Algorithm Documentation

Breadth First Search without Visited List

	Details
State Representation	Puzzle object Puzzle(initialState, goalState); Original class. The representation of the puzzle's state in the Puzzle class is an integer, to allow for hashing. Also contains the path.
Q	Standard library queue of puzzle objects. std::queue<Puzzle> q; STL. The standard library implementation of a queue gives all of the functions necessary (push, push_back, pop, pop_back, etc.) and can be assumed to be very efficient given that it is in STL. The Puzzle objects hold the partial path in their path variable.
Pseudo code of your algorithm implementation	

```
create queue<Puzzle> q;
create Puzzle(initialState, goalState) object with initial
state;
add Puzzle object to queue<Puzzle>;
loop while solution not found and queue not empty {
    pop the front element off the queue;
    if goal state {
        save path;
        break;
    }
    increment number of expansions;
    for each direction L, U, R, D {
        if direction can be moved into {
            create new puzzle object;
            if locally visited {
                delete object;
                increment number of local loops avoided;
            }
            else add to the back of the queue;
        }
    }
    update max queue length value;
    if memory overload {
        return "memory overload";
    }
}
return path;
```

Breadth First Search with Visited List

	Details
State Representation	Puzzle object Puzzle(initialState, goalState); Original class. The representation of the puzzle's state in the Puzzle class is an integer, to allow for hashing. Also contains the path.
Q	Standard library queue of puzzle objects. std::queue<Puzzle> q; STL and original Puzzle class. The standard library implementation of a queue gives all of the functions necessary (push, push_back, pop, pop_back, etc.) and can be assumed to be very efficient given that it is in STL. The Puzzle objects hold the partial path in their path variable.
Visited List	Combination of maps and vectors. map<int, vector<vector<map<int, int>>>> visitedLists; STL. Initially this was just a vector, however the linear search through one vector was too slow. Adding another vector halved the number of states to search through (in worst case), however the performance gain was not enough. It seemed that vectors could only go two vectors deep, so I used maps to add further dividing of the states. The final map is (int, int) which is (state, depth). In this case, depth is always 0 as it is ignored.

Pseudo code of your algorithm implementation

```
create queue<Puzzle> q;
create map<int, vector<vector<map<int, int>>>>
visitedLists;
create Puzzle(initialState, goalState) object with initial
state;
add Puzzle object to queue<Puzzle>;
loop while solution not found and queue not empty {
    pop the front element off the queue;
    if goal state {
        save path;
        break;
    }
    increment number of expansions;
    for each direction L, U, R, D {
        if direction can be moved into {
            create new puzzle object;
            use digits of state to find appropriate visited
list to search;
            if state not yet visited {
                add to the back of the queue;
            }
            else delete object;
        }
    }
    update max queue length value;
    if memory overload {
        return "memory overload";
    }
}
```

	`return path;`
Extra work: Original Hash Function	Hash: Not exactly a hash, but an improvement on a straight linear search. The map<int, vector<vector<map<int, int>>>> structure allows for the 9! possible states to be divided by 9*3. This is due to the first digit of a puzzle's state being used to find which map<int, vector<vector<map<int, int>>>> entry to look at, the second digit for the vector<vector<map<int, int>>> entry, and the third digit for the vector<map<int, int>> entry. Instead of, for worst case, 362,880 states having to be linearly searched through, only a maximum of 4,480 have to be searched. As this does not use a hash table there are no collisions.

Progressive Deepening Search without Visited List

	Details
State Representation	Puzzle object Puzzle(initialState, goalState); Original class. The representation of the puzzle's state in the Puzzle class is an integer, to allow for hashing. Also contains the path.
Q	Standard library queue of puzzle objects. std::queue<Puzzle> q; STL and original Puzzle class. The standard library implementation of a queue gives all of the functions necessary (push, push_back, pop, pop_back, etc.) and can be assumed to be very efficient given that it is in STL. The Puzzle objects hold the partial path in their path variable.
Pseudo code of your algorithm implementation	

```
create queue<Puzzle> q;
initialise max depth to 1;
create Puzzle(initialState, goalState) object with initial
state;
add Puzzle object to queue<Puzzle>;
loop while solution not found {
    if queue empty and ultimate max depth not reached {
        increment max depth;
        create Puzzle(initialState, goalState) object with
initial state;
        add Puzzle object to queue<Puzzle>;
    }
    else if ultimate max depth reached {
        return "ultimate max depth reached";
    }
    pop the front element off the queue;
    if goal state {
        save path;
        break;
    }
    increment number of expansions;
    for each direction D, R, U, L {
        if direction can be moved into {
            create new puzzle object;
            if locally visited {
                delete object;
                increment number of local loops avoided;
            }
            else add to the front of the queue;
        }
    }
    update max queue length value;
    if memory overload {
        return "memory overload";
    }
}
return path;
```

