Introduction To Artificial Intelligence: Homework 1 Report

Andres Ponce(彭思安)
0616110

April 10, 2020

1 Introduction

The first assignment for the course involved a chess board, and placing a knight on that board. Given a starting and ending position, how do we move the knight along the "best" path from the start to the finish? Using several of the methods described in the textbook, the task focused on comparing the performance of each one. Some criteria may include: completeness, time and space complexity, among other general observations.

1.1 Breadth-First-Search

The first algorithm to be implemented was the classic Breadth-First-Search. In general, this algorithm keeps a "frontier" data structure where to-be-expanded nodes are stored. Every iteration, we pop a node from this set and examine its children. Since we can move by either $(\pm 1, \pm 2)$ or $(\pm 2, \pm 1)$, every node has a possible 8 children.

In this implementation, we first check the validity of the specific child node(i.e. if it has already been explored), and if we have not encountered it and is withing the board bounds, we add it to the frontier list, implemented here by `std::list<Node*>`. If we already explored a node,then we already know it does not contain a shortest path, otherwise the algorithm would have halted already.

To store the individual nodes information, we use a special `struct Node`, which essentially just stores the node's correspondent x and y values in the board. Then, a quick hash function can turn the x and y values into the index in the node array.

Breadth-First Search appears to not be the most efficient algorithm to use as is, unless we add some form of heuristic when we choose which nodes to explore. Our algorithm will eventually reach the target node, however given a large enough board the time would likely be prohibitive. Since the branching factor is 8, since we can make 8 possible moves at every node, the worst-case time complexity is $O(d^8)$, where d is the depth of the solution path.

1.2 Depth-First-Search

For depth-first-search(DFS), we first carry out a thread to the end, and as we are left without valid states to move on to that we start backtracking up the chain. On average, it tends to be slower than BFS, since we can explore a large amount of nodes before getting to the target if the target is near the starting point. However, the *memory* requirements are quite lower. This is because at any point, only the nodes on the current search path are kept in memory – the ones we have to search – rather than all the nodes at a given level such as BFS.

From some of the sample tests, DFS resulted in sometimes significantly longer search paths, since it would search for paths that lead to dead ends.

Depending on the problem, DFS would probably not be the most ideal search algorithm. These two first algorithms usually have no heuristic to select nodes, so considerable time may be spent on nodes that ultimately have no bearing on the final result. Thus, uninformed algorithms in general might not be the wisest move.

An interesting experiment or topic for further discussion given enough time might be to check the actual average difference between random points on the board.

1.3 Iterative Deepening Search

Iterative Deepening Search, sometimes referred to as *Iterative Deepening Depth-First Search*, attempts to combine the useful properties of BFS and DFS in one single algorithm. First, we set a limit on the depth of the solution, and call a depth-first search on the starting node. Thus, whenever we find the solution node, it should be done using the least amount of ndoes possible.

Every time we don't find the solution node, we increase the limit and try again. This might result in the top nodes of the search tree being generated often, however the difference still turns out not to be too great, since most nodes reside in the lower levels of the search tree(assuming constant branching factor). Even though the asymptotic running time remains $O(b^d)$ (same as BFS), the memory complexity is that of DFS $O(bd)$. For uninformed strategies with an unknown solution depth, IDS might be the best choice since like DFS for a finite space, it is guaranteed to find the solution becuase of the optimality property.

For our implementation, we use a helper function, since we recursively call the helper function. When first entering the helper function, besides from just checking wehhter we are at the target, we also have to check whether the limit has been reached. Other than that, we proceed in a very similar manner to DFS.

1.4 A* Search

The a* algorithm usually turns out to be one of the most effective search algorithms, since we use a heuristic to make a decision on which nodes to expand. The algorithm itself is relatively straightforward: for every node, we keep track of three variables $f, g,$ and h. g is the cost of getting to node n, while h is the (estimated) cost of arriving at the solution node. Then, for each node $f = g + h$. From the child nodes, we choose one that has a lower f cost than its parent.

This is where the problems begin. Due to the unconventional movement of the knight in chess, the program does not find the correct answer (and sometimes no answer at all) when we have to make a move away from the target, to a node that is actually *farther away* from the target node than the one we are currently in right now. For example, the sample problem involves going from $(0, 0)$ to $(2, 2)$. In this case, the program would move to $(2, 1)$ and then get stuck because the best move is actually one that is farther away from $(2, 2)$, i.e. a node with a higher f value.

Adapting this algorithm this situation would most definitely be an interesting research question. A* remains widely used in many industries, from computer vision to video games, due to its efficiency and correctness. Since we only expand nodes that grant a better chance of arriving at a target, then we can overall expand fewer nodes.

In this implemenation, if the target node can be reached directly without going to a further point, then the algorithm will reach the target node with fewer expanded nodes than with an uninformed search.

1.5 Iterative Deepening A* Algorithm

This algorithm attempts to mix the Iterative Deepening algorithm while using a heuristic similar to the A* algorithm. We keep incrementing the bound on the node that we call the main search function on. However, instead of this bound coming from the depth of the search process, we use the nodes f value. The main algorithm, `IDA_STAR_search` function, returns either if the node is found or if there is no possible path.

Similar to regular A*, if the heuristic we use is admissible, meaning it never underestimates the distance to the target node (which the sample heuristic in `Prog1.pdf` does not). Thus, at every iteration the f values of the nodes will not exceed the previous node's f value. Therefore, every node we select will have value at most the parent's f value. This guarantees that the optimal node will actually not be overlooked due to the estimation using the heursitic function.

However, the potential benefit of IDA* comes in memory utilization, since there is no need to keep a list of the nodes that we will explore as is the case in A*. In A*, the nodes were checked against the Open set to see if we were duplicating nodes. This ensured that nodes were only expanded once. IDA* does not utilize this approach, so some nodes might be expanded multiple times during the recursion process.

2 Results

Below are some of the results obtained from the testing on the board. Using a standard board of 8 rows and columns, we tested the amount of expanded nodes going from $(0,0)$ to $(2,2)$ as in the example, and then proceeded to test on a $16x16$ board going from $(0,0)$ to $(7,7)$ and $(0,0)$ to $(15,15)$.

Route	BFS	DFS	IDS	A*	IDA*
(8x8)(0, 0)->(2,2)	83	25	55	6	2
(8x8)(0,0)->(7,7)	583	68	10	24	10
(16x16)(0,0)->(7,7)	925	101	136	42	111
(16x16)(0,0)->(15, 15)	65809	259	18	634	18

As shown, some of the results are surprising. While some may be attributable to poor implementation, however, with the amount of nodes that some visit, even if momentarily, can be quite large. For the BFS, we check all the descendants without any sort of heuristic, so larger numbers are not too unexpected.

The ease of getting to some of the testing nodes might also play a part in the pathfinding. For example, sometimes we need to "snake" around the board until we reach the target node, even if the algorithm is still relatively close to the target node. For the informed algorithms, there are usually less nodes required to arrive at the destination. As mentioned, for the informed search algorithms there might be some problems if we have to search a node that is farther from the target node because the heuristic might cause that node to be overlooked. However, when we do get a solution, it usually requires considerably less nodes than an algorithm such as BFS.

