Métodos de búsqueda Desinformados e Informados

Miguel Di Luca (58460), Facundo Astiz (58333), GRUPO 7

[1] Introducción

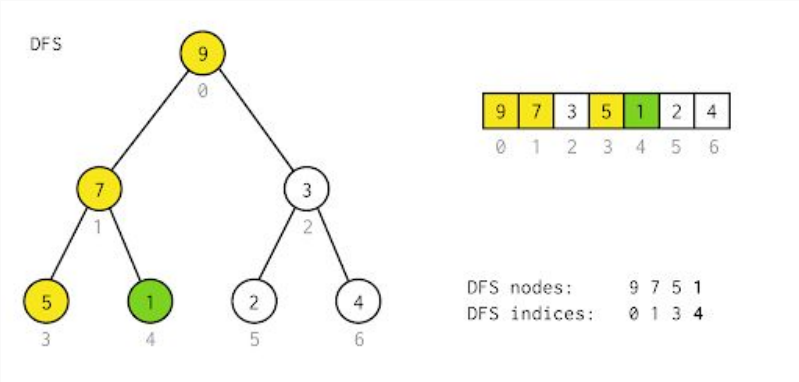
[1] Introducción

- Nuestro trabajo consiste en resolver mapas del juego Sokoban
- Utilizamos dos tipos de métodos de búsqueda:
  - Métodos informados
  - Métodos no informados

[1.1] Métodos no informados

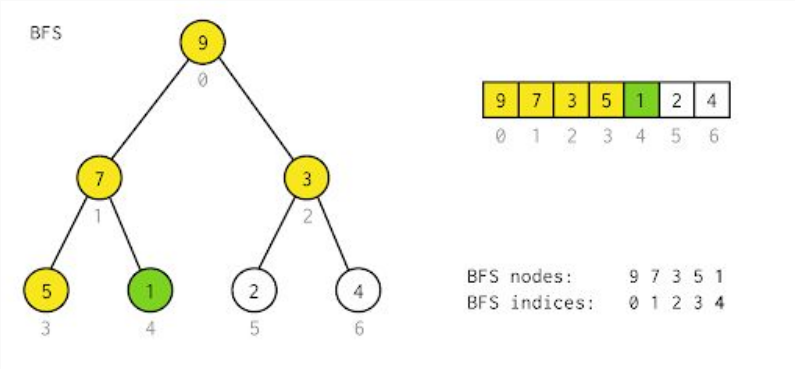
[1.1] Métodos no informados - DFS

- Búsqueda en profundidad
- Se exploran los nodos de forma recurrente
- Cuando llega al fin del camino realiza backtracking



[1.1] Métodos no informados - BFS

- Búsqueda en anchura
- Se explora por niveles



[1.1] Métodos no informados - IDDFS

- DFS limitado por profundidad
- Realiza DFS para cada profundidad entre cero e infinito
- Se puede setear una profundidad máxima

[1.2] Métodos informados

[1.2] Métodos informados - A*

- Busca según **f(n) = g(n) + h(n)**
- En cada paso busca globalmente en todo el árbol
- Expande siempre el nodo frontera con menor valor de **f**

[1.2] Métodos informados - IDA*

- Búsqueda similar a IDDFS
- Se limita por **f(n) = g(n) + h(n)**
- Toma como límite el mínimo **f** de los nodos frontera al finalizar la iteración (la raíz en el primer caso)

[1.2] Métodos informados - Greedy Search

- Busca según **h(n)**
- En cada paso busca globalmente en todo el árbol
- Expande siempre el nodo frontera con menor valor de **h**

[2] Implementación

[2.1] Árbol de juego

[2.1] Árbol de juego

Se consideraron dos posibilidades de representar los nodos:

1. Cada nodo es una instancia de un juego, este posee el tablero, las posiciones de las cajas, del jugador, y de los objetivos. Mayor costo de memoria.
2. Cada nodo es una acción a realizar, se considera una acción a realizar un movimiento de una caja, no se considera mover únicamente el jugador. Al crear un nuevo nodo se debe calcular la lista de acciones desde la raíz. Mayor costo de procesamiento.

Se optó por la opción número uno, ya que el resultado es un código más claro y simple. Además, en cualquier caso podríamos procesar el resultado y optimizar los movimientos del jugador entre desplazamientos de caja.

