Example of Disjktra Algorithm using Heap

acarlstein.com/

Posted by Alejandro G. Carlstein Ramos Mejia on October 15, 2010 November 2, 2010 About Programming / Algorithms / ANSI/POSIX C

NOTIFICATION: These examples are provided for educational purposes. The use of this code and/or information is under your own responsibility and risk. The information and/or code is given 'as is'. I do not take responsibilities of how they are used.

Example of Disjktra algorithm using heap.

disjktra_heap.c:

```
/*
 * Program: 09
 * Author: Alejandro G. Carlstein
 * Description: Disjktra Algorithm using heap
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <errno.h>
#include <string.h>
#include <malloc.h>
#include <limits.h>

#define MAX_EDGES 100001
#define MAX_VERTICES  500001
#define LEFT(i) ((i << 1) + 1)
#define RIGHT(i) ((i << 1) + 2)
#define DIV_BY_2(i) (i >> 1)
#define FALSE 0
#define TRUE 1

#define DBG_LV0 0
#define DBG_LV1 0
#define DBG_LV2 0
#define DBG_LV3 0

struct Edge{
  int startVertex;
  int endVertex;
  unsigned int length;
} edges[MAX_EDGES];

struct Vertex{
 unsigned int id;
 unsigned int weight;
 struct Vertex *predecessor;
 short visited;
} vertices[MAX_VERTICES];

struct AdjList{
 struct Vertex *vertexPointer;
 int edgeIndex;
 struct AdjList *next;
} *adjList[MAX_VERTICES], *headAdj, *newAdj;

void init(void);
void readStandardInput(void);
void printEdges(void);
void disjktraAlgorithm(void);
void initializeSingleSource(void);
void printVertices(void);
void buildAdjacentVertexList(void);
void insertAdjVerticesOf(int vertexIndex);
void insertAdjVertex(int vertexIndex, int adjVertexIndex, int edgeIndex);
void printAdjList(int vertexIndex);
void makePriorityQueue(int heapArray[], int *length);
void printArray(int array[], int length);
void buildMinHeap(int heapArray[], int length);
void minHeapify(int heapArray[], int index, int heapSize);
int heapExtractMin(int heapArray[], int *heapSize);
void exchange(int *a, int *b);
void addVertexToList(struct AdjList *vertexList, struct Vertex *vertex);
void relax(int startVertex, int endVertex, int edge);
void printRoute(void);
void debug(int debugLevel, char *fmt, ...);
void errorDoExit(char *fmt, ...);

int numVertices, numEdges;
int main(int argc, char *argv[]){

 init();
 disjktraAlgorithm();
 printVertices();
 return 0;
}

void init(void){
 debug(DBG_LV0, 'init()');
 readStandardInput();
}

void readStandardInput(void){
 debug(DBG_LV0, 'readStandardInput()');

 scanf('%d %d', &numVertices, &numEdges);
 debug(DBG_LV1, '# of vertices: %d, # of edges: %d', numVertices, numEdges);
 printEdges();
}
```

```
void printEdges(void){
 debug(DBG_LV0, 'printEdges()');

 int i;
 for(i = 0; i < numEdges; ++i){
  scanf('%d %d %d', &edges[i].startVertex, &edges[i].endVertex, &edges[i].length);
  debug(DBG_LV1, '[%d](%d <%d> %d)', i, edges[i].startVertex, edges[i].length,
edges[i].endVertex);
 }
}

// Disjktra(G, w, s) //s is source vertex - w is weight
void disjktraAlgorithm(void){
 debug(DBG_LV0, 'disjktraAlgorithm()');

 // InitializeSingleSource(G, s);
 initializeSingleSource();

 // S <- 0    // It will hold vertices
 struct AdjList *vertexList = NULL;

 // Q <- V[G] // Make a priority Queue for the vertices