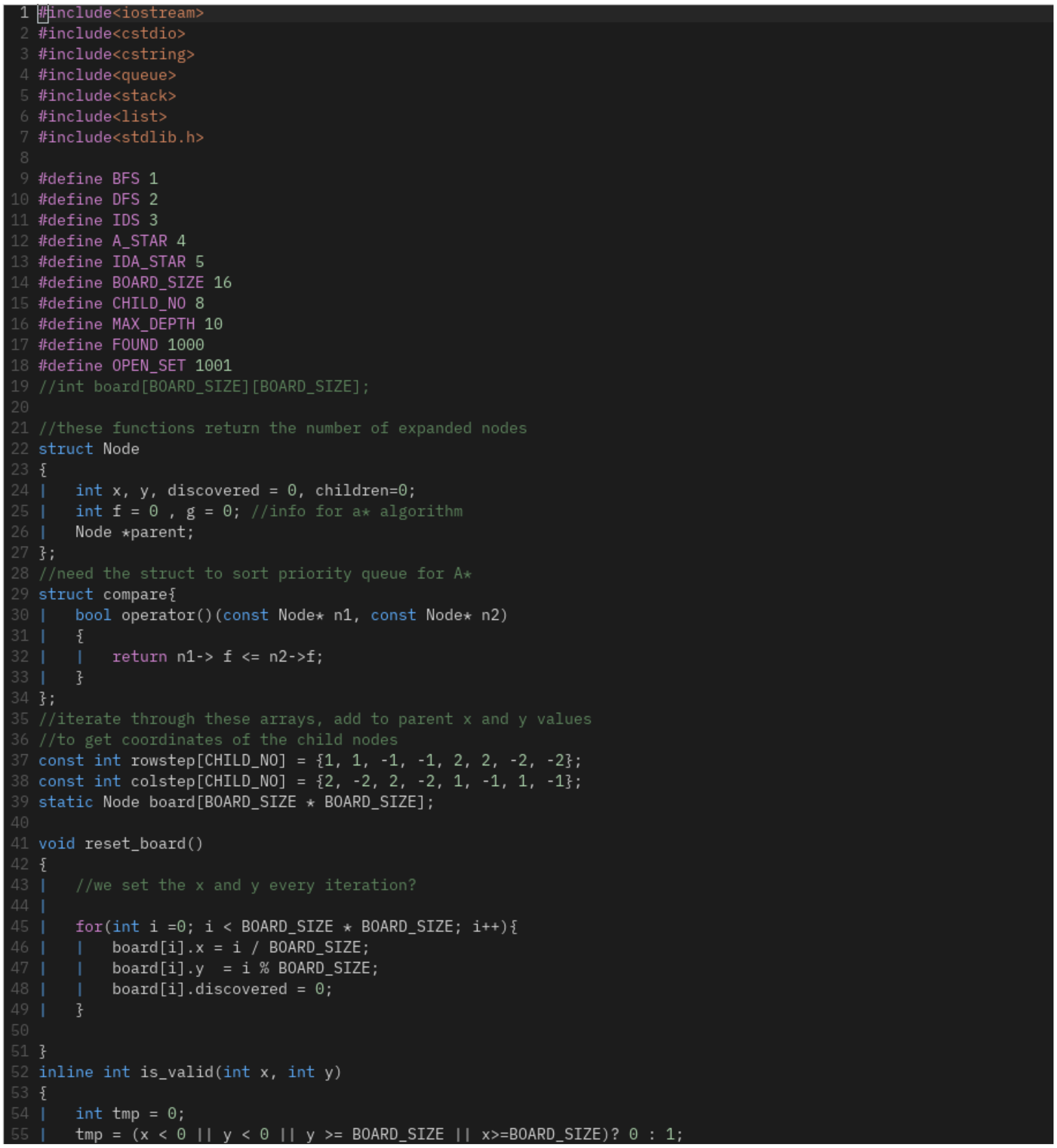
3 Lessons

The biggest personal takeaway from this assignment was the sheer difference between informed and uninformed searches. Spending valuable time expanding nodes that do not influence the final result made the optimal path take considerably longer to compute. However, the problem was solvable even with a relatively straightforward approach such as BFS and DFS. A* would take considerably fewer nodes, usually less than 10 expansions. The total f values would quickly get very small,which means that the algorithm was moving in a targetetd manner towards the target node.

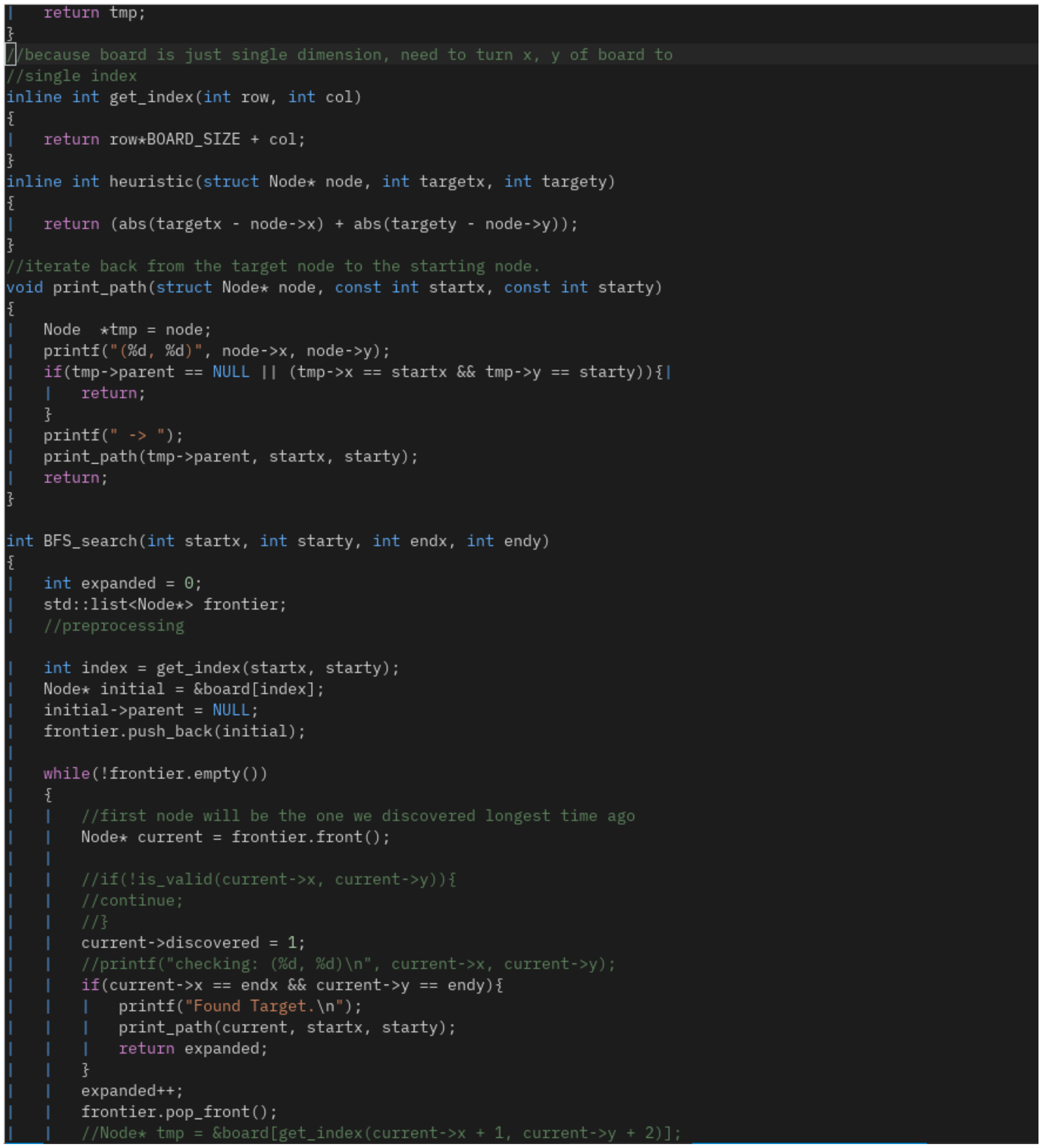
The actual implementation of the algorithms also proved useful. Although I had already implemented most of them in previous courses, this time finding a better approach to writing the code took more priority. Also, even though the small changes from algorithm to algorithm, or the way some of the algorithms tested utilize ideas from the other algorithms showed how even small tweaks might influence the algorithms. Then, combining ideas from the two algorithms can sometimes also be applied to situations with different constraints. For example, IDS and IDA* combine and build upon DFS and A*, respectively.