[2.1] Árbol de juego

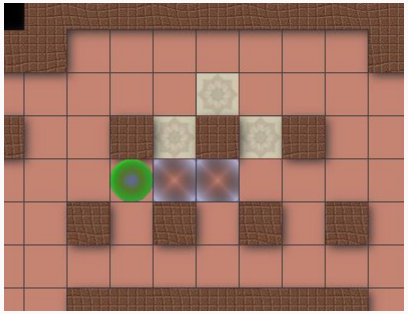
El árbol de juego tal y como está implementado posee ciclos. Optamos por no incluir detección de ciclos dentro de nuestro árbol dejándolo como responsabilidad para los métodos de búsqueda utilizados. Esto es debido a que depende del algoritmo cual es la colección óptima para evitar caer en uno. Por ejemplo, para el dfs utilizamos un set mientras que para un IDDFS utilizamos un mapa.

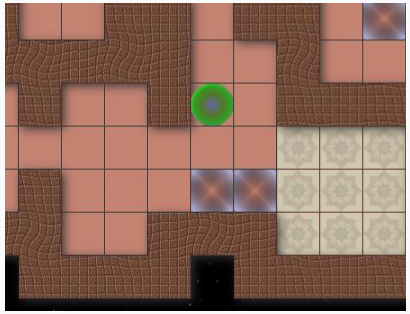
Otra consideración que tuvimos en cuenta fue podar aquellos subárboles que partían de un nodo que se encuentra en estado de *deadlock*. Es decir, cuando a partir de ese nodo es imposible llegar a una solución porque una de las cajas quedo trabada.

[2.1] Árbol de juego

Para los deadlocks se consideraron sólo dos tipos simples:

- Cuando una caja se encuentra en una esquina.
- Cuando dos cajas se encuentran pegadas, sin posibilidad de separarlas.





[2.2] Implementación de los algoritmos

[2.2] Implementación de los algoritmos

Para los algoritmos DFS, BFS, Greedy Search y A*.

- La búsqueda se realiza de igual manera solo que se opera bajo una colección distinta en cada caso.
- Se utiliza un HashSet para guardar los estados ya visitados.

```
private boolean search() {

    while (!collection.isEmpty()) {
        Game game = collection.pop();

        if (game.gameFinished()) {
            this.gameSolved = game;
            return true;
        }

        AtomicBoolean childAdded = new AtomicBoolean( initialValue: false);
        game.calculateChildren().stream()
                .filter(child -> !hashSet.contains(child))
                .forEach(child -> {
                    childAdded.set(true);
                    if (heuristic != null)
                        child.setHeuristicValue(heuristic.evaluate(child));
                    collection.add(child);
                    hashSet.add(child);
                });

        if(childAdded.get())
            this.expandedNodes++;
    }

    return false;
}
```

[2.2] Implementación de los algoritmos

En el momento de hacer un pop de la frontera se realiza una acción distinta dependiendo del algoritmo.

- DFS: Toma el último nodo agregado. Se utiliza un Stack.
- BFS: Toma el nodo con mayor tiempo en la cola. Se utiliza una Queue.
- A*: Toma el nodo con menor valor de **f**. Se utiliza una PriorityQueue, compara según **f**.
- Greedy search: Toma el nodo con menor valor de **h**. Se utiliza una PriorityQueue para comparar según **h**.

