Aquarium

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Unidade Curricular: Inteligência Artificial

Grupo 36:
Diogo Filipe de Oliveira Santos
Jéssica Mireie Fernandes do Nascimento
João Vítor Freitas Fernandes

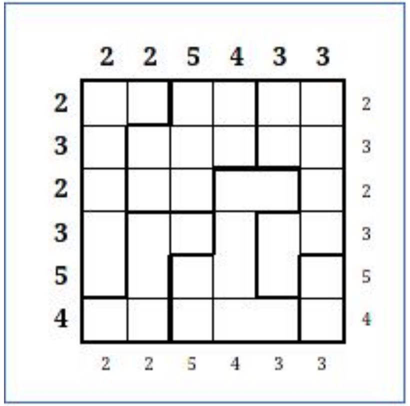
Link: https://www.puzzle-aquarium.com/

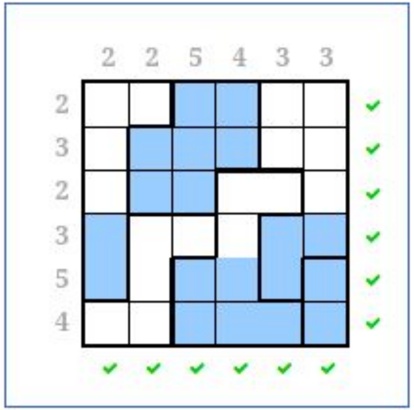
Aquarium

The puzzle is composed by a square grid with NxN dimension, divided into blocks called “aquariums”.

The goal is to “fill” them with water up to a certain level, which makes up the entire width of the aquarium, or leave it empty. An important detail is that you can’t “fill” a cell if any cell below from the same aquarium is empty.

Also, the numbers outside the grid shown are the total of cells per row or per column filled with water.





Formulation of the problem as search problem

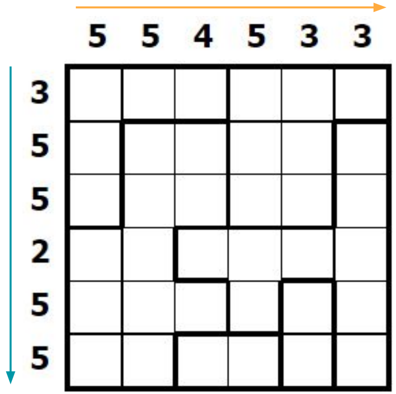
State Representation:

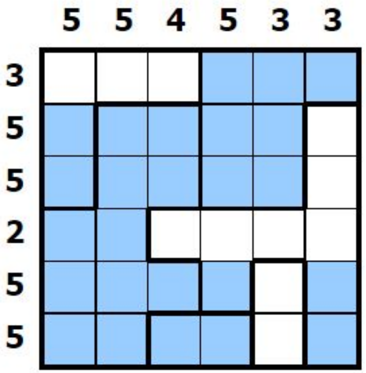
- ➔ **Aquarium**: Matrix NxN
  Each cell has an id representing the aquarium which they're in.
  If the number is negative, the cell is empty; if it is positive then the cell is full.
- ➔ **RowCap**: List with the number of cells for each row that must be filled
- ➔ **ColCap**: List with the number of cells for each column that must be filled

Initial State:

- ➔ Aquarium with negative values for each cell
- ➔ RowCap is organized from top to bottom
- ➔ ColCap is organized from left to right

Example:
Aquarium:[[-1, -1, -1, -2, -2, -2]
[-1, -3, -3, -2, -2, -4]
[-1, -3, -3, -2, -2, -4]
[-3, -3, -4, -4, -4, -4]
[-3, -3, -3, -4, -5, -4]
[-3, -3, -6, -6, -5, -4]]
RowCap: [3, 5, 5, 2, 5, 5]
ColCap: [5, 5, 4, 5, 3, 3]





Formulation of the problem as search problem

Objective Test:

➔ Fill each aquarium with water up until a certain level, keeping in mind all of the rows and columns capacities

```
# Counts how many cells are filled in row
def numCellsFilledInRow(aquarium, row):
    return len([l for l in aquarium[row] if l > 0])
```

```
# Counts how many cells are filled in col
def numCellsFilledInCol(aquarium, col):
    ret = 0
    for i in range(len(aquarium)):
            if aquarium[i][col] > 0:
                ret += 1
    return ret
```

```
def isObjective(node):
    aquarium = node.state.aquarium
    rowCap = node.state.rowCap
    colCap = node.state.colCap

    # verify if each row as reached the desired capacity
    for i in range(len(aquarium)):
        if rowCap[i] != numCellsFilledInRow(aquarium, i) or colCap[i] != numCellsFilledInCol(aquarium, i):
                return False

    return True
```