The ideas behind graph search are applicable in an incredible wide range of areas, such as GPS services video games, and making game-playing adversaries, to name a few examples. One possible area of future research might include making an agent that can intelligently play some sort of game, now knowing how some of the major algorithms compare to each other.

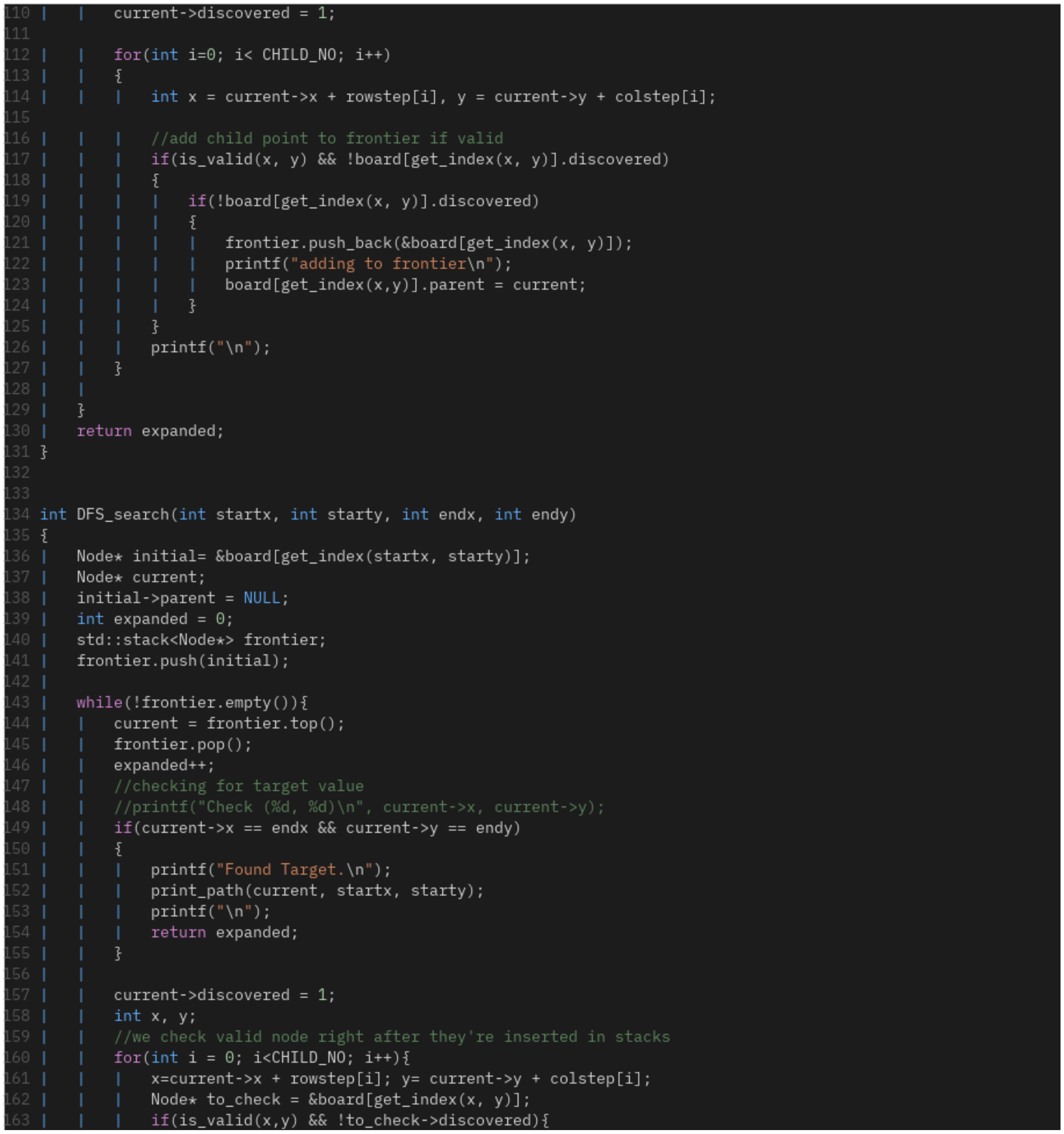
4 Code Images



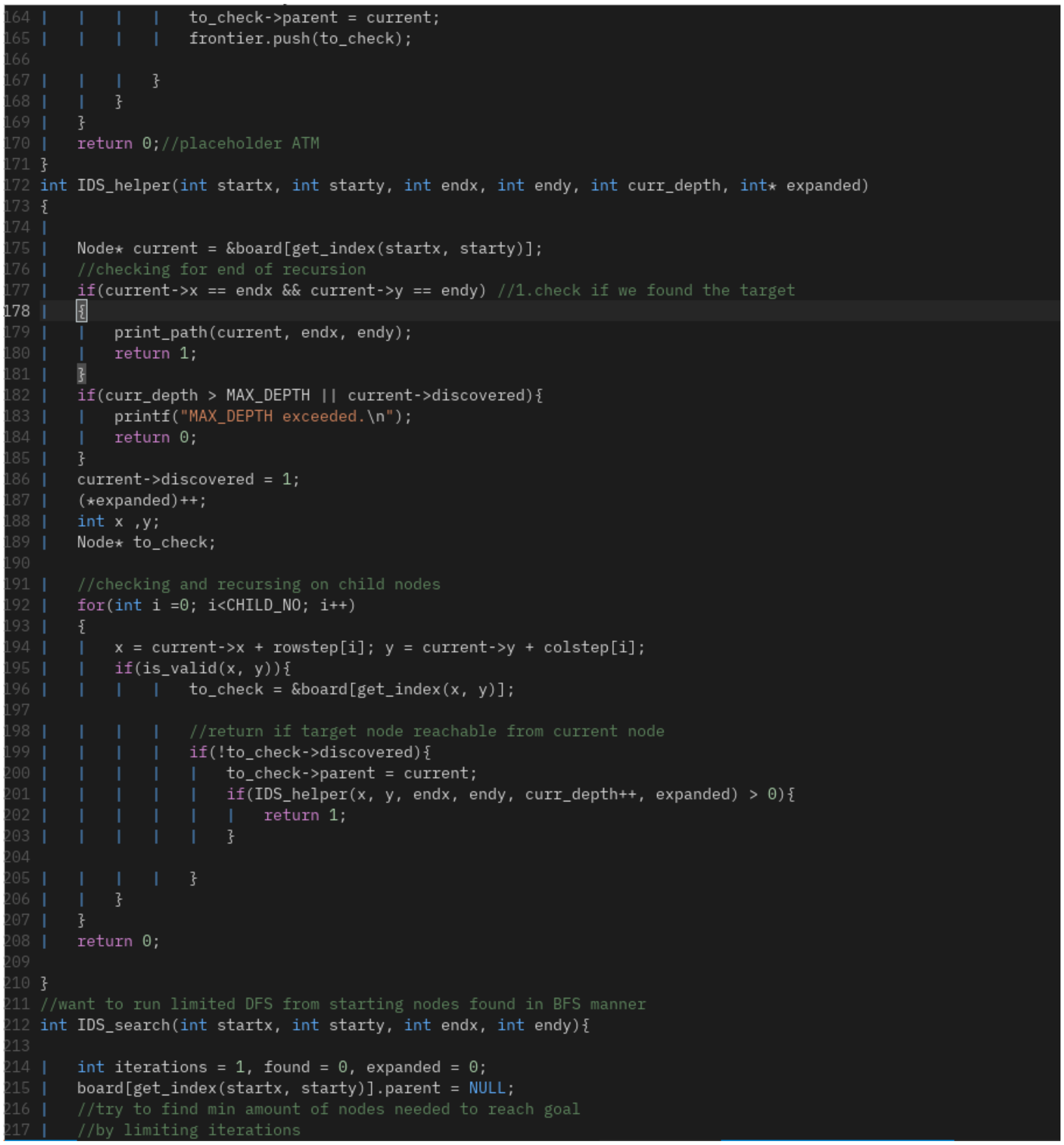
```
#include<iostream>
#include<cstdio>
#include<cstring>
#include<queue>
#include<stack>
#include<list>
#include<stdlib.h>

#define BFS 1
#define DFS 2
#define IDS 3
#define A_STAR 4
#define IDA_STAR 5
#define BOARD_SIZE 16
#define CHILD_NO 8
#define MAX_DEPTH 10
#define FOUND 1000
#define OPEN_SET 1001
//int board[BOARD_SIZE][BOARD_SIZE];

//these functions return the number of expanded nodes
struct Node
{
    int x, y, discovered = 0, children=0;
    int f = 0 , g = 0; //info for a* algorithm
    Node *parent;
};
//need the struct to sort priority queue for A*
struct compare{
    bool operator()(const Node* n1, const Node* n2)
    {
        return n1-> f <= n2->f;
    }
};
//iterate through these arrays, add to parent x and y values
//to get coordinates of the child nodes
const int rowstep[CHILD_NO] = {1, 1, -1, -1, 2, 2, -2, -2};
const int colstep[CHILD_NO] = {2, -2, 2, -2, 1, -1, 1, -1};
static Node board[BOARD_SIZE * BOARD_SIZE];

void reset_board()
{
    //we set the x and y every iteration?

    for(int i =0; i < BOARD_SIZE * BOARD_SIZE; i++){
        board[i].x = i / BOARD_SIZE;
        board[i].y  = i % BOARD_SIZE;
        board[i].discovered = 0;
    }

}
inline int is_valid(int x, int y)
{
    int tmp = 0;
