Tutorial for Graph Algorithm

Hengfeng Wei

hengxin0912@gmail.com

December 19, 2011

Outline

BFS and DFS

"For fundamental achievements in the design and analysis of algorithms and data structures."

— Turing Award 1986

Figure: Robert Endre Tarjan (April 30, 1948).

Figure: John Edward Hopcroft (October 7, 1939).

Outline

Types of edges on DFS

DFS on digraph:

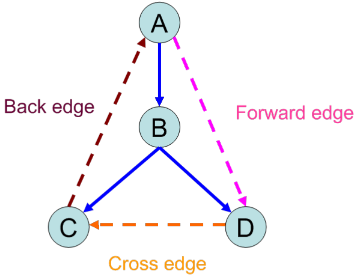


Figure: DFS on digraph.

DFS on undirected graph ([P_{380} 7.28]):

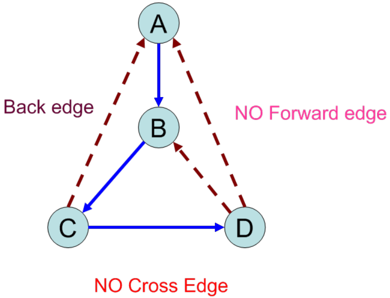


Figure: DFS on undirected graph.

Types of edges on BFS

BFS on digraph:

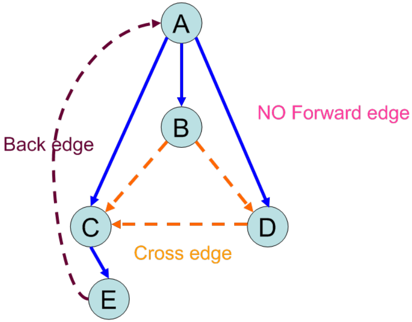


Figure: BFS on digraph.

BFS on undirected graph:

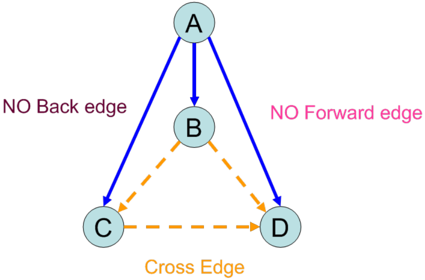
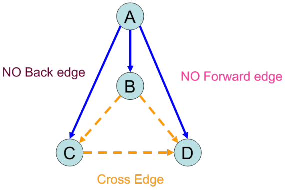
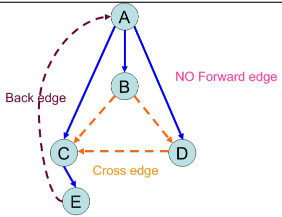
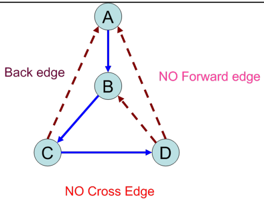
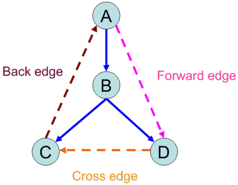


Figure: BFS on undirected graph.

Cycle detection problems

	Undirected graph	Directed graph ([P_{379} 7.17])
BFS	 (Cross edge)	 (Back edge?)
DFS	 Back edge	 Back edge

Cycle detection problems

Using BFS on ~~un~~directed graph for cycle detection:

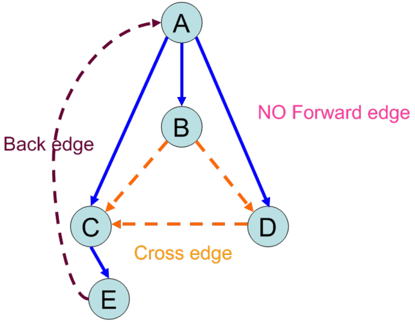


Figure: BFS on digraph with back edges.

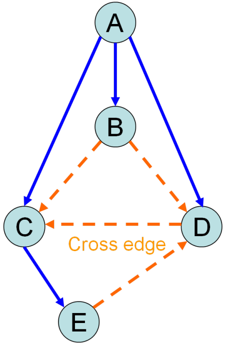


Figure: BFS on digraph without back edges.

