Data structure [A15]

김종규, PhD

2017-06-12

Outline

- Application of DFS
  - Strongly connected component
    - Why this algorithm works
  - Topological sort
- Advanced BFS
  - Shortest path: Dijkstra algorithm
  - Importance of data structure for implementing algorithms

Strongly connected component

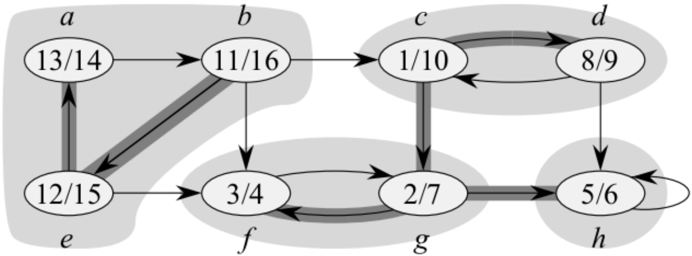


그림: Strongly connected component

Strongly connected component

STRONGLY-CONNECTED-COMPONENTS(G)

```
1  call DFS(G) to compute finishing times u.f for each vertex u
2  compute G^T
3  call DFS(G^T), but in the main loop of DFS, consider the vertices
       in order of decreasing u.f (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
       separate strongly connected component
```

그림: Strongly connected component algorithm

Parenthesis theorem

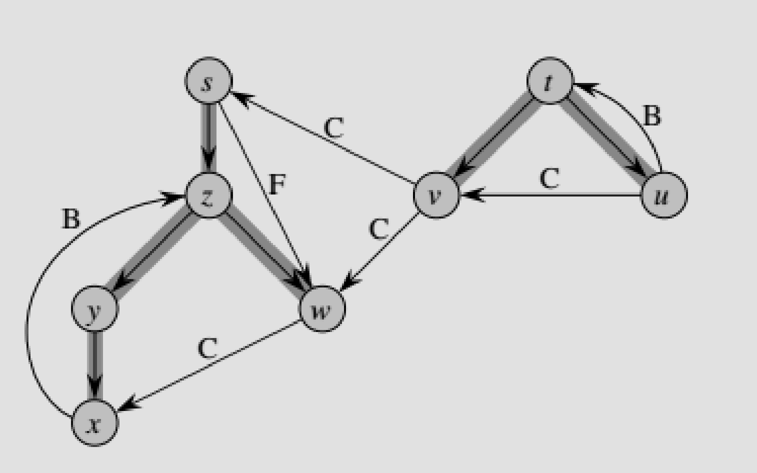


그림: d

Types of edges

1. ***Tree edges*** are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
2. ***Back edges*** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.
3. ***Forward edges*** are those nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
4. ***Cross edges*** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

그림: q

Parenthesis theorem

Theorem 22.7 (Parenthesis theorem)
In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

- the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
- the interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, and u is a descendant of v in a depth-first tree, or
- the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$, and v is a descendant of u in a depth-first tree.

