**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**

**TRƯỜNG ĐẠI HỌC BÁCH KHOA**

**KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH**

·▪•🙢🙞🕮🙜🙠•▪·



**Nhập môn trí tuệ nhân tạo (CO3061)**

**ASSIGNMENT 1**

**GVHD**: Vương Bá Thịnh

**Lớp**: L01

**SVTH:** 1914674 - Vũ Đình Phú

1912288 - Hồ Đức Trí

2011671 - Dương Đức Nghĩa

Tp. Hồ Chí Minh, Tháng 11/2022

**Mục lục**

[**1. WaterSort**](#_uowylk7ibshu) **2**

[1.1. Introduction](#_h3jo6t4hj322) 2

[1.2. Input and output format](#_oamp5cq9h86c) 3

[1.2.1. Input](#_77pmtc94ksy3) 3

[1.2.2. Output](#_v63do23hepmy) 3

[1.3. Algorithms used](#_4tkgaltdjr85) 5

[1.3.1. DFS](#_cae0t4c3gof9) 5

[1.3.2. A\*](#_qsdr2s811dyf) 6

[1.3.3. Algorithm comparison](#_6ejsluhf5yfu) 8

[**2. Bloxorz**](#_pyioocc6ugi1) **10**

[2.1. Introduction](#_z3b770xb403a) 10

[2.2. Input and output format](#_e3o2q5jqvpef) 11

[2.2.1. Input](#_2u11p88k9ma0) 11

[2.2.1. Output](#_3rnh0lignpz3) 11

[2.3. Algorithms used](#_aojat8sqf947) 11

[2.3.1. BFS](#_orzlimpvn77n) 11

[2.3.2. Genetic](#_9l46g7np8h1n) 12

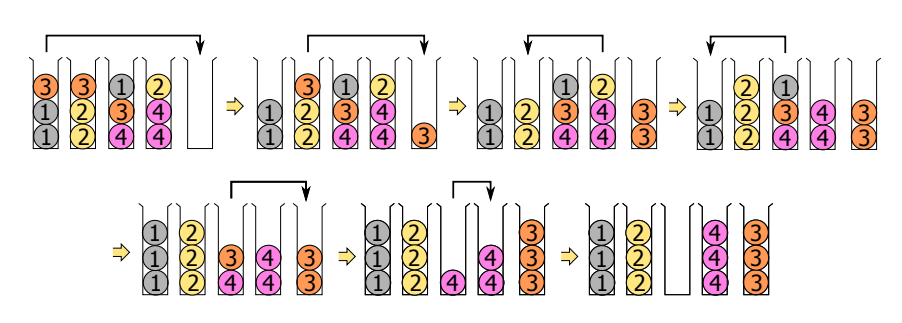
[2.3.3. Algorithm comparison](#_hukdur70namb) 15

# 1. WaterSort

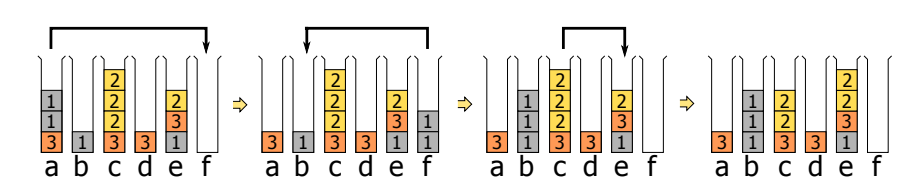
## 1.1. Introduction

The ball sort puzzle and the water sort puzzle are popularized recently via smartphone apps.1 Both puzzles involve bins filled with some colored units (balls or water), and the goal is to somehow sort them. The most significant feature of these puzzles is that each bin works as a stack. That is, the items in a bin have to follow the “last-in first-out” (LIFO) rule.

