




Part10-Graph search and connectivity

 Key videos	https://www.coursera.org/learn/algorithms-graphs-data-structures/lecture/JZRXz/breadth-first-search-bfs-the-basics https://www.coursera.org/learn/algorithms-graphs-data-structures/lecture/ZAAJA/bfs-and-shortest-paths https://www.coursera.org/learn/algorithms-graphs-data-structures/lecture/BTVWh/bfs-and-undirected-connectivity
 Note	Introduction to breadth and depth first search algorithms for graphs and their applications to compute topological orderings and strong connected components of the directed graphs
 Period	@2020/04/20 → 2020/04/23

Note

▼ Overview

SUMMARY: Week 1 is all about graph search and its applications (section X). We'll cover a selection of fundamental primitives for reasoning about graphs. One very cool aspect of this material is that all of the algorithms that we'll cover are insanely fast (linear time with small constants); and, it can be quite subtle to understand why they work! The culmination of these lectures --- computing the strongly connected components of a directed graph with just two passes of depth-first search --- vividly illustrates the point that fast algorithms often require deep insight into the structure of the problem that you're solving. There are also lecture notes for this last topic (available for download underneath the videos). (If you're feeling REALLY motivated, you might read up on Tarjan's earlier algorithm for computing SCCs that needs only a single DFS pass!)

THE VIDEOS: We begin with an overview video, which gives some reasons why you should care about graph search, a general strategy for searching a graph without doing any redundant work, and a high-level introduction to the two most important search strategies, breadth-first search (BFS) and depth-first search (DFS). The second video discusses BFS in more detail, including applications to computing shortest paths and the connected components of an undirected graph. The third video drills down on DFS, and shows how to use it to compute a topological ordering of a directed acyclic graph (i.e., to sequence tasks in a way that respects precedence constraints). The fourth and fifth videos cover the description and analysis, respectively, of the aforementioned algorithm for computing SCCs. A final optional video --- hopefully one of the more fun ones in the course --- discusses the structure of the Web, and lightly

touches on topics like Google's PageRank algorithm and the "six degrees of separation" phenomenon in social networks.

VIDEOS AND SLIDES: Videos can be streamed or downloaded and watched offline (recommended for commutes, etc.). We are also providing PDF lecture slides (typed versions of what's written in the lecture videos), as well as subtitle files (in English and in some cases other languages as well). And if you find yourself wishing that I spoke more quickly or more slowly, note that you can adjust the video speed to accommodate your preferred pace.

THE HOMEWORK: Problem Set #1 should help you solidify your understanding of graph representations and graph search. The programming assignment asks you to implement the SCC algorithm from the lectures, and report your findings about the SCCs of a large graph. Programming Assignment #1 is the most difficult one of the course (and one of the more difficult ones in the entire specialization); as always, I encourage you to use the discussion forums to exchange ideas, tips, and test cases. If you can get through the first week of the course, it should be all downhill from there!

▼ Graph Search - Overview

- 图搜索的应用
 1. 确认网络连通性;
 2. 驾驶方向的判断;
 3. 制定计划 (formulate a plan)
 4. 计算全图的分块 (compute the components of a graph)
- generic graph search 一般图搜索: 即从已知点出发, 寻找任何可以到达的点

Generic Algorithm (given graph G , vertex s)

-- initially s explored, all other vertices unexplored

-- while possible : (if none, halt)

-- choose an edge (u,v) with u explored and v unexplored

-- mark v explored

- 两大类主要搜索算法: BFS与DFS
 - BFS: 以层优先; 可以直接搜索最短路径; 可以直接计算无向图中部件的个数
 - DFS: 一路向前直接搜索; 可以计算导出图的拓扑结构; 可以直接计算有向图中部件的个数

Breadth-First Search (BFS)

- explored nodes in "layers"
- can compute shortest paths
- can compute connected components of an undirected graph

explored / unexplc
Crossing
edges

$O(m+n)$ time
using a queue
(FIFO)

Depth-First Search (DFS)