Outline

Exploring DFS's active intervals

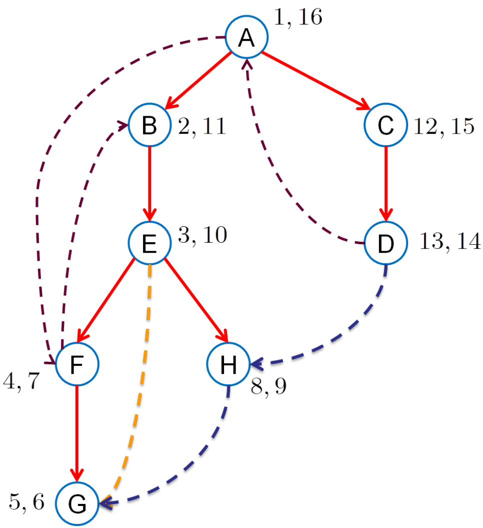


Figure: Active intervals in DFS on digraph.

Exploring DFS's active intervals

1. Who is ancestor ?
2. How many descendants ?

Exploring DFS's active intervals

1. Who is ancestor ?
2. How many descendants ?
3. topological sorting (on digraph)
4. strongly-connected components (on digraph)
5. biconnected components (on undirected graph)

DAG and topological sorting

1. absence of back edges = acyclicity (DAG)
2. topological sorting
   - source, sink

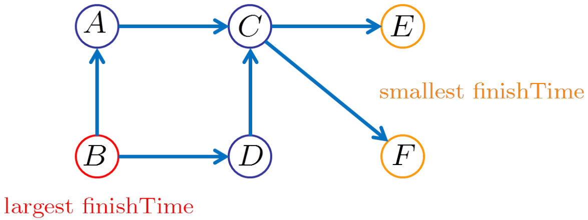


Figure: Example of DAG.

DAG and topological sorting

- critical path $\rightarrow$ longest path in DAG

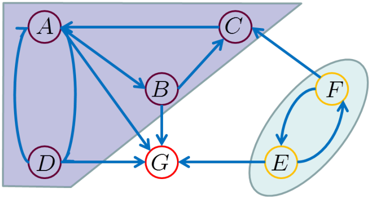
DAG and topological sorting

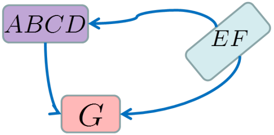
- critical path $\rightarrow$ longest path in DAG

Note: In general graphs, longest path problem is NP-hard !

SCC of digraph

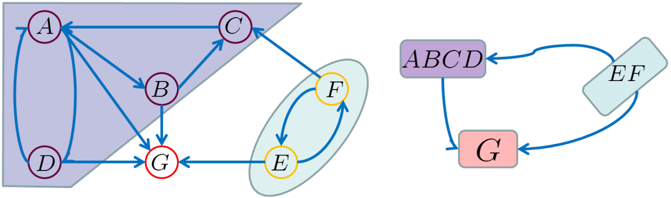
"Two-trier" structure of digraph ([P_{379} 7.22]):





SCC of digraph

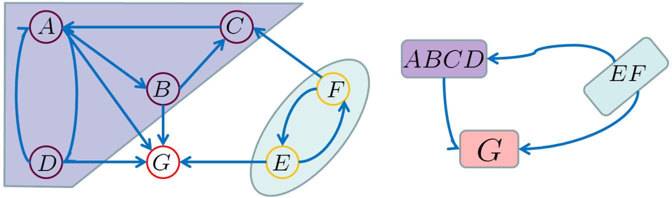
"Two-trier" structure of digraph ([P_{379} 7.22]):



1. "The node that has highest finishTime in DFS must lie in a source SCC";

SCC of digraph

"Two-trier" structure of digraph ([P_{379} 7.22]):



1. "The node that has highest finishTime in DFS must lie in a source SCC";
2. What we need is sink! So try G^R; ([P_{380} 7.26])
3. When a sink SCC is found (by connectivity), just delete it and continue recursively.

Biconnected components of an undirected graph

Algorithms for biconnected components:

Brute force: try every vertex and check connectivity: $O\big(n(m+n)\big)$.

Clever approach: single DFS making use of *back edges and active intervals*.

Biconnected components of an undirected graph

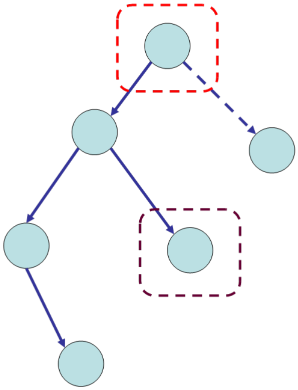


Figure: What if DFS tree is the entirety of the graph ?

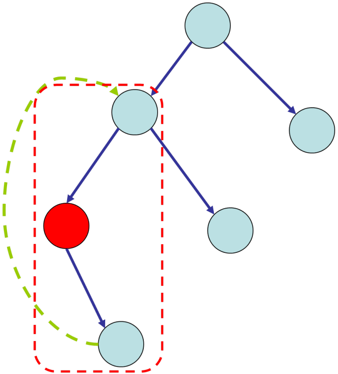


Figure: Back edge, cycle, no articulation vertices.

