obtener soluciones hasta la ficha con valor 2048 debido a que el algoritmo pierde su eficacia pasado ese tope. En comparación con la implementación de A*, el algoritmo mencionado anteriormente permite llegar a

una solución en menor cantidad de pasos a causa de la gran poda que realiza y la eliminación rápida de nodos o estados de baja potencia.

5. Conclusiones

El principal objetivo de este proyecto fue la aplicación de técnicas de diseño de algoritmos para lograr una aplicación idéntica a la que se conoce del juego y que llegue a ganar y superar el 2048. Esto se ha logrado mediante la aplicación de una metodología que conservó las buenas prácticas de la programación, basada en clases, e independiente de la interfaz gráfica. Destacamos que todo el proceso ha sido desarrollado una vez cursado el 2do año de la carrera donde se dicta la materia, lo que ha implicado un gran desafío y ha sido el primer proyecto de esta escala que hemos abordado.

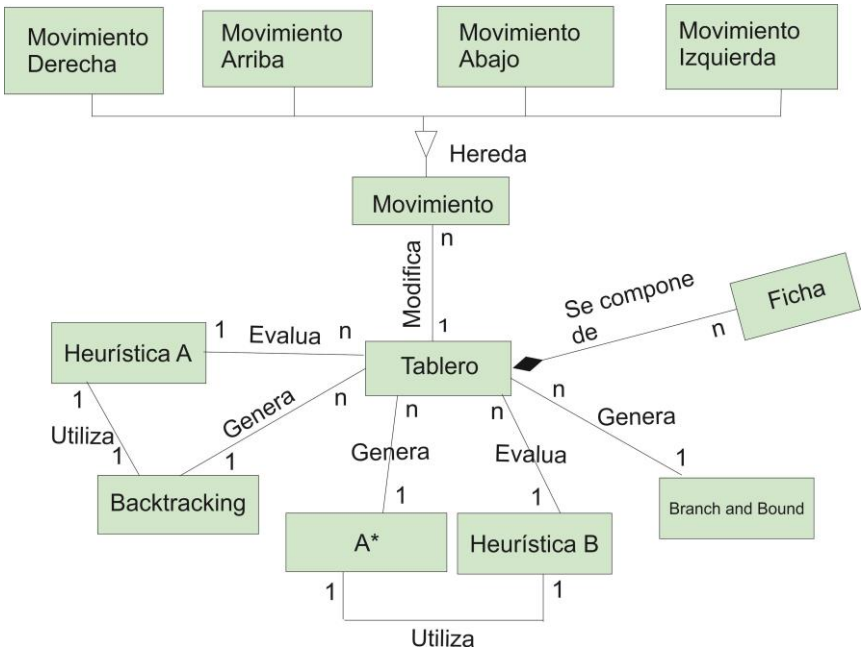
Referencias

1. Digia Plc. *Qt Documentation*. http://doc.qt.io
2. The c++ Resource Network http://www.cplusplus.com/
3. Stroustroup, Bjarne. *Programming: Principles and Practice using C++*. First edition. Addison-Wesley Professional, 2008.
4. Cormen, T.; Lieserson, C.; Rivest, R. *Introduction to Algorithms*. Ed. The MIT Press. 2009.
5. Horowitz, E.; Sahni, S.; Rajasekaran, S. *Computer Algorithms / C++*. Silicon Press; 2 edition, 2007
6. Aho, A. Ullman, J. *Foundations of Computer Science*. C Edition. Computer Science Press. 1995.
7. Vasilis V. Using artificial intelligence to solve the 2048 game. *Blog on Machine Learning, Statistics & Software Development*.http://blog.datumbox.com/
8. Anexo B

Anexo A

Diagrama de interacción entre clases/métodos más relevantes

En el siguiente diagrama se muestra la relación de las clases más importantes junto con algunos métodos de la clase “búsquedas” que se consideraron ejes del proyecto cuyas funcionalidades se basan en encontrar la solución del juego siguiendo el lineamiento de cada algoritmo. Cabe aclarar que, el esquema no es un diagrama de clases ni mucho menos, sino es un simple grafico que intenta demostrar la relación de aspectos considerados significativos en el diseño, que incluye tanto clases, como los métodos más relevantes y la nomenclatura de técnicas algorítmicas utilizadas.



