Sistemas de Inteligencia Artificial

Estrategias de búsqueda informadas y no informadas - Simple Squares

Introducción:

El juego Simple Squares tiene como objetivo llevar a cada bloque a su correspondiente target. En el siguiente informe se dispondrán varias estrategias de búsqueda informada y no informada para resolver el juego y se realizará una comparación entre ellas.

Desarrollo del juego y jugadas válidas

El juego toma lugar en un tablero de n x n casilleros. Los bloques son provistos por el juego con una disposición inicial así como también sus targets y las flechas giratorias. La cantidad de bloques por tablero es variable y es diferente para cada nivel.
Para poder cumplir el objetivo del juego se le permite a los bloques moverse de a un casillero en su propia dirección, empujar a otro y/o rotar su dirección al caer en un casillero con flecha.
En el readme del proyecto se encuentran las indicaciones a seguir para correr el programa.
Se utilizó como base el motor propuesto por la cátedra con algunas modificaciones.

Solución Propuesta

Se tomó como base el GPSEngine propuesto por la cátedra, pero se realizaron algunas modificaciones. Por ejemplo, decidimos que resultaría más eficiente quitar un nodo de la frontera y de los nodos ya explotados si encontramos otro nodo con el mismo estado pero un costo menor. Otra modificación que se realizó fue la utilización de una cola de prioridades para almacenar los nodos frontera en A*.
Para representar las soluciones gráficamente y poder verificar si se estaba realizando lo deseado, implementamos la creación de gráficos en cada corrida. Estos muestran los árboles recorridos y los caminos elegidos.

Función de costo: constante de valor 1 ya que al aplicar una regla solo se puede dar un paso en reael juego.

Estructuras de datos utilizadas

Estado:

Se dispone de dos listas:
List<Block> blocks = new ArrayList<Block>();
List<Arrow> arrows = new ArrayList<Arrow>();
Block: Contiene su posición actual, su target y su direccion.
Arrow: Consta de su posición actual y una direccion, la cual va a adquirir un *bloque* si llega a esta posición.

Para poder obtener un nuevo estado utilizamos una regla, que consiste en mover un bloque en la dirección permitida. El conjunto de reglas posibles se aplica a cada casillero del tablero, se mira si hay un bloque y si lo hay, se lo puede mover según su dirección. Si no hay bloque, la regla no es aplicable.

Estado final: Se optó por modificar la implementación original de la función de estado final del motor, por una función que verifica si cada uno de los bloques se encontraba en su posición destino.

Estrategias de búsqueda desinformadas:

- ***DFS***: Recorre el árbol de soluciones en profundidad, explora la rama y luego aplica backtracking.
  Esté método no asegura que la solución sea óptima.
  En la mayoría de los casos este método resulta uno de los más lentos ya que en el juego suele haber una unica solucion y esto hace que se pierda mucho tiempo explorando ramas sin encontrar soluciones. Para evitar que el DFS no recorra ramas infinitas, se agregó un límite máximo virtual al tablero, simulando que es de 10x10 para ponerle un fin a las ramificaciones. A su vez el motor mismo se defiende de los ciclos ya que verifica que no se analicen estados repetidos si tienen mayor costo.

- ***BFS***: Recorre el árbol de soluciones por nivel y aplica las reglas a lo ancho. En nuestro caso, como el costo de aplicar una regla siempre es uno, la profundidad del árbol de soluciones generado por bfs equivale a la cantidad de movimientos que hay que realizar para resolver el juego. Suele ser una solución más óptima que DFS si no hay múltiples soluciones.

- ***Iterative Deepening:*** Se podría decir que este método combina BFS y DFS, ya que toma lo mejor de cada una de la estrategias. Se aplica el algoritmo DFS variando la profundidad. Es decir, primero se ejecuta DFS solo hasta profundidad uno, luego (si no se encontró la solución) se aumenta la profundidad a dos y se realiza DFS de nuevo, desde el comienzo, y así sucesivamente. Al igual que BFS, asegura que la profundidad para la cual se encontró un estado solución es la cantidad de movimientos que hay que realizar para resolver el juego. Resulta ineficiente si hay una gran cantidad de reglas, dado que se construye el árbol de soluciones para cada profundidad, analizando estados reiteradas veces.

Estrategias de búsqueda informadas:

- **Greedy**: Consiste en analizar el mejor estado posible a partir del estado anterior. Utiliza una función heurística que proporciona un puntaje para cada estado, siendo 0 el puntaje para un estado solución. El próximo estado que se considerará para aplicarle las reglas será el que tenga menor valor heurístico. Generalmente, no llega a la solución óptima. Esto se debe a que solo analiza los nodos recién expandidos, y si la solución se encuentra en otra rama, tardará más en encontrarla . Por otro lado esto favorece la eficiente asignación de recursos, haciendolo una buena elección de estrategia si contamos con un hardware limitado.

- **A***: Algoritmo de búsqueda que siempre toma el mejor de los nodos frontera. Esto lo realiza teniendo en cuenta el valor heurístico del nodo así como también su costo. Además de ser muy eficiente, dado que siempre elige el camino más conveniente, tiene la particularidad de que si se le aplica con una heurística admisible, es decir que no sobreestima el valor del costo, siempre encontrará el camino más óptimo.

