Breadth first search:

|  |  |  |
| --- | --- | --- |
| **Algorithm Name:** | **Details** | **Instructions** |
| State Representation | using Map\_t = Map<width, height>;  Map\_t& currentState;  It is an original implementation and I used it so that I don’t have to re-find the position of the empty tile. | Specify the data structure you used. Write the variable declarations. Indicate if it’s an original implementation of a data structure, or if you are using a library (e.g. STL, etc.) Briefly explain why the data structure you picked is appropriate. |
| Q (partial paths container) | Deque<ListNode<Map\_t>\*> posiblePaths;  Deque is an original implementation. I used it because it supports O(1) push\_back, frond, and pop\_front. |
| Expanded List (if applicable) |  |
| Non-Strict Visited List (if applicable) |  |
| Pseudo code of your algorithm implementation | Deque posiblePaths;  posiblePaths.push\_back(start);  bool goalFound = false;  while(!posiblePaths.empty())  {  auto current = posiblePaths.front();  if(current == goal)  {  goalFound = true;  break;  }  posiblePaths.pop\_front();  if(current.canMoveRight())  {  posiblePaths.push\_back(current.moveRight());  }  if(current.canMoveDown())  {  posiblePaths.push\_back(current.moveDown());  }  if(current.canMoveLeft())  {  posiblePaths.push\_back(current.moveLeft());  }  if(current.canMoveUp())  {  posiblePaths.push\_back(current.moveUp());  }  }  if(goalFound)  {  return movesFromPath(posiblePaths.front());  }  else return “”; | Specify the main looping mechanisms involved, conditional statements, as well as the main variables. Do not copy and paste what we have in the lecture slides. This should reflect the skeleton of your own implementation of the algorithm. |
| 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? |  | (e.g. original implementation of heap data structure, original hash function implementation) |

Breadth first search with visited list:

|  |  |  |
| --- | --- | --- |
| **Algorithm Name:** | **Details** | **Instructions** |
| State Representation | using Map\_t = Map<width, height>;  Map\_t& currentState;  It is an original implementation and I used it so that I don’t have to re-find the position of the empty tile. | Specify the data structure you used. Write the variable declarations. Indicate if it’s an original implementation of a data structure, or if you are using a library (e.g. STL, etc.) Briefly explain why the data structure you picked is appropriate. |
| Q (partial paths container) | Deque<ListNode<Map\_t>\*> posiblePaths;  Deque is an original implementation. I used it because it supports O(1) push\_back, frond, and pop\_front. |
| Expanded List (if applicable) |  |
| Non-Strict Visited List (if applicable) |  |
| Pseudo code of your algorithm implementation | Deque posiblePaths;  HashSet visited;  posiblePaths.push\_back(start);  bool goalFound = false;  while(!posiblePaths.empty())  {  auto current = posiblePaths.front();  if(current == goal)  {  goalFound = true;  break;  }  posiblePaths.pop\_front();  if(current.canMoveRight()  && !visited.contains(current.moveRight()))  {  posiblePaths.push\_back(current.moveRight());  visited.insert(current.moveRight());  }  if(current.canMoveDown()  && !visited.contains(current.moveDown())  {  posiblePaths.push\_back(current. moveDown ());  posiblePaths.push\_back(current.moveDown());  }  if(current.canMoveLeft()  && !visited.contains(current. moveLeft ())  {  posiblePaths.push\_back(current. moveLeft ());  posiblePaths.push\_back(current.moveLeft());  }  if(current.canMoveUp()  && !visited.contains(current. moveUp ())  {  posiblePaths.push\_back(current. moveUp ());  posiblePaths.push\_back(current.moveUp());  }  }  if(goalFound)  {  return movesFromPath(posiblePaths.front());  }  else return “”; | Specify the main looping mechanisms involved, conditional statements, as well as the main variables. Do not copy and paste what we have in the lecture slides. This should reflect the skeleton of your own implementation of the algorithm. |
| 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? | Original Hash Function:  Convert the string version of the state into a size\_t as if it’s a number in base 10. This creates a unique hash. | (e.g. original implementation of heap data structure, original hash function implementation) |

