

GRAPH ALGORITHMS

Chapters 13(but not 13.5), 14(but not 14.3), 15(but not 15.4) of GT

*DISCLAIMER: Report an error / typo to the author so that he resolves it.
These notes are NOT to be distributed outside of NJIT without the expressly written permission of A. Gerbessiotis*

Graph representations

Introduction

For the sake of repetition we recite some relevant facts from Subject 3.

For a graph $G = (V, E)$ let $n = |V|$ and $m = |E|$.

A graph G is **dense** if $m \gg n$, and **sparse** otherwise. In some cases the former condition is restated as $m = \Theta(n^2)$ and the latter condition $m = o(n^2)$.

There are two major ways one can use to represent a graph

- (1) **Adjacency matrix** representation. A graph $G = (V, E)$ on n vertices and m edges is represented by an $n \times n$ two-dimensional array A . Element $A(i, j)$ is 1 if there is an edge (i, j) in the graph, otherwise it is 0. For undirected graphs, A will be symmetric since for every edge (i, j) both $A(i, j), A(j, i)$ are set to 1. In general, m entries in the directed case and $2m$ entries in the undirected case are 1, whereas the remaining ones are 0. If the edges of the graph have weights, then instead of storing an 1 we store in A the weight of the edge. Total space requirements are $O(|V|^2)$.
- (2) **Distance Matrix** is similar to the adjacency matrix; an 1 entry is replaced by the distance between the two vertices (in the direction implied by the edge), and a 0 becomes ∞ to denote that no edge exists.
- (3) **Adjacency list** representation. The graph is represented by n linked lists, one for each vertex of G . The linked list for vertex i contains all vertices that are neighbors of i , ie it contains all j such that (i, j) is an edge. For an undirected graph, j is in the linked list of i and i is in the list of j . Total space requirements are $O(|E|)$. The problem with this representation is that we cannot answer quickly questions of the form "Is j adjacent to i ?"

Note. The number of ones in the adjacency matrix is $2m$ for an undirected graph and m for a directed graph. This is a consequence of the following facts.

Fact Undirected. In an undirected graph $\sum_{v \in V} \deg(v) = 2m$.

Fact Directed. In a directed graph $\sum_{v \in V} \text{out-deg}(v) = m$.

Graph Search

Depth First Search on Undirected graphs

Depth-Search Search (DFS for short) is a method for traversing graphs. In essence it is a generalization on a graph of the depth-first order traversal on trees described earlier. Below we give a very brief outline of how the method works for the case of an undirected graph.

Method 1 (DFS on Unidirected Graphs).

```
graph    G(V,E);
boolean d[|V|] = false;    // d for discovered
label    t[|E|] = unlabeled; // t for tree

    // It just explores all vertices reachable from u

    U-DFS(G,u) // Graph G=(V,E) is undirected.
0. d[u] = true;
1. for all edges e incident on u
2.   if edge e has not been labeled
3.     let v be the other end-point of e i.e. e=(u,v)
4.     if v is unexplored // i.e. d[v] == false
5.       t[e] = tree;
6.       U-DFS(G,v);
7.     else
8.       t[e] = back;
```

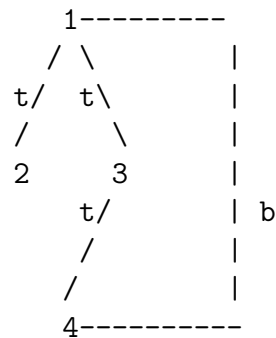
Starting from u we check the adjacency list of u . For every edge $e = (u, v)$ in this adjacency list we identify the other end-point leading from u to it. If v is undiscovered, the edge e is labeled as a **tree** edge, and a new depth-first-search exploration is initiated at v . Otherwise, if v has already been explored, we label e as a **back** edge. We can keep track of explored/unexplored vertices v through an array of vertices where we mark an explored vertex.

Procedure U-DFS has some interesting applications. The following problem can be solved easily. We have seen solutions to those problems earlier by utilizing a Union-Find data structure. DFS might provide a faster solution.

Depth First Search on Undirected graphs

Examples and Applications

In picking a vertex from the adjacency list of a vertex, say u we are going to use the convention lowest vertex label first. This means that vertices of the adjacency list of a vertex u are visited in increasing Vertex label.



Problem 1 (Reachability). Determine whether u is reachable from w .

Solution. Run U-DFS(w). If `discovered[u]` is `true` it means u is reachable from w . ■

Problem 2 (Connectivity). Determine whether graph is connected.

Solution. Run U-DFS(w). If for some vertex z `discovered[z]` is `false` it means z is unreachable from w , i.e. the graph is not connected. ■

Depth First Search on Undirected graphs

Problems continued

Problem 3 (Cycles). Does G have a cycle that includes u ?

Solution. Run $\text{U-DFS}(u)$. If there is a cycle that includes u then in the U-DFS one of its edges would be labeled as **back**. The existence of such an edge indicates a cycle. To determine whether that cycle contains u , we must trace the cycle. For that in line 5 of U-DFS we add a line that does $\text{parent}[v]=u$ i.e. it keeps track of tree edges in an alternate form. Alternatively, we can trace the cycle by walking back the path/cycle in a more laborious way. If for example we are in v we check the adjacency list of v and identify the edge e that is **tree** that leads to v from some other vertex. The other end-point (i.e. u) is traced similarly and so on. ■

Problem 4 (Connected Components). Find the connected components of G .

Solution. We start with $\text{U-DFS}(u)$ for an arbitrary u . This gives the connected component of u . If there is a vertex w that is still undiscovered (we can check this by checking $\text{discovered}[]$) we then initiate a $\text{U-DFS}(w)$ to discover the connected component of w , and go on until we discover all vertices. ■

Problem 5 (Bipartiteness). Determine whether the graph is bipartite, i.e whether we can color the vertices R or B so that no two adjacent vertices have the same color.

Solution. Left as an exercise. ■

Graph Search

Depth First Search on Directed graphs

A more general framework for DFS is shown on the following page(s). It searches a whole graph, not only the portion reachable from a given vertex such as u . It works for both directed and undirected graphs.

This generalized framework labels all vertices with two timestamps:

- $D[u]$ first visit or discovery of vertex u ,
- $F[u]$ final visit of vertex u , when all vertices in the adjacency list of u have been discovered and searched.

In addition DFS labels the edges of the graph with a total of four different labels for the directed case.

- Label tree (t),
- Label back (b),
- Label forward (f),
- Label cross (c),

For the undirected case it only uses the tree and back labels as it also does the variant of the previous section (U-DFS). Although the labelings are different for an undirected and a directed graph, in both cases, an edge labeled b indicates the existence of a cycle in the graph. The collection of all this information in depth-first-search (for a directed graph) can facilitate the easy solution of the problem of **topological sorting**.

Depth First Search on directed graphs: Starting a search.

Let $G = (V, E)$ with $|V| = n$ and $|E| = m$ be a directed graph. A DFS invocation includes the initialization of the appropriate Data Structures (arrays) by calling $\text{DFS}(G)$ and then a complete and exhaustive search of the Graph by issuing $\text{dfs}(G, u)$ function calls for every vertex u that has been left unexplored in previous invocations. The argument to $\text{DFS}(G)$ implies that G has the appropriate representation either adjacency matrix or adjacency list as needed. Several times we only need to issue one or few $\text{dfs}(G, u)$ calls. Then we initialize as needed $\text{DFS}(G)$, and issue only the relevant function call invocations instead of performing an exhaustive exploration.

DFS Its data structures

Scalar value **time** is used to timestamp entries of D and F . Timestamps are integers from 1 to $2 * n$. L is not used by default but it can collect information about the topological sorting of the vertices of the graph, if this is possible. If a topological sorting exists, at the conclusion of DFS, traversing L head to tail, would list the vertices of G on a line so that all edges could be drawn in a left to right direction: this is the definition of topological sorting.

All arrays are of length n and indexed by a vertex.

- **color[u]**. Every vertex of G is colored during DFS of G . the Color of a vertex is W, G, B. Initially, every vertex is a W (white). After exploration of u starts ($dfs(G, u)$ starts) the color of a vertex u becomes G (gray), and when it is about to complete ($dfs(G, u)$ ends) the color of u becomes B (black) and stays so. For an undirected graph only W and B are possible.
- **D[u]** records the time the exploration of u with $dfs(G, u)$ starts (when the color of u would change from W into G).
- **F[u]** records the time the exploration of u with $dfs(G, u)$ ends (when the color of u would change from G into B).
- **p[v]** records the edge (u, v) that is traversed to reach v for the first time; following this traversal v will be explored in a $dfs(G, v)$ exploration that starts immediately after the setting of $p[v] = u$.

Edges can be labeled t , f , c , and b for a directed graph G , similarly to the labels t and b utilized in the DFS of an undirected graph.

- Edge label tree (t); an edge (u, v) is labeled **t** to indicate a **tree** edge when we traverse (u, v) and determine $color[v]=W$.
- Edge label back (b); an edge (u, v) is labeled **b** to indicate a **back** edge when we traverse (u, v) and $color[v]=G$. A **b** also indicates the existence of a cycle. If a cycle exists a graph CANNOT have a topological sorting.
- Edge label cross (c); an edge (u, v) is labeled **c** to indicate a **cross** edge when we traverse (u, v) and $color[v]=B$ but $D[u] < D[v]$.
- Edge label forward (f); an edge (u, v) is labeled **f** to indicate a **forward** edge when we traverse (u, v) and $color[v]=B$ but $D[u] > D[v]$.

DFS Its data structures

Note 1.

