`char` … 1 byte

`int,float` … 4 bytes

`double` … 8 bytes

`any_type *` … 4 bytes

1. `insertionSort()`
2. `numbers[0]`
3. `n`
4. `array[0]`
5. `element`

1. code
2. global
3. stack
4. global
5. stack

```
TicketT *tickets = malloc(1000 * sizeof(TicketT));
assert(tickets != NULL);
```

`tickets` is a variable located in the stack

`*tickets` is in the heap (after `malloc`'ing memory)

A program which does not `free()` each object before the last reference to it is lost contains a memory leak.

```
struct node {
   int data;
   struct node *next;
};
```

```
typedef struct node {
   int data;
   struct node *next;
} NodeT;
```

does not work:

```
typedef struct {
   int data;
   NodeT *next;
} NodeT;
```

illegal in C:

```
struct node {
   int data;
   struct node recursive;
};
```

Linked list nodes are typically located in the heap

- because nodes are dynamically created

```
#include <stdlib.h>
#include <assert.h>
#include "stack.h"

typedef struct node {
   int data;
   struct node *next;
} NodeT;

typedef struct StackRep {
   int     height;   // #elements on stack
   NodeT *top;       // ptr to first element
} StackRep;

// set up empty stack
stack newStack() {
   stack S = malloc(sizeof(StackRep));
   S->height = 0;
   S->top = NULL;
   return S;
}

// remove unwanted stack
void dropStack(stack S) {
   NodeT *curr = S->top;
   while (curr != NULL) {  // free the list
      NodeT *temp = curr->next;
      free(curr);
      curr = temp;
   }
   free(S);           // free the stack rep
}
```

Variables containing pointers to list nodes

- are likely to be local variables (in the stack)

```
// check whether stack is empty
int StackIsEmpty(stack S) {
   return (S->height == 0);
}

// insert an int on top of stack
void StackPush(stack S, int v) {
   NodeT *new = malloc(sizeof(NodeT));
   assert(new != NULL);
   new->data = v;
   // insert new element at top
   new->next = S->top;
   S->top = new;
   S->height++;
}

// remove int from top of stack
int StackPop(stack S) {
   assert(S->height > 0);
   NodeT *head = S->top;
   // second list element becomes new top
   S->top = S->top->next;
   S->height--;
   // read data off first element, then free
   int d = head->data;
   free(head);
   return d;
}
```

Constant ≅ 1

Logarithmic ≅ log *n*

Linear ≅ *n*

N-Log-N ≅ *n* log *n*

Quadratic ≅ n^2

Cubic ≅ n^3

Exponential ≅ 2^n

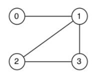
A graph with *V* vertices has at most *V(V-1)/2* edges

- if *E* is closer to V^2, the graph is dense
- if *E* is closer to *V*, the graph is sparse

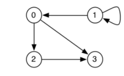
Array-of-edges Representation

Edges are represented as an array of **`Edge`** values (= pairs of vertices)

- space efficient representation
- adding and deleting edges is slightly complex
- undirected: order of vertices in an **`Edge`** doesn't matter
- directed: order of vertices in an **`Edge`** encodes direction



[(0,1), (1,2), (1,3), (2,3)]



[(1,0), (1,1), (0.2), (0,3), (2,3)]

```
newGraph(V):
|  Input  number of nodes V
|  Output new empty graph
|
|  g.nV = V    // #vertices (numbered 0..V-1)
|  g.nE = 0    // #edges
|  allocate enough memory for g.edges[]
|  return g
```

```
insertEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  i=0
|  while i<g.nE ∧ (v,w)≠g.edges[i] do
|     i=i+1
|  end while
|  if i=g.nE then          // (v,w) not found
|     g.edges[i]=(v,w)
|     g.nE=g.nE+1
|  end if
```

```
removeEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  i=0
|  while i<g.nE ∧ (v,w)≠g.edges[i] do
|     i=i+1
|  end while
|  if i<g.nE then                    // (v,w) found
|     g.edges[i]=g.edges[g.nE-1]     // replace by last edge in array
|     g.nE=g.nE-1
|  end if
```

Storage cost: *O(E)*

Cost of operations:

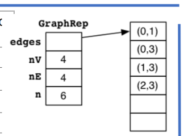
- initialisation: *O(1)*
- insert edge: *O(E)*
- delete edge: *O(E)*

If we maintain edges in order

- use binary search to find edge ⇒ *O(log E)*

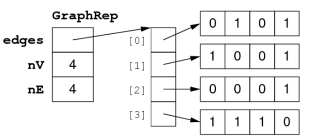
	array of edges	adjacency matrix	adjacency list
space usage	*E*	V^2	*V+E*
initialise	*1*	V^2	*V*
insert edge	*E*	*1*	*1*
remove edge	*E*	*1*	*E*

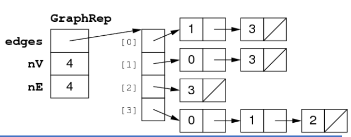
	array of edges	adjacency matrix	adjacency list
disconnected(v)?	*E*	*V*	*1*
isPath(x,y)?	*E·log V*	V^2	*V+E*
copy graph	*E*	V^2	*V+E*
destroy graph	*1*	*V*	*V+E*



Depth-first Search

```
hasPath(G,src,dest):
|  Input  graph G, vertices src,dest
|  Output true if there is a path from src to dest in G,
|         false otherwise
|
|  return dfsPathCheck(G,src,dest)

dfsPathCheck(G,v,dest):
|  mark v as visited
|  for all (v,w)∈edges(G) do
|  |  if w=dest then            // found edge to dest
|  |     return true
|  |  else if w has not been visited then
|  |     if dfsPathCheck(G,w,dest) then
|  |        return true         // found path via w to dest
|  |     end if
|  |  end if
|  end for
|  return false                 // no path from v to dest
```

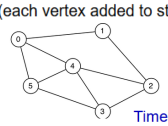




Time complexity of DFS: *O(V+E)* (adjacency list representation)

(via a stack Time complexity is the same: *O(V+E)*

(each vertex added to stack once, each element in vertex's adjacency list visited once)



		1	2	3			
		4	4	4	4		
(empty) →	0 →	5 →	5 →	5 →	5 →	5 →	(empty)

Push neighbours in descending order

Time complexity of BFS: *O(V+E)*

Breadth-first Search

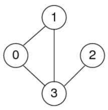
```
visited[] // array of visiting orders, indexed

findPathBFS(G,src,dest):
|  Input   graph G, vertices src,dest
|
|  for all vertices v∈G do
|     visited[v]=-1
|  end for
|  found=false
|  visited[src]=src
|  enqueue src into new queue q
|  while ¬found ∧ q is not empty do
|  |  dequeue v from q
|  |  for each neighbour w of v do
|  |  |  if visited[w]=-1 then
|  |  |     visited[w]=v
|  |  |     if w=dest then
|  |  |        found=true
|  |  |     else
|  |  |        enqueue w into q
|  |  |     end if
|  |  |  end if
|  |  end for
|  end while
|  if found then
|     display path in dest..src order
|  end if
```

Computing Connected Components

```
components(G):
|  Input graph G
|
|  for all vertices v∈G do
|     componentOf[v]=-1
|  end for
|  compID=0
|  for all vertices v∈G do
|  |  if componentOf[v]=-1 then
|  |     dfsComponents(G,v,compID)
|  |     compID=compID+1
|  |  end if
|  end for

dfsComponents(G,v,id):
|  componentOf[v]=id
|  for all vertices w adjacent to v do
|     if componentOf[w]=-1 then
|        dfsComponents(G,w,id)
|     end if
|  end for
```

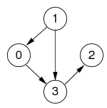
Adjacency Matrix

if few edges (sparse) ⇒ memory-inefficient



Undirected graph

A	0	1	2	3
0	0	1	0	1
1	1	0	0	1
2	0	0	0	1
3	1	1	1	0



Directed graph

A	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	0	0	0	0
3	0	0	1	0

```
newGraph(V):
|  Input   number of nodes V
|  Output  new empty graph
|
|  g.nV = V    // #vertices (numbered 0..V-1)
|  g.nE = 0    // #edges
|  allocate memory for g.edges[][]
|  for all i,j=0..V-1 do
|     g.edges[i][j]=0     // false
|  end for
|  return g
```

```
insertEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if g.edges[v][w]=0 then
|     g.edges[v][w]=1
|     g.edges[w][v]=1
|     g.nE=g.nE+1
|  end if
```

```
show(g):
|  Input graph g
|
|  for all i=0 to g.nV-1 do
|  |  for all j=i+1 to g.nV-1 do
|  |     if g.edges[i][j]≠0 then
|  |        print i"—"j
|  |     end if
|  |  end for
|  end for
```