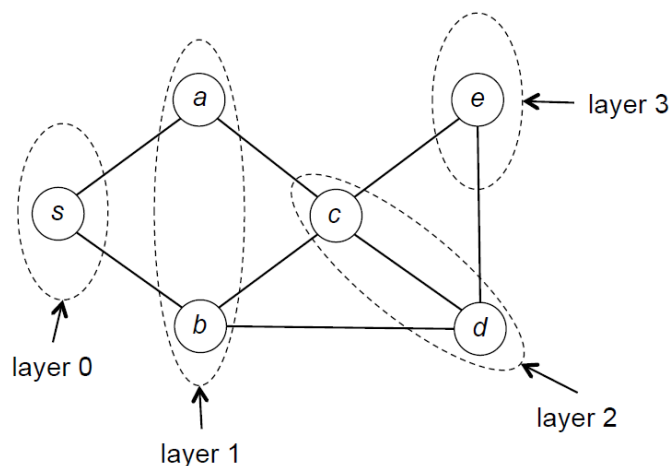
- explore aggressively like a maze, backtrack only when necessary
- compute topological ordering of a directed acyclic graph
- compute connected components in directed graphs

$O(m+n)$ time using a stack (LIFO)
(or via recursion)

▼ Breadth-First Search (BFS): The Basics

特点：以层优先；可以直接搜索最短路径；可以直接计算无向图中部件的个数

- 范例



- 伪代码：关键需要使用queue这种数据结构，对应python的 `queue` 或 `deque` 模块

BFS

Input: graph $G = (V, E)$ in adjacency-list representation, and a vertex $s \in V$.

Postcondition: a vertex is reachable from s if and only if it is marked as “explored.”

```
1 mark  $s$  as explored, all other vertices as unexplored
2  $Q :=$  a queue data structure, initialized with  $s$ 
3 while  $Q$  is not empty do
4     remove the vertex from the front of  $Q$ , call it  $v$ 
5     for each edge  $(v, w)$  in  $v$ 's adjacency list do
6         if  $w$  is unexplored then
7             mark  $w$  as explored
8             add  $w$  to the end of  $Q$ 
```

- BFS基本性质：由于代码每次会遍历到与初始点所有相邻的点，因此 n 与 m 则是对应所有可到达的点与边的数

Claim #1 : at the end of BFS, v explored \iff
 G has a path from s to v .

Reason : special case of the generic algorithm

Claim #2 : running time of main while loop
 $= O(n_s + m_s)$, where $n_s = \#$ of nodes reachable from s
 $m_s = \#$ of edges reachable from s

▼ BFS and Shortest Paths

拓展应用：找到最短路径

- 伪代码：与基本BFS代码的区别，初始化距离（起始点为0，未探索点为无穷大）；增加额外语句（当前仅当边未探索，则距离+1）

Augmented-BFS

Input: graph $G = (V, E)$ in adjacency-list representation, and a vertex $s \in V$.

Postcondition: for every vertex $v \in V$, the value $l(v)$ equals the true shortest-path distance $dist(s, v)$.

```
1 mark  $s$  as explored, all other vertices as unexplored
2  $l(s) := 0, l(v) := +\infty$  for every  $v \neq s$ 
3  $Q :=$  a queue data structure, initialized with  $s$ 
4 while  $Q$  is not empty do
5     remove the vertex from the front of  $Q$ , call it  $v$ 
6     for each edge  $(v, w)$  in  $v$ 's adjacency list do
7         if  $w$  is unexplored then
8             mark  $w$  as explored
9              $l(w) := l(v) + 1$ 
10            add  $w$  to the end of  $Q$ 
```

▼ BFS and Undirected Connectivity

拓展应用：探索无向图连通性

- 问题构成：输入-无向图 $G(V, E)$ ；输出- G 的联通部件
- 伪代码：遍历所有的点集，然后一个个进行BFS，如果当前点不在已有的connected component里，则numCC+1

UCC

Input: undirected graph $G = (V, E)$ in adjacency-list representation, with $V = \{1, 2, 3, \dots, n\}$.

