Finding Graph Patterns

Theory, Techniques, and Applications for Community Detection

Andrea Marino

PhD Course on Graph Mining Algorithms,
Università di Pisa

February, 2018

- Finding all the communities in a network.
- Finding all the (eventually shortest) paths from a source to a target
- In general,

Finding all the subgraphs satisfying a certain property.

Useful also in machine learning

Comparing two (sub)graphs via graph kernels

1. Determine the frequency of some subgraph patterns.
2. Apply some function to combine these frequencies:
   - often feature vectors for machine learning.

For instance, considering labeled networks.

- Walks, i.e. counting common walks.
- Shortest paths.
- Subtree-like patterns:
  - Weisfeiler-Lehman transform belongs to this class.
- Simple cycles.
- Subgraphs of limited size k, called graphlets.
- Paths of fixed length.

Part I

Theory

- Producing all the solutions of a problem is called "Enumeration" or "Listing".
- In theory, the number of solutions of a problem could be exponential
  - the algorithms are in the worst case exponential.
- New definitions of efficiency.

Worst case overall time

Bounding the total time with a function depending on the input size.

- Usually bounds of the following kind, for some constant c

$$O(c^n)$$

- These bounds do not depend on the number of solutions.
- In the case of Tomita et al. for maximal clique enumeration, they consider the graph with n nodes with maximum number of cliques as the worst case.

Often, we have less solutions than the worst case and the number of solutions is variable, from exponential to polynomial, e.g st-paths in a graph vs tree.
This motivates output sensitive enumeration.

Total Time

Bounding the total time with a function depending on the input size and the number of solutions.

- A listing algorithm runs in polynomial total time if its time complexity is polynomial in the input size and the output size. That is:

$$O(\alpha^h n^k)$$

with h, k constants, α is the number of solutions, n is the size of the instance

R.E. Tarjan: Enumeration of the elementary circuits of a directed graph.SIAM Journal on Computing (1973)

Enumerability

Measuring the average cost per solution.

- A listing algorithm P-enumerates the solutions of a relation R if the time required to output them is bounded by $p(n)\alpha$, where $p(n)$ is a polynomial in the input size n and α is the number of solutions to output. That is:

$$O(\alpha n^k)$$

L.G. Valiant: The complexity of computing the permanent. Theoretical Computer Science (1979)

Delay

Measuring the delay between consecutive solutions.

A listing algorithm has polynomial delay if it lists all solutions one after the other in some order, in such a way that:

- it takes polynomial time in the input size before producing the first solution or halting.
- after returning any solution, it takes polynomial time before producing another solution or halting.

D.S. Johnson, M. Yannakakis, and C.H. Papadimitriou.: On generating all maximal independent sets. Information Processing Letters (1988)

Part II

Techniques

What should our algorithm guarantee?

This is the Usual Correctness Theorem.

Theorem (CORRECTNESS PROTOTYPE)

- *All the solutions are listed*
- *Just the solutions are listed*
- *There are no duplicated solutions*