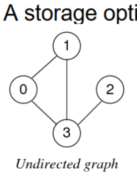
```
removeEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if g.edges[v][w]≠0 then
|     g.edges[v][w]=0
|     g.edges[w][v]=0
|     g.nE=g.nE-1
|  end if
```

Storage cost: $O(V^2)$

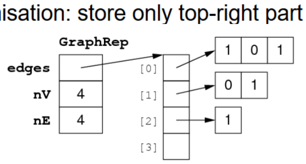
- initialisation: $O(V^2)$ (initialise *V*×*V* matrix)
- insert edge: *O(1)* (set two cells in matrix)
- delete edge: *O(1)* (unset two cells in matrix)

If the graph is sparse, most storage is wasted.

A storage optimisation: store only top-right part of matrix.



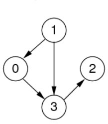
Undirected graph



New storage cost: *V-1* int ptrs + *V(V+1)/2* ints (but still $O(V^2)$)

Requires us to always use edges *(v,w)* such that *v* < *w*.

Adjacency List



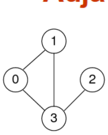
Directed graph

A[0] = <3>

A[1] = <0, 3>

A[2] = < >

A[3] = <2>



Undirected graph

A[0] = <1, 3>

A[1] = <0, 3>

A[2] = <3>

A[3] = <0, 1, 2>

Disadvantages:

- memory efficient if *E*:*V* relatively small
- one graph has many possible representations

Storage cost: *O(E)*

- initialisation: *O(V)*
- insert edge: *O(1)*
- delete edge: *O(E)*

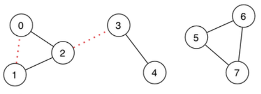
If vertex lists are sorted

- insert requires search of list ⇒ *O(E)*
- delete always requires a search, regardless of list order

```
removeEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if inLL(g.edges[v],w) then
|     deleteLL(g.edges[v],w)
|     deleteLL(g.edges[w],v)
|     g.nE=g.nE-1
|  end if
```

```
insertEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if ¬inLL(g.edges[v],w) then
|     insertLL(g.edges[v],w)
|     insertLL(g.edges[w],v)
|     g.nE=g.nE+1
|  end if
```

```
newGraph(V):
|  Input   number of nodes V
|  Output  new empty graph
|
|  g.nV = V    // #vertices (numbered 0..V-1)
|  g.nE = 0    // #edges
|  allocate memory for g.edges[]
|  for all i=0..V-1 do
|     g.edges[i]=NULL    // empty list
|  end for
|  return g
```



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
-1	-1	-1	-1	-1	-1	-1	-1
0	-1	-1	-1	-1	-1	-1	-1
0	-1	0	-1	-1	-1	-1	-1
0	0	0	-1	-1	-1	-1	-1
0	0	0	1	-1	-1	-1	-1
...							
0	0	0	1	1	2	2	2

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
-1	-1	-1	-1	-1	-1	-1	-1
0	-1	-1	-1	-1	-1	-1	-1
0	0	-1	-1	-1	-1	-1	-1
0	0	0	-1	-1	-1	-1	-1
...							
0	0	0	0	0	1	1	1

Hamiltonian Path

```
visited[]  // array [0..nV-1] to keep track of visited vertices

hasHamiltonianPath(G,src,dest):
|  for all vertices v∈G do
|     visited[v]=false
|  end for
|  return hamiltonR(G,src,dest,#vertices(G)-1)

hamiltonR(G,v,dest,d):
|  Input G     graph
|        v     current vertex considered
|        dest  destination vertex
|        d     distance "remaining" until path found
|
|  if v=dest then
|     if d=0 then return true else return false
|  else
|  |  visited[v]=true
|  |  for each (v,w)∈edges(G) ∧ ¬visited[w] do
|  |     if hamiltonR(G,w,dest,d-1) then
|  |        return true
|  |     end if
|  |  end for
|  end if
|  visited[v]=false            // reset visited mark
|  return false
```

worst case requires *(V-1)!* paths to be examined

Euler Path and Circuit

Theorem. A graph has an Euler circuit if and only if it is connected and all vertices have even degree