    tmp = (x < 0 || y < 0 || y >= BOARD_SIZE || x>=BOARD_SIZE)? 0 : 1;
```



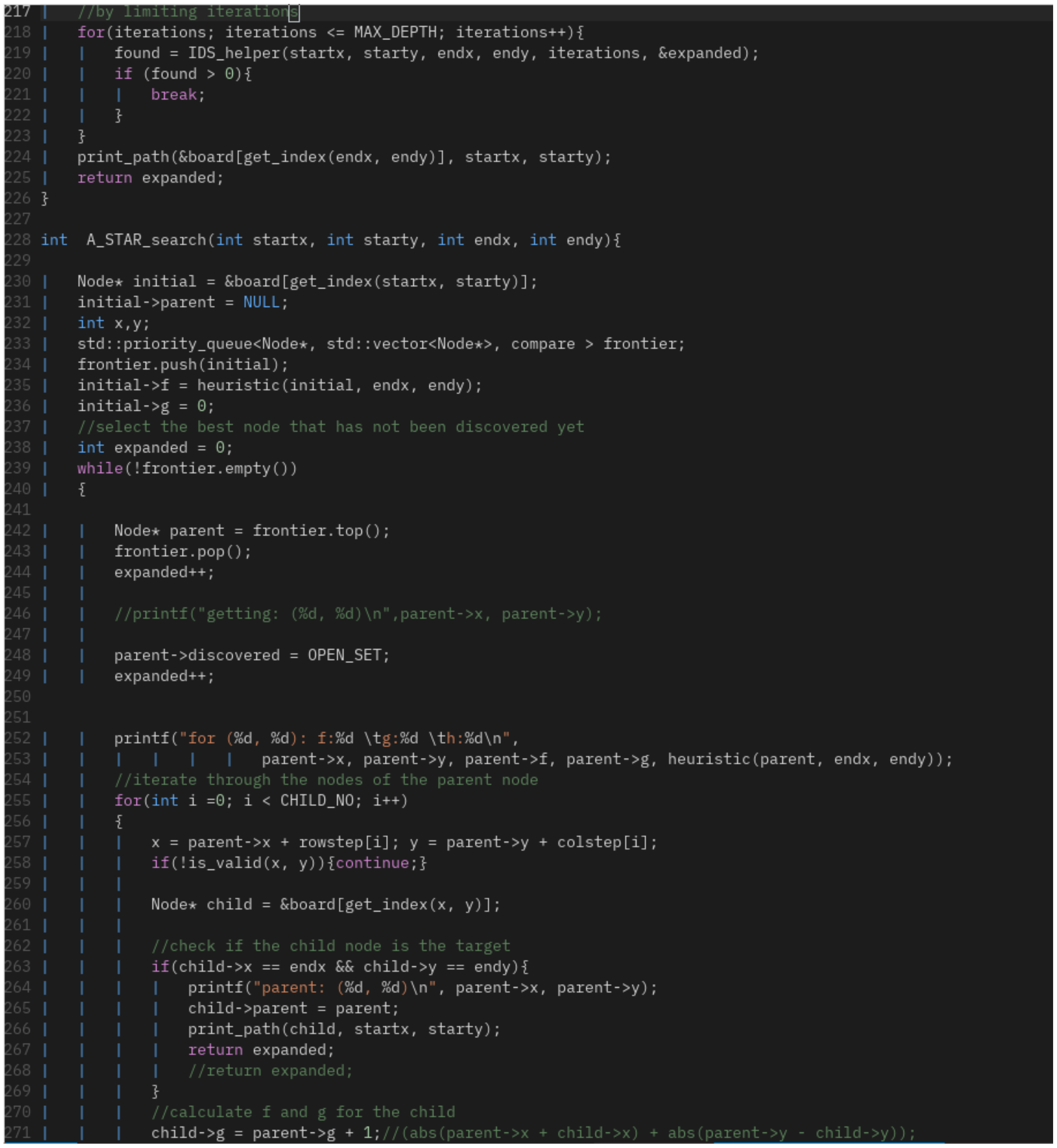
```
    return tmp;
}
//because board is just single dimension, need to turn x, y of board to
//single index
inline int get_index(int row, int col)
{
    return row*BOARD_SIZE + col;
}
inline int heuristic(struct Node* node, int targetx, int targety)
{
    return (abs(targetx - node->x) + abs(targety - node->y));
}
//iterate back from the target node to the starting node.
void print_path(struct Node* node, const int startx, const int starty)
{
    Node  *tmp = node;
    printf("(%d, %d)", node->x, node->y);
    if(tmp->parent == NULL || (tmp->x == startx && tmp->y == starty)){
        return;
    }
    printf(" -> ");
    print_path(tmp->parent, startx, starty);
    return;
}

int BFS_search(int startx, int starty, int endx, int endy)
{
    int expanded = 0;
    std::list<Node*> frontier;
    //preprocessing

    int index = get_index(startx, starty);
    Node* initial = &board[index];
    initial->parent = NULL;
    frontier.push_back(initial);

    while(!frontier.empty())
    {
        //first node will be the one we discovered longest time ago
        Node* current = frontier.front();

        //if(!is_valid(current->x, current->y)){
        //continue;
        //}
        current->discovered = 1;
        //printf("checking: (%d, %d)\n", current->x, current->y);
        if(current->x == endx && current->y == endy){
            printf("Found Target.\n");
            print_path(current, startx, starty);
            return expanded;
        }
        expanded++;
        frontier.pop_front();
        //Node* tmp = &board[get_index(current->x + 1, current->y + 2)];
```