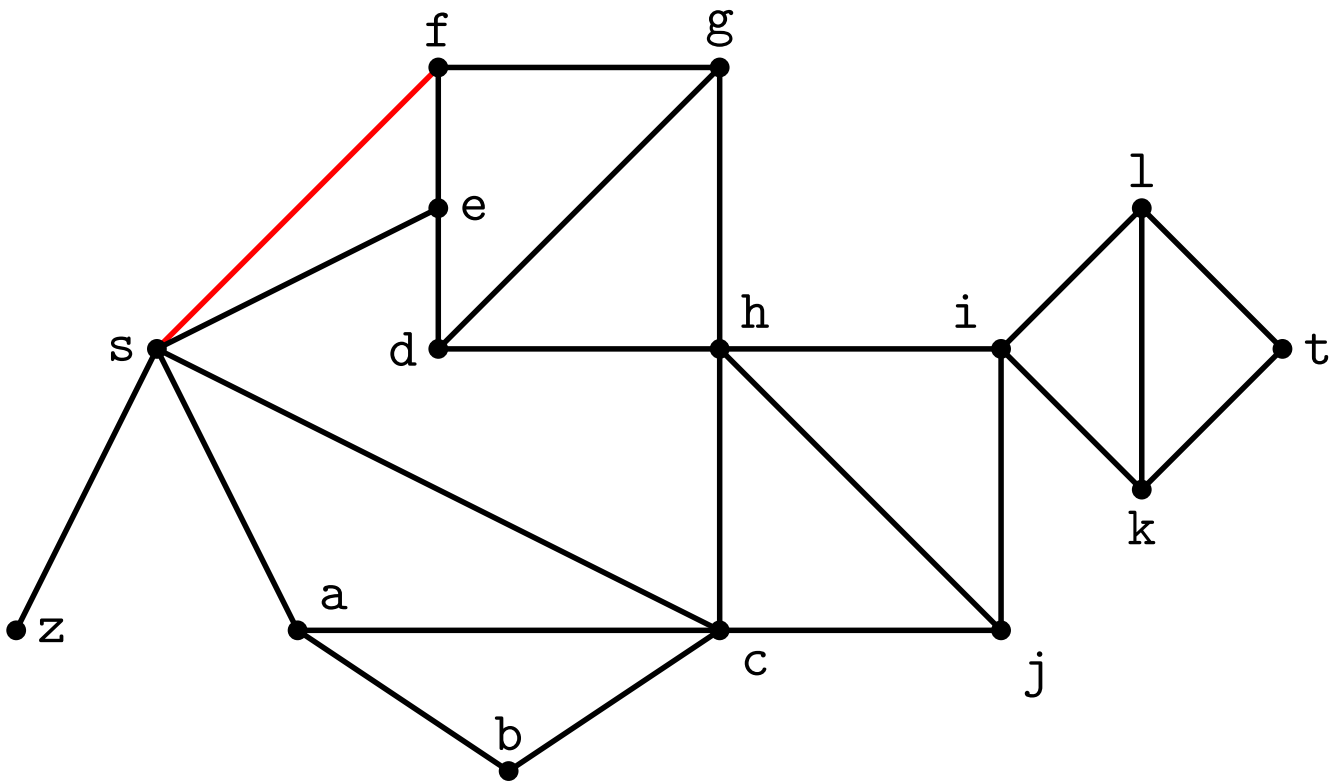
Brute Force

- **Generate** a new solution Y enlarging a partial solution X and check whether Y has been already generated.
- The **Generation** process could lead to duplication.
- Maintaining all the solutions in a database.

Recursive Binary Partition Scheme

Example: binary partition of the paths.

Paths from s to t, i.e. $\textsc{Paths}(G, s, t)$, are either the paths using the edge (s, f) or the ones not using that edge.



To solve $\textsc{Paths}(G, s, t)$, we reduce to $\textsc{Paths}(G - e, s, t)$ and $\textsc{Paths}(G - s, f, t)$

Recursive Binary Partition Scheme

Binary partition checking for useful calls.

Algorithm 1: $\textsc{Paths}(G, s, t, S)$

Input: A graph G, the vertices s and t, a sequence of vertices S
Output: All the paths from s to t in G
if $s = t$ **then** output S; **return**;
choose an arc $e = (s, f)$
if $\exists(s, t)$-*path in* $G - e$ **then** $\textsc{Paths}(G - e, s, t, S)$;
if $\exists(r, t)$-*path in* $G - s$ **then** $\textsc{Paths}(G - s, f, t, \langle S, s\rangle)$;

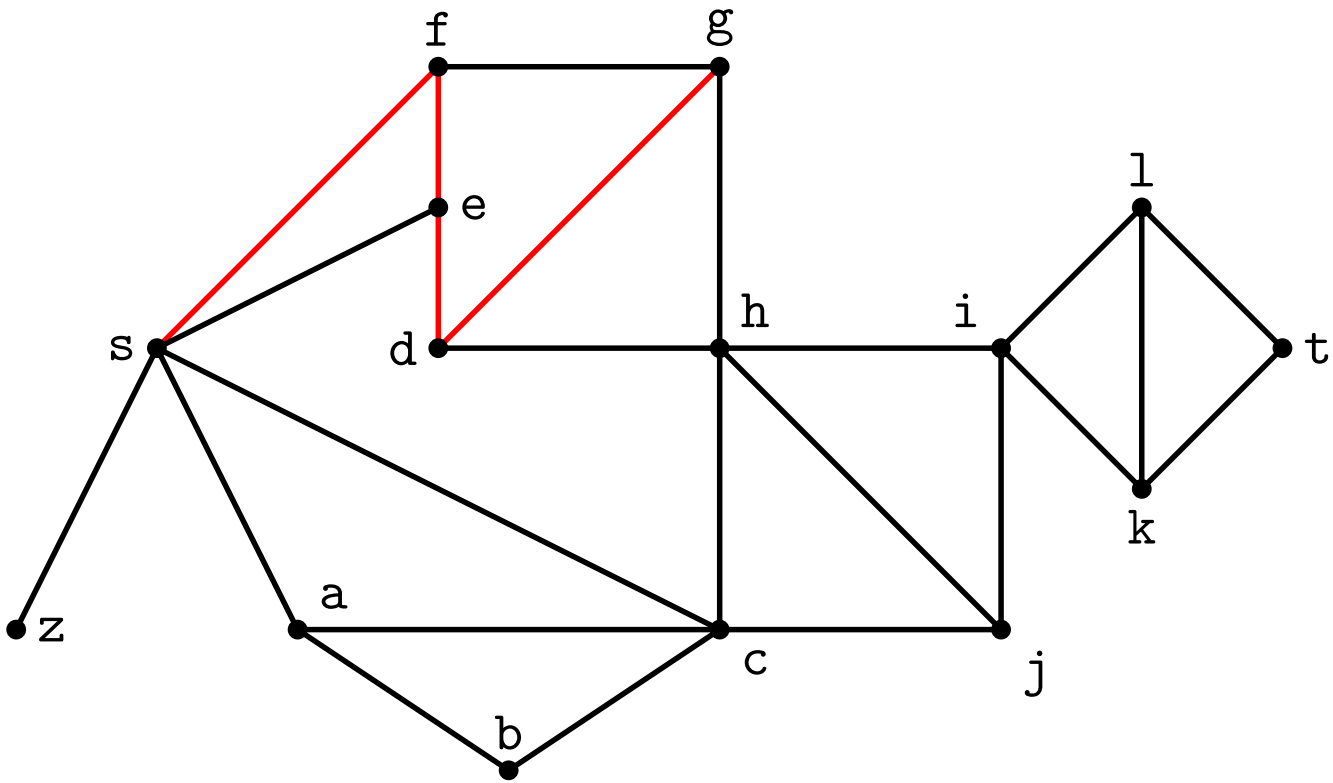
- Polynomial total time if the cost of a node is polynomial.
- If the height of the recursion tree is polynomial then the algorithm is polynomial delay.

