Introduction to AI Programming Assignment #1 Report

Student ID: 0716085 Name: 賴品樺

Code interpretation:

At first, I write 5 kind of algorithms in different code. Each kind of algorithm has different design of ***Block*** (structure of any square in the chessboard) and ***Frontier***. To combine 5 kind of algorithm in the same code, I union their structure and each algorithm only uses the components in the structure they need.
In the ***Block*** structure, there has 5 components. ***coordinate_p*** is a integer which records the parent coordinate, it used for tracing back the shortest path. ***min_step*** and ***min_fval*** are also integer, ***min_step*** used for recording the minimal steps to arrive this square from the start square, **min_fval** records minimal f(n) to arrive this square from the start square, n is the node (square in this problem). The last one is ***forward***, which is a boolean variable and used it to record whether the node been expanded or not.
In the **Frontier** structure, there has also 5 components. The function of ***coordinate_p*** is as same as the function in ***Block***. ***coordinate*** is an integer that records present ***Frontier*** coordinate. ***step*** and ***gval*** are also integer, ***gval*** records g(n), in the chessboard problem g(n) is the step of chess moving, ***step*** and ***gval*** record same thing in this problem. The last one is ***fval***, which is also an integer recording f(n).
The way I record coordinate in a single integer is coordinate x times 10 plus coordinate y.
To compare the different performance between each algorithm, I assign a variable called ***Frontier_count*** to calculate how many new nodes was expanded during the execution of the algorithm.
BFS: BFS only uses ***coordinate_p*** and ***forward*** in ***Block*** structure and ***coordinate_p*** and ***coordinate*** in ***Frontier*** structure, BFS uses the least memory compare to all other algorithm before combination. As a general form of BFS, I use queue in C++ STL to implement it. A node will be expanded if the ***forward*** value is false.
DFS: DFS uses ***coordinate_p*** and ***min_step*** in ***Block*** structure and all components except ***fval*** and ***gval*** in ***Frontier*** structure. Althougth many people use recursive function call to implement DFS, I use vector in C++

STL as stack structure to implement DFS to keep the structure of BFS. A node will be expanded if the ***min_step*** value is updated.
IDS: IDS is almost same as DFS. The only different between 2 algorithms is IDS has a variable ***depth*** to constrain the depth of DFS. A node will be expanded if the ***min_step*** value is updated and the new expanded node's depth can't bigger than ***depth***.
A*: A* uses ***coordinate_p*** and ***min_fval*** in ***Block*** structure and all components except ***step*** in ***Frontier*** structure. I use priority_queue in C++ STL to implement A*. A node will be expanded if the ***min_fval*** value is updated.
IDA*: IDA* is almost same as A*. IDA* is same as IDS that has variable ***depth***. A node will be expanded if the ***min_fval*** value is updated and the new expanded node's depth can't bigger than ***depth***.
The code is appended at the bottom of the report.

Testing result:

The heuristic function in this **testing result** section is as this assignment demand which is floor(($|x_1 - x_2| + |y_1 – y_2|$) / 3).
Here is the table of some test data and the correspond result of each algorithm, I will try to do the observation and conclusion by this table in the few next sections.

start	goal	BFS	DFS	IDS	A*	IDA*
(0,0)	(1,1)	283	877	67	92	162
(0,0)	(1,2)	3	877	4	3	4
(4,4)	(3,6)	33	929	10	21	10
(1,1)	(6,6)	397	901	123	136	229
(6,6)	(1,1)	416	888	95	105	217
(1,5)	(6,2)	433	908	111	134	268
(6,2)	(1,5)	432	864	139	99	227
(0,0)	(7,7)	691	877	266	165	509
(7,7)	(0,0)	691	881	231	133	471

The value in above table is the ***Frontier_count*** inferred in the **Code interpretation** section.