In the ball sort puzzle, we are given *hn* colored balls in *n* bins of capacity *h* and *k* additional empty bins. For a given (unsorted) initial configuration, the goal of this puzzle is to arrange the balls in order; that is, to make each bin either empty or full with balls of the same color. (The ordering of bins does not matter in this puzzle.) The rule of this puzzle is simple: (0) Each bin works like a stack, that is, we can pick up the top ball in the bin. (1) We can move the top ball of a bin to the top of another bin if the second bin is empty or it is not full and its top ball before the move and the moved ball have the same color. An example with *h* = 3, *n* = 4, and *k* = 1 is given in figure below.



The water sort puzzle is similar to the ball sort puzzle. Each ball is replaced by colored water of a unit volume in the water sort puzzle. In the water sort puzzle, the rules (0) and (1) are the same as the ball sort puzzle except one liquid property: Colored water units are merged when they have the same color and they are consecutive in a bin. When we pick up a source bin and move the top water unit(s) to a target bin, the quantity of the colored water on the top of the bin to be moved varies according to the following conditions (figure below). If the target bin has enough margin, all the water of the same color moves to the target bin. On the other hand, a part of the water of the same color moves up to the limit of the target



## 

## 1.2. Input and output format

### 1.2.1. Input

Our group maps the problem in the game WaterSort to Input in the form of a list as follows:

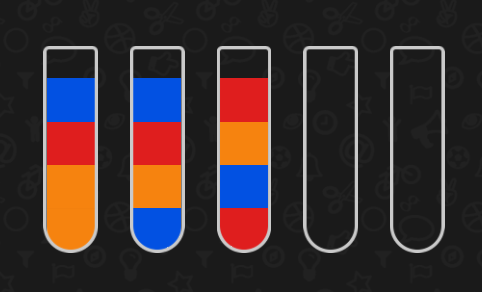
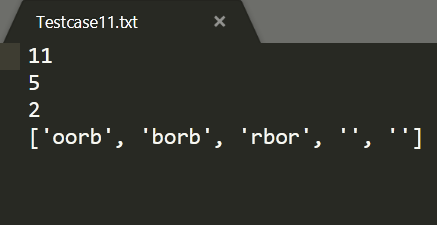
['aabb', 'abba', '']

Specifically, this list will contain elements that are Strings, each String representing a jar in the problem. The characters in each String will represent the colors in the original problem, respectively. The first characters in the String correspond to the colors at the bottom of the jar, and the characters at the end of the String correspond to the colors at the top of the jar. The empty jar will be represented by the empty String.

In addition to get more information for problem solving. Our text Input file also has additional lines for additional information. That is:

* Line 1 is the Level of the problem. This Level will usually be proportional to the difficulty of the Input
* Line 2 is the total number of jars contained in the problem
* Line 3 is the number of empty jars at the beginning of the problem
* As for line 4, the last line in the Input file is the List of problem mapping as we described above

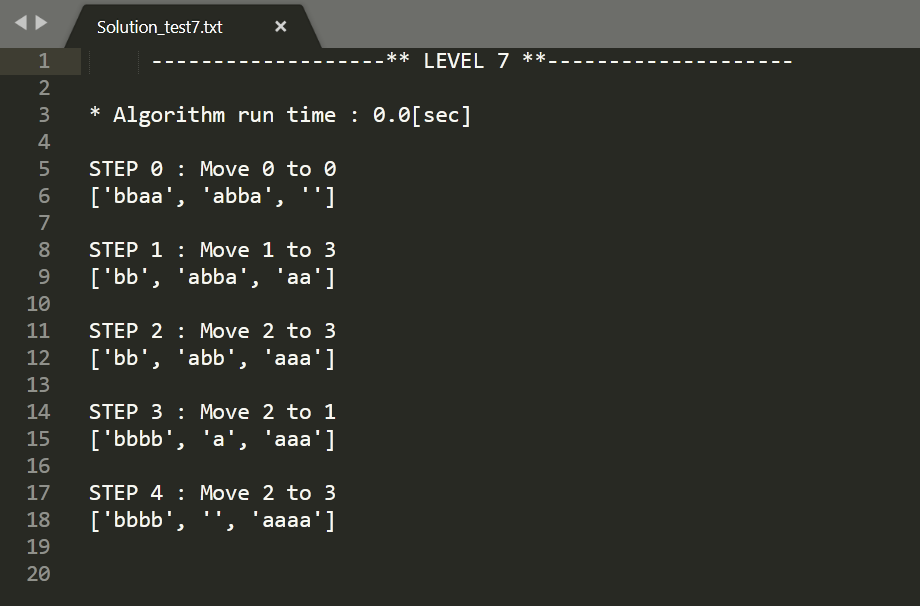
For example, with the problem as shown in the image below, it will be mapped in the Input File as:



**→**

### 1.2.2. Output

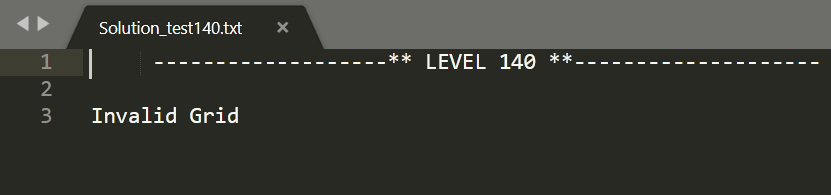
For solvable problems, the Output File will have the following format:



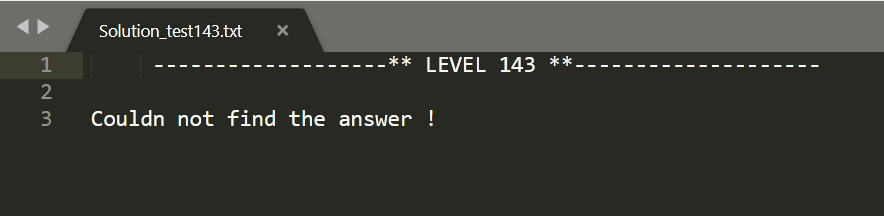
In there :

* "Algorithm run time": Indicates the running time of the algorithm to solve the problem. Measured in seconds
* "STEP 1: Move 1 to 3": Pour jar 1 to jar 3. Give details of each step to take to solve the problem.
* "['bb', 'abba', 'aa']" : the status of the jars after performing the above action.

For problems with wrong input data, the File Output will have the following format:



For unsolvable problems, the File Output will have the following format:



## 

## 1.3. Algorithms used

### 1.3.1. DFS

A standard DFS implementation puts each vertex of the graph into one of two categories:

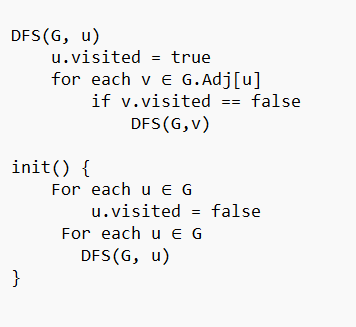
* Visited
* Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

* Start by putting any one of the graph's vertices on top of a stack.
* Take the top item of the stack and add it to the visited list.
* Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
* Keep repeating steps 2 and 3 until the stack is empty.

Pseudo code:



Backtracking is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

In the water sort problem, we will create a backtracking algorithm combined with DFS to solve it. When applied to solve this problem, we will replace vertices with state. We define the state of this problem as a sorted concatenated string of color in the stack. For example: if the current stack 1 is ‘aabb’, stack 2 is ‘baab’, stack 3 is empty then the current state is ‘;aabb;baab;’ (each stack is separated with a semicolon, the order is stack 3, stack 1, stack 2).

* First, create a *visited* set which will save the visited states, the input is an initial state.
* Next, we will inspect the next valid move, change the state, save it into *visited* and check if the problem is solved.
* If it is, stop the algorithm and return the answer.
* If not, start recursion with the next valid state.
* If the algorithm has check all existed state and the problem is still not solved, stop the algorithm and conclude “Could not find the answer”

