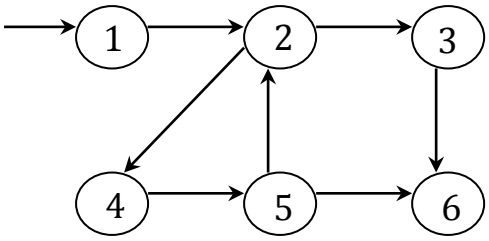
第八章

8.3 考虑广义地理学游戏，其中起始节点就是又无源箭头指入的节点。选手 I 有必胜策略吗？选手 II 呢？给出理由。



I	II	I	II	I	Winner
2	3	6	×		I
	4	5	6	×	II

由表上来看选手 II 有必胜策略

I2→II4(不能选 3)→I5→II6(不能选 2)→I×。

8.4 证明 PSPACE 在并、补和星号运算下封闭。

证明：(1) 并：

对任意 L_1, $L_2 \in$PSPACE，设有 n^a 空间图灵机 M_1 和 n^b 空间图灵机 M_2 判定它们，且 c=max{a,b}。

对 $L_1 \cup L_2$ 构造判定器 M:

M=“对于输入字符串 $w=w_1w_2...w_n$:

1) 初始化将带子改写为 $w_1w_1\ w_2w_2...w_nw_n$。
2) 在 w 上分别运行 M_1 和 M_2，即在奇数格运行 M_1 和在偶数格运行 M_2。
3) 若有一个接受则接受，否则拒绝。”

空间复杂度：$n^a+n^b=O(n^c)$,

所以 $L_1 \cup L_2$ 属于 PSPACE，即 PSPACE 在并的运算下封闭。

(2) 补：

对任意 L_1 属于 PSPACE，设有空间 n^a 判定器 M_1 判定它。令 L_2 为 L_1 的补，构造 L_2 的判定器 M:

M=“对于输入字符串 w :

1) w 上运行 M_1。
2) 若 M_1 接受则拒绝，若 M_1 拒绝则接受。”

空间复杂度为：n^a。所以 L_2 属于 PSPACE 类，即 PSPACE 在补的运算下封闭。

(3)星号：

设 M 为判定 A 的空间 n^a 图灵机。设计如下 TM：

D=“对于输入 $y=y_1y_2...y_n$:

1) 若 y= ε ，则接受；

2) 对于 i,j＝1,2,...,n(i≥j)重复(3)。

3) 在 $y_iy_{i+1}...y_j$ 上运行 M。若接受，则令 T(i, j)=“yes”。

4) 重复下面步骤直到表 T 不再改变。

5) 对于 i,j＝1,2,...,n(i≥j)重复下面步骤。

6) 若 T(i,j)=“yes”，转(5)。否则继续。

7) 对于 k = i,...,j-1,若 T(i,k)=T(k+1,j)= “yes”,则令 T(i,j)=“yes”.

8) 若 T(1, n)=yes，则接受；否则拒绝。”

运行空间：第 3)步模拟 M 运行需要 n^a 空间，存储表 T 需要 n^2 空间，所以 A^*属于 PSPACE。

8.5 证明 NL 在并、补和星号运算下封闭。

证明：(1) 并：

对任意 $L_1, L_2 \in NL$，设有 O(logn)空间图灵机 M_1 和 M_2 判定它们，且 c=max{a,b}。

对 $L_1 \cup L_2$ 构造判定器 M:

M=“对于输入字符串 $w=w_1w_2...w_n$:

1) 初始化将带子改写为 $w_1w_1\ w_2w_2...w_nw_n$。

2) 在 w 上分别运行 M_1 和 M_2，即在奇数格运行 M_1 和在偶数格运行 M_2。

3) 若有一个接受，则接受；否则拒绝。”

空间复杂度：2O(logn)=O(logn),

所以 $L_1 \cup L_2$ 属于 NL，即 NL 在并的运算下封闭。

(2) 并：

对任意 $L_1, L_2 \in NL$，设有 O(logn)空间图灵机 M_1 和 M_2 判定它们，且 c=max{a,b}。

对 $L_1 \cap L_2$ 构造判定器 M:

M=“对于输入字符串 $w=w_1w_2...w_n$:

1) 初始化将带子改写为 $w_1w_1\ w_2w_2...w_nw_n$。
2) 在 w 上分别运行 M_1 和 M_2，即在奇数格运行 M_1 和在偶数格运行 M_2。
3) 若两个都接受，则接受；否则，拒绝。”

空间复杂度：2O(logn)=O(logn),

所以 $L_1 \cap L_2$ 属于 NL，即 NL 在交的运算下封闭。

(3)星号：

设 M 为判定 A 的空间 O(logn)图灵机。设计如下 TM：

D=“对于输入 $y=y_1y_2...y_n$，

1) 令 i=1.
2) 若 i≤n,非确定的选取 j≤n,重复以下步骤。
3) 以二进制将 i,j 存储于工作带。
4) 对于 $y_iy_{i+1}...y_j$ 运行 M。
5) 若 M 拒绝，则拒绝；若 M 接受，则令 i=j+1。
6) 接受。

运行空间：第 4)步模拟 M 运行需要 O(logn)空间，存储 i,j 需要 2logn 空间，所以 A^*属于 NL。

8.6 证明 PSPACE 难的语言也是 NP 难的。

证明:称 B 是 PSPACE 难的，若对于任意 A 属于 PSPACE,都有 $A \leq_P B$。

称 B 是 NP 难的，若对于任意 A 属于 NP,都有 $A \leq_P B$。

现在设 B 是 PSPACE 难的。对于任意 A 属于 NP，由于 NP⊆PSPACE，所以 $A \leq_P B$。这说明 B 也是 NP 难的。

8.7 证明 A_{DFA} 属于 L。

证明：构造如下双带只读输入 TM：

P="对于输入<M,w>,M 是 DFA，w 是串：

1) 在 w 上模拟 M 运行，每次以二进制记录当前状态编号和当前所读符号个数。
2) 若 w 读完后 M 接受，则接受；否则，拒绝。"

设 M 的状态数加 w 长度为 n 则 P 需要 2logn 空间存储当前状态编号，已读符号个数。

3. (Sipser 8.9) A *ladder* is a sequence of strings $s_1, s_2, \ldots, s_k$, wherein every string differs from the preceding one in exactly one character. For example the following is a ladder of English words, starting with "head" and ending with "free": head, hear, near, fear, bear, beer, deer, deed, feed, feet, fret, free.

   Let $LADDER_{\text{DFA}} = \{\langle M, s, t\rangle \mid M \text{ is a DFA and } L(M) \text{ contains a ladder of strings, starting with } s \text{ and ending with } t\}$. Show that $LADDER_{\text{DFA}}$ is in **PSPACE**.
   [30 points]

   SOLUTION: It suffices to show that $LADDER \in$ **NPSPACE**. The idea is as follows: given $\langle M, s, t\rangle$, reject if $|s| \neq |t|$. Otherwise, consider a graph G of exponential size whose vertices are indexed by strings in $\Sigma^{|s|}$, and there is a directed edge from w_1 to w_2 iff w_1 and w_2 differ in exactly one character, and $w_1, w_2 \in L(M)$. Then, $\langle M, s, t\rangle \in LADDER$ iff there is a path from s to t in G. This we can check in **NPSPACE** by guessing the path (akin to the **NL** algorithm for *PATH*), and at each step, storing only the name of current vertex (which is a string in $\Sigma^{|s|}$. To guess the path, at vertex w_1, we will nondeterministic select a new vertex w_2 that differs from w_1 in exactly one character, and verify that M accepts w_2. (We can ensure that the machine always halts by keeping a counter and incrementing it with each guess, and rejecting when the counter hits $|\Sigma|^{|s|}$. This is because if there exists a chain from s to t, then there exists one of length at most $|\Sigma|^{|s|}$ by removing loops.)

