Formal Modeling with Propositional Logic

Assaf Kfoury

February 6, 2017 (last modified: September 22, 2018)

Contents

1 The Pigeon Hole Principle **2**

2 Graph Problems **3**

2.1 Paths in Directed Graphs 3

2.2 Coloring of Undirected Graphs 8

3 A Two-Player Game: Tic-Tac-Toe **9**

4 A One-Player Game: Yashi **11**

4.1 How to Automate the Yashi Game? 14

4.2 Using SAT Solvers 16

4.3 Using SMT Solvers 19

4.4 Using SMT Solvers with an API 20

Before we can resort to an automated tool (to test or verify a property, to analyze the behavior software or hardware system, etc.), we have to do some *formal modeling*, *i.e.*, formally represent a system and its properties in the syntactic conventions that the tool understands and can process. Sometimes this formal modeling is straightforward; sometimes it is not; and, most typically, there are several ways of doing it for the same application and with the same tool (which can greatly affect the performance of the latter).

1 The Pigeon Hole Principle

For every natural number $n \geqslant 2$, the *Pigeon Hole Principle* (PHP) states: "If n pigeons sit in $(n-1)$ holes, then some hole contains more than one pigeon." We want to formalize PHP in propositional logic (PL). There are different ways of doing this, but perhaps the most natural is:

- Use propositional atom $P_{i,j}$ to indicate that *pigeon* i is in *hole* j, where $1 \leqslant i \leqslant n$ and $1 \leqslant j < n$; *i.e.*, $P_{i,j}$ is *true* if *pigeon* i is in *hole* j, and $P_{i,j}$ is *false* if *pigeon* i is not in *hole* j.[1]

With this denotation of "*pigeon* i is in *hole* j" we can formalize PHP with the following formula φ_n:

$$\varphi_n \;:=\; \bigwedge_{1\leqslant i\leqslant n} \Big(\bigvee_{1\leqslant j<n} P_{i,j} \Big) \rightarrow \bigvee_{1\leqslant i<k\leqslant n} \Big(\bigvee_{1\leqslant j<n} (P_{i,j} \wedge P_{k,j}) \Big)$$

where $\bigwedge$ and $\bigvee$ are shorthand notation to write long sequences of conjunctions and disjunctions, respectively. Note that we have formalized PHP with a separate φ_n for every $n \geqslant 2$. Informally, the part of the formula φ_n to the left of '$\rightarrow$' says:

> "for every pigeon i, there is a hole j, such that i is in j,"

and the part of φ_n to the right of '$\rightarrow$' says:

> "there are two distinct pigeons i and k, and there is a hole j, such that i and k are in j."

It is worth remembering that the logical connectives '$\bigwedge$' and '$\bigvee$' correspond to, respectively, *universal quantification* ("for all ..." or "for every ...") and *existential quantification* ("there is ..." or "there are ..." or "there exists ..."). This is an important correspondence, and there is much more to it than meets the eye. We will come back to it when we take up various forms of quantificational logic.

For the case $n = 3$, the formula φ_n reads as follows:

$$\begin{aligned} \varphi_3 \;=\; & (P_{1,1} \vee P_{1,2}) \wedge (P_{2,1} \vee P_{2,2}) \wedge (P_{3,1} \vee P_{3,2}) \rightarrow \\ & (P_{1,1} \wedge P_{2,1}) \vee (P_{1,2} \wedge P_{2,2}) \vee (P_{1,1} \wedge P_{3,1}) \vee (P_{1,2} \wedge P_{3,2}) \vee (P_{2,1} \wedge P_{3,1}) \vee (P_{2,2} \wedge P_{3,2}) \end{aligned}$$

which formally translates the following assertion in plain English:

[1] I use interchangeably the expressions 'propositional atom', 'propositional variable', 'Boolean atom', and 'Boolean variable' – although it is sometimes better to use 'variable' than 'atom' in order to emphasize there is another quantity (a function) whose values depend on varying propositional/Boolean arguments, which are naturally called 'variables', just as it is sometimes natural to use 'Boolean' instead of 'propositional' because there is some connection to be made with a Boolean algebra.

"if there are 3 pigeons and 2 holes, then there are two pigeons in the same hole."

Unless we know what each $P_{i,j}$ stands for, we cannot say that φ_3 is a formal translation of the preceding assertion in English – and it may indeed be a formal translation of some other assertion.

Exercise 1 Let f be a total function from $\{1,\ldots,n\}$ to $\{1,\ldots,n-1\}$, and use the same propositional atoms $P_{i,j}$ used in the formalization of PHP, to define PL formulas α_n and β_n such that:

1. α_n is satisfiable iff f is a total function on $\{1,\ldots,n\}$,
2. β_n is satisfiable iff f is not one-to-one.

Another way of formalizing PHP in PL is to define the formula $\psi_n := \alpha_n \to \beta_n$, which says "if f is a total function from $\{1,\ldots,n\}$ to $\{1,\ldots,n-1\}$, then f cannot be one-to-one." (You can take f as a binary relation which is said *univalent* or *right-unique*.) □

Exercise 2 We noted that we have formalized PHP with a separate φ_n for every $n \geqslant 2$. Can you write a single WFF Φ of propositional logic which is a formal representation of PHP for all $n \geqslant 2$? If 'yes', write the details of your proposed Φ; if 'no', justify your answer in a few lines. □

Exercise 3 This is an implementation exercise. The WFF φ_n that formalizes PHP is not only satisfiable, but also valid (or a tautology), *i.e.*, *all* valuations of the atoms $P_{i,j}$ should satisfy φ_n. Use your preferred SAT solver off-the-shelf, or any automated proof-assistant of your choice (Isabelle, Coq, Alloy, etc.), to establish that φ_n is valid. Do the implementation for at least two cases, $n = 3$ and $n = 4$. Do you notice any difference in the execution times? How would you handle the case $n = 10$? □

2 Graph Problems

There are many problems on graphs that can be formally translated into WFF's of propositional logic. The satisfiability of such a WFF φ (*some* truth-value assignment makes φ true) or its validity (*every* truth-value assignment makes φ true) can be then translated back into a solution of the corresponding graph-theoretic problem.

2.1 Paths in Directed Graphs

We make two simplifying assumptions about directed graphs, in order to make our formal modeling a little easier. We assume that in all our graphs:

(i) there are no self-loops, and
(ii) there is at most one edge between any two vertices.

With (i) and (ii), given a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, we can take E as:

$$E \subseteq \{\, \langle v, w\rangle \mid v, w \in V \text{ and } v \neq w \,\}.$$

These two assumptions do not fundamentally affect the analysis below, which is easily adjusted for the case when self-loops and/or multi-edges are allowed.

