**National University of Computer and Emerging Sciences**



**Assignment # 2**

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

**Language C++**

|  |
| --- |
| **Greedy Best First Search** |
| Greedy best-first search tries to expand the node that is closest to the goal because it is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, f(n)=h(n).  The worst-case complexity for greedy search is O(bm).  **Main Code:**  void applyGBFS(){  while (Stack->isEmpty() == false)  {  recordStack \*tempStack = Stack->getStackWithSmallestFn();  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, graphObj->getHuristicOf(currentEdgeValue), tempStack->level + 1);  }  tempConectedEdgesHead = tempConectedEdgesHead->E\_Next;  }  }  }  } |

|  |
| --- |
| **A\*** |
| A\* is an informed search algorithm, or a best-first search, meaning that formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance traveled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.  f(n)=g(n)+h(n)  The time complexity of A\* depends on the heuristic. In the worst case of an infinite search space, the number of nodes expanded is exponential in the depth of the solution (the shortest path) d: O(bd), where b is the branching factor (the average number of successors per state).  **Main Code:**  void applyAstar(){  while (Stack->isEmpty() == false)  {  recordStack \*tempStack = Stack->getStackWithSmallestFn();  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)  {  //----------------(previous[f(n)] - previous[h(n)]) + new[g(n)]+ new[h(n)]  double tempFn = ((tempStack->getFn() - graphObj->getHuristicOf(tempStack->rsHead->V)) + tempConectedEdgesHead->GnCost + graphObj->getHuristicOf(currentEdgeValue));  Stack->PUSH(tempStack, currentEdgeValue, tempFn, tempStack->level + 1);  }  tempConectedEdgesHead = tempConectedEdgesHead->E\_Next;  }  }  }  } |

|  |
| --- |
| **IDA\*** |
| Iterative-deepening-A\* works as follows: at each iteration, perform a depth-first search, cutting off a branch when its total cost f(n)=g(n)+h(n) exceeds a given threshold. This threshold starts at the estimate of the cost at the initial state and increases for each iteration of the algorithm. At each iteration, the threshold used for the next iteration is the minimum cost of all values that exceeded the current threshold.  As in A\*, the heuristic has to have particular properties to guarantee optimality (shortest paths).  **Main Code:**  void itterativelyApply(double limtFn){  bool Y = false;  for (double i = 0; i <= limtFn; 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, graphObj->getHuristicOf(startState), 0);  Visted->PUSH(startState);  applyIDAstar(i);  if (resultentPath != NULL)  {  Y = true;  break;  }  }  if (Y == false)  {  cout << "\nUNABLE to Reach the GOAL with F(n)Limt of " << this->maxlimtFn << ".";  }  }  void applyIDAstar(double limtFn){  while (Stack->isEmpty() == false)  {  recordStack \*tempStack = Stack->getStackWithSmallestFn();  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)  {  //----------------(previous[f(n)] - previous[h(n)]) + new[g(n)]+ new[h(n)]  double tempFn = ((tempStack->getFn() - graphObj->getHuristicOf(tempStack->rsHead->V)) + tempConectedEdgesHead->GnCost + graphObj->getHuristicOf(currentEdgeValue));  if (tempFn <= limtFn)  {  Stack->PUSH(tempStack, currentEdgeValue, tempFn, tempStack->level + 1);  }  else  {  //we added befor chacking f(n) so Pop it from Visted Stack  Visted->pop();  }  }  tempConectedEdgesHead = tempConectedEdgesHead->E\_Next;  }  }  }  } |

|  |
| --- |
| **RBFS** |