8.11 Show that if every NP-hard language is also PSPACE-hard, then PSPACE = NP.

We prove that if NP-hard $\subseteq$ PSPACE-hard, then PSPACE = NP. We know that NP $\subseteq$ PSPACE by the arguments given in Sipser (see p. 282). We must show the opposite inclusion, namely that PSPACE $\subseteq$ NP. To see this, note that SAT $\in$ PSPACE and SAT is NP-complete. By the definition of NP-complete, SAT is also NP-hard, and by our assumption, SAT is PSPACE-hard. Thus, for any $L \in$ PSPACE, we have $L \leq_P$ SAT. But since SAT $\in$ NP, we have that L $\in$ NP. Therefore PSPACE $\subseteq$ NP.

8.13

Problem 1: (Sipser 8.13) Show that TQBF restricted to formulas where the part following the quantifiers is in conjunctive normal form is still PSPACE-complete.

Solution 1: (borrowed from Spring 2001) To show that TQCNF is PSPACE-complete, we need to show that TQCNF $\in$ PSPACE and that TQBF $\leq_P$ TQCNF.

TQCNF $\in$ PSPACE because TQCNF is a special case of TQBF, so we can use the same decider for TQCNF as for TQBF.

To show that TQBF $\leq_P$ TQCNF, we need to construct a polynomial time computable translation function, f, such that $\langle\phi\rangle \in$ TQBF $\iff f(\langle\phi\rangle) \in$ TQCNF. The following procedure F indicates, with one bug, how to compute f:

F = "On input $\langle\phi\rangle$:

1. Let Q be the part of $\langle\phi\rangle$ containing all the quantifiers.
2. Let ϕ be the part of $\langle\phi\rangle$ containing no quantifiers.
3. Construct ϕ' that is the equivalent to ϕ in conjunctive normal form:
   - Using DeMorgan's laws ensure that all NOTs in ϕ are only on individual variables and not on expressions.
   - Apply distributivity

   - Using DeMorgan's laws ensure that all NOTs in ϕ are only on individual variables and not on expressions.
   - Apply distributivity

     $$(A \wedge B) \vee C = (A \vee C) \wedge (B \vee C)$$

     where possible.
   - Repeat until no more conversions of the above type are possible.
4. Output $Q[\phi']$."

Since ϕ' is equivalent ϕ, we have that $\langle\phi\rangle \in$ TQBF $\iff f(\langle\phi\rangle) \in$ TQCNF.

Now we consider the time complexity of the reduction, and we discover the bug. The conversion from ϕ to ϕ' may blow up exponentially: the problem is in the applications of distributivity, because each application of the law converts one copy of C into two copies, potentially doubling the size of the formula at each application.

The fix is to construct ϕ' equivalent to ϕ such that ϕ' is a *quantified* CNF Boolean formula but $|\phi'|$ remains bounded by a polynomial in $|\phi|$. The trick for doing this is to show a slightly more general construction: construct from any Boolean formula, ϕ, a quantified Boolean formula, $E_{\phi,x}$, with one extra free variable, x, such that (i) $E_{\phi,x}$ is equivalent to the formula $x \equiv \phi$, (ii) $E_{\phi,x}$ would be in CNF if we ignored the quantifiers, and (iii) $|E_{\phi,x}|$ is bounded by a polynomial in $|\phi|$. Namely,

***8.14** The cat-and-mouse game is played by two players, "Cat" and "Mouse," on an arbitrary undirected graph. At a given point, each player occupies a node of the graph. The players take turns moving to a node adjacent to the one that they currently occupy. A special node of the graph is called "Hole." Cat wins if the two players ever occupy the same node. Mouse wins if it reaches the Hole before the preceding happens. The game is a draw if a situation repeats (i.e., the two players simultaneously occupy positions that they simultaneously occupied previously, and it is the same player's turn to move).

$$\textit{HAPPY-CAT} = \{\langle G, c, m, h\rangle \mid G, c, m, h \text{ are respectively a graph, and positions of the Cat, Mouse, and Hole, such that Cat has a winning strategy if Cat moves first}\}.$$

Show that *HAPPY-CAT* is in P. (Hint: The solution is not complicated and doesn't depend on subtle details in the way the game is defined. Consider the entire game tree. It is exponentially big, but you can search it in polynomial time.)