그림: a

Parenthesis theorem

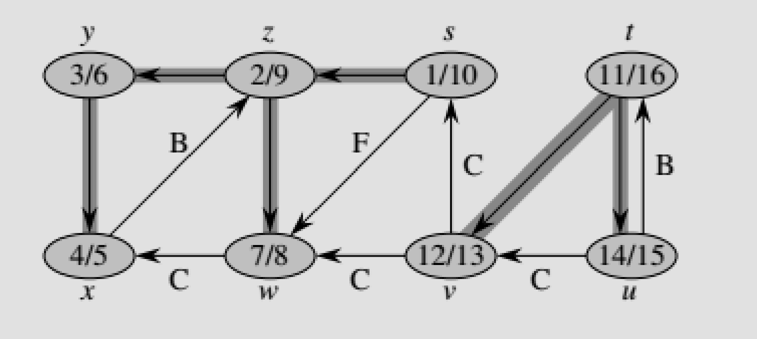


그림: b

Parenthesis theorem

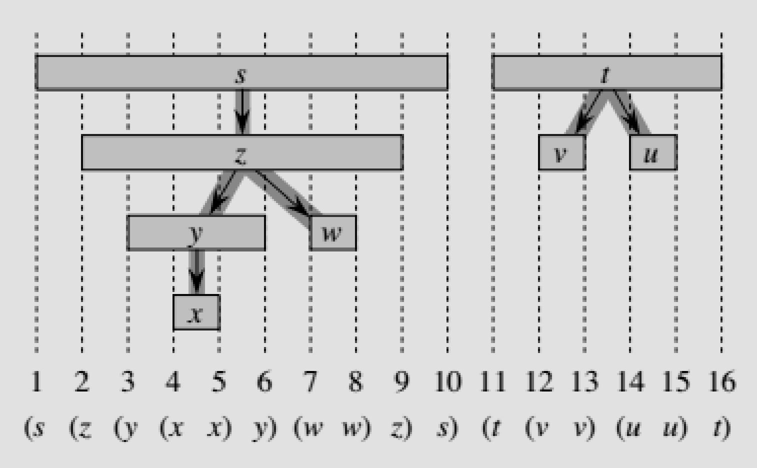


그림: c

Strongly connected component

Lemma 22.13
Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$, let $u, v \in C$, let $u', v' \in C'$, and suppose that G contains a path $u \leadsto u'$. Then G cannot also contain a path $v' \leadsto v$.

그림: m

Strongly connected component

Lemma 22.14
Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

그림: n

Strongly connected component

Corollary 22.15
Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E^{\mathrm{T}}$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

그림: o

Topological sort

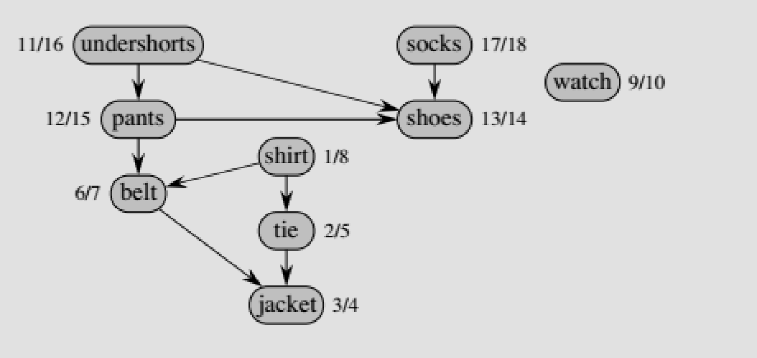


그림: j

Topological sort

그림: k

Topological sort


```
TOPOLOGICAL-SORT(G)
call DFS(G) to compute finishing times v.f for each vertex v
as each vertex is finished, insert it onto the front of a linked list
return the linked list of vertices
```

그림: I

Single source shortest path

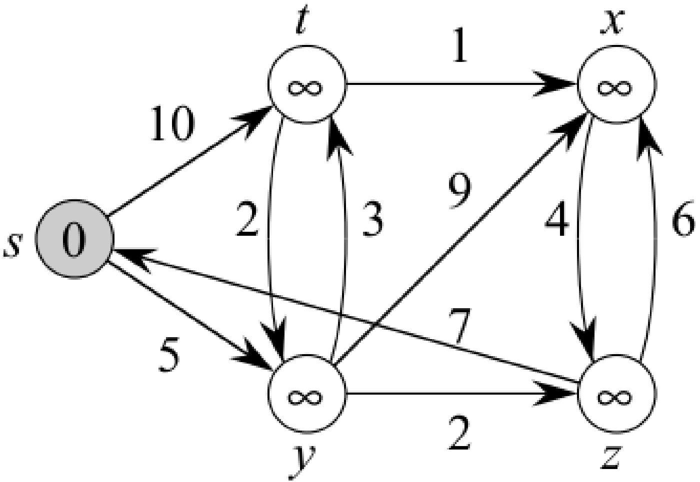


그림: Problem

Single source shortest path

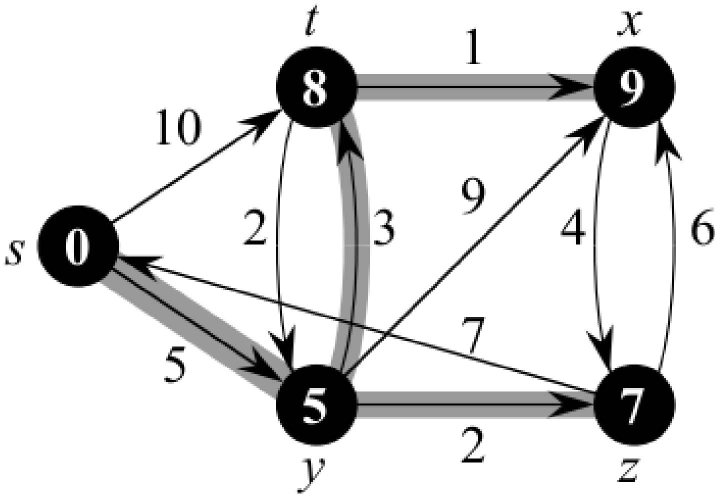


그림: Solution

Administrivia

- 기말고사: 6/19 (월) 12:00-13:40
- Final project
  - Due: 6/22 (목)
  - red black tree 를 확장하여 string 을 다루도록 하시오
- 수요일 수업: 보강
  - 질의/응답