Backtracking ('70 style)

- **Exploring all the ways of enlarging a partial solution: given a set $S \subseteq U$, a call returns all the solutions containing S.**
- The raw schema works well when any part of a solution is a solution. Otherwise it should be more sophisticated:
- Bron-Kerboch Algorithms for Maximal Clique (1973).
- Johnson Algorithm for listing paths (1975).

Backtracking for paths

Exploiting all the ways of enlarging a partial solutions, e.g. the path s, f, e, d, g.



In the case of Johnson Algorithm, some vertices are blocked (e.g. f) because they are in the prefix or some vertex in the prefix is blocking their only way to go to t.

Reverse Search (made in Japan)

- Define parent-children relationship between solutions, that induce a tree and apply depth first search.
- Cost per solution equal to cost per iteration.
- The delay dominated by the cost of an iteration by using alternative output
  - Otherwise returning from the recursive call is expensive.

Algorithm 2: ReverseSearch

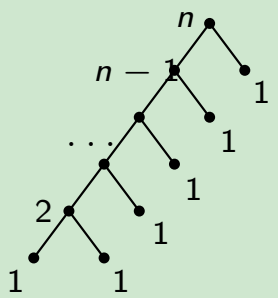
output S
foreach *child* S' *of* S **do**
  ReverseSearch(S')
end

Analysis by Amortization (made in Japan)

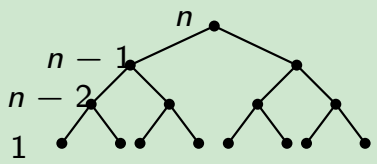
Average cost per solution by studying recursion tree shape.

Some examples

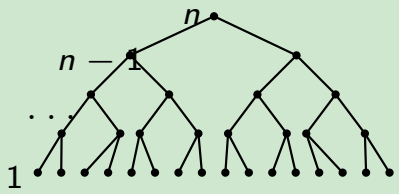
Amortized cost
$O(n)$



Amortized cost
$O(n)$



Amortized cost
$O(1)$



- **Basic amortization:** if the height is n and each internal node has degree at least 2, then many leaves to charge.
- **Amortization by children:** express the cost per node with respect to the number of its children.
- **Push out amortization:** if the children grow faster than the sum of their cost; or if there is sufficient quantity of output or children per node.

Algorithm & Analysis: Amortization with Certificate (made in Italy)

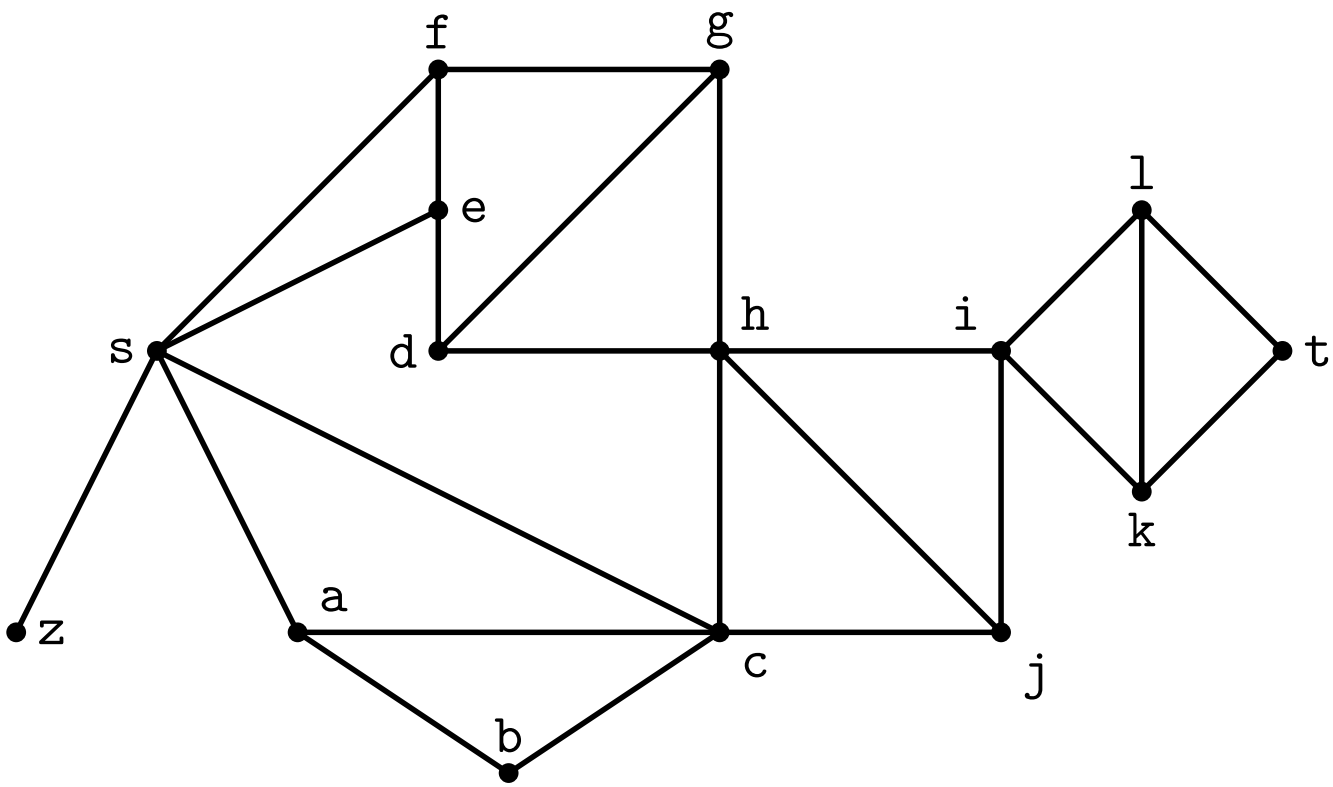
- Recursive approach ensuring that each call is productive using a data structure helping the decision, i.e. the *certificate*.
- The cost to update the data structure while recurring is amortized by the number of solutions.