Observation of test:

First, I make an explanation why DFS appears to be the worst in this

problem. The reason why DFS expands such a great number of nodes than other algorithm is because there is no goal test in the algorithm. DFS can't ensure that once arrive goal then it is an optimal path, so the goal test can't exist in DFS. In this case, DFS is used for traversal. It needs to expanded all nodes which may construct optimal path.
Ranking the performance to 5 algorithms. In row 1 to row 7 in this table, A* seems to perform equal to IDS by this table. They both have great performance. Continue on is IDA* then is BFS and the last is DFS. In last 2 row, it shows that A* is better than IDS.

Conclusion of test:

I expected A*'s performance will overwhelm all other algorithms in all case at first, but the result shows that A* performs almost equal to IDS. Further, IDS can beat A* slightly in some cases, it surprises me a lot. The reason why it turns to this result is I put a strong constraint on the expanding rule of IDS. I don't expand a node which can't lead to the optimal path by comparing the step counting in ***Frontier*** to the best counting of step to next square that may been expanded. If the step counting in ***Frontier*** is bigger, which means I don't need to do the expanding because it can't get optimal path by doing it. In fact, I do the same constraint in DFS, but we can't clearly observe it because the reason inferred in the previous section. Moreover, IDS has another constraint called depth to limit the depth of DFS which makes it to own the advantage of BFS.
In the last 2 rows in the table above, it is the case that the furthest squares in the chessboard. It shows clearly that A* performs better than IDS. I make a assumption that the reason why row 1 to row 7 can't show the power of A* is because the advantage of A* can only appear in large case, this 7 cases is not large enough to show the power of A*.

Assume. A* can perform better than IDS in large case:

To check if this assumption holds true, I double the chessboard into 16x16. Hence, I need to modify the way I record coordinate into coordinate x times 100 plus coordinate y.
To simplify the analyzing, I don't use BFS and DFS in this section because these 2 algorithms will expand a great number of ***Frontier***, I only use IDS, A* and IDA* in this section.
Below is the table of experiment.

board	start	goal	IDS	A*	IDA*	IDS/A*
8x8	(0,0)	(4,4)	67	101	155	0.66
8x8	(0,0)	(7,7)	266	165	509	1.61
16x16	(0,0)	(4,4)	67	105	157	0.63
16x16	(0,0)	(7,7)	310	256	667	1.21
16x16	(0,0)	(8,8)	347	151	553	2.29
16x16	(0,0)	(12,12)	1078	397	1613	2.71
16x16	(0,0)	(15,15)	2240	411	3040	5.45
16x16	(15,0)	(0,15)	2212	455	3119	4.86

By this table, apparently, A* is the best in this chessboard problem. The last column in the table is the performance rate of IDS and A*. We can see that the large the problem is, A* can perform much better than IDS. The reason why A* can't defeat IDS in small problem is the heuristic function can't show enough difference to determine the order of node expanding, instead, heuristic function will become a burdensome thing.

Experiment of different heuristic function:

I also don't use BFS and DFS in this section.

1. Increase the influence of heuristic function:

h(n) = floor((dx| + |dy|) / 3),

f(n) = h(n) + g(n) ==> F(n) = 2h(n) + g(n)

chessboard type = 8x8

start	goal	IDS	A*(F)	IDA*(F)	A*(f)	IDA*(f)
(0,0)	(1,1)	67	70	136	92	162
(4,4)	(3,6)	10	21	10	21	10
(1,1)	(6,6)	123	56	130	136	229
(6,6)	(1,1)	95	61	133	105	217
(0,0)	(7,7)	266	81	380	165	509
(7,7)	(0,0)	231	83	382	133	471

The table shows that A* and IDA* perform better than before. Moreover, A* can defeat IDS in small case. This can proof my explanation in above assumption section that original heuristic function can't show enough difference to determine the order of node expanding. Once I double the influence of heuristic function, a clear order of node expanding can be determined, so it can find optimal path by expanding node less than before.