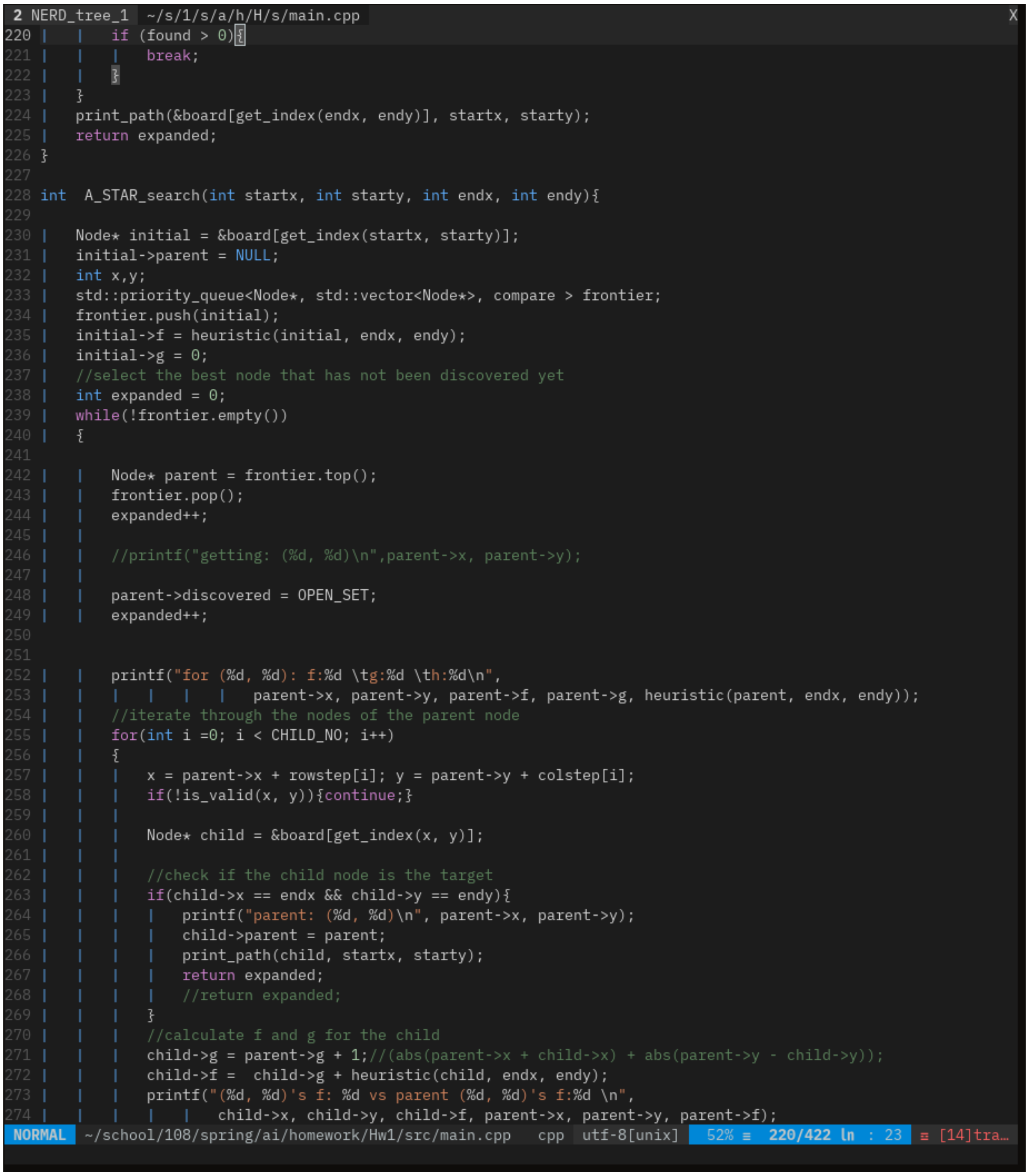
```
        current->discovered = 1;

        for(int i=0; i< CHILD_NO; i++)
        {
            int x = current->x + rowstep[i], y = current->y + colstep[i];

            //add child point to frontier if valid
            if(is_valid(x, y) && !board[get_index(x, y)].discovered)
            {
                if(!board[get_index(x, y)].discovered)
                {
                    frontier.push_back(&board[get_index(x, y)]);
                    printf("adding to frontier\n");
                    board[get_index(x,y)].parent = current;
                }
            }
            printf("\n");
        }

    }
    return expanded;
}


int DFS_search(int startx, int starty, int endx, int endy)
{
    Node* initial= &board[get_index(startx, starty)];
    Node* current;
    initial->parent = NULL;
    int expanded = 0;
    std::stack<Node*> frontier;
    frontier.push(initial);

    while(!frontier.empty()){
        current = frontier.top();
        frontier.pop();
        expanded++;
        //checking for target value
        //printf("Check (%d, %d)\n", current->x, current->y);
        if(current->x == endx && current->y == endy)
        {
            printf("Found Target.\n");
            print_path(current, startx, starty);
            printf("\n");
            return expanded;
        }

        current->discovered = 1;
        int x, y;
        //we check valid node right after they're inserted in stacks
        for(int i = 0; i<CHILD_NO; i++){
            x=current->x + rowstep[i]; y= current->y + colstep[i];
            Node* to_check = &board[get_index(x, y)];
            if(is_valid(x,y) && !to_check->discovered){
```



