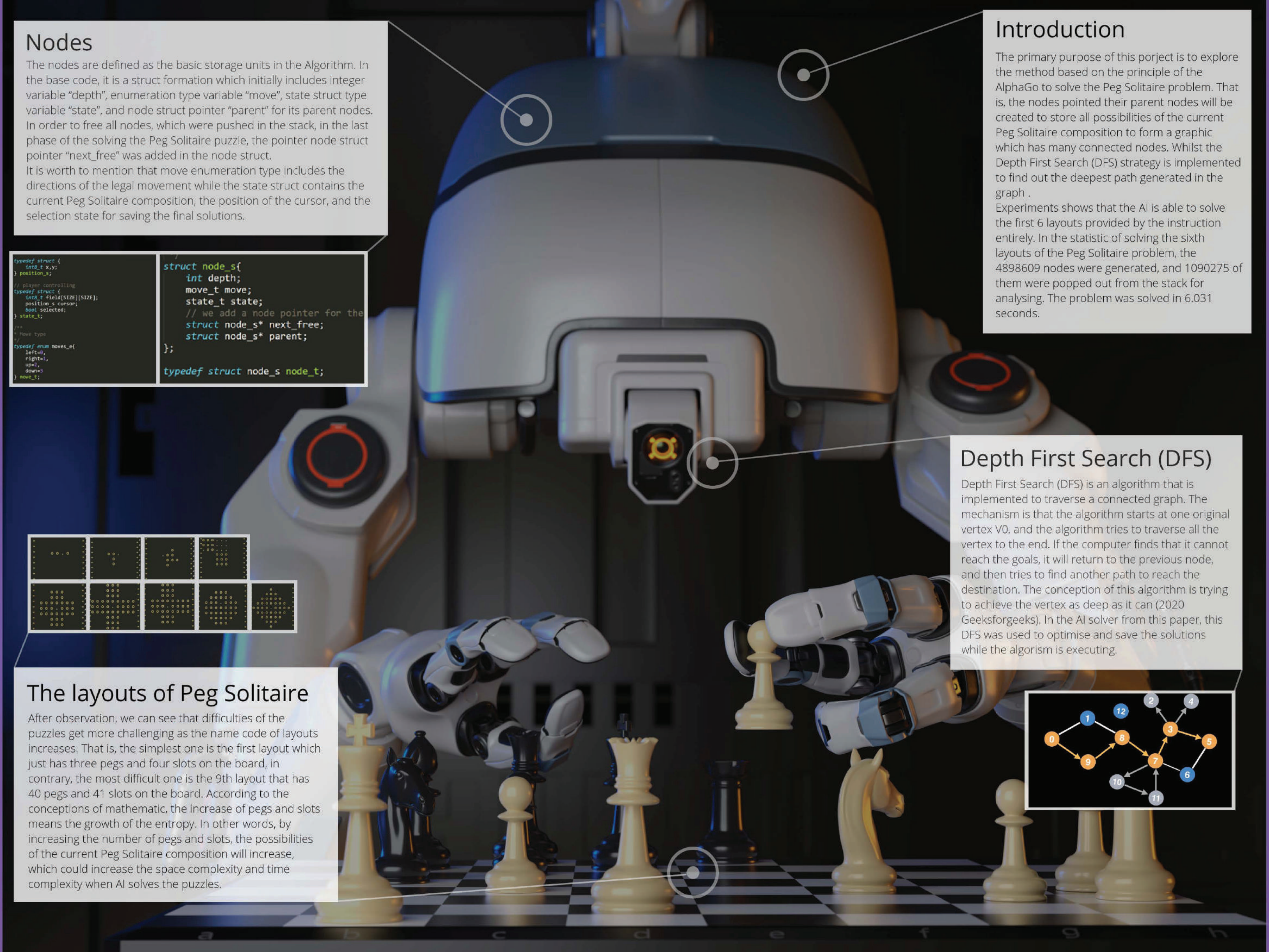
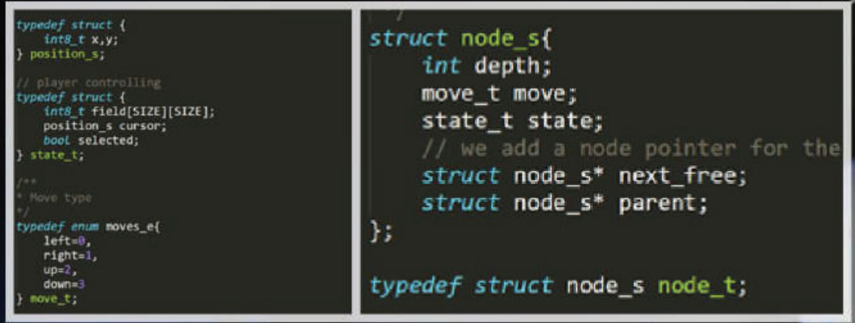
# PEG SOLITAIRE AI SOLVER

