## CSE491/596 Fri. 11/6: NP-Completeness By Component Design IV: Path Problems

The basic path problem is, gives a graph $G$ and nodes $s, t \in V$, is there a path from $s$ to $t$ in $G$? Whether $G$ is a directed or undirected graph, this is in P by breadth-first search. But when we talk about more than one path and put constraints on the paths, problems become NP-hard and complete. One natural constraint is that multiple paths avoid each other---meaning they use different vertices.

### Disjoint Connecting Paths

**Instance**: An undirected graph $G = (V, E)$, start nodes $s_1, \dots, s_k$, and target nodes $t_1, \dots, t_k$.
**Question**: Are there disjoint paths $P_1, \dots, P_k$ with each $P_i$ going from $s_i$ to $t_i$?

For path problems we use other ideas besides "rungs" and "ladders." We need to set up zones of possible conflict between paths, or where paths must contend with their constraints. In this case, the idea is to have two sets of start and target nodes. One set stands for the variables, the other for the clauses. Thus given $\phi(x_1, \dots, x_n) = C_1 \wedge \cdots \wedge C_m$, we allocate start nodes

$$S = \{s_1, s_2, \dots, s_n\} \cup \{S_1, S_2, \dots, S_m\}$$

and target nodes---note that $k = n + m$:

$$T = \{t_1, t_2, \dots, t_n\} \cup \{T_1, T_2, \dots, T_m\}$$

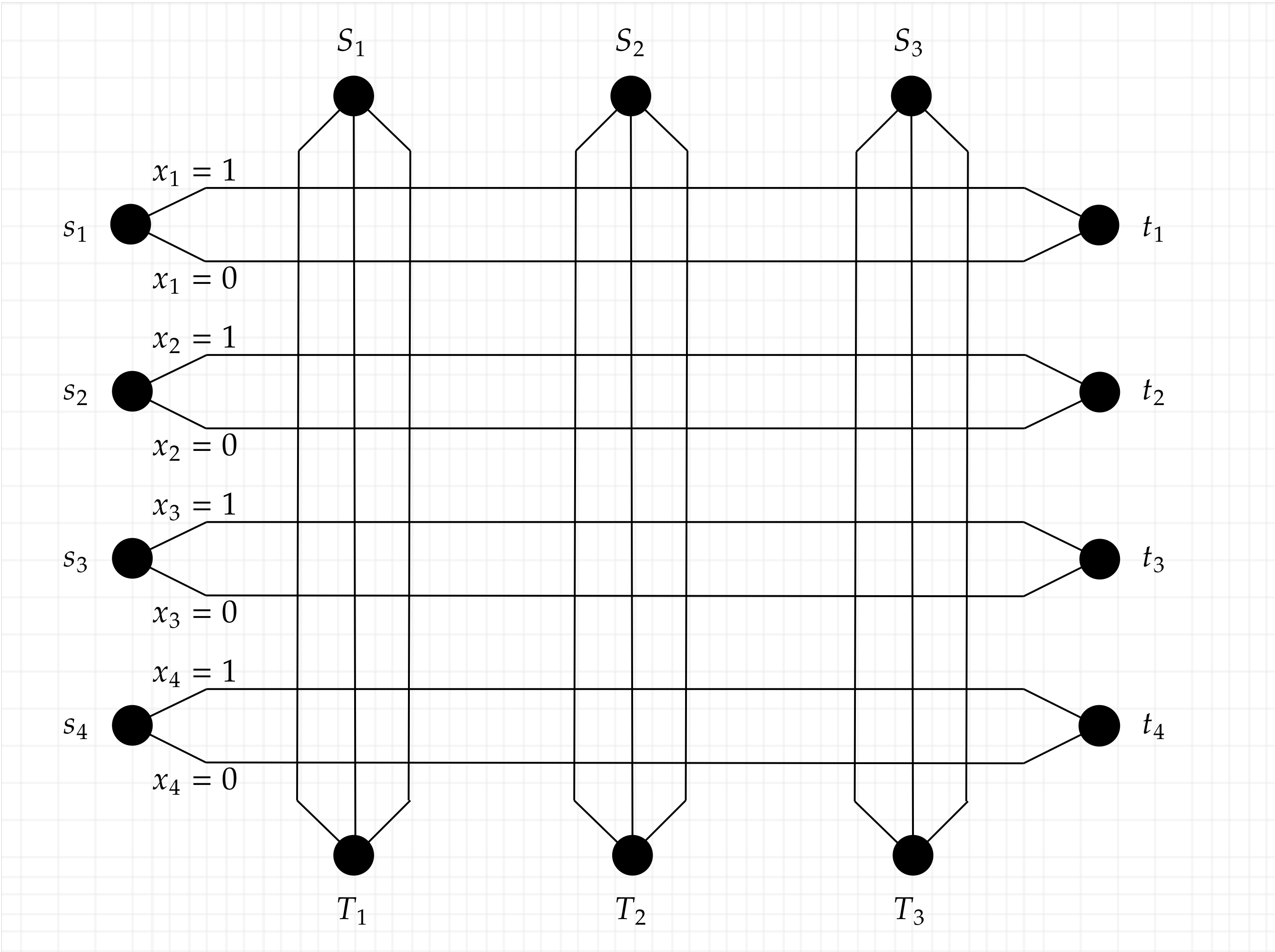
We also need a mechanism to say a variable is true or false. This is done by giving each variable path two possible "tracks"---say upper for true, lower for false---where the paths connecting the start $s_i$ and terminal $t_i$ for each variable $x_i$ will run horizontally.