Biconnected components of an undirected graph

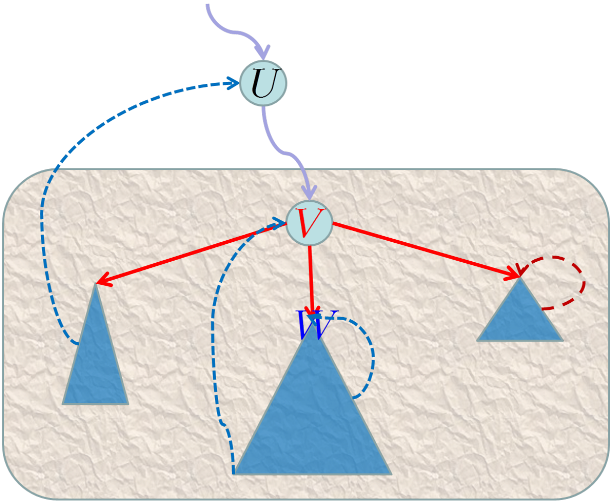


Figure: Three cases of articulations (1).

Biconnected components of an undirected graph

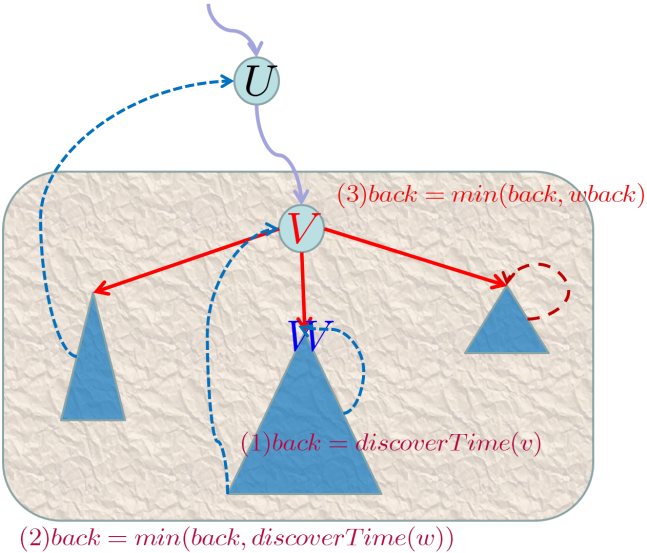


Figure: Three cases of articulations (2)

Biconnected components of an undirected graph

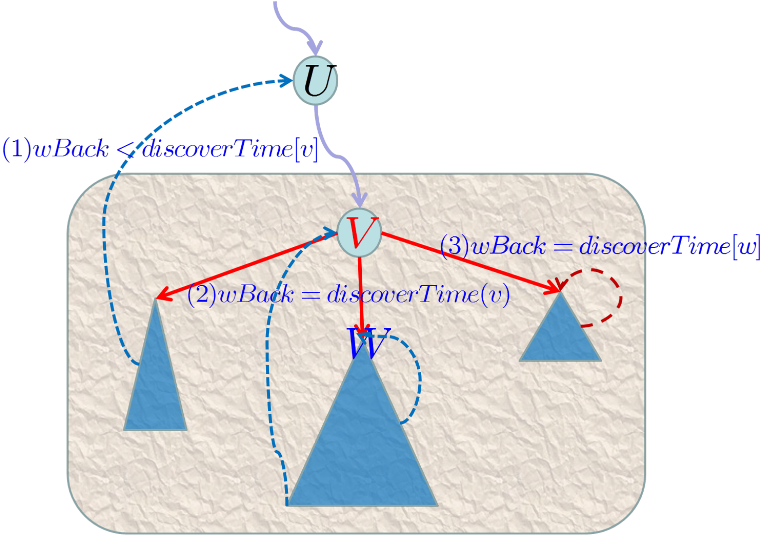


Figure: Three cases of articulations (3).

Biconnected components of an undirected graph

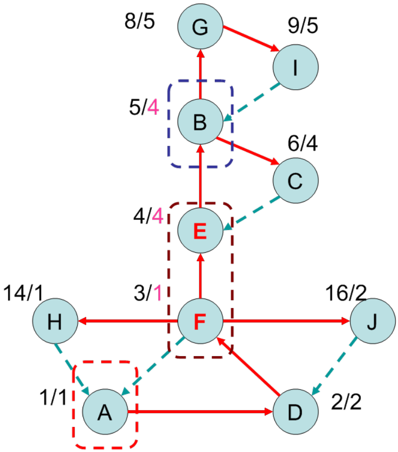


Figure: Three cases of articulations.