Heurísticas propuestas

- **“Default”**: Es la predeterminada porque es considerada por el resto de las heurísticas planteadas. Identifica los estados donde ya no es posible ganar y devuelve un valor heurístico máximo, para asegurarse de que nunca se tomarán. Esto es cuando un bloque queda en una posición afuera de la “grilla inicial” (La disposición total donde comienzan los bloques) y con

una dirección que no sea hacia adentro de la grilla inicial, por lo tanto, este bloque ya no podrá volver.

No admisibles:

- **“In Path”**: Calcula el camino necesario para alcanzar el objetivo, es decir, cuantas veces hay que mover al bloque consecutivamente para que llegue a su posición destino. Si el bloque no se encuentra encaminado hacia su objetivo, se devuelve la distancia de manhattan hasta este. Luego se suman los resultados de todos los bloques del tablero y así se llega a un valor heurístico. Esta heurística es muy útil en tableros en los que hay muchas flechas que cambian al bloque de dirección y para los cuales el bloque deberá “dar muchas vueltas” para llegar a su objetivo. In Path no es admisible, dado que existe la posibilidad de que en el camino se empuje a otros bloques, por lo que se estaría contando el movimiento de un bloque más de una vez, y así sobrestimando el costo real.

- **“Min Distance 1”**: Calcula la distancia de Manhattan de todos los bloques a sus objetivos y devuelva la suma de todas esas distancias. Si bien está heurística calcula la menor cantidad de movimientos que un bloque debe realizar para llegar a su destino, no tiene en cuenta que quizás podría estar empujando a otro o mismo estar siendo empujado por otro, y de esta forma llegar a destino en menos movimientos. Es por esta razón que Min Distance 1 resulta no admisible. De todas formas, resulta muy efectiva para tableros en los que los bloques tienen que hacer un camino directo (siempre tienden a acercarse a su objetivo) y no tienen que dar demasiadas vueltas.

- “**Min Distance 2**”: La heurística consiste en calcular las distancias de Manhattan de cada uno de los bloques hasta su objetivo, al igual que en Min Distance 1, pero la diferencia es que ésta también se fija cuántos bloques hay en el “área” que se genera entre el bloque y su objetivo. Por cada bloque, se dividirá la distancia de Manhattan por la cantidad de bloques que haya, por lo cual, se está calculando como si durante todo el camino, el bloque estaría siendo empujado. Esta aproximación pareciera ser admisible pero no lo es, ya que existen casos en los que hay bloques afuera del área, que podria entrar al área y empujar al bloque.

- “**Min Distance 3**”: Variante de Min Distance 1. Esta vez analiza para cada bloque, si existen bloques que puedan acercar al bloque en cuestión a su destino. Luego se divide cada distancia de Manhattan por este número y se los suma todos normalmente (ver pseudocódigo en anexo). No considera los casos en que un bloque que inicialmente no sirve para acercar a otro bloque, pueda hacerlo luego de ser rotado por una flecha negra y es por esto que no resulta admisible ya que se podría estar sobreestimando el valor

- “**Not Admissible Combination”**: Esta heurística combina las dos mejores de las heurísticas no admisibles, la “Min Distance”, y la “In Path”. Si bien para heurísticas no admisibles lo mejor sería siempre tomar la de menor valor, en este caso, ambas tienen un valor por debajo de h* en la mayoría de los casos, y por esto, habiendolo comprobado empíricamente, decidimos que lo mejor sería a ser elegir siempre el de mayor valor entre ambos.

Admisibles:

- “**Admissible In Path**”: Calcula los mismos recorridos que In Path para cada bloque y se queda con el mayor de ellos. Se evita sobreestimar los costos y se toma el mínimo camino para llegar a un objetivo. Sin embargo por ser tratarse de una solución “relajando” el problema, la heurística no resulta del todo eficiente.

- **“Admissible Min Distance**”: Tiene el mismo comportamiento que Min Distance1 pero en vez de sumar todas las distancias, solo se queda con la de mayor valor y así se asegura la admisibilidad. Al igual que en Admissible In Path, al “relajar” tanto el problema, la solución no es del todo buena.

- “**Admissible Combination**”: Combina las dos anteriores. En cada nodo calcula ambas heurísticas, tanto “Admissible Min Distance” como “Admissible In Path” y retorna el de mayor valor. Al usar esta combinación, el valor se acerca más al de h*, para cada nodo elegido.

Análisis de resultados

Observando los gráficos y tablas adjuntas se pueden notar las diferencias al aplicar cada estrategia/heurística.En DFS se puede observar como solo se expanden los nodos de una rama y se analiza en profundidad. Esto se observa ya que la mayoría de los nodos que se mueven a la lista de nodos frontera son luego expandidos. Lo mismo sucede al aplicar Greedy.
Por lo contrario en BFS y A*, primero se abren muchos estados y luego se van analizando, pero tienden a quedar muchos más nodos abiertos en frontera.
La cantidad máxima de nodos que analizarán en BFS es $n^m - 1$, siendo n = cantidad de bloques y m = cantidad mínima de movimientos para llegar a la solución; en Iterative Deepening será $\sum_{1}^{m} n^m - 1$.

Board 4:
Este nivel contiene dos bloques (solo hay dos movimientos posibles), por lo cual el factor de ramificación (número de nodos hijos de cada nodo) es siempre dos. Al ser éste considerablemente pequeño en relación a otros niveles con más bloques, todos los algoritmos pueden llegar a la solución en un tiempo razonable.

