CS 240 Homework 2

Wei Yuxiang ID: 206529292

Q1. Prove that Parity $\in \mathrm{NC}^1$

We show that $\mathrm{Parity}(x_1, \ldots, x_n) = x_1 \oplus x_2 \oplus \cdots \oplus x_n$ can be computed by a Boolean circuit of depth $O(\log n)$ and size $O(n)$.

Each XOR gate can be expressed as

$$a \oplus b = (a \vee b) \wedge \neg(a \wedge b),$$

which is a constant-depth circuit using $\{\textsc{and}, \textsc{or}, \textsc{not}\}$ gates.

Arrange the XOR gates in a balanced binary tree: each internal node computes the XOR of its two children, and the root outputs the overall parity. Hence,

$$\text{depth} = \lceil \log_2 n \rceil = O(\log n), \quad \text{size} = n - 1 = O(n).$$

Since a log-space machine can describe this circuit family, it is uniform. Therefore,

$$\textsc{Parity} \in \mathrm{NC}^1.$$

Q2. Prove that $NC^i \subseteq AC^i \subseteq NC^{i+1}$

Definitions. A language is in NC^i if it can be computed by a family of Boolean circuits of polynomial size, depth $O(\log^i n)$, and bounded fan–in (at most 2). The class AC^i is defined similarly but allows *unbounded* fan–in for AND and OR gates.

(1) $NC^i \subseteq AC^i$. Every bounded fan–in gate is a special case of an unbounded one. Hence, any NC^i circuit is also an AC^i circuit with the same depth and size. Therefore $NC^i \subseteq AC^i$.

(2) $AC^i \subseteq NC^{i+1}$. Conversely, consider an AC^i circuit of depth $O(\log^i n)$. Each unbounded fan–in gate (say an AND or OR with k inputs) can be replaced by a balanced binary tree of the same gate type, where each node has fan–in 2. This increases the depth by $O(\log k)$. Since $k \leq n^{O(1)}$, we have $O(\log k) = O(\log n)$. Thus the overall circuit depth becomes

$$O(\log^i n) + O(\log n) = O(\log^{i+1} n),$$

and the size remains polynomial.

Conclusion. Replacing each unbounded gate by a binary tree increases the depth by at most one logarithmic factor. Hence any AC^i circuit can be simulated by an NC^{i+1} circuit. Together we have

$$NC^i \subseteq AC^i \subseteq NC^{i+1}.$$

Q3. Prove that the evaluation of FO formulae is in AC^0

We consider the *data complexity* of evaluating a fixed first-order (FO) sentence on a finite structure. Let dom be the domain of size n. An FO sentence is built from atomic predicates, Boolean connectives, and quantifiers $\exists, \forall$.

Translation to circuits. Each atomic predicate $R(x_1, \ldots, x_k)$ is evaluated by reading the corresponding entry in the input relation R. Boolean connectives correspond to standard logic gates:

$$\neg\varphi \;\mapsto\; \textsc{not}(\varphi), \qquad \varphi_1 \wedge \varphi_2 \;\mapsto\; \textsc{and}(\varphi_1, \varphi_2), \qquad \varphi_1 \vee \varphi_2 \;\mapsto\; \textsc{or}(\varphi_1, \varphi_2).$$

Quantifiers are interpreted as unbounded fan-in gates over all elements of the domain:

$$\exists x\, \varphi(x) \;\mapsto\; \bigvee_{a \in \mathrm{dom}} \varphi(a), \qquad \forall x\, \varphi(x) \;\mapsto\; \bigwedge_{a \in \mathrm{dom}} \varphi(a).$$

Depth and size. If the FO sentence has k nested quantifiers, the resulting circuit has depth

$$\text{depth} = O(k),$$

since each quantifier adds one layer of unbounded gates. Because the formula is fixed, k is a constant, so depth $= O(1)$. Each quantifier ranges over n elements, giving polynomially many gates:

$$\text{size} = n^{O(1)}.$$

Conclusion. The circuit family evaluating the FO sentence has constant depth, polynomial size, and unbounded fan-in AND/OR gates. Therefore,

$$\text{FO evaluation} \in \mathrm{AC}^0.$$

Q4. JOB 33c as a Conjunctive Query (join-only)