Theorem. A graph has a non-circuitous Euler path if and only if it is connected and exactly two vertices have odd degree

```
hasEulerPath(G,src,dest):
|  Input  graph G, vertices src,dest
|  Output true if G has Euler path from src to dest
|         false otherwise
|
|  if src≠dest then
|     if degree(G,src) is even ∨ degree(G,dest) is even then
|        return false
|     end if
|  else if degree(G,src) is odd then
|     return false
|  end if
|  for all vertices v∈G do
|     if v≠src ∧ v≠dest ∧ degree(G,v) is odd then
|        return false
|     end if
|  end for
|  return true
```

- assume that degree is available via *O(1)* lookup
- single loop over all vertices ⇒ *O(V)*

If degree requires iteration over vertices

- cost to compute degree of a single vertex is *O(V)*
- overall cost is $O(V^2)$

Digraph Representation

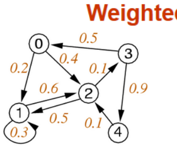
deg(v) = #edges leaving *v*

Overall, adjacency list representation is best

- real graphs tend to be sparse
  (large number of vertices, small average degree *deg(v)*)
- algorithms frequently iterate over edges from *v*

	array of edges	adjacency matrix	adjacency list
space usage	*E*	V^2	*V+E*
insert edge	*E*	*1*	*1*
exists edge *(v,w)*?	*E*	*1*	*deg(v)*
get edges leaving *v*	*E*	*V*	*deg(v)*

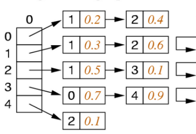
Weighted Graph



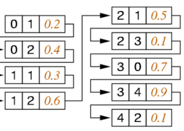
Weighted Digraph

	0	1	2	3	4
0	*	0.2	0.4	*	*
1	*	0.3	0.6	*	*
2	*	0.5	*	0.1	*
3	0.5	*	*	*	0.9
4	*	*	0.1	*	*

Adjacency Matrix



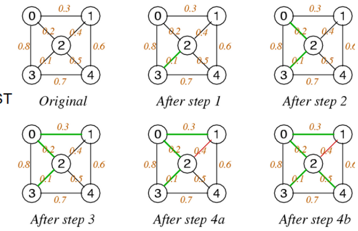
Adjacency Lists



Edge List

Kruskal's Algorithm

1. start with empty MST
2. consider edges in increasing weight order
   - add edge if it does not form a cycle in MST
3. repeat until *V-1* edges are added




```
KruskalMST(G):
|  Input  graph G with n nodes
|  Output a minimum spanning tree of G
|
|  MST=empty graph
|  sort edges(G) by weight
|  for each e∈sortedEdgeList do
|  |  MST = MST ∪ {e}
|  |  if MST has a cyle then
|  |     MST = MST \ {e}
|  |  end if
|  |  if MST has n-1 edges then
|  |     return MST
|  |  end if
|  end for
```

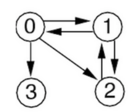
Rough time complexity analysis …

- sorting edge list is *O(E·log E)*
- at least *V* iterations over sorted edges
- on each iteration …
  - getting next lowest cost edge is *O(1)*
  - checking whether adding it forms a cycle

Transitive Closure Matrix

```
make tc[][] a copy of edges[][]
for all i∈vertices(G) do
   for all s∈vertices(G) do
      for all t∈vertices(G) do
         if tc[s][i]=1 ∧ tc[i][t]=1 then
            tc[s][t]=1
         end if
      end for
   end for
end for
```