To simplify the notation a little, we write $\overline{v\,w}$ to denote $\langle v, w\rangle$, the directed edge from vertex v to vertex w. Given a graph $G = (V, E)$, we need to associate pieces of G with propositional atoms, before we can translate properties of G into WFF's of propositional logic. The simplest and natural way is to introduce a propositional atom $A_{\overline{vw}}$ for every ordered pair $\langle v, w\rangle \in V \times V$, which is then set to *true* or *false* depending on whether $\overline{v\,w}$ is or is not an edge of G.

Let $\mathcal{A} = \{\, A_{\overline{vw}} \mid v, w \in V\}$ denote the set of all propositional atoms, which is finite if V is finite and infinite if V is infinite. Every graph $G = (V, E)$, finite or infinite, induces a truth-value assignment $\sigma^G : \mathcal{A} \to \{\mathit{true}, \mathit{false}\}$ as follows:

$$\sigma^G(A_{\overline{vw}}) \;=\; \begin{cases} \mathit{true} & \text{if } \overline{v\,w} \in E, \\ \mathit{false} & \text{if } \overline{v\,w} \notin E. \end{cases}$$

Given a propositional WFF φ over $\mathcal{A}$, instead of writing $\sigma^G \models \varphi$, which is the standard way of indicating that σ^G satisfies φ, we can write more conveniently $G \models \varphi$, *i.e.*, $G \models \varphi$ iff $\sigma^G \models \varphi$, in words: "the graph G satisfies φ iff the truth-value assignment σ^G induced by G satisfies φ."

Let $\Gamma = \{\, \varphi_i \,\}_{i\in I}$ be a set of propositional WFF's over $\mathcal{A}$, indexed with $i \in I$. The index set I may be finite or infinite, allowing for the possibility that the graph G may be infinite. We write $G \models \Gamma$ to mean that $G \models \varphi_i$ for every $\varphi_i \in \Gamma$. If I is finite, then $G \models \Gamma$ is equivalent to the assertion $G \models \bigwedge_{i\in I} \varphi_i$ where $\bigwedge_{i\in I} \varphi_i$, also written as $\bigwedge \Gamma$, is now a single propositional WFF; using conjunction, the satisfaction of *every* φ_i in Γ is thus transformed into the satisfaction of a single propositional WFF.

Similarly, if I is a finite set, we can write $G \models \bigvee_{i\in I} \varphi_i$ where $\bigvee_{i\in I} \varphi_i$, also written as $\bigvee \Gamma$, is now a single propositional WFF, in order to assert that *some* φ_i in the set Γ is satisfied by G. This kind of transformation from a set Γ of propositional WFF's to a single propositional WFF is not possible if the set Γ is infinite.

Example 4 Let $G = (V, E)$ be a directed graph without self-loops and without multi-edges, according to our standing assumptions (i) and (ii). Suppose we want to test whether G is acyclic, *i.e.*, whether G contains no cycles. The existence of one or more cycle in G makes G cyclic.

There are several good algorithms for detecting the existence of a cycle in such a graph, and they can do this in low-degree polynomial time – provided G is a finite graph. For example, using the well-known fact that a directed graph G is acyclic iff G can be topologically sorted, we can first try to topologically sort G in time $\mathcal{O}(\big|V\big| + \big|E\big|)$; if we succeed, then G is acyclic.[2]

But the point of this example is to write a propositional WFF, or a set Γ^G_{DAG} of propositional WFF's, such that: G is a directed acyclic graph (or DAG) iff $G \models \Gamma^G_{\text{DAG}}$.

Let $\vec{v}$ be a sequence of $k \geqslant 2$ distinct vertices in G, say, $\vec{v} = (v_1, v_2, \ldots, v_k)$. The sequence $\vec{v}$ corresponds to a cycle in G iff each of the following vertex pairs are edges of G:

$$\overline{v_1\,v_2}\,,\quad \overline{v_2\,v_3}\,,\quad \overline{v_3\,v_4}\,,\quad \ldots\quad ,\quad \overline{v_k\,v_1}$$

This is equivalent to saying that the propositional WFF $\varphi_{\vec{v}}$ defined by:

$$\varphi_{\vec{v}} \;:=\; A_{\overline{v_1v_2}} \;\wedge\; A_{\overline{v_2v_3}} \;\wedge\; A_{\overline{v_3v_4}} \;\wedge \cdots \wedge\; A_{\overline{v_kv_1}}$$

[2]Look up the definition of *topological sorting*, and its time complexity, in any standard text on graph algorithms.

is made *true* by the truth-value assignment σ^G. Hence, the sequence $\vec{v}$ corresponds to a cycle in G iff $G \models \varphi_{\vec{v}}$. Define now the set $\Gamma^G_{\text{DAG},k}$ of propositional WFF's as follows:

$$\Gamma^G_{\text{DAG},k} \;:=\; \Big\{ \neg\varphi_{\vec{v}} \;\Big|\; \vec{v} \text{ is a sequence of } k \geqslant 2 \text{ distinct vertices in } G \Big\}.$$

Note the negation '$\neg$' in the WFF '$\neg\varphi_{\vec{v}}$' which is thus satisfied iff the sequence $\vec{v}$ does *not* correspond to a cycle in G. Hence, G contains no cycle with k distinct vertices iff $G \models \Gamma^G_{\text{DAG},k}$. If we define the set Γ^G_{DAG} by:

$$\Gamma^G_{\text{DAG}} \;:=\; \Gamma^G_{\text{DAG},2} \;\cup\; \Gamma^G_{\text{DAG},3} \;\cup\; \Gamma^G_{\text{DAG},4} \;\cup\; \cdots \;,$$

then G is acyclic iff $G \models \Gamma^G_{\text{DAG}}$. If $G = (V, E)$ is finite, with $\big|V\big| = n \geqslant 2$, then $\Gamma^G_{\text{DAG},k} = \varnothing$ for all $k > n$, in which case $\Gamma^G_{\text{DAG}} = \Gamma^G_{\text{DAG},2} \cup \cdots \cup \Gamma^G_{\text{DAG},n}$. □

Exercise 5 This is a continuation of Example 4. The set Γ^G_{DAG} of propositional WFF's is finite or infinite depending on whether $G = (V, E)$ is finite or infinite, respectively.

1. Suppose $G = (V, E)$ is finite, with $\big|V\big| = n \geqslant 2$. Write a single propositional WFF φ^G_{DAG} which is equivalent to the set Γ^G_{DAG}, *i.e.*, $G \models \varphi^G_{\text{DAG}}$ iff $G \models \Gamma^G_{\text{DAG}}$. What is the size of φ^G_{DAG} as a function of n? Count only the number of propositional atoms in φ^G_{DAG}.
2. Suppose $G = (V, E)$ is infinite. In this case, we cannot write a single propositional WFF φ^G_{DAG} which is equivalent to the set Γ^G_{DAG}. Nonetheless, how do you propose to use the infinite set Γ^G_{DAG} to test the acyclicity of G? □

Exercise 6 Read Example 4 and Exercise 5 before attempting this exercise.