Biconnected components of an undirected graph

Q: How the reachability relation impacts whether v is an articulation vertex ?

1. Root cut-nodes ([P_{382} 7.37]).
2. Bridge cut-nodes.
3. Parent cut-nodes.

Biconnected components of an undirected graph

Q: How the reachability relation impacts whether v is an articulation vertex ?

1. Root cut-nodes ([P_{382} 7.37]).
2. Bridge cut-nodes.
3. Parent cut-nodes.

Q : $\text{Back} \geq \text{discoverTime}[v]$ ([P_{383} 7.42]).

Biconnected components of an undirected graph

Q: How the reachability relation impacts whether v is an articulation vertex ?

1. Root cut-nodes ([P_{382} 7.37]).
2. Bridge cut-nodes.
3. Parent cut-nodes.

Q : $\text{Back} \geq \text{discoverTime}[v]$ ([P_{383} 7.42]).

A : $\text{Back} = \text{discoverTime}[v]$. It just detects Bridge cut-nodes.

Minimum Spanning Tree

- (Evaluate Prim's MST algorithm implemented with min-heap ([P_{417} 8.9]):)
    1. Find the asymptotic order of the number of comparisons of edge weights in the worst case.
    2. Find the asymptotic order of the number of comparisons of edge weights on a bounded-degree family.
    3. Find the asymptotic order of the number of comparisons of edge weights on a planar graph.

Minimum Spanning Tree

$$T(n,m) = O\big(nT(getMin) + nT(deleteMin) + mT(decreaseKey)\big).([P_{395}])$$

Minimum Spanning Tree

$$T(n,m) = O\big(nT(getMin) + nT(deleteMin) + mT(decreaseKey)\big).([P_{395}])$$

- $getMin$: $O(1)$.

  no comparison of edge weights.
- $deleteMin$: $O(\log n)$.

  NOT: $O(\log m)$.
- $decreaseKey$: $O(\log n)$.

  NOT: $O(n + \log n)$ where $O(n)$ for search ([P_{296}]).
- "Else if newWgt less than fringeWgt for w" ([P_{395}]) : $O(m)$.
- In total,

$$T(n,m) = O(n\log n + m\log n + 2m) = O((n+m)\log n).$$

Minimum Spanning Tree

- $m \leq \frac{nk}{2}, \Rightarrow T(n,m) = O((n+m)\log n) = O((n+\frac{nk}{2})\log n) = O(n\log n)$.

Minimum Spanning Tree

- $m \leq \frac{nk}{2}, \Rightarrow T(n,m) = O((n+m)\log n) = O((n+\frac{nk}{2})\log n) = O(n\log n)$.
- First, we should know the relation between m and n.

$$n - m + f = 2. \tag{1}$$

If $deg(f_i) \geq l$,

$$2m = \sum_{i=1}^{f} deg(R_i) \geq lf. \tag{2}$$

$$m \leq \frac{l}{l-2}(n-2) \tag{3}$$

Minimum Spanning Tree

- $m \leq \frac{nk}{2}, \Rightarrow T(n, m) = O((n+m)\log n) = O((n + \frac{nk}{2})\log n) = O(n \log n)$.
- First, we should know the relation between m and n.

$$n - m + f = 2. \tag{1}$$

If $deg(f_i) \geq l$,

$$2m = \sum_{i=1}^{f} deg(R_i) \geq lf. \tag{2}$$

$$m \leq \frac{l}{l-2}(n-2) \tag{3}$$

If G is tree, $m = n - 1$; Else, $l \geq 3$.

$$m \leq \frac{l}{l-2}(n-2) \leq 3(n-2) = 3n - 6 \tag{4}$$

$T(n, m) = O((n+m)\log n) = O((n+3n)\log n) = O(n \log n)$.

Shortest paths

Different editions of shortest paths problems:

1. shortest(longest) path in DAG

   Dynamic Programming

2. single-source shortest paths

   - No negative edges.

     Dijkstra algorithm.

   - With negative edges (No negative cycle).

     Bellman-Ford algorithm.

3. all pairs shortest paths

   Floyd-Warshall algorithm.

Shortest paths

DAG can be topologically sorted, so *DP* works.