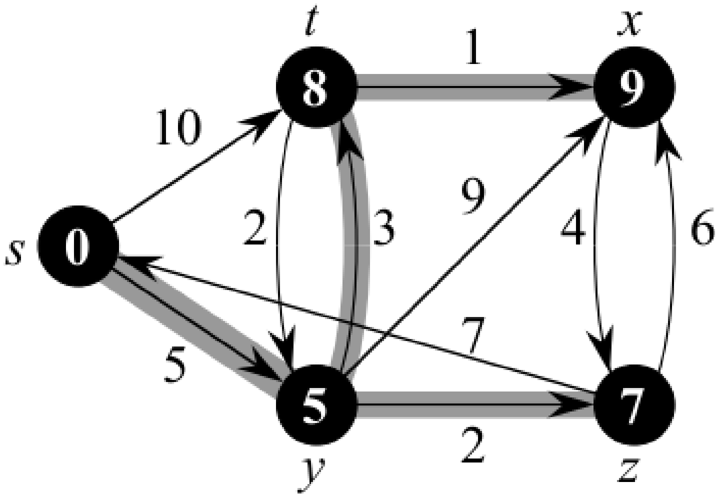
Dijkstra algorithm

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

그림: Dijkstra algorithm

Dijkstra algorithm (6/6)

- $G_w = (V, E, w)$



Single source shortest path


```
g = Dijkstra()
s = g.add_vertex('s')
...
z = g.add_vertex('z')
s.add(t,10)
...
z.add(x,6)
s.d = 0
g.shortest_path()
g.print_vertices()
```

Shortest path algorithm

```
class Dijkstra:
    def shortest_path(self):
        q = self.q
        vset = self.vertices
        for v in vset:
            n = PrioNode(v.d, v.n)
            v.set_priority(n)
            q.insert(n)
        while not q.is_empty():
            u = q.extract()
            self.relax(vset[u.n])
```

Decrease key


```
class MinQueue(MaxQueue):
    def decrease_key(self,i,key):
        A = self.A
        if key > A[i].key:
            print ("Error")
            sys.exit(-1)
        A[i].key = key
        self.update_key(i)
```

Implement decrease key

```
class DijkVertex(Vertex):
    def decrease_key(self, q):
        prio = self.priority
        ndx = prio.ndx
        q.decrease_key(ndx, self.d)
```

Do not trust google


```
def dijsktra(graph, initial):
    visited = {'initial': 0}
    path = {}
    nodes = set(graph.nodes)
    while nodes:
        min_node = find_min(nodes)
        nodes.remove(min_node)
        current_weight = visited[min_node]

        relax(graph, visited, current_weight, min_node
    return visited, path
```

The Devil is in the detail


```
def find_min(nodes, visited):
    min_node = None
    for node in nodes:
        if node in visited:
            if min_node is None:
                min_node = node
            elif visited[node] < visited[min_node]:
                min_node = node
    return min_node
```

Extract-min

- 모든 vertex 를 queue 에 저장
- 가장 앞에는 가장 작은 d 값을 갖는 vertex 가 저장됨
- 이 vertex 주변의 vertex 의 d 값을 조정
- 조정된 값을 기반으로 queue 에서의 순서가 바뀜

$\longrightarrow$ 정렬될 필요가 있을까? 없다 (Heap 이면 충분)

Priority queue

- `q.insert(v)`
- `q.extract_min() -> v`
- `q.decrease_key(v)`

Priority queue (min)

- insert(x)
- minimum()
- extract-min()
- decrease-key(i)
- empty()

Priority queue


```
def main():
    print("---- main ----")
    q = MinQueue()
    q.insert(PrioNode(0,8))
    q.insert(PrioNode(1,4))
    print(q.A)
    q.decrease_key(1,1)
    print(q.A)
```