Let $\vec{v}$ be a sequence of $k \geqslant 2$ distinct vertices in G, say, $\vec{v} = (v_1, v_2, \ldots, v_k)$. The sequence $\vec{v}$ corresponds to a path in G iff each of the following vertex pairs are edges of G:

$$\overline{v_1\, v_2}\,, \quad \overline{v_2\, v_3}\,, \quad \overline{v_3\, v_4}\,, \quad \ldots \quad , \quad \overline{v_{k-1}\, v_k}$$

Call such a sequence a *k-path*. This is equivalent to saying that the propositional WFF $\psi_{\vec{v}}$ defined by:

$$\psi_{\vec{v}} \;:=\; A_{\overline{v_1 v_2}} \;\wedge\; A_{\overline{v_2 v_3}} \;\wedge\; A_{\overline{v_3 v_4}} \;\wedge \cdots \wedge\; A_{\overline{v_{k-1} v_k}}$$

is *satisfied*, *i.e.*, made *true*, by the truth-value assignment σ^G. Note the difference between $\psi_{\vec{v}}$ here and $\varphi_{\vec{v}}$ in Example 4. Satisfaction of $\varphi_{\vec{v}}$ implies satisfaction of $\psi_{\vec{v}}$, but not the other way around. Hence, the sequence $\vec{v}$ corresponds to a k-path in G iff $G \models \psi_{\vec{v}}$.

1. For every $k \geqslant 2$, your task is to define a set $\Gamma^G_{\text{PATH},k}$ of propositional WFF's over $\mathcal{A}$ such that:

   there is $\psi \in \Gamma^G_{\text{PATH},k}$ such that $G \models \psi$ iff G contains a k-path.

Let the set Γ^G_{PATH} be defined by:

$$\Gamma^G_{\text{PATH}} \;:=\; \Gamma^G_{\text{PATH},2} \;\cup\; \Gamma^G_{\text{PATH},3} \;\cup\; \Gamma^G_{\text{PATH},4} \;\cup\; \cdots \;.$$

2. Let $G = (V, E)$ be finite, with $\big|V\big| = n \geqslant 2$. Your task is to select the smallest subset $\Delta \subseteq \Gamma^G_{\text{PATH}}$, where Γ^G_{PATH} is defined in Part 1, so that: There is a Hamiltonian path in G iff $G \models \bigvee \Delta$.[3] □

[3] A *Hamiltonian path* in a graph is a path that visits each vertex exactly once. In these notes, a *Hamiltonian graph* is one where there is a Hamiltonian path. (Some use a more restrictive definition: A graph is Hamiltonian if there is a Hamiltonian cycle, not just a path, in the graph.)

In preparation for Exercises 7 and 8, we mention additional facts about DAG's. Let $G = (V, E)$ be a finite DAG with $n \geqslant 2$ vertices, and consider a topologically sorting of G. Suppose the resulting linear ordering of V is $\vec{v}$, with:

$$\vec{v} \ := \ (v_1,\ v_2,\ \ldots\ ,\ v_n).$$

In general, the sequence $\vec{v}$ does not correspond to a path in G, *i.e.*, one or more of the vertex pairs in the following list:

$$\overline{v_1\ v_2},\ \overline{v_2\ v_3},\ \ldots\ ,\ \overline{v_{n-1}\ v_n}$$

are not edges in E. Call the preceding sequence $\vec{v}$ a *quasipath* in the DAG G. If we define another graph $G' = (V', E')$ by setting $V' := V$ and:

$$E' \ := \ E \cup \{\overline{v_1\ v_2},\ \overline{v_2\ v_3},\ \ldots\ ,\ \overline{v_{n-1}\ v_n}\},$$

then G' is a DAG, of which G is a subgraph,[4] where the preceding sequence $\vec{v}$ now forms a *path* in G', not just a *quasipath* in G'. Hence, the sequence $\vec{v}$ defines a Hamiltonian path in the supergraph G', but not in the subgraph G.

To conclude, if $G = (V, E)$ is a DAG, it does not necessarily mean that G is Hamiltonian, but it does mean there exists another graph $G' = (V', E')$ such that:

- G' is Hamiltonian, and
- G' is a supergraph of G.

Exercise 7 Read Example 4, Exercises 5 and 6, and the preceding comments about DAG's, before attempting this exercise.

1. Suppose $G = (V, E)$ is a graph and $G' = (V', E')$ a supergraph of of G. Think of G as being fixed, while G' varies over all supergraphs of G. The set of propositional atoms associated with G is $\mathcal{A} = \{A_{\overline{vw}} \mid v, w \in V\}$, and the set associated with G' is $\mathcal{X} = \{X_{\overline{vw}} \mid v, w \in V'\}$. We use the letter '$X$' to suggest that these are Boolean variables, rather than fixed Boolean atoms, ranging over the values of the different supergraphs G'. The truth-value assignment induced by G is $\sigma^G : \mathcal{A} \to \{true, false\}$, the truth-value assignment induced by a supergraph G' of G is $\sigma^{G'} : \mathcal{X} \to \{true, false\}$. We define a combined assignment $\sigma^{G,G'} : \mathcal{A} \cup \mathcal{X} \to \{true, false\}$ by setting:

$$\sigma^{G,G'}(Z) \ := \ \begin{cases} \sigma^G(A_{\overline{vw}}) & \text{if } Z = A_{\overline{vw}} \in \mathcal{A}, \\ \sigma^{G'}(X_{\overline{vw}}) & \text{if } Z = X_{\overline{vw}} \in \mathcal{X}. \end{cases}$$

   Based on the preceding, your task is to write a set $\Gamma^{G,G'}_{\text{SUBGRAPH}}$ of propositional WFF's over $\mathcal{A} \cup \mathcal{X}$ such that:

   G is a subgraph of G' iff $\sigma^{G,G'} \models \Gamma^{G,G'}_{\text{SUBGRAPH}}$.

   The set $\Gamma^{G,G'}_{\text{SUBGRAPH}}$ is finite or infinite depending on whether G is finite or infinite, respectively.

[4] $G = (V, E)$ is a *subgraph* of $G' = (V', E')$, and $G' = (V', E')$ is a *supergraph* of $G = (V, E)$, if $V = V'$ and $E \subseteq E'$; in words, G and G' have the same set of vertices, but G omits some (possibly none, possibly all) of the edges of G'.

2. Let $G = (V, E)$ be finite, with $\big|V\big| = n \geqslant 2$. Write a single propositional WFF $\varphi^{G,G'}_{\text{SUBGRAPH}}$ over $\mathcal{A} \cup \mathcal{X}$ which is equivalent to the set $\Gamma^{G,G'}_{\text{SUBGRAPH}}$ in Part 1. □

Exercise 8 Read all of the preceding exercises and accompanying comments in this section before tackling this exercise.