VERTEX	STATE	COLOR	ARRAYS
	undiscovered/ unexplored	W	color[.] one of { W,G,B} p[.] parent of array D[.] discovery time
u	being explored	G	F[.] Finalization time/ end discovery time
u	discovered/ explored	B	
=====			
	Edge u->v traversed (u,v)		D[u] : record time of event color[u]=G
	t tree color[v]=W		F[u] : record time of event color[u]=B
label (u,v)	b back color[v]=G		
	f forward color[v]=B		=====
	and		Variable time : timer for D,F
	D[u]<D[v]		
	c cross color[v]=B		Optional: Linked list L
	and		(maintains topological sorting
	D[u]>D[v]		provided G has NO cycle)
=====			
	Edge u->v traversed and labeled tree edge		FOR AN UNDIRECTED GRAPH
	p[v] = u		u is either B or W
			(u,v) is labeled t or b
			everything else redundant
=====			

DFS In lieu of pseudocode

```
    dfs(G,u)      /* Note can also be adjusted for undirected graphs */
1.  color[u]=G;
2.  time++;
3.  D[u]=time;
4.  foreach (edge (u,v) in E ) { // Visit edge (u,v)
5.    if (color[v]== W ) {      // if v is W first visit
6.      p[v]=u;                // (u,v) is t, tree edge
7.      dfs(G,v);              // Go Visit v i.e. Discover v
8.    } // if
9.  } // foreach ----->9a elseif (color[v]==G )
10. color[u]=B;                9b      (u,v) is b, back edge
11. time++;                    9c      else if (color[v]==B)
12. F[u]=time;                 9d      if D[u]<D[v] (u,v) is f, forward edge
13. // Insert u at head of L    9e      else (u,v) is c, cross edge
                                9f } //This is line 9 closing brace of line 4

DFS(G) //G=(V,E)
1.  for each vertex u {
2.    color[u]=W; D[u]=F[u]=-1;
3.    p[u] = NULL;
4.  }
5.  time=0; L=NULL;           // Start timer ; L is optional (see line 13 of dfs(G,u))
6.  for each vertex u {       // Start Search
7.    if color[u] == W        { // Node u non visited yet?
8.      dfs(G,u);             // Go for u!
9.    }
10. }
```

DFS In lieu of pseudocode

The following is an alternative to function DFS(G) when we search only one vertex called w !

```
search( $G, w$ )
1. for each vertex  $u$  { //Initialize
2.   color[ $u$ ]=W;D[ $u$ ]=F[ $u$ ]=0;
3.   p[ $u$ ] = NULL;
4. }
5. time=0; L= NULL ;    // Start timer ; init L (optional)
6.   dfs( $G, w$ );
```

DRAFT

DFS1 Updated DFS

Later on we shall develop an algorithm that will find the strongly connected components of a directed graph. We need to slightly modify DFS and dfs for that. We thus generate dfs1(G,u) and DFS1(G) as shown below.

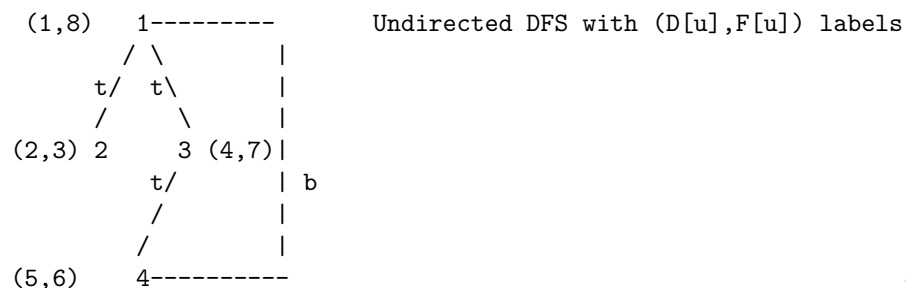
```
dfs1(G,u)
1. color[u]=G;
2. time++;
3. D[u]=time;
4. foreach (edge (u,v) in E ) { // Visit edge (u,v)
5.   if (color[v]== W ) {       // if v is W first visit
6.     p[v]=u;                  // (u,v) is t, tree edge
7.     dfs(G,v);                // Go Visit v i.e. Discover v
8.   }// if
9. }
10. color[u]=B;
11. time++;
12. F[u]=time;
13. scc(u)=s;
```

```
DFS1 (G)      // To be used in SCC(G)
1. for each vertex u {
2.   color[u]=W; D[u]=F[u]=-1;
3.   p[u] = NULL;
4. }
5. time=0; L=NULL;
6. for each vertex u {
7.   if color[u] == W {
7'.   s = u;    // To be used in SCC(G)
8.   dfs(G,u);
9.   }
10. }
```

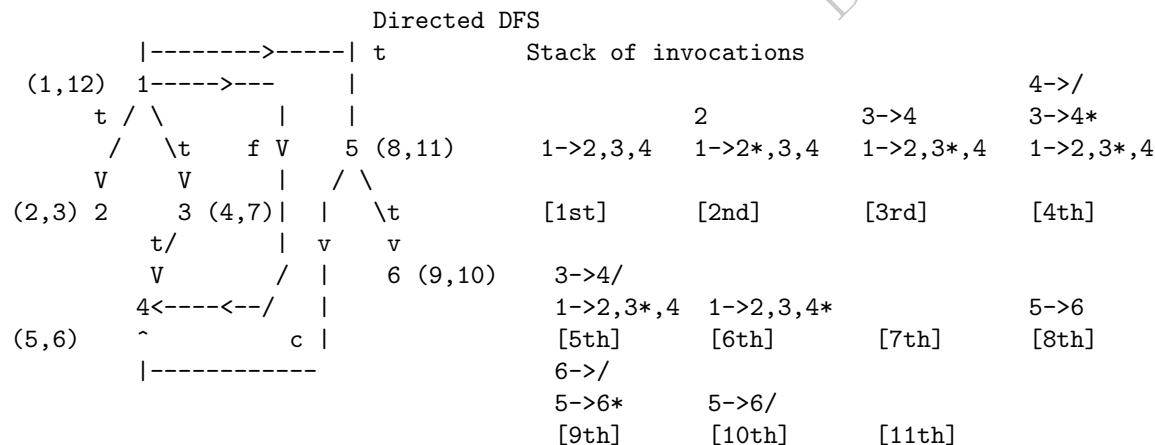
DFS Example

In picking a vertex from the adjacency list of a vertex, say u we are going to use the convention lowest vertex label first. This means that vertices of the adjacency list of a vertex u are visited in increasing Vertex label.

We could use some of the arrays of the directed graph DFS to enhance the capabilities of the undirected graph DFS pseudocode given earlier. For example, we can collect $D[u]$, $F[u]$ information for every vertex. In the examples below, both for an undirected and also a directed graph this information is shown as a pair of labels $(D[u], F[u])$ on the side of a vertex label.



Edges $(1,2)$, $(1,3)$, $(3,4)$, $(1,4)$, $(1,5)$, $(5,6)$, $(5,4)$ in directed graph below.



Problem 6 (Topological sorting).

Input: A directed graph $G = (V, E)$ that does not have a cycle.

Output: Listing of its edges on a line so that edges can be drawn in a left to right direction.

(More formally, a topological sort is a linear ordering of all its vertices such that if G contains an edge (u, v) then u appears before v in the ordering.)

Solution 1 for topological sorting. There are two solutions available at the conclusion of DFS for topological sorting. We provide first a simple solution that utilizes linked list L .

```
TopologicalSort(G)
1.  DFS(G);
2.  if G has no b edge print L
```

The running time of topological sorting is $O(DFS) + \Theta(n)$. The latter term is because L can have n vertices to list. ■

Solution 2 for topological sorting. We provide a simple solution that utilized linked list L .

```
TopologicalSort(G)
1.  DFS(G);
2.  if G has no b edge
3.      print vertices of V in decreasing order of F[u]
```

The running time is $O(DFS) + \Theta(n)$ which is $O(DFS)$. The linear bound of the second term absorbed DFS-related costs but also the cost of CountSort that is being used to sort the vertices of the graph in decreasing $F[u]$ order. We have n vertices to sort. Each $F[.]$ label is an integer from 1 to $2 * n$. This range is thus a subset of $0..2n$. The running time of CountSort is $\Theta(n + (2n + 1)) = \Theta(n)$. ■

DFS More applications

Problem 7 (Cycle).

Input. Directed graph G

Output. YES if G has a cycle, NO otherwise.

Solution. The solution is part of Topological sorting.

```
    Cycle(G)
1.   DFS(G);
2.   if G has a b edge  print YES
3.   else                print NO
```



Problem 8 (Social network of u : Is v a “friend”?).

Input. Graph G and vertices $u, v \in V$.

Output. YES if in G $u \rightsquigarrow v$, NO otherwise.

Solution.

```
    Friend(G,u,v)
1.   search(G,u);
2.   if color[v]==B    print YES
3.   else              print NO
```



DFS More applications

Problem 9 (Social network of u : All “friends”).

Input. Graph G and vertex $u \in V$.

Output. List of friends of u i.e. all v such that $u \rightsquigarrow v$.

Solution.

```
Friends(G,u)
1.  search(G,u);
2.  for all v in V and v !=u do
3.    if color[v]==B    print v is friend of u
```

■

A directed graph is strongly connected if for every two vertices u, v there is a path from u to v and also a path from v to u . One way to determine whether G is strongly connected is to initiate a DFS (more precisely a `dfs(u)`) starting from every vertex u of the graph. A better way is to replace the top call of $DFS(G)$ with a call to `search(G, u)`, the alternative to `DFS(G)` highlighted earlier. This needs to be repeated for all possible u . This is equivalent to running DFS n times altogether. A better approach requires only two DFS calls. how we can determine strong connectivity in just $O(n + m)$ time instead.

DFS More applications

Problem 10 (Transitive closure T of G).

Input. Graph G .

Output. Transitive closure T of G ; $T(i, j) = 1$ if $i \rightsquigarrow j$, and $T(i, j) = 0$ otherwise.

Solution.

```
TransitiveClosure(G)
1.  repeat for all u in V {
2.    Friends(G,u);
3.    if v is friend of u T[u][v]=1
4.    else                  T[u][v]=0
5.  }
```

Implied in line 2 is a DFS operation for every vertex u of the graph. The running time of this approach for transitive closure yields $n \times O(DFS)$ that is $O(n(n + m))$ or $O(n^3)$ depending on the graph representation. ■

DFS More applications

Problem 11 (Strongly Connected).

Input. Graph G .

