Descsription and Experience of my project

Xiaoqing Tang

November 20, 2008

This scheme program can solve the N-puzzle problem, though N can't be too large. Generally it should be less than 5. The program can't guarantee for all inputs with N greater or equal to 5 that it can solve them in a reasonable time limit.

Technical Detail This section includes technical details of the program, including the implement of ID-DFS (IDS) for extra credit.

In general, the program is based on vectors. The advantage of using vectors is that I can get access to an item with a given index quickly and easily. Any state is a vector of the following items:

1. Board

   The board vector indicates the current state of the board, with 0 indicating the empty block. The length of this vector (as an item of the State vector) should be N*N.

2. step

   This number indicates how many steps it is needed to get to the current state from the initial step. If we have a state which has a final board distribution, the step field is the steps of moves we want.

3. Empty Block

   The Empty Block vector indicates where the empty block is. It is a vector of length 3, each indicates:

   (a) n: Board size

   (b) x: Row index of the empty block, $0 \leq x < n$

   (c) y: Column index of the empty block, $0 \leq y < n$

   The top-left block has the position of $x = 0$ and $y = 0$.

4. Current Sequence

   This list constains the sequence of moves in *reverse order* (since I can use `cons` to implement appending an element to the "end"). Each move

is represented in a capitalized character "U","D","L","R", for up, down, left, and right, respectively.

Through remarks in program, you can see that my program is divided into some parts, and each part containing some functions which behaves in the same category.

1. I/O functions

   These are just redirection of output ports, and customized `printf` so that we won't write the port object in each output sentence. (Mainly it's for debugging more easily)

2. Vector-related functions

   The `id` function is just an alias for the `vector-ref`, in order to type less. The `cp` function is used to create another instance for a vector. I've found that if I used the `id` function to get access to an element in a vector, or more specifically, get access to the board field of a state, and if I tried to use `set!` to change the value, the original vector would be changed. But I didn't want to do that because I need to create a new state to perform searching. Therefore I convert the vector to a list and convert the list back to vector. This indeed solves my problem because converting does create new instances. The last `x-y-to-id` function converts the position of empty block (x,y) into an index of board vector.

   The following part is functions that deal with states.

3. State-retrieving functions

   The 4 functions in this section are used to retrieve certain field of a state. It's just for convenience, just as what `id` is used for.

4. `equal-board?`

   This function judges whether two states have the same *board* vector. It is important because when we're scanning the open-list and closed-list to find out whether the state is already in the lists, we should only consider whether they have the same board state. Actually, we also have to consider the number of moves (the step field) because if the current state is equal-board to another state but has fewer moves, it should still be expanded. Therefore, this function is designed to implement only the simple match of board field, and leave the more complicated logic to the expand function.

5. `find?` and `swap`

   `find?` scans a list (which can be either open-list or closed-list) and returns whether there is a state which is equal-board to the given state, *and* the state in the list has fewer moves than the given state (rigiously less than or equal to). If it returns yes, it means that we needn't expand this state because we have expanded or will expand a state with the same board field but better solution.

`swap` swaps the empty block with a given block, increases the move counter(step field) by one, and add the symbol of move to the sequence field.

6. Four direction expansion

   Since I've written the `swap` function, I can call `swap` with different parameters to implement the expansion in four directions.

7. Generation of initial and final state

   Since the input is a vector of board, a function is needed to construct an initial state from the input board field. The function is init-state, using find-zero to find the empty block, and board-size to find the size of board (in my program, N is not part of input), i.e. my program can automatically determine those data from the board given.

   The final state is generated from the board field `(1 2 ... n*n-1 0)`

8. push functions

   `push-head` and `push-tail` are two different ways to push a state into a list. They're used by DFS and BFS seperately. Further discussion is on seperate search function section afterwards.

9. output function

   When a solution is found, this function is called to output the number of moves and the operating sequence. Since the sequence is in reverse order, the function output the sequence reversely so that the output is in correct order.

10. open-list and closed-list functions

    This section defines the open-list and closed-list, and a function that updates a list. The term "update" means, if such a state does not exist in the list, append the state into the list by the parameter function `push-func`. Otherwise, the new state overwrites the existing corresponding state in the list. The function returns the new list. There's one thing I would like to mention. The overwritting mechanism makes it possible for me to get rid of `rem-all` because the state I want to remove from open-list is the first element, and it won't have duplicates in `cdr` of the list.

11. **search-base function**

    This is the core of my program. Since DFS, BFS, along with ID-DFS has a similar searching framework, I abstracted the framework and treat it as a base function, like a base class in OOP. The parameters are `push-func`, `cut`, and `break-once-found`. Since I use the same way to pop a state from open-list (retrieve the first element in the list), I should use different pushing functions to distinguish DFS and BFS. This is why I implemented `push-head` and `push-tail`. For `cut`, it's a function used to cut states that needn't expanding. Each search performs different cut functions, so it should also be a parameter. The last `break-once-found` is a bool

parameter for BFS and ID-DFS. They have a feature that once it finds a solution, the search immediately quits because They can ensure that the first solution found is the best. Therefore, this bool should be true for them, and false for other searches (in fact, only for DFS in this case).