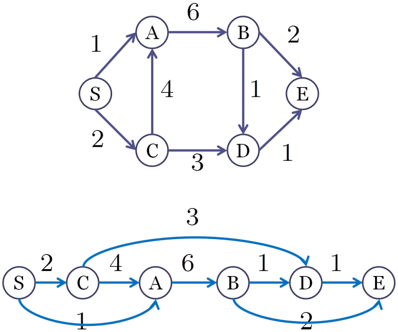


Figure: A dag and its topological sorting.

$$dist(D) = \min\{dist(B) + 1, dist(C) + 3\}.$$

Shortest paths

Shortest path without negative edges : *Dijkstra algorithm*

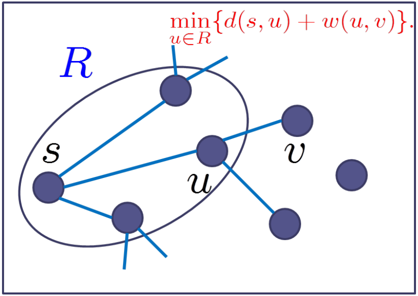


Figure: Property of shortest paths.

Priority queue implementations

Complexity of Dijkstra algorithm:

1. makequeue, $|V| \cdot insert$
2. $|V| \cdot deletemin$
3. $|E| \cdot descreaseKey$

Priority queue implementations

Complexity of Dijkstra algorithm:

1. makequeue, $|V| \cdot insert$
2. $|V| \cdot deletemin$
3. $|E| \cdot descreaseKey$

Different implementations of priority queue:

Implementation	deletemin	insert, decreaseKey	total
Array	$O(V)$	$O(1)$	$O(V^2)$
Binary heap	$O(\log V)$	$O(\log V)$	$O((V+E)\log V)$
Fibonacci heap	$O(\log V)$	$O(1)$	$O(V \log V + E)$

Shortest path with negative edges: *Bellman-Ford algorithm*

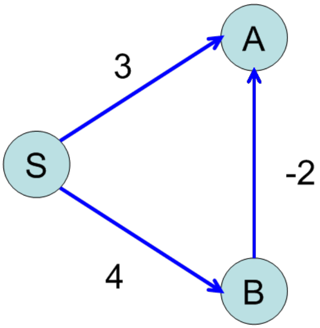
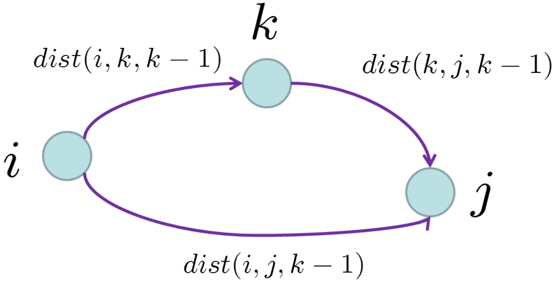


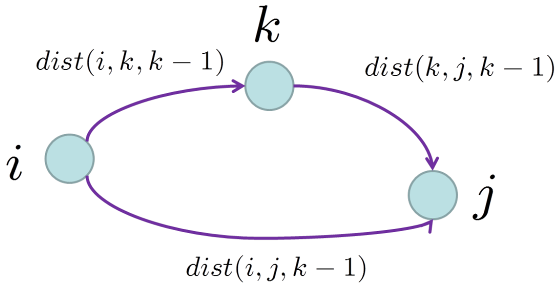
Figure: Dijkstra algorithm fails if there are negative edges ([P_{418} 8.14]).

All pairs of shortest paths: *Floyd-Warshall algorithm*



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

All pairs of shortest paths: *Floyd-Warshall algorithm*



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

Assumption: No negative cycles.

All pairs of shortest paths: *Floyd-Warshall algorithm*

- Routing table for all-pair shortest path ([P_{448} 9.10]).
- Length of shortest cycle in digraph ([P_{448} 9.12]).

All pairs of shortest paths: *Floyd-Warshall algorithm*

- Routing table for all-pair shortest path ([P_{448} 9.10]).

```
if path[i][k] + path[k][j] < path[i][j] then
   path[i][j] := path[i][k]+path[k][j];
   next[i][j] := k;
```

```
procedure GetPath (i,j)
   if path[i][j] equals infinity then
      return "no path";
   int intermediate := next[i][j];
   if intermediate equals 'null' then
      return " ";   /* there is an edge from i to j, with no vertices between */
   else
      return GetPath(i,intermediate) + intermediate + GetPath(intermediate,j);
```

Figure: Construction of routing table.

All pairs of shortest paths: *Floyd-Warshall algorithm*

- Length of shortest cycle in digraph ([P_{448} 9.12]).

$$path[i][i]$$

$$path[i][i] < 0?$$

Fail for undirected graph:

$$\{v, w\} \to (v, w, v).$$


Thank
You!