Basic algorithms and special graphs

31.10.2019
Here, algorithm and *computer program* will be used interchangeable.

An algorithm (program) $A$ takes

- input $I$, i.e. a graph $G = (V, E)$
- runs steps according to program
- and either
  - stops with some output $O$
  - or runs forever without terminating

Thus, $A$ computes a partial function $I \mapsto O$
Algorithms

Runtime of algorithms is analysed by their scaling behavior:

▶ Let \( n \) denote the “size” of the input (in some suitable definition).

▶ Then growth rate is considered: \( O(f(n)) \)

\[
g(n) \in O(f(n)) \iff \lim_{n \to \infty} \frac{g(n)}{f(n)} < \infty
\]

\( O(f(n)) \): Set of functions that grow at most as fast as \( f \).
Algorithms

Runtime of algorithms is analysed by their scaling behavior:

- Let \( n \) denote the “size” of the input (in some suitable definition).
- Then growth rate is considered: \( O(f(n)) \)

\[
g(n) \in O(f(n)) \iff \lim_{n \to \infty} \frac{g(n)}{f(n)} < \infty
\]

\( O(f(n)) \): Set of functions that grow at most as fast as \( f \).

Example: The size of a graph \( G = (V, E) \):

- \( N \) vertices and \( M \) edges
- \( M \in O(N^2) \)

Note: For sparse graphs, \( M \) might grow much slower and thus, a more detailed analysis often uses \( \text{size}(G) \in O(N + M) \).
**P vs NP**

Important complexity classes in computer science:

- **P** Set of all problems that can be solved by polynomial-time algorithm, i.e. run-time $O(n^d)$ for some $d$.

- **NP** Set of all problems where correctness of solution can be checked in polynomial time.

Open problem in computer science

$$P = NP?$$

- **NP** problems can be solved in exponential time:

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>1.000</td>
<td>1.024</td>
</tr>
<tr>
<td>20</td>
<td>400</td>
<td>8.000</td>
<td>1.048.576</td>
</tr>
<tr>
<td>30</td>
<td>900</td>
<td>27.000</td>
<td>1.073.741.824</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>10.000</td>
<td>1.000.000</td>
<td>1.267.650.600.228.229.401.496.703.205.376 $\approx 10^{30}$</td>
</tr>
</tbody>
</table>

- Widely believed that $P \neq NP$
Graph algorithms

Classical graph problems

1. Eulerian path: Does there exist a path visiting all edges exactly once?
2. Hamiltonian path: Does there exist a path visiting all vertices exactly once?

Also known as “travelling salesman problem”
Graph algorithms

Classical graph problems

1. Eulerian path: Does there exist a path visiting all edges exactly once?
2. Hamiltonian path: Does there exist a path visiting all vertices exactly once?
   Also known as “travelling salesman problem”

1. Euler’s proof gives polynomial algorithm with runtime $O(N^2)$. Why?
2. The Hamilton path problem is proved to be $NP$-complete, i.e. ∈ $NP$ and as difficult as any $NP$-problem.
   Best known algorithm: $O(1.657^n)$
   Note: Checking correctness of solution is easy!
Degree sequence

Write an algorithm that

- takes a graph $G = (V, E)$ as input, e.g. as an adjacency matrix $A$
- and outputs the degree sequence of $G$
Degree sequence

Write an algorithm that

▶ takes a graph \( G = (V, E) \) as input, e.g. as an adjacency matrix \( A \)

▶ and outputs the degree sequence of \( G \)

Implementation in R:

```r
degree.seq <- function (A) {
  N <- nrow(A);
  degs <- vector(length = N);
  for (i in 1:N) {
    tmp <- 0;
    for (j in 1:ncol(A)) {
      tmp <- tmp + A[i, j];
    }
    degs[i] <- tmp/2;
  }
  degs
}
```
Degree sequence

def degree.seq <- function (A) {
    N <- nrow(A); # O(1)
degs <- vector(length = N); # O(N)
    for (i in 1:N) {
        tmp <- 0; # O(1)
        for (j in 1:ncol(A)) {
            tmp <- tmp + A[i, j]; # O(1)
        }
        degs[i] <- tmp/2; # O(1)
    }
    degs # O(1)
}

Total runtime:

\[O(1) + O(N) + N \cdot (O(1) + N \cdot O(1) + O(1)) + O(1) = O(N^2)\]

- Growth rate only consider fastest growing terms, e.g.

\[O(N) + O(N^2) = O(N^2)\]

- Know runtime of basic operations, i.e. vector access

- Could you do better?
Shortest paths and breadth-first search

How to find shortest paths?