[2.2] Implementación de los algoritmos

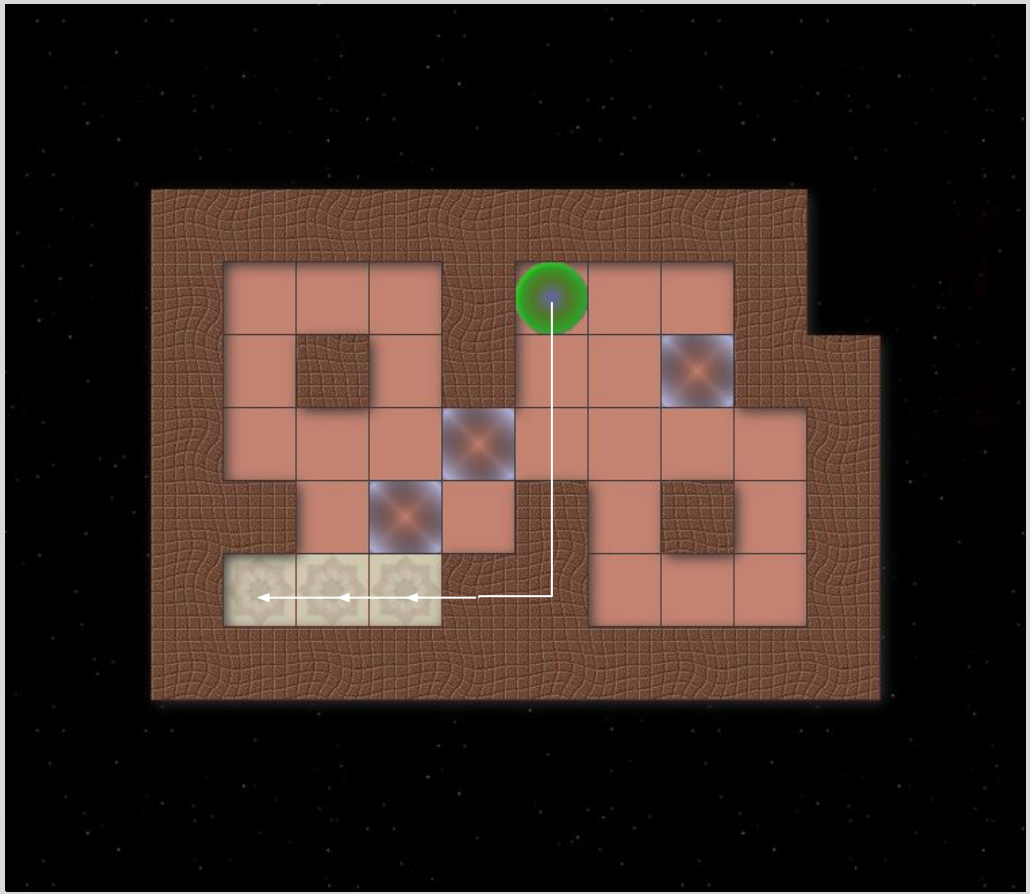
Para los algoritmos IDDFS e IDA*

- Para IDDFS e IDA* es similar, se utiliza un stack y realiza la búsqueda iterativamente según el límite.
- Si luego de una iteración no quedan nodos a expandir, finaliza.
- Se utiliza un HashMap para guardar los nodos visitados.
- Si se visita un nodo repetido a una profundidad menor, se explora, de lo contrario se descarta.

```
private boolean searchIDAStar(int limit) {
    while (!stack.isEmpty()) {
        Game game = stack.pop();

        int f = getFunctionValue(game);
        if (limit < f) {
            remainingToSearch = true;
            if (f < newLimit || newLimit == -1) {
                newLimit = f;
            }
        } else {

            if (game.gameFinished()) {
                this.gameSolved = game;
                return true;
            }


            AtomicBoolean childAdded = new AtomicBoolean( initialValue: false);
            game.calculateChildren().stream()
                    .filter(child -> Utils.checkIfHashMapContainsElementAndReplace(hashMap, child))
                    .forEach(child -> {
                        childAdded.set(true);
                        child.setHeuristicValue(heuristic.evaluate(child));
                        stack.add(child);
                        hashMap.put(child, child.getDepth());
                    });

            if(childAdded.get())
                this.expandedNodes++;
        }
    }
    return false;
}
```