Output. YES if G strongly connected; NO otherwise

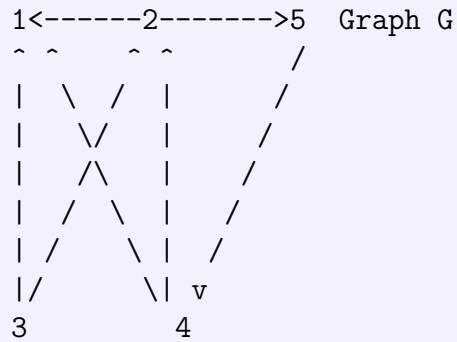
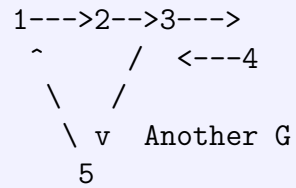
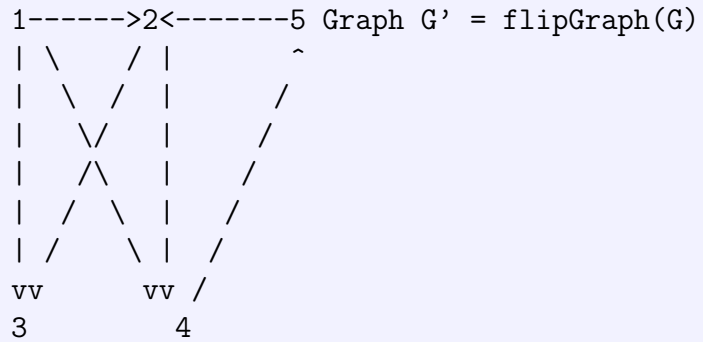
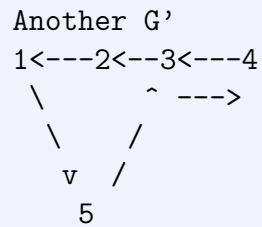
Solution.

```
    StronglyConnected
1.    T=TransitiveClosure(G);
2.    count=0
3.    for all u, v
4.        count += T[u][v]
5.    if count == n*n    print YES
6.    else                print NO
```

This approach requires n DFS operations per Transitive Closure. Later on we will present an efficient method that requires only 2 DFS operations. ■

DFS
Flip graph of G

Example 1.



DFS

More applications

Problem 12 (Strong Connectivity).

Input. Graph G .

Output. YES if G is strongly connected and NO otherwise.

Solution. StrongConnectivity(G, V, E)

```
1. Pick arbitrary  $u$  of  $V$ .
2. dfs( $u, G$ ); // Use  $G$  as input; otherwise identical to dfs( $u$ ) of page 5
3. If there is a vertex  $w$  with color[ $w$ ] == WHITE
4.   return("G is not strongly connected");
5. else // all vertices are reachable from  $u$ 
6.   create  $G' = (V, E')$  where  $E'$  contains the reverse of the edges of  $E$ 
7.   i.e. for every  $e = (a, b)$  of  $E$  add  $e' = (b, a)$  to  $E'$ 
8.   dfs( $u, G'$ ) // page 5 dfs; use  $G'$  not  $G$  as input
9.   if there is a vertex  $w$  with color[ $w$ ] == WHITE
10.    return("G is not strongly connected");
11.   else
12.    return("G is strongly connected");
```

The dfs call of line 2 can only determine whether G is NOT strongly connected. If lines 6-12 are executed, this means that all vertices are reachable from u . We then reverse the directions of all the edges of G to get G' . In G' we run DFS from u again as we did in line 2 for G . If all vertices are reachable from u in G' (i.e. we are in line 12 rather than line 10), this means that in G' we can go from u to every other vertex w . Given that G' is the reverse of G this means that we can go from every w to u in G . Therefore reaching line 12 we can do two things: (a) From u we can reach every other w (line 2 leads to line 6), and (b) from every w we can reach u (line 6 leads to line 12). This is equivalent to saying that G is strongly connected. If we want to go from a to b , we first go from a to u and then from u to b . Of course this defines not necessarily a path, but probably a tour. But it suffices to show that G is strongly connected anyway. A refinement can show how one can determine a path from this potential tour. ■

DFS More applications

Example 2.

Pick vertex u of StrongConnectivity to be vertex 25

In line 2 `run dfs(25,G);`

if line 4 DOES NOT return then (`--/-->` means path)

[A] $\overset{G}{25 \text{ --/-->} u}$ and $\overset{G}{25 \text{ --/-->} v}$

In line 8 `run dfs(25,G')`

if line 10 DOES NOT return then `====>`

$\overset{G'}{25 \text{ --/-->} u}$ and $\overset{G'}{25 \text{ --/-->} v}$
or equivalently

[B] $\overset{G}{u \text{ --/-->} 25}$ and $\overset{G}{v \text{ --/-->} 25}$

FOR ALL u FOR ALL v

[B] $\overset{G}{u \text{ --/-->} 25}$
and $\implies u \text{ --/-->} v$

[A] $\overset{G}{25 \text{ --/-->} v}$

[B] $\overset{G}{v \text{ --/-->} 25}$
and $\implies v \text{ --/-->} u$

[A] $\overset{G}{25 \text{ --/-->} u}$

Running time of DFS

Graph representation dependent

In the remainder we examine separately the running time for

- an adjacency matrix representation of G , and
- an adjacency list representation of G .

Theorem 1 (DFS running time: adjacency matrix). *The running time of $DFS(G)$, where $G = (V, E)$ is a directed graph depends on the representation of graph G . For an adjacency matrix representation the running time $T(n, m)$ of $DFS(G)$ is $T(n, m) = \Theta(n^2)$.*

Proof. (Number of dfs calls.) In order to assist the search process, vertices are colored with three colors during DFS, W, G and B. Initially all vertices are W (i.e. undiscovered). A call to $dfs(u)$ or $dfs(v)$ is always preceded by a check of the corresponding vertex's color: see line 5 of $dfs(G, u)$ or line 7 of $DFS(G)$. As soon as the corresponding **dfs** operations starts (line 7 of $dfs(G, u)$ or line 8 of $DFS(G)$) the color of the vertex being visited changes from W into a G. Eventually it will be changed into a B prior to the conclusion of the dfs operation. Every vertex is visited through dfs once and only once. If the vertex is not visited through the dfs of another vertex (Line 7 of dfs), then it will be visited through DFS in line 8 of $DFS(G)$. Thus the total number of $dfs(.)$ calls is the number of vertices and it is n .

Number of edge (u, v) visits. Every edge (u, v) is visited once through line 4 of $dfs(G, u)$ contributing an $\Omega(m)$ to $T(n, m)$. Furthermore, for an adjacency matrix representation the m edges must be extracted (identified) out of the n^2 entries of the adjacency matrix (separate the ones from the zeroes) and this would require $\Omega(n^2)$ time.

Running time of DFS. Every vertex is visited once: this means a dfs call on vertex u is issued only once. This implies an $\Omega(n)$ bound contribution to $T(n, m)$. Combining the two we have, given that $m < n^2$,

Running time of DFS

Adjacency matrix

Continued. **AdjacencyMatrix** : $T(n, m) = \Omega(n^2 + m + n) = \Omega(n^2)$. We now show that this bound is tight and thus can replace the Ω with a Θ instead. The best way to conceptualize the process of analyzing the recursive pseudocode of $\text{DFS}(G)$ that calls recursively $\text{dfs}(G, u)$ that can then call recursively $\text{dfs}(G, v)$ is to consider using $n + 1$ stop watches. One of them is active when the code runs $\text{DFS}(G)$. We stop the time of the stopwatch when line 8 is reached. The other n stopwatches are associated with the $\text{dfs}(G, u)$ and stop measuring time when a $\text{dfs}(G, v)$ invocation starts in line 7 and resumes when the semicolon of line 7 is encountered. (Program stack invocations trigger a resume or a stop of the corresponding stopwatch.) For the adjacency matrix representation we observe the following: the number of invocations of $\text{dfs}(G, u)$ is n and each invocation spends $\Theta(n)$ time overall within $\text{dfs}(G, u)$ since the adjacency list of u is formed by scanning row u of the adjacency matrix and we extract the vertices corresponding to the one entries of the adjacency matrix, a $\Theta(n)$ operation. Therefore,

$$T(n, m) = \overbrace{\Theta(n)}^{\text{n dfs(.) calls}} + \sum_{u \in V} \overbrace{\Theta(n)}^{\text{row u of adjacency Matrix}} = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

■

Running time of DFS Adjacency list

Theorem 2 (DFS running time: Adjacency list). *The running time of $DFS(G)$, where $G = (V, E)$ is a directed graph depends on the representation of graph G . For an adjacency list representation the running time $T(n, m)$ of $DFS(G)$ is $T(n, m) = \Theta(n + m)$.*

Proof. The prior discussion related to Number of dfs calls, and number of edge (u, v) visits remains the same. We deduce that

$$\text{AdjacencyList} : T(n, m) = \Omega(n + m),$$

accounting for the fact that we have n dfs calls, one per vertex, and every edge is visited once by taking advantage of the adjacency list of every node u , and thus implying an $\Omega(m)$ bound on the running time for the adjacency list representation.

We now show that this bound is tight and thus can replace the Ω with a Θ instead. We use the same stopwatch method to analyse the recursive code of $DFS(G)$. For the adjacency list representation we observe the following: the number of invocations of $dfs(G, u)$ is still n and each invocation spends $\Theta(outd(u))$ overall within $dfs(G, u)$ since the adjacency list $A(u)$ of u is $outd(u)$ vertices long, contains vertices v such that $(u, v) \in E$, and it is fully traversed in steps 4-9 of $dfs(G, u)$. Therefore

$$T(n, m) = \overbrace{\Theta(n)}^{\text{n dfs(.) calls}} + \sum_{u \in V} \overbrace{\Theta(outd(u))}^{A(u) \text{ of adjacency List}} = \Theta(n) + \Theta(m) = \Theta(n + m).$$

Thus $T(n, m) = \Theta(n + m)$. ■

DFS Further applications

Problem 13 (Strongly Connected Components).

Input. Graph G .

Output. The strongly connected components of G .

Solution.

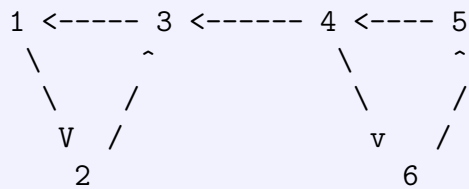
```
    SCC(G)
1.  Let  $G'$  be the flipgraph of  $G$ ;
2.  DFS ( $G'$ );
3.  Relabel  $u$  of  $V$  on decreasing  $F[.]$  value
4.  DFS1 ( $G$ );
5.  output  $scc(u)$  for all  $u$  in  $V$ 
    // note the relabeling of step 3 in step 5.
```

This approach requires two DFS operations as well. Note that another approach is to perform DFS(G) in line 2 and DFS1(G') in line 4. ■

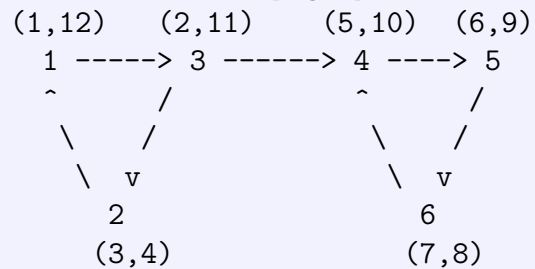
SCC Example

Example 3.

