**CS486 - Homework Assignment 1 Chen Lang 20528863**

1.1.

(a) Define the heuristic function of misplaced tiles heuristic is , and the heuristic function of Manhattan distance heuristic is .

Manhattan distance heuristic is expected to perform best. for all since any misplaced tile only contributes 1 to , but any misplaced tile will contribute at least 1 to based on the definition of each heuristic function.

Also, both of them are admissible, so Manhattan distance heuristic gives more close estimation compared with the real cost of a path to the goal.

(b) (1) Misplaced tiles heuristic is consistent.

Prove:

Define that we need to misplace tiles from state A to the goal, and we also need to misplace tiles from state B to the goal. Based on the rule of 8-puzzle, we can only move 1 tile in each step, so .

If one move from state A to state B reduces 1 misplaced tile, . In this case, .

If one move from state A to state B increases 1 misplaced tile, . In this case, .

So, for all tuples . Thus, Misplaced tils heuristic is consistent.

(2) Manhattan distance heuristic is consistent.

Prove:

Define that we need to move tiles for times from state A to the goal, and we also need to move tiles for times from state B to the goal. Based on the rule of 8-puzzle, we can only move 1 tile in each step, so .

If one move is a part of the path from state A to state B defined by Manhattan distance, . In this case, .

If one move is not a part of the path from state A to state B defined by Manhattan distance (i.e. taking a detour), . In this case, .

So, for all tuples . Thus, Manhattan distance heuristic is consistent.

1.2

(a) IDA\* has exponential time complexity. A\* search has exponential time complexity, so for each iteration of cost , time complexity is where c is a constant and ) is the depth of search. Thus, the total time is ;

IDA\* has linear space complexity. IDA\* search only keeps tracking on the nodes on the current searching path, and it does not save all the nodes it iterated like A\* search. Thus, IDA\* has linear space complexity.

(b) (1) IDA\* is complete if heuristic is consistent. Considering an iteration with the restriction of any cutoff value of , always increases along any path, so if there is any solution with a cost that is less than , will eventually reach it, and the solution will be found. If there is no solution found in this iteration with current cutoff value , the cutoff value will increase to . Eventually, the cost of the existed solution will be reached, and a solution will be found if it exists.

(2) IDA\* is not complete if heuristic is not consistent. Since searching will get stuck if there is a cycle branched from the searching path with a node that has smaller heuristic, but the node to the goal has much larger heuristic.

(c) (1) IDA\* is optimal if heuristic is consistent. Considering the base case of IDA\*, the cutoff value of is the heuristic value . If a solution is found in base case, which means , this solution is optimal since heuristic is consistent and is the minimum cost in this iteration.

So, we can make the induction hypothesis that IDA\* is optimal if heuristic is consistent for some .

Consider the cutoff value of . If no solution with cost is found where , the solution must have the cost that is larger than . This is because IDA\* always expands the nodes with larger cost only if all nodes with cost less than are explored. If a solution with cost is found where , the solution is optimal since optimal solution will be reported before the cutoff value of reaches if there is a solution has less cost than . This is guaranteed by induction hypothesis.

Thus, IDA\* is optimal if heuristic is consistent by induction.

(2) IDA\* is not optimal if heuristic is not consistent. By (b)(2), IDA\* is not complete if heuristic is not consistent. Since IDA\* is not complete under that condition, IDA\* is not optimal.

2.1

* 81 variables, where and . represents the row index, and represents the column index.
* Domain of each variable is .
* Constraints:

1. For all , where and ;
2. For all , where and ;
3. For all , and , where , , and .

2.2

(a) See the source files packed in the submitted zip file:

sudoku.cpp: Plain backtracking

sudoku\_fc.cpp: Backtracking + Forward checking

sudoku\_fc\_h.cpp: Backtracking + Forward checking + 3 heuristic function

Makefile: For the building purpose.

(b) Solution:

**medium**

5 4 8 6 1 3 7 2 9

3 2 9 4 5 7 8 6 1

1 6 7 8 9 2 5 4 3

6 7 5 9 2 1 4 3 8

4 3 2 5 7 8 9 1 6

8 9 1 3 6 4 2 5 7

9 8 3 1 4 5 6 7 2

2 5 6 7 3 9 1 8 4

7 1 4 2 8 6 3 9 5

**easy**

4 6 1 8 9 7 3 5 2

8 5 9 3 2 4 7 6 1

7 3 2 5 1 6 4 8 9

9 1 3 6 5 2 8 4 7

2 4 6 7 8 1 5 9 3

5 7 8 9 4 3 2 1 6

3 8 4 2 6 9 1 7 5

1 9 7 4 3 5 6 2 8

6 2 5 1 7 8 9 3 4

**evil**

7 6 1 8 2 5 3 4 9

3 5 2 9 6 4 8 7 1

9 4 8 7 1 3 6 5 2

4 8 7 5 3 2 9 1 6

1 9 5 6 8 7 2 3 4

2 3 6 4 9 1 5 8 7

5 2 3 1 7 9 4 6 8

8 1 4 2 5 6 7 9 3

6 7 9 3 4 8 1 2 5

**hard**

8 4 1 6 2 5 9 7 3

7 6 2 1 3 9 5 4 8

5 3 9 4 7 8 2 6 1

2 1 7 3 9 6 4 8 5

6 8 3 7 5 4 1 2 9

9 5 4 2 8 1 7 3 6

4 2 5 8 1 3 6 9 7

3 9 6 5 4 7 8 1 2

1 7 8 9 6 2 3 5 4

(c)

**Time (Seconds)**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **B** | **B+FC** | **B+FC+H** |
| **Easy** |  |  |  |
| **Medium** |  |  |  |
| **Hard** |  |  |  |
| **Evil** |  |  |  |

**# of Nodes**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **B** | **B+FC** | **B+FC+H** |
| **Easy** |  |  |  |
| **Medium** |  |  |  |
| **Hard** |  |  |  |
| **Evil** |  |  |  |

(d) When forward checking is used, the lists of assignable values for corresponding nodes will be generated. The nodes, which represent the variables will be explored in the next step, will be cut out from the searching path if no values can be assigned to these nodes based on the table. So, forward checking avoids exploring on a “dead” path in advance to reduce the explored nodes and searching time. Thus, the backtracking algorithm with forward checking explores less nodes and spends less time than the plain backtracking algorithm.

When 2 heuristic functions “most constrained variables” and “most constraining variables” are used, the algorithm picks up the most promising variable to assign a value. Based on the definition of these heuristic function, the picked variable has less uncertainty and assigning value to that variable reduces more uncertainty of its related variables compared with the rest variables, so attempts on a certain step is reduced. Also, by using “least constraining values” heuristic function, the picked value has the property of causing less effects to the its related variables. This allows the related variables to reduce conflicts and have more choices of selecting a value. Thus, the backtracking algorithm with both forward checking and 3 heuristic functions explores the least nodes and spends the least time.

When the techniques used in the searching algorithm is fixed, Sudoku problem with higher difficulties tends to explore more nodes and spend more time in solving. Based on the sample Sudoku problems, more difficult problem has more variables, which means more blanks are required to be filled. In a backtracking algorithm, the number of variables determine the depth of searching tree, so “more variables” means more nodes are required to be explored, and more time are consumed.

By randomly shuffling the order variables and values, the searching tree will have different depth and number of branches, so the algorithms without heuristic functions tend to have large standard deviation. By using heuristic functions, the order of explored variables and values will mainly depend on the current state, so the randomly shuffling will causes only tiny effects on the searching. Thus, the algorithm with heuristic functions have significantly small standard deviation on both explored nodes and consumed time.