Artificial Intelligence Software Program Development



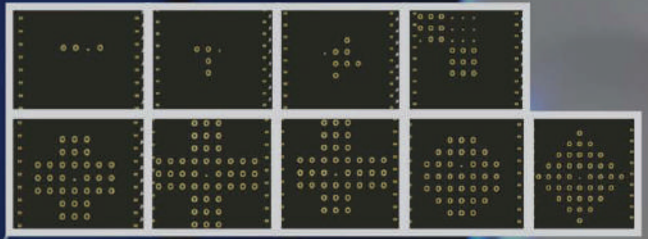
## Nodes

The nodes are defined as the basic storage units in the Algorithm. In the base code, it is a struct formation which initially includes integer variable "depth", enumeration type variable "move", state struct type variable "state", and node struct pointer "parent" for its parent nodes. In order to free all nodes, which were pushed in the stack, in the last phase of the solving the Peg Solitaire puzzle, the pointer node struct pointer "next_free" was added in the node struct.
It is worth to mention that move enumeration type includes the directions of the legal movement while the state struct contains the current Peg Solitaire composition, the position of the cursor, and the selection state for saving the final solutions.



```
typedef struct {
    int8_t x,y;
} position_s;

// player controlling
typedef struct {
    int8_t field[SIZE][SIZE];
    position_s cursor;
    bool selected;
} state_t;

/**
* Move type
*/
typedef enum moves_e{
    left=0,
    right=1,
    up=2,
    down=3
} move_t;
```

```
struct node_s{
    int depth;
    move_t move;
    state_t state;
    // we add a node pointer for the
    struct node_s* next_free;
    struct node_s* parent;
};

typedef struct node_s node_t;
```



## The layouts of Peg Solitaire

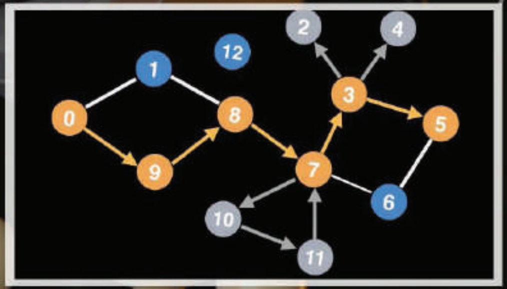
After observation, we can see that difficulties of the puzzles get more challenging as the name code of layouts increases. That is, the simplest one is the first layout which just has three pegs and four slots on the board, in contrary, the most difficult one is the 9th layout that has 40 pegs and 41 slots on the board. According to the conceptions of mathematic, the increase of pegs and slots means the growth of the entropy. In other words, by increasing the number of pegs and slots, the possibilities of the current Peg Solitaire composition will increase, which could increase the space complexity and time complexity when AI solves the puzzles.