Formulation of the problem as search problem

Operator:

- Fill(node,aquariumID)
  - node: current state;
  - aquariumID: aquarium id to be filled

- Preconditions:

```
def canFillRow(self, cells):
    # Makes sure row cap isn't exceeded
    return len(self.getFullElemInRow(cells[0][0])) + len(cells) <= self.rowCap[cells[0][0]]

# Verifies if we can apply the operator
def preconditions(self, cells):
    return (self.canFillRow(cells)) and (self.canFillCol(cells)) # capacities not exceded
```

```
def canFillCol(self, cells):
    # Makes sure column cap isn't exceeded
    for cell in cells:
        col = self.getFullElemInCol(cell[1])
        if len(col) + 1 > self.colCap[cell[1]]:
            return False

    return True
```

- Effects:
  Negative Aquarium Id -> Positive Aquarium Id

```
def fillCells(self, cells):
    newAquarium = copy_list(self.node.state.aquarium)

    for [y,x] in cells:
        newAquarium[y][x] = self.aquariumID

    return newAquarium
```

- Cost:
  - Each fill operation has cost of 1

Implementation Work

Programming language: Python3

Development environment: Visual Studio/ Spyder

Data Structure:

- ➔ Priority Queue
- ➔ Queue
- ➔ Stack

File Structure:

- ➔ Data Structures
- ➔ Operators
- ➔ State
- ➔ Utils
- ➔ Interface

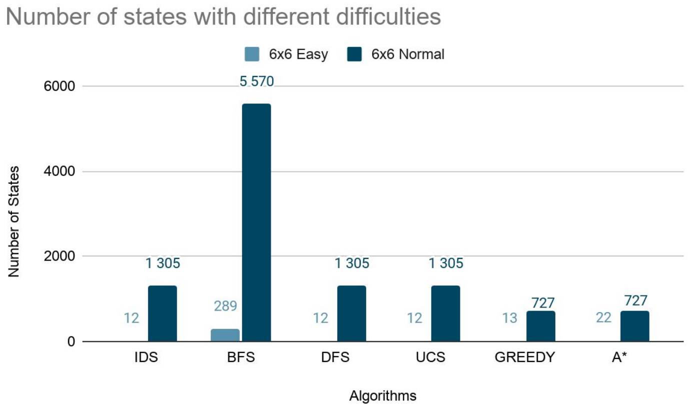
Heuristics

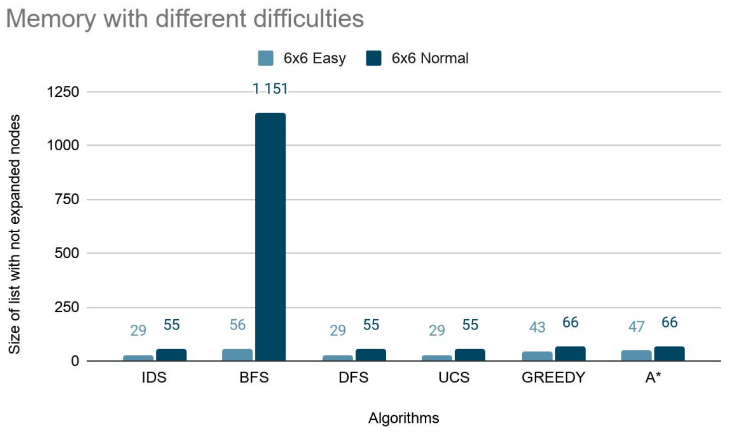
- **hRow():**
  - Returns the number of lines that don’t have the number of filled cells equal to the row capacity.
- **hCol():**
  - Returns the number of columns that don’t have the number of filled cells equal to the column capacity.
- **hRowCol():**
  - Returns the number of lines plus columns that don’t have the number of filled cells equal to the row and column capacity.
- **lineMoves():**
  - Returns the sum of the minimum number of movements needed to reach the row capacity.