Board 5:
Este nivel es corto ya que se necesitan pocos movimientos para ganar. Al tener 3 bloques, la ramificación es mayor que en el nivel anterior y por lo tanto las estrategias sin heurísticas expandirán una gran cantidad de nodos por nivel. Esto se ve en DFS y Greedy, ya que ambos algoritmos tienden a irse por una rama. Las ramas serán muy grandes y en caso de no elegir la correcta se deberán expandir muchos nodos.
Por otro lado, A* resulta muy bueno en este nivel dado que al no necesitar hacer muchos movimientos, los caminos con menor costo van a terminar siendo siempre los más convenientes para llegar a una solución rápida.

Board 16: La principal diferencia con los otros tableros es que para resolverlo, el bloque rojo pasa por su objetivo, se aleja y luego se vuelve a acercar. Por este motivo, la heurística inPath resulta más conveniente que el minDistance. Llega un momento en el cual el bloque debe alejarse (minDistance aumenta) pero la flecha que apunta hacia la izquierda permite que el bloque siga encaminado y de esta forma se lo resuelve.

Board 21:
Este nivel tiene dos características fundamentales: 4 bloques y flechas que forman un ciclo. Para poder ganarlo, los bloques deben salir de este ciclo. Es importante entender que los algoritmos no informados no pueden hallar una solución para dicho número de bloques, no al menos para el tiempo o tamaño de memoria con el que contamos para este trabajo. En BFS por ejemplo, esto se debe a que el árbol se expande tanto en ancho que nunca se llega a la profundidad 23, cantidad mínima de movimientos para ganar. Con Greedy sucede algo similar, ya que, si no elige las ramificaciones correctas inicialmente, no se podrá encontrar la solución. A*, en cambio, logra solucionarlo ya que al tener en cuenta todos los caminos posibles con sus costos, opta por salir del ciclo y luego puede llegar a una solución. En una

primera instancia supusimos que convenía usar “In Path” ya que dentro del ciclo ese algoritmo llevaría a dar una aproximación más acertada del camino que queda por recorrer, mientras que la distancia de Manhattan retornaría en la mayoría de los casos un “atajo” que no es posible de recorrer. Sin embargo, min distance 1 e inPath tomaron un tiempo similar, diferenciándose en que Min Distance 1 explotó más nodos que In Path. Esto puede deberse a que calcular las distancias de Manhattan resulta mucho más rápido que aplicar el algoritmo de InPath.

Conclusiones

De los algoritmos de búsqueda no informados, podemos concluir que si hay una cantidad reducida de soluciones, BFS e Iterative Deepening suelen ser mejores opciones que DFS, mientras que si hay una gran cantidad de soluciones posibles, DFS puede llegar a encontrar la solución en un menor tiempo. Sin embargo, al no incluir información intrínseca del problema, ninguna de estas estrategias de búsqueda suelen encontrar una solución para estados más complejos en un tiempo razonable. Por lo tanto, incluir información del problema nos proporciona una gran herramienta para reducir el tiempo de búsqueda de la solución. Sin embargo, interpretar información propia del problema es más difícil y se pierde cierta objetividad.Para tratar de atacar el problema de la mejor forma posible se lo “relaja” y se intenta definir heurísticas que ayudan a los algoritmos a encontrar una solución.
Así podemos ver como cada heurística fue pensada para resolver algún tipo de problema común del juego.
La heurística Min Distance 1 resuelve mejor el problema de los niveles en lo que los bloques tienen caminos directos a sus objetivos y no requieren que un bloque empuje a otro para hallar la solución. Min Distance 2 resuelve mejor los niveles que requieren que los bloques empujen a los otros y Min Distance 3 se encarga de resolver el problema al revés donde los otros bloques deben empujar al que se está analizando.

Luego vemos como Greedy suele resultar mejor que DFS ya que elige siempre la mejor solución entre los próximos estados posibles, mientras que Iterative Deepening y BFS dependen de la profundidad en la cual se encuentre la solución, o sea la cantidad de movimientos necesarios para encontrarla.

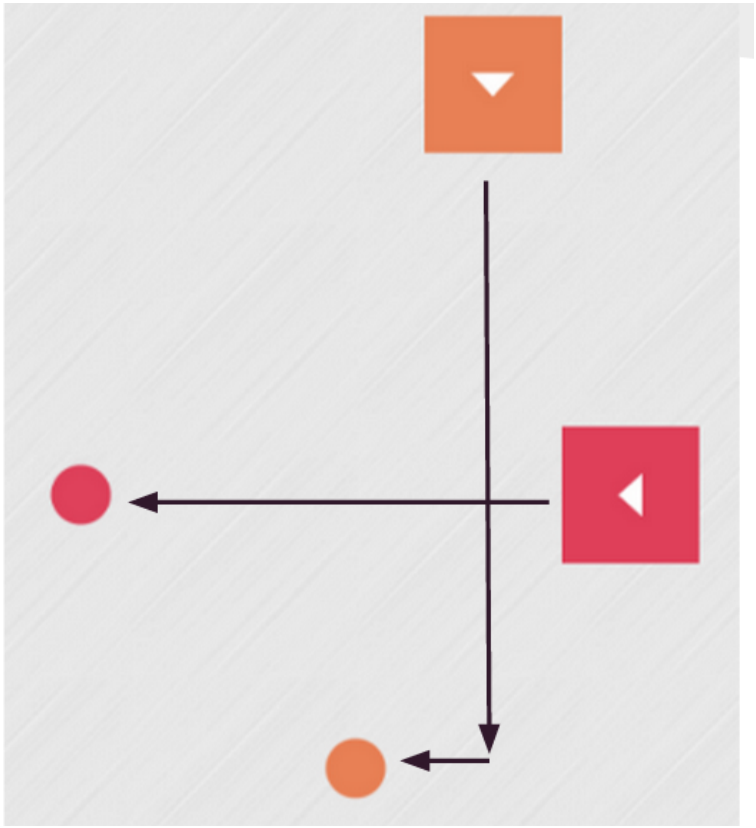
Astar, agrega más dificultad al problema, pero resulta la mejor estrategia de búsqueda. La principal dificultad es que para asegurarse de encontrar la solución óptima, las heurísticas utilizadas deben de ser admisibles. Generalmente, está característica es difícil de percibir: en nuestro caso, tuvimos que “relajar” el problema para lograrlo. En Admissible Min Distance al no considerar que los bloques tienen una direccion para moverse, ni que existen flechas que cambian el sentido del bloque y siempre suponer el $h^*(x)$ mínimo, es decir en el caso en que el bloque más lejano puede mover a todos los otros y situarlos en sus objetivos, sin tener estos que desplazarse por sigo mismos. Ahora bien, al convertir min Distance 1 en admisible, resignamos una característica fundamental: si bien este algoritmo sobreestima en algunos casos, su valor, suele estar muy cerca de $h^*(x)$, mientras que su solución admisible es generalmente mucho menor a $h^*(x)$. En otras palabras, admissible min distance expande casi siempre más nodos que minDistance, tardando más tiempo en encontrar la solución.