Let $G = (V, E)$ be a directed graph with $V = \{v_1, v_2, v_3, \ldots\}$, which may be finite or infinite. Define the following finite subsets of V:

$$V_1 := \{\, v_1 \,\},\ V_2 := \{\, v_1, v_2 \,\},\ V_3 := \{\, v_1, v_2, v_3 \,\},\ \ldots$$

It follows that $V_1 \subseteq V_2 \subseteq V_3 \subseteq \cdots \subseteq V$ and $\bigcup_{k \geqslant 1} V_k = V$. In this exercise, we consider two graphs G and G', and we use the set $\Gamma^{G,G'}_{\text{SUBGRAPH}}$ of WFF's defined in Exercise 7, according to which:

$$G \text{ is a subgraph of } G' \text{ iff } \sigma^{G,G'} \models \Gamma^{G,G'}_{\text{SUBGRAPH}}.$$

The set of propositional atoms associated with G and G' are $\mathcal{A}$ and $\mathcal{X}$, respectively. Corresponding to each V_k, define the sets $\mathcal{A}_k$ and $\mathcal{X}_k$ of propositional atoms by:

$$\mathcal{A}_k := \{\, A_{\overline{v\,w}} \mid v, w \in V_k \text{ and } v \neq w \,\},$$
$$\mathcal{X}_k := \{\, X_{\overline{v\,w}} \mid v, w \in V_k \text{ and } v \neq w \,\}.$$

$\mathcal{A}_1 = \mathcal{X}_1 = \varnothing$ since $\big|V_1\big| = 1$, and $\mathcal{A}_k = \mathcal{X}_k = k \cdot (k-1)$ for every $k \geqslant 2$, since $\big|V_k\big| = k$. Thus, every $\mathcal{A}_k$ and $\mathcal{X}_k$ is finite. We also have:

$$\mathcal{A}_1 \subseteq \mathcal{A}_2 \subseteq \mathcal{A}_3 \subseteq \cdots \subseteq \mathcal{A} \qquad \text{and} \qquad \mathcal{X}_1 \subseteq \mathcal{X}_2 \subseteq \mathcal{X}_3 \subseteq \cdots \subseteq \mathcal{X},$$

with $\bigcup_{k \geqslant 1} \mathcal{A}_k = \mathcal{A}$ and $\bigcup_{k \geqslant 1} \mathcal{X}_k = \mathcal{X}$.

1. For every $k \geqslant 2$, your task is to define a set $\Delta^{G'}_{\text{PATH},k}$ of propositional WFF's over $\mathcal{X}_k$ such that:

   there is $\theta \in \Delta^{G'}_{\text{PATH},k}$ such that $G' \models \theta$ iff G' contains a k-path over the vertices in V_k.

   Note the difference between the definition of $\Gamma^{G}_{\text{PATH},k}$ in Exercise 6 and $\Delta^{G'}_{\text{PATH},k}$ here, when V is an infinite set of vertices: the set $\Gamma^{G}_{\text{PATH},k}$ is infinite because $\mathcal{A}$ is infinite, while the set $\Delta^{G'}_{\text{PATH},k}$ is finite because $\mathcal{X}_k$ finite, even though V and $\mathcal{X}$ are infinite.

Since $\Delta^{G'}_{\text{PATH},k}$ is a finite set, we can define the single propositional WFF by the disjunction $\bigvee \Delta^{G'}_{\text{PATH},k}$. In Parts 2 and 3 below, we use the combined truth-value assignment $\sigma^{G,G'}$ defined in Exercise 7, and assume that V is an infinite set of vertices.

2. For every $k \geqslant 2$, define a set Θ_k of propositional WFF's which is satisfied by some $\sigma^{G,G'}$ iff:
   - G is a DAG, and
   - there is supergraph G' of G which is a DAG, and
   - there is a k-pseudopath in G, which is a k-path in G', over the vertices in $V_k = \{v_1, \ldots, v_k\}$.

   Prove that, if the given G is a DAG, then Θ_k is always satisfied by some $\sigma^{G,G'}$.
3. Define a set Θ of propositional WFF's which is satisfied by some $\sigma^{G,G'}$ iff:

– G is a DAG, and

– there is supergraph G' of G which is a DAG, and

– there is an infinite pseudopath in G, which is a path in G', over the full set V.

In Part 3, we do not ask you to show that: if the given G is a DAG, then Θ is always satisfiable. This is indeed the case, but more difficult to prove. Interestingly, it shows that a familiar fact about finite DAG's, namely, '*every finite DAG can be topologically sorted*', can be extended to infinite DAG's. □

2.2 Coloring of Undirected Graphs

We make the same simplifying assumptions, (i) and (ii), about undirected graphs as in Section 2.1. Given a graph $G = (V, E)$, every edge in E is a two-vertex set, *i.e.*:

$$E \subseteq \Big\{ \{v, w\} \;\Big|\; v, w \in V \text{ and } v \neq w \Big\}.$$

Again here, assumptions (i) and (ii) do not fundamentally affect the analysis below, but they make its presentation easier. We still denote an edge by writing '$\overline{v\ w}$', which now denotes the two-vertex set $\{v, w\}$, not the ordered pair $\langle v, w\rangle$.

In this section we consider questions of colorability of undirected graphs. For simplicity, suppose we are interested in 4-colorability: N (for *navy blue*), O (for *orange*), P (for *purple*), and R (for *red*). The first thing we need to settle for a formal modeling with propositional logic is the selection of propositional variables. For the four colors in question, it is natural to introduce the variables:

$$\mathcal{C} \;:=\; \{\, N_v \mid v \in V\} \;\cup\; \{\, O_v \mid v \in V\} \;\cup\; \{\, P_v \mid v \in V\} \;\cup\; \{\, R_v \mid v \in V\}$$

A variable in $\mathcal{C}$, say R_v, will be set to *true* or *false* depending on whether vertex v is colored *red* or not. As in Section 2.1, we also introduce a propositional atom $A_{\overline{vw}}$ for every two-vertex set $\{v, w\} \subseteq V$, which will be set to *true* or *false* depending on whether $\overline{v\ w}$ is or is not an edge of the given graph G. The set $\mathcal{A}$ of all propositional atoms is:

$$\mathcal{A} \;=\; \{\, A_{\overline{vw}} \mid v, w \in V \text{ and } v \neq w\}.$$

All our propositional WFF's in this section are written over $\mathcal{A} \cup \mathcal{C}$. Satisfaction of any such WFF is relative to a truth-value assignment $\sigma^{\mathcal{A},\mathcal{C}} : \mathcal{A} \cup \mathcal{C} \to \{\mathit{true}, \mathit{false}\}$.

Exercise 9 Let $G = (V, E)$ be an arbitrary undirected graph satisfying assumptions (i) and (ii). Write a set $\Gamma^G_{\text{4-COLORABLE}}$ of propositional WFF's such that $\sigma^{\mathcal{A},\mathcal{C}} \models \Gamma^G_{\text{4-COLORABLE}}$ iff G is 4-colorable.

