**National University of Computer and Emerging Sciences**



**Assignment # 1**

17F-9434 Usama Bin Haider | 17F-9428 M shaheer Rahi

**Language C++**

|  |
| --- |
| **Breadth First Search** |
| Breadth First Traversal (or Search) for a graph is similar to Breadth-First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles so that we may come to the same node again. To avoid processing a node more than once, we use a Boolean visited array. For simplicity, it assumed that all vertices are reachable from the starting vertex.  void applyBFS(){  while (Queue->isEmpty() == false)  {  recordStack \*tempStack = Queue->POP();  int topOnStack = tempStack->rsHead->V;  if (Expanded->isAlreadyPush(topOnStack) == false)  {  if (topOnStack == this->goalState)  {  this->resultentPath = tempStack;  break;  }  VERTIX \*vertixWithConectedNodes = graphObj->getVertixWithValue(topOnStack);  EDGE \*tempConectedEdgesHead = vertixWithConectedNodes->E\_HEAD;  while (tempConectedEdgesHead != NULL)  {  int currentEdgeValue = tempConectedEdgesHead->E\_Value;  if (Visted->isAlreadyPush(currentEdgeValue) == false)  {  Queue->PUSH(tempStack, currentEdgeValue, 0, tempStack->level + 1);  }  tempConectedEdgesHead = tempConectedEdgesHead->E\_Next;  }  }  }  } |

|  |
| --- |
| **Uniform Cost Search** |
| Uniform-Cost Search is a variant of Dijkstra's algorithm. Here, instead of inserting all vertices into a priority queue, we add the only source, then one by one insert when needed. In every step, we check if the item is already in the priority queue (using visited array). If yes, we perform a decrease key, and else we insert it.  void applyUCS(){  while (Stack->isEmpty() == false)  {  recordStack \*tempStack = Stack->getStackWithSmalestFn();  int topOnStack = tempStack->rsHead->V;  if (Expanded->isAlreadyPush(topOnStack) == false)  {  if (topOnStack == this->goalState)  {  this->resultentPath = tempStack;  break;  }  VERTIX \*vertixWithConectedNodes = graphObj->getVertixWithValue(topOnStack);  EDGE \*tempConectedEdgesHead = vertixWithConectedNodes->E\_HEAD;  while (tempConectedEdgesHead != NULL)  {  int currentEdgeValue = tempConectedEdgesHead->E\_Value;  if (Visted->isAlreadyPush(currentEdgeValue) == false)  {  Stack->PUSH(tempStack, currentEdgeValue, (tempStack->getFn() + tempConectedEdgesHead->GnCost), tempStack->level + 1);  }  tempConectedEdgesHead = tempConectedEdgesHead->E\_Next;  }  }  }  } |

|  |
| --- |
| **Depth First Search** |
| Depth First Traversal (or Search) for a graph is similar to Depth-First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles so that we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.  void applyDFS(){  while (Stack->isEmpty() == false)  {  recordStack \*tempStack = Stack->POP();  int topOnStack = tempStack->rsHead->V;  if (Expanded->isAlreadyPush(topOnStack) == false)  {  if (topOnStack == this->goalState)  {  this->resultentPath = tempStack;  break;  }  VERTIX \*vertixWithConectedNodes = graphObj->getVertixWithValue(topOnStack);  EDGE \*tempConectedEdgesHead = vertixWithConectedNodes->E\_HEAD;  while (tempConectedEdgesHead != NULL)  {  int currentEdgeValue = tempConectedEdgesHead->E\_Value;  if (Visted->isAlreadyPush(currentEdgeValue) == false)  {  Stack->PUSH(tempStack, currentEdgeValue, 0, tempStack->level + 1);  }  tempConectedEdgesHead = tempConectedEdgesHead->E\_Next;  }  }  }  } |

|  |
| --- |
| **Iterative Deepening Search** |
| Iterative deepening search or more specifically iterative deepening depth-first search] (IDS or IDDFS) is a state space/graph search strategy in which a depth-limited version of depth-first search is run repeatedly with increasing depth limits until the goal found. IDDFS is optimal like breadth-first search, but uses much less memory; at each iteration, it visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes first visited is effectively breadth-first.  