Progressive deepening search:

|  |  |  |
| --- | --- | --- |
| **Algorithm Name:** | **Details** | **Instructions** |
| State Representation | using Map\_t = MapWithDepth<width, height>; Map\_t& currentState;  It is an original implementation and I used it so that I don’t have to re-find the position of the empty tile and to store the number of moves taken to get to this state. | Specify the data structure you used. Write the variable declarations. Indicate if it’s an original implementation of a data structure, or if you are using a library (e.g. STL, etc.) Briefly explain why the data structure you picked is appropriate. |
| Q (partial paths container) | Deque<ListNode<Map\_t>\*> posiblePaths;  Deque is an original implementation. I used it because it supports O(1) push\_back, back, and pop\_back. |
| Expanded List (if applicable) |  |
| Non-Strict Visited List (if applicable) |  |
| Pseudo code of your algorithm implementation | Deque posiblePaths;  posiblePaths.push\_back(start);  bool goalFound = false;  unsigned int maxDepth = 1;  while (true)  {  bool solutionMightExist = false;  while(!posiblePaths.empty())  {  auto current = posiblePaths.back();  if(current == goal)  {  goalFound = true;  break;  }  posiblePaths.pop\_back();  if(current.depth == maxDepth)  {  solutionMightExist = true;  continue;  }  if(current.canMoveRight())  {  posiblePaths.push\_back(  current.moveRight());  }  if(current.canMoveDown())  {  posiblePaths.push\_back(  current.moveDown());  }  if(current.canMoveLeft())  {  posiblePaths.push\_back(  current.moveLeft());  }  if(current.canMoveUp())  {  posiblePaths.push\_back(  current.moveUp());  }  }  if (goalFound || !solutionMightExist) break;  ++maxDepth;  posiblePaths.push\_back(start);  }  if(goalFound)  {  return movesFromPath(posiblePaths.front());  }  else return “”; | Specify the main looping mechanisms involved, conditional statements, as well as the main variables. Do not copy and paste what we have in the lecture slides. This should reflect the skeleton of your own implementation of the algorithm. |
| 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? |  | (e.g. original implementation of heap data structure, original hash function implementation) |

Progressive deepening search with non-strict visited list:

|  |  |  |
| --- | --- | --- |
| **Algorithm Name:** | **Details** | **Instructions** |
| State Representation | using Map\_t = MapWithDepth<width, height>; Map\_t& currentState;  It is an original implementation and I used it so that I don’t have to re-find the position of the empty tile and to store the number of moves taken to get to this state. | Specify the data structure you used. Write the variable declarations. Indicate if it’s an original implementation of a data structure, or if you are using a library (e.g. STL, etc.) Briefly explain why the data structure you picked is appropriate. |
| Q (partial paths container) | Deque<ListNode<Map\_t>\*> posiblePaths;  Deque is an original implementation. I used it because it supports O(1) push\_back, back, and pop\_back. |
| Expanded List (if applicable) |  |
| Non-Strict Visited List (if applicable) | HashMap<Map\_t\*, Hasher<Map\_t, 10>,  EqualityTester<Map\_t>> visited;  HashMap is an original implementation that I used because supports average case O(1) insert and find. |
| Pseudo code of your algorithm implementation | Deque posiblePaths;  HashMap visited;  posiblePaths.push\_back(start);  visited.insert(start);  bool goalFound = false;  unsigned int maxDepth = 1;  while (true)  {  bool solutionMightExist = false;  while(!posiblePaths.empty())  {  auto current = posiblePaths.back();  if(current == goal)  {  goalFound = true;  break;  }  posiblePaths.pop\_back();  if(current.depth == maxDepth)  {  solutionMightExist = true;  continue;  }  if(current.canMoveRight())  {  auto visitedState = visited.find(  current.moveRight());  if(visitedState != visited.end())  {  if((\*visitedState).depth > current.depth)  {  \*visitedState = current;  }  }  else  {  visited.insert(current.moveRight());  posiblePaths.push\_back(  current.moveRight());  }  }  if(current.canMoveDown())  {  auto visitedState = visited.find(  current.moveDown());  if(visitedState != visited.end())  {  if((\*visitedState).depth > current.depth)  {  \*visitedState = current;  }  }  else  {  visited.insert(current.moveDown ());  posiblePaths.push\_back(  current.moveDown ());  }  }  if(current.canMoveLeft())  {  auto visitedState = visited.find(  current.moveLeft());  if(visitedState != visited.end())  {  if((\*visitedState).depth > current.depth)  {  \*visitedState = current;  }  }  else  {  visited.insert(current.moveLeft());  posiblePaths.push\_back(  current.moveLeft());  }  }  if(current.canMoveUp())  {  auto visitedState = visited.find(  current.moveUp());  if(visitedState != visited.end())  {  if((\*visitedState).depth > current.depth)  {  \*visitedState = current;  }  }  else  {  visited.insert(current.moveUp());  posiblePaths.push\_back(  current.moveUp());  }  }  }  if (goalFound || !solutionMightExist) break;  ++maxDepth;  posiblePaths.push\_back(start);  }  if(goalFound)  {  return movesFromPath(posiblePaths.front());  }  else return “”; | Specify the main looping mechanisms involved, conditional statements, as well as the main variables. Do not copy and paste what we have in the lecture slides. This should reflect the skeleton of your own implementation of the algorithm. |
| 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? | Original Hash Function:  Convert the string version of the state into a size\_t as if it’s a number in base 10. This creates a unique hash. | (e.g. original implementation of heap data structure, original hash function implementation) |