Applied for instance for paths and cycles enumeration

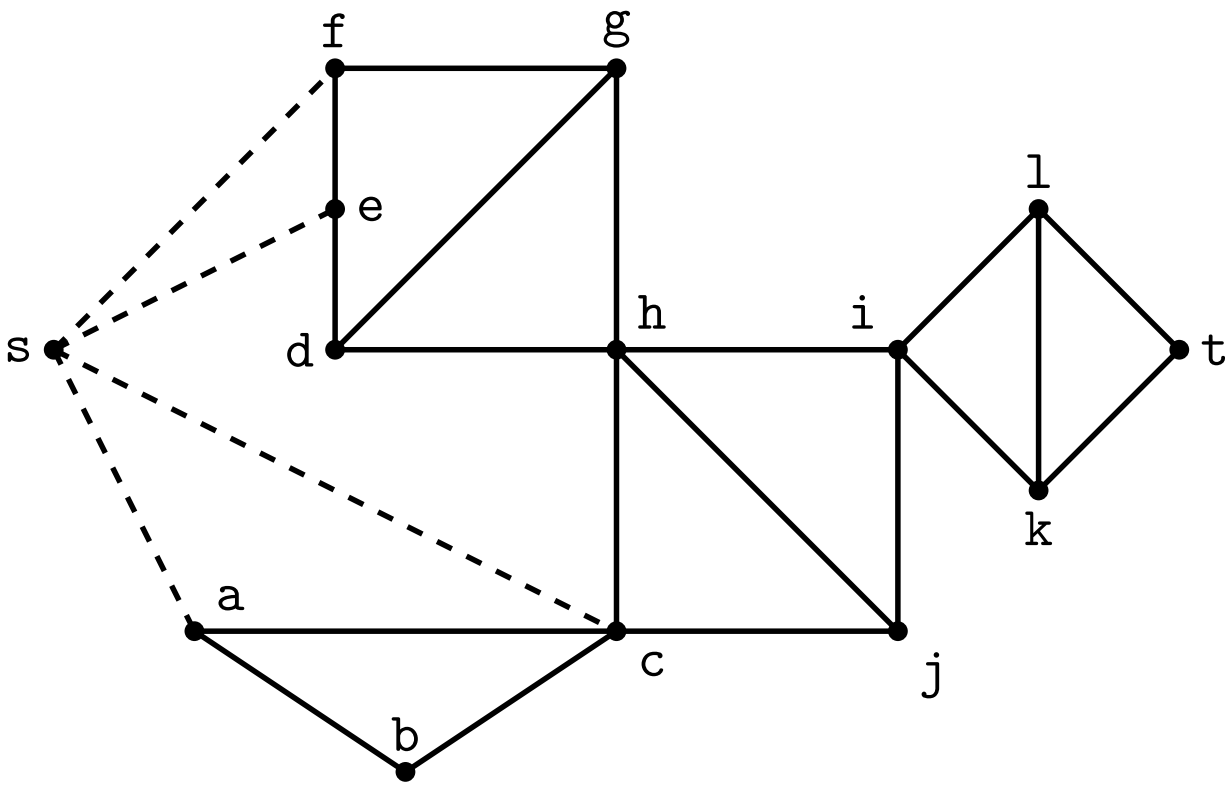
Algorithm 3: $\textsc{Paths}(G, u, t, S)$

Input: A graph G, the vertices u and t, a sequence of vertices S
Output: All the paths from u to t in G
if $u = t$ **then** output S; **return**;
for *each useful neighbor* v *of* u **do**
  $\textsc{Paths}(G - u, v, t, \langle S, u \rangle)$;
end