Hint 1: For every vertex v, the set $\Gamma^G_{\text{4-COLORABLE}}$ must include a WFF specifying that v is assigned exactly one color in $\{N, O, P, R\}$.

Hint 2: For every two distinct vertices v and w, the set $\Gamma^G_{\text{4-COLORABLE}}$ must include a WFF specifying that, if there is an edge connecting v and w, then v and w are not assigned the same color. □

Exercise 10 In this exercise we assume that $G = (V, E)$ is a planar graph. The set $\Gamma^G_{\text{4-COLORABLE}}$ defined in Exercise 9 is finite or infinite, depending on whether V is a finite or infinite set.

1. Let V be finite. Show there is a truth-value assignment $\sigma^{\mathcal{A},\mathcal{C}}$ satisfying $\Gamma^G_{\text{4-COLORABLE}}$.

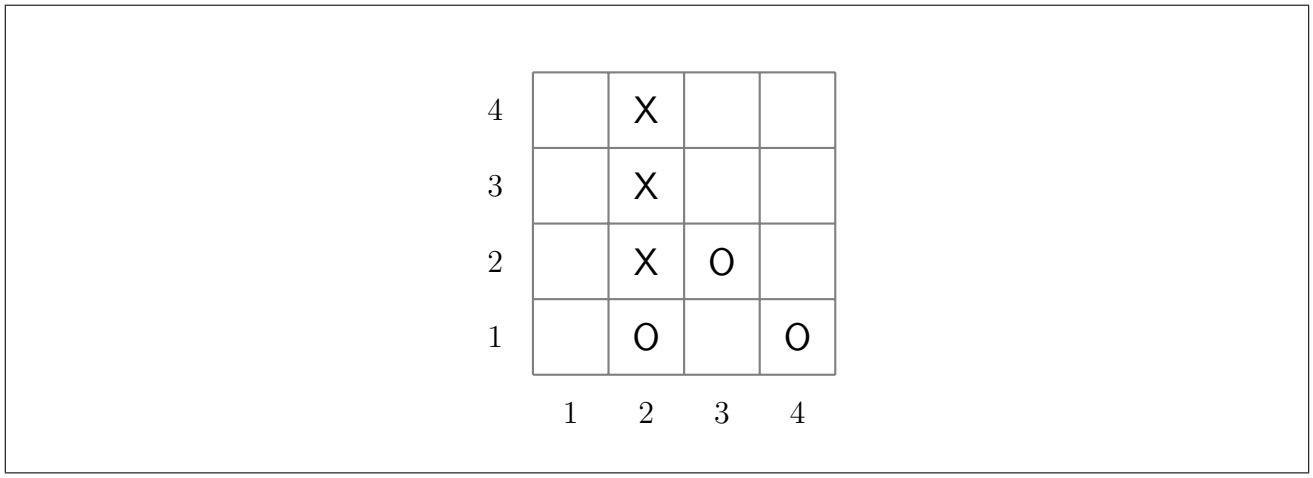

Figure 1: Tic-Tac-Toe on a 4×4 board after 6 moves, with 3 for each of the two players.

2. Let V be infinite. Show again there is a truth-value assignment $\sigma^{\mathcal{A},\mathcal{C}}$ satisfying $\Gamma^{G}_{\text{4-COLORABLE}}$.

Thus, whether V is finite or infinite, the planar graph G is always 4-colorable.

Hint 1: For Part 1, you can invoke well-known facts about finite planar graphs – which are difficult to prove, though their invocation is easy.

Hint 2: Part 2 is not easy. You may get some inspiration by doing Exercise 8. □

3 A Two-Player Game: Tic-Tac-Toe

There are different ways, in different formal logics, of modeling Tic-Tac-Toe. If we only want to model the *starting configuration* and the *winning configuration* in the game, then propositional logic (PL) will suffice.

To make the game a little more interesting, consider Tic-Tac-Toe on a $K \times K$ board where $K \geqslant 3$. The game for $K = 3$ is the usual version. The game for $K = 4$ is shown in Figure 1, with a possible configuration of the board after 6 moves.

The first thing we need to do is to choose the propositional atoms for our modeling. For convenience, we write $[K]$ to denote the set of indeces $\{1, 2, \ldots, K\}$. Here is a plausible choice:

- Use two-indexed propositional atoms, $P_{i,j}$ and $Q_{i,j}$ with $i, j \in [K]$, to identify the squares where X and O are located on the board. Specifically,

$$P_{i,j} \;=\; \begin{cases} \mathit{true} & \text{if square } \langle i,j\rangle \in [K]\times[K] \text{ contains X},\\ \mathit{false} & \text{if square } \langle i,j\rangle \in [K]\times[K] \text{ does not contain X}, \end{cases}$$

$$Q_{i,j} \;=\; \begin{cases} \mathit{true} & \text{if square } \langle i,j\rangle \in [K]\times[K] \text{ contains O},\\ \mathit{false} & \text{if square } \langle i,j\rangle \in [K]\times[K] \text{ does not contain O}. \end{cases}$$

The *starting configuration* is the configuration when no X and no O are yet placed on the board, which

can be modeled by:

$$\varphi_{\mathsf{start}} := \Big(\bigwedge_{\langle i,j\rangle\in[K]\times[K]} \neg P_{i,j}\Big) \wedge \Big(\bigwedge_{\langle i,j\rangle\in[K]\times[K]} \neg Q_{i,j}\Big)$$

It should be clear that φ_{start} is satisfied, *i.e.*, made *true*, by the valuation that makes every $P_{i,j}$ and every $Q_{i,j}$ *false*. Thus, φ_{start} is a formal way of asserting in plain English:

"for every position $\langle i, j\rangle$, X is not on $\langle i, j\rangle$, and for every position $\langle i, j\rangle$, O is not on $\langle i, j\rangle$."

We next model a *winning configuration* for X. But what is a winning configuration for X when $K \geqslant 4$? There are four possible ways in which the X-player can win:

- K occurrences of X are placed in the same row of the board,
- K occurrences of X are placed in the same column of the board,
- K occurrences of X are placed along the first diagonal of the board,
- K occurrences of X are placed along the second diagonal of the board.