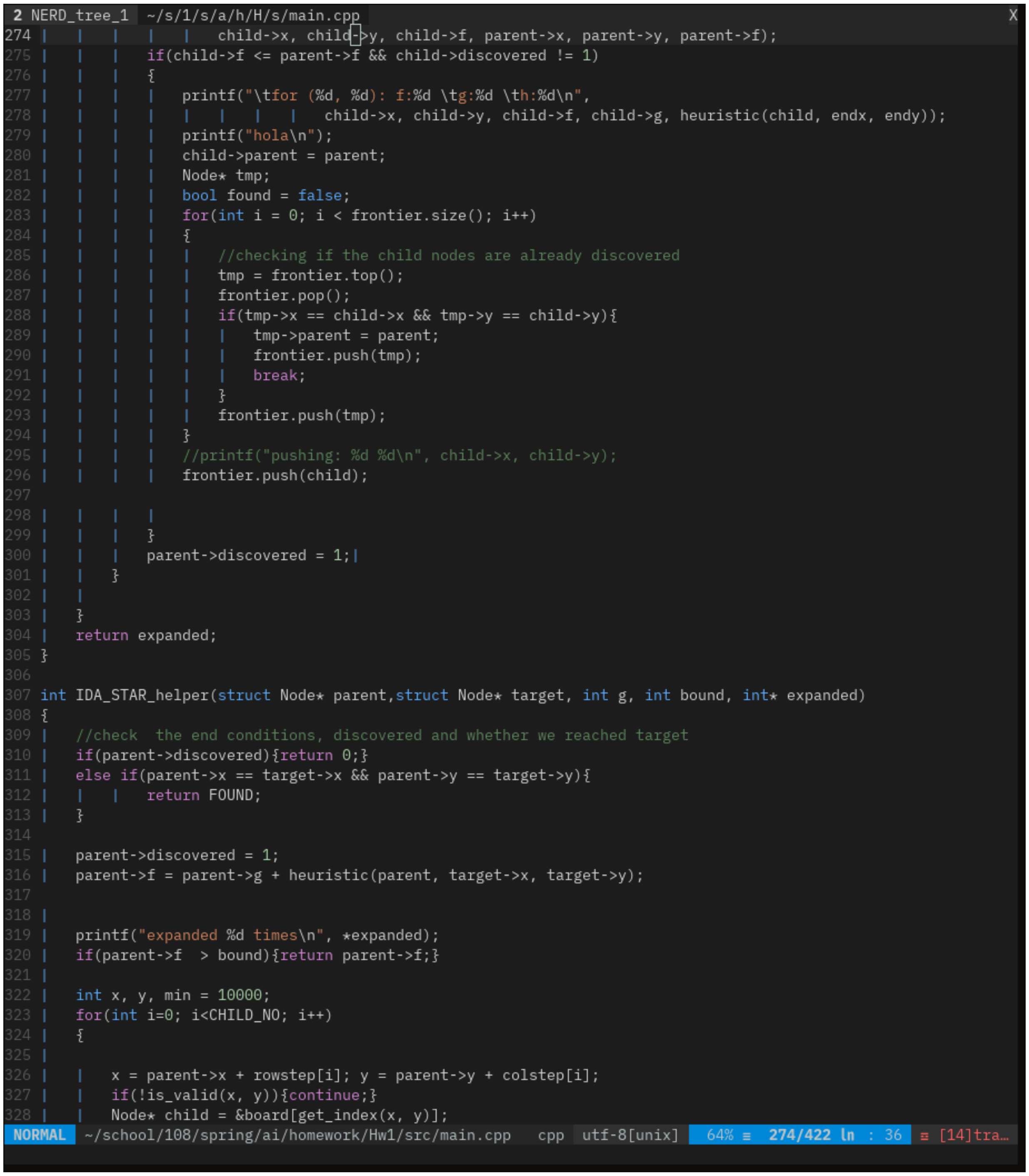
```
                to_check->parent = current;
                frontier.push(to_check);

            }
        }
    }
    return 0;//placeholder ATM
}
int IDS_helper(int startx, int starty, int endx, int endy, int curr_depth, int* expanded)
{

    Node* current = &board[get_index(startx, starty)];
    //checking for end of recursion
    if(current->x == endx && current->y == endy) //1.check if we found the target
    {
        print_path(current, endx, endy);
        return 1;
    }
    if(curr_depth > MAX_DEPTH || current->discovered){
        printf("MAX_DEPTH exceeded.\n");
        return 0;
    }
    current->discovered = 1;
    (*expanded)++;
    int x ,y;
    Node* to_check;

    //checking and recursing on child nodes
    for(int i =0; i<CHILD_NO; i++)
    {
        x = current->x + rowstep[i]; y = current->y + colstep[i];
        if(is_valid(x, y)){
                to_check = &board[get_index(x, y)];

                //return if target node reachable from current node
                if(!to_check->discovered){
                    to_check->parent = current;
                    if(IDS_helper(x, y, endx, endy, curr_depth++, expanded) > 0){
                        return 1;
                    }

                }
        }
    }
    return 0;

}
//want to run limited DFS from starting nodes found in BFS manner
int IDS_search(int startx, int starty, int endx, int endy){

    int iterations = 1, found = 0, expanded = 0;
    board[get_index(startx, starty)].parent = NULL;
    //try to find min amount of nodes needed to reach goal
    //by limiting iterations
```