### 1.3.2. A\*

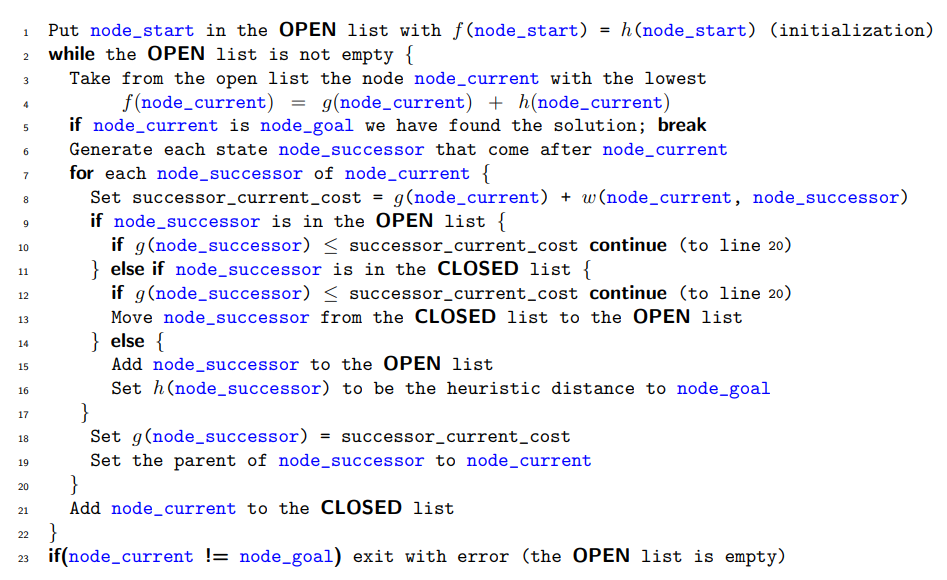
A\* Algorithm Basic Concepts:

* A\* search algorithm is an algorithm that separates it from other traversal techniques. This makes A\* smart and pushes it much ahead of conventional algorithms.
* A\* algorithm works based on heuristic methods, and this helps achieve optimality. A\* is a different form of the best-first algorithm. Optimality empowers an algorithm to find the best possible solution to a problem. Such algorithms also offer completeness; if there is any solution possible to an existing problem, the algorithm will definitely find it.
* When A\* enters into a problem, firstly, it calculates the cost to travel to the neighboring nodes and chooses the node with the lowest cost. If The f(n) denotes the cost, A\* chooses the node with the lowest f(n) value. Here ‘n’ denotes the neighboring nodes. The calculation of the value can be done as shown below:

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

***g(n)*** = shows the shortest path’s value from the starting node to node n

***h(n)*** = The heuristic approximation of the value of the node



**Figure 1**

Apply to WaterSort problem:

Heuristic function definition:

Heuristic function is used to calculate the value of *h(n).*

* The value of the Heuristic function at the target state is 0. The smaller the value of the Heuristic function, the closer it is to the target state.
* Heuristic function is calculated by the formula:

***h(n) = different\_Color\_point + 5\*bonus\_Stack\_used***

* Where the variable *"different\_Color\_point"* is calculated by determining the color pairs that are next to each other in each tube. If 2 colors next to each other are different, then *"different\_Color\_point = different\_Color\_point + 500"* means increase the variable *"different\_Color\_point"* by 500 units. The purpose of this variable is to evaluate the goodness of a state.
* The variable *"bonus\_Stack\_used"* is used to calculate the number of Stacks that we are using in the *"bonusStack"*. The purpose of *"bonus\_Stack\_used"* is to reduce the number of pipes used in a state as small as possible to quickly get to the target state.
* The value of *"different\_Color\_point"* grows 100 times faster than *"bonus\_Stack\_used"* because we are prioritizing solving the problem of color difference over the problem of using the fewest tubes. The reason for choosing an increment of 500 for the variable *"different\_Color\_point"* is because in the function *f(n) = g(n) + h(n)* where *g(n)* is the number of steps taken, in complex cases the number of steps need to go to solve a problem that can be > 100 steps. Meanwhile we need the variable *"different\_Color\_point"* to have the highest priority in the function *f(n)* because it is the deciding factor needed to consider the *"open\_set"* queue to quickly arrive at the solution, reducing the time and the number of states to consider. So *"different\_Color\_point"* needs to have the highest weight.