Note the indeces of the first diagonal are $\{\langle 1, 1\rangle, \langle 2, 2\rangle, \ldots, \langle K, K\rangle\}$, while those of the second diagonal are $\{\langle 1, K\rangle, \langle 2, K-1\rangle, \ldots, \langle K, 1\rangle\}$. Although there are several ways of formally modeling a winning configuration for the X-player, a particular simple one is the following formula:

$$\varphi_{\mathsf{x\text{-}win}} := \underbrace{\Big(\bigvee_{i\in[K]}\bigwedge_{j\in[K]} P_{i,j}\Big)}_{\text{all X's in a row}} \vee \underbrace{\Big(\bigvee_{j\in[K]}\bigwedge_{i\in[K]} P_{i,j}\Big)}_{\text{all X's in a column}} \vee \underbrace{\Big(\bigwedge_{i\in[K]} P_{i,i}\Big)}_{\text{all X's in first diagonal}} \vee \underbrace{\Big(\bigwedge_{i\in[K]} P_{i,K+1-i}\Big)}_{\text{all X's in second diagonal}}$$

We can define $\varphi_{\mathsf{o\text{-}win}}$ in a similar way, or in many other ways too.

Exercise 11 The O-player in Tic-Tac-Toe wins by preventing the X-player from reaching a winning configuration. Define a propositional WFF $\varphi_{\mathsf{o\text{-}win}}$ which formally models a winning configuration for the O-player by denying a winning configuration for the X-player. The desired WFF $\varphi_{\mathsf{o\text{-}win}}$ must simultaneously model a winning configuration for the O-player *and* a non-winning configuration for the X-player.

Hint 1: A final configuration, when every position contains X or O, may have as sub-configurations both a winning configuration for the X-player and a winning configuration for the O-player.

Hint 2: A winning configuration for the O-player is not 'symmetric' to a winning configuration for the X-player, *i.e.*, the former cannot be obtained from the latter by replacing every $P_{i,j}$ by $Q_{i,j}$ in $\varphi_{\mathsf{x\text{-}win}}$. □

Exercise 12 Generalize the notion of *winning configuration* for the X-player (and similarly for the O-player) as follows:

- X-player wins iff K occurrences of X are placed in K contiguous squares on the board.

Call the resulting game Tic-Tac-Toe*.

Two squares (or positions) of the board are *contiguous* iff they have a side in common. For example, squares $\langle 2,3\rangle$ and $\langle 2,4\rangle$ are contiguous, but squares $\langle 2,4\rangle$ and $\langle 3,3\rangle$ are not contiguous. In this version of the game, K occurrences of X in the same row (or in the same column) is a winning configuration, just as it is in the usual Tic-Tac-Toe, but in contrast to the usual game, K occurrences of X along the first or the second diagonal is not a winning configuration.

Write a propositional WFF $\psi_{\mathsf{x\text{-}win}}$ which is satisfied iff $\psi_{\mathsf{x\text{-}win}}$ represents a winning configuration for the X-player in Tic-Tac-Toe*. □

4 A One-Player Game: Yashi

An instance of the Yashi game is specified by a $n \times n$ integer grid for some $n \geqslant 2$, on which $p \geqslant 2$ nodes are placed. A *solution* of the game consists in drawing horizontal and vertical segments, satisfying the following conditions:

($\diamondsuit$1) Every segment connects two and only two nodes.

($\diamondsuit$2) No two segments overlap.

($\diamondsuit$3) No two segments cross each other.

($\diamondsuit$4) The segments form a tree, *i.e.*, they form a graph without cycles. Put differently still, for every two nodes a and b there is exactly one path between a and b.

(We do not need the two first conditions, because ($\diamondsuit$2) implies ($\diamondsuit$1) and ($\diamondsuit$3) implies ($\diamondsuit$2). We include them here for clarity. Conditions ($\diamondsuit$3) and ($\diamondsuit$4) suffice for the definition of a solution.)

You can find out more about this game from the website Yashi. Given an instance $\mathcal{G}$ of Yashi, we can consider different versions of the game:

Version 1. Decide if there is a solution for $\mathcal{G}$. If there is, return one solution.

Version 2. Decide if there is a solution for $\mathcal{G}$. If there is, return the number of solutions.

Version 3. Decide if there is a solution for $\mathcal{G}$. If there is, return a minimum-length solution.

The usual Yashi game corresponds to **Version 1** above. In these three versions, if there is *no* solution, there is nothing else to do. More elaborate versions require that something be done, if there is *no* solution. For example, **Version 4** below is an extension of **Version 3** that requires further action:

Version 4. Decide if there is a solution for $\mathcal{G}$.

1. If there is, return a minimum-length solution.
2. If there is not, determine the least number k of new nodes to be placed on the grid in order to induce a solution.
3. If there is not, and k is the least number of new nodes to be placed on the grid in order to induce a solution, place k new nodes so that a minimum-length solution is induced.

We illustrate the preceding definitions and the different versions with two small examples – small enough that we can inspect the existence and non-existence of solutions by inspection (not relying on

programming or automation of any kind).

We identify a node on a $n \times n$ grid by its horizontal-vertical coordinates $\langle i, j\rangle$ with $0 \leqslant i, j \leqslant n$. A segment connecting node $\langle i, j\rangle$ and node $\langle k, \ell\rangle$ is represented by a two-element pair $\big(\langle i, j\rangle, \langle k, \ell\rangle\big)$.

Example 13 The top row in Figure 2 shows a 6×6 grid with 6 nodes. This is an instance of the Yashi game; call it $\overline{\mathcal{G}}_1$ for later reference. There are 4 possible non-overlapping segments $\{X_1, X_2, X_3, X_4\}$:

$$X_1 = \big(\langle 1,1\rangle, \langle 1,5\rangle\big), \quad X_2 = \big(\langle 2,2\rangle, \langle 2,4\rangle\big), \quad X_3 = \big(\langle 5,3\rangle, \langle 5,5\rangle\big), \quad X_4 = \big(\langle 1,5\rangle, \langle 5,5\rangle\big)$$

A solution is a subset of $\{X_1, X_2, X_3, X_4\}$ satisfying conditions $\{(\diamondsuit 3), (\diamondsuit 4)\}$. By inspection, it is easy to see there is no solution for this instance of Yashi. For **Version 1**, **Version 2**, and **Version 3**, the answer is "no" and there is nothing else to do.

For **Version 4**, it is easy to see that placing one new node on the grid induces several solutions. Three of these are shown in the middle row in Figure 2. A possible answer for **Version 4** of the game is to return the leftmost grid in the middle row in Figure 2.

If we are to place two new nodes on the grid, we can induce a solution of shorter length, as shown in the bottom row in Figure 2, but this corresponds to a different way of playing Yashi, according to **Version 5** defined next. □

Version 5. Decide if there is a solution for $\mathcal{G}$.

1. If there is, return a minimum-length solution.
2. If there is not, determine the least number k of new nodes to be placed on the grid in order to induce a solution.
3. If there is not, and k is the least number of new nodes to be placed on the grid to induce a solution, place $k + \ell$ new nodes so that a minimum-length solution is induced, where $\ell \geqslant 0$.

Parts 1 and 2 are the same as in **Version 4**; in part 3, we are allowed to place $k + \ell$ new nodes, where $\ell \geqslant 0$, instead of just k as in **Version 4**.