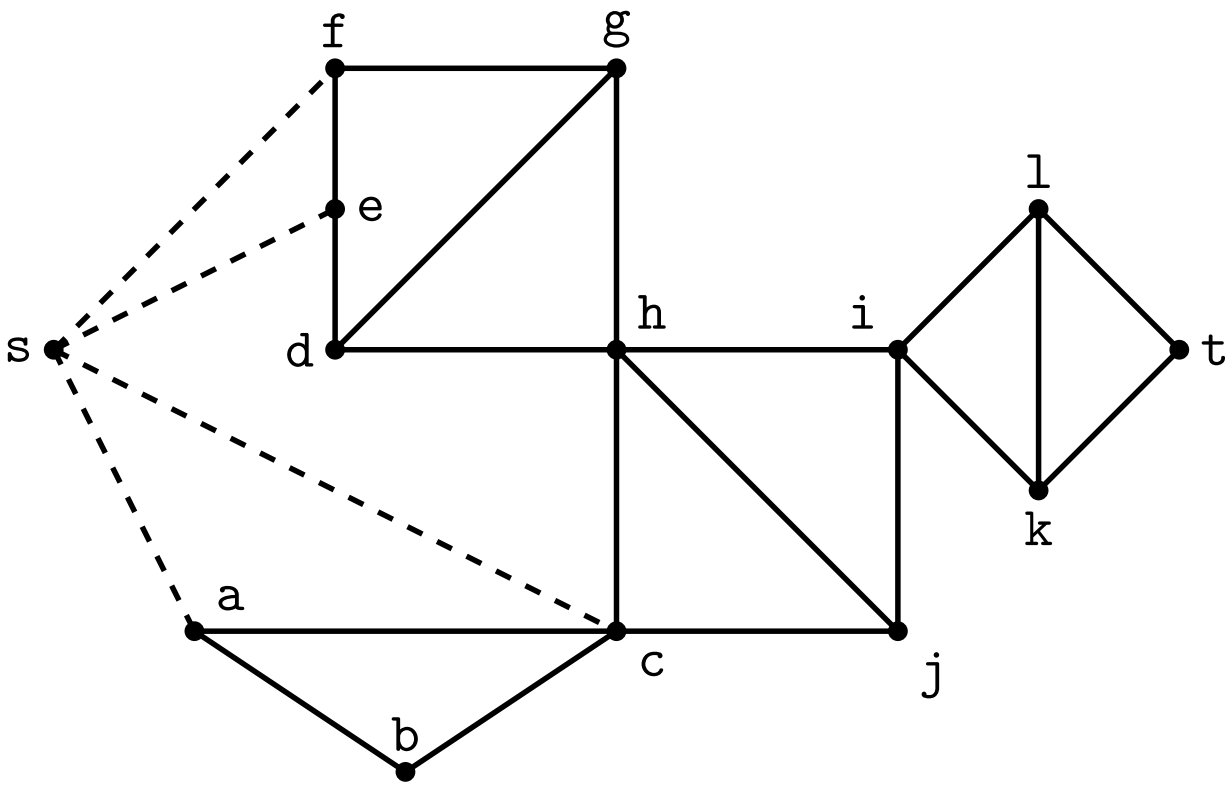
The certificate are the biconnected components of the current graph updated while recurring.



The *useful* neighbors of s are the ones in the biconnected component of s leading to t, e.g. z is a useless neighbor for s.



- The parent s will compute the biconnected components of $G - s$ for all its children in one shot.
- When s is deleted, the recursion starting from f,e,c,a will use the same decomposition.



- When s is deleted, just the first biconnected component modifies, we can reuse the others.
- We can pay for this cost, since we have a lower bound on the number of solutions a branch will produce.

Measure & Conquer (made in Norway)

- Given the instance I, defines $\mu(I)$ as the measure.
- Denote with $T(\mu)$ as an upper bound on the number of leaves of the search trees modeling the recursive calls of the algorithm for all I with $\mu(I) < \mu$.
- Define rules reducing I and observing how the measure and $T(\mu)$ modifies on the new instances.

Part III

Finding Communities in Graphs

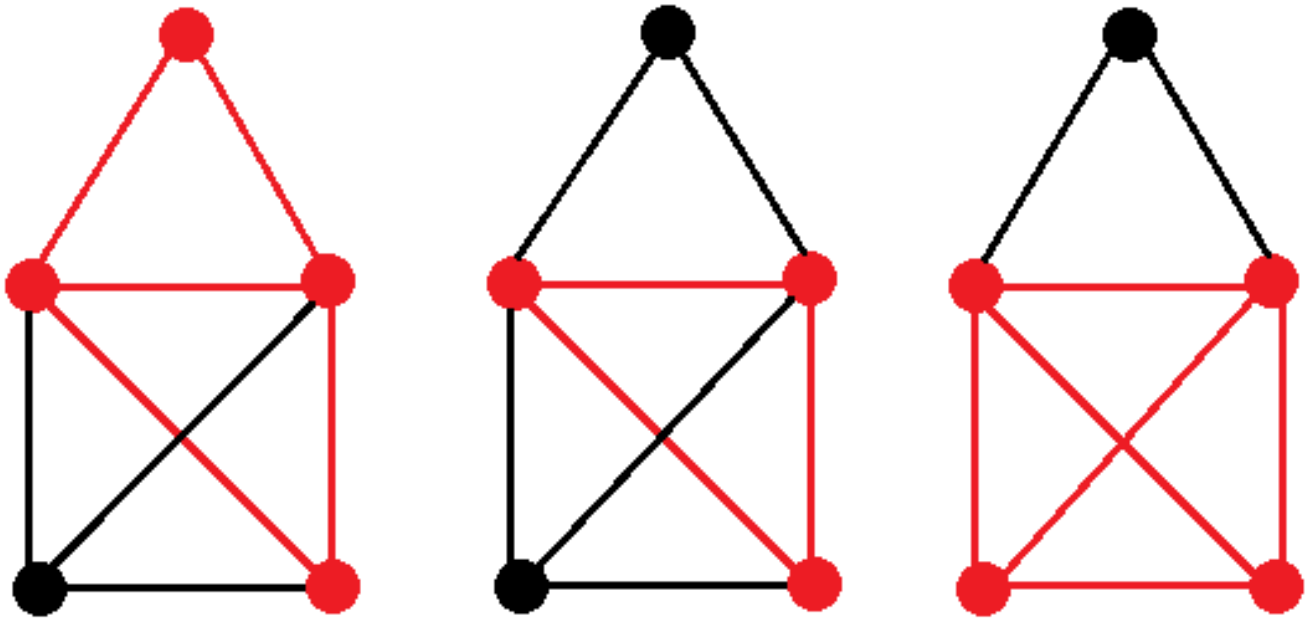
Several Notions of Communities

Induced Subgraph S, $d_S(u)$ is the degree of u inside S.