 int heapArray[numVertices];
 int heapLength = 0;
 makePriorityQueue(heapArray, &heapLength);

 // while Q != 0
 while(heapLength > 0){

  //  do u <- extractMin(Q)
  if (DBG_LV1) printArray(heapArray, heapLength);
  int vertexIndex = heapExtractMin(heapArray, &heapLength);
  debug(DBG_LV1, '[!]vertex[%d] Weight: %d\n', vertexIndex,
vertices[vertexIndex].weight);
  if (DBG_LV1) printArray(heapArray, heapLength);

  //S <- S U {u}
  addVertexToList(vertexList, &vertices[vertexIndex]);

  //  for each vertex v e Adj[u]
  struct AdjList *tempAdj;
  tempAdj = adjList[vertexIndex];
  while(tempAdj != NULL){
   //   do relax(u, v, w);
   relax(vertexIndex, tempAdj->vertexPointer->id, tempAdj->edgeIndex);
   tempAdj = tempAdj->next;
  }
  vertices[vertexIndex].visited = TRUE;
 }

}
```

```
void initializeSingleSource(void){
 debug(DBG_LV0, 'initializeSingleSource()');

 int i;
 for(i = 0; i < numVertices; ++i){
  vertices[i].id = i;
  vertices[i].weight = INT_MAX;
  vertices[i].predecessor = NULL; // p[v] predecessor that is either another vertex or
NULL
  vertices[i].visited = FALSE;
 }
 vertices[0].weight = 0;

 if (DBG_LV1) printVertices();

 buildAdjacentVertexList();

 if (DBG_LV1){
  debug(DBG_LV1, '*List of all adj. vertices*');
  for (i = 0; i < numVertices; ++i)
   printAdjList(i);
 }

}

void printVertices(void){
 debug(DBG_LV0, 'printVertices()');

 printf('[ ]');
 int i;
 for (i = 0; i < numVertices; ++i)
  printf('[%d]', i);

 if (DBG_LV1){
  printf('\n[i]');
  for (i = 0; i < numVertices; ++i)
   printf(' %d ' , vertices[i].id);
 }

 printf('\n[W]');
 for (i = 0; i < numVertices; ++i)
  if (vertices[i].weight == INT_MAX){
   printf(' ! ');
  }else{
   printf(' %d ' , vertices[i].weight);
  }

 if(DBG_LV1){
  printf('\n[*]');
  for (i = 0; i < numVertices; ++i)
   if (vertices[i].predecessor == NULL){
    printf(' N ');
   }else{
    printf(' %d ' , (int)vertices[i].predecessor->id);
   }
 }

 if(DBG_LV1){
  printf('\n[V]');
  for (i = 0; i < numVertices; ++i)
   if (vertices[i].visited == TRUE){
    printf(' Y ');
   }else{
    printf(' N ');
   }
 }

 printf('\n');
}

void buildAdjacentVertexList(void){
 debug(DBG_LV0, 'buildAdjacentVertexList()');

 int vertexIndex;
 for (vertexIndex = 0; vertexIndex < numVertices; ++vertexIndex){
  insertAdjVerticesOf(vertexIndex);
 }
}

void insertAdjVerticesOf(int vertexIndex){
 debug(DBG_LV0, 'insertAdjVerticesOf(vertexIndex: %d)', vertexIndex);

 debug(DBG_LV1, ' Searching for adjacent Vertices');
 int edgeIndex;
 for (edgeIndex = 0; edgeIndex < numEdges; ++edgeIndex){

  int startVertex = edges[edgeIndex].startVertex;
  int endVertex = edges[edgeIndex].endVertex;
  debug(DBG_LV1, '  Edge[%d](%d <%d> %d)', edgeIndex, startVertex,
edges[edgeIndex].length, endVertex);

  debug(DBG_LV1, '  (startVertex (%d) == (%d) vertexIndex)?', startVertex,
vertexIndex);
  if (startVertex == vertexIndex){
   debug(DBG_LV1, '   YES');
   insertAdjVertex(vertexIndex, endVertex, edgeIndex);
   insertAdjVertex(endVertex, vertexIndex, edgeIndex);
  }
 }
 if(DBG_LV1) printAdjList(vertexIndex);
}

void insertAdjVertex(int vertexIndex, int adjVertexIndex, int edgeIndex){

 struct AdjList *headAdj;
