ECE 2400 Computer Systems Programming
Spring 2026

Topic 16: Graphs

School of Electrical and Computer Engineering
Cornell University

revision: 2026-04-29-11-52

1 Graph Concepts 3

2 Graph Storage 4

3 Directed Graphs 5

4 Finding a Path Between Two Vertices 10

4.1. Depth-First Search 12

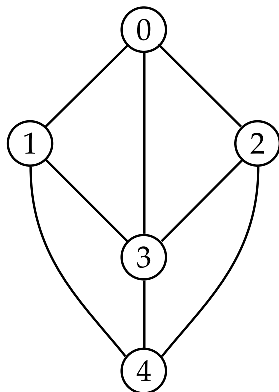
4.2. Breadth-First Search 14

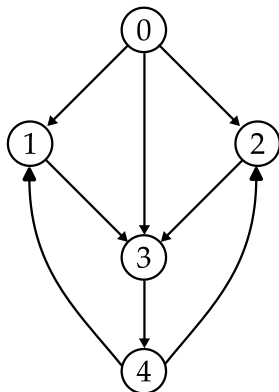
5 Other Common Search Algorithms 16

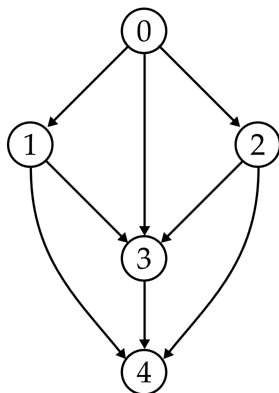
6 Constructing a Minimum Spanning Tree 16

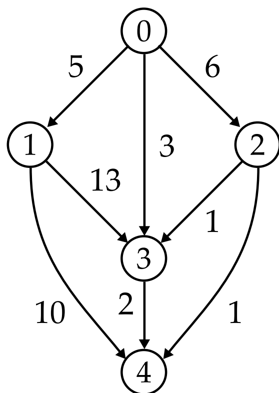
7 Interplay between Algorithms and Data Structures 17

1. Graph Concepts

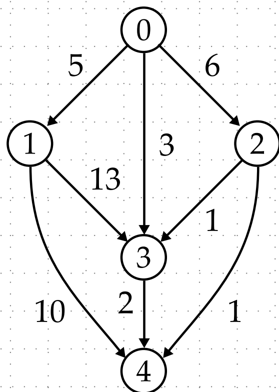








2. Graph Storage



Adjacency Matrix

(rows: dest; columns: src; "weight" marks the entry 1)

dest \ src	0	1	2	3	4
0					
1	5				
2	6				
3	3	13	1		
4		10	7	2	

Adjacency List (with inner `Vector`)

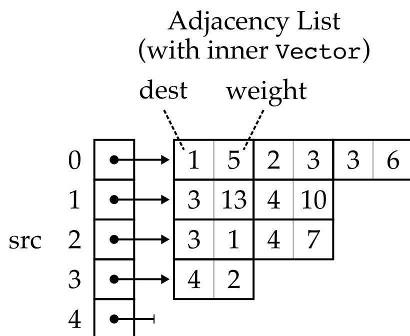
(each row: src; pairs of dest, weight)

src						
0	1	5	2	3	3	6
1	3	13	4	10		
2	3	1	4	7		
3	4	2				
4						

3. Directed Graphs

- Focus on object-oriented adjacency-list-based directed graphs storing `int` weights
  - Could apply same approach to undirected graphs
  - Could use object-oriented programming and dynamic polymorphism
  - Could use generic programming and static polymorphism
  - Later: can use concurrent programming to analyze graph in parallel

```
class GraphInt
{
 public:

  int          add_vertex();
  void         add_edge( int src_id, int dest_id, int w );
  Vector<int> get_neighbors( int id );
  int          get_weight( int src_id, int dest_id );

 private:
  Vector< Vector< Pair<int,int> > > m_graph;
};
```



```
int GraphInt::add_vertex()
{
  m_graph.push_back( Vector<Pair<int,int>>() );
  return m_graph.size() - 1;
}

void GraphInt::add_edge( int src_id, int dest_id, int w )
{
  m_graph.at(src_id).push_back(
    Pair<int,int>( dest_id, w ) );
}

Vector<int> GraphInt::get_neighbors( int id )
{
  Vector<int> neighbors;
  for ( auto e : m_graph.at(id) )
    neighbors.push_back( e.first );
  return neighbors;
}

int GraphInt::get_weight( int src_id, int dest_id )
{
  for ( auto e : m_graph.at(src_id) )
    if ( e.first == dest_id )
      return e.second;
  assert(false);
}
```