void iteerativeApply(){  bool Y = false;  for (int i = 0; i <= this->depthLimt; i++)  {  Stack = new allRecordsStack();  Visted = new simpleStack();  Expanded = new simpleStack();  startState = getValue(start[0], start[1]);  goalState = getValue(goal[0], goal[1]);  Stack->PUSH(NULL, startState, 0, 0);  Visted->PUSH(startState);  applyIDS(i);  if (resultentPath != NULL)  {  Y = true;  break;  }  }  if (Y == false)  {  cout << "\nUNABLE to Reach the GOAL with DepthLimt of " << this->depthLimt << ".";  }  }  void applyIDS(int depthLimt){  while (Stack->isEmpty() == false)  {  recordStack \*tempStack = Stack->POP();  int topOnStack = tempStack->rsHead->V;  if (Expanded->isAlreadyPush(topOnStack) == false)  {  if (topOnStack == this->goalState)  {  this->resultentPath = tempStack;  break;  }  VERTIX \*vertixWithConectedNodes = graphObj->getVertixWithValue(topOnStack);  EDGE \*tempConectedEdgesHead = vertixWithConectedNodes->E\_HEAD;  while (tempConectedEdgesHead != NULL&& tempStack->level <depthLimt)  {  int currentEdgeValue = tempConectedEdgesHead->E\_Value;  if (Visted->isAlreadyPush(currentEdgeValue) == false)  {  Stack->PUSH(tempStack, currentEdgeValue, 0, tempStack->level + 1);  }  tempConectedEdgesHead = tempConectedEdgesHead->E\_Next;  }  }  }  } |

|  |
| --- |
| **Depth Limited Search** |
| The unbounded tree problem appeared in DFS can be fixed by imposing a limit on the depth that DFS can reach, this limit we will call depth limit l, this solves the infinite path problem. The limited path introduces another problem which is the case when we choose l < d, in which is our DLS will never reach a goal, in this case, we can say that DLS is not complete. One can view DFS as a particular case of the depth DLS, that DFS is DLS with l = infinity. DLS is not optimal even if l > d.  void applyDLS(int depthLimt){  while (Stack->isEmpty() == false)  {  recordStack \*tempStack = Stack->POP();  int topOnStack = tempStack->rsHead->V;  if (Expanded->isAlreadyPush(topOnStack) == false)  {  if (topOnStack == this->goalState)  {  this->resultentPath = tempStack;  break;  }  VERTIX \*vertixWithConectedNodes = graphObj->getVertixWithValue(topOnStack);  EDGE \*tempConectedEdgesHead = vertixWithConectedNodes->E\_HEAD;  while (tempConectedEdgesHead != NULL&& tempStack->level < depthLimt)  {  int currentEdgeValue = tempConectedEdgesHead->E\_Value;  if (Visted->isAlreadyPush(currentEdgeValue) == false)  {  Stack->PUSH(tempStack, currentEdgeValue, 0, tempStack->level + 1);  }  tempConectedEdgesHead = tempConectedEdgesHead->E\_Next;  }  }  }  } |

|  |
| --- |
| **Summary:** |
| **Data Structures Used For:**   1. Visited List (Stack) 2. Expanded List (Stack) 3. Node Path (A stack which contain stack objects)   also (A queue which contain stack objects)   1. Adjacency List (for Graph)   **Best Search for This Problem:**  The best search algorithm from above is Uniform Cost Search for that problem.  **Moves:**    **Instructions:**   * Grid.txt file and the Assignment1.exe must be in the same folder to run the exe file. * Double click, and it will run the code given below. |

|  |
| --- |
| **Complete Code:** |
| #include<iostream>  #include <iomanip>  #include<fstream>  using namespace std;  int \*\*grid;  int totalRows;  int totalCols;  //0->Rows ; 1->Cols  int start[2];  int goal[2];  void printGrid();  int getValue(int i, int j);  int getIndexOfCol(int val);  int getIndexOfRow(int val);  double huristicValue(int x1, int y1, int x2, int y2);  bool readGridFile();  class stackNode  {  public:  int V;  stackNode \*Next;  //------------------------------  stackNode(int Value){  this->V = Value;  this->Next = NULL;  };  };  class simpleStack  {  stackNode \*sHead;  public:  simpleStack(){  sHead = NULL;  }  simpleStack(int value){  sHead = NULL;  PUSH(value);  }  stackNode\* pop(){  stackNode\* tempHead = this->sHead;  sHead = sHead->Next;  return tempHead;  }  bool isAlreadyPush(int value){  bool Y = false;  stackNode \*tempsHead = sHead;  while (tempsHead != NULL)  {  if (tempsHead->V == value){  Y = true;  return Y;  }  tempsHead = tempsHead->Next;  }  this->PUSH(value);  return Y;  }  void PUSH(int value){  stackNode \* newNode = new stackNode(value);  newNode->Next = this->sHead;  this->sHead = newNode;  }  };  class recordStack  {  