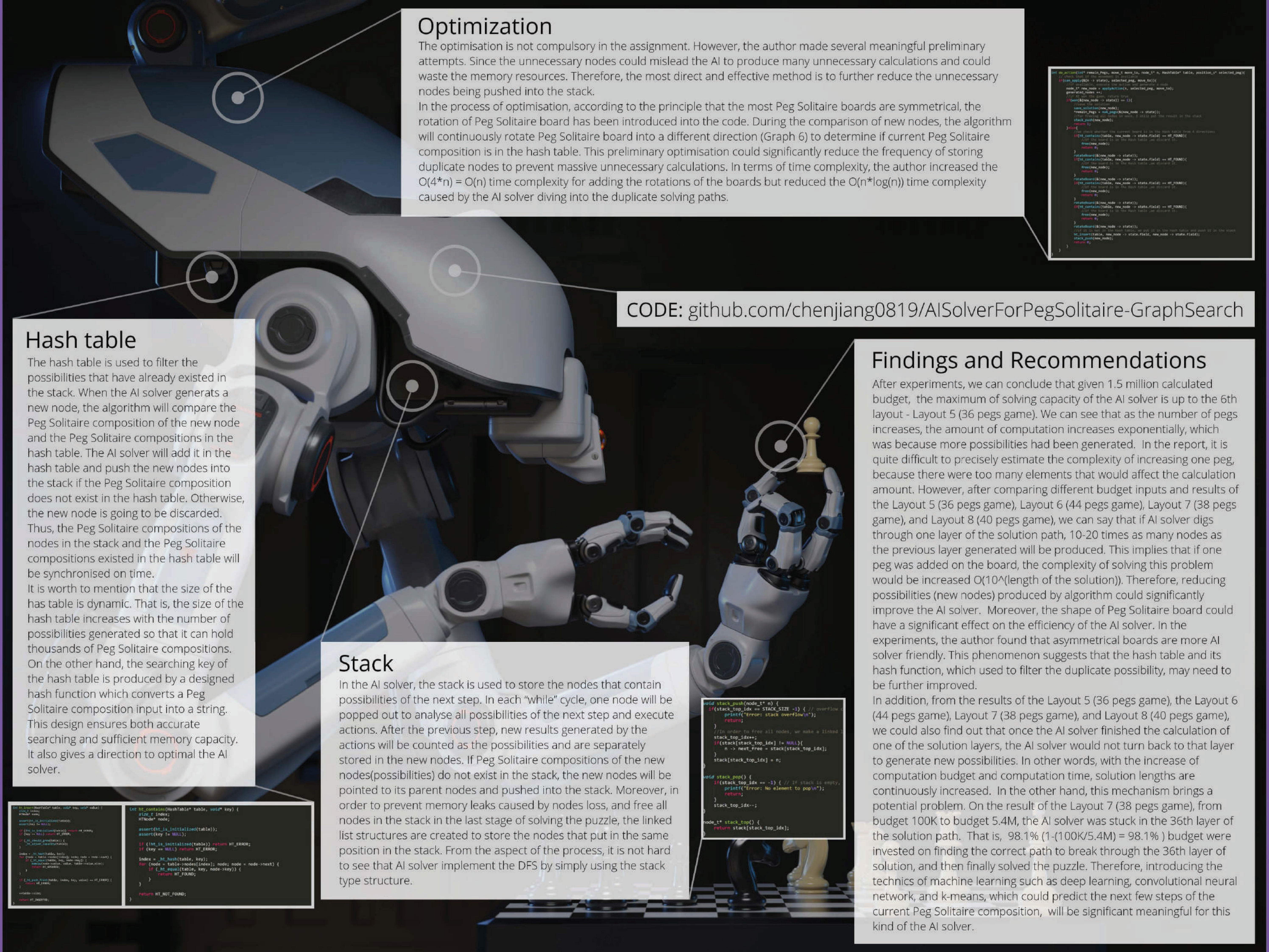
## Introduction

The primary purpose of this porject is to explore the method based on the principle of the AlphaGo to solve the Peg Solitaire problem. That is, the nodes pointed their parent nodes will be created to store all possibilities of the current Peg Solitaire composition to form a graphic which has many connected nodes. Whilst the Depth First Search (DFS) strategy is implemented to find out the deepest path generated in the graph .
Experiments shows that the AI is able to solve the first 6 layouts provided by the instruction entirely. In the statistic of solving the sixth layouts of the Peg Solitaire problem, the 4898609 nodes were generated, and 1090275 of them were popped out from the stack for analysing. The problem was solved in 6.031 seconds.

## Depth First Search (DFS)

Depth First Search (DFS) is an algorithm that is implemented to traverse a connected graph. The mechanism is that the algorithm starts at one original vertex V0, and the algorithm tries to traverse all the vertex to the end. If the computer finds that it cannot reach the goals, it will return to the previous node, and then tries to find another path to reach the destination. The conception of this algorithm is trying to achieve the vertex as deep as it can (2020 Geeksforgeeks). In the AI solver from this paper, this DFS was used to optimise and save the solutions while the algorism is executing.