Answer: We first note that the game can have only $2n^2$ configuration, defined by position of the cat, position of the mouse and if it is cat's turn. So we can construct a directed graph consisting of $2n^2$ nodes where each node corresponds to a game configuration and there is an edge from node u to node v, if we can go from configuration corresponding to u to configuration corresponding to v in one move. Now following algorithm solves HAPPY-CAT.
1. Mark all nodes (a,a,x) where a is a node in G, and $x \in \{true, false\}$.
2. If for a node u = (a,b,true), there is a node v = (c,b,false) which is marked and (u,v) is an edge then mark u.
3. If for a node u = (a,b,false), all nodes v = (a,c,true) are marked and (u,v) is an edge then mark u.
4. Repeat steps 2 and 3 until no new nodes are marked.
5. Accept if start node s = (c,m,true) is marked.

The algorithm takes $O(n^2)$ time to perform step 1, $O(n^2)$ time per iteration of the loop and loop is executed $O(n^2)$ times. Hence runs in polynomial time.

Problem 8.16

Answer:
Part a: Consider following algorithm
On input F:
1. For each string s such that $|s| < |F|$, if s is a valid representation of a formula (this can be easily checked) which is equivalent to F (this can be checked in polynomial space by evaluating both F and s over all possible truth assignments and comparing the results) then reject.
2. If all string have been tried without rejecting, accept.

Correctness of the algorithm should be evident. Only space used by the algorithm is for storing forumula F, string s and current assignment of literals which amounts to polynomial space. Hence MIN-FORUMLA $\in$ PSPACE.
Part b: It fails because it is not known if we can verify equivalance of two boolean formulae is polynomial time.

Solution to Exercise 1 (Sipser's Problem 8.17) Let A be the language of properly nested parentheses. For example, `(())` and `((()(()))())` are in A, but `)(` is not. We show that A is in **L**.

Proof: Our **L** machine that decides A works as follows.

"On input $w \in \{(,)\}^*$:

1. initialise counter c to 0 on the work tape
2. read the next letter ℓ of the input word
3. case ℓ of

   '(' : increment c

   ')' : if $c = 0$ then *reject*, otherwise decrement c

   '$\sqcup$' : if $c = 0$ then *accept*, else *reject*
4. go to 2"

(Sipser 8.23) Define $UCYCLE = \{\langle G \rangle \mid G$ is an undirected graph that contains a simple cycle$\}$. Show that $UCYCLE \in \mathbf{L}$. (Note: G may not be connected.)
[20 points]

Hint: We can try to search the tree by always traversing the edges incident on a vertex in lexicographic order i.e. if we come in through the ith edge, we go out through the $(i+1)$th edge or the first edge if the degree is i. How does this algorithm behave on a tree? How about a graph with a cycle?

SOLUTION: Note that the above process performs a DFS on a tree and we always come back to a vertex through the edge we went out on. However, if the graph contains a cycle, there must exist at least one vertex u and at least one starting edge (u, v) such that if we start the traversal through (u, v), we will come back to u through an edge different that (u, v). Hence, we enumerate all the vertices and all the edges incident on them, and start a traversal through each one of them. If we come back to the starting vertex through an edge different than the one we started on, we declare that the graph contains a cycle. Since we can enumerate all vertices and edges in logspace and also remember the starting vertex and edge using logarithmic space, this algorithm shows $UCYCLE \in \mathbf{L}$.

(Sipser 8.25) An undirected graph is bipartite if its nodes can be divided into two sets such that all edges go from a node in one set to a node in the other set. Show that a graph is bipartite iff it does not contain a cycle that contains an odd number of nodes. Let $BIPARTITE = \{\langle G \rangle \mid G$ is a bipartite graph$\}$. Show that $BIPARTITE \in \mathbf{NL}$.
[25 points]

SOLUTION: Dividing the graph into two sets is the same as 2-coloring the graph such that all the edges are between vertices of different color. If the graph has an odd cycle, then the odd cycle in particular cannot be 2-colored and hence the graph cannot be bipartite. If the graph does not contain an odd cycle, we consider the DFS tree (or forest, if it is not connected) of the graph and color the alternate levels (say) red and blue. By construction, all the tree edges are between vertices of different color. Suppose an edge not in the tree connects two vertices of the same color. But these vertices must be connected by a path of even length in the tree and this extra edge will create an odd cycle, which is not possible. Hence, the above 2-coloring is a valid one which proves that the graph is bipartite.