private:  //------------------------------  recordStack(){  rsHead = NULL;  fn = 0;  }  stackNode\* createNodeForRecordStack(int value){  stackNode \*sN = new stackNode(value);  return sN;  }  public:  stackNode \*rsHead;  double fn;  int level;  recordStack(recordStack \*preStack, int value, double fofn, int lvl){  setFn(fofn);  this->level = lvl;  rsHead = NULL;  pushValueToRecordStack(preStack, value);  }  void push(int value){  stackNode \*newNode = createNodeForRecordStack(value);  newNode->Next = this->rsHead;  this->rsHead = newNode;  }  void print(){  if (rsHead == NULL)  {  cout << "empty";  return;  }  else  {  stackNode \*tempHead = rsHead;  cout << "[F(n)= " << this->fn << "]" << " [Lvl= " << this->level << "]";  while (tempHead != NULL)  {  cout << " " << tempHead->V;  tempHead = tempHead->Next;  }  }  }  void setFn(double fofn){  this->fn = fofn;  }  double getFn(){  return this->fn;  }  bool isExist(int value){  stackNode \*tempHead = rsHead;  while (tempHead != NULL)  {  if (tempHead->V == value){  return true;  }  tempHead = tempHead->Next;  }  return false;  }  private:  void pushValueToRecordStack(recordStack \*prevStack, int value){  if (prevStack != NULL)  {  recordStack\* prev = prevStack;  recordStack\* reversStack = new recordStack();  stackNode \* tempPrevStackNode;  stackNode \*tempHead = prev->rsHead;  while (tempHead != NULL)  {  tempPrevStackNode = prev->pop();  reversStack->push(tempPrevStackNode->V);  tempHead = tempHead->Next;  }  tempHead = reversStack->rsHead;  while (tempHead != NULL)  {  tempPrevStackNode = reversStack->pop();  prev->push(tempPrevStackNode->V);  this->push(tempPrevStackNode->V);  tempHead = tempHead->Next;  }  }  push(value);  }  stackNode\* pop(){  stackNode\* tempHead = this->rsHead;  rsHead = rsHead->Next;  return tempHead;  }  };  class oneRecordStack  {  public:  recordStack \*stackObj;  oneRecordStack \*Next, \*Previous;  public:  oneRecordStack(recordStack \* prevStack, int value, double fofn, int lvl){  this->stackObj = new recordStack(prevStack, value, fofn, lvl);  Next = NULL;  Previous = NULL;  }  void stackObjprint(){  stackObj->print();  }  };  class allRecordsStack  {  oneRecordStack \*sHead;  public:  allRecordsStack(){  sHead = NULL;  }  allRecordsStack(recordStack \* prevStack, int value, double fofn, int lvl){  sHead = NULL;  PUSH(prevStack, value, fofn, lvl);  }  bool isEmpty(){  if (sHead == NULL)  {  return true;  }  else return false;  }  void print(){  //stackObjprint  if (sHead == NULL)  {  cout << "empty";  return;  }  oneRecordStack\* tempHead = sHead;  while (tempHead != NULL)  {  cout << "[";  tempHead->stackObjprint();  cout << "] ";  tempHead = tempHead->Next;  }  }  void PUSH(recordStack \*prevStack, int value, double fofn, int lvl){  oneRecordStack \*newNode = new oneRecordStack(prevStack, value, fofn, lvl);  newNode->Next = sHead;  sHead = newNode;  }  recordStack \*POP(){  recordStack \*tempStack = NULL;  if (sHead != NULL)  {  tempStack = sHead->stackObj;  oneRecordStack \*tempHead = sHead;  sHead = sHead->Next;  delete tempHead;  }  return tempStack;  }  recordStack \*getStackWithSmalestFn(){  double smaleVal;  oneRecordStack \*tempSNode = sHead;  oneRecordStack \*previousNode = NULL;  oneRecordStack \*actualPreviousNode = NULL;  recordStack \*smalestSNode = NULL;  if (sHead != NULL)  {  smaleVal = (tempSNode->stackObj->getFn() + 100);  }  while (tempSNode != NULL)  {  if (tempSNode->stackObj->getFn() < smaleVal){  smaleVal = tempSNode->stackObj->getFn();  smalestSNode = tempSNode->stackObj;  actualPreviousNode = previousNode;  }  previousNode = tempSNode;  tempSNode = tempSNode->Next;  }  if (smalestSNode != NULL)  {  if (actualPreviousNode == NULL)  {  sHead = sHead->Next;  }  else  {  actualPreviousNode->Next = actualPreviousNode->Next->Next;  }  }  return smalestSNode;  }  };  class allRecordsQueue  {  private:  oneRecordStack \*qHead, \*qTail;  public:  allRecordsQueue(){  qHead = NULL;  qTail = NULL;  }  allRecordsQueue(recordStack \* prevStack, int value, double fofn, int lvl){  qHead = NULL;  qTail = NULL;  