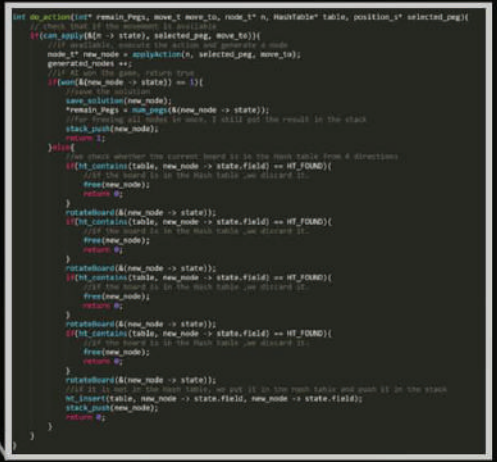




## Optimization

The optimisation is not compulsory in the assignment. However, the author made several meaningful preliminary attempts. Since the unnecessary nodes could mislead the AI to produce many unnecessary calculations and could waste the memory resources. Therefore, the most direct and effective method is to further reduce the unnecessary nodes being pushed into the stack.

In the process of optimisation, according to the principle that the most Peg Solitaire boards are symmetrical, the rotation of Peg Solitaire board has been introduced into the code. During the comparison of new nodes, the algorithm will continuously rotate Peg Solitaire board into a different direction (Graph 6) to determine if current Peg Solitaire composition is in the hash table. This preliminary optimisation could significantly reduce the frequency of storing duplicate nodes to prevent massive unnecessary calculations. In terms of time complexity, the author increased the O(4*n) = O(n) time complexity for adding the rotations of the boards but reduced the O(n*log(n)) time complexity caused by the AI solver diving into the duplicate solving paths.