Warshall's algorithm



digraph

	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	0	1	0	0
3	0	0	0	0

adj matrix

1st iteration `i=0`:

tc	[0]	[1]	[2]	[3]
[0]	0	1	1	1
[1]	1	1	1	1
[2]	0	1	0	0
[3]	0	0	0	0

2nd iteration `i=1`:

tc	[0]	[1]	[2]	[3]
[0]	1	1	1	1
[1]	1	1	1	1
[2]	1	1	1	1
[3]	0	0	0	0

Cost analysis:

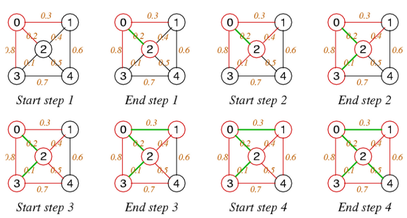
- storage: additional V^2 items (each item may be 1 bit)
- computation of transitive closure: V^3
- computation of `reachable()`: *O(1)* after having generated `tc[][]`

Alternative: use DFS in each call to `reachable()`
Cost analysis: (for adjacency matrix)

- storage: cost of queue and set during reachable
- computation of `reachable()`: cost of DFS = $O(V^2)$

Prim's Algorithm

1. start from any vertex *s* and empty MST
2. choose edge not already in MST to add to MST
   - must be incident on a vertex already connected to *s* in MST
   - must have minimal weight of all such edges
3. repeat until MST covers all vertices



Rough time complexity analysis …

- *V* iterations of outer loop
- in each iteration …
  - find min edge with set of edges is *O(E)* ⇒ *O(V·E)* overall
  - find min edge with priority queue is *O(log E)* ⇒ *O(V·log E)* overall

```
PrimMST(G):
|  Input  graph G with n nodes
|  Output a minimum spanning tree of G
|
|  MST=empty graph
|  usedV={0}
|  unusedE=edges(g)
|  while |usedV|<n do
|  |  find e=(s,t,w)∈unusedE such that {
|  |     s∈usedV ∧ t∉usedV ∧ w is min weight of all such edges
|  |  }
|  |  MST = MST ∪ {e}
|  |  usedV = usedV ∪ {t}
|  |  unusedE = unusedE \ {e}
|  end while
|  return MST
```

Dijkstra's Algorithm

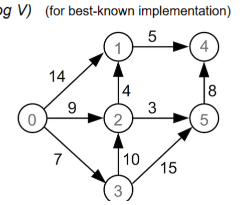
```
dist[]  // array of cost of shortest path from s
pred[]  // array of predecessor in shortest path from s

dijkstraSSSP(G,source):
|  Input graph G, source node
|
|  initialise dist[] to all ∞, except dist[source]=0
|  initialise pred[] to all -1
|  vSet=all vertices of G
|  while vSet≠∅ do
|  |  find s∈vSet with minimum dist[s]
|  |  for each (s,t,w)∈edges(G) do
|  |     relax along (s,t,w)
|  |  end for
|  |  vSet=vSet\{s}
|  end while
```

Each edge needs to be considered once ⇒ *O(E)*.

Outer loop has *O(V)* iterations.

Implementing "**`find s∈vSet with minimum dist[s]`**"

1. try all `s∈vSet` ⇒ cost = *O(V)* ⇒ overall cost = $O(E + V^2) = O(V^2)$
2. using a PQueue to implement extracting minimum
   - can improve overall cost to *O(E + V·log V)* (for best-known implementation)



	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	∞	∞	∞	∞	∞
pred	–	–	–	–	–	–

dist	0	14	9	7	∞	∞
pred	–	0	0	0	–	–

dist	0	14	9	7	∞	22
pred	–	0	0	0	–	3

dist	0	13	9	7	∞	12
pred	–	2	0	0	–	2

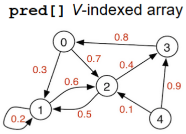
dist	0	13	9	7	20	12
pred	–	2	0	0	5	2

dist	0	13	9	7	18	12
pred	–	2	0	0	1	2

Single-source Shortest Path (SSSP)

`dist[]` *V*-indexed array of cost of shortest path from *s*

`pred[]` *V*-indexed array of predecessor in shortest path from *s*



	0	1	2	3	4
dist	0	0.3	0.7	1.1	inf
pred	-	0	0	2	-

Shortest paths from s=0