En conclusión podemos observar que en términos generales las estrategias de búsqueda informadas suelen ser más convenientes en la mayoría de los casos e incluso permiten hallar soluciones en ciertos casos complejos en los que las estrategias de búsqueda no informadas no pueden hacerlo. Por lo tanto concluimos que las heurísticas planteadas resultan útiles para resolver el juego y se acercan a la realidad.

Anexo

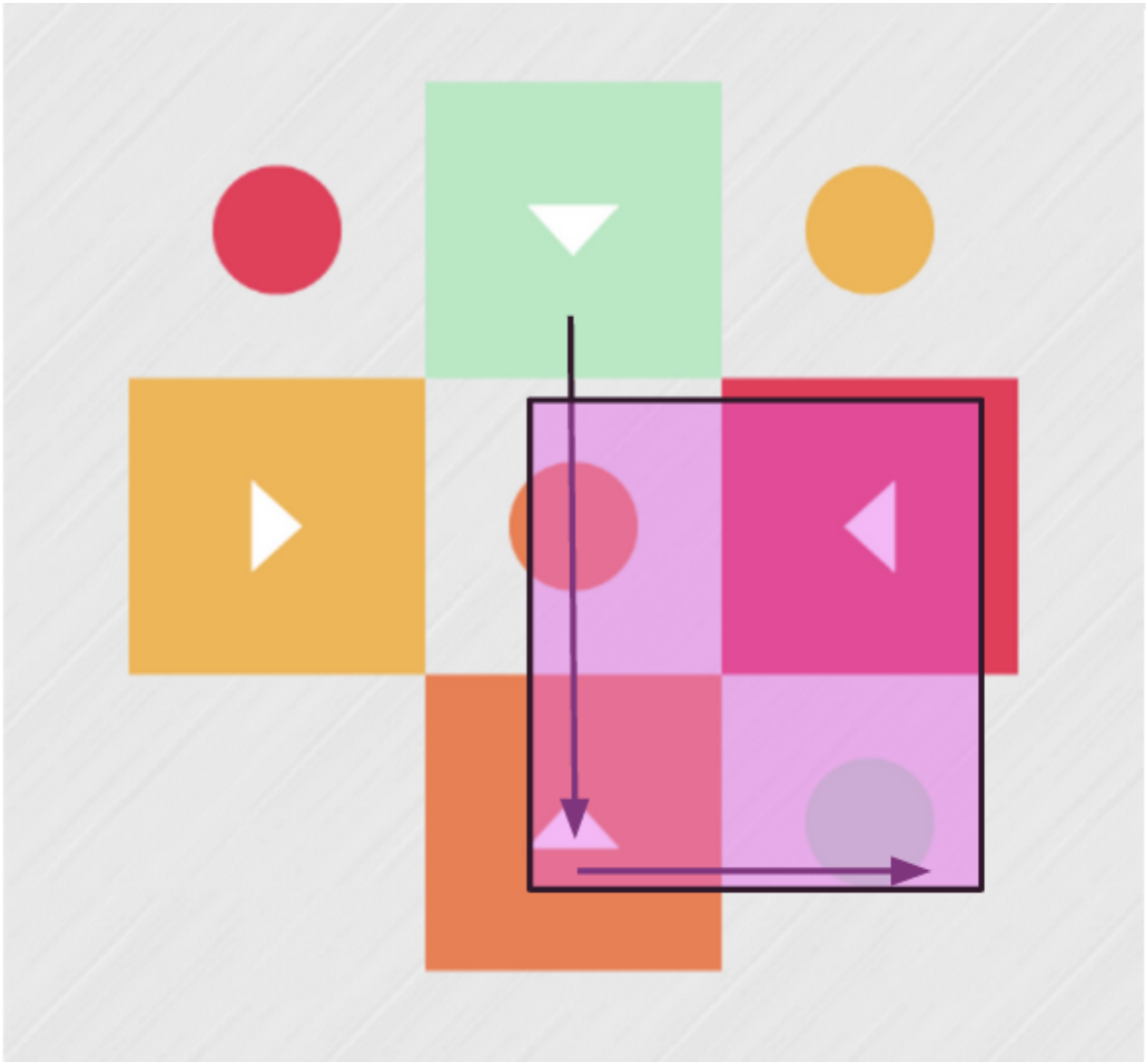
Heurísticas:

Min Distance 1



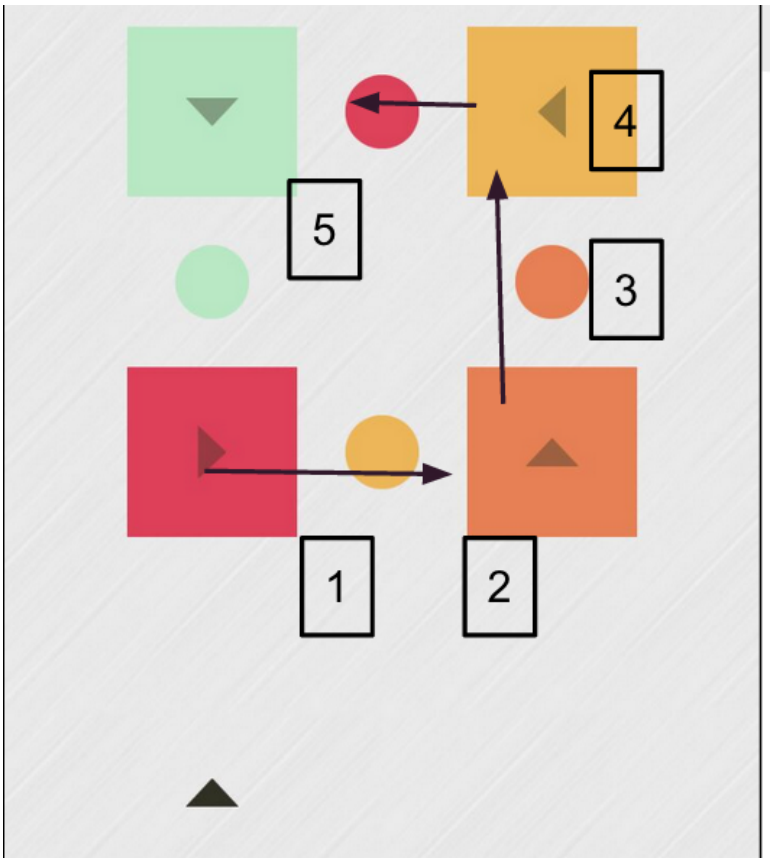
En la figura se muestran las distancias de Manhattan utilizadas por la heurística.

Mindistance 2



En la figura se muestra el área generado por las distancias de manhattan que se utiliza para calcular la heurística.

In Path



La figura muestra uno de los paths utilizado por la heurística.

Nivel 4

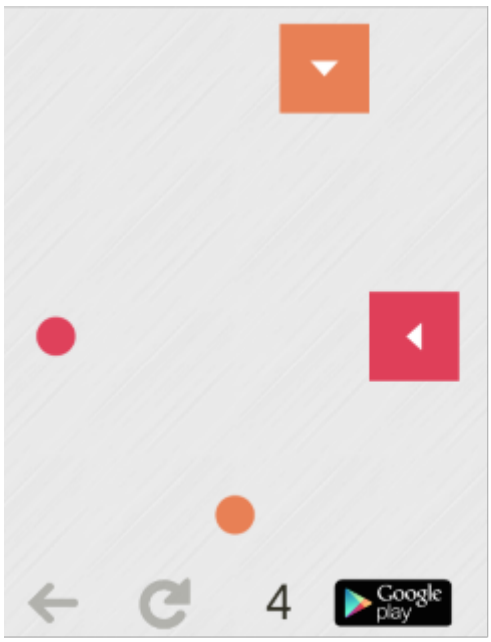
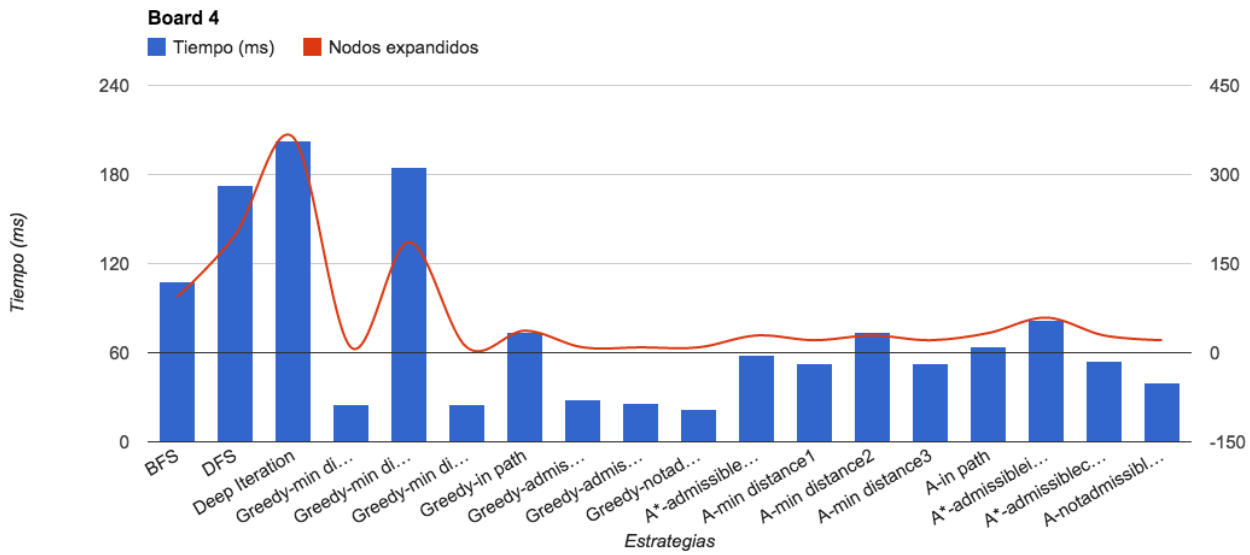


