**Checklist/Other technical details**

Please accomplish the following check list in order to allow for accurate marking of your assignment.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Item** | **your assignment details** | |
| 1 | Names and ID numbers of Group Members | Jordan Drumm - 17044923  Benjamin Upton - 16463710  Zane Lamb - 15160640 | |
| 2 | Operating System used for testing your codes | Windows 10 Pro | |
| 3 | Compiler used | gcc 8.2 | |
| 4 | IDE used | SublimeText 3 | |
| 5 | Complete source codes (cpp, h files), makefile | BFS\_NO\_VLIST | full |
| BFS\_VLIST | full |
| PDS\_NO\_VLIST | full |
| UNIFORM COST\_EXP\_LIST | full |
| ASTAR\_EXP\_LIST MISPLACED TILES HEURISTIC | full |
| ASTAR\_EXP\_LIST MANHATTAN DISTANCE HEURISTIC | full |
| 6 | Is your program able to run with the 2 batch files given? | Yes | |
| 7 | Experiment Results in Excel Worksheet | Yes | |
| 8 | Extra work (Bonus): Enhancements/Optimisations included | Yes. Both the Heap data structure and hashtable was implemented. | |

**ALGORITHM DOCUMENTATION**

|  |  |
| --- | --- |
| **Algorithm Name: Breadth First Search - No Visited List** | **Details** |
| State Representation | Puzzle class. Maintains a deque named expansionPath to record the path each piece has taken from the initial state. |
| Q (partial paths container) | A queue (STL) named Q is used to record the partial paths, or states to be expanded. |
| Expanded List (if applicable) | N/A |
| Pseudo code of your algorithm implementation | The current state being observed is set to the initial state  Then begin looping: while the current state does not match the goal state:   * In order: up, right, down, left examine if the current state can make the directional move then if so: - check if the move would create a state already in the state’s path. If not, then add the future state onto the Q. * Check if Q is empty (if it is, there are no new states and no solution can be found). * Otherwise select the next state to check by choosing the front of the Q.   When the loop has finished, check whether a result was found, and register the path of the successful path if it exists. |
| Extra work (Bonus):  Original Heap data structure   * Specify how you are able to delete elements in the middle of the heap   Original Hash Function   * Specify and explain your formula. Does it guarantee that there will be no duplicate hash values? | N/A for this algorithm |

|  |  |
| --- | --- |
| **Algorithm Name: Breadth First Search - with Visited List** | **Details** |
| State Representation | Puzzle2 class, an original implementation. Differs from the original Puzzle class as it does not record its path as this will be covered by the visited list which we have implemented with the hashtable. |
| Q (partial paths container) | Q is represented as a queue(STL container) to manage partial paths, or states to be expanded . |
| Expanded List (if applicable) | N/A |
| Pseudo code of your algorithm implementation | The current state being observed is set to the initial state  Then begin looping: while the current state does not match the goal state:   * In order: up, right, down, left examine if the current state can make the directional move then if so: - check if the move would create a state already in the hashtable. If not, then add the future state onto the Q. * Check if Q is empty (if it is, there are no new states and no solution can be found). * Otherwise select the next state to check by choosing the front of the Q.   When the loop has finished, check whether a result was found, and register the path of the successful path if it exists. |
| Extra work (Bonus):  Original Heap data structure   * Specify how you are able to delete elements in the middle of the heap   Original Hash Function   * Specify and explain your formula. Does it guarantee that there will be no duplicate hash values? | Hash - As each state was an original number, we didn’t hash it and instead just converted it to an integer such that we could work with it. When referencing our hashtable, it included an array of 370000 Node pointers (slightly more than the maximum number of possible states reachable from any given starting position). We took the remainder of the state by 370000 to produce an individual index where we checked the corresponding node to determine if the state had been visited. Our implementation doesn’t guarantee that a unique index will be found however it does avoid all possible collisions as each index in the hashtable is a pointer to a linked list of Nodes each holding a visited/expanded state. Therefore, if multiple states reference the same index, both of them can share said index (This is however very uncommon and just a failsafe). |

|  |  |
| --- | --- |
| **Algorithm Name: Progressive Deepening Search (No Visited List)** | **Details** |
| State Representation | Puzzle class. Maintains a deque named expansionPath to record the path each piece has taken from the initial state. |
| Q (partial paths container) | Q is represented as a stack (STL container) to manage partial paths, or states to be expanded . |
| Expanded List (if applicable) | N/A |
| Pseudo code of your algorithm implementation | Start by assigning the current state to the initial state and setting the current depth to 1.  Then begin looping until the goal state is found:   * In order: up, right, down, left examine if the current state can make the directional move with regard to the current depth being observed, then if so: - check if the move would create a state already in the state’s path. If not, then add the future state onto the Q. * Check if Q is empty. If so, increment the current depth. Then, if the current depth exceeds the max depth, there is no solution. If the current depth is within the max depth however, restart the loop with the new depth and the initial state once more. * Otherwise, if Q was not empty, select the next state to check by choosing the top of the Q.   When the loop has finished, check whether a result was found, and register the path of the successful path if it exists. |
| Extra work (Bonus):  Original Heap data structure   * Specify how you are able to delete elements in the middle of the heap   Original Hash Function   * Specify and explain your formula. Does it guarantee that there will be no duplicate hash values? | N/A for this algorithm. |