Postcondition: for every $u, v \in V$, $cc(u) = cc(v)$ if and only if u, v are in the same connected component.

mark all vertices as unexplored

$numCC := 0$

```

for  $i := 1$  to  $n$  do           // try all vertices
    if  $i$  is unexplored then      // avoid redundancy
         $numCC := numCC + 1$       // new component
        // call BFS starting at  $i$  (lines 2-8)
         $Q :=$  a queue data structure, initialized with  $i$ 
        while  $Q$  is not empty do
            remove the vertex from the front of  $Q$ , call it  $v$ 
             $cc(v) := numCC$ 
            for each  $(v, w)$  in  $v$ 's adjacency list do
                if  $w$  is unexplored then
                    mark  $w$  as explored
                    add  $w$  to the end of  $Q$ 

```

- 复杂度简单分析：n（所有点）+m（调用BFS时的复杂度）

-- initialize all nodes as unexplored $O(n)$

[assume labelled 1 to n]

-- for $i = 1$ to n $O(n)$

-- if i not yet explored [in some previous BFS]

-- BFS(G, i) [discovers precisely i 's connected component]

Note : finds every connected component.

Running time : $O(m+n)$

$O(1)$ per node

$O(1)$ per edge in each BFS

▼ Depth-First Search (DFS): The Basics

- 特点
 1. explore aggressively, only backtrack when necessary
 2. compute a topological ordering of a directed acyclic graph 计算有向无环图的结构
 3. compute strongly connected components of directed graphs
- 伪代码，基于BFS两大改动
 1. swap in a stack data structure (which is last-in first-out) for the queue (which is first-in first-out);
 2. postpone checking whether a vertex has already been explored until after removing it from the data structure
- 递归版：每次相对搜索更小的图结构

DFS (Recursive Version)

Input: graph $G = (V, E)$ in adjacency-list representation, and a vertex $s \in V$.

Postcondition: a vertex is reachable from s if and only if it is marked as “explored.”

```
// all vertices unexplored before outer call
mark  $s$  as explored
for each edge  $(s, v)$  in  $s$ 's adjacency list do
    if  $v$  is unexplored then
        DFS ( $G, v$ )
```

- 性质：在计算算法复杂度中（由于可能回溯操作，所以至多可能两次访问边）

▼ Topological Sort

通过DFS可以计算出有向图中不同的topological orderings

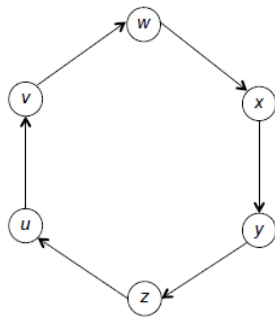
- 定义

Topological Orderings

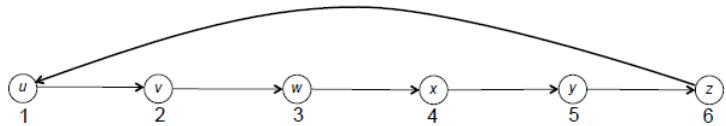
Let $G = (V, E)$ be a directed graph. A *topological ordering* of G is an assignment $f(v)$ of every vertex $v \in V$ to a different number such that:

for every $(v, w) \in E$, $f(v) < f(w)$.

- 基础：不是所有图都有拓扑顺序（a graph consisting solely of a directed cycle doesn't have topological orderings）；当且仅当有向无循环图（a directed acyclic graph, or simply a DAG）存在



(a) A directed cycle



(b) A non-topological ordering

- 相关定理

源节点：A **source vertex** of a directed graph is a vertex with no incoming edges.

⇒ Lemma (**Every DAG Has a Source**) Every directed acyclic graph has at least one source vertex.

⇒ Theorem (**Every DAG Has a Topological Ordering**) Every directed acyclic graph has at least one topological ordering.

- 问题描述

Problem: Topological Sort

Input: A directed acyclic graph $G = (V, E)$.

Output: A topological ordering of the vertices of G .

- 伪代码：相当于一个从大到小给label的算法，然后一步步进行找outgoing的边标记

Input: directed acyclic graph $G = (V, E)$ in adjacency-list representation.

