Exercise Sheet 4: Social and Graph Data Management

Pablo Mollá Chárlez

Contents

1 Exercise 1: Treewidth & Properties

- **Question 1.** Show that the maximal graphs of pathwidth 1 are caterpillars trees (trees where each vertex is at distance 1 from a central path).
- **Question 2.** Give a tree decomposition of width 3 for the following graph 1 (Hint: you may include a bag "$\{a, c, f, h\}$ in the decomposition):

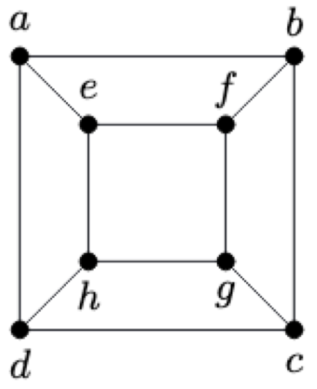

Figure 1: Graph Q2

- **Question 3.** Give the treewidth and pathwidth for a star with n vertices.
- **Question 4.** An interval graph is the intersection graph from a set of intervals on the real line: each interval is a vertex of the interval graph, and vertices are connected by an edge if the intervals overlap.

How can you alternatively define the maximum clique size in an interval supergraph containing G? Prove it.

- **Question 5.** Show that there exists an algorithm which decides if a graph can be drawn on a doughnut without crossing edges.

1.1 Answers

- **Question 1.** Show that the maximal graphs of pathwidth 1 are caterpillars trees (trees where each vertex is at distance 1 from a central path).

  Response: This is **Theorem 6.2** in book.

  We can first prove that caterpillars have pathwidth 1: the central path essentially provides the bag decomposition:

  - We order the vertices as follows: vertices on the central path are ranked in ascending order based on their position on the path. Every vertex outside the central path is inserted immediately after the vertex of the central path it is attached to.
  - We construct a nice path decomposition of width 2 by introducing the vertices in order, always keeping in the bag the last vertex from the central path. This vertex is forgotten only after the next vertex on the central path is introduced. Vertices outside the central path are forgotten immediately after they are introduced.
  - We then prove that caterpillars are maximal graphs with pathwidth 1: adding an edge to a caterpillar increases the pathwidth to 3 because the graph then contains a cycle, and a cycle has a triangle minor, which implies a pathwidth of at least 3.

  Finally, we prove that maximal graphs with pathwidth 1 are caterpillars: a nice tree decomposition provides the central path. By maximality, the nice tree decomposition will never have an empty bag, and when a vertex is forgotten, it must be connected to the other vertex that remains in the bag.

- **Question 2.** Give a tree decomposition of width 3 for the following graph 1 (Hint: you may include a bag "$\{a, c, f, h\}$ in the decomposition): The tree decomposition can be found on figure 2

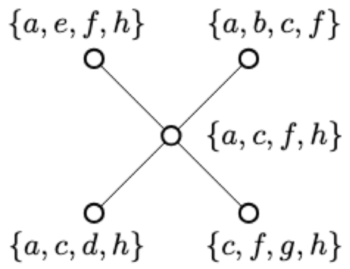

Figure 2: Tree Decomposition of Width 3

- **Question 3.** Give the treewidth and pathwidth for a star with n vertices.

  No answer

- **Question 4.** An interval graph is the intersection graph from a set of intervals on the real line: each interval is a vertex of the interval graph, and vertices are connected by an edge if the intervals overlap. How can you alternatively define the maximum clique size in an interval supergraph containing G? Prove it.

  No answer

- **Question 5.** Show that there exists an algorithm which decides if a graph can be drawn on a doughnut without crossing edges.

  The property is closed under minors. Therefore, by the Robertson-Seymour theorem, there exists a fixed, finite set of "forbidden minors," i.e., a finite set of graphs $S = \{G_1, \ldots, G_n\}$ such that a graph can be drawn on a torus without crossing edges if and only if it does not contain any graph from S as a minor. The algorithm checks whether each graph in S is a minor of G.

  This approach is completely impractical because (i) the set S is unknown; we only know that it exists, and (ii) checking if a graph is a minor of G is not computationally efficient—it is quadratic in the size of G but extremely expensive in terms of the size of the minor. However, this demonstrates that solving the problem is theoretically possible with a cost polynomial in the size of G.

2 Exercise 2: Percolation

The original question solved by bond percolation is checking when exists a path from one side of the lattice to the opposite side. We consider bond percolation (adding a fraction of edges, as opposed to site percolation) on a $2D$ lattice. Give an intuition of the percolation threshold, assuming there is a single phase transition (you may consider a dual lattice). For a rigorous proof, and a discussion of why the intuition is not enough, see, e.g., article.

2.1 Answers

Consider the dual lattice (shifted by $\frac{1}{2}, \frac{1}{2}$): each edge of the dual crosses exactly one edge from the original lattice (each horizontal edge is crossed by a vertical one, and vice versa). We consider an edge to be present in the dual if and only if the corresponding (crossed) edge is absent in the original lattice.