PUSH(prevStack, value, fofn, lvl);  }  void print(){  //stackObjprint  if (qHead == NULL)  {  cout << "empty";  return;  }  oneRecordStack\* tempHead = qHead;  while (tempHead != NULL)  {  cout << "[";  tempHead->stackObjprint();  cout << "] ";  tempHead = tempHead->Next;  }  }  void PUSH(recordStack \*prevStack, int value, double fofn, int lvl){  oneRecordStack \*newNode = new oneRecordStack(prevStack, value, fofn, lvl);  if (qHead == NULL)  {  qHead = newNode;  qTail = newNode;  }  else  {  newNode->Previous = qTail;  qTail->Next = newNode;  qTail = newNode;  }  }  recordStack \*POP(){  recordStack \*tempStack = NULL;  if (qHead != NULL)  {  tempStack = qHead->stackObj;  oneRecordStack \*tempHead = qHead;  qHead = qHead->Next;  if (qHead == NULL)  {  qTail = NULL;  }  else  {  qHead->Previous = NULL;  }  delete tempHead;  }  return tempStack;  }  bool isEmpty(){  if (qHead == NULL)  {  return true;  }  return false;  }  };  struct EDGE  {  int E\_Value;  double GnCost;  EDGE \*E\_Next;  };  class VERTIX  {  EDGE \*CREAT\_EDGE(int CH, double D)  {  EDGE \*E = new EDGE;  E->E\_Value = CH;  E->GnCost = D;  E->E\_Next = NULL;  return E;  }  void ADD\_P\_EDGE(EDGE \*E, int CH, double D)  {  if (E->E\_Value == CH)  {  return;  }  if (E->E\_Next == NULL)  {  E->E\_Next = CREAT\_EDGE(CH, D);  return;  }  else  {  ADD\_P\_EDGE(E->E\_Next, CH, D);  }  }  public:  double Huristic;  int V\_value;  EDGE \*E\_HEAD;  VERTIX \*V\_NEXT;  VERTIX(){  V\_NEXT = NULL;  E\_HEAD = NULL;  }  void ADD\_EDGE(int CH, double D)  {  if (E\_HEAD == NULL)  {  E\_HEAD = CREAT\_EDGE(CH, D);  }  else  {  this->ADD\_P\_EDGE(E\_HEAD, CH, D);  }  }  void PRINT\_EDGES()  {  if (E\_HEAD == NULL)  {  cout << " non";  }  else  {  EDGE \*EP = E\_HEAD;  while (EP != NULL)  {  cout << " " << EP->E\_Value;  EP = EP->E\_Next;  }  }  }  };  class linkListGraph  {  VERTIX \*V\_HEAD;  private:  VERTIX \*CREAT\_VERTIX(int CH, double huristic)  {  VERTIX \*V = new VERTIX;  V->E\_HEAD = NULL;  V->V\_NEXT = NULL;  V->V\_value = CH;  V->Huristic = huristic;  return V;  }  void ADD\_P\_VERTIX(VERTIX \*&V, int CH, double huristic)  {  if (V->V\_value == CH)  {  return;  }  if (V->V\_NEXT == NULL)  {  V->V\_NEXT = CREAT\_VERTIX(CH, huristic);  return;  }  else  {  ADD\_P\_VERTIX(V->V\_NEXT, CH, huristic);  }  }  void ADD\_VERTIX(int CH, double huristic)  {  if (V\_HEAD == NULL)  {  V\_HEAD = CREAT\_VERTIX(CH, huristic);  }  else  {  ADD\_P\_VERTIX(V\_HEAD, CH, huristic);  }  }  void CREAT\_linkListGraph(){  //----------------------------------------------  VERTIX \*V = V\_HEAD;  int A, B;  double D;  cout << "\n Moves Allowd:";  cout << "\n\t-Right one cell (cost is 3)";  //-----------------------------------------------------------------------------------  D = 3;  for (int i = 0; i < totalRows; i++)  {  for (int j = 0; j < totalCols - 1; j++)  {  if (grid[i][j] == 0 && grid[i][j + 1] == 0)  {  A = getValue(i, j);  B = getValue(i, j + 1);  double huristic = huristicValue(goal[0], goal[1], i, j);  ADD\_VERTIX(A, huristic);  huristic = huristicValue(goal[0], goal[1], i, j + 1);  ADD\_VERTIX(B, huristic);  V = V\_HEAD;  while (V != NULL)  {  if (V->V\_value == A)  {  V->ADD\_EDGE(B, D);  break;  }  V = V->V\_NEXT;  }  }  }  }  cout << "\n\t-Diagonally up towards the right (cost is 2)";  //--------------------------------------------------------------------------------------------  D = 2;  for (int i = 0; i < totalRows - 1; i++)  {  for (int j = 0; j < totalCols - 1; j++)  {  if (grid[i][j] == 0 && grid[i + 1][j + 1] == 0)  {  A = getValue(i, j);  B = getValue(i + 1, j + 1);  double huristic = huristicValue(goal[0], goal[1], i, j);  ADD\_VERTIX(A, huristic);  huristic = huristicValue(goal[0], goal[1], i, j + 1);  ADD\_VERTIX(B, huristic);  V = V\_HEAD;  while (V != NULL)  {  if (V->V\_value == A)  {  V->ADD\_EDGE(B, D);  break;  }  V = V->V\_NEXT;  }  }  }  }  cout << "\n\t-Up one cell (cost is 1)";  //-----------------------------------------------------------------------------------  D = 1;  for (int i = 0; i < totalRows - 1; i++)  {  for (int j = 0; j < totalCols; j++)  {  if (grid[i][j] == 0 && grid[i + 1][j] == 0)  {  A = getValue(i, j);  B = getValue(i + 1, j);  double huristic = huristicValue(goal[0], goal[1], i, j);  ADD\_VERTIX(A, huristic);  huristic = huristicValue(goal[0], goal[1], i, j + 1);  ADD\_VERTIX(B, huristic);  V = V\_HEAD;  while (V != NULL)  {  if (V->V\_value == A)  {  V->ADD\_EDGE(B, D);  break;  }  V = V->V\_NEXT;  }  }  }  }  cout << "\n\n";  }  public:  linkListGraph(){  V\_HEAD = NULL;  CREAT\_linkListGraph();  }  VERTIX \* getVertixWithValue(int value){  VERTIX \*tempV = V\_HEAD;  while (tempV != NULL)  {  if (value == tempV->V\_value)  {  return tempV;  }  tempV = tempV->V\_NEXT;  }  return tempV;  }  void PRINT\_VERTIX()  {  if (V\_HEAD == NULL)  {  cout << "\nNO VERTIX EXIST\n";  }  else  {  VERTIX \*VO = V\_HEAD;  cout << "\n\n\nVERTIX:-\t";  while (VO != NULL)  {  cout << " " << VO->V\_value;  VO = VO->V\_NEXT;  }  }  }  void PRINT\_GRAPH()  {  cout << "\n\n\t:-:GRAPH:-:\n";  if (V\_HEAD == NULL)  {  cout << "\nNO VERTIX EXIST\n";  }  else  {  cout << "\n VERTIX EDGE";  VERTIX \*VO = V\_HEAD;  while (VO != NULL)  {  cout << "\n " << VO->V\_value << " :-> ";  VO->PRINT\_EDGES();  VO = VO->V\_NEXT;  }  }  }  };  linkListGraph \*graphObj = NULL;  //Breadth First Search  class BFS  {  allRecordsQueue \*Queue = NULL;  simpleStack \*Visted = NULL, \*Expanded = NULL;  int startState, goalState;  recordStack \*resultentPath = NULL;  public:  BFS(){  Queue = new allRecordsQueue();  Visted = new simpleStack();  Expanded = new simpleStack();  startState = getValue(start[0], start[1]);  goalState = getValue(goal[0], goal[1]);  Queue->PUSH(NULL, startState, 0, 0);  Visted->PUSH(startState);  applyBFS();  }  void print(){  if (showPath() == true){  printBFSGraphWithPath();  }  else  {  printBFSGraphWithoutPath();  }  }  bool showPath(){  cout << "\n(BFS) Goal to Start Path: ";  if (resultentPath == NULL)  {  cout << "Does Not exist";  return false;  }  else  {  resultentPath->print();  return true;  }  }  private:  void applyBFS(){  while (Queue->isEmpty() == false)  {  recordStack \*tempStack = Queue->POP();  int topOnStack = tempStack->rsHead->V;  if (Expanded->isAlreadyPush(topOnStack) == false)  {  if (topOnStack == this->goalState)  {  this->resultentPath = tempStack;  break;  }  VERTIX \*vertixWithConectedNodes = graphObj->getVertixWithValue(topOnStack);  EDGE \*tempConectedEdgesHead = vertixWithConectedNodes->E\_HEAD;  while (tempConectedEdgesHead != NULL)  {  int currentEdgeValue = tempConectedEdgesHead->E\_Value;  if (Visted->isAlreadyPush(currentEdgeValue) == false)  {  Queue->PUSH(tempStack, currentEdgeValue, 0, tempStack->level + 1);  }  tempConectedEdgesHead = tempConectedEdgesHead->E\_Next;  }  }  }  }  void printBFSGraphWithPath(){  cout << "\n\n\n";  for (int i = totalRows - 1; i >= 0; i--)  //for (int i = 0; i < totalRows; i++)  {  cout << endl;  for (int j = 0; j < totalCols; j++)  {  if (grid[i][j] == 0)  {  int value = getValue(i, j);  if (resultentPath->isExist(value) == true)  {  if ((start[0] == i&& start[1] == j) && (goal[0] == i&& goal[1] == j)){  cout << "| " << "P" << "\t";  }  else if (start[0] == i&& start[1] == j)  {  cout << "| " << "S" << "\t";  }  else if (goal[0] == i&& goal[1] == j)  {  cout << "| " << "G" << "\t";  }  else  {  cout << "| " << "\*" << "\t";  }  }  else  {  cout << "| \t";  }  }  else  {  cout << "| " << -1 << "\t";  }  }  cout << "|" << endl;  }  cout << endl;  }  void printBFSGraphWithoutPath(){  cout << "\n\n\n";  for (int i = totalRows - 1; i >= 0; i--)  //for (int i = 0; i < totalRows; i++)  {  cout << endl;  for (int j = 0; j < totalCols; j++)  {  if (grid[i][j] == 0)  {  cout << "| \t";  }  else  {  cout << "| " << -1 << "\t";  }  }  cout << "|" << endl;  }  cout << endl;  }  };  //Depth First Search  class DFS  {  allRecordsStack \*Stack = NULL;  simpleStack \*Visted = NULL, \*Expanded = NULL;  int startState, goalState;  recordStack \*resultentPath = NULL;  public:  DFS(){  Stack = new allRecordsStack();  Visted = new simpleStack();  Expanded = new simpleStack();  startState = getValue(start[0], start[1]);  