Clique	Each vertex in S is neighbour of all the others, i.e. $d_S(u) = \|S\| - 1$.
k-Core	$d_S(u) \geq k$.
k-Plex	$d_S(u) \geq \|S\| - k$.
k_d-Clique	$diam_G(S) \leq k_d$.
k-Club	$diam_S(S) \leq k_d$.

Finding Maximal Cliques

Given a graph G, list all the maximal cliques in G.



not a clique

non-maximal clique

maximal clique

Well-Studied Problem

Algorithm	Time			Space
	Setup	Delay	Overall	
Bron-Kerbosch	$O(m)$	unbounded	unbounded	$O(n+q\Delta)$
Tomita et al.	$O(m)$	$\Omega(n^3)$	$O(3^{n/3})$	$O(n+q\Delta)$
Eppstein et al.	$O(m)$	$\Omega(3^{n/6})$	$O(d(n-d)3^{d/3})$	$O(n+d\Delta)$
Johnson et al.	$O(mn)$	$O(mn)$	$\alpha\, O(mn)$	$O(\alpha n)$
Tsukiyama et al.	$O(n^2)$	$O((n^2-m)n)$	$\alpha\, O((n^2-m)n)$	$O(n^2)$
Chiba-Nishizeki	$O(m)$	$O(md)$	$\alpha\, O(md)$	$O(m)$
Makino-Uno	$O(mn)$	$O(\Delta^4)$	$\alpha\, O(\Delta^4)$	$O(m)$
Chang et al.	$O(m)$	$O(\Delta h^3)$	$\alpha\, O(\Delta h^3)$	$O(m)$
Makino-Uno*	$O(n^2)$	$O(n^{2.37})$	$\alpha\, O(n^{2.37})$	$O(n^2)$
Comin-Rizzi*	$O(n^{5.37})$	$O(n^{2.09})$	$\alpha\, O(n^{2.09})$	$O(n^{4.27})$
Conte et al.	$\tilde{O}(m)$	$\tilde{O}(qd(\Delta+qd))$	$\alpha\, \tilde{O}(qd(\Delta+qd))$	$O(q)$
Conte et al.	$\tilde{O}(m)$	$\tilde{O}(\min\{md, qd\Delta\})$	$\alpha\, \tilde{O}(\min\{md, qd\Delta\})$	$O(d)$

n = #vertices, Δ = max degree, α = #maximal cliques
m = #edges, d = degeneracy, q = largest clique size

Bounds for maximal clique enumeration, where $q-1 \leq d \leq \Delta \leq n-1 \leq m$,

$d = O(\sqrt{m})$.

A popular Recursive Backtrack Algorithm

- The Bron-Kerbosch algorithm was designed by Dutch scientists Coenraad Bron and Joep Kerbosch, who published its description in 1973.
- It is simple and very efficient in practice.
- It is well-known and widely used in application areas of graph algorithms such as computational chemistry.
- To find other variation of communities, sometimes they modify this algorithm.

- The basic form of the Bron-Kerbosch algorithm is a recursive backtracking which solve the following core problem.

Core Problem

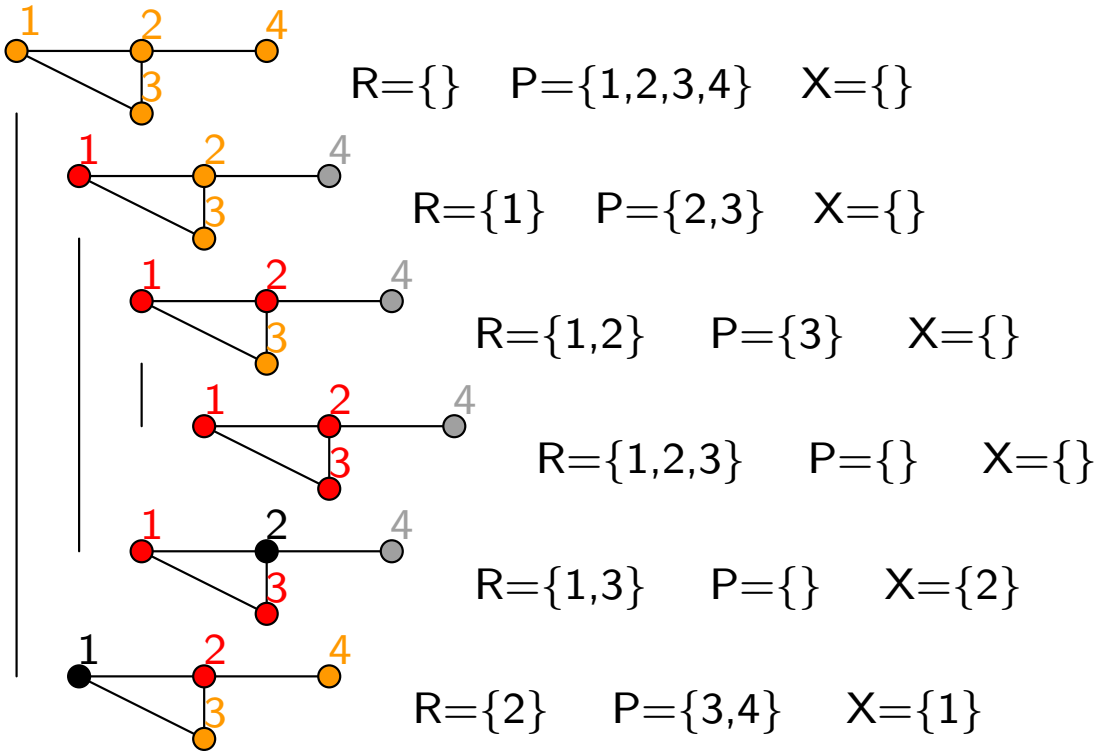
Given three sets R, P, and X, find all the maximal cliques that include:

- all of the vertices in R,
- some of the vertices in P, and
- none of the vertices in X.

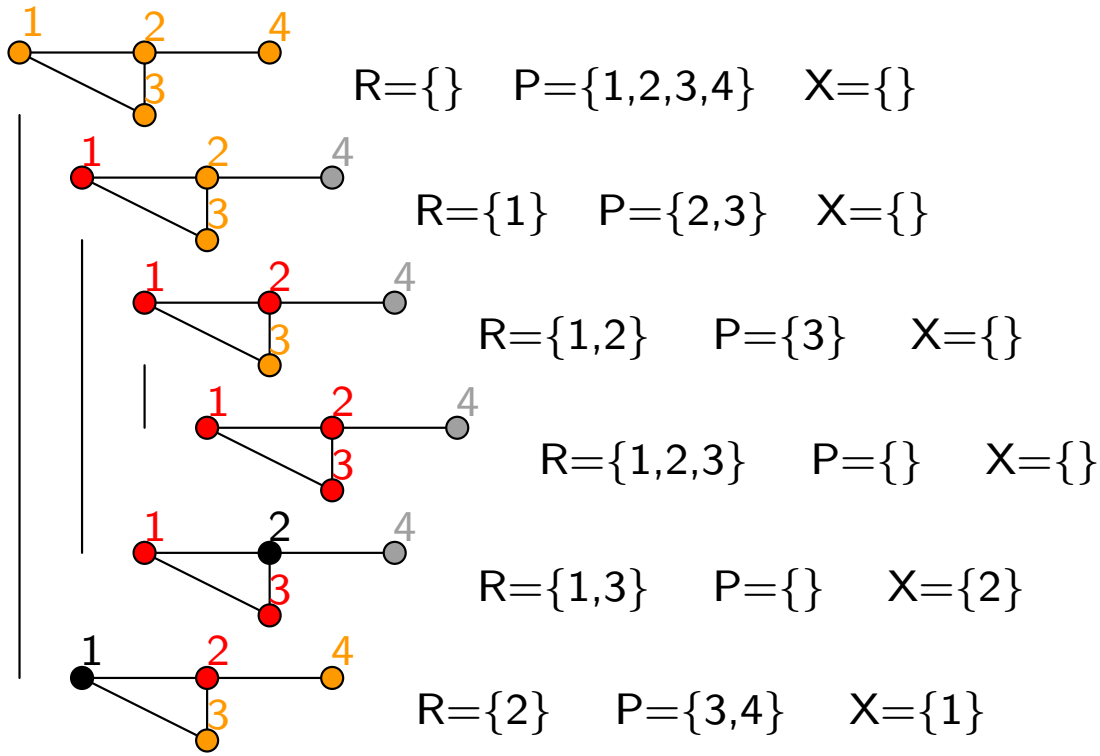
Assuming that $P \cap X = \emptyset$ and for each $v \in P \cup X$, $R \cup \{v\}$ is a clique, i.e. $v \in N(R)$.

- To find all the maximal cliques set $R = \emptyset$, $P = V$, and $X = \emptyset$.
- When P and X are both empty there are no further elements that can be added to R, so R is a maximal clique and the algorithm outputs R.

- Starting from $R = \emptyset$, $P = V$, and $X = \emptyset$,
- if $P = \emptyset$, it either report R as a maximal clique (if X is empty), or backtrack.
- For each vertex $v \in P$, do a recursive call finding all clique extensions of R that contain v, i.e. setting:
  - v is added to R
  - P and X are restricted to the neighbor set $N(v)$ of v,
- When backtracking move v from P to X to exclude it from consideration in future cliques and continues with the next vertex in P.




```
BK1(R, P, X):
        if P and X are both empty:
            report R as a maximal clique
        for each vertex v in P:
            BK1(R union {v},P inters N(v), X inters N(v))
            P := P \ {v}
            X := X union {v}
```



Often we lose time to go in dead-ends: each time $P = \emptyset$ and $X \neq \emptyset$ we back-track and we do not produce anything

Can we reduce the number of these bad cases? Yes, choosing a pivot vertex.

With Pivoting

- Each time choose a "pivot vertex" u, chosen from $P \cup X$.
- Any maximal clique must include either u or one of its non-neighbors, for otherwise the clique could be augmented by adding u to it.