Priority node


```
class PrioNode:
    def __init__(self, n, key):
        self.n = n
        self.key = key
    def __repr__(self):
        return "(%d:%d,%d)" % (self.ndx,self.n, self.
```

Min queue

```
class MinQueue(MaxQueue):
    def __init__(self):
    def compare(self,a,b):
    def update_key(self,i):
    def decrease_key(self,i,key):
```

Max queue


```
class MaxQueue(Heap):
    def __init__(self):
    def compare(self,a,b):
    def exchange(self,i,j):
    def update_key(self,i):
    def increase_key(self,i,key):
    def insert(self,n):
    def extract(self):
    def empty(self):
```

Heap


```
class Heap:
    def __init__(self):
    def parent(self,n):
    def left(self,n):
    def right(self,n):
    def compare(self,a,b):
    def exchange(self,i,j):
    def heapify(self,i):
```

Dijkstra algorithm (1/6)

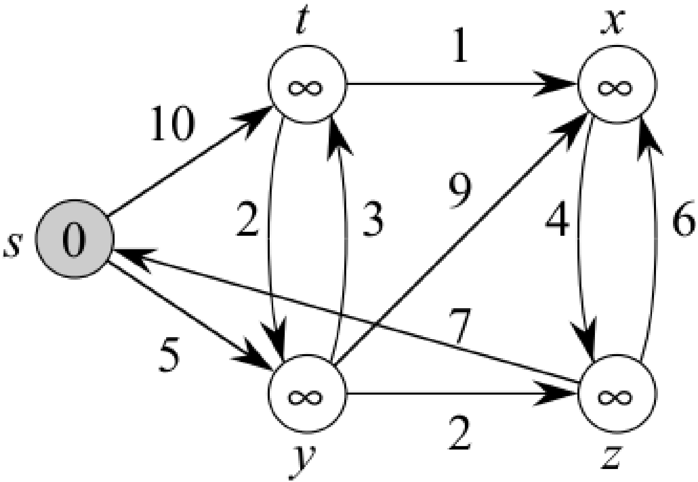


그림: Dijkstra algorithm (1/6)

Dijkstra algorithm (2/6)

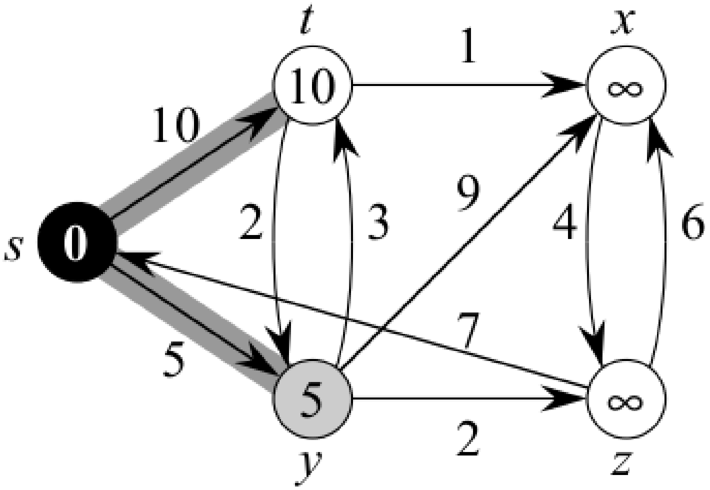


그림: Dijkstra algorithm (2/6)

Dijkstra algorithm (3/6)

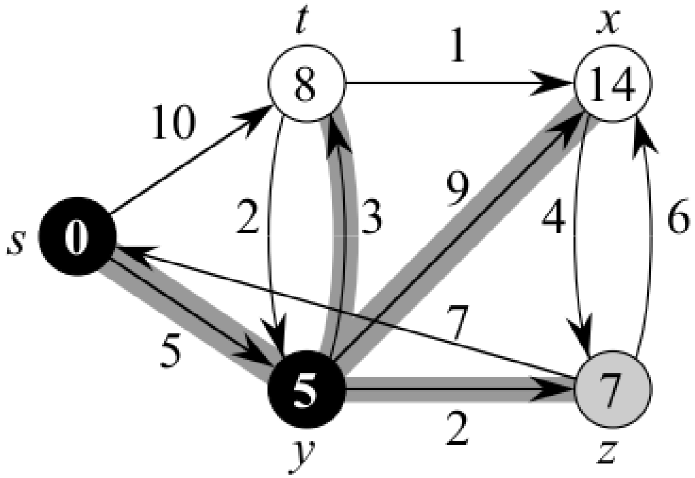


그림: Dijkstra algorithm (3/6)