goalState = getValue(goal[0], goal[1]);  Stack->PUSH(NULL, startState, 0, 0);  Visted->PUSH(startState);  applyDFS();  }  void print(){  if (showPath() == true){  printDFSGraphWithPath();  }  else  {  printDFSGraphWithoutPath();  }  }  bool showPath(){  cout << "\n(DFS) Goal to Start Path: ";  if (resultentPath == NULL)  {  cout << "Does Not exist";  return false;  }  else  {  resultentPath->print();  return true;  }  }  private:  void applyDFS(){  while (Stack->isEmpty() == false)  {  recordStack \*tempStack = Stack->POP();  int topOnStack = tempStack->rsHead->V;  if (Expanded->isAlreadyPush(topOnStack) == false)  {  if (topOnStack == this->goalState)  {  this->resultentPath = tempStack;  break;  }  VERTIX \*vertixWithConectedNodes = graphObj->getVertixWithValue(topOnStack);  EDGE \*tempConectedEdgesHead = vertixWithConectedNodes->E\_HEAD;  while (tempConectedEdgesHead != NULL)  {  int currentEdgeValue = tempConectedEdgesHead->E\_Value;  if (Visted->isAlreadyPush(currentEdgeValue) == false)  {  Stack->PUSH(tempStack, currentEdgeValue, 0, tempStack->level + 1);  }  tempConectedEdgesHead = tempConectedEdgesHead->E\_Next;  }  }  }  }  void printDFSGraphWithPath(){  cout << "\n\n\n";  for (int i = totalRows - 1; i >= 0; i--)  //for (int i = 0; i < totalRows; i++)  {  cout << endl;  for (int j = 0; j < totalCols; j++)  {  if (grid[i][j] == 0)  {  int value = getValue(i, j);  if (resultentPath->isExist(value) == true)  {  if ((start[0] == i&& start[1] == j) && (goal[0] == i&& goal[1] == j)){  cout << "| " << "P" << "\t";  }  else if (start[0] == i&& start[1] == j)  {  cout << "| " << "S" << "\t";  }  else if (goal[0] == i&& goal[1] == j)  {  cout << "| " << "G" << "\t";  }  else  {  cout << "| " << "\*" << "\t";  }  }  else  {  cout << "| \t";  }  }  else  {  cout << "| " << -1 << "\t";  }  }  cout << "|" << endl;  }  cout << endl;  }  void printDFSGraphWithoutPath(){  cout << "\n\n\n";  for (int i = totalRows - 1; i >= 0; i--)  //for (int i = 0; i < totalRows; i++)  {  cout << endl;  for (int j = 0; j < totalCols; j++)  {  if (grid[i][j] == 0)  {  cout << "| \t";  }  else  {  cout << "| " << -1 << "\t";  }  }  cout << "|" << endl;  }  cout << endl;  }  };  //Uniform Cost Search  class UCS  {  allRecordsStack \*Stack = NULL;  simpleStack \*Visted = NULL, \*Expanded = NULL;  int startState, goalState;  recordStack \*resultentPath = NULL;  public:  UCS(){  Stack = new allRecordsStack();  Visted = new simpleStack();  Expanded = new simpleStack();  startState = getValue(start[0], start[1]);  goalState = getValue(goal[0], goal[1]);  Stack->PUSH(NULL, startState, 0, 0);  Visted->PUSH(startState);  applyUCS();  }  void print(){  if (showPath() == true){  printUCSGraphWithPath();  }  else  {  printUCSGraphWithoutPath();  }  }  bool showPath(){  cout << "\n(UCS) Goal to Start Path: ";  if (resultentPath == NULL)  {  cout << "Does Not exist";  return false;  }  else  {  resultentPath->print();  return true;  }  }  private:  void applyUCS(){  while (Stack->isEmpty() == false)  {  recordStack \*tempStack = Stack->getStackWithSmalestFn();  int topOnStack = tempStack->rsHead->V;  //cout << "\n";  //tempStack->print();  if (Expanded->isAlreadyPush(topOnStack) == false)  {  if (topOnStack == this->goalState)  {  this->resultentPath = tempStack;  break;  }  VERTIX \*vertixWithConectedNodes = graphObj->getVertixWithValue(topOnStack);  EDGE \*tempConectedEdgesHead = vertixWithConectedNodes->E\_HEAD;  while (tempConectedEdgesHead != NULL)  {  int currentEdgeValue = tempConectedEdgesHead->E\_Value;  if (Visted->isAlreadyPush(currentEdgeValue) == false)  {  Stack->PUSH(tempStack, currentEdgeValue, (tempStack->getFn() + tempConectedEdgesHead->GnCost), tempStack->level + 1);  }  tempConectedEdgesHead = tempConectedEdgesHead->E\_Next;  }  }  }  cout << "\n"; cout << "\n"; cout << "\n";  }  void printUCSGraphWithPath(){  cout << "\n\n\n";  for (int i = totalRows - 1; i >= 0; i--)  //for (int i = 0; i < totalRows; i++)  {  cout << endl;  for (int j = 0; j < totalCols; j++)  {  if (grid[i][j] == 0)  {  int value = getValue(i, j);  if (resultentPath->isExist(value) == true)  {  if ((start[0] == i&& start[1] == j) && (goal[0] == i&& goal[1] == j)){  cout << "| " << "P" << "\t";  }  else if (start[0] == i&& start[1] == j)  {  cout << "| " << "S" << "\t";  }  else if (goal[0] == i&& goal[1] == j)  {  cout << "| " << "G" << "\t";  }  else  {  cout << "| " << "\*" << "\t";  }  }  else  {  cout << "| \t";  }  }  else  {  cout << "| " << -1 << "\t";  }  }  cout << "|" << endl;  }  cout << endl;  }  void printUCSGraphWithoutPath(){  cout << "\n\n\n";  for (int i = totalRows - 1; i >= 0; i--)  //for (int i = 0; i < totalRows; i++)  {  cout << endl;  for (int j = 0; j < totalCols; j++)  {  if (grid[i][j] == 0)  {  cout << "| \t";  }  else  {  cout << "| " << -1 << "\t";  }  }  cout << "|" << endl;  }  cout << endl;  }  };  //Depth Limited Search  class DLS  {  allRecordsStack \*Stack = NULL;  simpleStack \*Visted = NULL, \*Expanded = NULL;  int startState, goalState;  recordStack \*resultentPath = NULL;  int depthLimt;  public:  DLS(int depthLimt){  this->depthLimt = depthLimt;  Stack = new allRecordsStack();  Visted = new simpleStack();  Expanded = new simpleStack();  startState = getValue(start[0], start[1]);  goalState = getValue(goal[0], goal[1]);  Stack->PUSH(NULL, startState, 0, 0);  Visted->PUSH(startState);  applyDLS(this->depthLimt);  }  void print(){  if (showPath() == true){  printDLSGraphWithPath();  }  else  {  printDLSGraphWithoutPath();  }  }  bool showPath(){  cout << "\n(DLS) Depthlimit: " << depthLimt << "\n Goal to Start Path: ";  if (resultentPath == NULL)  {  cout << "Does Not exist";  return false;  }  else  {  resultentPath->print();  return true;  }  }  private:  void applyDLS(int depthLimt){  while (Stack->isEmpty() == false)  {  recordStack \*tempStack = Stack->POP();  int topOnStack = tempStack->rsHead->V;  if (Expanded->isAlreadyPush(topOnStack) == false)  {  if (topOnStack == this->goalState)  {  this->resultentPath = tempStack;  break;  }  VERTIX \*vertixWithConectedNodes = graphObj->getVertixWithValue(topOnStack);  EDGE \*tempConectedEdgesHead = vertixWithConectedNodes->E\_HEAD;  while (tempConectedEdgesHead != NULL&& tempStack->level <depthLimt)  {  int currentEdgeValue = tempConectedEdgesHead->E\_Value;  if (Visted->isAlreadyPush(currentEdgeValue) == false)  {  Stack->PUSH(tempStack, currentEdgeValue, 0, tempStack->level + 1);  }  tempConectedEdgesHead = tempConectedEdgesHead->E\_Next;  }  }  }  }  void printDLSGraphWithPath(){  cout << "\n\n\n";  for (int i = totalRows - 1; i >= 0; i--)  //for (int i = 0; i < totalRows; i++)  {  cout << endl;  for (int j = 0; j < totalCols; j++)  {  if (grid[i][j] == 0)  {  int value = getValue(i, j);  if (resultentPath->isExist(value) == true)  {  if ((start[0] == i&& start[1] == j) && (goal[0] == i&& goal[1] == j)){  cout << "| " << "P" << "\t";  }  else if (start[0] == i&& start[1] == j)  {  cout << "| " << "S" << "\t";  }  else if (goal[0] == i&& goal[1] == j)  {  cout << "| " << "G" << "\t";  }  else  {  cout << "| " << "\*" << "\t";  }  }  else  {  cout << "| \t";  }  }  else  {  cout << "| " << -1 << "\t";  }  }  cout << "|" << endl;  }  cout << endl;  }  void printDLSGraphWithoutPath(){  cout << "\n\n\n";  for (int i = totalRows - 1; i >= 0; i--)  //for (int i = 0; i < totalRows; i++)  {  cout << endl;  for (int j = 0; j < totalCols; j++)  {  if (grid[i][j] == 0)  {  cout << "| \t";  }  else  {  cout << "| " << -1 << "\t";  }  }  cout << "|" << endl;  }  cout << endl;  }  };  //Iterative Deepening Search  class IDS  {  allRecordsStack \*Stack = NULL;  simpleStack \*Visted = NULL, \*Expanded = NULL;  int startState, goalState;  recordStack \*resultentPath = NULL;  int depthLimt;  public:  IDS(int depthLimt){  this->depthLimt = depthLimt;  iteerativeApply();  }  void iteerativeApply(){  bool Y = false;  for (int i = 0; i <= this->depthLimt; i++)  {  Stack = new allRecordsStack();  Visted = new simpleStack();  Expanded = new simpleStack();  startState = getValue(start[0], start[1]);  goalState = getValue(goal[0], goal[1]);  Stack->PUSH(NULL, startState, 0, 0);  Visted->PUSH(startState);  