Self-assessment of the group's heuristic function and explanation:

* *Admissibility*: Satisfy the criteria for Admissibility because it can effectively estimate the real distance between a node ‘n’ and the end node. It never overestimates; if it ever does, it will be denoted by ‘d’, which also denotes the accuracy of the solution.
* Consistency: Not completely satisfied because the Heuristic function is not fully optimized. In some cases the estimate of a given heuristic function turns out to be not equal to or more than the distance between the goal (n) and a neighbor and the cost calculated to reach that neighbor. The team evaluates this as a point to improve in this Heuristic function.

Apply the algorithm to the problem:

To apply the A\* algorithm to the problem. We perform the following main steps:

* Convert the problem into a path-finding graph by treating the states as nodes of the graph.
* Define information for each node including: current state, number of steps passed, source stack, destination stack.
* Write a function *"get\_neighbors"* to determine a set of adjacent nodes to consider. Since this is a cyclic graph, it is possible to repeat the states considered, so we add a *"visited"* list to save all the states that have been passed.
* Finally, apply the code according to the pseudocode in *Figure 1* with some small changes to arrive at the solution of the problem.

### **1.3.3.** A**lgorithm** comparison

**Performance comparison**

Valid test case

| **Level** | **DFS** | | **AStar** | |
| --- | --- | --- | --- | --- |
|  | **Steps** | **Time** | **Steps** | **Time** |
| 10  ['aaab', 'bccc', 'babc', '', ''] | 9 | 0.0s | 7 | 0.0009s |
| 20  ['bbdc', 'dcbb', 'cada', 'dcaa', '', ''] | 14 | 0.0s | 12 | 0.0039s |
| 30  ['abcb', 'abce', 'debc', 'edaa', 'ecdd', '', ''] | 19 | 0.0009s | 15 | 0.0079s |
| 50  ['cbcg', 'gacd', 'fecf', 'aegb', 'edae', 'bfdf', 'bdag', '', ''] | 27 | 0.0009s | 25 | 0.0159s |
| 70  ['hafe', 'bbbb', 'gaeg', 'icid', 'fcae', 'hhfc', 'ecgg', 'difi', 'ddah', '', ''] | 26 | 0.0009s | 26 | 0.0448s |
| 100  ['fabh', 'agdd', 'ehlc', 'hgab', 'khij', 'lief', 'ecjl', 'iegi', 'bjkd', 'dkjb', 'ffcg', 'kalc', '', ''] | 45 | 0.0538s | 42 | 1.3683s |
| 130  ['bcoi', 'kina', 'ahoo', 'ndlm', 'donk', 'cebh', 'jhjm', 'ifan', 'gemk', 'hfld', 'clbg', 'jaml', 'igjk', 'cfef', 'gdeb', '', ''] | 58 | 0.0817s | 55 | 105.54s |

Invalid test case

| **Level** | **DFS** | | **AStar** | |
| --- | --- | --- | --- | --- |
|  | **Steps** | **Time** | **Steps** | **Time** |
| 140  ['abcb', 'abce', 'debc', 'eda', 'ecdd', '', '']  (stack 4 has 3 level) |  | 0.0s |  | 0.0s |
| 142  ['abcb', 'ab', 'debc', 'ed', 'ecd', '', '']  (stack 2, 4 have 2 level, stack 3 has 3 level) |  | 0.0s |  | 0.0s |
| 145  ['abcb', 'abce', 'debb', 'edaa', 'ecdd', '', '']  (color ‘b’ has 5 level ) |  | 0.0s |  | 0.0s |
| 147  ['rjis', 'untl', 'wfdm', 'daiv', 'dork', 'ripl', 'tsaq', 'fghl', 'oagk', 'hbtu', 'omkw', 'fnej', 'qide', 'bcpn', 'pjah', 'vsom', 'ufrq', 'vgkw', 'lhwb', 'sbce', 'jtce', 'uqgc', 'vnyzpm', '']  (stack 23 has 6 level) |  | 0.0s |  | 0.0s |

Explain: Having a pre-check function helps detecting invalid grid efficiently.