Postcondition: the f -values of vertices constitute a topological ordering of G .

```
mark all vertices as unexplored
curLabel := |V|           // keeps track of ordering
for every  $v \in V$  do
    if  $v$  is unexplored then // in a prior DFS
        DFS-Topo ( $G, v$ )
```

DFS-Topo

Input: graph $G = (V, E)$ in adjacency-list representation, and a vertex $s \in V$.

Postcondition: every vertex reachable from s is marked as “explored” and has an assigned f -value.

```
mark  $s$  as explored
for each edge  $(s, v)$  in  $s$ 's outgoing adjacency list do
    if  $v$  is unexplored then
        DFS-Topo ( $G, v$ )
 $f(s) := curLabel$  //  $s$ 's position in ordering
 $curLabel := curLabel - 1$  // work right-to-left
```

- 推论

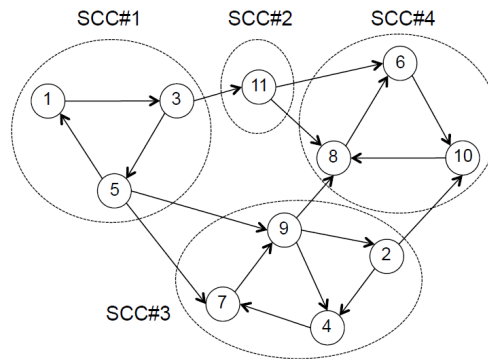
Theorem 8.8 (Properties of TopoSort) For every directed acyclic graph $G = (V, E)$ in adjacency-list representation:

- At the conclusion of *TopoSort*, every vertex v has been assigned an f -value, and these f -values constitute a topological ordering of G .
- The running time of *TopoSort* is $O(m + n)$, where $m = |E|$ and $n = |V|$.

▼ Computing Strong Components: The Algorithm

- 强连通分量定义与示意：

A strongly connected component or SCC of a directed graph is a maximal subset S from V of vertices such that there is a directed path from any vertex in S to any other vertex in S



- Kosaraju算法步骤：使用两次DFS：先在 G^{rev} 进行DFS后得到逻辑排序；然后根据排序一次对 G 进行DFS

Kosaraju (High-Level)

1. Let G^{rev} denote the input graph G with the direction of every edge reversed.
 2. Call DFS from every vertex of G^{rev} , processed in arbitrary order, to compute a position $f(v)$ for each vertex v .
 3. Call DFS from every vertex of G , processed from highest to lowest position, to compute the identity of each vertex's strongly connected component.
- 伪代码

Kosaraju

Input: directed graph $G = (V, E)$ in adjacency-list representation, with $V = \{1, 2, 3, \dots, n\}$.

Postcondition: for every $v, w \in V$, $scc(v) = scc(w)$ if and only if v, w are in the same SCC of G .

```

 $G^{rev} := G$  with all edges reversed
mark all vertices of  $G^{rev}$  as unexplored

// first pass of depth-first search
// (computes  $f(v)$ 's, the magical ordering)
TopoSort ( $G^{rev}$ )

// second pass of depth-first search
// (finds SCCs in reverse topological order)
mark all vertices of  $G$  as unexplored
numSCC := 0 // global variable
for each  $v \in V$ , in increasing order of  $f(v)$  do
    if  $v$  is unexplored then
        numSCC := numSCC + 1
        // assign scc-values (details below)
        DFS-SCC ( $G, v$ )

```

DFS-SCC

Input: directed graph $G = (V, E)$ in adjacency-list representation, and a vertex $s \in V$.

Postcondition: every vertex reachable from s is marked as “explored” and has an assigned scc -value.