Anexo B

Algoritmos para la resolución del juego 2048

1.1 Backtracking(Sin heuristica)

```
bool Busquedas::DFS(int objetivo,Tablero inicial, list<Tablero> & solucion)
{
    if (inicial.EsSolucion(objetivo))
    {
        solucion.push_back(inicial);
        return 1;
    }
    else
    {
        list<Movimiento*> posiblesMov;
        inicial.posiblesMovimientos(posiblesMov);
        list<Movimiento*>::iterator it=posiblesMov.begin();
        while (it!=posiblesMov.end())
        {
            if (!inicial.perdio())
            {
                Tablero anterior=inicial;
                solucion.push_back(inicial);
                estados++;
                it->setTab(inicial);
                it->mover();
                if (DFS(objetivo,inicial,solucion))
                    return 1;
                solucion.pop_back();
                inicial=anterior;
            }
            it++;
        }
    }
    return 0;
}
```

1.2 Backtracking(con heurística A)

```
int Busquedas::back(Tablero cop,int &eleccion, int prof){
    int contador=0;
    int j;
    int aux=1;
    int a=-12340;
    Tablero b;
    if (prof >=0){
        for (int it=0;(it<3);it++){
            b=cop;
            movs[it]->setTab(&b);
            movs[it]->mover();
            if ((movs[it]->cambio())&& (!b.perdio())){
                contador++;
                aux=it;
                j= Heu.evaluar(&b) + back(b,aux,prof-1);
                if (j>=a){
                    eleccion = movs[it]->getNombre();
                    a=j;
                }
            }
        }
        if(contador == 0){
            eleccion = movs[3]->getNombre();
            return -10000;
        }
    }
    return a;
}
```

1.3 Heurística A

```
#include "heuristica.h"

Heuristica::Heuristica()
{
}

int Heuristica::evaluar(Tablero *t){
    return t->getPuntaje() + (t->movdis()*t->fichaAlta()) + sumar(t);
}

int Heuristica::sumar (Tablero *t){
    int aux=0;
    Ficha* au;
    for (int i=0; i<4; i++)
        for(int j=0; j<40; j+=10){
            au = t->getFicha(i+j);
            aux= aux + au->getValor()* mat[i][j/10];
        }
    return aux;
}
```

1.4 Algoritmo A*

```
void Busquedas::Heuristica(int objetivo, Tablero inicial,list<Tablero> & solucion)
{
    list<Nodo> NodosVivos;
    Nodo padre;
    padre.tablero=inicial;
    NodosVivos.push_back(padre);
    estados=0;
    padre.f=heurist(1,padre.tablero);
    padre.g=0;
    bool encontro=0;
    while ( (!encontro) && (!NodosVivos.empty()))
    {

        Nodo padre=NodosVivos.front();
        NodosVivos.pop_front();
        list<Movimientos*> posiblesMov;
        padre.tablero.posiblesMovimientos(posiblesMov);
        while (!posiblesMov.empty())
        {
            Nodo actual;
            actual.tablero=padre.tablero;
            posiblesMov.front()->setTab(actual);
            posiblesMov.front()->mover();
            estados++;
            posiblesMov.pop_front();
            actual.g=padre.g++;
            actual.h=heurist(actual.solucion.size()+1,actual.tablero);
            actual.f=actual.h+actual.g;
            actual.solucion=padre.solucion;
            actual.solucion.push_back(padre.tablero);
            if (actual.tablero.EsSolucion(objetivo))
            {
                encontro=1;
                solucion=actual.solucion;
                solucion.push_back(actual.tablero);
            }
            if   (!actual.tablero.perdio())
                InsertarOrdenadoH(actual, NodosVivos);
        }
    }
}
```

1.5 Heurística B

```
int Busquedas::heurist(int pasos,Tablero tablero)
{
    int h=( tablero.Puntuacion()/pasos )+pasos*tablero.CasillerosLibres() ;
    return (-1*h);
}
```

1.6 Branch and Bound

```
void Busquedas::BranchAndBound(int objetivo, Tablero inicial,list<Tablero> & solucion)
{
    list<Nodo> NodosVivos;
    Nodo padre;
    padre.tablero=inicial;
    padre.cota=objetivo;
    NodosVivos.push_back(padre);
    estados=0;
    bool encontro=0;
    int CotaSolucion=objetivo;
    while ( (!encontro) && (!NodosVivos.empty()))
    {
        Nodo padre=NodosVivos.front();
        qDebug ()<< padre.cota;
        NodosVivos.pop_front();
        list<Movimiento*> posiblesMov;
        padre.tablero.posiblesMovimientos(posiblesMov);
        while (!posiblesMov.empty())
        {
            Nodo actual;
            actual.tablero=padre.tablero;
            posiblesMov.front()->mover();
            actual.solucion=padre.solucion;
            actual.solucion.push_back(padre.tablero);
            actual.cota=Cota(actual.tablero);
            estados++;
            posiblesMov.pop_front();
            if ( (CotaSolucion<=actual.cota) && (!actual.tablero.perdio()) )
            {
                InsertarOrdenado(actual, NodosVivos);
            }
            if (actual.tablero.EsSolucion(objetivo))
            {
                encontro=1;
                solucion=actual.solucion;
                solucion.push_back(actual.tablero);
            }
        }
    }
}
```