**Conclusion**

Pathfinding problems like Water Sort are found to be easily solvable by searching algorithms and their performance differs depending on the search strategy being used and the strategy of the algorithm itself. In this paper, a single agent was implemented to solve Water Sort games using Depth First Search (DFS) and AStar (A\*) searching algorithms. AStar algorithm will most likely have a number of step smaller or equal DFS algorithms (optimal solution). On the other hand, DFS is not guaranteed to find the optimal path but it sometimes saves time and gives better performance than AStar. DFS algorithm cannot always find the optimal path. Its strategy depends on finding the first correct path it comes across. DFS can reach the goal by the optimal path only if the optimal path is the first explored path the algorithm came across.

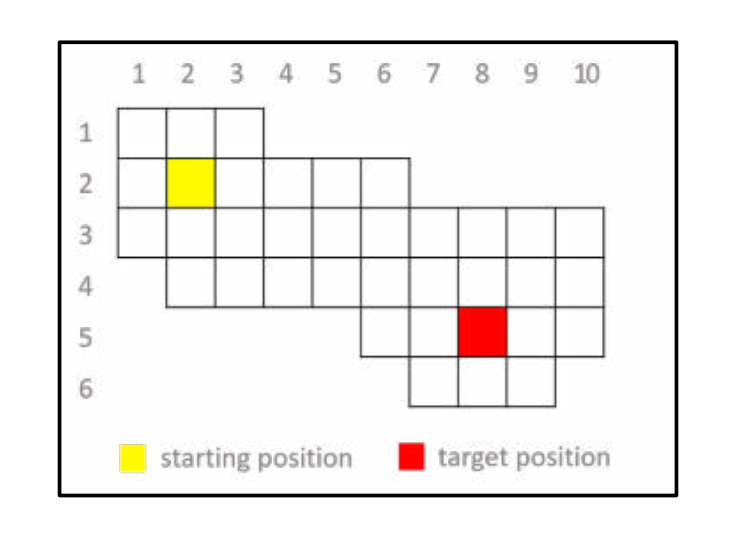
# 2. Bloxorz

## 2.1. Introduction

Bloxorz is a block sliding puzzle game that can be categorized as a pathfinding problem. Pathfinding problems are well known problems in Artificial Intelligence field. In this paper, we proposed a single agent implementation to solve level-1 of Bloxorz game using these algorithms: Breath-First Search (BFS), and Genetic algorithms. The agent solves the problem using the three algorithms to compare their performance and conduct a conclusion that may help in improving the use of searching algorithms in this area. In this paper, breadth-first search algorithms are found to be more convenient to solve this problem.

Bloxorz game is a path-finding problem or can be described as a block sliding puzzle game. This problem can be solved by a single agent based on searching algorithms. In this paper, an implementation of Bloxorz level-1 solver agent is proposed by three searching algorithms: Breadth-first search (BFS), Genetic. The paper aims to compare the results of the two algorithms.

Bloxorz is a 3-D block sliding puzzle game consisting of a terrain that is built by 1×1 tile with a special shape and size, and a 1×1×2 size block. This game is a single agent path-finding problem that involves moving the block from its initial position using four directions (right, left, up, and down) and ensuring that its ends are always within the terrain boundary, until it falls into a 1×1 square hole in the terrain that represents our goal state. The block can be in three states, standing, lying horizontally, and lying vertically. When the block reaches the hole, it must be in a standing state to fall in it.



## 2.2. Input and output format

### **2.2.1. Input**

### **2.2.1. Output**

## 2.3. Algorithms used

### **2.3.1. BFS**

A standard BFS implementation puts each vertex of the graph into one of two categories:

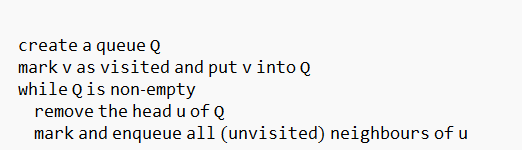
* Visited
* Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The BFS algorithm works as follows:

* Start by putting any one of the graph's vertices at the back of a queue.
* Take the front item of the queue and add it to the visited list.
* Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
* Keep repeating steps 2 and 3 until the queue is empty.

