Tutorial for Graph Algorithms

Hengfeng Wei (魏恒峰)

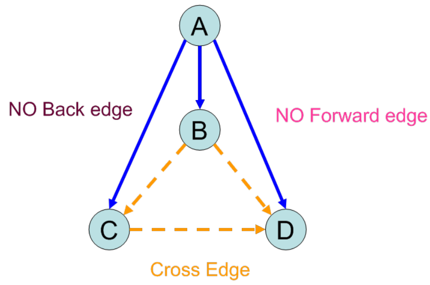
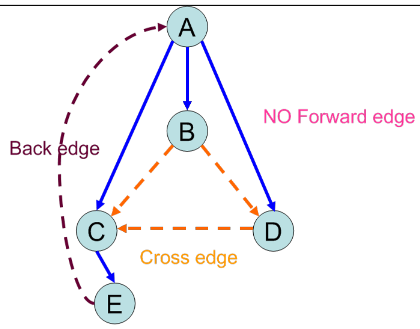
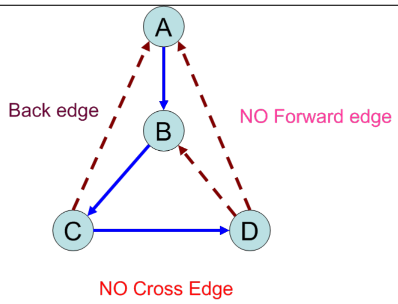
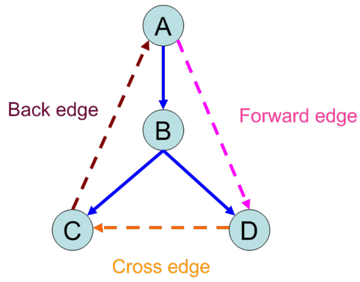
2011 年 12 月 18 日

Contents

1 BFS and DFS

BFS 和DFS 算法本身并不困难, 但是它们却蕴含着极强的解决其他图论算法问题的能力. 究其原因, 我们可以归纳出四点.

Table 1: BFS and DFS on undirected graph and directed graph

	Undirected graph	Directed graph
BFS		
DFS	 ([P_{380} 7.28])	

1. BFS, DFS 提供了一种exploring graph 的框架. 我们需要掌握该框架, 理解图中每个顶点v 的状态的变化, 以及在每种状态下, 我们所拥有的信息和可以对v进行的操作.
2. BFS, DFS 将graph 的边做了classification. 不同类型的边具有不同的性质, 在具体的图论算法中又扮演着不同的角色. 理解这些不同的类型和角色对于一些算法来说至关重要.
3. DFS 的time interval (下文详述) 提供了点与点之间的各种关系, 可以用来高效求解一些重要问题.
4. BFS 与Dijkstra, Prim 的关系.

1.1 Exploring Types of Edges on Both BFS and DFS

1.1.1 Types of Edges on Both BFS and DFS

表1区分了BFS和DFS作用于undirected graph 和directed graph 时所得边的类型.

Table 2: Cycle detection

	Undirected graph	Directed graph ([P_{379} 7.17])
BFS	Cross edge	NOTE: What if there are no BACK edges?
DFS	Back edges	absence of back edges = acyclicity = linearizability (DAG)

1. BFS
   (a) On undirected graph
   - properties of non-tree edges: layer constrains (different from that on Digraph).
   - Applications: Connected components; Bipartite.

   (b) On Digraph
2. DFS
   (a) On undirected graph.
   - Notice that there are only two types of edges: tree edge and back edge.
   -

   (b) On Digraph.

1.1.2 Cycle Detection Problems

我们分别考察在无向图和有向图上的cycle detection问题,并且分别考虑BFS和DFS两种解决方案:

Problem 1: Existence of cycle ?

表2 总结了cycle detection 算法类型.

Problem 2: All cycles ?

Problem 3: Shortest cycle ?

1. Directed graph : Floyd-Warshall algorithm $D[i][i]$
2. Undirected graph :

Problem 4: Length=k cycle ?

1.2 Exploring DFS's Time Intervals

在DFS中,我们顺带记录了访问每个顶点的discoverTime 和finishTime. 这可以看作每个顶点在DFS算法中的生存期, 它们共同定义了一个time interval. 该time interval 在许多场合中都有重要应用.