Draw the conceptual graph and the adjacency list storage resulting from this code sequence:

```
1  GraphInt g;
2
3  int v0 = g.add_vertex();
4  int v1 = g.add_vertex();
5  int v2 = g.add_vertex();
6  int v3 = g.add_vertex();
7  int v4 = g.add_vertex();
8  int v5 = g.add_vertex();
9  int v6 = g.add_vertex();
10
11 g.add_edge( v0, v1, 1 );
12 g.add_edge( v0, v2, 1 );
13 g.add_edge( v0, v3, 1 );
14 g.add_edge( v1, v6, 1 );
15 g.add_edge( v2, v4, 1 );
16 g.add_edge( v3, v5, 1 );
17 g.add_edge( v4, v6, 1 );
18 g.add_edge( v5, v4, 1 );
```

Time and space complexity analysis for different storage

- Let a graph G be a pair (V, E)
  - V is a set of vertices, $|V|$ is the number of vertices
  - E is a set of edges, $|E|$ is the number of edges
  - we often informally just use V and E to represent $|V|$ and $|E|$

	Adjacency Matrix	**Adjacency List:** Inner data structure is ... `Vector`	`BST`	`HashTable`
Space Usage				
`add_vertex`				
`add_edge`				
`get_neighbors`				
`get_weight`				

Adjacency Matrix

src

dest		0	1	2	3	4
	0					
	1	5				
dest	2	6				
	3	3	13	1		
	4		10	7	2	

(weight)

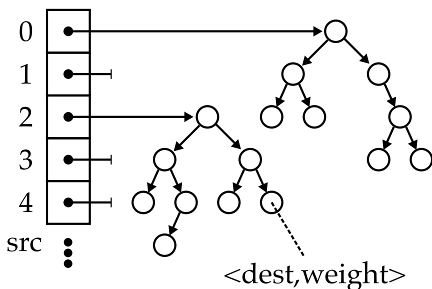
Adjacency List (with inner `Vector`)

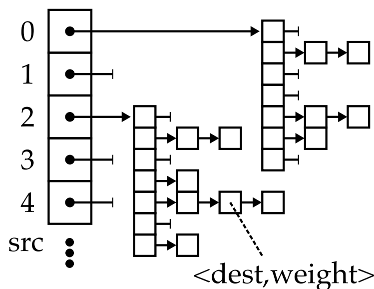
src	dest, weight pairs
0	1 5, 2 3, 3 6
1	3 13, 4 10
2	3 1, 4 7
3	4 2
4	(null)

(dest, weight)

- Can we use an alternative inner data structure to improve the performance of getting the weight for a given edge?
  - `InnerDataStruct<K,V>` is a map implemented with BST or hash table

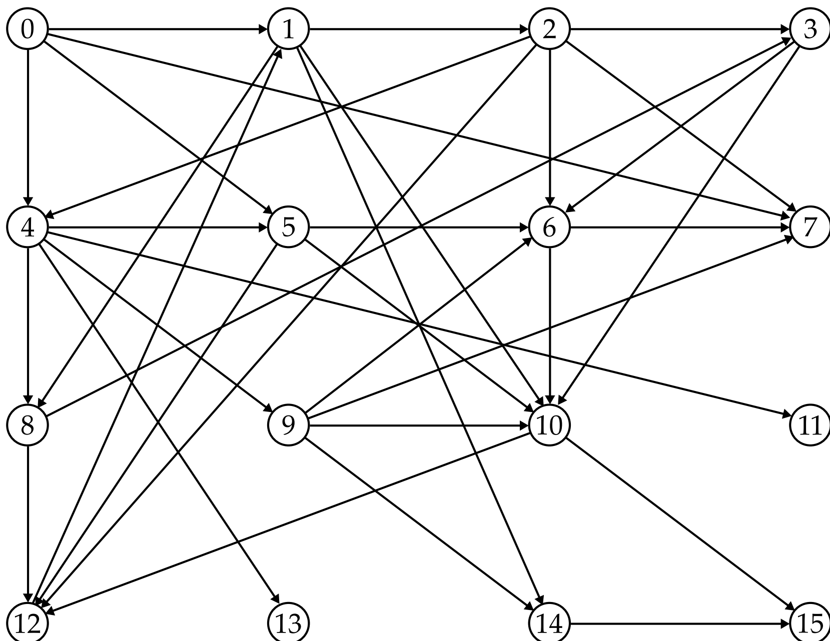
```
int GraphInt::add_vertex() {
  m_graph.push_back( InnerDataStruct<int,int>() );
  return m_graph.size() - 1;
}

void GraphInt::add_edge( int src_id, int dest_id, int w ) {
  m_graph.at(src_id).add( dest_id, w ) );
}

Vector<int> GraphInt::get_neighbors( int id ) {
  Vector<int> neighbors;
  for ( auto n : m_graph.at(id) )
    neighbors.push_back( n.first );
  return neighbors;
}

int GraphInt::get_weight( int src_id, int dest_id ) {
  return m_graph.at(src_id).lookup(dest_id);
}
```