12. DFS, BFS, and ID-DFS functions

    Since search-base function is implemented as above, the DFS, BFS and ID-DFS are just calling the search-base function with different parameters. I treat the limit of search depth as a global variable just for convenience, similaly for the implementation of ID-DFS by calling DFS.

    `bfs-test` and `id-dfs-test` are used for finding the max level where the search can go.

COMPLEXITY It can be easily seen that the number of all possible boards (though not all of them are accessible) is $N^2!$. However, for DFS, if we restrain our depth to k, we have 4^k possible moves because there are 4 possible moves for one move. Therefore there at most 4^k different boards. So, for a DFS with a restrained depth, the number of possible boards is $\min\{N^2!, 4^k\}$.

To calculate the complexity for the program, we need to explore how many calculation needed to expand a state. Since I used brute force to find whether a state is in open-list or closed-list, the worst case is that I need to scan the two lists completely. However, for the sum of the lengths of the two lists, the worst case is that it exceedes the total number of possible moves. Therefore, if the number of possible moves is $f(n)$, the complexity will be $f(n)^2$. This means, the worst-case complexity is $O((N^2!)^2)$ for BFS and ID-DFS, $O((min\{N^2!, 4^k\})^2)$ for DFS (here k means the input of search depth limit).

However, in classical C/C++ programming, the finding of the existence of a state in lists can be implemented by balanced binary trees (BBST) or hash tables. The BBST can guarantee a $O(log(f(n)))$ searching, resulting in $O(f(n) \log f(n))$ complexity. The hash table is even better. Its $O(1)$ searching can result in $O(f(n))$ complexity, which is far better than my program.

However, the Big-O notation means that it's just the worst case complexity. The performance of search algorithms is affected greatly by the input data. Time spent on different input data may vary a lot even if they're of the same size.

Analysis of some extra issues

1. How many states?

   $N^2!$ for BFS and ID-DFS, and $\min\{N^2!, 4^k\}$ for DFS. These two numbers are both worst-case numbers and generally they won't be exceeded.

2. Implement IDS

   It's ID-DFS in my program.

3. Test the max-level

   By using `bfs-test` and `id-dfs-test` on the large input `board-test`, and running for several minutes before I get annoyed, it seems that for the 5*5 puzzle, we can achieve the level of 12 moves at most.

4. Hash-tables

   The memory usage is not acceptable for N is greater than 3. For a 4*4 puzzle, as the analysis above indicates, the number of possible states is $4^2! = 16! = 2.09228 \times 10^{13}$. Even if each state only occupies one byte, and the actual number of states visited is only 1/1000 of them, the memory usage will be $\frac{4^2!}{1000 \times 2^{30}} GB = 19.48GB$. Therefore, I refuse to implement hash-tables because even if I finally implement this, it won't solve the puzzles whose N is greater than 3.

   However, we can implement the closed-list by hash-table with chaining, with a relatively high load factor, i.e. the number of slots is limited, and each slot is considered as a scheme list containing states with the same hash value. This might work because it reduced a lot of memory usage. I think the memory usage for this implementation is almost the same as implementing in the original way (or actually we can see our original closed-list as a hash-table with only one slot). But the coding won't be easy, and the debugging will be probably very ugly. Therefore I gave up.

5. Limited-length closed-list

   Again the coding won't be easy so I didn't implement this. I'll just do a brief analysis. The idea sounds good, but it's likely to go infinite loops because it's very easy to construct a sequence of moves that do not actually change the board, and the number of the moves is greater than the closed-list size limit. Therefore, for a certain state S, and there are k(¿size-limit) moves that can turn S back to S. When the predecessor of S is being visited, S must not be in the closed-list because the only states in the closed-list is k ancestors of S. So, the new S, which is put into the closed-list again, has a new solution which is greater than the original state by k. Such infinite loops, if happens, will severely affect the algorithm.

6. Longest path

   As I've said before, there are sequences of moves which can preserve a state. Thus if we allow duplication of state, the answer will be $+\inf$. If duplication is not allowed, the problem will be very hard. At least, it can be reduce to a longest path problem on a graph with $N!$ vertices. So at least an $O(2^{N!})$ can be designed to solve this problem (though not feasible).

7. Other rules of expanding

   No matter the puzzle is toroidal, or a Klein bottle, or a projective plane, the only thing that changes is how a state which is being visited expands.

Originally such a state expands four adjacent states. New rules only add new adjacent states to the state and we have to expand more when visiting the state. Since I implemented the search framework in a base function, it's possible to keep the general framework and add only the new rules of expanding to it.