The probability for edges in the dual is $p' = 1 - p$, where p is the probability in the original lattice. The percolation problem with p' on the dual is equivalent, by symmetry (rotation by $\pi/2$), to the problem on the original lattice with p. There is an infinite path from top to bottom in the original lattice if and only if there is no infinite path from left to right in the dual. Intuitively, if there is a single phase transition and the vertical path appears in the lattice almost surely when $p > p_c$, then we should have $p_c = \frac{1}{2}$. However, proving this phase transition rigorously is not straightforward (see the URL above).

Remark: The URL refers to a lecture by Hugo Duminil-Copin at IHÉS. Duminil-Copin is a French mathematician who received the Fields Medal in 2022 for his work on phase transitions in percolation theory. The Fields Medal is often regarded as the equivalent of the Nobel Prize for mathematicians, with some differences: it is awarded to young and promising mathematicians, while the Abel Prize in mathematics is closer in spirit to the Nobel Prize, as it is awarded to mathematicians of any age, often as recognition for long-standing achievements.

3 Exercise 3: Independent Set

The (vertex-weighted) maximum independent set problem takes as input a graph $G = (V, E)$ with a weight function $w : V \to N$ and asks for an independent set $S \subseteq V$ of maximum weight. (an independent set is a set of vertices S such that $S \times S \cap E = \varnothing$).

- **(a)** Give a dynamic programming algorithm for the problem on a tree.
- **(b)** Give a dynamic programming algorithm for the problem on a grid $k_1 \times k_2$ with $k_2 >> k_1$.
- **(c)** Generalize.

3.1 Answers

- **(a)** On a tree, we have to process the tree bottom-up. For each vertex v, and denoting by T_v the subtree below v:

  1. Record c_v, the weight of the largest independent set in T_v that contains v.
  2. Record c'_v, the weight of the largest independent set in T_v that does not contain v.

  This can be computed using dynamic programming with the equations:

$$c_v = w(v) + \sum_{v' \text{ child of } v} c'_{v'}$$

$$c'_v = \sum_{v' \text{ child of } v} \max(c_{v'}, c'_{v'})$$

  where we adopt the convention that an empty sum equals 0 (for the leaves).

  To conclude, the algorithm returns $\max(c_r, c'_r)$, where r is the root of the tree. This provides the maximal weight W_M achievable by an independent set. A corresponding independent set (with weight W_M) can be constructed by tracing back through the dynamic programming table or recording partial solutions during the table construction.

- **(b)** On a grid $k_1 \times k_2$ with $k_2 \gg k_1$, a similar idea applies: for each column of the grid, record, for any valid subset $S \subseteq \{1, \dots, k_1\}$, the weight $c(S, i)$ of the best independent set whose intersection with column i is exactly S (i.e., it contains (x, i) for all $x \in S$, and does not contain (x, i) for all $x \notin S$). This can again be computed by dynamic programming with the equation:

$$c(S, i) = \max_{S' \text{ compatible with } S} c(S', i-1) + \sum_{x \in S} w(x, i),$$

  where:

  - A valid subset S means S cannot contain consecutive integers k_j and $j+1$.
  - **Compatible subsets** S and S' mean $S \cap S' = \varnothing$ (independent sets must not include adjacent vertices).

  The max evaluates to 0 for the first column, by convention. To conclude, the algorithm returns $\max_S c(S, n)$, where n is the last column of the grid. As before, this only provides the maximal weight, but the corresponding set of vertices can be constructed using standard dynamic programming techniques.

- **(c)** We can generalize to graphs of bounded treewidth, assuming we are provided with a (nice) tree decomposition. For each node x of the nice tree decomposition, denote by G_x the subgraph induced by the vertices of G appearing below x in the (bags of the) tree decomposition. Let $B \subseteq V$ be the bag of vertices at node x of the decomposition. For every valid $S \subseteq B$, compute $c_x(S)$, the maximal weight of an independent set on G_x whose intersection with B is exactly S. For simplicity, write $c_B(S)$ instead of $c_x(S)$.

  A **valid subset** S means that S cannot contain pairs of vertices connected by an edge: $S^2 \cap E = \varnothing$. The computation proceeds as follows on a nice tree decomposition:

  1. **Introduce** ($B' = B \cup \{v\}$) for all valid $S \subseteq B$:
     - If $\exists u \in S$ such that $(u, v) \in E$: $c_{B'}(S \cup \{v\}) = -\infty$.
     - Otherwise: $c_{B'}(S \cup \{v\}) = c_B(S) + w(v)$.
     - Also: $c_{B'}(S) = c_B(S)$.
  2. **Forget** ($B' = B \setminus \{v\}$):
$$c_{B'}(S) = \max_{S' \supseteq S} c_B(S').$$
  3. **Join** ($B' = B_1 = B_2$):
$$c_{B'}(S_1 \cup S_2) = c_{B_1}(S_1) + c_{B_2}(S_2) - |S_1 \cap S_2|.$$

Note: $c_B(S)$ should actually be computed for each node B in the tree decomposition. Since multiple nodes may share the same bag of vertices but have different records $c_B(S)$, we slightly abuse notation by treating B as a subset of V, referring instead to "the set of vertices stored in the bag at node B."