1. Who is an ancestor?
2. How many descendants?
$$\frac{finishTime - discoverTime}{2}.$$
3. topological sorting
4. strongly-connected components
5. biconnected components

1.2.1 DAG and Topological Sorting

DAG 具有topological sorting, 其重要性在于它提供了一种不违背顶点间依赖次序的处理顺序.该顺序在Job scheduling, Critical path 和Dynamic Programming 问题中至关重要.

1. Check: back edge; Search: Reverse topological ordering.
2. Critical path. 其实是DAG 中的longest path. 在General digraph 中,
3. DP (略. 在以后的课程中介绍).

1.2.2 Strongly Connected Components of a Digraph

"Two-trier" structure of digraph :

1. SCC
2. DAG ([P_{379} 7.22])

SCC 算法的思想:

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

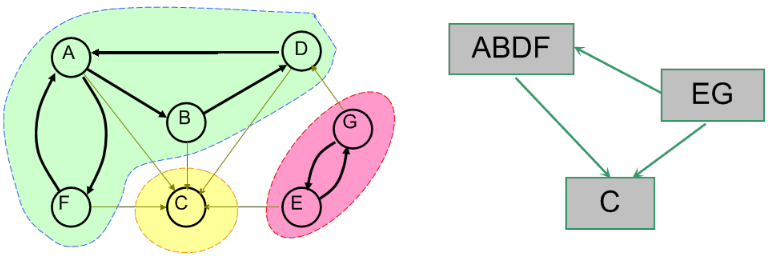


Figure 1: Strong connected components and corresponding DAG.

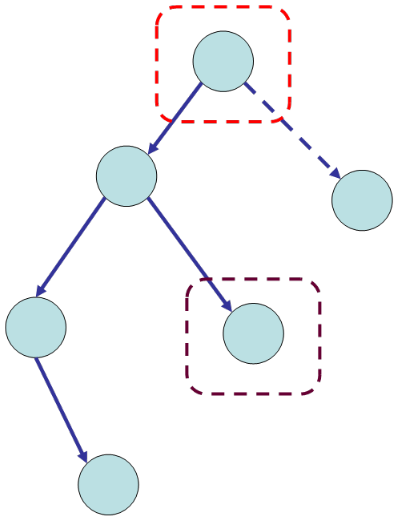


Figure 2: What if DFS tree is the entirety of the graph ?

2. What we need is sink! So try G^R; ([P_{380} 7.26])

3. When a sink SCC is found (connectivity) , just delete it and continue.

1.2.3 Biconnected Components of an Undirected Graph

- By brute force : choose one vertex to cut and check the connectivity of remaining graph using BFS or DFS. Complexity: $O(n(m+n))$.

- Clever approach: A single DFS making use of back edges and time intervals.

    1. 基本思想见图2,图3和图4.

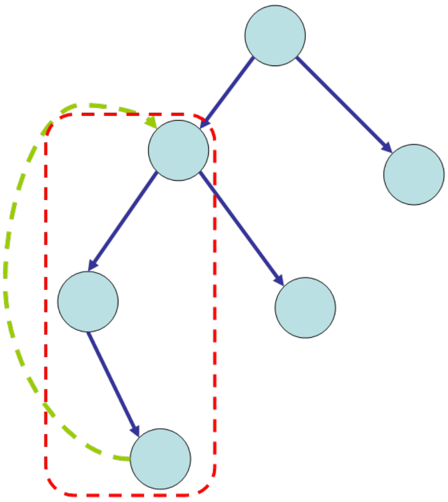


Figure 3: Back edge, cycle, no articulation vertices.

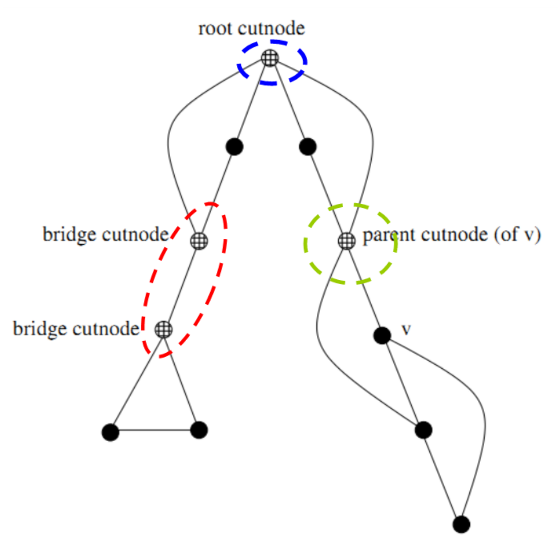
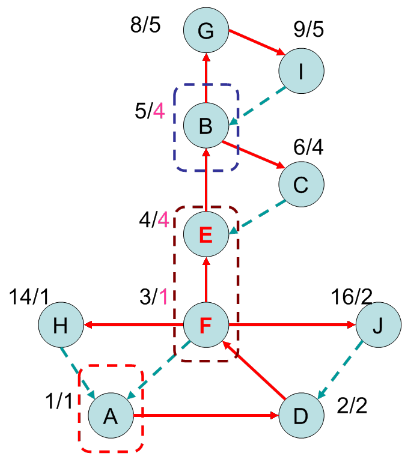


Figure 4: Three cases of articulations.



