Graph Representations Redux

System Software Development

Eric Aubanel, Fall 2017

Recall Graph Representations

- Adjacency Matrix
- Adjacency List
- Compressed Sparse Row

Main disadvantage of CSR: can't easily modify graph

Adjacency Lists

- Resizable ragged 2D array?
- Linked list of linked lists?
- Array of linked lists?

Discuss!

Array of Linked Lists

- Modified from *Algorithms in C, Third Edition*, by Robert Sedgewick, Addison-Wesley, 2002.
- Abstract Data Type (ADT)
  - Data type defined by the operations performed by the user
  - Abstract because implementation is hidden

ADT: graph.h

```
typedef struct {
    int v;
    int w;
    int wt;
} Edge;

/* GRAPHinit and GRAPHinsertE return 1 if succ
and return 0 if malloc fails */
int GRAPHinit(int);
int GRAPHinsertE(Edge);
void GRAPHdestroy();
int *SSSP(int);
```

Limitation

- This is an ADT, but can only create one graph!
- In C we can use *opaque pointers* to implement ADTs
- See *Algorithms in C*

ADT: graph.c

```
#include <stdlib.h>
#include "graph.h"
typedef struct graphnode{
    int v;
    int wt;
    struct graphnode *next;
} Node;
typedef struct graph{
    int V;
    int E;
    Node **adj;  //array of ptrs to linked list
}Graph;
Graph *G;
```

Initialize

```
int GRAPHinit(int V){
    G = malloc(sizeof(Graph));
    if(G == NULL)
        return 0;
    G->V = V;
    G->E = 0;
    G->adj = calloc(V, sizeof(Node *));
    if(G->adj == NULL){
        free(G);
        return 0;
    }
    return 1;
}
```

Insert Edge

```
static Node *constructEdge(int v, int wt,
        Node *next){
    Node *p = malloc(sizeof(Node));
    if(p == NULL)
        return NULL;
    p->v = v;
    p->wt = wt;
    p->next = next;
    return p;
}
```

```
int GRAPHinsertE(Edge e){
    int v = e.v;
    G->adj[v] =
            constructEdge(e.w, e.wt, G->adj[v])
    if(G->adj[v] == NULL)
        return 0;
    G->E++;
    return 1;
}
```

Destroy!

```
void GRAPHdestroy(){
    for(int v=0; v < G->V; v++){
        Node *t = G->adj[v];
        while(t != NULL){
            Node *temp = t;
            t = t->next;
            free(temp);
        }
    }
    free(G->adj);
    free(G);
}
```

SSSP Using Graph ADT

```
/* Using Graph ADT to solve SSSP
 * Requires graph input in edge list
 */
#include <stdio.h>
#include "graph.h"
int main(){
  int n, m;
  Edge e;
  scanf("%d %d",&n, &m);
  if(!GRAPHinit(n)){
    printf("couldn't allocate memory for graph\n
    return 1;
  }
```

```
for(int k=0; k<m; k++){
  if(scanf("%d %d %d", &e.v, &e.w, &e.wt)!=3)
    printf("input invalid\n");
    return 1;
  }
  if(e.v > n-1 || e.w > n-1){
    printf("vertex index too large\n");
    return 1;
  }
  if(!GRAPHinsertE(e)){
    printf("couldn't insert edge\n");
    return 1;
  }
}
```

```
int s;
scanf("%d",&s);
if(s >= n || s < 0){
  printf("invalid source vertex\n");
  return 1;
}
int *D = SSSP(s);
if(D == NULL){
      printf("SSSP malloc error\n");
      return 1;
}
for(int i=0;i<n;i++)
    printf("%d ",D[i]);
printf("\n");
GRAPHdestroy();
```

Bellman-Ford SSSP in graph.c

```
int *SSSP(int s){
  int n = G->V;
  int *queue;//circular buffer to hold FIFO queue
  int *inQueue;//sparse representation of queue
  int front = 0; //index of front of queue
  int size = 1; //size of list (has source vertex initia
  int *D;//distance to each vertex
  if((queue = malloc(n*sizeof(int))) == NULL) return NULL
  if((inQueue = malloc(n*sizeof(int))) == NULL) return NU
  if((D = malloc(n*sizeof(int))) == NULL) return NULL;
  for(int i=0; i <n; i++){
      D[i] = INT_MAX;
      inQueue[i] = 0;
  }
  D[s] = 0, queue[0] = s, inQueue[s] = 1;
```

```
while(size > 0){
  int i = queue[front];
      front = (front+1)%n, size--;
      inQueue[i] = 0;
      for(Node *t = G->adj[i]; t!= NULL; t = t->next){
          int j = t->v;
          int wt = t->wt;
          if(D[j] > D[i]+wt){
              D[j] = D[i] + wt;
              if(!inQueue[j]){
                  queue[(front+size)%n] = j;
                  size++;
                  inQueue[j] = 1;
              }
          }
      }
}
return D;
```