[3] Heurísticas

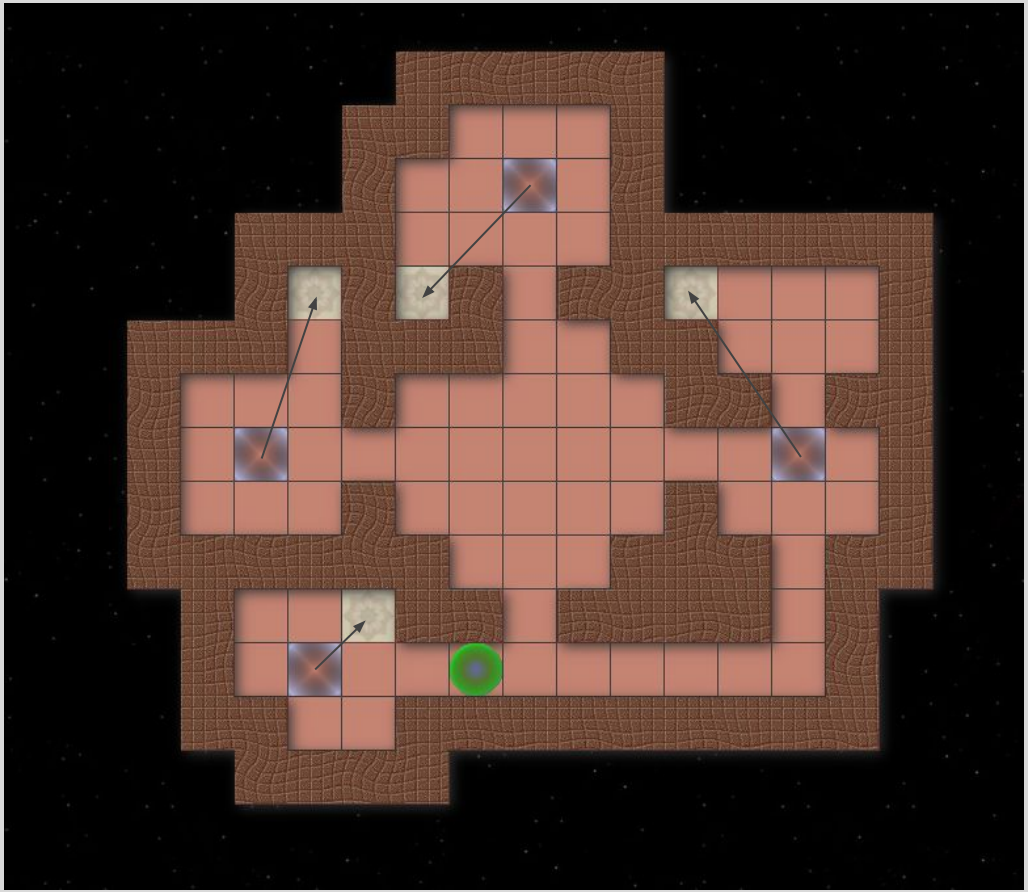
Heurística experimentativa



Distancia caminando

Se calcula por fuerza bruta el camino más corto posible para recorrer todos los goals caminando (ignorando las paredes).

Heurísticas de asignación



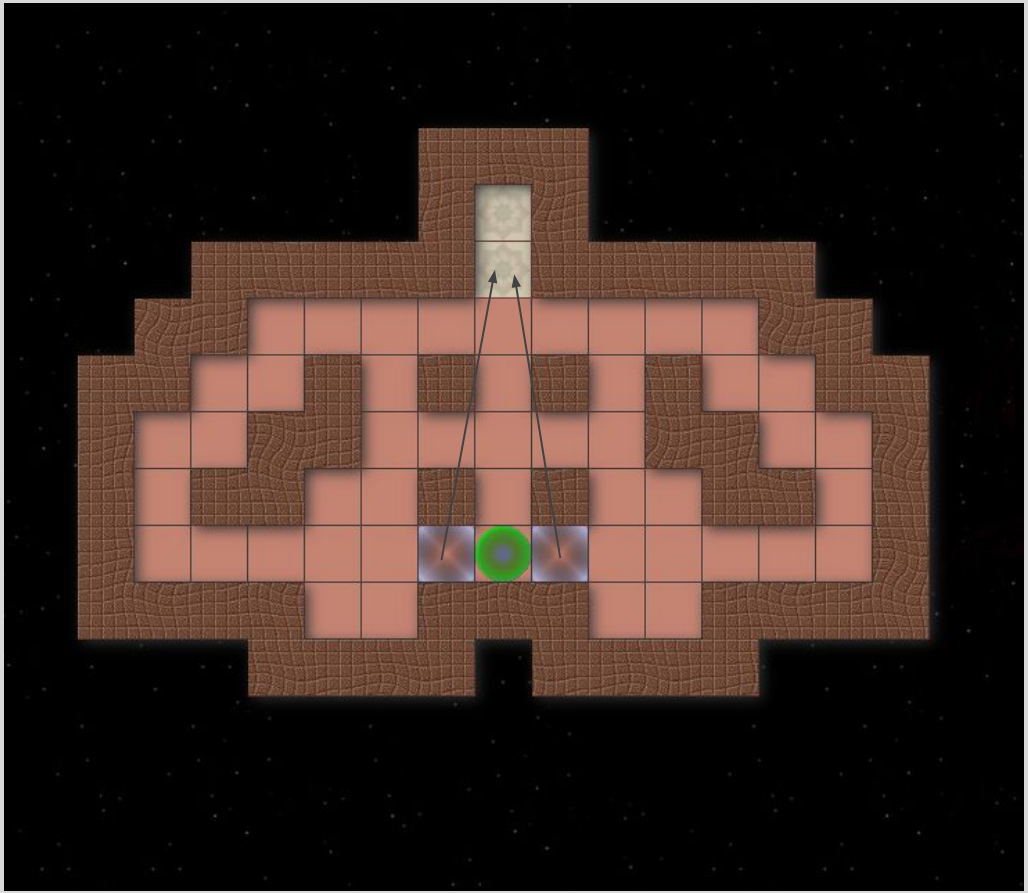
¿De que se trata?

Se trata de asignar cajas a goals y luego calcular las distancias manhattan para cada una de ellas.