Imagen 2.1 - Nivel 4 de Simple Squares

Estrategia	Tiempo (ms)	Movimientos	Nodos expandidos	Nodos frontera	Estados generados
BFS	108	10	94	33	128
DFS	173	10	198	2	201
Deep Iteration	203	10	363	2	122
Greedy-min distance1	25	10	9	9	19
Greedy-min distance2	185	10	186	5	192
Greedy-min distance3	25	10	9	9	19
Greedy-in path	74	10	37	8	46
Greedy-admissiblemindistance	28	10	9	9	19
Greedy-admissiblecombination	26	10	9	9	19
Greedy-notadmissiblecombination	22	10	9	9	19
A*-admissiblemindistance	58	10	29	19	49
A-min distance1	53	10	21	12	34
A-min distance2	74	10	29	20	50
A-min distance3	53	10	21	12	34
A-in path	64	10	33	35	95
A*-admissibleinpath	82	10	59	35	95
A*-admissiblecombination	54	10	29	19	49
A-notadmissiblecombination	40	10	21	12	34

Tabla 1: Comparación de las heurísticas en el tablero 4.



Nivel 5

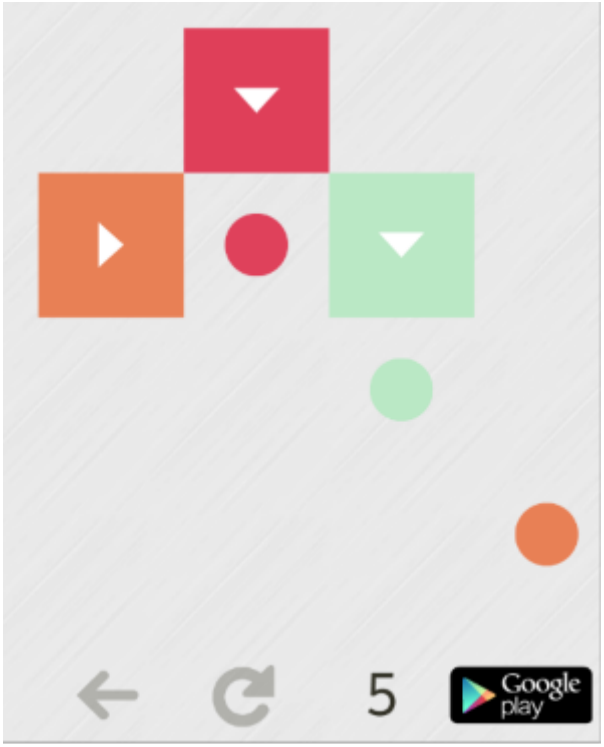
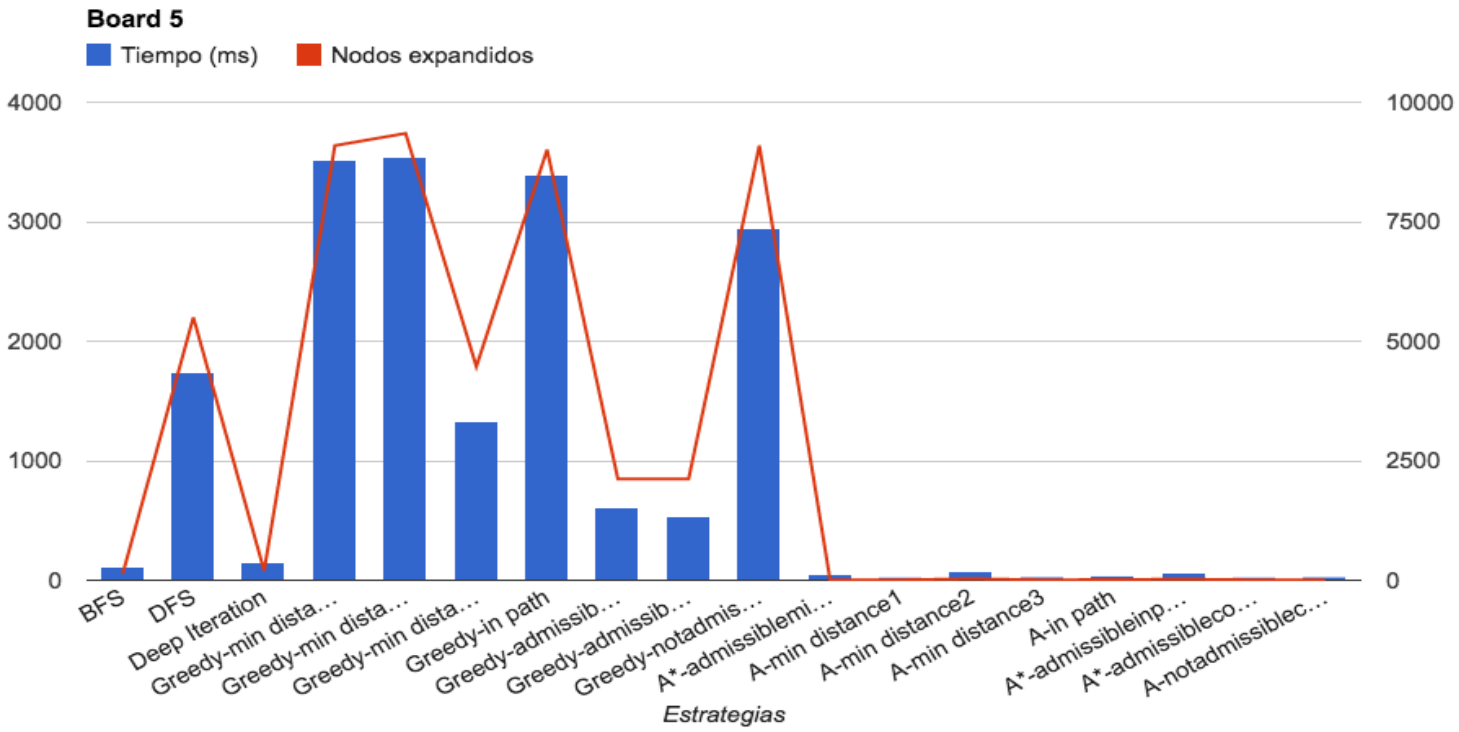