Example 14 Figure 3 shows a 12×12 grid with 12 nodes. Call this is an instance of the Yashi game $\overline{\mathcal{G}}_2$ for later reference. There are 14 possible non-overlapping segments $\{X_1, \ldots, X_{14}\}$:

$$X_1 = \big(\langle 2,1\rangle, \langle 6,1\rangle\big), \quad X_2 = \big(\langle 6,1\rangle, \langle 8,1\rangle\big), \quad X_3 = \big(\langle 6,5\rangle, \langle 9,5\rangle\big), \quad X_4 = \big(\langle 0,7\rangle, \langle 2,7\rangle\big),$$
$$X_5 = \big(\langle 2,7\rangle, \langle 4,7\rangle\big), \quad X_6 = \big(\langle 4,7\rangle, \langle 6,7\rangle\big), \quad X_7 = \big(\langle 6,7\rangle, \langle 9,7\rangle\big), \quad X_8 = \big(\langle 6,9\rangle, \langle 8,9\rangle\big),$$
$$X_9 = \big(\langle 2,1\rangle, \langle 2,7\rangle\big), \quad X_{10} = \big(\langle 6,1\rangle, \langle 6,5\rangle\big), \quad X_{11} = \big(\langle 6,5\rangle, \langle 6,7\rangle\big), \quad X_{12} = \big(\langle 6,7\rangle, \langle 6,9\rangle\big),$$
$$X_{13} = \big(\langle 8,1\rangle, \langle 8,9\rangle\big), \quad X_{14} = \big(\langle 9,5\rangle, \langle 9,7\rangle\big).$$

This instance of Yashi has many solutions; two are shown in Figure 3.

By inspection, it is easy to see that the segments in $\{X_1, \ldots, X_{14}\}$ form 5 simple cycles, *i.e.*, cycles without repeated nodes. We can represent these cycles by the following sequences (starting with the

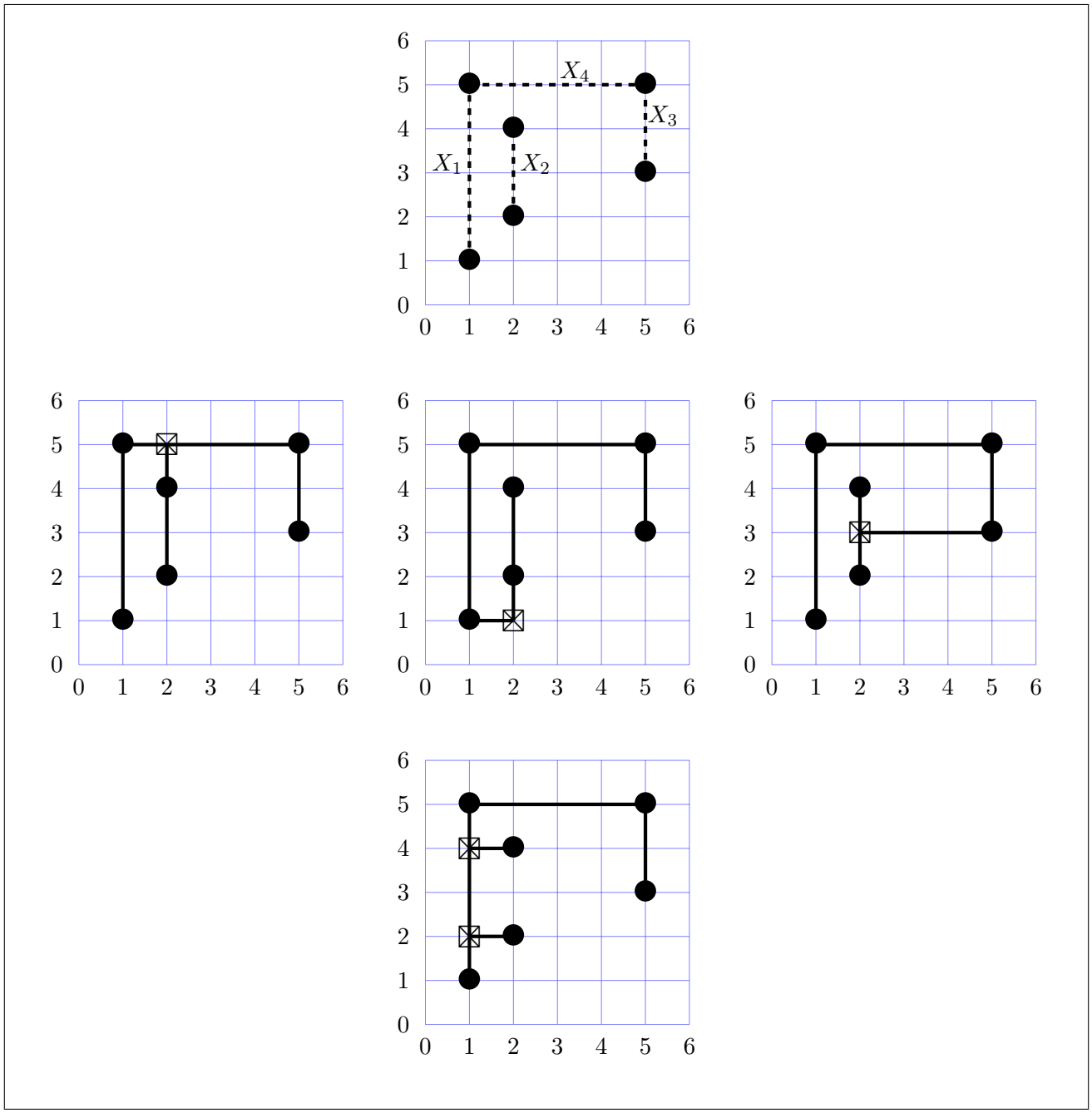

Figure 2: An instance of the Yashi game, a 6×6 grid with 6 nodes. The initial grid is in the top row, showing all possible horizontal and vertical segments, and has no solution. In the middle row, there are three grids, showing that placing one new node (denoted ⊠) induces a solution, of different length in each case (from left to right: 13, 14, and 15, respectively). In the bottom row, the grid shows that if we place two new nodes, we can induce a solution of shorter length 12.

leftmost horizontal segment in each case and traversing the cycles in a clockwise direction):

$X_5\ X_6\ X_{11}\ X_{10}\ X_1\ X_9$

$X_5\ X_6\ X_7\ X_{14}\ X_3\ X_{10}\ X_1\ X_9$

$X_5\ X_6\ X_{12}\ X_8\ X_{13}\ X_2\ X_1\ X_9$

$X_8\ X_{13}\ X_2\ X_{10}\ X_{11}\ X_{12}$

$X_7\ X_{14}\ X_3\ X_{11}$

A solution is a subset of $\{X_1, \ldots, X_{14}\}$ which excludes these 5 cycles as well as all the crossings. In this example, there are two crossings, represented by the following pairs: $\{X_3, X_{13}\}$ and $\{X_7, X_{13}\}$. □

4.1 How to Automate the Yashi Game?

One approach to solving the Yashi game is to write a program that will exhaustively search for a solution. Let's call it the ***programming approach***. Specifically, given an instance $\mathcal{G}$ of Yashi (assumed to contain no crossings for a simpler preliminary case), proceed as follows:

1. Use an efficient algorithm to decide whether $\mathcal{G}$ is *connected* as a graph. If it is, proceed to the next step. If it is not, stop and report there is no solution.
2. Use an efficient algorithm to compute a *minimum spanning tree* of $\mathcal{G}$ as a plane graph. The minimum spanning tree thus computed is a solution for $\mathcal{G}$.