Next, we need a mechanism to say a clause is satisfied. Naturally, we make $C_j$ be satisfied if and only if we can get a path from $S_j$ to $T_j$, and it helps to imagine these paths running vertically. As for which literal satisfies it, we create three vertical "tracks," one for each literal in the clause.

Last, we need to say how $C_j$ is satisfied when a literal $\ell_i$ in it is or is not made true. The idea is:

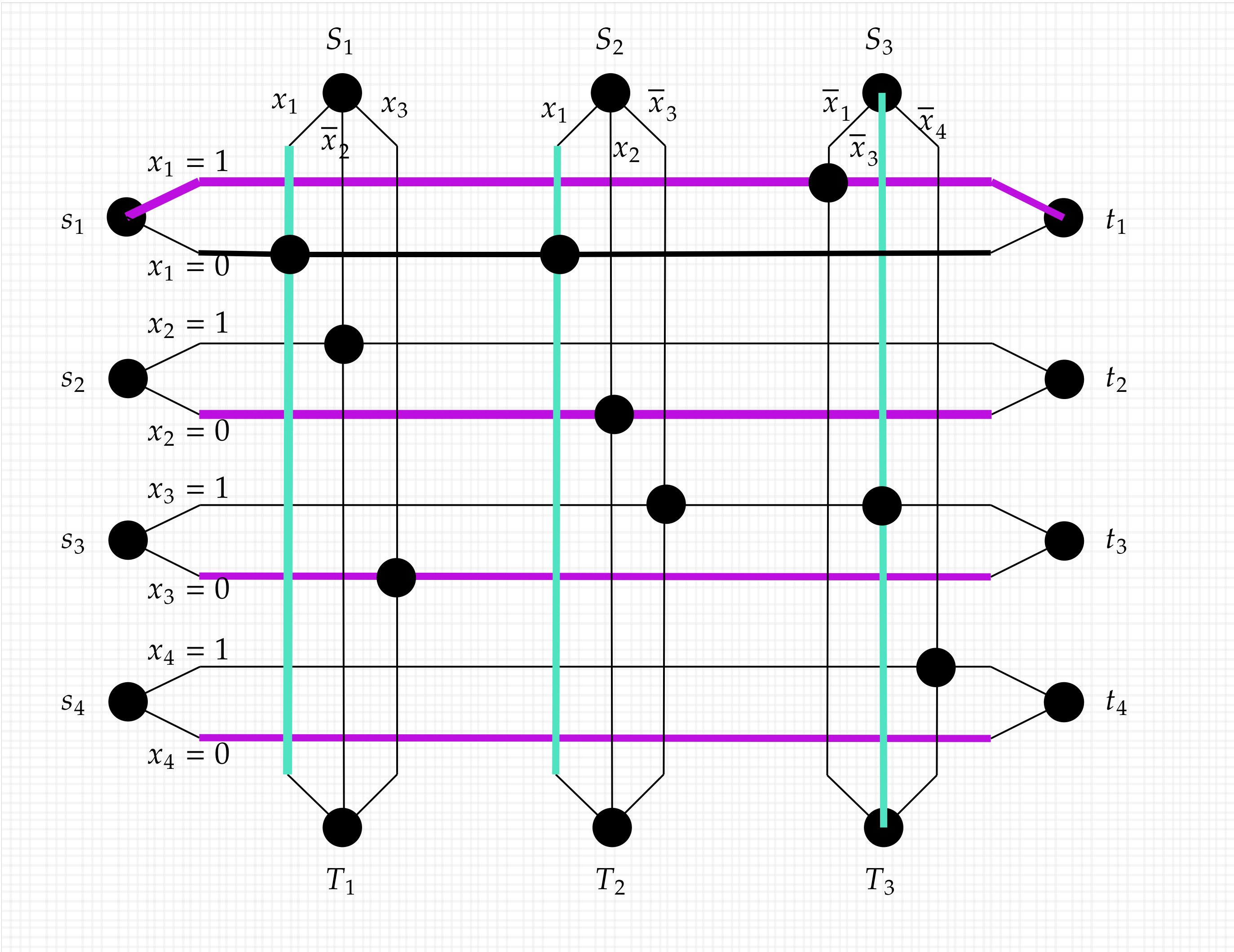
Make the horizontal track in which $\ell_i$ is false block the vertical track for $\ell_i$ in $C_j$.