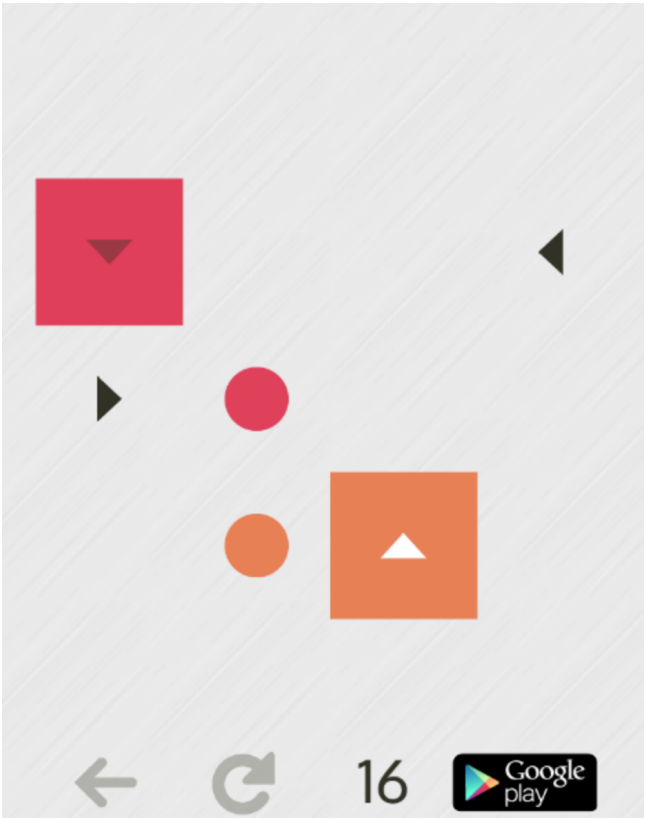
Imagen 3.1 - Nivel 5 de Simple Squares

Estrategia	Tiempo (ms)	Movimientos	Nodos expandidos	Nodos frontera	Estados generados
BFS	117	6	133	123	241
DFS	1741	6	5500	4	5260
Deep Iteration	146	6	199	4	94
Greedy-min distance1	3520	6	9092	5	8559
Greedy-min distance2	3535	6	9348	4	8808
Greedy-min distance3	1330	6	4469	5	4316
Greedy-in path	3394	6	9008	3319	8559
Greedy-admissiblemindistance	604	6	2120	8	2129
Greedy-admissiblecombination	532	6	2120	8	2129
Greedy-notadmissiblecombination	2948	6	9092	5	8559
A*-admissiblemindistance	45	6	9	17	27
A-min distance1	27	6	9	16	26
A-min distance2	73	6	31	48	80
A-min distance3	31	6	9	15	25
A-in path	40	6	14	25	40
A*-admissibleinpath	62	6	24	38	63
A*-admissiblecombination	27	6	9	17	27
A-notadmissiblecombination	30	6	9	16	26

Tabla 2: Comparación de las heurísticas para en el tablero 5.