| The memory limitation of the heuristic path algorithm can be overcome by merely replacing the best-first search with IDA\* search using the specific weighted evaluation function. IDA\* search is no longer a best-first search since the total cost of a child can be less than that of its parent, and thus nodes are not necessarily expanded in best-first order. Recursive Best-First Search (RBFS) is an alternative algorithm. Recursive best-first search is a best-first search that runs in space that is linear concerning the maximum search depth, regardless of the cost function used. Even with an admissible cost function, Recursive Best-First Search generates fewer nodes than IDA\*, and is generally superior to IDA\*, except for a small increase in the cost per node generation.  An alternative path from any ancestor node one-step look-ahead If the current node exceeds this value, recursion unwinds back to the alternative path same idea as contour as recursion unwinds, replaces f(n)-value of a node with best f(n)-value of children. Allows remembering whether to re-expand path at a later time.  void applyRBFS(){  while (QueueStacks->isEmpty() == false)  {  this->smallest = QueueStacks->getStackWithSmallestFn();  this->secondSmallest = QueueStacks->getStackWithSmallestFn();  if (smallest->rsHead->V == this->goalState)  {  resultentPath = smallest;  return;  }  if (secondSmallest == NULL)  {  addAllChildOfParentToQueueStack(smallest, graphObj->getVertixWithValue(smallest->rsHead->V));  this->Expanded->PUSH(smallest->rsHead->V);  }  else  {  double checkNegValue = isAnyChildValuesLessThenSecSmallestValue(graphObj->getVertixWithValue(smallest->rsHead->V), (smallest->fn - graphObj->getHuristicOf(smallest->rsHead->V)), secondSmallest->fn);  if (checkNegValue != NULL)  {  addAllChildOfParentToQueueStack(smallest, graphObj->getVertixWithValue(smallest->rsHead->V));  this->Expanded->PUSH(smallest->rsHead->V);  this->addRecodStackAgain(secondSmallest);  }  else  {  double childeSmallest = anySmallestValueInParent(graphObj->getVertixWithValue(smallest->rsHead->V), (smallest->fn - graphObj->getHuristicOf(smallest->rsHead->V)));  double diffrence = childeSmallest - smallest->fn;  this->smallest->fn += diffrence;  graphObj->setHuristicWhereValue(smallest->rsHead->V, (graphObj->getHuristicOf(smallest->rsHead->V) + diffrence));  this->addRecodStackAgain(smallest);  this->addRecodStackAgain(secondSmallest);  }  }  }  } |

|  |
| --- |
| **Summary:** |
| **Data Structures Used For:**   1. Visited List (Stack) 2. Expanded List (Stack) 3. Node Path (A stack which contain stack objects) 4. Adjacency List (for Graph)   **Best Search for This Problem:**  The best search algorithm from above is RBFS for that problem.  **Moves:**    **Instructions:**   * Grid.txt file and the Assignment2.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>  #include <limits>  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;  }  int POP(){    int tempValue = -1;  if (rsHead != NULL)  {  stackNode \*tempHead = rsHead;  tempValue = rsHead->V;  rsHead = rsHead->Next;  delete tempHead;  }  return tempValue;  }  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 \*getStackWithSmallestFn(){  double smaleVal;  oneRecordStack \*tempSNode = sHead;  oneRecordStack \*previousNode = NULL;  oneRecordStack \*actualPreviousNode = NULL;  recordStack \*smallestSNode = NULL;  if (sHead != NULL)  {  smaleVal = (tempSNode->stackObj->getFn() + 100);  }  while (tempSNode != NULL)  {  if (tempSNode->stackObj->getFn() < smaleVal){  smaleVal = tempSNode->stackObj->getFn();  smallestSNode = tempSNode->stackObj;  actualPreviousNode = previousNode;  }  previousNode = tempSNode;  tempSNode = tempSNode->Next;  }  if (smallestSNode != NULL)  {  if (actualPreviousNode == NULL)  {  sHead = sHead->Next;  }  else  {  actualPreviousNode->Next = actualPreviousNode->Next->Next;  }  }  return smallestSNode;  }  recordStack \*getStackWithTopValue(int value){    oneRecordStack \*tempSNode = sHead;  oneRecordStack \*previousNode = NULL;  oneRecordStack \*actualPreviousNode = NULL;  recordStack \*sNodeWithFn = NULL;    while (tempSNode != NULL)  {  if (tempSNode->stackObj->rsHead->V == value){  sNodeWithFn = tempSNode->stackObj;  actualPreviousNode = previousNode;  break;  }  previousNode = tempSNode;  tempSNode = tempSNode->Next;  }  if (sNodeWithFn != NULL)  {  if (actualPreviousNode == NULL)  {  sHead = sHead->Next;  }  else  {  actualPreviousNode->Next = actualPreviousNode->Next->Next;  }  }  return sNodeWithFn;  }  };  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;  bool visted = false;  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 << "[g(n)= " << EP ->GnCost<< "]";  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-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 + 1, j);  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(start[0], start[1], i, j);  ADD\_VERTIX(A, huristic);  huristic = huristicValue(start[0], start[1], i+1, 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-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(start[0],start[1],i , j );  ADD\_VERTIX(A,huristic);  huristic = huristicValue(start[0], start[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;  }  double getHuristicOf(int value){  VERTIX \*tempV = V\_HEAD;  while (tempV != NULL)  {  if (value == tempV->V\_value)  {  return tempV->Huristic;  }  tempV = tempV->V\_NEXT;  }  return -10;  }  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 << "\n" << VO->V\_value << "\th(n)=" << VO->Huristic;  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;  }  }  }  void setHuristicWhereValue(int value,double hn){    VERTIX \*tempV = V\_HEAD;  while (tempV != NULL)  {  if (tempV->V\_value == value)  {  tempV->Huristic = hn;  return;  }  tempV = tempV->V\_NEXT;  }  }  };  linkListGraph \*graphObj = NULL;  //Greedy Best First Search  class GBFS  {  allRecordsStack \*Stack = NULL;  simpleStack \*Visted = NULL, \*Expanded = NULL;  int startState, goalState;  recordStack \*resultentPath = NULL;  public:  GBFS(){  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, graphObj->getHuristicOf(startState), 0);  Visted->PUSH(startState);  applyGBFS();  }  void print(){  if (showPath() == true){  printGBFSGraphWithPath();  }  else  {  printGBFSGraphWithoutPath();  }  }  bool showPath(){  cout << "\n(GBFS) Goal to Start Path: ";  if (resultentPath == NULL)  {  cout << "Does Not exist";  return false;  }  else  {  resultentPath->print();  return true;  }  }  private:  void applyGBFS(){  while (Stack->isEmpty() == false)  {  recordStack \*tempStack = Stack->getStackWithSmallestFn();  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, graphObj->getHuristicOf(currentEdgeValue), tempStack->level + 1);  }  tempConectedEdgesHead = tempConectedEdgesHead->E\_Next;  }  }  }  }  void printGBFSGraphWithPath(){  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 printGBFSGraphWithoutPath(){  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;  }  };  //A\*  class Astar  {  allRecordsStack \*Stack = NULL;  simpleStack \*Visted = NULL, \*Expanded = NULL;  int startState, goalState;  recordStack \*resultentPath = NULL;  public:  Astar(){  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, graphObj->getHuristicOf(startState), 0);  Visted->PUSH(startState);  applyAstar();  }  void print(){  if (showPath() == true){  printAstarGraphWithPath();  }  else  {  printAstarGraphWithoutPath();  }  }  bool showPath(){  cout << "\n(Astar) Goal to Start Path: ";  if (resultentPath == NULL)  {  cout << "Does Not exist";  return false;  }  else  {  resultentPath->print();  return true;  }  }  private:  void applyAstar(){  while (Stack->isEmpty() == false)  {  recordStack \*tempStack = Stack->getStackWithSmallestFn();  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)  {  //----------------(previous[f(n)] - previous[h(n)]) + new[g(n)]+ new[h(n)]  double tempFn = ((tempStack->getFn() - graphObj->getHuristicOf(tempStack->rsHead->V)) + tempConectedEdgesHead->GnCost + graphObj->getHuristicOf(currentEdgeValue));  Stack->PUSH(tempStack, currentEdgeValue,tempFn, tempStack->level + 1);  }  tempConectedEdgesHead = tempConectedEdgesHead->E\_Next;  }  }  }  }  void printAstarGraphWithPath(){  