Pseudo code:



In this search strategy, the “Expand” data structure is a simple FIFO (first in first out) queue. When a current node gets expanded, its children get inserted at the end of the queue and when polling out, always pull out the node at the first. This strategy searches in a wide development direction. Pseudocode of the algorithm is as follows:

* Step 1: Predefine the initial node (initial position of the block) and the goal node (position of the hole).
* Step 2: Initialize the MoveSequence string of the initial node to empty.
* Step 3: Add the initial node into the Expand queue and the Visited list.
* Step 4: Check if the queue is empty → there is no solution and exit, otherwise continue.
* Step 5: Poll out the first node of the Expand queue (called parent node).
* Step 6: Check if it is equal to the goal node (block position defined in this node equals hole position defined in the goal node) → solution is found and return the MoveSequence string of that node, otherwise continue.
* Step 7: Expand the node and generate its child nodes which represent the new positions of the block after moving in all 4 directions from the current position defined in the parent node.
* Step 8: For each of the generated nodes, if it is valid (block position is valid in the terrain), its MoveSequence equals to its parent’s plus the move it generated from, add the node to the Expand queue and the Visited list. Finally go back to step 4.

### 2.3.2. Genetic

Genetic Algorithms(GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. They are commonly used to generate high-quality solutions for optimization problems and search problems.

Genetic algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to next generation. In simple words, they simulate “survival of the fittest” among individual of consecutive generation for solving a problem. Each generation consist of a population of individuals and each individual represents a point in search space and possible solution. Each individual is represented as a string of character/integer/float/bits. This string is analogous to the Chromosome.

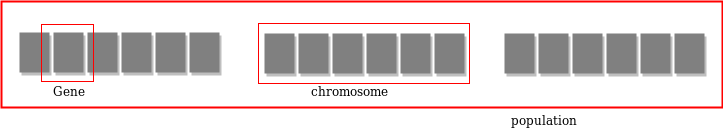
**Foundation of Genetic Algorithms**

Genetic algorithms are based on an analogy with genetic structure and behaviour of chromosomes of the population. Following is the foundation of GAs based on this analogy –

1. Individual in population compete for resources and mate
2. Those individuals who are successful (fittest) then mate to create more offspring than others
3. Genes from “fittest” parent propagate throughout the generation, that is sometimes parents create offspring which is better than either parent.
4. Thus each successive generation is more suited for their environment.

**Search space**

The population of individuals are maintained within search space. Each individual represents a solution in search space for given problem. Each individual is coded as a finite length vector (analogous to chromosome) of components. These variable components are analogous to Genes. Thus a chromosome (individual) is composed of several genes (variable components).



**Fitness Score**

A Fitness Score is given to each individual which shows the ability of an individual to “compete”. The individual having optimal fitness score (or near optimal) are sought.

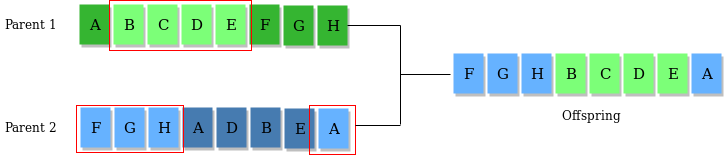
The GAs maintains the population of n individuals (chromosome/solutions) along with their fitness scores.The individuals having better fitness scores are given more chance to reproduce than others. The individuals with better fitness scores are selected who mate and produce better offspring by combining chromosomes of parents. The population size is static so the room has to be created for new arrivals. So, some individuals die and get replaced by new arrivals eventually creating new generation when all the mating opportunity of the old population is exhausted. It is hoped that over successive generations better solutions will arrive while least fit die.

Each new generation has on average more “better genes” than the individual (solution) of previous generations. Thus each new generations have better “partial solutions” than previous generations. Once the offspring produced having no significant difference from offspring produced by previous populations, the population is converged. The algorithm is said to be converged to a set of solutions for the problem.