We show that $BIPARTITE \in \mathbf{coNL}$ or $\overline{BIPARTITE} \in \mathbf{NL}$, which suffices since $\mathbf{NL} = \mathbf{coNL}$. However, $\overline{BIPARTITE}$ is precisely the set of all graphs which contain an odd cycle. We guess a starting vertex v, guess an (odd) cycle length l and go for l steps from v, guessing the next vertex in the cycle at each step. If we come back to v (we can remember the starting vertex in logspace), we found a tour of odd length. Since any odd tour must contain an odd (simple) cycle, we accept and declare that the graph is non-bipartite.

8.27 Recall that a directed graph is ***strongly connected*** if every two nodes are connected by a directed path in each direction. Let

$$\textit{STRONGLY-CONNECTED} = \{\langle G \rangle \mid G \text{ is a strongly connected graph}\}.$$

Show that *STRONGLY-CONNECTED* is NL-complete.

We'll show that STRONGLY CONNECTED is **NL**-complete. First, we observe that it is in **NL**: we can go through all (ordered) pairs of vertices (x, y) and nondeterministically guess a path from x to y one vertex at a time (as we did for S-T CONNECTIVITY). We reject whenever one of these paths fails to actually reach y from x. If the graph is strongly connected, then some sequence of guesses will succeed, and we will accept; otherwise no sequence of guesses will succeed (in particular, there will be some pair (x, y) with no directed path from x to y), and all computation paths will reject.

Now, we reduce S-T CONNECTIVITY (which we know to be **NL**-complete) to STRONGLY CONNECTED as follows. Given a directed graph $G = (V, E)$ and vertices s and t we produce the directed graph $G' = (V, E')$ where $E' = E \cup \{(v, s) : v \in V\} \cup \{(t, v) : v \in V\}$. We can easily perform this reduction in logspace, as we only need to step through the vertices one by one, adding the required two edges for each. We claim that G' is strongly connected if and only if there is a directed path from s to t in G. ($\Leftarrow$) If there is a directed path from s to t in G, then to get from v to v' in G' we can traverse the edge from v to s, then the directed path from s to t, and finally the edge from t to v'. ($\Rightarrow$) If G' is strongly connected, then in particular there must be a directed path from s to t in G', and then there also must be a directed path from s to t in G, since we only added ingoing edges to s and outgoing edges to t (which cannot have introduced a directed path from s to t if there was not one to begin with).

By the claim, we have reduced S-T CONNECTIVITY to STRONGLY CONNECTED in logspace, and thus STRONGLY CONNECTED is **NL**-complete. Therefore, if STRONG CONNECTED is in **L**, then **NL** = **L**. Note that for this conclusion it would have been sufficient to only prove that STRONGLY CONNECTED is **NL**-hard.

8.29 证明 A_{NFA} 是 NL 完全的。

证明：首先证明 A_{NFA} 属于 NL。构造图灵机

N="对于输入<M,w>,其中 M 是 NFA, $w=w_1w_2...w_n$ 是串:

1) 设置当前状态为起始状态，以二进制记录其编号。
2) 对于 i=1,2,...,n,n+1 重复以下步骤。
3) 非确定的选取 $0 \le j \le k$，其中 k 是 M 的状态数。根据转移函数，由ε箭头转移 j 次，每次非确定的选择下一个状态。
4) 若 $i \le n$，根据转移函数，当前状态和 w_i，非确定的选择下一个状态。
5) 若当前状态是接受状态，则接受。"