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 printAstarGraphWithoutPath(){  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;  }  };  //Itterative Deepning Search  class IDAstar  {  allRecordsStack \*Stack = NULL;  simpleStack \*Visted = NULL, \*Expanded = NULL;  int startState, goalState;  recordStack \*resultentPath = NULL;  double maxlimtFn = 0;  public:  IDAstar(double maxlimtFn){  this->maxlimtFn = maxlimtFn;  itterativelyApply(this->maxlimtFn);  }  void print(){  if (showPath() == true){  printIDAstarGraphWithPath();  }  else  {  printIDAstarGraphWithoutPath();  }  }  bool showPath(){  cout << "\n(IDAstar) Iterative F(n)Limit: " << this->maxlimtFn << "\n Goal to Start Path: ";  if (resultentPath == NULL)  {  cout << "Does Not exist";  return false;  }  else  {  resultentPath->print();  return true;  }  }  private:  void itterativelyApply(double limtFn){  bool Y = false;  for (double i = 0; i <= limtFn; 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, graphObj->getHuristicOf(startState), 0);  Visted->PUSH(startState);  applyIDAstar(i);  if (resultentPath != NULL)  {  Y = true;  break;  }  }  if (Y == false)  {  cout << "\nUNABLE to Reach the GOAL with F(n)Limt of " << this->maxlimtFn << ".";  }  }  void applyIDAstar(double limtFn){  while (Stack->isEmpty() == false)  {  recordStack \*tempStack = Stack->getStackWithSmallestFn();  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)  {  //----------------(previous[f(n)] - previous[h(n)]) + new[g(n)]+ new[h(n)]  double tempFn = ((tempStack->getFn() - graphObj->getHuristicOf(tempStack->rsHead->V)) + tempConectedEdgesHead->GnCost + graphObj->getHuristicOf(currentEdgeValue));  if (tempFn <= limtFn)  {  Stack->PUSH(tempStack, currentEdgeValue, tempFn, tempStack->level + 1);  }  else  {  //we added befor chacking f(n) so Pop it from Visted Stack  Visted->pop();  }  }  tempConectedEdgesHead = tempConectedEdgesHead->E\_Next;  }  }  }  }  void printIDAstarGraphWithPath(){  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 printIDAstarGraphWithoutPath(){  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;  }  };  //Recursive Breath First Search  class RBFS  {  allRecordsStack \*QueueStacks = NULL;  simpleStack \*Visted = NULL, \*Expanded = NULL;  recordStack \*smallest= NULL, \*secondSmallest= NULL;  int startState, goalState;  recordStack \*resultentPath = NULL;    public:  RBFS(){  QueueStacks = new allRecordsStack();  Visted = new simpleStack();  Expanded = new simpleStack();  startState = getValue(start[0], start[1]);  goalState = getValue(goal[0], goal[1]);  QueueStacks->PUSH(NULL, startState, graphObj->getHuristicOf(startState), 0);  Visted->PUSH(startState);  applyRBFS();  }  void applyRBFS(){  while (QueueStacks->isEmpty()==false)  {  this->smallest = QueueStacks->getStackWithSmallestFn();  this->secondSmallest = QueueStacks->getStackWithSmallestFn();    if (smallest->rsHead->V == this->goalState)  {  resultentPath = smallest;  return;  }  if (secondSmallest == NULL)  {  //Just ADD CHILD of smallest Node  // and add smallest node to the ExpandedStack  addAllChildOfParentToQueueStack(smallest, graphObj->getVertixWithValue(smallest->rsHead->V));  this->Expanded->PUSH(smallest->rsHead->V);  }  else  {  //checkChildValuesLessThenSecSmallestValue Fn with The secund smallest value if its tre then add child  //otherwise make the correction in huristic value of smallestNode  // it will give the ParentVertisx Where All the Child Exists ,Gn from start to goal , secoundSmallestFn  // if smallesr child value does not exist then value of checkNegValue will be equal to -5 or NULL  double checkNegValue = isAnyChildValuesLessThenSecSmallestValue(graphObj->getVertixWithValue(smallest->rsHead->V),(smallest->fn - graphObj->getHuristicOf(smallest->rsHead->V)), secondSmallest->fn);  if (checkNegValue != NULL)  {  // if smallest value exist in the child nodes  // add all child nodes in QueueStack  // also