$G=(V,E)$



$G'=(V,E')$ [flip graph]



Line 1: Generate G' from G

Line 2: Run DFS on G' and generate $F[.]$ values

Line 3: Order vertices on decreasing $F[.]$ values and 'relabel'

Initially	1	2	3	4	5	6
	12	4	11	10	9	8

Relabeled	1	3	4	5	6	2
	12	11	10	9	8	4

Line 5: Run DFS1 on G

scc	1	2	3	4	5	6
----->	1	1	1	4	4	4

Breadth First Search

Another search process

In BFS we explore vertices starting from a given vertex s by first finding all vertices adjacent to s , then all vertices reachable from s within two edge traversals, i.e. at distance two, then those at distance three, and so on.

This graph exploration process can be summarized as follows.

BFS (s)

```
1. label[s] = REACHED;
2. QUEUE = <s>;
3. while QUEUE is not empty {
4.   remove u from queue; // A Dequeue operation
5.   for every vertex v in adjacency list of u do {
6.     if label[v] != REACHED {
7.       add v to QUEUE; // An Enqueue operation
8.       label v REACHED;
9.       Maintain fact that v was discovered while traversing (u,v);
10.    }
11. }
```

A more robust implementation of the pseudocode, label vertices by assigning them colors as before for DFS. Step 9 requires a single array and is equivalent to assigning $p[v] = u$ i.e. indicating that the parent of v is vertex u . The QUEUE is implemented as a FIFO queue. In addition we may keep track of the distance of a vertex v from s by using a distance array $d[]$ and in step 9 also performing the update step $d[v] = d[u] + 1$.

Breadth First Search Revisited

```
BFS(s) //s is the start vertex of the search process on Graph G=(V,E)
1.  for each u in V-{s}    {
2.      color[u]=W;
3.      d[u] = infinity; // d[u] gives the distance of u from s.
4.      p[u] = NULL      // p[u] represents the parent of u in BFS search
5.  }
6.  color[s]=G;
7.  d[s]=0;
8.  p[s]=NULL; // s is to be expanded first
9.  queue= <s> //A single queue containing s.
10. while queue != <> { // <> stands for an empty queue
11.     u = Dequeue(queue) //remove an element from queue
12.     foreach ((u,v) an edge of E) {
13.         if (color(v)== W) {
14.             color[v]=G;
15.             d[v]=d[u]+1;
16.             p[v]=u;
17.             queue = Enqueue(queue,v);
18.         } // if statement ends
19.     } // foreach statement ends
20.     color[u]=B // We are done with u (all its neighbors examined)
21. } // while loop ends
```

All-pair shortest path problem

Introduction

Problem 14 (All-pair shortest path problem).

Input. Graph $G = (V, E)$ and its associated weight matrix or weight list W .

Output. For every $i \in V$ and every $j \in V$ the weight of the path with minimal total weight from i to j maintained in a matrix $D_{i,j}$.

In this problem we do not minimize the number of edges in the path; we minimize the sum of the weights of the edges of the path. The weight of the minimal weight path from i to j will be maintained in $D_{i,j}$. If the cost of a shortest path is ∞ , then it will mean that no such path exists. A minimal weight path may not be the “shortest” in terms of number of edges in the path.

The term shortest path, minimum weight path, and minimum cost path will be used interchangeably in the remainder.

A solution to this problem for a special class of graphs utilizes dynamic programming. Dynamic programming, like the divide-and-conquer method, solves problems by dividing a problem into subproblems, solving the subproblems, and combining the solutions to subproblems. In divide and conquer, subproblems are usually independent of each other. Dynamic programming however, is used in cases where subproblems are not independent. A dynamic programming algorithm solves subproblems once, saves the solutions in a table and avoids recomputing the answer for a problem that has already been solved before. The development of a dynamic programming algorithm is broken into a sequence of steps.

- (1) Characterize the structure of an optimal solution.
- (2) Recursively define the value of an optimal solution.
- (3) Compute the value of an optimal solution in a bottom-up fashion.
- (4) Construct an optimal solution from computed information.

Solution for All-pair shortest path. A dynamic-programming based algorithm due to Floyd and Warshall can be used to solve the all-pairs shortest path problem (i.e. all-pair minimal weight path problem). The algorithm works correctly even in the presence of negative edge weights as long as there are NO NEGATIVE COST CYCLES. ■

All-pair shortest path problem

Floyd-Warshall's algorithm

The algorithm works with graphs with negative weight edges. The graphs however should not have a negative cost cycle: this is a cycle where the sum of the weights of the edges of the cycle is negative.

Theorem 3 (Floyd-Warshall). *Algorithm FloydWarshall computes in $D_{i,j}^{(n)}$ the cost of the minimal weight path from i to j . The running time of FloydWarshall is $T(n) = \Theta(n^3)$. Graph is allowed to have cycles as long as it does not have negative cost cycles (negative weight edges allowed).*

Proof.

FloydWarshall(G,A,W)

(0)

1. $D(i,j) = d(i,j)$
2. for $k=1$ to n
3. for $i=1$ to n
4. for $j=1$ to n

5. $D(i,j) = \min(D^{(k-1)}(i,j), D^{(k-1)}(i,k) + D^{(k-1)}(k,j));$

(n)

6. return($D=D^{(n)}$);

Floyd-Warshall's algorithm

Proof of correctness

Continued. We first analyze the performance of algorithm FloydWarshall. The Algorithm has three nested for loops. Thus its running time is $T(n) = \Theta(n^3)$. Although one may consider its space requirements as being $\Theta(n^3)$ since it seems to need a new $n \times n$ matrix in every iteration, in truth, we can fulfill the computation with two copies, one for the previous and one for the current iteration. Thus $S(n) = \Theta(n^2)$ for the space requirements, if $S(n)$ are the space requirements for FloydWarshall.

We provide a proof correctness using induction.

Inductive assumption. At completion of iteration k , $D_{i,j}^{(k)}$ is the cost of the minimal weight path from i to j that ONLY USES in-between vertices from set $\{1, \dots, k\}$.

Basis of induction $k = 0$. Before the loop is executed for the first time, all paths with no internal vertices are single edges; $D_{i,j}^{(0)} = W_{i,j}$ is the cost of an edge path between i and j with NO internal vertices i.e. $k \leq 0$, noting that vertex indexes are positive.

Inductive Step: $k - 1$ to k . Assume that the inductive hypothesis is true for $k - 1$ and consider the triangular inequality for k , ie.

$$D_{i,j}^{(k)} = \min\{D_{i,j}^{(k-1)}, D_{i,k}^{(k-1)} + D_{k,j}^{(k-1)}\}$$

We have two cases to consider.

Case 1. If shortest path from i to j with inbetween vertices at most k does not go through k then the part utilizing the first term of the min correctly updates D for iteration k ($D_{i,j}^{(k)} = D_{i,j}^{(k-1)}$).

Case 2. If shortest path from i to j with inbetween vertices at most k does go through k we need to check $D_{i,j}^{(k-1)}$ vs $D_{i,k}^{(k-1)} + D_{k,j}^{(k-1)}$. By inductive assumption all $D_{i,j}^{(k-1)}$, $D_{i,k}^{(k-1)}$, $D_{k,j}^{(k-1)}$ are optimal.

Conclusion: At the conclusion of iteration n , $D_{i,j}^{(n)}$ gives the cost of the shortest path from i to j . ■

Floyd-Warshall's algorithm

Example

Example 4. Run Floyd-Warshall on the graph whose W is $D^{(0)}$.

Proof.

(0) $D = \begin{array}{c|ccc} & 0 & 8 & 5 \\ \hline & 3 & 0 & \infty \\ \hline & \infty & 2 & 0 \end{array}$

(1) $D = \begin{array}{c|ccc} & 0 & 8 & 5 \\ \hline & 3 & 0 & 8_1 \\ \hline & \infty & 2 & 0 \end{array}$ $\begin{array}{l} \text{path 2 to 3 through 1 } \min(\infty, 3 + 5) = 8 \text{ through 1} \\ \text{path 3 to 2 through 1 } \min(2, \infty + 8) = 2 \text{ no change} \end{array}$

(2) $D = \begin{array}{c|ccc} & 0 & 8 & 5 \\ \hline & 3 & 0 & 8_1 \\ \hline & 5_2 & 2 & 0 \end{array}$ $\begin{array}{l} \text{path 1 to 3 through 2 } \min(5, 8 + 8) = 16 \text{ no change} \\ \text{path 3 to 1 through 2 } \min(\infty, 2 + 3) = 5 \text{ through 2} \end{array}$

(3) $D = \begin{array}{c|ccc} & 0 & 7_3 & 5 \\ \hline & 3 & 0 & 8_1 \\ \hline & 5_2 & 2 & 0 \end{array}$ $\begin{array}{l} \text{path 1 to 2 through 3 } \min(8, 5 + 2) = 7 \text{ through 3} \\ \text{path 2 to 1 through 3 } \min(3, 8 + 5) = 3 \text{ no change} \end{array}$

■

Floyd-Warshall's algorithm

Transitive Closure and Shortest Paths

Problem 15 (Transitive Closure Problem).

Input: A graph $G = (V, E)$ and its adjacency matrix or list.

Output: A matrix T , the transitive closure matrix. A $T_{i,j} = 1$ indicates there is a path from i to j , and $T_{i,j} = 0$ otherwise.

Solutions.

Solution 1. Run n DFS operations in other words, run `search(G, i)` for all $i = 1, \dots, n$. Set $T_{i,j} = 1$ if `color[j]=B`, otherwise $T_{i,j} = 0$. One DFS is $\Theta(n + m)$ or $\Theta(n^2)$, and thus n DFS operations lead to an $\Theta(n(n + m))$ or $\Theta(n^3)$ algorithm.