*Breadth-first search:*

**Init:** Initialize queue $Q = [s]$ and distances

$$d(v) = \begin{cases} 0 & v = s \\ \infty & \text{else} \end{cases}$$

**Step:** While $Q$ is not empty:

- Remove $v$ from queue
- For all neighbors $w$ of $v$ do:
  - If $d(w) = \infty$:
    - Set $d(w) = d(v) + 1$ and append $w$ onto $Q$
Shortest paths and breadth-first search

How to find shortest paths?

_Breadth-first search:_

**Init:** Initialize queue $Q = [s]$ and distances

$$
    d(v) = \begin{cases} 
    0 & v = s \\
    \infty & \text{else}
    \end{cases}
$$

**Step:** While $Q$ is not empty:
- Remove $v$ from queue
- For all neighbors $w$ of $v$ do:
  - If $d(w) = \infty$:
    - Set $d(w) = d(v) + 1$ and append $w$ onto $Q$

- Single run of the algorithm: Finds shortest (geodesic) distance from a source vertex $s$ to every _other_ vertex in the same component of the network

- In a second step the algorithm can also be used to compute shortest paths by constructing the so-called _shortest path tree_
Shortest paths and breadth-first search

Runtime of breadth-first search:

- Initialization: $O(N)$ for distance vector
- Main loop:
  - Every vertex enqueued at most once
    $\Rightarrow$ while-loop runs $N$ times
  - Loop over all neighbours of $v$: $O(M)$ (worst case)
  - Update distances and enqueue: $O(1)$
- Total runtime

\[
O(N) + N \cdot O(M) = O(NM) \subseteq O(N^3)
\]

Closer inspection reveals, every vertex and edge needs to be investigated only once:

- Best complexity achieved with clever data structures

\[
O(N + M) \subseteq O(N^2)
\]

- Linear in the size of the graph!
Special graphs
Complete graph

The *complete graph* on \( n \) vertices is denoted by \( K_n \):

- \( K_n \) is undirected
- \( K_n \) contains all possible edges, i.e. \( K_n \) has \( \frac{n(n-1)}{2} \) edges
- Every vertex of \( K_n \) has degree \( n - 1 \)

\( K_n \) is an example of a *regular graph* where each vertex has the same degree.
Cliques

*Maximal* subset of vertices in an undirected network, in which every vertex is connected to every other vertex.

Alternatively: *Maximal* complete subgraph of order $n$.
$k$-plex

**Maximal** subset of $n$ vertices in an undirected network, in which every vertex is connected to at least $n - k$ of the other vertices in the subset.
Alternatively: **Maximal** connected subgraph of order $n$ in which every vertex has degree $n - k$.

- Weaker requirement than for cliques.
- 1-plex is a clique.
- As cliques, $k$-plexes can overlap.
**k-core and k-plex**

**k-plex**: *Maximal* subset of $n$ vertices in an undirected network, in which every vertex is connected to at least $n - k$ of the other vertices in the subset. Alternatively: *Maximal* connected subgraph of order $n$ in which every vertex has at least degree $n - k$.

**k-core**: *Maximal* subset of $n$ vertices in an undirected network, in which every vertex is connected to at least $k$ of the other vertices in the subset. Alternatively: *Maximal* connected subgraph of order $n$ in which every vertex has at least degree $k$. 
k-core

- k-cores are nested.
- Simple algorithm to find all k-cores in a network:
  - Repeatedly remove all vertices with degree less than k.
  - Connected components of remaining graph are k-cores.
- Here:
  - 2-core: whole network
  - 3-core: \{1, 2, 3, 4, 5, 6, 7, 8\}
  - 4-core: none
Trees

A cycle is a closed path of length $r \geq 1$, i.e. $v_1 - v_2 - \ldots - v_r$ with $v_1 = v_r$ and no repeated edges.

A tree is a connected and acyclic (cycle-free) graph.
Trees

- A collection of trees is called a *forest*.
- Trees play an important role for random graph models.
- In a tree, there is exactly one path between any pair of vertices.
- A tree of $n$ vertices always has exactly $n - 1$ edges.
- Any connected network with $n$ vertices and $n - 1$ edges is a tree.
A *planar graph* is a graph that can be drawn on a 2D-plane without having any edges cross.

Examples:
- Trees
- Road networks (approximately)
- Power grids (approximately)
- Shared borders between countries, etc.
Planar networks - Four-color theorem

In mathematics, the four color theorem, or the four color map theorem, states that, given any separation of a plane into contiguous regions, producing a figure called a map, no more than four colors are required to color the regions of the map so that no two adjacent regions have the same color. Two regions are called adjacent if they share a common boundary that is not a corner, where corners are the points shared by three or more regions.