El resultado de la heurística es la suma de las distancias calculadas.

Todas estas heurísticas serán admisibles.

Heurísticas de asignación - Simple



Asignación simple

Para cada caja buscamos el goal más cercano sin importar que ya haya sido asignado.

De todos los métodos de asignación implementados es el que requiere menos tiempo de procesamiento.

Heurísticas de asignación - Greedy

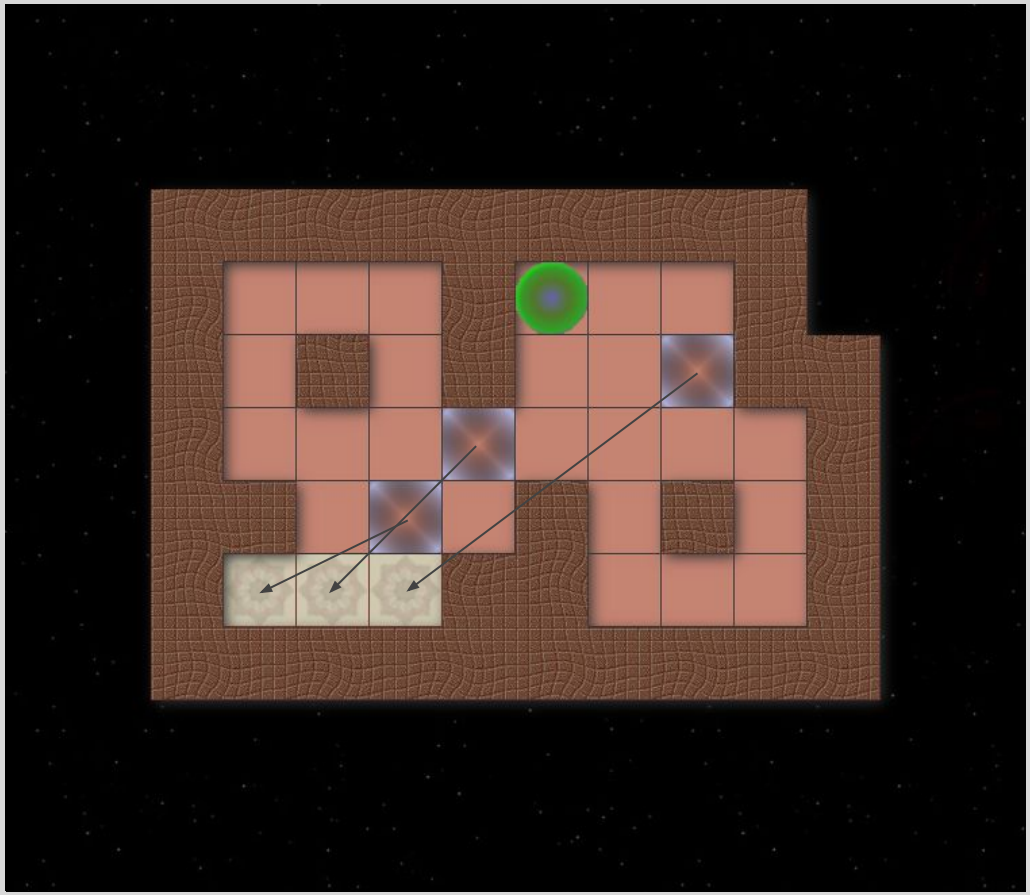
Los números representan la iteración en la que cada par fue asignado.

Asignación greedy

En primer lugar ordena las distancias entre cada par de caja y goal de menor a mayor en función de la distancia.

Luego hasta que la lista ordenada quede vacía se realiza la siguiente iteración: se toma el primer par y después se eliminan de la lista ordenada los pares que contengan una caja o goal que sea también contenida en el par que se sacó de la lista.