A\* using expanded list:

|  |  |  |
| --- | --- | --- |
| **Algorithm Name:** | **Details** | **Instructions** |
| State Representation | using Map\_t = MapWithHuristic<width, height>;  Map\_t& currentState;  It is an original implementation and I used it so that I don’t have to re-find the position of the empty tile. | Specify the data structure you used. Write the variable declarations. Indicate if it’s an original implementation of a data structure, or if you are using a library (e.g. STL, etc.) Briefly explain why the data structure you picked is appropriate. |
| Q (partial paths container) | using QueueElement =  LocationTracker<ListNode<Map\_t>\*>;  using QueueType = PriorityQueue<QueueElement,  Vector<QueueElement>,  GreaterHeuisticTracker<Map\_t>>;  QueueType posiblePaths;  PriorityQueue is an original implementation based on a heap. I used it because it supports O(log(n)) pop\_top, priortyIncreased and insert and O(1) top. |
| Expanded List (if applicable) | HashMap<Map\_t\*, Hasher<Map\_t>,  EqualityTester<Map\_t>> expanded;  HashMap is an original implementation that I used because supports average case O(1) insert and find. |
| Non-Strict Visited List (if applicable) |  |
| Pseudo code of your algorithm implementation | PriorityQueue posiblePaths;  HashMap expanded;  HashMap queueLookup;  posiblePaths.insert(start);  queueLookup.insert(posiblePaths.top());  bool goalFound = false;  while(!posiblePaths.empty())  {  auto current = posiblePaths.top();  queueLookup.erase(current);  if(current == goal)  {  goalFound = true;  break;  }  posiblePaths.pop();  if(current.canMoveRight())  {  auto right = current.moveRight();  if(!expanded.contains(right))  {  right.updateHeuristic(current,  heuristicFunction);  auto posInQueue = queueLookup.find(right)  if(posInQueue != queueLookup.end())  {  If(posInQueue->f > right.f)  {  posInQueue->f = right.f;  posInQueue->g = right.g;  posInQueue->path = right.path;  posiblePaths.priorityIncreased(  posInQueue->pos);  }  }  else  {  posiblePaths.push(right);  queueLookup.insert(right);  }  }  }  if(current.canMoveDown())  {  auto down = current.moveDown();  if(!expanded.contains(down))  {  down.updateHeuristic(current,  heuristicFunction);  auto posInQueue = queueLookup.find(down)  if(posInQueue != queueLookup.end())  {  If(posInQueue->f > down.f)  {  posInQueue->f = down.f;  posInQueue->g = down.g;  posInQueue->path = down.path;  posiblePaths.priorityIncreased(  posInQueue->pos);  }  }  else  {  posiblePaths.push(down);  queueLookup.insert(down);  }  }  }  if(current.canMoveLeft())  {  auto left = current.moveLeft();  if(!expanded.contains(left))  {  left.updateHeuristic(current,  heuristicFunction);  auto posInQueue = queueLookup.find(left)  if(posInQueue != queueLookup.end())  {  If(posInQueue->f > left.f)  {  posInQueue->f = left.f;  posInQueue->g = left.g;  posInQueue->path = left.path;  posiblePaths.priorityIncreased(  posInQueue->pos);  }  }  else  {  posiblePaths.push(left);  queueLookup.insert(left);  }  }  }  if(current.canMoveUp())  {  auto up = current.moveUp();  if(!expanded.contains(up))  {  right.updateHeuristic(current,  heuristicFunction);  auto posInQueue = queueLookup.find(up)  if(posInQueue != queueLookup.end())  {  If(posInQueue->f > up.f)  {  posInQueue->f = up.f;  posInQueue->g = up.g;  posInQueue->path = up.path;  posiblePaths.priorityIncreased(  posInQueue->pos);  }  }  else  {  posiblePaths.push(up);  queueLookup.insert(up);  }  }  }  expanded.insert(current);  }  if(goalFound)  {  return movesFromPath(posiblePaths.front());  }  else return “”; | Specify the main looping mechanisms involved, conditional statements, as well as the main variables. Do not copy and paste what we have in the lecture slides. This should reflect the skeleton of your own implementation of the algorithm. |
| 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? | Original Heap data structure:  To delete from the middle of a heap, if the item to delete is the last item delete it and reduce the heap size by 1 otherwise move the last item into its position, reduce the heap size by one and then move the old last item into the position of its child with the highest priority while at least one of its children have a higher priority than it.  Original Hash Function:  Convert the string version of the state into a size\_t as if it’s a number in base 10. This creates a unique hash. | (e.g. original implementation of heap data structure, original hash function implementation) |