To conclude, the algorithm returns $\max_S c_B(S)$, where B is the node at the root of the nice tree decomposition. As before, this provides the maximal weight, and the corresponding set of vertices can be reconstructed using dynamic programming techniques.

4 Exercise 4: Vertex Cover

The vertex-cover problem takes as input a graph $G = (V, E)$, and integer k, and must output "True" if there is a subset of V denoted S of size k such that every edge of G has at least one endpoint in S. Give an algorithm that computes the cover in linear time for constant k if we are provided a tree decomposition of constant with h.

4.1 Answers

We assume, without loss of generality (w.l.o.g.), that the tree decomposition provided is **nice**. For every bag B of the tree decomposition (processed bottom-up), and for every $S \subseteq V$, let $c_B(S)$ denote the best cost achieved on a partial solution for bag B (i.e., for the subgraph induced by the vertices appearing in the subtree below B), under the condition that the vertices from B selected for the vertex cover are precisely S. The dynamic programming steps are:

1. **Introduce ($B' = B \cup \{v\}$):** For all $S \subseteq B$:
   (a) If $\exists u \in B \setminus S$ such that $(u, v) \in E$:
   $$c_{B'}(S) = \infty$$
   (This ensures the partial solution is invalid if v is added without covering all edges adjacent to v.)
   (b) Otherwise:
   $$c_{B'}(S) = c_B(S)$$
   (c) For cases where v is included in the vertex cover ($S \cup \{v\}$):
   $$c_{B'}(S \cup \{v\}) = c_B(S) + 1$$
2. **Forget ($B' = B \setminus \{v\}$):** For all $S \subseteq B'$:
   $$c_{B'}(S) = \min_{S' \supseteq S} c_B(S')$$
   (This step ensures that we account for all valid extensions of the partial solution after removing v from consideration.)
3. **Join ($B' = B_1 = B_2$):** For all $S_1, S_2 \subseteq B'$:
   $$c_{B'}(S_1 \cup S_2) = c_{B_1}(S_1) + c_{B_2}(S_2) - |S_1 \cap S_2|$$
   (This step merges two partial solutions while subtracting the overcounted cost for vertices shared by S_1 and S_2.)

Notational Clarification: As in similar algorithms for nice tree decompositions, the notation $c_B(\cdot)$ and $c_{B'}(\cdot)$ refers to the dynamic programming states associated with nodes of the tree decomposition. However, B or B' also represents the **bags of vertices** stored at the corresponding nodes. We abuse the notation for simplicity, instead of introducing additional symbols. The algorithm concludes by checking:

$$\min_S c_B(S) \leq k$$

where B is the node at the root of the nice tree decomposition. If this condition holds, the algorithm returns True, indicating that there exists a vertex cover of size at most k.

5 Modularity

- **(a)** Explain the variables in the equation defining modularity:

$$\frac{1}{2m}\sum_{i,j}(A_{ij} - \frac{k_i k_j}{2m})\delta_{c_i,c_j}$$

- **(b)** Rewrite the formula into one that involves only k_c, the sum of degrees inside the community, L_c the number of edges inside the community and m.

5.1 Answers

- **(a)** Explain the variables in the equation defining modularity:

$$\frac{1}{2m}\sum_{i,j}(A_{ij} - \frac{k_i k_j}{2m})\delta_{c_i,c_j}$$

  The variables included are:

  - m: number of edges in the graph
  - δ_{c_i,c_j}: 1 if vertices i and j belong to the same community ($c_i = c_j$), 0 otherwise
  - $k_i(k_j)$: degree of node i (and j)
  - $A_{i,j}$ equals 1 if i and j are connected by an edge, 0 otherwise.

- **(b)** Rewrite the formula into one that involves only k_c, the sum of degrees inside the community, L_c the number of edges inside the community and m.

  The equation can be manipulated to use the previously mentioned variables:

$$\frac{1}{2m}\sum_{i,j}(A_{ij} - \frac{k_i k_j}{2m})\delta_{c_i,c_j}$$

$$= \frac{1}{2m}\sum_{c}\sum_{i,j}(A_{ij} - \frac{k_i k_j}{2m})$$

$$= \frac{1}{2m}\sum_{c}(2L_c - \sum_{i,j\in c}\frac{k_i k_j}{2m})$$

$$= \frac{1}{2m}\sum_{c}(2L_c - \frac{(\sum_{i,j} k_i)^2}{2m})$$

$$= \sum_{c}(\frac{L_c}{2m} - (\frac{k_c}{2m})^2)$$