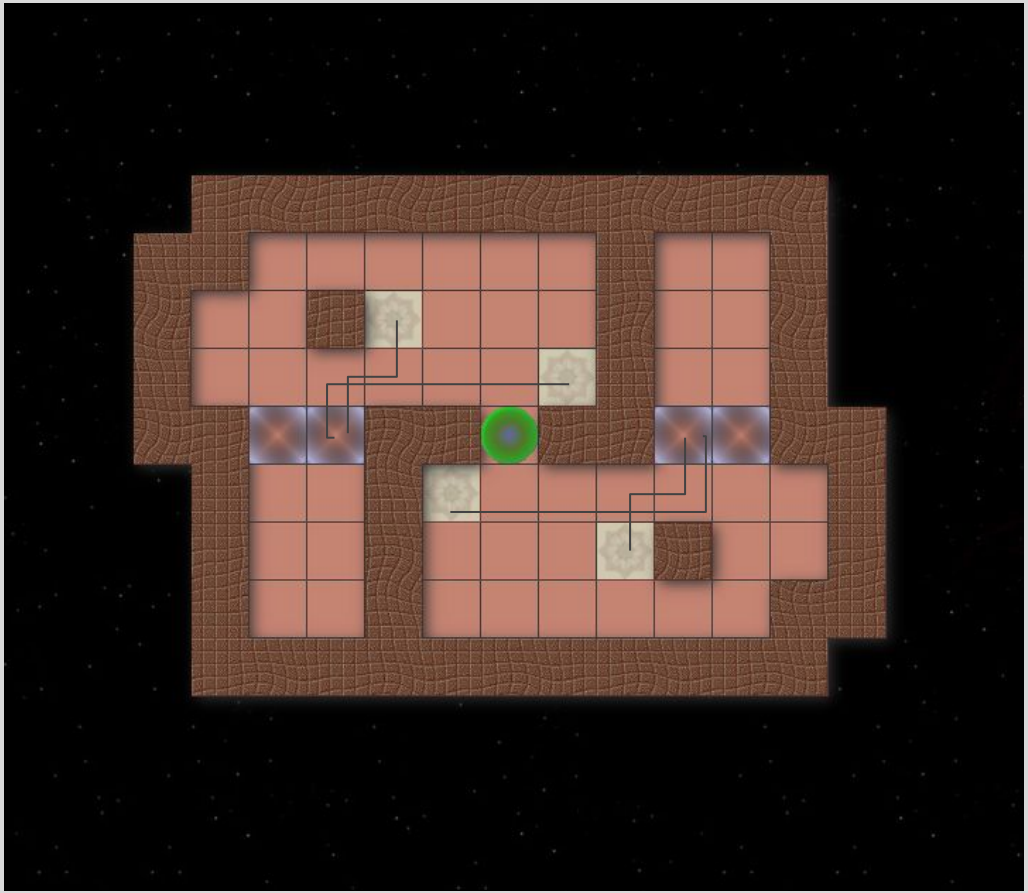
[2.3] Heurísticas de asignación



Asignación por fuerza bruta

Se prueban todas las combinaciones posibles entre cada caja y cada goal (tomando en cuenta que por cada goal hay una caja). Y nos quedamos con la combinación que resulte en la suma de distancias más pequeña.