Whereas, the horizontal track in which $\ell_i$ is true allows the vertical track to pass through---they might look like they cross when drawn in the plane, but really one goes over the other---without an "at-grade intersection" represented by a node. Here is the field of play for $n = 4$, $m = 3$:




Here is the whole thing for the formula used before:

$$\phi = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee \overline{x}_3 \vee \overline{x}_4)$$




$$\phi = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee \overline{x}_3 \vee \overline{x}_4)$$

Once again, the size of the graph is $O(m+n)$ nodes and the complexity of the reduction function is similar. The one detail that is slightly trickier than before is that the edges out of the node for the $x_1$ in clause $C_2$ have to "know" that there was an $x_1$ in clause $C_1$, so the horizontal edge goes left to there rather than all the way back to the start node $s_1$. But by labeling nodes $\pm x_{ij}$ for $x_i$ or $\overline{x}_i$ in clause $C_j$ and sorting on one index or the other, one can determine all the edges needed. Such use of sorting is typical of **quasilinear** time. (Note that each vertical track has exactly one node, so this issue does not arise for the vertical edges.) The correctness of this reduction is a self-study exercise.

Moving on from here, what we note is that generating the graph edge-by-edge only requires space to manipulate indices of size $O(\log n)$. If we back off the use of sorting, we can generate $G_\phi$ node-by-node, needing only to store the current indices $i$ and $j$, on pain of hunting through $\phi$ multiple times---for running time order $(m+n)^2$. The reduction function then runs in logarighmic space. The notation is:

- The class of languages decidable in $O(\log n)$ space is denoted by either L or DLOG.
- The class of functions computable in $O(\log n)$ space is denoted by FL.

**Definition**: A language $A$ mapping-reduces to a language $B$ in logarithmic space (logspace), written $A \leq_m^{\log} B$, if $A$ mapping-reduces to $B$ via a function $f \in$ FL.

- Every regular reduction is a logspace reduction, in fact, computable in zero space.
- The non-regular reduction $f(x) = \langle N_A, x, @^{p(|x|)} \rangle$ from an arbitrary language $A$ in NP to $K_{\text{NP}}$ is computable in logspace because the arithmetic needed to compute a polynomial function $p(n)$ is computable in $O(\log n)$ space.
- The Cook-Levin reduction can be computed in $O(\log n)$ space because it only needs to keep track of one wire label $w_k$ at a time and determine which gates it comes from and goes to. Thus 3SAT is complete for NP under the $\leq_m^{\log}$ relation too.
- The reduction functions by component design are likewise in FL.

One advantage of $\leq_m^{\log}$ is that it enables finer distinctions in complexity.

- The class of languages decidable by NTMs running in $O(\log n)$ space is denoted by NL.

**Graph Accessibility Problem** (**GAP**)
**Instance**: A directed graph $G = (V, E)$ and nodes $s, t \in V$.
**Question**: Is there a path from $s$ to $t$ in $G$?

**Theorem**: **GAP** is in NL.

**Proof.** (by picture) A nondeterministic Turing machine $N$ can guess a path from $s$ to $t$ (when one exists) by keeping $t$ on one tape and maintaining its current node (initially $s$) on another.

Read-only input tape, not counted against space usage

$\boxed{\langle (s,p), (s,q), (p,r), (p,u), (q,v), \ldots, \textit{Graph } G = (V,E) \textit{ given as edge list} \rangle, \textit{ plus } s;\ t}$

$O(\log n)$-size
bounded tapes

$\boxed{s}$ Always has current node, initially $s$.

$\boxed{t}$ Fixed copy of $t$ for easy comparison

$\boxed{n}$ Since there is a path of length $< n$ if there is one at all, $N$ can decrement from $n$ at each step and halt and reject on $0$.

$N$ first copies $s$, $t$, and $n = |V|$ onto worktapes as shown. A nondeterministic computation path by $N$

begins with a "guess" of an out-neighbor of $s$, such as $p$ or $q$ above. This overwrites $s$, and then $N$ compares it against $t$ to see if the goal has been reached. If so, this computation path by $N$ accepts, and this makes the whole machine accept. If not, $N$ decrements its third tape, which acts as a "countdown clock." If the clock hits $0$, this particular path by $N$ halts and does not accept (other paths might accept); thus we enforce the condition that $N$ cannot have computations that loop forever. Else, $N$ reiterates the "guess" step to move to another node.

The point is that $N$ need only maintain one current node along a path. On a lucky guess of those neighbors that make progress toward $t$, eventually reaching $t$ itself, the path will accept. Thus $N$ accepts $\langle G, s, t\rangle$ if and only if there is a path from $s$ to $t$, and using only the work space shown---which is logarithmic. (Whereas, a deterministic TM using breadth-first search would need linear space to store all the nodes already visited.) ⊠