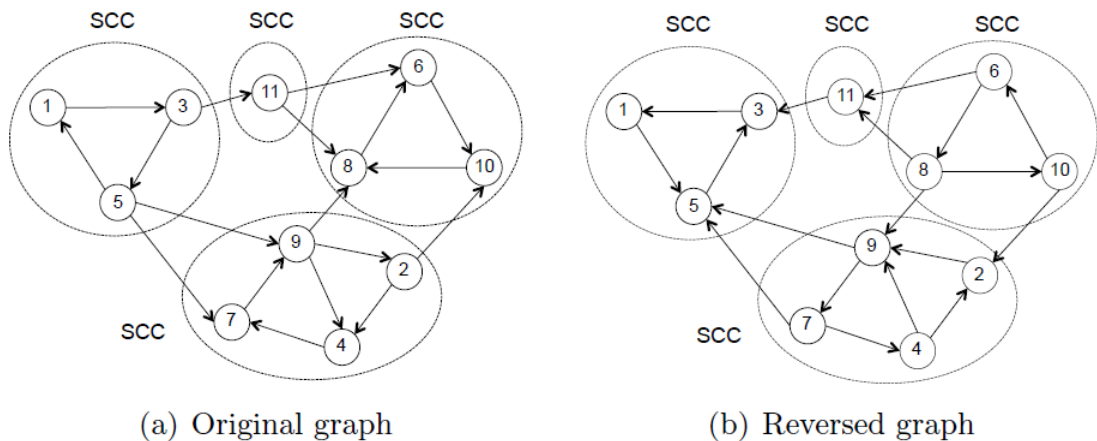
```

mark  $s$  as explored
scc( $s$ ) := numSCC // global variable above
for each edge  $(s, v)$  in  $s$ 's outgoing adjacency list do
    if  $v$  is unexplored then
        DFS-SCC ( $G, v$ )

```

▼ Computing Strong Components: The Analysis

- 样例观察



- 相关性质：在上一节所计算的伪代码中的scc对于两个点而言，当且仅当属于同个SCC时相等

Theorem 8.12 (Properties of Kosaraju) *For every directed graph $G = (V, E)$ in adjacency-list representation:*

- (a) *At the conclusion of Kosaraju, for every pair v, w of vertices, $scc(v) = scc(w)$ if and only if v and w belong to the same strongly connected component of G .*

▼ Problem Set #1

1. Given an adjacency-list representation of a directed graph, where each vertex maintains an array of its outgoing edges (but *not* its incoming edges), how long does it take, in the worst case, to compute the in-degree of a given vertex? As usual, we use n and m to denote the number of vertices and edges, respectively, of the given graph. Also, let k denote the maximum in-degree of a vertex. (Recall that the in-degree of a vertex is the number of edges that enter it.)

- ☐ $\theta(n)$
- ☐ Cannot determine from the given information
- ☐ $\theta(m)$
- ☐ $\theta(k)$

C

Correct

Without explicitly maintaining a list of incoming edges, you might have to scan all the edges to identify the incoming arcs.

Consider the following problem: given an undirected graph G with n vertices and m edges, and two vertices s and t , does there exist at least one s - t path?

If G is given in its adjacency list representation, then the above problem can be solved in $O(m + n)$ time, using BFS or DFS. (Make sure you see why this is true.)

Suppose instead that G is given in its adjacency *matrix* representation. What running time is required, in the worst case, to solve the computational problem stated above? (Assume that G has no parallel edges.)

- ☐ $\theta(m + n \log n)$
- ☐ $\theta(n * m)$
- ☐ $\theta(m + n)$
- ☐ $\theta(n^2)$

D

Correct

For the lower bound, observe that you might need to look at every entry of the adjacency matrix (e.g., if it has only one "1" and the rest are zeroes). One easy way to prove the upper bound is to first build an adjacency list representation (in $\Theta(n^2)$ time, with a single scan over the given adjacency matrix) and then run BFS or DFS as in the video lectures.

This problem explores the relationship between two definitions about graph distances. In this problem, we consider only graphs that are undirected and connected. The *diameter* of a graph is the maximum, over all choices of vertices s and t , of the shortest-path distance between s and t . (Recall the shortest-path distance between s and t is the fewest number of edges in an s - t path.)

Next, for a vertex s , let $l(s)$ denote the maximum, over all vertices t , of the shortest-path distance between s and t . The *radius* of a graph is the minimum of $l(s)$ over all choices of the vertex s .

Which of the following inequalities always hold (i.e., in every undirected connected graph) for the radius r and the diameter d ? [Select all that apply.]

- ☐ $r \leq d$
- ☐ $r \leq d/2$
- ☐ $r \geq d/2$
- ☐ $r \geq d$

AC

Correct

By the definitions, $l(s) \leq d$ for every single choice of s .