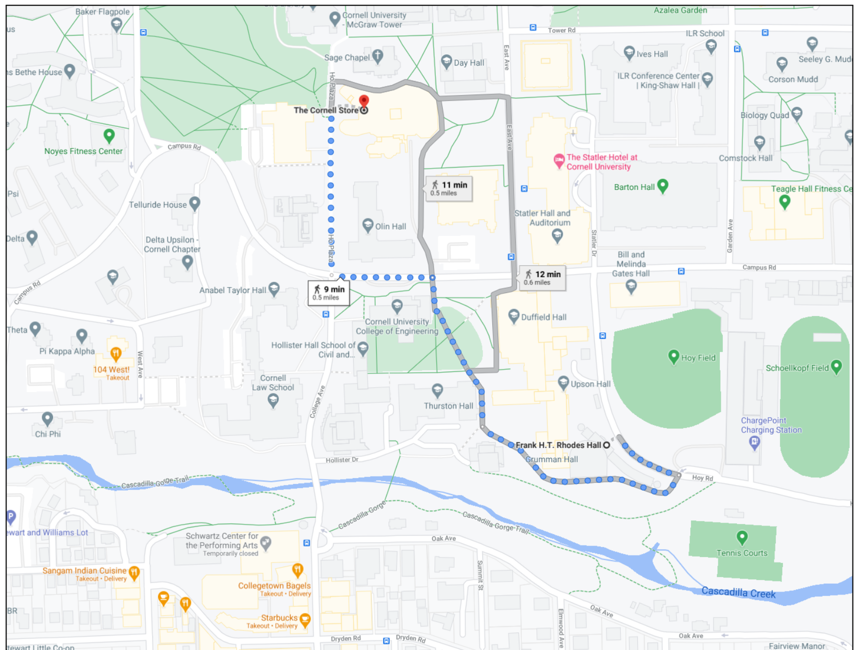




4. Finding a Path Between Two Vertices

- Given
  - graph $G = (V, E)$
  - source vertex V_s
  - destination vertex V_d
- Find a path from V_s to V_d



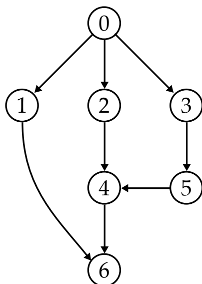


- We will explore three different algorithms:
  - Depth-First Search: finds a path if it exists
  - Breadth-First Search: finds a path if it exists

```
1  class GraphInt
2  {
3   public:
4     ...
5     Vector<int> dfs      ( int src_id, int dest_id );
6     Vector<int> bfs      ( int src_id, int dest_id );
7  };
```

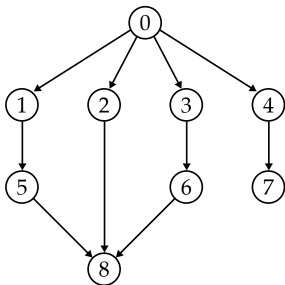
4.1. Depth-First Search

```
1 def GraphInt::dfs( src_id, dest_id ):
2   set visited  to be a set    # vertices already visited
3   set worklist to be a stack  # pending paths to search
4
5   worklist.push( [src_id] )
6   while worklist is not empty:
7     path = worklist.pop()
8     set v to be final vertex in path
9
10    if v == dest_id:
11      return path
12
13    if v not in visited:
14      visited.add( v )
15      for n in get_neighbors( v ):
16        worklist.push( path + n )
```



visited:

worklist:



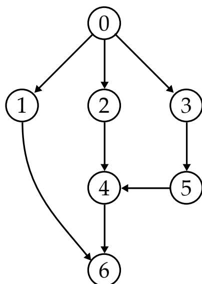
visited:

worklist:

```
1   Vector<int> GraphInt::dfs( int src_id, int dest_id )
2   {
3     Set<int>            visited;  // vertices already visited
4     Stack<Vector<int>> worklist; // pending paths to search
5
6     // Initialize worklist w/ path containing just source vertex
7     Vector<int> p; p.push_back(src_id); worklist.push( p );
8
9     // Keep working until worklist is empty
10    while ( worklist.size() != 0 ) {
11
12      // Pop path from _top_ of stack
13      auto path = worklist.pop();
14
15      // Check if final vertex in current path is destination
16      int v = path.at( path.size()-1 );
17      if ( v == dest_id ) return path;
18
19      // Check if final vertex has already been visited
20      if ( !visited.contains( v ) ) {
21
22        // Mark final vertex as visited
23        visited.add( v );
24
25        // Iterate through neighbors
26        auto neighbors = get_neighbors( v );
27        for ( int n : neighbors ) {
28
29          // Create temporary new path with neighbor at end
30          auto temp = path;
31          temp.push_back(n);
32
33          // Push this new path onto _top_ of stack
34          worklist.push( temp );
35        }
36      }
37    }
38  }
```

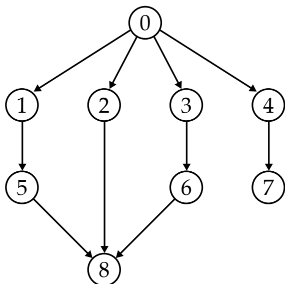
4.2. Breadth-First Search

```
1 def GraphInt::bfs( src_id, dest_id ):
2   set visited  to be a set    # vertices already visited
3   set worklist to be a queue  # pending paths to search
4
5   worklist.enq( [src_id] )
6   while worklist is not empty:
7     path = worklist.deq()
8     set v to be final vertex in path
9
10    if v == dest_id:
11      return path
12
13    if v not in visited:
14      visited.add( v )
15      for n in get_neighbors( v ):
16        worklist.enq( path + n )
```



visited:

worklist:



visited:

worklist:

```
1  Vector<int> GraphInt::bfs( int src_id, int dest_id )
2  {
3    Set<int>             visited;  // vertices already visited
4    Queue<Vector<int>> worklist; // pending paths to search
5
6    // Initialize worklist w/ path containing just source vertex
7    Vector<int> p; p.push_back(src_id); worklist.enq( p );
8
9    // Keep working until worklist is empty
10   while ( worklist.size() != 0 ) {
11
12     // Dequeue path from _head_ of queue
13     auto path = worklist.deq();
14
15     // Check if final vertex in current path is destination
16     int v = path.at( path.size()-1 );
17     if ( v == dest_id ) return path;
18
19     // Check if final vertex has already been visited
20     if ( !visited.contains( v ) ) {
21
22       // Mark vertex as visited
23       visited.add( v );
24
25       // Iterate through neighbors
26       auto neighbors = get_neighbors( v );
27       for ( int n : neighbors ) {
28
29         // Create temporary new path with neighbor at end
30         auto temp = path;
31         temp.push_back(n);
32
33         // Enqueue this path on _tail_ of queue
34         worklist.enq( temp );
35       }
36     }
37   }
38 }
```

5. Other Common Search Algorithms

Dijkstra's Shortest Path Algorithm
Greedy Search
A* Search

6. Constructing a Minimum Spanning Tree

Prim's Algorithm
Kruskal's Algorithm

Algorithms	Data Structures
mul: iter, single step sqrt: iter, recur	chain of nodes array of elements
search: linear, binary sort: insertion, selection, merge, quick, hybrid, bucket set intersection, set union find path: DFS, BFS	list, vector stack, queue, set, map tree, table, graph

- Simple algorithms do not use a non-trivial data structure
- Simple data structures do not provide non-trivial operations
- Many algorithms operate on a simple data structure
- Many data structures provide operations which are implemented using an algorithm that operates on a simple data structure

- Sometimes our programs are more algorithm centric, sometimes they are more data-structure centric, but they almost always use both algorithms *and* data structures

Algorithm + Data Structure = Program