We ignore all non-join predicates and drop attributes not used in joins. Using shared variables to encode equality joins, a boolean CQ for JOB 33c is:

$$\begin{aligned} Q() \;:-\; & \mathit{company\_name}(c_1),\ \mathit{company\_name}(c_2), \\ & \mathit{info\_type}(it_1),\ \mathit{info\_type}(it_2), \\ & \mathit{kind\_type}(k_1),\ \mathit{kind\_type}(k_2), \\ & \mathit{link\_type}(\ell), \\ & \mathit{title}(m_1, k_1),\ \mathit{title}(m_2, k_2), \\ & \mathit{movie\_link}(\ell, m_1, m_2), \\ & \mathit{movie\_info\_idx}(it_1, m_1),\ \mathit{movie\_info\_idx}(it_2, m_2), \\ & \mathit{movie\_companies}(c_1, m_1),\ \mathit{movie\_companies}(c_2, m_2). \end{aligned}$$

Variable meaning: m_1, t_1-side movie id; m_2, t_2-side linked movie id; k_1, k_2 kind ids; it_1, it_2 info_type ids; c_1, c_2 company ids; ℓ link_type id.

Q5: Query Hypergraph for 33c

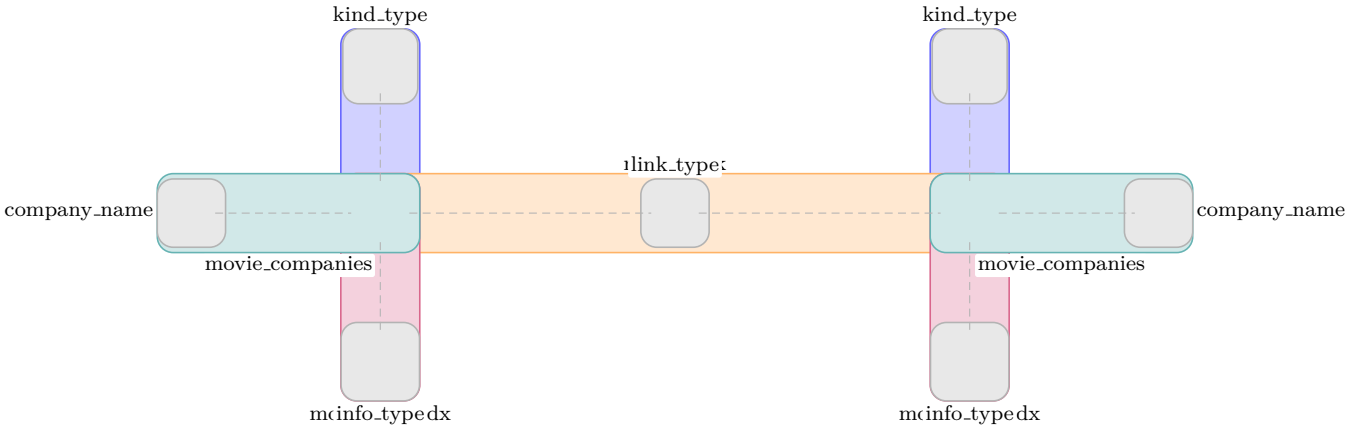
We use the conjunctive query (from Problem 4) that keeps only join predicates and join-participating attributes. Let the variable set (hypergraph vertices) be

$$V = \{\, c_1, c_2,\ it_1, it_2,\ k_1, k_2,\ \ell,\ m_1, m_2 \,\}.$$

Each relation atom induces a hyperedge on the variables it mentions. Thus the hyperedge set is

$$\begin{aligned} E = \{\ & \{c_1\}\text{\_company\_name},\ \{c_2\}\text{\_company\_name},\ \{it_1\}\text{\_info\_type},\ \{it_2\}\text{\_info\_type}, \\ & \{k_1\}\text{\_kind\_type},\ \{k_2\}\text{\_kind\_type},\ \{\ell\}\text{\_link\_type}, \\ & \{m_1, k_1\}\text{\_title},\ \{m_2, k_2\}\text{\_title}, \\ & \{\ell, m_1, m_2\}\text{\_movie\_link}, \\ & \{it_1, m_1\}\text{\_movie\_info\_idx},\ \{it_2, m_2\}\text{\_movie\_info\_idx}, \\ & \{c_1, m_1\}\text{\_movie\_companies},\ \{c_2, m_2\}\text{\_movie\_companies}\ \}. \end{aligned}$$