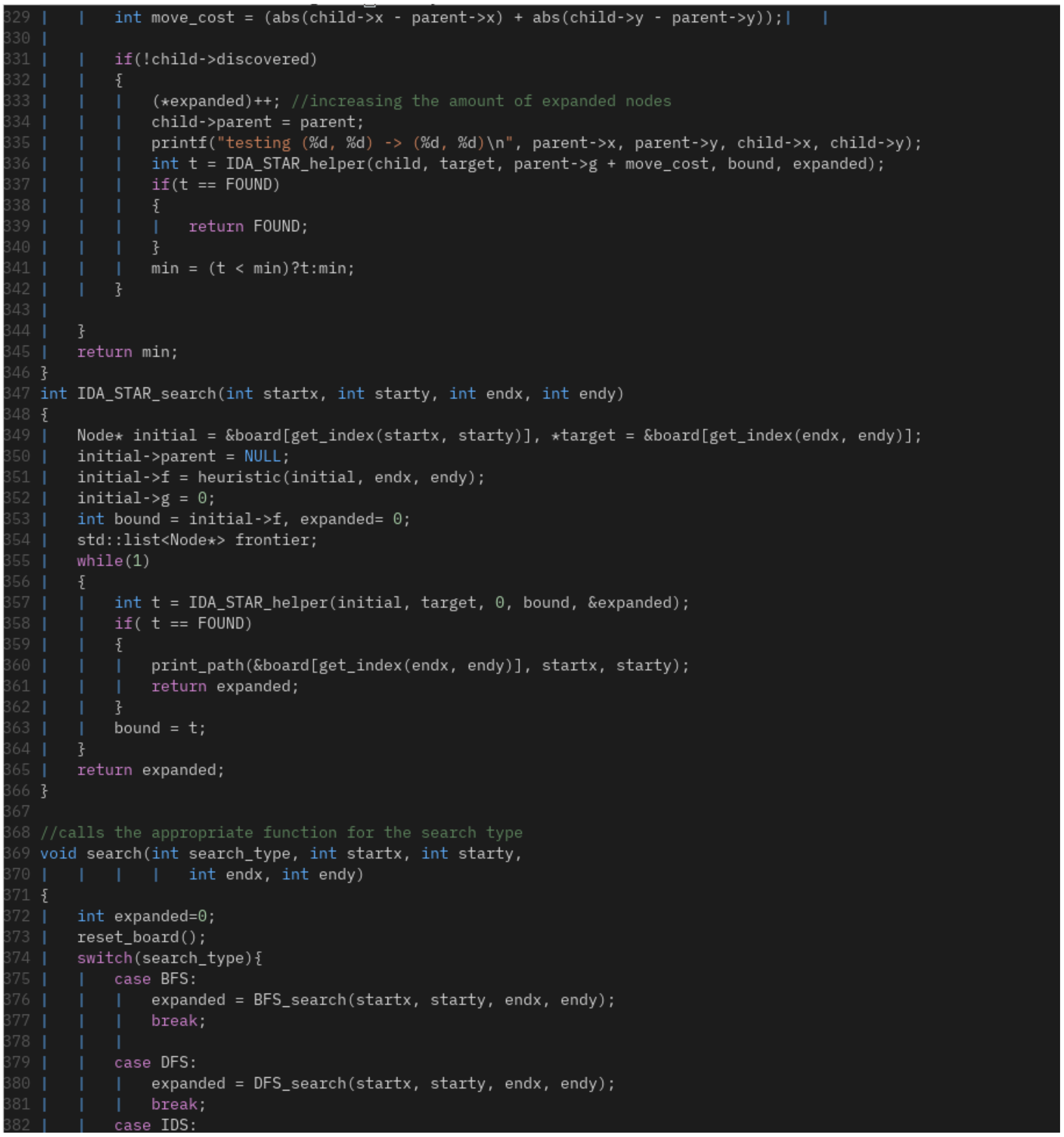
```
    //by limiting iterations
    for(iterations; iterations <= MAX_DEPTH; iterations++){
        found = IDS_helper(startx, starty, endx, endy, iterations, &expanded);
        if (found > 0){
            break;
        }
    }
    print_path(&board[get_index(endx, endy)], startx, starty);
    return expanded;
}

int  A_STAR_search(int startx, int starty, int endx, int endy){

    Node* initial = &board[get_index(startx, starty)];
    initial->parent = NULL;
    int x,y;
    std::priority_queue<Node*, std::vector<Node*>, compare > frontier;
    frontier.push(initial);
    initial->f = heuristic(initial, endx, endy);
    initial->g = 0;
    //select the best node that has not been discovered yet
    int expanded = 0;
    while(!frontier.empty())
    {

        Node* parent = frontier.top();
        frontier.pop();
        expanded++;

        //printf("getting: (%d, %d)\n",parent->x, parent->y);

        parent->discovered = OPEN_SET;
        expanded++;


        printf("for (%d, %d): f:%d \tg:%d \th:%d\n",
                        parent->x, parent->y, parent->f, parent->g, heuristic(parent, endx, endy));
        //iterate through the nodes of the parent node
        for(int i =0; i < CHILD_NO; i++)
        {
            x = parent->x + rowstep[i]; y = parent->y + colstep[i];
            if(!is_valid(x, y)){continue;}

            Node* child = &board[get_index(x, y)];

            //check if the child node is the target
            if(child->x == endx && child->y == endy){
                printf("parent: (%d, %d)\n", parent->x, parent->y);
                child->parent = parent;
                print_path(child, startx, starty);
                return expanded;
                //return expanded;
            }
            //calculate f and g for the child
            child->g = parent->g + 1;//(abs(parent->x + child->x) + abs(parent->y - child->y));
```