applyIDS(i);  if (resultentPath != NULL)  {  Y = true;  break;  }  }  if (Y == false)  {  cout << "\nUNABLE to Reach the GOAL with DepthLimt of " << this->depthLimt << ".";  }  }  void print(){  if (showPath(this->depthLimt) == true){  printIDSGraphWithPath();  }  else  {  printIDSGraphWithoutPath();  }  }  private:  bool showPath(int depthLimt){  cout << "\n(IDS) Depthlimit: " << depthLimt << "\n Goal to Start Path: ";  if (resultentPath == NULL)  {  cout << "Does Not exist";  return false;  }  else  {  resultentPath->print();  return true;  }  }  void applyIDS(int depthLimt){  while (Stack->isEmpty() == false)  {  recordStack \*tempStack = Stack->POP();  int topOnStack = tempStack->rsHead->V;  if (Expanded->isAlreadyPush(topOnStack) == false)  {  if (topOnStack == this->goalState)  {  this->resultentPath = tempStack;  break;  }  VERTIX \*vertixWithConectedNodes = graphObj->getVertixWithValue(topOnStack);  EDGE \*tempConectedEdgesHead = vertixWithConectedNodes->E\_HEAD;  while (tempConectedEdgesHead != NULL&& tempStack->level <depthLimt)  {  int currentEdgeValue = tempConectedEdgesHead->E\_Value;  if (Visted->isAlreadyPush(currentEdgeValue) == false)  {  Stack->PUSH(tempStack, currentEdgeValue, 0, tempStack->level + 1);  }  tempConectedEdgesHead = tempConectedEdgesHead->E\_Next;  }  }  }  }  void printIDSGraphWithPath(){  cout << "\n\n\n";  for (int i = totalRows - 1; i >= 0; i--)  //for (int i = 0; i < totalRows; i++)  {  cout << endl;  for (int j = 0; j < totalCols; j++)  {  if (grid[i][j] == 0)  {  int value = getValue(i, j);  if (resultentPath->isExist(value) == true)  {  if ((start[0] == i&& start[1] == j) && (goal[0] == i&& goal[1] == j)){  cout << "| " << "P" << "\t";  }  else if (start[0] == i&& start[1] == j)  {  cout << "| " << "S" << "\t";  }  else if (goal[0] == i&& goal[1] == j)  {  cout << "| " << "G" << "\t";  }  else  {  cout << "| " << "\*" << "\t";  }  }  else  {  cout << "| \t";  }  }  else  {  cout << "| " << -1 << "\t";  }  }  cout << "|" << endl;  }  cout << endl;  }  void printIDSGraphWithoutPath(){  cout << "\n\n\n";  for (int i = totalRows - 1; i >= 0; i--)  //for (int i = 0; i < totalRows; i++)  {  cout << endl;  for (int j = 0; j < totalCols; j++)  {  if (grid[i][j] == 0)  {  cout << "| \t";  }  else  {  cout << "| " << -1 << "\t";  }  }  cout << "|" << endl;  }  cout << endl;  }  };  //--------------------------  void main(){  if (readGridFile()){  graphObj = new linkListGraph();  BFS bfs;  bfs.print();  cout << "\n\n";  DFS dfs;  dfs.print();  cout << "\n\n";  UCS ucs;  ucs.print();  cout << "\n\n";  DLS dls(16);  dls.print();  cout << "\n\n";  IDS ids(16);  ids.print();  cout << "\n\n";  }  cout << "\n\n\n";  system("pause");  }  //--------------------------  bool readGridFile(){  bool fileExist = false;  ifstream read;  read.open("grid.txt");  if (read)  {  read >> totalCols >> totalRows;  cout << "\nRows: " << totalRows << "\t Col:" << totalCols << endl;  read >> start[1] >> start[0];  read >> goal[1] >> goal[0];  cout << "\nStart: " << start[0] << " " << start[1] << endl;  cout << "\nGoal: " << goal[0] << " " << goal[1] << endl;  grid = new int\*[totalRows];  for (int i = 0; i < totalRows; i++)  {  grid[i] = new int[totalCols];  }  int val;  for (int i = totalRows - 1; i >= 0; i--)  {  for (int j = 0; j < totalCols; j++)  {  read >> grid[i][j];  }  }  //printGrid();  cout << "\n\n";  fileExist = true;  read.close();  }  else  {  cout << "\nFile Does Not Exist";  }  return fileExist;  }  int getValue(int i, int j)  {  return ((i\*totalRows) + j);  }  int getIndexOfCol(int val)  {  return(val % totalRows);  }  int getIndexOfRow(int val)  {  return(val / totalRows);  }  void printGrid(){  for (int i = 0; i < totalRows; i++)  {  cout << endl;  for (int j = 0; j < totalCols; j++)  {  if (grid[i][j] == 0)  {  cout << "| " << getValue(i, j) << "\t";  }  else  {  cout << "| " << -1 << "\t";  }  }  cout << "|" << endl;  }  cout << endl;  }  double huristicValue(int x1, int y1, int x2, int y2){  double huristic = sqrt(((x1 - x2)\*(x1 - x2)) + ((y1 - y2)\*(y1 - y2)));  return huristic;  } |