|  |  |
| --- | --- |
| **Algorithm Name: Uniform Cost Search** | **Details** |
| State Representation | Puzzle2 class, an original implementation. Differs from the original Puzzle class as it does not record its path as this will be covered by the visited list which we have implemented with the hashtable. |
| Q (partial paths container) | Q is represented as a min heap which was an an original implementation, to manage partial paths, or states to be expanded . |
| Expanded List (if applicable) | Implemented using our original implementation of a hashtable. Named ExpandedList, records all states that have already been expanded. |
| Pseudo code of your algorithm implementation | The current state being observed is set to the initial state  Then begin looping: while the current state does not match the goal state:   * check if current state has already been expanded, by searching the expanded list then if so:   + delete current state grab root of Q * else in order: up, right, down, left examine if the current state can make the directional move then if so: - check if the move would create a state already in the hashtable. If not, then update the fcost and add the future state onto Q. * Check if Q is empty (if it is, there are no new states and no solution can be found). * Otherwise select the next state to check by choosing the root of Q.   When the loop has finished, check whether a result was found, and register the path of the successful path if it exists. |
| Extra work (Bonus):  Original Heap data structure   * Specify how you are able to delete elements in the middle of the heap   Original Hash Function   * Specify and explain your formula. Does it guarantee that there will be no duplicate hash values? | Hash - As each state was an original number, we didn’t hash it and instead just converted it to an integer such that we could work with it. When referencing our hashtable, it included an array of 370000 Node pointers (slightly more than the maximum number of possible states reachable from any given starting position). We took the remainder of the state by 370000 to produce an individual index where we checked the corresponding node to determine if the state had been visited. Our implementation doesn’t guarantee that a unique index will be found however it does avoid all possible collisions as each index in the hashtable is a pointer to a linked list of Nodes each holding a visited/expanded state. Therefore, if multiple states reference the same index, both of them can share said index (This is however very uncommon and just a failsafe).  Heap - The heap is implemented as a min heap. We based the sorting off of the heuristic value associated with each puzzle piece, keeping the smallest value as the root of the heap thus allowing for efficient retrieval of the next state for exploring. We allowed for insertions from the middle of the heap by replacing the deleted value with the value positioned at the last point in the heap and then resorting accordingly (whether it was smaller than the parents or larger than it’s children etc). |

|  |  |
| --- | --- |
| **Algorithm Name: A\*** | **Details** |
| State Representation | Puzzle2 class, an original implementation. Differs from the original Puzzle class as it does not record its path as this will be covered by the visited list which we have implemented with the hashtable. |
| Q (partial paths container) | Q is represented as a min heap which was an an original implementation, to manage partial paths, or states to be expanded . |
| Expanded List (if applicable) | Implemented using our original implementation of a hashtable. Named ExpandedList, records all states that have already been expanded. |
| Pseudo code of your algorithm implementation | The current state being observed is set to the initial state  Then begin looping: while the current state does not match the goal state:   * Check if the state has already been expanded. If it has, get the next state and restart the loop. If not, add it to the expanded list and continue * In order: up, right, down, left examine if the current state can make the directional move then if so: - check if the move would create a state already in the expanded list. If not, then add the future state onto the heap and sort the heap accordingly. The heuristic value is decided using either manhattan distance or misplaced tiles. * Check if Q is empty (if it is, there are no new states and no solution can be found). * Otherwise select the next state with the next lowest heuristic (the root of the heap)   When the loop has finished, check whether a result was found, and register the path of the successful path if it exists. |
| Extra work (Bonus):  Original Heap data structure   * Specify how you are able to delete elements in the middle of the heap   Original Hash Function   * Specify and explain your formula. Does it guarantee that there will be no duplicate hash values? | Hash - As each state was an original number, we didn’t hash it and instead just converted it to an integer such that we could work with it. When referencing our hashtable, it included an array of 370000 Node pointers (slightly more than the maximum number of possible states reachable from any given starting position). We took the remainder of the state by 370000 to produce an individual index where we checked the corresponding node to determine if the state had been visited. Our implementation doesn’t guarantee that a unique index will be found however it does avoid all possible collisions as each index in the hashtable is a pointer to a linked list of Nodes each holding a visited/expanded state. Therefore, if multiple states reference the same index, both of them can share said index (This is however very uncommon and just a failsafe).  Heap - The heap is implemented as a min heap. We based the sorting off of the heuristic value associated with each puzzle piece, keeping the smallest value as the root of the heap thus allowing for efficient retrieval of the next state for exploring. We allowed for insertions from the middle of the heap by replacing the deleted value with the value positioned at the last point in the heap and then resorting accordingly (whether it was smaller than the parents or larger than it’s children etc). |