Progressive Deepening Search with Non-Strict Visited List

	Details
State Representation	Puzzle object Puzzle(initialState, goalState); Original class. The representation of the puzzle's state in the Puzzle class is an integer, to allow for hashing. Also contains the path.
Q	Standard library queue of puzzle objects. std::queue<Puzzle> q; STL and original Puzzle class. The standard library implementation of a queue gives all of the functions necessary (push, push_back, pop, pop_back, etc.) and can be assumed to be very efficient given that it is in STL. The Puzzle objects hold the partial path in their path variable.
Non-Strict Visited List	Combination of maps and vectors. map<int, vector<vector<map<int, int>>>> visitedLists; STL. Initially this was just a vector, however the linear search through one vector was too slow. Adding another vector halved the number of states to search through (in worst case), however the performance gain was not enough. It seemed that vectors could only go two vectors deep, so I used maps to add further dividing of the states. The final map is (int, int) which is (state, depth).
Pseudo code of your algorithm implementation	see below

```
create queue<Puzzle> q;
create map<int, vector<vector<map<int, int>>>> visitedLists;
initialise max depth to 1;
create Puzzle(initialState, goalState) object with initial state;
add Puzzle object to queue<Puzzle>;
loop while solution not found {
    if queue empty and ultimate max depth not reached {
        increment max depth;
        clear visited lists;
        create Puzzle(initialState, goalState) object with initial state;
        add Puzzle object to queue<Puzzle>;
    }
    else if ultimate max depth reached {
        return "ultimate max depth reached";
    }
    pop the front element off the queue;
    if goal state {
        save path;
        break;
    }
    increment number of expansions;
    for each direction D, R, U, L {
        if direction can be moved into {
            create new puzzle object;
            use digits of state to find appropriate visited list to search;
            if state in visited list {
                if depth of current state less that depth
of entry in visited list {
                    add to the front of the queue;
                    update depth in visited list;
                }
                else delete object;
            }
            else add to the front of the queue;
        }
    }
    update max queue length value;
    if memory overload {
        return "memory overload";
    }
}
return path;
```

Extra work: Original Hash Function	Hash: Not exactly a hash, but an improvement on a straight linear search. The map<int, vector<vector<map<int, int>>>> structure allows for the 9! possible states to be divided by 9*3. This is due to the first digit of a puzzle's state being used to find which map<int, vector<vector<map<int, int>>>> entry to look at, the second digit for the vector<vector<map<int, int>>> entry, and the third digit for the vector<map<int, int>> entry. Instead of, for worst case, 362,880 states having to be linearly searched through, only a maximum of 4,480 have to be searched. As this does not use a hash table there are no collisions.

A* with Strict Expanded List – Misplaced Tiles

	Details
State Representation	Puzzle object Puzzle(initialState, goalState); Original class. The representation of the puzzle's state in the Puzzle class is an integer, to allow for hashing. Also contains the path.
Q	Heap Heap heap; Original class. Uses vector of Puzzle objects to simulate an "inverted" heap (lowest depth value puzzle on top). Elements can be inserted and sorted into heap, or deleted from top of heap after which heap is sorted. Alternatively, elements can be deleted from the middle of the heap.
Expanded List	Combination of maps and vectors. map<int, vector<vector<map<int, int>>>> expandedLists; STL. Initially this was just a vector, however the linear search through one vector was too slow. Adding another vector halved the number of states to search through (in worst case), however the performance gain was not enough. It seemed that vectors could only go two vectors deep, so I used maps to add further dividing of the states. The final map is (int, int) which is (state, depth). In this case, depth is always 0 as it is ignored.
Pseudo code of your algorithm implementation	(see below)

```
create Heap;
create map<int, vector<vector<map<int, int>>>>
expandedLists;
create Puzzle(initialState, goalState) object with initial
state;
update F cost based on misplaced tiles;
add Puzzle object to heap;
loop while solution not found and queue not empty {
     get the top element in the heap;
     sort heap;
     if goal state {
          save path;
          break;
     }
     else {
          use digits of state to find expanded list to add
to;
          add path to expanded list;
          delete any other elements in heap with same state
and save value as number of deletions from middle of heap;
          increment number of expansions;
     }
     for each direction L, U, R, D {
          if direction can be moved into {
               create new puzzle object;
               use digits of state to find appropriate
expanded list to search;
               if state not yet expanded {
                    update F cost based on misplaced tiles;
```

```
                add object into heap;
                sort heap;
            }
            else delete object;
        }
    }
    update max heap length value;
    if memory overload {
        return "memory overload";
    }
}
return path;
```