 headAdj = adjList[vertexIndex];
 struct AdjList *newAdj = (struct AdjList *) malloc(sizeof(struct AdjList));
 newAdj->vertexPointer = &vertices[adjVertexIndex];
 newAdj->edgeIndex = edgeIndex;
 newAdj->next = NULL;

 if (headAdj == NULL){
  newAdj->next = adjList[vertexIndex];
  adjList[vertexIndex] = newAdj;
 }else{
  struct AdjList *currentAdj = adjList[vertexIndex];
  while (currentAdj->next != NULL)
   currentAdj = currentAdj->next;
  newAdj->next = currentAdj->next;
  currentAdj->next = newAdj;
 }
}

void printAdjList(int vertexIndex){
 debug(DBG_LV0, '**printAdjList(vertexIndex: %d)', vertexIndex);

 debug(DBG_LV1, 'Adjacent Vertices of Vertex: %d', vertexIndex);

 struct AdjList *tempAdj;
 tempAdj = adjList[vertexIndex];

 if (tempAdj == NULL){
  debug(DBG_LV1, ' tempAdj is empty');
 }else{

  printf('[%d]', vertexIndex);
  int i = 0;
  while(tempAdj != NULL){
   printf('[%d]', i++);
   tempAdj = tempAdj->next;
  }

  printf('\n[i]');
  tempAdj = adjList[vertexIndex];
  while(tempAdj != NULL){
   printf(' %d ', tempAdj->vertexPointer->id);
   tempAdj = tempAdj->next;
  }

  printf('\n[W]');
  tempAdj = adjList[vertexIndex];
  while(tempAdj != NULL){
   if (tempAdj->vertexPointer->weight == INT_MAX){
    printf(' ! ');
   }else{
    printf(' %d ' , tempAdj->vertexPointer->weight);
   }
   tempAdj = tempAdj->next;
  }

  printf('\n[*]');
  tempAdj = adjList[vertexIndex];
  while(tempAdj != NULL){
   if (tempAdj->vertexPointer->predecessor == NULL){
    printf(' N ');
   }else{
    printf(' %d ' , tempAdj->vertexPointer->predecessor->id);
   }
   tempAdj = tempAdj->next;
  }

  printf('\n[V]');
  tempAdj = adjList[vertexIndex];
  while(tempAdj != NULL){
   if (tempAdj->vertexPointer->visited == TRUE){
    printf(' Y ');
   }else{
    printf(' N ');
   }
   tempAdj = tempAdj->next;
  }

  printf('\n');
 }
}

void makePriorityQueue(int heapArray[], int *length){
 debug(DBG_LV0, 'makePriorityQueue(length: %d)', *length);

 int i;
 for (i = 0; i < numVertices; ++i)
  heapArray[i] = i;

 *length = numVertices;
 if (DBG_LV1) printArray(heapArray, *length);
 buildMinHeap(heapArray, *length);
 if (DBG_LV1) printArray(heapArray, *length);
 if (DBG_LV1) printVertices();
}

void printArray(int array[], int length){
 debug(DBG_LV0, 'printArray(length: %d)', length);

 int i;
 for (i = 0; i < length; ++i)
  printf('[%d]', i);

 printf('\n');
 for (i = 0; i < length; ++i)
  printf(' %d ', array[i]);
 printf('\n');
```

```
}

void buildMinHeap(int heapArray[], int length){
 debug(DBG_LV0, 'buildMinHeap(length: %d)', length);

 int heapSize = length;

 int index;
 for (index = DIV_BY_2(length); index > -1; --index)
  minHeapify(heapArray, index, heapSize);
}

void minHeapify(int heapArray[], int index, int heapSize){
 debug(DBG_LV2, 'minHeapify(index: %d, heapSize: %d)', index, heapSize);

 int smallestIndex = index;
 int leftIndex = LEFT(index);
 int rightIndex = RIGHT(index);
 debug(DBG_LV1, '-SmallestIndex: %d, leftIndex: %d, rightIndex: %d', smallestIndex,
leftIndex, rightIndex);

 if (leftIndex <= heapSize && rightIndex <= heapSize){

  unsigned int indexValue = vertices[heapArray[index]].weight;
  debug(DBG_LV1,
     'INDEX: vertices[heapArray[%d]: %d].id: %d, .weight: %d',