Correct

Let cc minimize $l(s)$ over all vertices s . Since every pair of vertices s and t have paths to cc with at most r edges, stitching these paths together yields an s - t path with only $2r$ edges; of course, the shortest s - t path is only shorter.

Consider our algorithm for computing a topological ordering that is based on depth-first search (i.e., NOT the "straightforward solution"). Suppose we run this algorithm on a graph G that is NOT directed acyclic. Obviously it won't compute a topological order (since none exist). Does it compute an ordering that minimizes the number of edges that go backward?

For example, consider the four-node graph with the six directed edges $(s, v), (s, w), (v, w), (v, t), (w, t), (t, s)$. Suppose the vertices are ordered s, v, w, t . Then there is one backwards arc, the (t, s) arc. No ordering of the vertices has zero backwards arcs, and some have more than one.

- ☐ Sometimes yes, sometimes no
- ☐ Always
- ☐ Never
- ☐ If and only if the graph is a directed cycle

B

Incorrect

There are counterexamples. Can you find one?

D

Incorrect

The algorithm might get lucky even if the graph is not a directed cycle.

A

Correct

On adding one extra edge to a directed graph G , the number of strongly connected components...?

- ☐ ...never decreases (no matter what G is)
- ☐ ...never decreases by more than 1 (no matter what G is)
- ☐ ...could remain the same (for some graphs G)
- ☐ ...will definitely not change (no matter what G is)

C

Correct

For example, the graph might already be strongly connected.

▼ Programming Assignment #1

hibetterheyj/Note-Everyday

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or window. Reload to refresh your session. Reload to refresh your session.

https://github.com/hibetterheyj/Note-Everyday/tree/master/Slides_Note/Stanford_Algorithms/PythonCode/Course2/Week1



Reference

• Graph traversal 图遍历

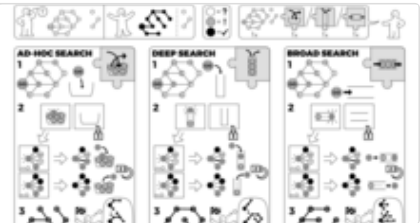
图的遍历

W <https://zh.wikipedia.org/wiki/%E5%9B%BE%E7%9A%84%E9%81%8D%E5%8E%86>

Graph traversal

In computer science, graph traversal (also known as graph search) refers to the process of visiting (checking and/or updating) each vertex in a graph. Such traversals are classified by the order in which the

W https://en.wikipedia.org/wiki/Graph_traversal



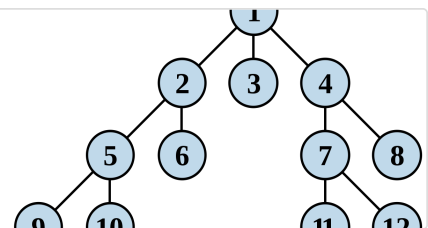
<https://cs.stanford.edu/people/abisee/gs.pdf>

• Breadth-first search 广度优先搜索

广度优先搜索

广度优先搜索算法（英語：Breadth-First Search，縮寫為BFS），又譯作寬度優先搜索，或橫向優先搜索，是一種圖形搜索演算法。簡單的說，BFS是從根節點開始，沿着樹的寬度遍歷樹的節點。如果所有節點均被訪

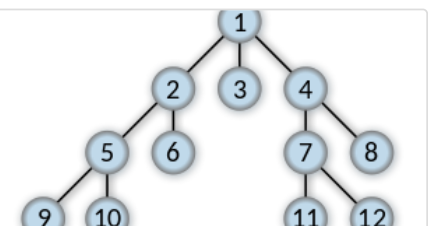
W <https://zh.wikipedia.org/wiki/%E5%B9%BF%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2>