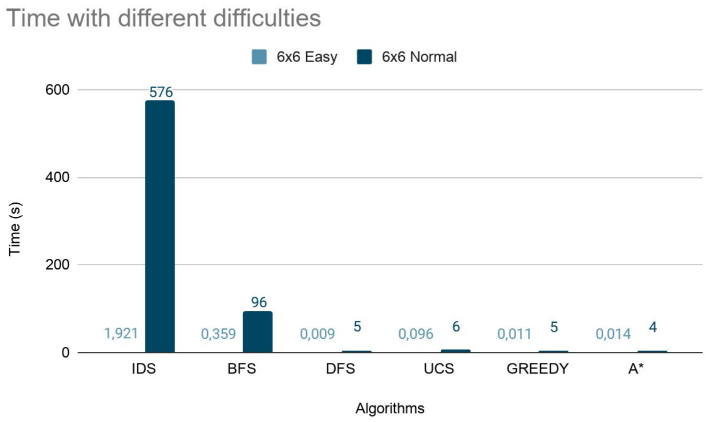
Algorithms Implemented

- Breadth First Search (BFS)
- Depth First Search (DFS)
- Uniform Cost Search (UCS)
- Iterative Deepening Search (IDS)
- Greedy Search
- A*

Algorithms Analysis

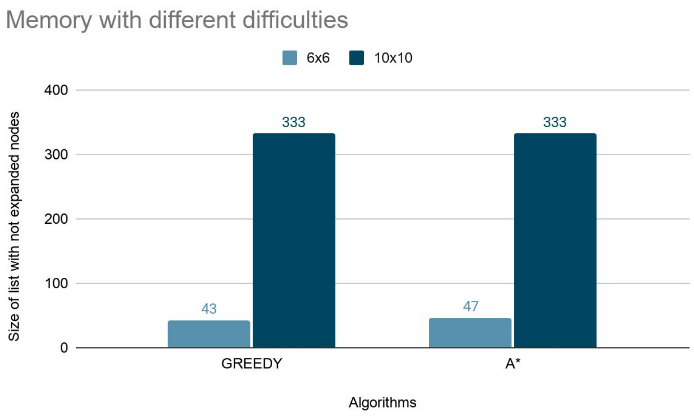


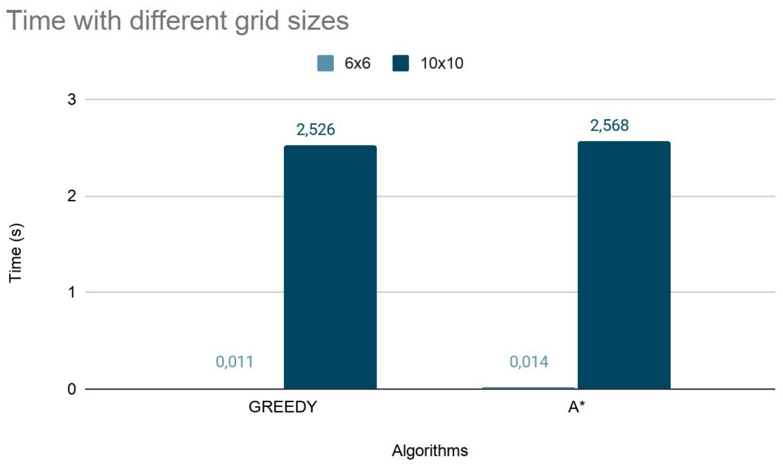


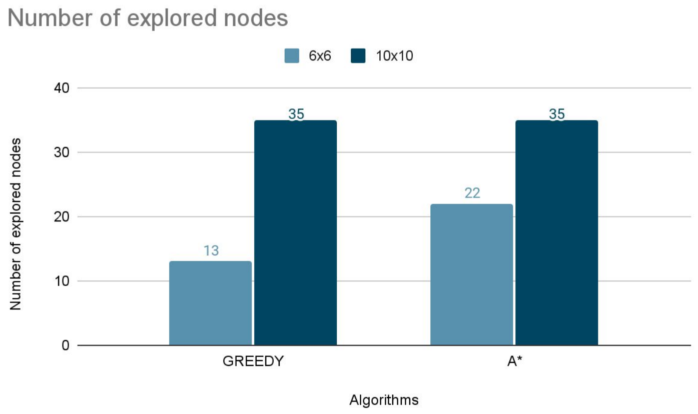


As observed, the more difficult the puzzle the more time, memory and nodes explored are consumed

Algorithms Analysis







As we can see, the bigger the puzzle the more time, memory and nodes explored are consumed

Conclusion

With this project, we conclude that for small and easy puzzles (low branching factor) the difference between blind and heuristic algorithms is minimal. Sometimes the blind algorithms perform faster due to their simplicity; however, as the difficulty increases the heuristic algorithms (greedy and A*) outperform the non-heuristic ones by a very large margin.