Solution 2. Form a weight matrix W with $W_{i,j} = 1$ if edge $(i, j) \in E$ and 0 otherwise. Maintain ∞ in the diagonal entries. Run the all-pairs shortest path `FloydWarshall`. If for a non-diagonal entry of D^k we have $D^k(i, j) > 0$ it means there is a path from i to j of length that value. This is equivalent to solving the transitive closure problem. We have also solved the **strong connectivity** problem by extension.

Solution 3. Modify line 5 of Floyd Warshall. The MIN can be replaced by an OR and the + by an AND. Explain why this works as claimed.

For matrix multiplication in the remainder + is replaced with $\vee$ and $\cdot$ is replaced with $\wedge$. If graph G has adjacency matrix A , then A^k is the adjacency matrix of G^k , where matrix multiplication utilizes $\vee, \wedge$. We use an inductive proof. Base case $k = 1$, $G^1 = G$ and $A^1 = A$ is obviously true. if A^{k-1} is the adjacency matrix of G^{k-1} we shall show that if A^k is the adjacency matrix of G^k . Since $A^{k-1}ij = 1$ if and only if there is a tour in G of length $k - 1$ from i to j . We then show that $A^k_{ij} = 1$ if and only if there is a tour of length k . Suppose $A^k_{ij} = 1$. Since $A^k = A^{k-1}A$ we have

$$A^k_{ij} = A^{k-1}_{i1}A_{1j} + A^{k-1}_{i2}A_{2j} + A^{k-1}_{i3}A_{3j} + \dots A^{k-1}_{in}A_{nj}$$

Since $A^k_{ij} = 1$ there is one m such that $A^{k-1}_{im}A_{mj} = 1$. Then $A^{k-1}_{im} = 1$ and also $A_{mj} = 1$. Then there is a tour of length $k - 1$ from i to m and an edge from m to j i.e. a tour of length k from i to j .

If there is a tour of length k from i to j , let m be the last vertex before j . Then there is a tour of length $k - 1$ from i to m and an edge from m to j . Thus $A^k_{ij} = 1$.

Solution 4. Let A be the adjacency matrix with ones in diagonals. In other words A becomes $A + I$, where I is the unit matrix. For matrix multiplication in the remainder + is replaced with $\vee$ and $\cdot$ is replaced with $\wedge$. Then compute $(A \vee I)^{n-1}$. If $(A \vee I)^{n-1}_{ij} = 1$ then there is a path from i to j .

■

Floyd-Warshall's algorithm
Conclusion

Question 1. Is Floyd-Warshall faster than Solution 1 ? Explain.

Question 2. How can we keep track of the paths rather than just the lengths of these paths in Floyd-Warshall? Explain.

Question 3. Think of the MIN as a + and the + as a x (times). Can you relate Floyd-Warshall with matrix multiplication? What is A^n ?

Question 4. How fast can you compute A^n ?
(**Hint.** Think of Strassen and the exponentiation problem applied to matrices).

Single source shortest paths problems (SSSP)

Introduction

We examine now a simple version of the all-pair shortest path problem. It is the single source shortest path problem sometimes known as SSSP.

Problem 16 (Single source shortest path).

Input. Graph $G = (V, E)$ and its associated weight matrix or weight list W ; a source vertex $s \in V$.

Output. For every $u \in V$, $D[u]$ maintains the weight of the minimal weight path from s to u .

Solution. A greedy algorithm due to Dijkstra can be used to solve the single source shortest path problem. The algorithm is sensitive and to work correctly the graph must not have negative edge weights.

The algorithm works with graphs that have no negative weight edges. Thus the class of graphs for Dijkstra algorithm is more restrictive than that of Floyd-Warshall.

```
Dijkstra(G,W,s)
1. for every u in V
2.   D[u]= ∞ ; p[u] = NULL
3. S={ }; D[s]=0;
4. Q=V ; BuildMinHeap(H,[V,D]);
5. while Q is not empty do
6.   u= RemoveMin(H); //EXTRACT node u with min D(u)
7.   S=S U {u}; Q=Q-{u};
8.   for each v in A(u) AND v in Q do
9.     if (D[v] > D[u] + W[u][v])
10.      D[v] = D[u] + W[u][v];
11.      p[v] = u;
```

■

Greedy Algorithm for SSSP

Dijkstra's algorithm

Theorem 4 (Dijkstra). *Algorithm Dijkstra computes in D_u the weight of the minimal weight path from a given input source vertex s to u . The running time of Dijkstra for a heap-based implementation is $T(n, m) = O((n + m) \lg n)$, which for $m > n$ is $T(n, m) = O(m \lg n)$.*

Proof. We provide an analysis of the running time of Dijkstra's algorithm first. The analysis assume an adjacency list representation of the graph. Q is implemented as a priority queue that utilizes a MINHEAP with priorities the elements of $D[\cdot]$. The operation of line 10 decreases the element v of the heap with priority $D[v]$ to a lower value $D[u] + W[u][v]$. Such an operation in MINHEAP with n keys takes time $O(\lg n)$ to resolve

Lines 1-3 take $\Theta(n)$ time and so the Building of a MINHEAP in line 4. The while loop is performed n times total. For vertex u line 6 is $O(\lg n)$ by Subject 5. Line 7's $\Theta(1)$ is absorbed to that cost. Lines 8-10 are executed $\text{outdeg}(u)$ time with a cost $O(\lg n)$ as explained earlier and a total cost for one vertex u of $\text{outdeg}(u) \times O(\lg n)$. The total cost over all n iterations of Dijkstra is thus

$$\underbrace{\Theta(n)}_{\text{Lines 1-3}} + \underbrace{\sum_u O(\lg n)}_{\text{Lines 6-7}} + \underbrace{\sum_u \text{outdeg}(u) \times O(\lg n)}_{\text{Lines 8-9}} = O(n \lg n) + O(m \lg n) = O((n + m) \lg n).$$

Thus Dijkstra's running time for $m > n$ is $T(n, m) = O(m \lg n)$.

We continue on the following with a proof of the correctness of the algorithm.

Dijkstra's algorithm

Proof of correctness

Continued: proof of correctness. We provide a proof of correctness of Dijkstra's algorithm. It is by induction on the cardinality of S .

In Dijkstra's algorithm, initially $|S| = 0$ and $|Q| = n$, and during the course of the execution, the size of S increases by one in every iteration, and the size of Q decreases by one. Therefore, at the conclusion of the execution of Dijkstra's algorithm $|S| = n$ and $|Q| = 0$.

The following invariants will be maintained as part of the inductive hypothesis.

Invariant-1. For $u \in S$, $D[u]$ is the length of the shortest path from s to u .

Invariant-2. For $u \notin S$, $D[u]$ is the length of the shortest path from s to u with intermediate vertices in S .

Base case $|S| = 1$. In the first execution of line 6, s is extracted from Q and moved into S . $D[s]$ is 0 by default and given that there are no negative weights we have concluded the base case.

Moreover, Invariant-1 is true for s when it records $D[s] = 0$, the only member of S by default. For Invariant-2, we note that $D[\cdot] = W[s][\cdot]$ in line 9-11 is the length of the shortest edge path from s to an arbitrary $\cdot$ if such edge exists. Thus both invariants are true.

Inductive Step: from $|S| = k$ **to** $|S| = k + 1$. Let the inductive hypothesis apply to any set S of size k , i.e. $|S| = k$. We are going to prove that the hypothesis is true for a set S such that $|S| = k + 1$ (inductive step). Let u be the vertex outside S with MINIMUM $D[u]$. Thus any other $b \in Q$ is such that $D[b] > D[u]$. Then the shortest path from s to u (by rule 2 above) is internal in S except for a single edge that goes from a vertex of S directly to u and $u \in Q$. Let us call this path P . For otherwise, if P was not the shortest path, then there would be another path P' whose intermediate vertices were not internal in S and thus there would be another vertex say b in P' from s to u that is also in Q and that is closer to s than u , a contradiction to the minimality of $D[u]$. That is, if the shortest path would not be internal in S , there would be a path P' of the form $s \rightsquigarrow a$, $a \rightarrow b$ and $b \rightsquigarrow u$, where all vertices of the path from s to a are internal in S , b is the first node in Q , i.e. outside of S , and then the path from b to u might also include nodes internal or external to S . Then $\text{cost}(P') < \text{cost}(P)$ and therefore $\text{cost}(s \rightsquigarrow^{P'} b) + \text{cost}(b \rightsquigarrow^{P'} u) < \text{cost}(s \rightsquigarrow^P u)$. The first term is $D[b]$ and the last term is $D[u]$ thus $D[b] + \text{cost}(b \rightsquigarrow^{P'} u) < D[u]$. Given that all edges are positive this means $D[b] < D[u]$ which contradicts $D[b] > D[u]$ i.e. the minimality of $D[u]$. This satisfies Invariant-1 for u just inserted into S . Moreover, the line 9 update with respect to $D[u]$ guarantees the maintenance of Invariant-2, for vertices still in Q . ■

Dijkstra's algorithm

Proof of correctness: a concluding remark

Remark 1. Invariant-1 and Invariant-2 look only on paths where intermediate vertices are in S . They ignore all paths with intermediate vertices in Q . Why? Because of the greedy principle behinds Dijkstra. When Dijkstra pulls a u from Q into S the recorded $D[u]$ which by induction show the shortest path cost from s to u using intermediate vertices in S can not be beaten by an path that use an $a \in Q$. The reason is that u is pulled into S because its $D[u] < D[a]$. Any path utilizing a would be longer than $D[u]$ up to a , let alone the remainder from a to u . The latter is going to make the path through a even more expensive than $D[u]$ since the edges cannot be negative. The absence of negative edges is of paramount importance.

Proof of correctness by example G with source vertex $s=1$

2 Apply Dijkstra's method

```

1-----> 2      S = { }      Q={ 1 , 2 , 3 }
|              0      oo      oo
|
|      /
|      /
|      /
|      /
5 |      / -10
|      /
|      /
|      /
| /
v/
3

```

One can say Dijkstra's method made the wrong decision to add 2 to S and not 3. The edge path $1 \rightarrow 2$ is of cost 2, based on this decision.

Had Dijkstra's method picked 3 to add into S the path $1 \rightarrow 3 \rightarrow 2$ of cost -5 better than 2 would have been captured.

But Dijkstra never makes wrong decision because -10 never exist as weights to edges!

Dijkstra's algorithm

Example

Example 5. Run Dijkstra's algorithm on the graph whose weighed adjacency list is know below. (In a weighed adjacency list $A(i)$, in addition to j , such that $(i, j) \in E$, we also store $W_{i,j}$.)