Add Parent to ExpandedStack  // and also add the secound smallest in the Queskack if it is not NULL  addAllChildOfParentToQueueStack(smallest, graphObj->getVertixWithValue(smallest->rsHead->V));  this->Expanded->PUSH(smallest->rsHead->V);  //----adding secoundSmallest Stack again to QueueStacks  this->addRecodStackAgain(secondSmallest);    }  else  {  //if not true so update huristic and also the F(n) f smallest  //add both smallest and the secoundSmallest in QueueStack  //cout << "\nhuristic is incorrect";  double childeSmallest = anySmallestValueInParent(graphObj->getVertixWithValue(smallest->rsHead->V), (smallest->fn - graphObj->getHuristicOf(smallest->rsHead->V)));  double diffrence = childeSmallest - smallest->fn;  this->smallest->fn += diffrence;  graphObj->setHuristicWhereValue(smallest->rsHead->V, (graphObj->getHuristicOf(smallest->rsHead->V) + diffrence));  //addSmallest Again  this->addRecodStackAgain(smallest);  //addSecoundSmallestAgain  this->addRecodStackAgain(secondSmallest);  }  }  }  }  void addRecodStackAgain(recordStack \* rS){  //----adding secoundSmallest Stack again to QueueStacks  int popHeadValue = rS->POP();  this->QueueStacks->PUSH(rS, popHeadValue, rS->fn, rS->level);  }  double anySmallestValueInParent(VERTIX \* parentVertex, double parentGn){  EDGE \* parentChild = parentVertex->E\_HEAD;  double smlChildValue = std::numeric\_limits<double>::max();  while (parentChild != NULL)  {  double childHuristic = graphObj->getHuristicOf(parentChild->E\_Value);  double startToCurrentNodeGn = parentChild->GnCost + parentGn;  double Fn = childHuristic + startToCurrentNodeGn;  if (Fn <= smlChildValue)  {  smlChildValue = Fn;  }  parentChild = parentChild->E\_Next;  }  return smlChildValue;  }  void addAllChildOfParentToQueueStack(recordStack \* parent, VERTIX \* parentVertex){    double parentGn = (parent->fn - graphObj->getHuristicOf(parent->rsHead->V));    EDGE \* parentChild = parentVertex->E\_HEAD;  while (parentChild != NULL)  {  //if (this->Visted->isAlreadyPush(parentChild->E\_Value)== false)  {  //if not in the visted List  double childHuristic = graphObj->getHuristicOf(parentChild->E\_Value);  double startToCurrentNodeGn = parentChild->GnCost + parentGn;  double Fn = childHuristic + startToCurrentNodeGn;  int level = parent->level + 1;  this->QueueStacks->PUSH(parent, parentChild->E\_Value, Fn, level);  }  //move to next child  parentChild = parentChild->E\_Next;  }  }  double isAnyChildValuesLessThenSecSmallestValue(VERTIX \* parentVertex, double parentGn, double secSmallestFn){  //Is Any Child Value have Less or equla value Then the SecoundSmallest Value  double smlChildValue = NULL;  EDGE \* parentChild = parentVertex->E\_HEAD;  while (parentChild != NULL)  {  double childHuristic = graphObj->getHuristicOf(parentChild->E\_Value);  double startToCurrentNodeGn = parentChild->GnCost + parentGn;  double Fn = childHuristic + startToCurrentNodeGn;  if (Fn <= secSmallestFn)  {  smlChildValue = Fn;  }  parentChild = parentChild->E\_Next;  }  return smlChildValue;  }  void print(){  if (showPath() == true){  printRBFSGraphWithPath();  }  else  {  printRBFSGraphWithoutPath();  }  }  bool showPath(){  cout << "\n(RBFS) Goal to Start Path: ";  if (resultentPath == NULL)  {  cout << "Does Not exist";  return false;  }  else  {  resultentPath->print();  return true;  }  }  private:  void printRBFSGraphWithPath(){  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 printRBFSGraphWithoutPath(){  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();  //graphObj->PRINT\_VERTIX();  //graphObj->PRINT\_GRAPH();  //cout << "\nSTART:" << getValue(start[0], start[1]) << "\nGOAL: " << getValue(goal[0], goal[1]) << "\n\n\n";    GBFS gbfs;  gbfs.print();  cout << "\n\n";  Astar astar;  astar.print();  cout << "\n\n";  IDAstar idastar(100);  idastar.print();  cout << "\n\n";    RBFS rbfs;  rbfs.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;  } |