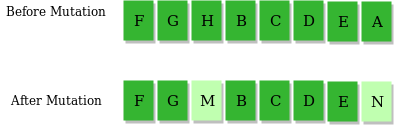
**Operators of Genetic Algorithms**

Once the initial generation is created, the algorithm evolves the generation using following operators –

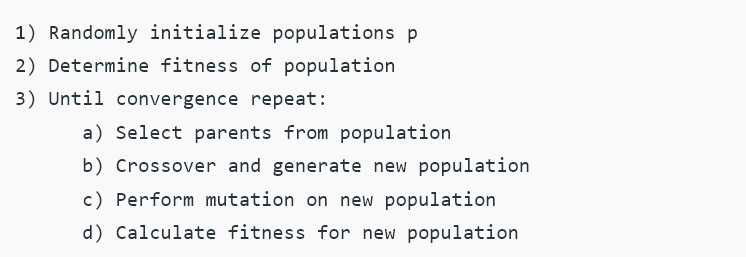
1. Selection Operator: The idea is to give preference to the individuals with good fitness scores and allow them to pass their genes to successive generations.
2. Crossover Operator: This represents mating between individuals. Two individuals are selected using selection operator and crossover sites are chosen randomly. Then the genes at these crossover sites are exchanged thus creating a completely new individual (offspring). For example



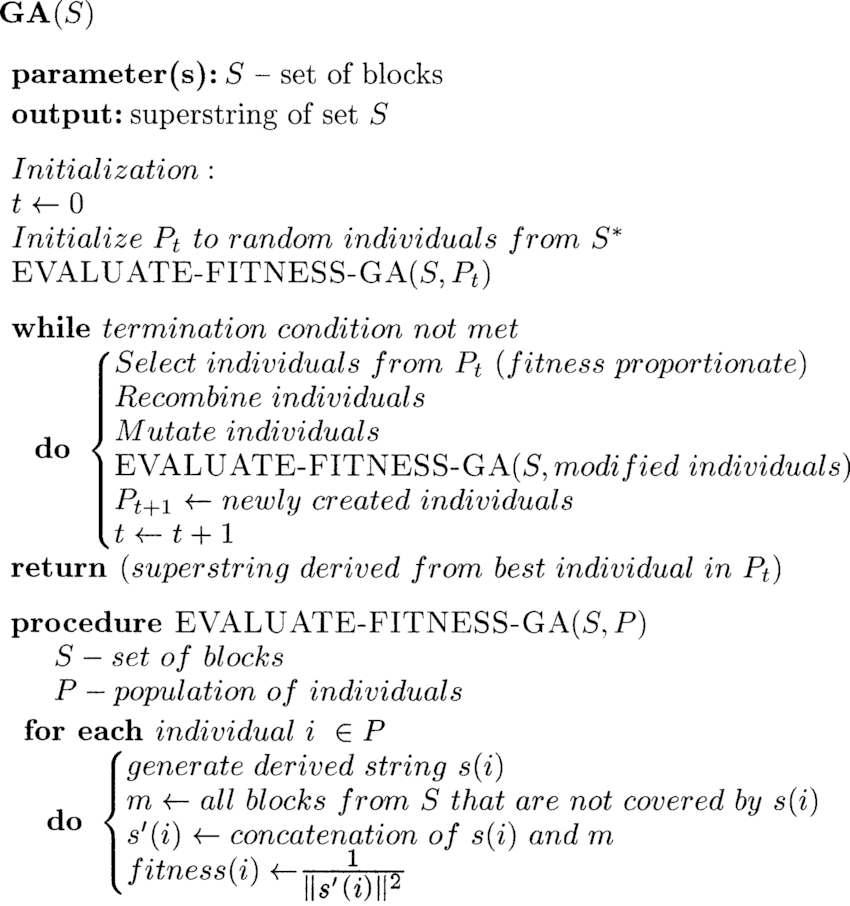
1. Mutation Operator: The key idea is to insert random genes in offspring to maintain the diversity in the population to avoid premature convergence. For example –



*The whole algorithm can be summarized as –*



Pseudo code:



### 

### 2.3.3. Algorithm comparison