Solution.

Graph $G=(V,E)$

```
1 ----> 2 [5] --> 3[9] --> 4[12] --> /.
2 ----> 3 [3] --> 4[6] --> /.
3 ----> 4 [2] ----> /.
4 ----> ./
```

$S=\{ \}$ $Q=\{ \begin{array}{c} \bar{1} \\ 0 \end{array}, \begin{array}{c} \bar{2} \\ \infty \end{array}, \begin{array}{c} \bar{3} \\ \infty \end{array}, \begin{array}{c} \bar{4} \\ \infty \end{array} \}$

$S=\{ \begin{array}{c} \bar{1} \\ 0 \end{array} \}$ $Q=\{ \begin{array}{c} 1 \\ 5 \end{array}, \begin{array}{c} 1 \\ 9 \end{array}, \begin{array}{c} 1 \\ 12 \end{array} \}$

$S=\{ \begin{array}{c} \bar{1} \\ 0 \end{array}, \begin{array}{c} 1 \\ 5 \end{array} \}$ $Q=\{ \begin{array}{c} 2 \\ 8 \end{array}, \begin{array}{c} 2 \\ 11 \end{array} \}$

$S=\{ \begin{array}{c} \bar{1} \\ 0 \end{array}, \begin{array}{c} 1 \\ 5 \end{array}, \begin{array}{c} 2 \\ 8 \end{array} \}$ $Q=\{ \begin{array}{c} 3 \\ 10 \end{array} \}$

$S=\{ \begin{array}{c} \bar{1} \\ 0 \end{array}, \begin{array}{c} 1 \\ 5 \end{array}, \begin{array}{c} 2 \\ 8 \end{array}, \begin{array}{c} 3 \\ 10 \end{array} \}$ $Q=\{ \}$

Path $\begin{array}{c} 3 \\ 4 \end{array} <--- \begin{array}{c} 2 \\ 3 \end{array} <--- \begin{array}{c} 1 \\ 2 \end{array} <--- \begin{array}{c} 1 \\ 0 \end{array} \quad \text{====>} \quad \begin{array}{c} \bar{10-8} \\ 4 \end{array} <--- \begin{array}{c} 8-5 \\ 3 \end{array} <--- \begin{array}{c} 5-0 \\ 1 \end{array}$

■

Spanning trees of undirected graphs

Introduction

The discussion on minimal cost spanning trees (MCST) is for **undirected graphs**. Recalling our earlier discussion, a tree is a connected acyclic undirected graph.

Fact 1. A tree with n vertices has $n - 1$ edges.

Therefore Depth First Search on a tree would only take $\Theta(n)$ time.

Definition 1 (Spanning tree of undirected graph G). A **spanning tree** of an undirected connected graph G is a tree that saturates every vertex of the graph and also has $n - 1$ edges.

Definition 2 (Minimal cost spanning tree (MCST)). If graph edges have weights (also known as distances, costs), a **minimal cost spanning tree** (MCST) is a spanning tree for which the sum of the weights of the edges of the tree is minimal.

1. Algorithmic design principle to be used. For this problem, the solution is going to be through a greedy algorithm. We present below two algorithms for finding minimal cost spanning trees: one due to Kruskal and one due to Prim. Both algorithms are greedy algorithms.

2. Greedy principle of a MCST algorithm: Pick the edge with the lowest cost to attempt to add it to a tree that is being formed (and thus currently has fewer than $n - 1$ edges, if its number of vertices is n) and do add if the addition does not cause the tree to have a cycle.

Spanning trees of undirected graphs

Kruskal's method

Definition 3 (Minimal Cost Spanning Tree: Problem Definition).

Input. Undirected graph $G = (V, E)$ and its associated weight matrix/list W .

Output. A spanning tree $T = (V, E')$, where $E' \subseteq E$, of minimal total weight cost, i.e. a minimal cost spanning tree T of G .

Proposition 1 (Kruskal's method). *Kruskal's method works as claimed.*

```
Kruskal(G,W) // Graph with edge weights
// Input  $G=(V,E)$  and  $W$ 
// Output minimal spanning tree  $T=(V, E(T))$ ,  $|E(T)| = n-1$ 
1.  $T = \{ \}$ ;
2. while ( $T$  has fewer than  $n-1$  edges)
3.   add to  $T$  the SHORTEST edge that does not make  $T$  have a cycle
```

MCST Kruskal's method

Proof. (By contradiction through the following Lemma.)

Lemma 2. *Let $G = (V, E)$ be a connected graph, and $T = (V, E(T))$ be a spanning tree of G . Then*

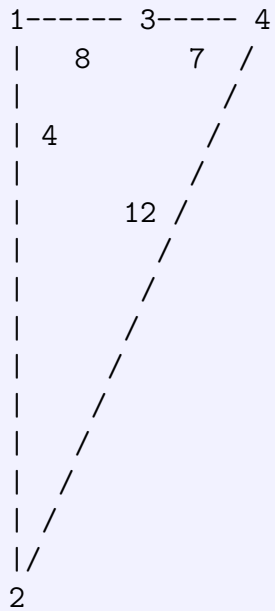
- (1) For all u, v the path from u to v is unique.*
- (2) If an edge $E - E(T)$ is added to T it causes T have a (unique) cycle.*
- (3) If an edge $E - E(T)$ is added to T it causes T have a (unique) cycle; furthermore if another edge gets deleted from the resulting cycle, a new spanning tree is obtained.*

The tree obtained by Algorithm Kruskal is a spanning tree as long as G is a connected graph. Let $T = \{e_1, \dots, e_{n-1}\}$ be the tree generated by Kruskal's algorithm. If G is connected there is no isolated vertex in G . Let T be not a minimal spanning tree. Let T' be the minimal spanning tree of G with the maximal number of edges (starting with e_1) common with T , and let this number be $k - 1$. Then $e_k = (a, b)$ is the first edge in T that is not in T' . It can't be the other way i.e. $e_k \notin T$ and $e_k \in T'$ because this would imply e_k caused a cycle in Kruskal (and thus was not picked). Then it would also cause a cycle in T' as up to that point both of the algorithms have common edges! Let P be the path in T' that connects a, b since $e_k = (a, b)$ is not in T' . (The addition of e_k would cause P plus e_k to become a cycle.) There must be an edge $w(e') > w(e_k)$ in P . If there was no such edge, Kruskal would have picked all those edges for T (they are lighter than e_k) and by including afterwards the heavier edge e_k would have generated a cycle: a contradiction. Thus an $e' > e_k$ exists. Consider then $T' - e' + e_k$. It is a spanning tree better than T' , a contradiction. (Moreover it has more edges in common with T a contradiction to the maximality of T' .) (Alternatively, let G_k be the connected component containing $\{e_1, \dots, e_{k-1}\}$ that touches a . Then there exists an edge e in P connecting G_k to a vertex outside G_k . Since Kruskal chooses e_k over e in T , $e_k \leq e$. But $T' - e + e_k$ is a spanning tree at least as cheap as T' , but with one more edge in common with T than T' a contradiction to the choice of T' .) ■

Kruskal's method

Example

Example 6 (Kruskal's method: example). An example that highlights the operations in Kruskal's algorithm is shown below.



```

Step 0:          T:    1      3      4
                  2
-----
Step 1: Pick (1,2) T:    1      3      4
           cost is 4      |
T={(1,2)}      no cycle  2
-----
Step 2: Pick (3,4) T:    1      3-----4
           cost is 7      |
T={(1,2),(3,4)} no cycle  2
-----
Step 3: Pick (1,3) T:    1-----3-----4
           cost is 8      |
T={(1,2),(3,4),(1,3)}    2
no cycle T has n-1 edges STOP

```

Kruskal's method Algorithm using UF-DS

Proposition 3 (Kruskal's algorithm implementation using UF-DS). *Let $G = (V, E)$ with $|V| = n$ and $|E| = m$ and W be the weight matrix or list associated with G . An implementation of Kruskal's algorithm using a Union-Find data structure is shown below. Its running time is $T(n, m) = O(m \lg n + n \lg n)$ and for $m > n$ it simplifies to $T(n, m) = O(m \lg n)$.*

```
Kruskal(G,W)  // G is connected
1. Order the edges so that 'pick SHORTEST edge' is possible efficiently
2. T=empty;
3. for each u in V
4.   do MakeSet(u);
5. while there are still edges to be picked {
6.   Pick SHORTEST edge (u,v) available (that has not already been picked)
7.   if Find(u) != Find(v)
8.     T=T U {(u,v)} ;
9.     Union(u,v);
10.}
```

Proof. (Analysis of the algorithm's running time.)

Running Time (Union-Find operations). Lines 3-4 perform n MakeSet operation which collectively cost $\Theta(n)$ (Subject 3). The number of Find operations is $2m$ since we can have m edges and each edge has two end-points. Cost is $O(m)$. (It is not clear for a given G if we exhaust all the edges or complete the tree T before hand.) Moreover the number of Union operations is no more than $n - 1$. Thus line 10. will be executed no more than $n - 1$ times and thus collectively all those operations have a cost $O(n \lg n)$ (per Subject 3 implementation of Union-Find). Therefore Union-Find related time is $O(m + n \lg n)$

Running Time (Edge ordering). We can sort the edges in $\Theta(m \lg m)$ time in Line 1. Then Line 6 is a constant step per edge, and thus the total execution time of Line 6 overall edges is a $O(m)$. Combining the two we get $\Theta(m \lg m)$. Alternatively, we can build a MINHEAP in $\Theta(m)$ time in Line 1. Then Line 6 involves an ExtractMIN operation in $O(\lg m)$ time, and thus the total execution time of Line 6 overall edges is a $O(m \lg m)$. Combining the two (Line 1 and Line 6 costs) we get $O(m \lg m)$.