再证明 A_{NFA} 是 NL 完全的，由于 PATH 是 NL 完全的，所以只需证明 PATH 可以多项式归约到即可。

对于<G,s,t>，其中 G 是有向图，s,t 是 G 的两个节点。构造 NFA N，以 G 为 N 的状态转移图，在每个有向边上标上ε，以 s 为起始状态，以{t}为接受状态集。再来构造归约，令

$$f(<G,s,t>)=<N, \varepsilon>.$$

最后，由 f 的构造可知$<G,s,t> \in PATH \Leftrightarrow <N, \varepsilon> \in A_{NFA}$。

Problem 8.30

Answer: We show that $N = \overline{E_{DFA}}$ is NL-complete and then use the fact that NL = co-NL to conclude that E_{DFA} is NL-complete.
Part I: N is in NL.
We can non-deterministically guess the string accepted by N and then verify that string is indeed accepted in log-space as all we need to keep track of is current state of N and current position in the string.

Part II: N is NL-complete.
We reduce PATH to N as follows. Given instance $< G(V, E), s, t >$ of PATH, Let k denote maximum outdegree of any node in G. So we construct a DFA $D = \{Q, \Sigma, \delta, q_0, F\}$ as follows. $Q = V \cup \{d\}$, where d is new dead state. $\Sigma = \{a_1, a_2, .., a_k\}$ be arbitrary set of input alphabet. Each outgoing edge from every state of D is labeled with arbitrary symbol from Σ. Note that we always have enough symbols to unquiely label outgoing edges from given state. If any state has less than k outgoing edges, we add transition from that state to dead state for all symbols not used in any transition from that state. Finally we set $q_0 = s$ and $F = \{t\}$. It is easy to see that graph G has a path from s to t if and only if DFA D accepts at least one string.

so that u has transitions for all letters. This step is done in log-space, since we remember at most one vertex and two edges at a time.

It is then easy to set s to be the start state, and t to be the final state. So we can do this reduction in log-space, and so E_{DFA} is NL-complete.

*8.31 Show that *2SAT* is NL-complete.

Since NL = coNL, it suffices to show that $\overline{\text{2SAT}}$ in NL-complete. In order to prove $\overline{\text{2SAT}} \in$ NL, construct the graph G as in Exercise 2.2.3. Convince yourself that this construction can be done in (deterministic) log-space. Now nondeterministically choose a vertex x in G. Use an NL algorithm for PATH to check if G contains both $x, \bar{x}$ and $\bar{x}, x$ paths. If both paths exist, accept, else reject. The correctness of this algorithm is established by the claim in Exercise 2.2.3.

In order to prove the NL-hardness of $\overline{\text{2SAT}}$, I show that PATH $\leqslant_L \overline{\text{2SAT}}$. Given G, s, t we have to construct a 2-cnf formula ϕ such that G has an s, t-path if and only if ϕ is unsatisfiable. Let $V(G) = \{s, t, y_1, \ldots, y_m\}$. The resulting formula ϕ consists of $m + 1$ variables $x, y_1, \ldots, y_m$. The vertex s is identified (relabeled) with x and the vertex t with $\bar{x}$. The clauses of ϕ will be $x \vee x$ and $\bar{u} \vee v$ for every edge (u, v) of G. That completes the construction of ϕ.

In order to show that this construction works, first assume that G has an s, t-path. Let this path be $s, u_1, \ldots, u_k, t$. Then ϕ contains the clauses $(\bar{x} \vee u_1), (\bar{u}_1 \vee u_2), \ldots, (\bar{u}_k, \bar{x})$. If we assign $x = 1$, at least one of these clauses is not satisfied (easy check). On the other hand, if $x = 0$, then the clause $(x \vee x)$ of ϕ is not satisfied. So ϕ is zero for any assignment of $x, y_1, \ldots, y_m$.

Conversely, suppose that G contains no s, t-path. Let U be the set of vertices in G reachable from s, V the set of vertices from which t is reachable and $W := V(G) \setminus (U \cup V)$. By hypothesis and construction, U, V, W are pairwise disjoint, and there are no edges from U to $V \cup W$ or from $U \cup W$ to V. Let me assign the true value to every variable in $U \cup W$ (including x) and the false value to every variable in V (including the *literal* $\bar{x}$). It is now simple to check that ϕ is satisfied for this truth assignment.