     index, heapArray[index], vertices[heapArray[index]].id,
     indexValue);

  unsigned int leftValue = vertices[heapArray[leftIndex]].weight;
  debug(DBG_LV1,
     'LEFT VALUE: vertices[heapArray[%d]: %d].id: %d, .weight: %d',
     leftIndex, heapArray[leftIndex], vertices[heapArray[leftIndex]].id,
     leftValue);

  if ((leftIndex < heapSize) && (leftValue < indexValue))
   smallestIndex = leftIndex;

  debug(DBG_LV1, 'smallestIndex: %d', smallestIndex);

  unsigned int rightValue = vertices[heapArray[rightIndex]].weight;

  debug(DBG_LV1,
     'LEFT VALUE: vertices[heapArray[%d]: %d].id: %d, .weight: %d',
     rightIndex, heapArray[rightIndex], vertices[heapArray[rightIndex]].id,
     rightValue);

  if ((rightIndex < heapSize) && (rightValue < indexValue))
   smallestIndex = rightIndex;

  debug(DBG_LV1, 'smallestIndex: %d', smallestIndex);

  if (smallestIndex != index){
   exchange(&heapArray[index], &heapArray[smallestIndex]);
   minHeapify(heapArray, smallestIndex, heapSize);
  }
 }
}

int heapExtractMin(int heapArray[], int *heapSize){
 debug(DBG_LV0, 'heapExtractMin(heapSize: %d)', *heapSize);

 if (*heapSize < 1)
  errorDoExit('Heap Underflow');

 buildMinHeap(heapArray, *heapSize);

 int min = heapArray[0];
 --*heapSize;
 heapArray[0] = heapArray[*heapSize];
 minHeapify(heapArray, 1, *heapSize);

 return min;
}

void exchange(int *a, int *b){
 debug(DBG_LV3, 'exchange(a: %d, b: %d)', *a, *b);

  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}

void addVertexToList(struct AdjList *vertexList, struct Vertex *vertex){
 debug(DBG_LV0, 'addVertexToList()');

 struct AdjList *newAdj = (struct AdjList *) malloc(sizeof(struct AdjList));
 newAdj->vertexPointer = vertex;
 if (vertexList == NULL){
  newAdj->next = vertexList;
  vertexList = newAdj;
 }else{
  struct AdjList *currentAdj = vertexList;
  while (currentAdj->next != NULL)
   currentAdj = currentAdj->next;
  newAdj->next = currentAdj->next;
  currentAdj->next = newAdj;
 }
}

void relax(int startVertex, int endVertex, int edgeIndex){
 debug(DBG_LV0, 'relax(startVertex: %d, endVerted: %d, edgeIndex: %d)', startVertex,
endVertex, edgeIndex);

 unsigned int weight = vertices[startVertex].weight + edges[edgeIndex].length;
 debug (DBG_LV1, 'vertices[%d].weight(%d) > (%d)weight', endVertex,
vertices[endVertex].weight, weight);
 if (vertices[endVertex].weight > weight){
  vertices[endVertex].weight = weight;
  vertices[endVertex].predecessor = &vertices[startVertex];
 }
}

void debug(int debugLevel, char *fmt, ...){
 if (debugLevel){
  va_list argp;
  fprintf(stdout, '[DBG] ');
  va_start(argp, fmt);
  vfprintf(stdout, fmt, argp);
  va_end(argp);
  fprintf(stdout, '\n');
 }
}

void errorDoExit(char *fmt, ...){
 va_list argp;
 fprintf(stderr, '[Error] ');
 va_start(argp, fmt);
 vfprintf(stderr, fmt, argp);
 va_end(argp);
 if (errno){
  fprintf(stderr, '=> %s\n', strerror(errno));
 }else{
  fprintf(stderr, '\n');
 }
 exit(1);
}
```

input.txt:

```
8 11
0 1 1
4 5 1
1 5 2
2 5 2
5 6 2
0 3 2
3 4 3
5 7 3
4 2 4
4 6 4
1 2 5
```

If you encounter any problems or errors, please let me know by providing an example of the code, input, output, and an explanation. Thanks.