[2.3] Heurísticas de asignación

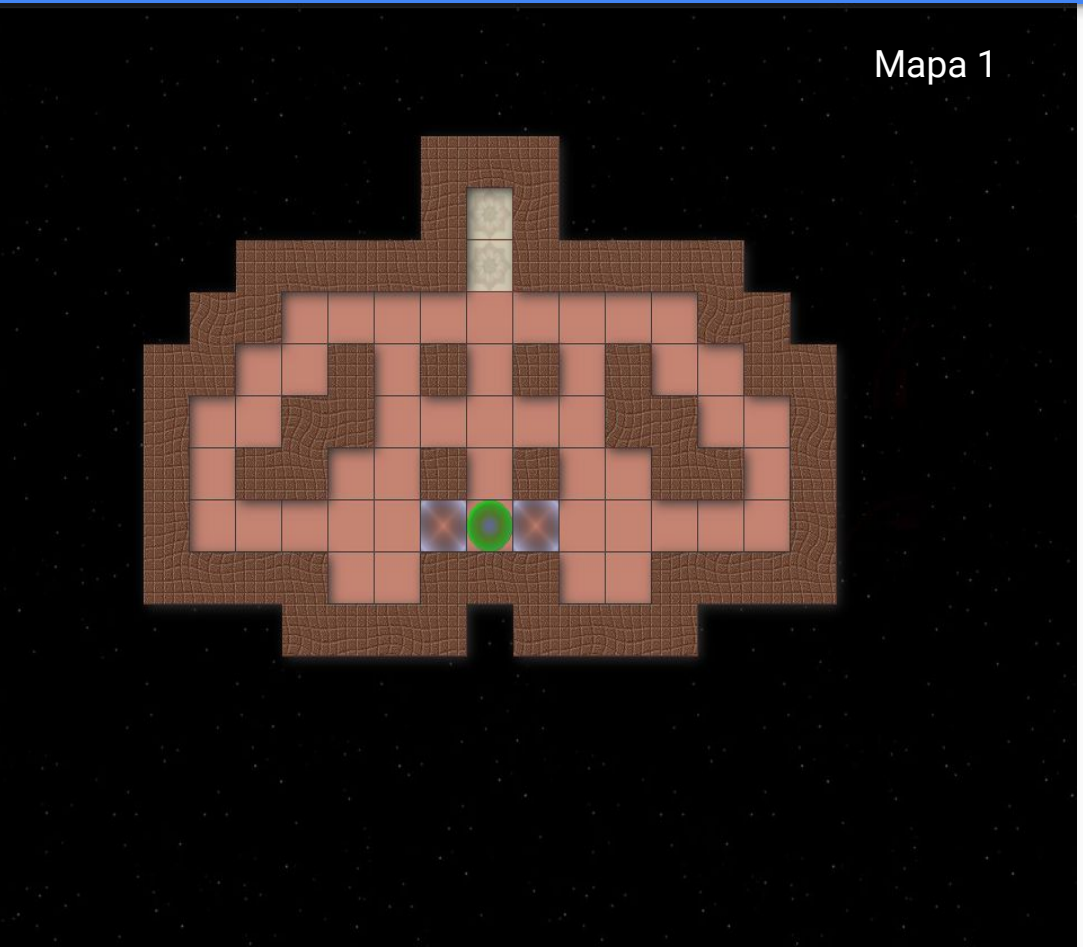


Asignación simple considerando paredes

Se encuentra para cada caja la distancia más cercana a un goal considerando las paredes. Múltiples cajas pueden compartir el mismo goal.