Hence, only u and its non-neighbors need to be tested as the choices for the vertex v that is added to R in each recursive call to the algorithm.

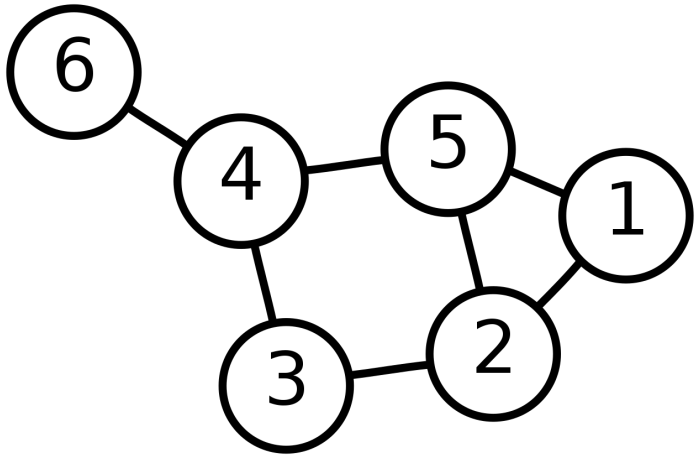
```
BK2(R,P,X):
    if P and X are both empty:
        report R as a maximal clique
    choose a pivot u in P union X
    for each vertex v in P \ N(u):
        BK2(R union {v},
            P inters N(v),
            X inters N(v))
        P := P \ {v}
        X := X union {v}
```

Tomita Pivoting

Choose the pivot $u \in P \cup X$ as the node having more neighbours in P.

Degeneracy

- The degeneracy of a graph G is the smallest number d such that every subgraph of G has a vertex with degree d or less.
- Every graph has a degeneracy ordering, an ordering of the vertices such that each vertex has d or fewer neighbors that come later in the ordering.
- A degeneracy ordering may be found in linear time by repeatedly selecting the vertex of minimum degree among the remaining vertices.



- The graph has degeneracy two.
- One possible degeneracy ordering is 6,4,3,1,2,5.
- In this ordering, each node has two forward neighbours.
- In social networks, d is very small.

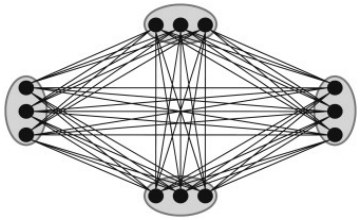
With Vertex Ordering

- Choosing the ordering of the recursive calls carefully in order to minimize the sizes of the sets P of candidate vertices within each recursive call.
- Using degeneracy ordering, then the set P of candidate vertices in each call (the neighbors of v that are later in the ordering) will be guaranteed to have size at most d.
- The set X of excluded vertices will consist of all earlier neighbors of v, and may be much larger than d.

```
BK3(G):
        P = V(G)
        R = X = empty
        for each vertex v in a degeneracy ordering of G:
            BK2(R union {v},
                  P inters N(v),
                  X inters N(v))
            P := P \ {v}
            X := X union {v}
```

Worst Case Analysis

- The Bron-Kerbosch algorithm is not an output-sensitive algorithm.
  - Unlike some other algorithms for the clique problem, it does not run in polynomial time per maximal clique generated.
- It is efficient in a worst-case sense: the worst-case running time (with a pivot strategy) is $O(3^{n/3})$
  - By Moon & Moser (1965), any n-vertex graph has at most $3^{n/3}$ maximal cliques, matching the bound.



- For sparse graphs, tighter bounds are possible. The vertex-ordering version runs in time $O(dn3^{d/3})$, where d is the degeneracy of the graph.
  - There exist d-degenerate graphs for which the total number of maximal cliques is $(n-d)3^{d/3}$, so this bound is close to tight.

Other Applications

- The Bron-Kerbosch algorithm is often used also to find the maximum cliques instead of all the maximal cliques:
  - output just the maximum clique found
  - as in branch and bound, try to cut the branches not leading to maximum cliques.
- The Bron-Kerbosch algorithm has been adapted to find other variations of communities, like K-plexes.

Devora Berlowitz, Sara Cohen, Benny Kimelfeld: Efficient Enumeration of Maximal k-Plexes. SIGMOD Conference 2015: 431-444

Other Applications: Finding Iomorphisms

An isomorphism between two graphs is a mapping among vertices preserving adjacencies.

- **From Common Subgraphs to Cliques:** Given two graphs G and H, all the isomorphisms among subset of vertices correspond to cliques in a new graph $P(H, G)$.
  - $P(H, G)$ is called product graph and has size $O(n^2)$ nodes and up to $O(n^4)$ edges: it is never materialized in the application.

G. Levi: A note on the derivation of maximal common subgraphs of two directed or undirected graphs, CALCOLO 9 (1973) 341352.

Ina Koch: Enumerating all connected maximal common subgraphs in two graphs. Theor. Comput. Sci. 250(1-2): 1-30 (2001)

Other Applications: Finding Motifs

A motif is a small graph of fixed size.

- **Maximum Common Subgraph:** If H is a motif, maximum cliques in $P(H, G)$ are occurrences of H in G.
  - The Ullman Algorithm can be seen as a variation of Bron-Kerbosch algorithm: it does not materialize P and it deals with maximum instead of maximal.

Julian R. Ullmann: An Algorithm for Subgraph Isomorphism. J. ACM 23(1): 31-42 (1976)