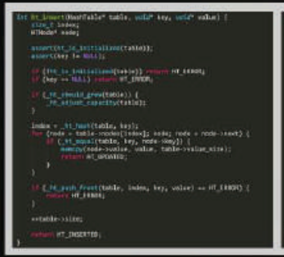
```c
int do_action(int* remain_Pegs, move_t move_to, node_t* n, HashTable* table, position_s* selected_peg){
    // check that if the movement is available
    if(can_apply(&(n -> state), selected_peg, move_to)){
        //if available, execute the action and generate a node
        node_t* new_node = applyAction(n, selected_peg, move_to);
        generated_nodes ++;
        //if AI won the game, return true
        if(won(&(new_node -> state)) == 1){
            //save the solution
            save_solution(new_node);
            *remain_Pegs = num_pegs(&(new_node -> state));
            //for freeing all nodes in once, I still put the result in the stack
            stack_push(new_node);
            return 1;
        }else{
            //we check whether the current board is in the Hash table from 4 directions
            if(ht_contains(table, new_node -> state.field) == HT_FOUND){
                //if the board is in the Hash table ,we discard it.
                free(new_node);
                return 0;
            }
            rotateBoard(&(new_node -> state));
            if(ht_contains(table, new_node -> state.field) == HT_FOUND){
                //if the board is in the Hash table ,we discard it.
                free(new_node);
                return 0;
            }
            rotateBoard(&(new_node -> state));
            if(ht_contains(table, new_node -> state.field) == HT_FOUND){
                //if the board is in the Hash table ,we discard it.
                free(new_node);
                return 0;
            }
            rotateBoard(&(new_node -> state));
            if(ht_contains(table, new_node -> state.field) == HT_FOUND){
                //if the board is in the Hash table ,we discard it.
                free(new_node);
                return 0;
            }
            rotateBoard(&(new_node -> state));
            //if it is not in the Hash table, we put it in the Hash table and push it in the stack
            ht_insert(table, new_node -> state.field, new_node -> state.field);
            stack_push(new_node);
            return 0;
        }
    }
}
```

**CODE:** github.com/chenjiang0819/AISolverForPegSolitaire-GraphSearch

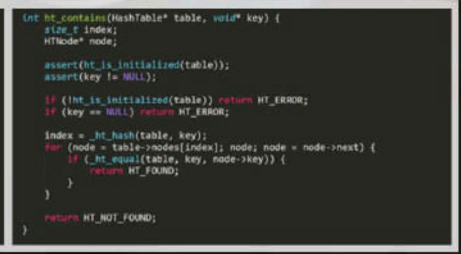
## Hash table

The hash table is used to filter the possibilities that have already existed in the stack. When the AI solver generats a new node, the algorithm will compare the Peg Solitaire composition of the new node and the Peg Solitaire compositions in the hash table. The AI solver will add it in the hash table and push the new nodes into the stack if the Peg Solitaire composition does not exist in the hash table. Otherwise, the new node is going to be discarded. Thus, the Peg Solitaire compositions of the nodes in the stack and the Peg Solitaire compositions existed in the hash table will be synchronised on time.

It is worth to mention that the size of the has table is dynamic. That is, the size of the hash table increases with the number of possibilities generated so that it can hold thousands of Peg Solitaire compositions. On the other hand, the searching key of the hash table is produced by a designed hash function which converts a Peg Solitaire composition input into a string. This design ensures both accurate searching and sufficient memory capacity. It also gives a direction to optimal the AI solver.



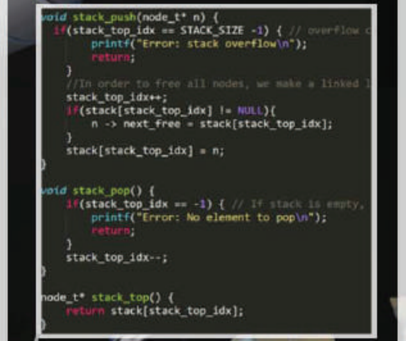
```c
int ht_insert(HashTable* table, void* key, void* value) {
    size_t index;
    HTNode* node;

    assert(ht_is_initialized(table));
    assert(key != NULL);

    if (!ht_is_initialized(table)) return HT_ERROR;
    if (key == NULL) return HT_ERROR;

    if (_ht_should_grow(table)) {
        _ht_adjust_capacity(table);
    }

    index = _ht_hash(table, key);
    for (node = table->nodes[index]; node; node = node->next) {
        if (_ht_equal(table, key, node->key)) {
            memcpy(node->value, value, table->value_size);
            return HT_UPDATED;
        }
    }

    if (_ht_push_front(table, index, key, value) == HT_ERROR) {
        return HT_ERROR;
    }

    ++table->size;

    return HT_INSERTED;
}
```



```c
int ht_contains(HashTable* table, void* key) {
    size_t index;
    HTNode* node;

    assert(ht_is_initialized(table));
    assert(key != NULL);

    if (!ht_is_initialized(table)) return HT_ERROR;
    if (key == NULL) return HT_ERROR;

    index = _ht_hash(table, key);
    for (node = table->nodes[index]; node; node = node->next) {
        if (_ht_equal(table, key, node->key)) {
            return HT_FOUND;
        }
    }

    return HT_NOT_FOUND;
}
```

