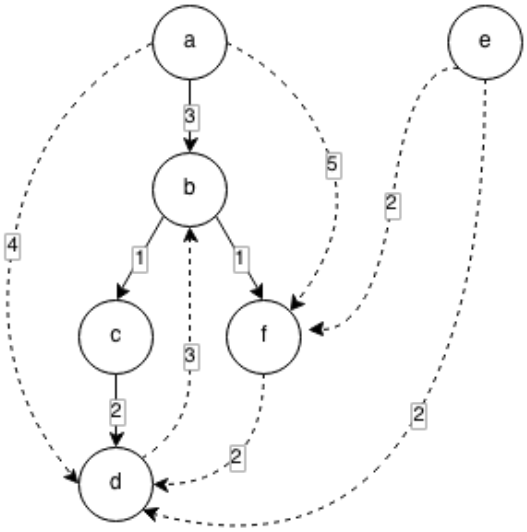
Tim Cheeseman
CS 260-003
Summer 2014
Assignment 9 Sample Solution

6.2)

We can model each task $T_1, T_2, \ldots T_n$ as a vertex in a directed acyclic graph. The constraints will be represented by directed edges (v, w) where T_v must be completed prior to T_w. Each vertex v would be associated with its runtime t_v.

We could use topological sort to find a topological sorting of the dag. If the tasks were completed in parallel, all tasks would be completed in the time it takes to complete the longest path. We could determine the longest path in the ordering using a greedy algorithm, following the topological sort in order and, for each vertex, keeping track of the longest incoming path ending at that vertex. The length of the longest path found would be the minimum length of time to complete all tasks.

6.8)



Edge	Category
a→b	Tree
b→c	Tree
b→f	Tree
c→d	Tree
a→d	Forward
a→f	Forward
d→b	Back
f→d	Cross
e→f	Cross
e→d	Cross

Vertex	DFS #
a	1
b	2
c	3
d	4
f	5
e	6

6.10)

If we perform a depth-first search starting from a vertex *r* in a dag *G*, and that recursive call visits all vertices (i.e. the resulting spanning forest consists of a single spanning tree), then *r* is a root of *G*. If no such roots are found, then the *G* is not rooted.

```
def is_rooted(G, n):
  """
  For a dag G with n vertices numbered 1..n, determine if G is rooted
  """
  visited = ∅
  for v from 1 to n:          # for each vertex in G
    dfs(G, v, visited)        # perform DFS starting at v
    if len(visited) == n:     # if all nodes were visited, root found
      return True
    visited = ∅               # reset for next starting vertex
  return False                # no root found

def dfs(G, v, visited):
  """
  Recursively build a set of visited vertices in G starting at v
  """
  visited = visited ∪ {v}
  for each edge v→w in G:     # visit each of v's unvisited neighbors
    if w ∉ visited:
      dfs(G, w, visited)
```

6.14)

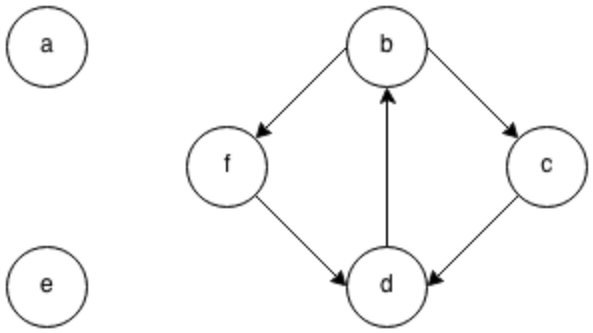
```
def longest_path(G):
  # keep a map of vertex to longest path length ending at that vertex
  # initialize to 0 for all vertices
  longest_paths = [0] * n

  sorting = top_sort(G) # get a topological sort of G
  for v in sorting:    # for each vertex in the sorting
    # set v's longest path length by examining its incoming edges and
    # choosing the neighbor w with the longest path, then adding 1 for w→v
    max_w = index of max value in longest_paths where an edge w→v exists
    longest_paths[v] = longest_paths[max_w] + 1

  # find the end of the path with maximum length
  v = index of max value in longest_paths

  # backtrack to find the reversed path
  max_path = [v]
  while longest_paths[v] > 0:# while v has incoming edges
    max_w = index of max value in longest_paths where an edge w→v exists
    max_path.append(max_w)   # add neighbor with longest path
    v = max_w                # continue search from chosen neighbor

  return reverse(max_path)
```

The topological sort is done in O(V+E) time.
Traversing the sorting is O(V) and finding the longest incoming path is O(E) totaling O(V·E).
Finding the max in longest paths is O(V).
Backtracking the paths is O(V·E).
Reversing the path is O(V).

Dominating term is O(V·E).

6.15)

The strong components of figure 6.38 are {a}, {e}, and {b, c, d, f}. They are shown below with edges between members of strongly connected components (edges between strong components are not shown).



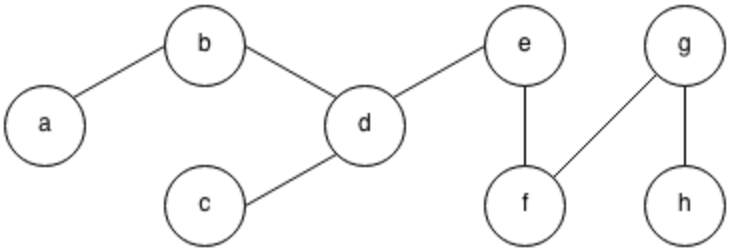
6.20)

```
def print_simple_paths(G, v, w):
  """
  Use BFS to print simple paths in G from v to w
  """
  paths = []
  paths.append([v])

  while not len(paths) == 0: # while path queue is non-empty
    path = paths.popleft()   # grab first path in queue
    if path[-1] == w:        # if last element is w, we have a path from v→w
      print path
    for each edge v→w in G:
      if w not in path:            # ignore paths with cycles (non-simple)
        paths.append(path + [w])   # add path to BFS queue
```

This is just BFS but keeping track of paths instead of visited vertices.
Running time is O(max(V, E)) which is typically O(E).

7.3)

c i)



c ii)