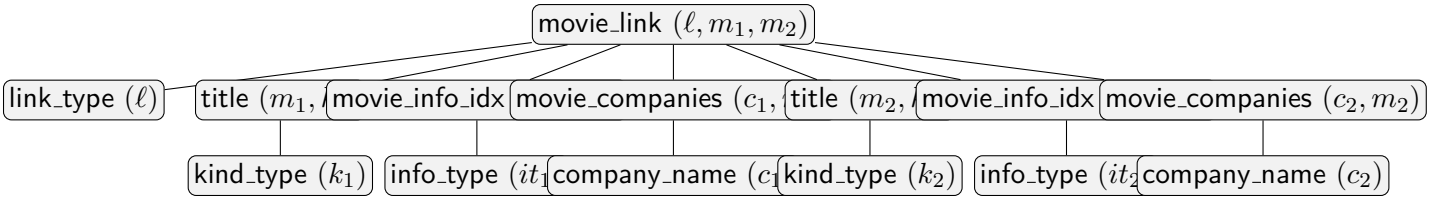
TikZ drawing of the hypergraph.



Explanation. Each vertex represents a join variable shared across relations. Each colored region denotes one relation (a hyperedge) covering the variables that appear in it. The central edge movie_link(ℓ, m_1, m_2) connects the two symmetric parts of the query through movies m_1 and m_2.

Q6: A Join Tree for 33c (rooted at movie_link)

A valid join tree that satisfies the running–intersection property can be rooted at movie_link(ℓ, m_1, m_2) and attach the atoms that share variables with it. Unaries then hang from their respective binary atoms.



This join tree satisfies the running–intersection property, with movie_link(ℓ, m_1, m_2) *as the root.*

Q7: Implement Query 33c with Yannakakis' Algorithm

Implemented Yannakakis' algorithm for query 33c using the join tree rooted at movie_link. The left branch includes $\{t_1, it_1, mi\_idx1, mc_1, cn_1, kt_1\}$, and the right branch includes $\{t_2, it_2, mi\_idx2, mc_2,$ All join and non-join predicates are included.

The algorithm follows three steps: (1) bottom-up semijoin reduction, (2) top-down semijoin reduction, (3) final join and aggregation. Non-join filters (e.g., `rating < 3.5`) are handled by DuckDB during data loading.

The result was compared with the SQL baseline run directly in DuckDB. Both outputs match exactly, each returning one row with all `NULL` values, confirming correctness.

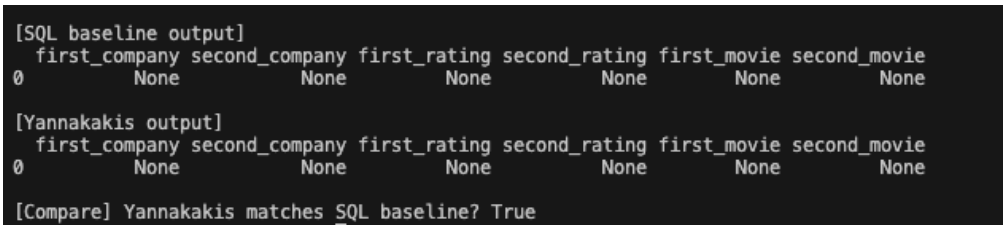


Figure 1: Comparison between SQL baseline and Yannakakis algorithm outputs. Both return a single row with all `NULL` values.

Reproducibility. I provide a single script `src/hw2_q7_yannakakis_33c.py`. It reads the original JOB CSVs from `./data` by default (configurable via `--data-dir`). No external DB is required; I use DuckDB's in-memory CSV reader. Running `python3 src/hw2_q7_yannakakis_33c.py` prints the SQL baseline result, our Yannakakis result, and a boolean equality check.