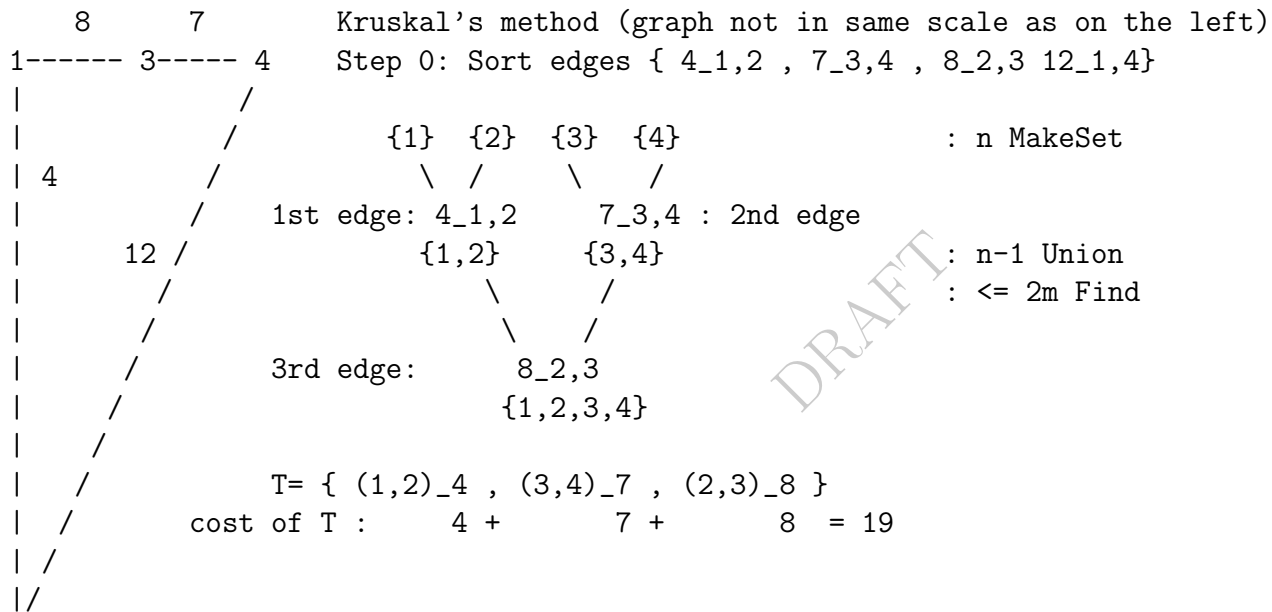
Therefore running time of Kruskal using Union-Find is $T(n, m) = O(n \lg n + m \lg m)$.

In most cases $m > n$, and because the graph is connected $m \geq n - 1$. Then it simplifies to $T(n, m) = O(m \lg m)$. ■

Kruskal's Algorithm using UF-DS

Example

Example 7 (Kruskal example using UF-DS). We show below an example when a UF-DS is being used in the implementation of Kruskal's algorithm.



Proposition 4 (Prim's method). *Prim's method works as claimed.*

```
Prim(G, W)
// Input G=(V, E) and W
// Output minimal spanning tree T=(V, E(T)), |E(T)| = n-1
1. T = { } ; X = {u}; Q = V - X; // u is an arbitrary vertex of V;
2. while ( X != V ) {
3.   let e=(x, q) be SHORTEST edge from an x in X to a q in Q
4.   T = T U {(x, q)};
4.   X = X U {x}; Q = Q - {q};
5. }
```

Proof. (By contradiction.) We are going to use the following Lemma.

Lemma 5. *Let $G = (V, E)$ be a connected graph, and $T = (V, E(T))$ be a spanning tree of G . Then*

- (1) *For all u, v the path from u to v is unique.*
- (2) *If an edge $E - E(T)$ is added to T it causes T have a (unique) cycle.*
- (3) *If an edge $E - E(T)$ is added to T it causes T have a (unique) cycle; furthermore if another edge gets deleted from the resulting cycle, a new spanning tree is obtained.*

Prim's method proof of correctness

Continued: proof of correctness. Let $X \subset V$. If (u, v) is the lowest cost edge from X to $Q = V - X$, then there is a minimum cost spanning tree that includes (u, v) .

The tree obtained by Method Prim is a spanning tree as long as G is a connected graph. Let $T = \{e_1, \dots, e_{n-1}\}$ be the tree generated by Prim's algorithm. and let $T_k = \{e_1, \dots, e_k\}$ be the Prim tree after k iterations. We claim that T is a minimal spanning tree.

Let us assume that T is not a minimal spanning tree, and let T' be a minimal spanning tree that has as the maximum number of edges in common with T .

We show by contradiction that $T' = T$ or we have a tree whose cost is smaller than the minimal spanning tree T' .

Let T' has $k - 1$ edges in common with T and $k - 1$ is thus the maximum number of edges a minimal spanning tree has in common with T .

The first edge T' and T differ is edge $e_k = (a, b)$. Let $S(T_{k-1})$ be the set of vertices saturated by the first $k - 1$ edges common to T' and T . Let us assume $a \in S(T_{k-1})$.

Since T' is a minimal spanning tree there is a path P in it from a to b . We have $a \in S(T_{k-1})$ and $b \notin S(T_{k-1})$. Thus the path P has an edge (u, v) , where $u \in S(T_{k-1})$ and $v \notin S(T_{k-1})$.

Consider edge (u, v) versus edge (a, b) . Then form $T_1 = T' - (u, v) + (a, b)$. T_1 is a spanning tree of G .

We have the following possibilities for T_1 .

Case 1: $w(a, b) < w(u, v)$. Prim chose (a, b) because it is of the lowest cost. The end results is that T_1 has cost $c(T_1) < c(T')$. We have found a spanning tree T_1 of cost smallest than the minimal cost spanning tree. Contradiction.

Case 2: $w(a, b) > w(u, v)$. Prim chose (a, b) instead of (u, v) . Impossible because Prim always picks the smallest edge and should have picked (u, v) . Contradiction to Prim's algorithmic choices.

Case 3: $w(a, b) = w(u, v)$. The two edges have the same cost and thus T_1 is such that $c(T_1) = c(T)$ and thus T_1 is a minimal spanning tree. It is possible that Prim picked one over the other of equal cost edges. Since Prim did not pick at that stage (u, v) it means that (u, v) is not one of the edges of T_{k-1} . But now we have found minimal spanning tree T_1 that has k edges in common with T versus T' 's $k - 1$ edges. This is a contradiction to the "maximality" of T' .

All three cases lead to contradiction because we assumed that there was something better than Prim. Prim is optimal. ■

Prim's method Example

Example 8 (Prim's method: example). An example that highlights the operations in Prim's algorithm is shown below.

1----- 3----- 4	Step 0: $X=\{4\}$	$Q=\{1,2,3\}$	1	3	4	l.c. means
8 7 /	$T=\{\}$					lowest
4 /	(3,4) l.c. from X to Q	2				cost
12 /	-----					
/	Step 1: $X=\{4,3\}$	$Q=\{1,2\}$	1	3----	4	
/	$T=\{(3,4)\}$					
/	(1,3) l.c. from X to Q	2				
/	-----					
/	Step 2: $X=\{4,3,1\}$	$Q=\{2\}$	1-----	3-----	4	
/	$T=\{(3,4), (1,3)\}$					
/	(1,2) l.c. form X to Q	2				
/	-----					
/	Step 3: $X=\{4,3,1,2\}$	$Q=\{\}$	1-----	3-----	4	
/	$T=\{(3,4), (1,3), (1,2)\}$					
/	T has $n-1=3$ edges	2				
2	STOP					

Prim's method Algorithm Using a Priority Queue

Proposition 6 (Prim's algorithm implementation using a Priority Queue). *Let $G = (V, E)$ with $|V| = n$ and $|E| = m$ and W be the weight matrix or list associated with G . An implementation of Prim's algorithm using a priority queue is shown below. Its running time is $T(n, m) = O(m \lg n + n \lg n)$ and for $m > n$ it simplifies to $T(n, m) = O(m \lg n)$.*

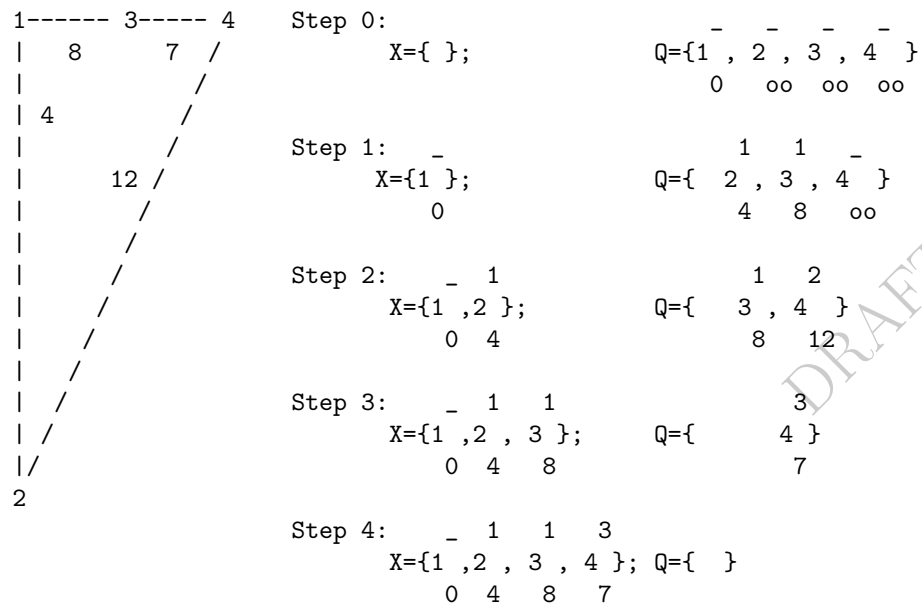
```
Prim    (G,W)    // G is a connected graph
1.  for every u in V
2.    cost[u]= oo ; p[u] = NULL
3.  cost[1]= 0;
4.  Q=V;X=empty; BuildMinHeap(H,[V,cost]); //Priorities in heap H are the cost[.]
5.  while Q is not empty do
6.    u= ExtractMin(H);
7.    X=X U {u}; Q=Q-{u};
8.    for each v in A(u) i.e. for each edge (u,v) and v in Q do
9.      if W[u][v] < cost[v];
10.        cost[v]= W[u][v]
11.        p[v] = u;
```

Proof. (Analysis of the algorithm's running time.) Prim's algorithm uses the structure of Dijkstra's algorithm. Thus the analysis of the running time is identical to that of Dijkstra's. Note that the test in line 8 of membership in Q is implemented by having one extra bit per vertex. This bit is set to 1 for a vertex in Q . For a removed vertex it is set to 0. Thus the use of X is redundant.

Prim's running time for $m > n$ is $T(n, m) = O(m \lg n)$. ■

Prim's algorithm Example

Example 9 (Prim example using a priority queue). We show below an example when a priority queue is being used in the implementation of Prim's algorithm.