Dijkstra algorithm (4/6)

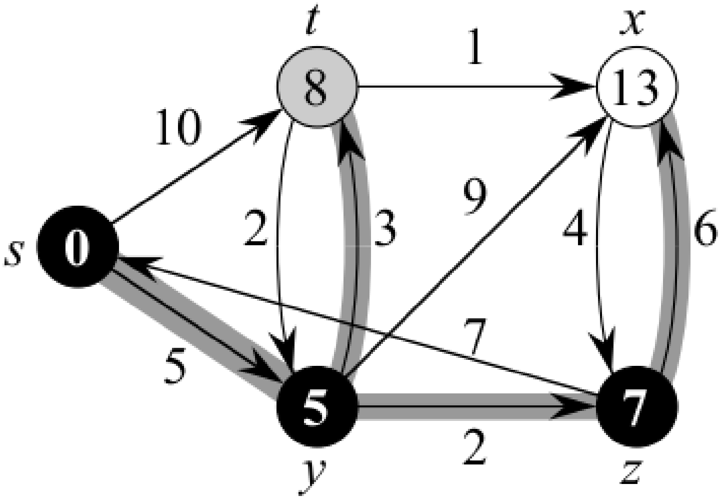


그림: Dijkstra algorithm (4/6)

Dijkstra algorithm (5/6)

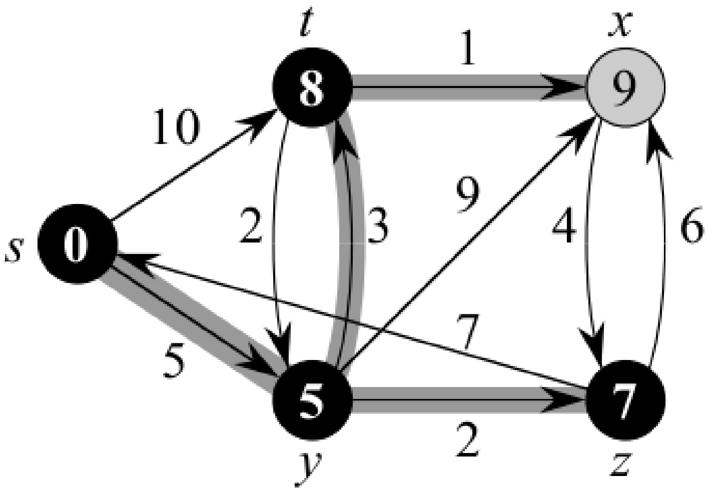


그림: Dijkstra algorithm (5/6)

Dijkstra algorithm (6/6)

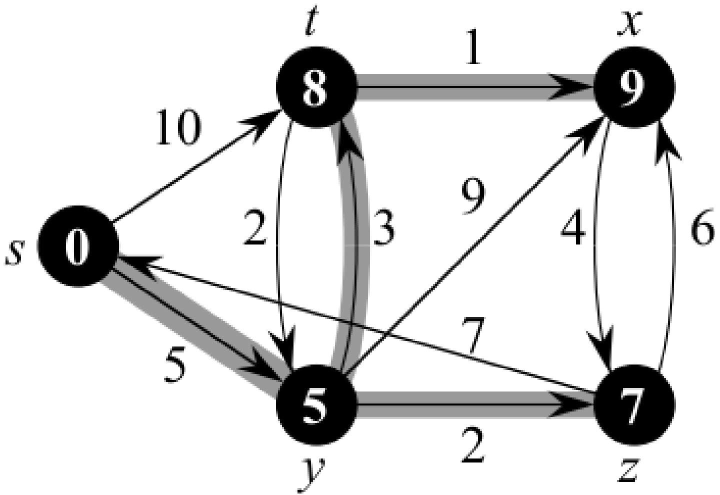


그림: Dijkstra algorithm (6/6)

Wrap-up (1/2)

- We reviewed applications of DFS
  - Why strongly connected components work
  - How to sort partially ordered relations
- We extended BFS
  - Single source shortest path algorithm (a.k.a, Dijkstra algorithm)

Wrap-up (2/2)

- We learned that algorithm description is based on abstract data type
  - A line in an algorithm usually translated to a data structure operation
  - The complexity of an algorithm is determined by this single line, i.e., an operation performed on a data structure
  - Data structure is usually hidden in algorithm description but plays very important role in software development in general