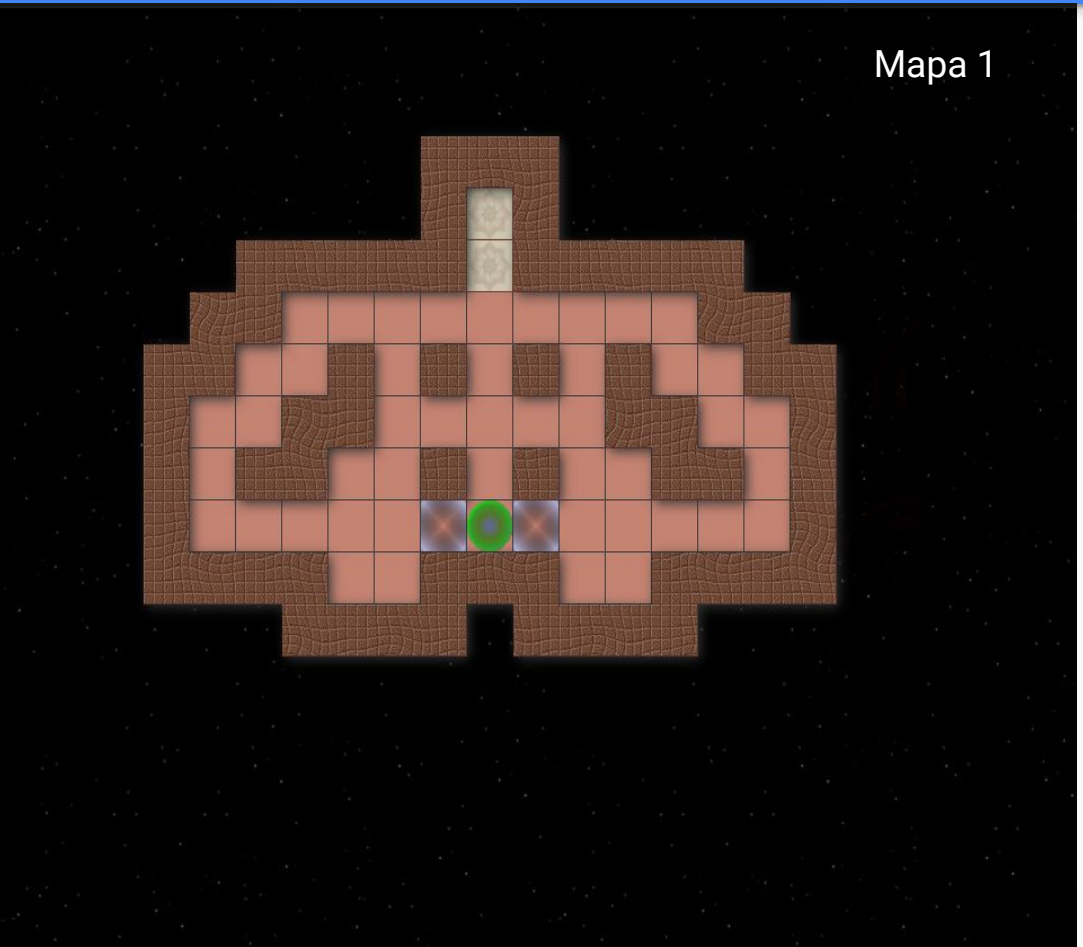
[4] Resultados

Comparación por tiempo



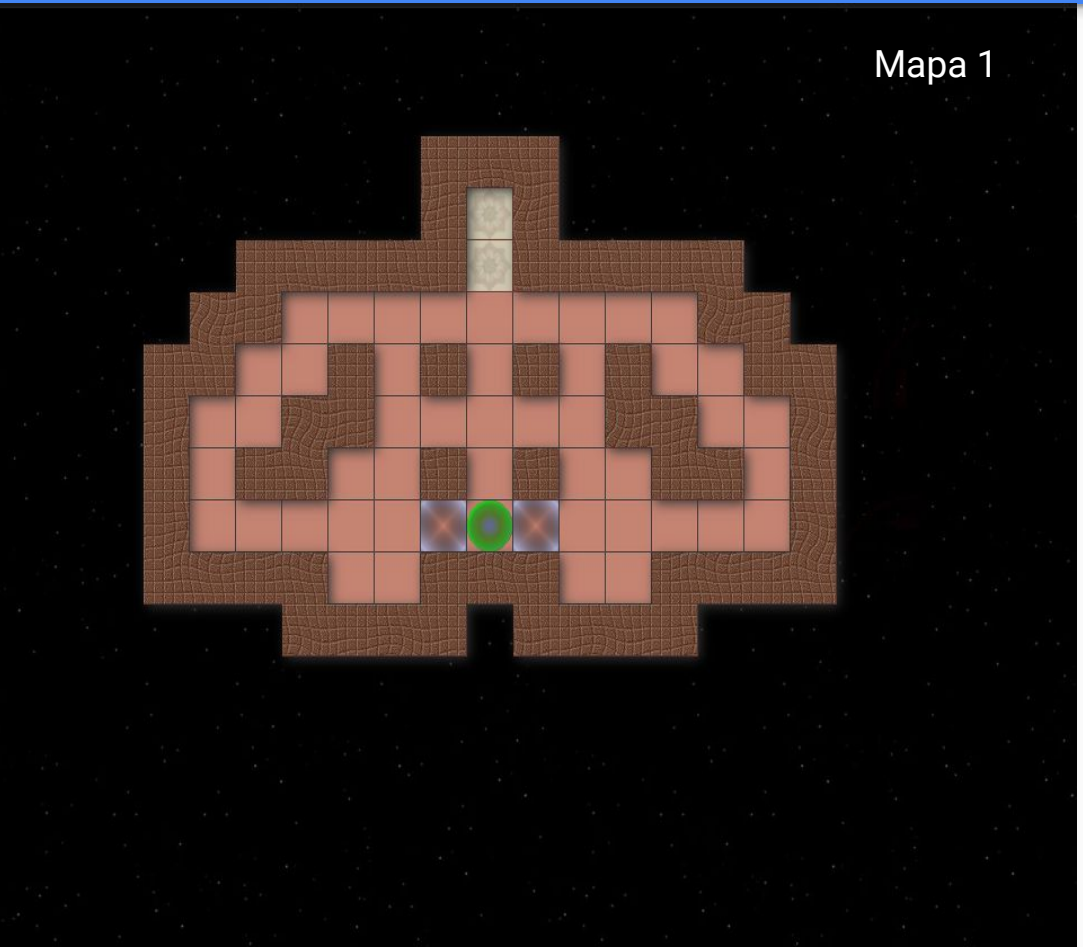
1. A* (simple) : 21 ms
2. Global Greedy (greedy): 15 ms
3. DFS: 65 ms
4. BFS : 169 ms
5. IDA*(bruteforce) : 5233 ms
6. IDDFS : 6487 ms

Comparación por nodos expandidos



1. Global Greedy (greedy): 480
2. A* (simple) : 789
3. DFS: 3223
4. BFS: 24125
5. IDA* (bruteforce) : 28207
6. IDDFS : 32102

Comparación por profundidad



1. A* (simple con paredes) : 78
2. BFS : 78
3. IDA* (todos) : 78
4. IDDFS : 78
5. DFS: 1628
6. Global Greedy (greedy): 114

[5] Conclusiones

[5] Conclusiones

- Greedy fue el más rápido.
- A* es un algoritmo balanceado.
- Se tuvo que encontrar una heurística que saque diferencia respecto del costo.