Source: https://en.wikipedia.org/wiki/Four_color_theorem
Power Grid expansion optimisation

Expansion condition: planar graph
Bipartite networks

Often a system can be represented as a network consisting of two kinds of vertices, with edges between vertices of different types only (group membership). Examples:

- Film actors: Actors, group: Cast of a film
- Coauthorship: Authors, group: Authors of an article
- Rail connections: Train stations, group: Route
- Brazilian soccer players: Players, group: Clubs
- Amazon: Products, group: Customers of a product
Bipartite networks: Adjacency matrix

\[ A = \begin{pmatrix}
0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\end{pmatrix} \]
Bipartite networks: Incidence matrix $\mathbf{B}$

Vertices of type 1: $i = 1, 2, \ldots, N$ (often groups)
Vertices of type 2: $j = 1, 2, \ldots, M$ (often people)

$$B_{ij} = \begin{cases} 
1 & \text{if there is an edge between vertices } i \text{ and } j \\
0 & \text{otherwise.}
\end{cases}$$

$$\mathbf{B} = \begin{pmatrix} 
1 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 \\
\end{pmatrix}$$

$N \times M$ matrix
Bipartite networks: One-mode projections

Projection to a (weighted) network with vertices of the second type only:

\[ P_{ij} = \sum_{k=1}^{N} B_{ki} B_{kj}. \]

That is \( P = B^T B \). Note:

\[ P_{ii} = \sum_{k=1}^{N} B_{ki} B_{ki} = \sum_{k=1}^{N} B_{ki}. \]

Adjacency matrix \((M \times M)\):

\[
A_{ij} = \begin{cases} 
\min\{1, P_{ij}\} & \text{if } i \neq j \\
0 & \text{if } i = j.
\end{cases}
\]
Bipartite networks: One-mode projections

Two one-mode projections based on

\[ P = \mathbf{B}^T \mathbf{B} \quad , \quad P' = \mathbf{B} \mathbf{B}^T. \]
Bipartite networks - Example: Amazon

Amazon collects data on

Customer $c_i$ bought product $p_j$

- Bipartite network with two types of nodes:
  1. Customers $c_1, \ldots, c_N$
  2. Products $p_1, \ldots, p_M$

Product co-purchasing network:
- Product $p_i$ purchased together with product $p_j$
- Corresponds to one-mode projection of above network
- Used for product recommendation
  Note: Ignores information about customers
Accessing edge labels with the incidence matrix

For a given network one can consider the edges as one type of vertices of a corresponding bipartite network, with the original vertices representing the second type.

Useful for directed networks, where heads and tails of directed edges are represented in the incidence matrix by $1$ and $-1$, respectively.

$$
B = \begin{pmatrix}
-1 & 1 & 0 & 0 \\
0 & -1 & 1 & 0 \\
0 & -1 & 0 & 1 \\
0 & 0 & -1 & 1 \\
\end{pmatrix}
$$
Directed acyclic graph

- Directed acyclic graph (DAG)
- Examples: power flow in an electricity grid, citation network of papers
- Topological ordering: For every directed edge $i \rightarrow j$, vertex $i$ comes before $j$ in the ordering: $(1, 2, 3, 4, 6, 9, 10, 11, 12, 8, 7, 5, 13)$
- With a topological ordering, the adjacency matrix of an acyclic directed network is strictly triangular
Directed acyclic graph

Consider a (strict) partial order $<$, i.e.

1. Irreflexive: $x \not< x \quad \forall x$
2. Transitive: $x < y \land y < z \implies x < z \quad \forall x, y, z$
3. Asymmetric: $x < y \implies y \not< x \quad \forall x, y$

Close relation to DAGs

- Every partial order is a DAG, i.e. $x < y \iff x \rightarrow y$.
- *Transitive closure* of a DAG is a partial order, i.e.

  $$x < y \iff \text{there is a path from } x \text{ to } y$$
Scheduling

Consider the following cooking recipe:\(^1\):

1. Melt 115g dark chocolate, 7g butter and 30 ml cream in a double boiler
2. Whisk 2 egg yolks into the mixture
3. Whisk 2-3 egg whites with a dash cream of tatar to soft peaks
4. Add 35g sugar and whisk to stiff peaks
5. Fold into chocolate mixture
6. Pour into ramekins and bake for 15 minutes at 190\(^\circ\)C

This can be represented as a DAG:

- What does an edge represent?
- Why is a topological ordering useful?
- Think of other applications . . .

\(^1\)From http://www.cookingforengineers.com/recipe/160/Dark-Chocolate-Souffle