T is { (1,2), (1,3), (3,4) }

Spanning tree T cost is 4 + 8 + 7 = 19

Prim's algorithm

Example restarted

Example 10 (Prim example using a priority queue). We show below an example when a priority queue is being used in the implementation of Prim's algorithm. We start now from 4

Diagram illustrating the steps of Prim's algorithm using a priority queue. The graph has 4 nodes (1, 2, 3, 4) and edges with weights: (1,2)=4, (1,3)=8, (1,4)=7, (2,3)=12, (3,4)=3.

Step 0:

```

X={ };
Q={1, 2, 3, 4 }
    oo oo oo 0

```

Step 1:

```

X={4 };
Q={1, 2, 3 }
    oo 12 7

```

Step 2:

```

X={4, 3 };
Q={1 2 }
    0 7   8 12

```

Step 3:

```

X={4, 3, 1 };
Q={ 2 }
    0 7 8   4

```

Step 4:

```

X={1, 2, 3, 4 }; Q={ }
    8 4 7 0

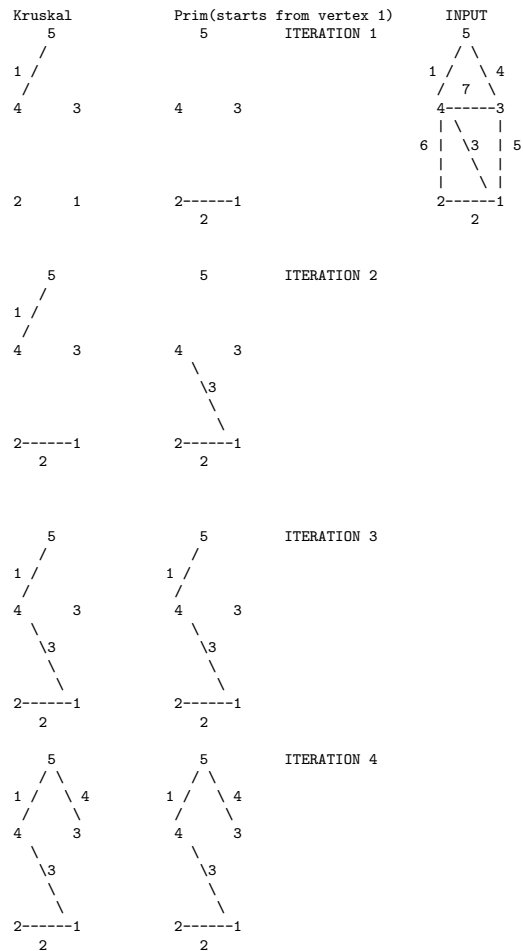
```

T is { (1,2), (1,3), (3,4) }

Spanning tree T cost is $4 + 8 + 7 = 19$

Prim's algorithm

Example restarted



DRAFT

Notice

Problem Set with Solutions

These Problem Sets are not for credit. They are provided as are for your practice. Several problems are provided with solutions. These can be discussed with the instructor after a student has studied the relevant material. Some other problems are provided with either no solutions at all or a hint of a solution. They are there for your reference and will not be discussed further. For some solutions provided you might have to fill in details related to calculations or intermediate steps.

Before you try these problems: (a) Study the designated textbook, (b) use the Subject notes (typewritten notes available through the course web-page from the password-protected section C) to review your study of the designated textbook material, (c) resolve the examples done in class, (d) then practice with some of these problems PS (Problem Set), and then if you have time, (e) solve some of the problems of the designated textbook, and (f) solve more problems from the PS especially the ones with minimal solution information.

Several PS problems appeared in past exams or homeworks.

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 1.

TRUE or FALSE? Anything other than a TRUE or FALSE is considered wrong!

1. It is always asymptotically faster to compute the $\text{outdeg}(u)$ using an adjacency matrix than an adjacency list representation.
2. It is always asymptotically faster to compute the $\text{indeg}(u)$ using an adjacency matrix than an adjacency list representation.
3. It is always asymptotically faster to compute m using an adjacency matrix than an adjacency list representation.
4. It is always asymptotically faster to generate the adjacency list of u using an adjacency matrix than an adjacency list representation.
5. Let $G = (V, E)$ be a simple undirected graph with weights on its edges and with more than three vertices and unique edge weights. Let M be a minimum cost spanning tree of G . Then the path from u to v in M is a shortest path from u to v in G as well.
6. Let $G = (V, E)$ be a simple undirected graph with weights on its edges and with more than three vertices and unique edge weight. Then the heaviest edge in G is in NO minimum-cost spanning tree.

Proof. 1. FALSE, 2. FALSE, 3. FALSE, 4. FALSE, 5. FALSE, 6. FALSE. ■

Problem Set Problem 2.

The maximum-spanning tree problem is defined similarly to the minimum-spanning tree problem except that we want to maximize the sum of the weights of the edges of the tree. Give an algorithm that finds the maximum-spanning tree of an undirected graph $G = (V, E, W)$ where edges have weights represented in the weight matrix W . What is the running time of the algorithm?

Proof. If the graph is $G = (V, E, W)$ consider graph $G' = (V, E, -W)$ i.e. we negate the weights of all the edges in G' . A maximum spanning tree T in G is a minimum spanning tree in G' . Note that the conversion of W into $-W$ takes time $O(m)$. If G is represented by an adjacency-matrix we find all the edges and negate the corresponding weights in $O(n^2)$ time. If G is represented by an adjacency-list we go through all the edges in time $O(m)$ and negate the corresponding weights.

Whatever the cost is $O(m \lg m)$, the running time of Kruskal's algorithm will contribute more.

The maximum-spanning tree problem is defined similarly to the minimum-spanning tree problem. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's workspace

Problem Set
Problem 3.

You are given a directed graph with positive weights on its edges. Show how you can solve the minimum cost directed cycle problem in that graph in time faster than $o(n^4)$ (this is a little-oh) in the worst-case. A directed cycle is a directed path that starts and ends in the same vertex and has at least one edge. A minimum cost directed cycle is one for which the sum of the weights of the edges of the cycle is minimal.

Proof. Set $a_{ii} = \infty$. Run Floyd-Warshall. Then a_{ii}^n will give the length of the smallest cycle from i to i . Find the minimum among all a_{ii}^n for all i . ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 4.

You are given an acyclic connected and undirected graph $G = (V, E)$. Can you determine in $O(n)$ time whether it has or not a cycle (note that the bound involves only n but not m)?

Proof. Yes! G is a tree and thus $m = n - 1$. Therefore depth-first-search solves this problem in $O(n + m)$ time which is $O(n)$ as well. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem 5.

You are given an undirected graph $G = (V, E)$. Can you determine in $O(n)$ time whether it has or not a cycle (note that the bound involves only n but not m)? Assume an adjacency list representation of the graph.

Proof. Yes! Run depth first search by also using two counters for each connected component discovered during the top-level call. One counts the vertices discovered and the other the edges discovered. If edges discovered is equal or more than vertices discovered. we have a cycle by the simple observation that in a tree edges are one less than vertices, and if in a connected component searched through depth-first-search we have discovered a number of edges equal to or more than the number of vertices, a cycle should exist there.

Note that in this algorithm the running time is $\sum_i O(n_i + m_i)$, where n_i is the number of vertices in the i -th component searched and m_i the number of edges searched in that component. Naturally $\sum_i n_i = n$. Given that the search process stops as soon as $m_i \geq n_i$, we have $m_i \leq n_i$. Therefore $\sum_i m_i = O(n)$. Combining the two we obtain that $\sum_i O(n_i + m_i) = O(n)$ as well. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web page

Problem Set
Problem 6.

Give an algorithm that detects a negative cost cycle.

Proof. Run Floyd-Warshall. At every iteration inspect the diagonal elements. Initially, they are all zero. If a value along the diagonal changes it can only grow smaller, i.e. negative. A negative $a_{ii}^k < 0$ show the existence of a cycle with end-points i whose all intermediate vertices are k or less. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved
Not to be displayed outside of copyright holder's web-page

Problem Set Problem 7.

We have an undirected weighed graph whose edges have weights that are only 10 or 20. Can you find the minimum spanning tree of this graph in $O(n + m)$ time? Explain.

Proof. Yes! We only need to maintain a heap whose operations are $O(1)$ rather than the traditional $O(\lg n)$. Note that in Prim's algorithm the key in a heap is the edge weight. We can only have two values 10 or 20. Therefore we can implement a heap with an array of size two that stores two (doubly) linked lists. We attach to element 0 a linked list of all edges of weight 10, and to element 1 a linked list of all edges of weight 20. We only insert or delete from the head of the list to maintain $O(1)$ time operations.

Insert inserts the weight of edge (u, v) into the Heap as the weight of v . We insert either to the linked list attached to element 0 or to the one attached to element 1. Either time is constant. At the same time we point $point[v]$ to this location (we can also choose to record in (u, v) information that this edge is related to v).

ExtractMin takes $O(1)$ time. Check list 0 first. If list 0 is not empty, extract its element pointed by head. If list 0 is empty go to list 1 and extract an element through the head pointer. If both lists 0 and 1 are empty return "underflow".

Decrease takes constant time because we use doubly linked lists. We might move an element (edge) from one list to the other. Note that in accordance to Prim's algorithm for every vertex in an array of size n we store a pointer $point[v]$ that points to the position of the vertex in the linked list/heap. Therefore DecreaseMin(v) (lines 10-12), first locates v in the corresponding linked list using $point[v]$ (and this should be the list attached to element 1) and if the value needs to be decreased (i.e. from 20 to 10) then the element pointed to by $point[v]$ is deleted in constant time, a new element is inserted in the linked list of element 0 and $point[v]$ points to this new location (and also the new location points to v). ■

Problem Set
Problem 8.

You are given a graph $G = (V, E)$. We want to determine $\forall u, v$ whether there exists a path from u to v . Give three different algorithms that solve this problem and their worst case running time in terms of n and m .

Proof. Improvise. No solutions for this problem.



Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 9.

A graph $G=(V,E)$ is given; both the adjacency list and adjacency matrix are provided. Find a cycle of length n that contains all vertices in ... jokingly ... polynomial time.

Proof. Kidding. This is the Hamiltonian cycle problem. The decision version is NP-complete. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved
Not to be displayed outside of copyright holder's web-page