Extra work (Bonus): Original Heap Data Structure	Heap: The heap is actually two heaps: one containing the Puzzle objects and one with just the integer representation of the Puzzles' states. Both are sorted exactly the same, allowing for faster searching of the integer heap to find elements of the same state. These elements of the same state as a currently expanding state can be deleted from the middle of the heap by copying the last element in the heap to the position of the element to delete, reducing the size of the heap by one, and sorting the moved element back down the heap.
Original Hash Function	Hash: Not exactly a hash, but an improvement on a straight linear search. The map<int, vector<vector<map<int, int>>>> structure allows for the 9! possible states to be divided by 9*3. This is due to the first digit of a puzzle's state being used to find which map<int, vector<vector<map<int, int>>>> entry to look at, the second digit for the vector<vector<map<int, int>>> entry, and the third digit for the vector<map<int, int>> entry. Instead of, for worst case, 362,880 states having to be linearly searched through, only a maximum of 4,480 have to be searched. As this does not use a hash table there are no collisions.

A* with Strict Expanded List – Manhattan Distance

	Details
State Representation	Puzzle object Puzzle(initialState, goalState); Original class. The representation of the puzzle's state in the Puzzle class is an integer, to allow for hashing. Also contains the path.
Q	Heap Heap heap; Original class. Uses vector of Puzzle objects to simulate an "inverted" heap (lowest depth value puzzle on top). Elements can be inserted and sorted into heap, or deleted from top of heap after which heap is sorted. Alternatively, elements can be deleted from the middle of the heap.
Expanded List	Combination of maps and vectors. map<int, vector<vector<map<int, int>>>> expandedLists; STL. Initially this was just a vector, however the linear search through one vector was too slow. Adding another vector halved the number of states to search through (in worst case), however the performance gain was not enough. It seemed that vectors could only go two vectors deep, so I used maps to add further dividing of the states. The final map is (int, int) which is (state, depth). In this case, depth is always 0 as it is ignored.
Pseudo code of your algorithm implementation	see below

```
create Heap;
create map<int, vector<vector<map<int, int>>>>
expandedLists;
create Puzzle(initialState, goalState) object with initial
state;
update F cost based on Manhattan Distance;
add Puzzle object to heap;
loop while solution not found and queue not empty {
    get the top element in the heap;
    sort heap;
    if goal state {
        save path;
        break;
    }
    else {
        use digits of state to find expanded list to add
to;
        add path to expanded list;
        delete any other elements in heap with same state
and save value as number of deletions from middle of heap;
        increment number of expansions;
    }
    for each direction L, U, R, D {
        if direction can be moved into {
            create new puzzle object;
            use digits of state to find appropriate
expanded list to search;
            if state not yet expanded {
                update F cost based on Manhattan Distance;
                add object into heap;
                sort heap;
            }
            else delete object;
        }
    }
    update max heap length value;
    if memory overload {
        return "memory overload";
    }
}
return path;
```

Extra work (Bonus): Original Heap Data Structure	Heap: The heap is actually two heaps: one containing the Puzzle objects and one with just the integer representation of the Puzzles' states. Both are sorted exactly the same, allowing for faster searching of the integer heap to find elements of the same state. These elements of the same state as a currently expanding state can be deleted from the middle of the heap by copying the last element in the heap to the position of the element to delete, reducing the size of the heap by one, and sorting the moved element back down the heap.
Original Hash Function	Hash: Not exactly a hash, but an improvement on a straight linear search. The map<int, vector<vector<map<int, int>>>> structure allows for the 9! possible states to be divided by 9*3. This is due to the first digit of a puzzle's state being used to find which map<int, vector<vector<map<int, int>>>> entry to look at, the second digit for the vector<vector<map<int, int>>> entry, and the third digit for the vector<map<int, int>> entry. Instead of, for worst case, 362,880 states having to be linearly searched through, only a maximum of 4,480 have to be searched. As this does not use a hash table there are no collisions.