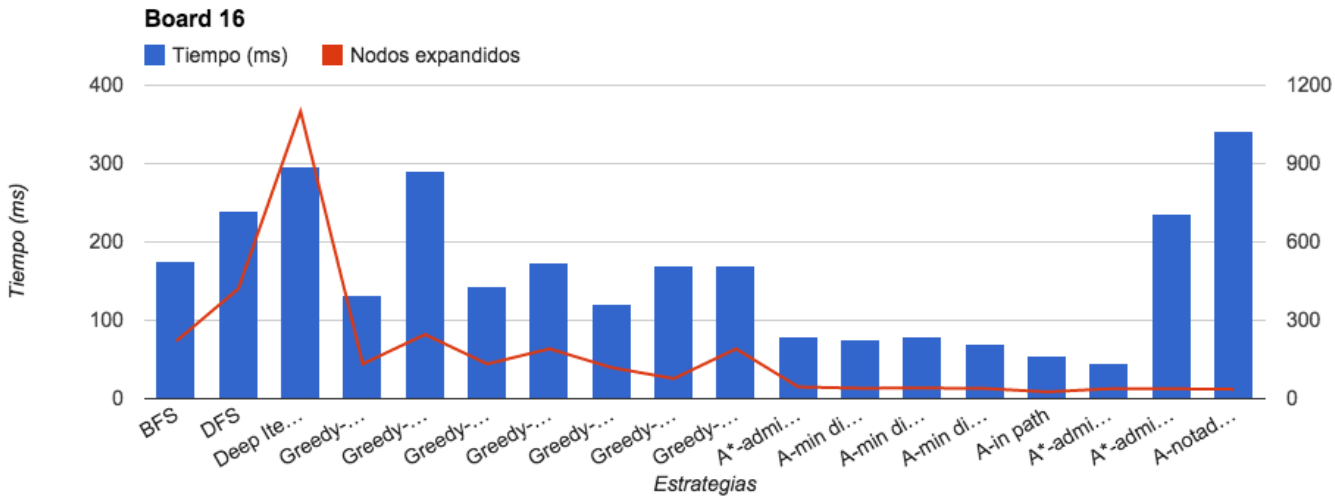

Board 5
Tiempo (ms)
Nodos expandidos
4000
3000
2000
1000
0
10000
7500
5000
2500
0
BFS
DFS
Deep Iteration
Greedy-min dista…
Greedy-min dista…
Greedy-min dista…
Greedy-in path
Greedy-admissib…
Greedy-admissib…
Greedy-notadmis…
A*-admissiblemi…
A-min distance1
A-min distance2
A-min distance3
A-in path
A*-admissibleinp…
A*-admissibleco…
A-notadmissiblec…
Estrategias

Nivel 16

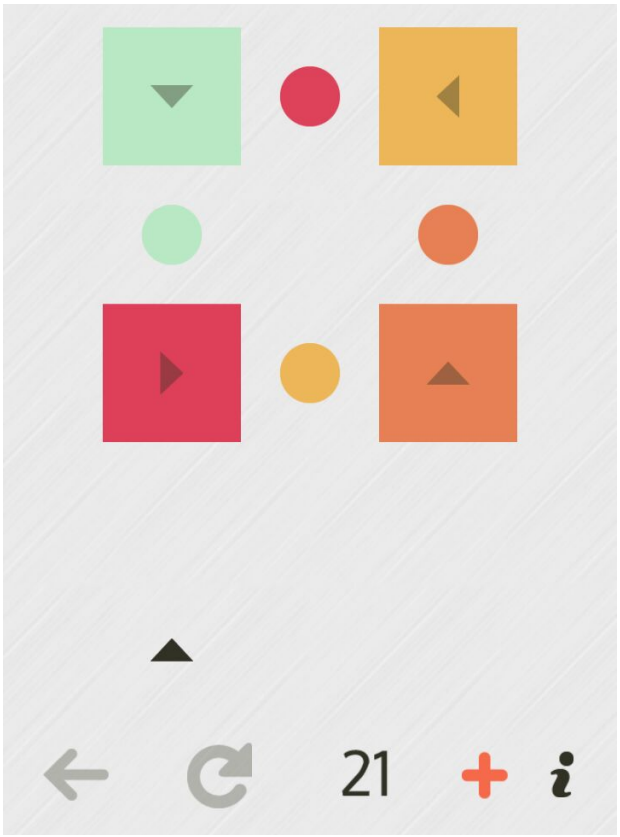


Estrategia	Tiempo (ms)	Movimientos	Nodos expandidos	Nodos frontera	Estados generados
BFS	175	13	220	20	241
DFS	239	13	421	5	336
Deep Iteration	297	13	1100	5	188
Greedy-min distance1	132	13	133	9	129
Greedy-min distance2	291	13	246	8	227
Greedy-min distance3	143	13	133	9	129
Greedy-in path	173	13	191	10	187
Greedy-admissiblemindistance	120	13	119	9	129
Greedy-admissiblecombination	330	13	339	6	344
Greedy-notadmissiblecombination	275	13	339	6	344
A*-admissiblemindistance	80	13	45	29	75
A-min distance1	75	13	40	26	67
A-min distance2	80	13	41	28	70
A-min distance3	69	13	39	27	67
A-in path	54	13	26	26	65
A*-admissibleinpath	45	13	88	26	65
A*-admissiblecombination	236	13	339	6	344
A-notadmissiblecombination	341	13	339	6	344

Tabla 3: Comparación de las heurísticas para en el tablero 16.



Board 21



Estrategia	Tiempo (ms)	Movimientos	Nodos expandidos	Nodos frontera	Estados generados
BFS	inf	n/a	n/a	n/a	n/a
DFS	inf	n/a	n/a	n/a	n/a
Deep Iteration	inf	n/a	n/a	n/a	n/a
Greedy-min distance1	inf	n/a	n/a	n/a	n/a
Greedy-min distance2	inf	n/a	n/a	n/a	n/a
Greedy-min distance3	inf	n/a	n/a	n/a	n/a
Greedy-in path	inf	n/a	n/a	n/a	n/a
Greedy-admissiblemindistance	inf	n/a	n/a	n/a	n/a
A*-admissiblemindistance	20271	23	10935	11888	22824
A-min distance1	2178	24	3605	4522	8083
A-min distance2	12728	23	8299	10281	18552
A-min distance3	10489	23	4803	6142	10925
A-in path	2167	23	3156	4189	7232
A*-admissibleinpath	68194	23	11140	12216	23313
A*-admissiblecombination	40793	23	9163	10270	19420
A-notadmissiblecombination	4399	24	3029	4106	7086

Tabla 4: Comparación de las heurísticas para en el tablero 21.