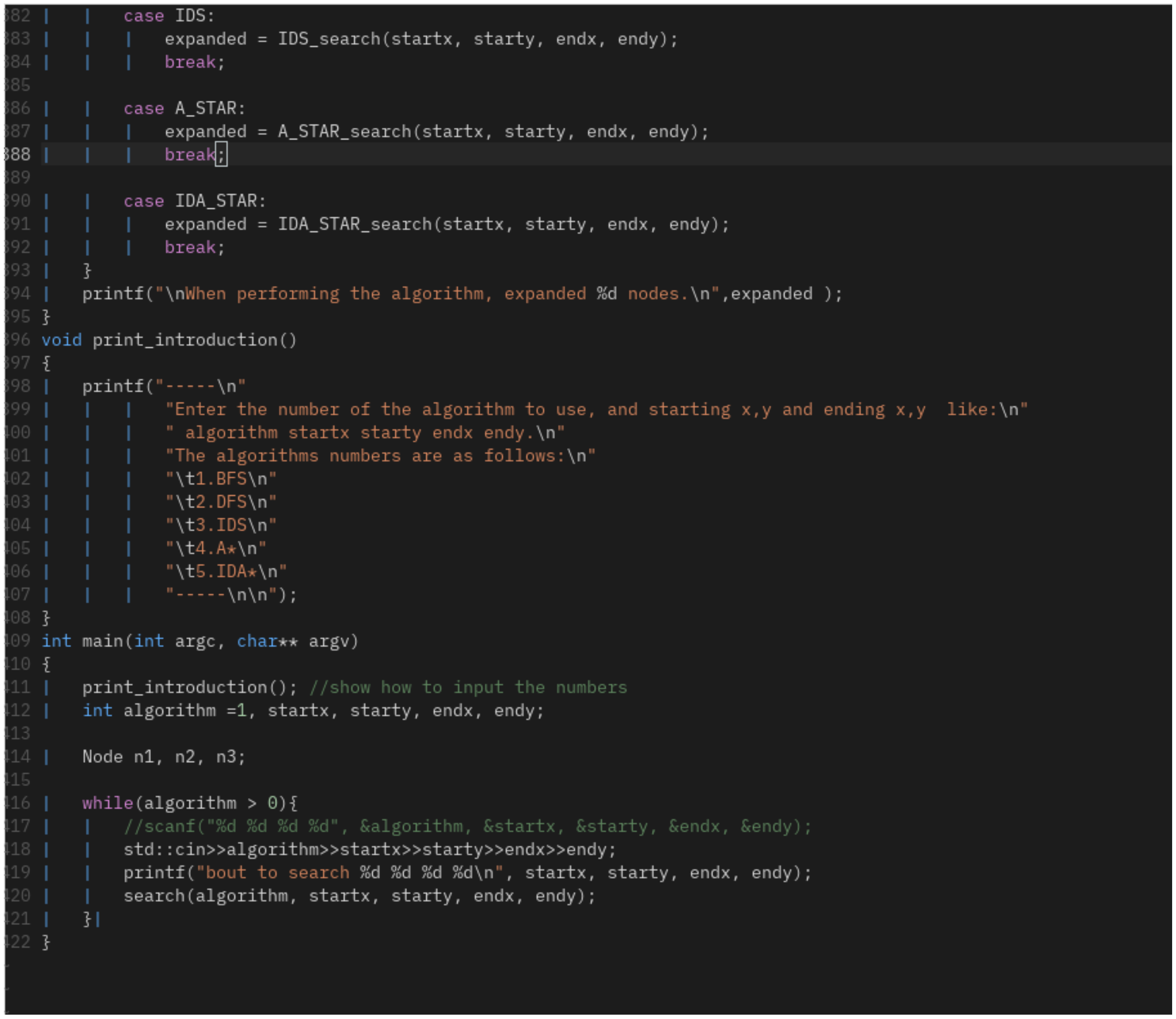
```
 2 NERD_tree_1   ~/s/1/s/a/h/H/s/main.cpp                                                              X
        if (found > 0){
            break;
        }
    }
    print_path(&board[get_index(endx, endy)], startx, starty);
    return expanded;
}

int  A_STAR_search(int startx, int starty, int endx, int endy){

    Node* initial = &board[get_index(startx, starty)];
    initial->parent = NULL;
    int x,y;
    std::priority_queue<Node*, std::vector<Node*>, compare > frontier;
    frontier.push(initial);
    initial->f = heuristic(initial, endx, endy);
    initial->g = 0;
    //select the best node that has not been discovered yet
    int expanded = 0;
    while(!frontier.empty())
    {

        Node* parent = frontier.top();
        frontier.pop();
        expanded++;

        //printf("getting: (%d, %d)\n",parent->x, parent->y);

        parent->discovered = OPEN_SET;
        expanded++;


        printf("for (%d, %d): f:%d \tg:%d \th:%d\n",
                        parent->x, parent->y, parent->f, parent->g, heuristic(parent, endx, endy));
        //iterate through the nodes of the parent node
        for(int i =0; i < CHILD_NO; i++)
        {
            x = parent->x + rowstep[i]; y = parent->y + colstep[i];
            if(!is_valid(x, y)){continue;}

            Node* child = &board[get_index(x, y)];

            //check if the child node is the target
            if(child->x == endx && child->y == endy){
                printf("parent: (%d, %d)\n", parent->x, parent->y);
                child->parent = parent;
                print_path(child, startx, starty);
                return expanded;
                //return expanded;
            }
            //calculate f and g for the child
            child->g = parent->g + 1;//(abs(parent->x + child->x) + abs(parent->y - child->y));
            child->f =  child->g + heuristic(child, endx, endy);
            printf("(%d, %d)'s f: %d vs parent (%d, %d)'s f:%d \n",
                    child->x, child->y, child->f, parent->x, parent->y, parent->f);
 NORMAL  ~/school/108/spring/ai/homework/Hw1/src/main.cpp   cpp  utf-8[unix]   52% ≡  220/422 ln  : 23  ≡ [14]tra…
```



```
2 NERD_tree_1   ~/s/1/s/a/h/H/s/main.cpp                                                        X
                    child->x, child->y, child->f, parent->x, parent->y, parent->f);
            if(child->f <= parent->f && child->discovered != 1)
            {
                printf("\tfor (%d, %d): f:%d \tg:%d \th:%d\n",
                                child->x, child->y, child->f, child->g, heuristic(child, endx, endy));
                printf("hola\n");
                child->parent = parent;
                Node* tmp;
                bool found = false;
                for(int i = 0; i < frontier.size(); i++)
                {
                    //checking if the child nodes are already discovered
                    tmp = frontier.top();
                    frontier.pop();
                    if(tmp->x == child->x && tmp->y == child->y){
                        tmp->parent = parent;
                        frontier.push(tmp);
                        break;
                    }
                    frontier.push(tmp);
                }
                //printf("pushing: %d %d\n", child->x, child->y);
                frontier.push(child);


            }
            parent->discovered = 1;
        }

    }
    return expanded;
}

int IDA_STAR_helper(struct Node* parent,struct Node* target, int g, int bound, int* expanded)
{
    //check  the end conditions, discovered and whether we reached target
    if(parent->discovered){return 0;}
    else if(parent->x == target->x && parent->y == target->y){
            return FOUND;
    }

    parent->discovered = 1;
    parent->f = parent->g + heuristic(parent, target->x, target->y);


    printf("expanded %d times\n", *expanded);
    if(parent->f  > bound){return parent->f;}

    int x, y, min = 10000;
    for(int i=0; i<CHILD_NO; i++)
    {

        x = parent->x + rowstep[i]; y = parent->y + colstep[i];
        if(!is_valid(x, y)){continue;}
        Node* child = &board[get_index(x, y)];
NORMAL  ~/school/108/spring/ai/homework/Hw1/src/main.cpp   cpp  utf-8[unix]   64% ≡  274/422 ln  : 36  ≡ [14]tra…
```



```
        int move_cost = (abs(child->x - parent->x) + abs(child->y - parent->y));

        if(!child->discovered)
        {
            (*expanded)++; //increasing the amount of expanded nodes
            child->parent = parent;
            printf("testing (%d, %d) -> (%d, %d)\n", parent->x, parent->y, child->x, child->y);
            int t = IDA_STAR_helper(child, target, parent->g + move_cost, bound, expanded);
            if(t == FOUND)
            {
                return FOUND;
            }
            min = (t < min)?t:min;
        }

    }
    return min;
}
int IDA_STAR_search(int startx, int starty, int endx, int endy)
{
    Node* initial = &board[get_index(startx, starty)], *target = &board[get_index(endx, endy)];
    initial->parent = NULL;
    initial->f = heuristic(initial, endx, endy);
    initial->g = 0;
    int bound = initial->f, expanded= 0;
    std::list<Node*> frontier;
    while(1)
    {
        int t = IDA_STAR_helper(initial, target, 0, bound, &expanded);
        if( t == FOUND)
        {
            print_path(&board[get_index(endx, endy)], startx, starty);
            return expanded;
        }
        bound = t;
    }
    return expanded;
}

//calls the appropriate function for the search type
void search(int search_type, int startx, int starty,
                int endx, int endy)
{
    int expanded=0;
    reset_board();
    switch(search_type){
        case BFS:
            expanded = BFS_search(startx, starty, endx, endy);
            break;

        case DFS:
            expanded = DFS_search(startx, starty, endx, endy);
            break;
        case IDS:
```



```
        case IDS:
            expanded = IDS_search(startx, starty, endx, endy);
            break;

        case A_STAR:
            expanded = A_STAR_search(startx, starty, endx, endy);
            break;

        case IDA_STAR:
            expanded = IDA_STAR_search(startx, starty, endx, endy);
            break;
    }
    printf("\nWhen performing the algorithm, expanded %d nodes.\n",expanded );
}
void print_introduction()
{
    printf("-----\n"
            "Enter the number of the algorithm to use, and starting x,y and ending x,y  like:\n"
            " algorithm startx starty endx endy.\n"
            "The algorithms numbers are as follows:\n"
            "\t1.BFS\n"
            "\t2.DFS\n"
            "\t3.IDS\n"
            "\t4.A*\n"
            "\t5.IDA*\n"
            "-----\n\n");
}
int main(int argc, char** argv)
{
    print_introduction(); //show how to input the numbers
    int algorithm =1, startx, starty, endx, endy;

    Node n1, n2, n3;

    while(algorithm > 0){
        //scanf("%d %d %d %d", &algorithm, &startx, &starty, &endx, &endy);
        std::cin>>algorithm>>startx>>starty>>endx>>endy;
        printf("bout to search %d %d %d %d\n", startx, starty, endx, endy);
        search(algorithm, startx, starty, endx, endy);
    }
}
```