Breadth-first search

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'), and explores

W https://en.wikipedia.org/wiki/Breadth-first_search



• 实现方法

Simple BFS (Breadth-First Search) for graph in Python

Simple BFS (Breadth-First Search) for graph in Python - bfs.py

 <https://gist.github.com/zenja/8070849>

 **GitHub** Gist

力扣

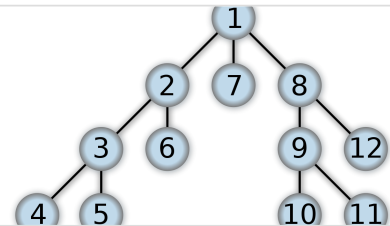
https://leetcode-cn.com/problems/find-largest-value-in-each-tree-row/discuss/167898/python-BFS-using-deque/?utm_source=LCUS&utm_medium=ip_redirect_q_uns&utm_campaign=transfer2china

• 深度优先搜索

深度优先搜索

深度优先搜索算法（英語：Depth-First-Search，DFS）是一种用于遍历或搜索 树或 图的 算法。这个算法会尽可能深的搜索树的分支。当节点v的所有边都被探寻过，搜索将回溯到发现节点v的那条边的起始节点。这一

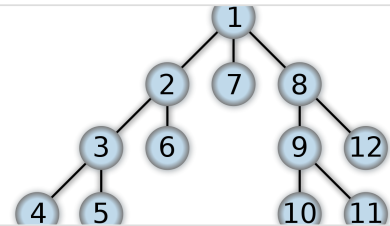
W <https://zh.wikipedia.org/wiki/%E6%B7%B1%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2>



Depth-first search

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph)

W https://en.wikipedia.org/wiki/Depth-first_search

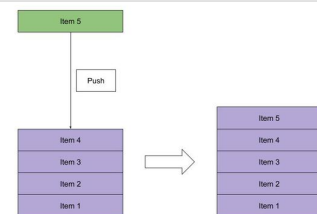


• 队列（queue）与堆栈（stack）

Python经典面试题: 用3种方法实现堆栈和队列并示例实际应用场景

数据结构在计算机中组织存储，以便我们可以有效地访问和更改数据。堆栈和队列 是计算机科学中定义的最早的数据结构。遵循后进先出 (Last-in-First-Out LIFO)原则。遵循先入先出(FIFO:First-in-First-Out)原则。Python的内置

 <https://www.jianshu.com/p/c990427ca608>



▼ Stacks and Queues using Python

<https://www.101computing.net/stacks-and-queues-using-python/>

▼ Using List as Stack and Queues in Python

Using List as Stack and Queues in Python - GeeksforGeeks

Prerequisite : list in Python The concept of Stack and Queue is easy to implement in Python. Stack works on the principle of "Last-in, first-out". Also, the inbuilt functions in Python make the code short

🔗 <https://www.geeksforgeeks.org/using-list-stack-queues-python/>



▼ Stack and Queue in Python using queue Module

Stack and Queue in Python using queue Module - GeeksforGeeks

A simple python List can act as queue and stack as well. Queue mechanism is used widely and for many purposes in daily life. A queue follows FIFO rule(First In First Out) and is used in programming for sorting and for many

🔗 <https://www.geeksforgeeks.org/stack-queue-python-using-module-queue/>



<https://www.geeksforgeeks.org/stack-queue-python-using-module-queue/>

- 他人笔记

<https://www.dazhuanlan.com/2019/10/07/5d9b3b582dbaf/>

<https://www.dazhuanlan.com/2019/10/07/5d9b3b582dbaf/>

- Kosaraju's algorithm

Kosaraju's algorithm

In computer science, Kosaraju's algorithm (also known as the Kosaraju-Sharir algorithm) is a linear time algorithm to find the strongly connected components of a directed graph. Aho, Hopcroft and Ullman credit it to S. Rao Kosaraju and Micha Sharir. Kosaraju suggested it in 1978 but did not publish it, while Sharir independently

🌐 https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm

Tarjan算法

此算法以一個有向圖作為輸入，並按照所在的強連通分量給出其頂點集的一個劃分。圖中的每個節點只在一個強連通分量中出現，即使是在有些節點單獨構成一個強連通分量的情況下（比如圖中出現了樹形結構或孤立節點）。算法的基本思想如下：任選一節點開始進行深度優先搜索（若深度優先搜索結束後仍有未訪問的節點，則再從中任選一點再次進

🌐 <https://zh.wikipedia.org/wiki/Tarjan%E7%AE%97%E6%B3%95>