2. Using straight line distance as heuristic function:

h(n) = floor((|dx| + |dy|) / 3) ==> H(n) = sqrt(dx^2 + dy^2)

f(n) = h(n) + g(n) ==> F(n) = H(n) + g(n)
chessboard type = 8x8

start	goal	IDS	A*(F)	IDA*(F)	A*(f)	IDA*(f)
(0,0)	(1,1)	67	61	136	92	162
(4,4)	(3,6)	10	9	10	21	10
(1,1)	(6,6)	123	39	136	136	229
(6,6)	(1,1)	95	43	140	105	217
(0,0)	(7,7)	266	53	352	165	509
(7,7)	(0,0)	231	53	352	133	471

This table shows that A* and IDA* perform better than before. If we compare 2 table between experiment 1 and experiment 2. Experiment performs better than experiment 1, which means using straight line distance is better than double the influence of original heuristic function. However, there has change of type in experiment 2, I use double variable to store f(n) and h(n). This change can increase the precision of the order of node expanding. It comes up with a question, what will happen if I increase the precision of original heuristic function?

3. Increasing the precision of original heuristic function:

h(n) = floor((|dx| + |dy|) / 3) ==> H(n) = (|dx| + |dy|) / 3
f(n) = h(n) + g(n) ==> F(n) = H(n) + g(n)
chessboard type = 8x8

start	goal	IDS	A*(F)	IDA*(F)	A*(f)	IDA*(f)
(0,0)	(1,1)	67	76	148	92	162
(4,4)	(3,6)	10	9	10	21	10
(1,1)	(6,6)	123	81	191	136	229
(6,6)	(1,1)	95	84	205	105	217
(0,0)	(7,7)	266	153	497	165	509
(7,7)	(0,0)	231	137	476	133	471

In row 1 to row 4, it shows that the method of increasing the precision has the contribution of decreasing the node expanding. In last 2 rows, this method seems to be useless. It can easily connect this phenomenon to our common sense that precision is useful only in microcosm

Conclusion of experiment of different heuristic function:

1. If a heuristic function is good enough, we can increase the influence of it to achieve a better performance.
2. Use a better heuristic function can efficiently promote the performance.
3. Increasing the precision of heuristic function can help the order of node

expanding becomes more specifically.
To summarize, heuristic function can have a great impact on A* and IDA*, so the most important problem is how can we design a good enough and find the best heuristic function?

Appendix:

main.cpp:

```
#include <iostream>
#include <queue>
#include <vector>
#include <climits> // INT_MAX
#include <algorithm> // min
#include <cmath> //fabs

using namespace std;

// def. class structure, coordinate recorded by x * 10 + y
int GOAL = -1; // goal position
int h(int n){ // heuristic function
	int x = n / 10,y = n % 10;
	int gx = GOAL / 10,gy = GOAL % 10;
	return (int)floor((fabs((double)x - gx) + fabs((double)y - gy)) / 3.0);
}
class Block{
public:
	// parent coordindate,minimal step,minimal f(n)
	int coordinate_p,min_step,min_fval;
	bool forward; // expanding check
	Block(int cp = -1,bool f = true,int ms = INT_MAX,int mf =
INT_MAX){
		coordinate_p = cp;
		forward = f;
		min_step = ms;
		min_fval = mf;
	}
};
class Frontier{
public:
    // coordinate,parent coordindate,step,f(n),g(n)
    int coordinate,coordinate_p,step,fval,gval;
    Frontier(int c,int cp,int s){
        coordinate = c;
        coordinate_p = cp;
        gval = step = s; // step = gval
        fval = gval + h(c); // f(c) = g(c) + h(c)
    }
    // operator overloading for priority_queue
    friend bool operator>(const Frontier & obj1,const Frontier & obj2){
        return obj1.fval > obj2.fval;
    }
};
// algorithm func.
void BFS(int start,int finish,vector<vector<Block> > & table);
void DFS(int start,int finish,vector<vector<Block> > & table);
void IDS(int start,int finish,vector<vector<Block> > & table);
void A_star(int start,int finish,vector<vector<Block> > & table);
void IDA_star(int start,int finish,vector<vector<Block> > & table);

int Frontier_count = 0;
int main(int argc, char const *argv[]){
    vector<vector<Block> > table(8,vector<Block>(8));
    int toa,sx,sy,gx,gy;
    printf("type of algorithm:
BFS(1),DFS(2),IDS(3),A*(4),IDA*(5)\ninput type of algorithm: ");
    scanf("%d",&toa);
    if(5 < toa || toa < 1){
        printf("[x] Unexpected input!!!\n");
        return -1;
    }
    printf("input starting_x starting_y goal_x goal_y: ");
    scanf("%d%d%d%d",&sx,&sy,&gx,&gy);
    if(7 < sx || sx < 0 || 7 < sy || sy < 0 || 7 < gx || gx < 0 || 7 < gy || gy <
0){
        printf("[x] Unexpected input!!!\n");
        return -1;
    }
```

```
	int start = sx * 10 + sy,goal = gx * 10 + gy;
	GOAL = goal;
	if(toa == 1)
		BFS(start,goal,table);
	else if(toa == 2)
		DFS(start,goal,table);
	else if(toa == 3)
		IDS(start,goal,table);
	else if(toa == 4)
		A_star(start,goal,table);
	else if(toa == 5)
		IDA_star(start,goal,table);

	printf("\nFrontier_count =    %d\n\n",Frontier_count);

	printf("Table:\n");
	for(int i = 0;i < 8;i++){
		for(int j = 0;j < 8;j++){
			if(table[i][j].coordinate_p >= 0)
				printf("(%d,%d)",table[i][j].coordinate_p /
10,table[i][j].coordinate_p % 10);
			else
				printf("(-,-)");
		}
		printf("\n");
	}
	printf("\n");

	vector<int> path;
	int pos = goal;
	while(pos >= 0){
		path.push_back(pos);
		pos = table[pos / 10][pos % 10].coordinate_p;
	}
	for(int i = path.size() - 1;i >= 0;i--)
		printf("step %d: (%d,%d)\n",path.size() - i - 1,path[i] /
10,path[i] % 10);
```

```
	return 0;
}

// knight moving direction
int dx[] = {1,2,1,-2,-1,2,-1,-2};
int dy[] = {2,1,-2,1,2,-1,-2,-1};
// BFS
void BFS(int start,int finish,vector<vector<Block> > & table){
	queue<Frontier> q;
	q.push(Frontier(start,-1,-1)); // coordinate,coordinate_p,step = gval
	Frontier_count++;
	while(!q.empty()){
		Frontier f = q.front();
		q.pop();
		int x = f.coordinate / 10,y = f.coordinate % 10;
		table[x][y].forward = false;
		table[x][y].coordinate_p = f.coordinate_p;
		if(f.coordinate == finish) // goal test
			break;
		for(int i = 0;i < 8;i++){
			int nx = x + dx[i],ny = y + dy[i];
			if(0 <= nx && nx <= 7 && 0 <= ny && ny <= 7 &&
table[nx][ny].forward){
				q.push(Frontier(nx * 10 + ny,f.coordinate,-1));
				Frontier_count++;
			}
		}
	}
}
// DFS
void DFS(int start,int finish,vector<vector<Block> > & table){
	vector<Frontier> s; // stack
	s.push_back(Frontier(start,-1,0)); // coordinate,coordinate_p,step =
gval
	Frontier_count++;
	while(!s.empty()){
		Frontier f = s.back();
```

```
            s.pop_back();
            int x = f.coordinate / 10,y = f.coordinate % 10;
            if(f.step < table[x][y].min_step){
                table[x][y].min_step = f.step;
                table[x][y].coordinate_p = f.coordinate_p;
                for(int i = 0;i < 8;i++){
                    int nx = x + dx[i],ny = y + dy[i];
                    if(0 <= nx && nx <= 7 && 0 <= ny && ny <= 7 &&
f.step + 1 < table[nx][ny].min_step){