## Stack

In the AI solver, the stack is used to store the nodes that contain possibilities of the next step. In each "while" cycle, one node will be popped out to analyse all possibilities of the next step and execute actions. After the previous step, new results generated by the actions will be counted as the possibilities and are separately stored in the new nodes. If Peg Solitaire compositions of the new nodes(possibilities) do not exist in the stack, the new nodes will be pointed to its parent nodes and pushed into the stack. Moreover, in order to prevent memory leaks caused by nodes loss, and free all nodes in the stack in the last stage of solving the puzzle, the linked list structures are created to store the nodes that put in the same position in the stack. From the aspect of the process, it is not hard to see that AI solver implement the DFS by simply using the stack type structure.



```c
void stack_push(node_t* n) {
  if(stack_top_idx == STACK_SIZE -1) { // overflow c
        printf("Error: stack overflow\n");
        return;
    }
    //In order to free all nodes, we make a linked l
    stack_top_idx++;
    if(stack[stack_top_idx] != NULL){
        n -> next_free = stack[stack_top_idx];
    }
    stack[stack_top_idx] = n;
}

void stack_pop() {
    if(stack_top_idx == -1) { // If stack is empty,
        printf("Error: No element to pop\n");
        return;
    }
    stack_top_idx--;
}

node_t* stack_top() {
    return stack[stack_top_idx];
}
```

## Findings and Recommendations

After experiments, we can conclude that given 1.5 million calculated budget, the maximum of solving capacity of the AI solver is up to the 6th layout - Layout 5 (36 pegs game). We can see that as the number of pegs increases, the amount of computation increases exponentially, which was because more possibilities had been generated. In the report, it is quite difficult to precisely estimate the complexity of increasing one peg, because there were too many elements that would affect the calculation amount. However, after comparing different budget inputs and results of the Layout 5 (36 pegs game), Layout 6 (44 pegs game), Layout 7 (38 pegs game), and Layout 8 (40 pegs game), we can say that if AI solver digs through one layer of the solution path, 10-20 times as many nodes as the previous layer generated will be produced. This implies that if one peg was added on the board, the complexity of solving this problem would be increased O(10^(length of the solution)). Therefore, reducing possibilities (new nodes) produced by algorithm could significantly improve the AI solver. Moreover, the shape of Peg Solitaire board could have a significant effect on the efficiency of the AI solver. In the experiments, the author found that asymmetrical boards are more AI solver friendly. This phenomenon suggests that the hash table and its hash function, which used to filter the duplicate possibility, may need to be further improved.

In addition, from the results of the Layout 5 (36 pegs game), the Layout 6 (44 pegs game), Layout 7 (38 pegs game), and Layout 8 (40 pegs game), we could also find out that once the AI solver finished the calculation of one of the solution layers, the AI solver would not turn back to that layer to generate new possibilities. In other words, with the increase of computation budget and computation time, solution lengths are continuously increased. In the other hand, this mechanism brings a potential problem. On the result of the Layout 7 (38 pegs game), from budget 100K to budget 5.4M, the AI solver was stuck in the 36th layer of the solution path. That is, 98.1% (1-(100K/5.4M) = 98.1% ) budget were invested on finding the correct path to break through the 36th layer of solution, and then finally solved the puzzle. Therefore, introducing the technics of machine learning such as deep learning, convolutional neural network, and k-means, which could predict the next few steps of the current Peg Solitaire composition, will be significant meaningful for this kind of the AI solver.