2. Important information: the extent to which back edges link chunks of the DFS tree back to ancestor nodes. 需要考察DFS算法框架，在每次处理v时更新back值:
   (a) v is first discovered. 初始化$back = discoverTime(v)$.
   (b) back edge vw. $back = min(back, discoverTime(w))$.
   (c) backtracking from w to v $back = min(back, wback)$.
3. Q: How the reachability relation impacts whether v is an articulation vertex (见图3).
   (a) Root cut-nodes.
   (b) Bridge cut-nodes ([P_{383} 7.42]).
   (c) Parent cut-nodes.
   Q : Back $\geq$ discoverTime[v] ([P_{383} 7.42]).
   A: Back $=$ discoverTime[v]. It just detects Bridge cut-notes.

2 Minimum Spanning Tree

3 Shortest Paths

3.1 Single-source Shortest Paths

1. DAG:
   acyclicity = linearizability = absence of back edges in DFS trees.

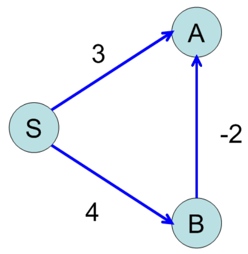


Figure 5: Dijkstra's algorithm will not work if there are negative edges.

2. No negative edges:
   Dijkstra 算法. 得到的是shortest path tree; 成功的关键部分在于positive ([P_{418} 8.14]), 因此如果$s \dots u - v$是$s \to v$的最短路径, 则$dist(u) < dist(v)$. Dijkstra 算法可以看作一系列的updates, 该系列是特定顺序的.

   Dijkstra alg 在with negative edges 情况下失效的例子如图5所示.

3. With negative edges:
   Bellman-Ford 算法,得到的不一定是tree (无向图也是一种有向图). 可不可以每次不update所有的边,而只涉及那些i步之内到达的顶点?

3.1.1 Priority queue implementations

Dijkstra's algorithm 算法复杂度:

1. makequeue $V \cdot insert$

2. $V \cdot deletemin$

3. $E \cdot descreaseKey$

Dijkstra algorithm 的性能依赖于Priority queue 所采用的数据结构.

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

3.2 K-edges Constrained Shortest Paths

3.3 All Pairs Shortest Paths

Floyd-Warshall 算法. Single-source shortest path 问题不能采取类似的DP,因为其source 确定, 不满足subproblem 的定义.

4 Dynamic Programming

Dynamic Programming 的基础是DAG. 在使用DP解题时,关键在于构建问题的DAG图.也就是要定义子问题结构(node),以及子问题之间的依赖(edge). 定义子问题是解决问题的关键,因此注定是非trivial 的.但是一般而言,有如下规律可循[1]：

1. Input 为$x_1x_2\dots x_n$, subproblem is $x_1x_2\dots x_i$; 如longest increasing sequence
2. Input 为$x_1x_2\dots x_m$, and $y_1y_2\dots y_n$, subproblem is $x_1x_2\dots x_i$ and $y_1y_2\dots y_j$; 如edit distance;
3. Input 为$x_1x_2\dots x_n$, subproblem is $x_i, x_{i+1}, \dots, x_j$; 如Chain Matrix Multiplication.
4. Input 为rooted tree, subproblem is rooted subtree; 如Chain Matrix Multiplication.
5. 在定义子问题时,可以将输入的某些量参数化, shrinking these parameters can get smaller problems. 有可能是一维的,比如Knapsack problem with repetitions；也可能是二维的, 比如Knapsack problem without repetitions. 多维就意味着多变量,而每一个变量都控制着某一方面的信息.
6. 还要时刻考虑问题背后的DAG图,并考虑是否可以与shortest path 或者longest path 相对应,并考虑是否可以generalize the problem.

References

[1] S. Dasgupta, C. Papadimitriou, and U. Vazirani, “Algorithms,” 2006.