                        s.push_back(Frontier(nx * 10 +
ny,f.coordinate,f.step + 1));
                        Frontier_count++;
                    }
                }
            }
    }
}
// IDS
void IDS(int start,int finish,vector<vector<Block> > & table){
    bool goal = false;
    int depth = 1; // depth limit
    while(!goal){
        for(int i = 0;i < 64;i++)
            table[i / 8][i % 8] = Block(); // initial table
        vector<Frontier> s; // stack
        s.push_back(Frontier(start,-1,0)); //
coordinate,coordinate_p,step = gval
        Frontier_count++;
        while(!s.empty()){
            Frontier f = s.back();
            s.pop_back();
            int x = f.coordinate / 10,y = f.coordinate % 10;
            if(f.step < table[x][y].min_step){
                table[x][y].min_step = f.step;
                table[x][y].coordinate_p = f.coordinate_p;
                if(f.coordinate == finish){ // goal test
                    goal = true;
                    break;
                    }
                    for(int i = 0;i < 8;i++){
                        int nx = x + dx[i],ny = y + dy[i];
                        if(0 <= nx && nx <= 7 && 0 <= ny && ny <= 7
&& f.step + 1 < min(depth,table[nx][ny].min_step)){
                            s.push_back(Frontier(nx * 10 +
ny,f.coordinate,f.step + 1));
                            Frontier_count++;
                        }
                    }
                }
            }
            printf(goal?"Depth%d: Success\n":"Depth%d:
Failed\n",depth++);
        }
}
// A*
void A_star(int start,int finish,vector<vector<Block> > & table){
    priority_queue<Frontier,vector<Frontier>,greater<Frontier> > q;
    q.push(Frontier(start,-1,0)); // coordinate,coordinate_p,gval = step
    Frontier_count++;
    while(!q.empty()){
        Frontier f = q.top();
        q.pop();
        int x = f.coordinate / 10,y = f.coordinate % 10;
        if(f.coordinate == finish){ // goal test
            table[x][y].min_fval = f.fval;
            table[x][y].coordinate_p = f.coordinate_p;
            break;
        }
        if(f.fval < table[x][y].min_fval){
            table[x][y].min_fval = f.fval;
            table[x][y].coordinate_p = f.coordinate_p;
            for(int i = 0;i < 8;i++){
                int nx = x + dx[i],ny = y + dy[i];
                if(0 <= nx && nx <= 7 && 0 <= ny && ny <= 7 &&
f.fval + 1 < table[nx][ny].min_fval){
                    q.push(Frontier(nx * 10 + ny,f.coordinate,f.gval
+ 1));
                                        Frontier_count++;
                                }
                        }
                }
        }
}
// IDA*
void IDA_star(int start,int finish,vector<vector<Block> > & table){
        bool goal = false;
        int depth = 1; // depth limit
        while(!goal){
                for(int i = 0;i < 64;i++)
                        table[i / 8][i % 8] = Block(); // initial table
                priority_queue<Frontier,vector<Frontier>,greater<Frontier> >
q;
                q.push(Frontier(start,-1,0)); // coordinate,coordinate_p,gval =
step
                Frontier_count++;
                while(!q.empty()){
                        Frontier f = q.top();
                        q.pop();
                        int x = f.coordinate / 10,y = f.coordinate % 10;
                        if(f.coordinate == finish){ // goal test
                                table[x][y].min_fval = f.fval;
                                table[x][y].coordinate_p = f.coordinate_p;
                                goal = true;
                                break;
                        }
                        if(f.fval < table[x][y].min_fval){
                                table[x][y].min_fval = f.fval;
                                table[x][y].coordinate_p = f.coordinate_p;
                                for(int i = 0;i < 8;i++){
                                        int nx = x + dx[i],ny = y + dy[i];
                                        if(0 <= nx && nx <= 7 && 0 <= ny && ny <= 7
&& f.fval + 1 < table[nx][ny].min_fval && f.gval + 1 < depth){
                                                q.push(Frontier(nx * 10 +
ny,f.coordinate,f.gval + 1));
```

```
                                                Frontier_count++;
                                        }
                                }
                        }
                }
                printf(goal?"Depth%d: Success\n":"Depth%d:
Failed\n",depth++);
        }
}
```