If $\mathcal{G}$ contains crossings, the situation is a bit more delicate. It requires a pre-processing step in which some of the segments involved in crossings have to be removed, before continuing with step 1 and step 2 above. This pre-processing is related to the problem of deleting the smallest number of edges in a graph in order to turn it into a plane graph; this has been studied by others in the research community.

As outlined above, the programming approach handles **Version 1** and **Version 3** of the Yashi game. Further complications arise, however, if we want to extend the programming approach to **Version 2**, **Version 4**, or **Version 5**. Such extensions are possible, though they require hard work to analyze, design, and implement without errors. We do not delve into these complications and turn instead our attention to an alternative approach, which we call the ***constraint-solving approach***.

In the constraint-solving approach, we first write *constraints* that model the conditions for a Yashi solution, and then let a *constraint solver* do all the computation for us – thus avoiding the hard and error-prone work of programming. The constraints we want to write must reflect the following:

Fact 15 *Let $\mathcal{G}$ be an instance of the Yashi game with $p \geqslant 2$ nodes, and let $\mathcal{X}$ be the set of all non-overlapping horizontal and vertical segments connecting two nodes of $\mathcal{G}$. Then $\mathcal{G}$ has a solution iff:*

there is a subset $\mathcal{Y} \subseteq \mathcal{X}$ of exactly $(p-1)$ segments satisfying two conditions:

1. *no two segments in $\mathcal{Y}$ cross each other, and*
2. *no subset of $\mathcal{Y}$ form a cycle.*

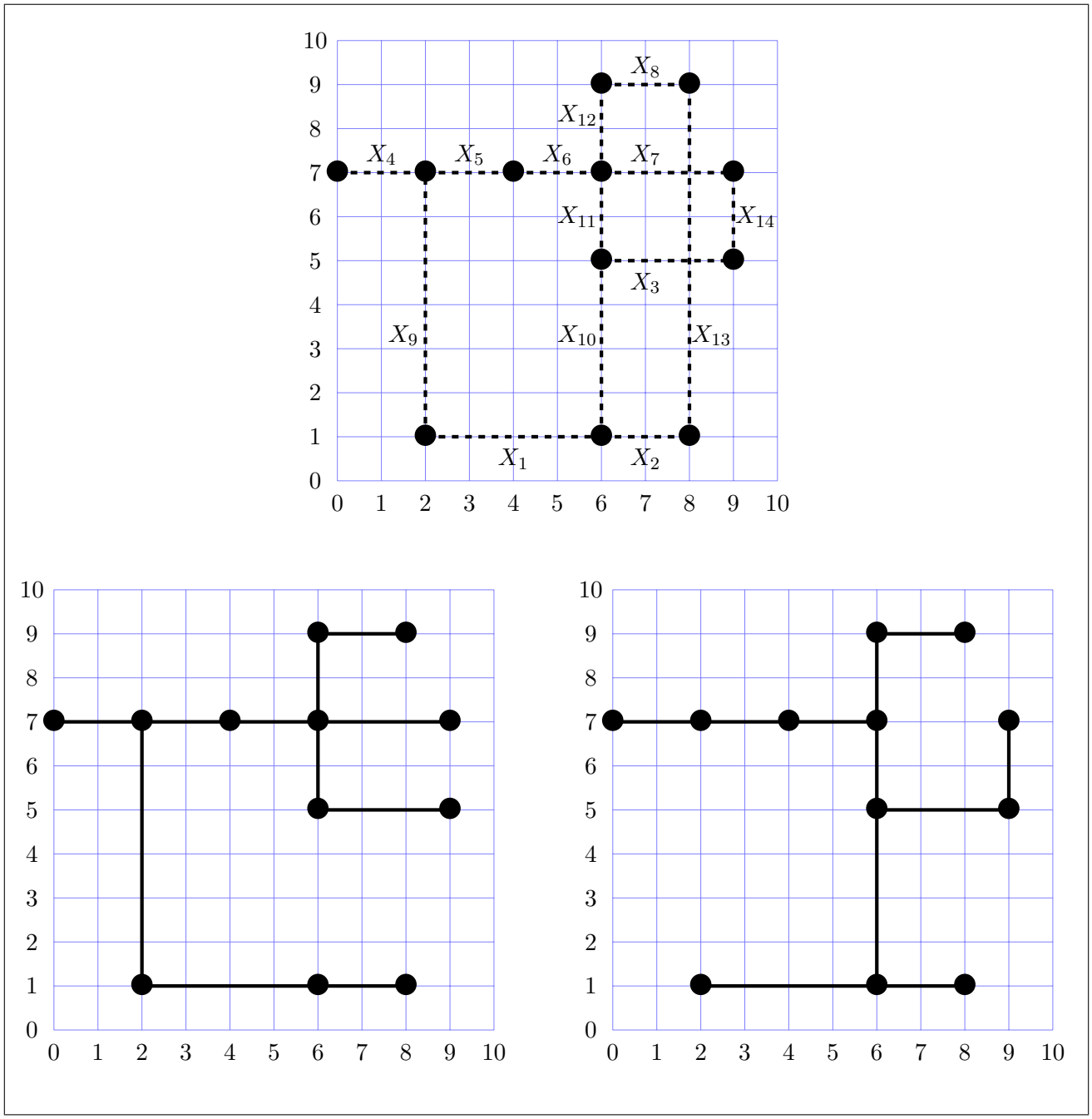

Figure 3: An instance of the Yashi game, a 10×10 grid with 12 nodes. Top grid shows all possible horizontal and vertical segments. Bottom left grid shows a solution of length 30. Bottom right grid shows a solution of length 27.

Fact 15 characterizes what makes a susbet $\mathcal{Y}$ of $\mathcal{X}$ a solution for the Yashi instance $\mathcal{G}$, but this requires a little proof because the definition of a Yashi solution in the opening paragraph of Section 4 says nothing about the number of segments in $\mathcal{Y}$ (here specified to be $p-1$).

Exercise 16 Show that if $\mathcal{Y}$ is a solution for the Yashi instance $\mathcal{G}$, then there are $(p-1)$ segments in $\mathcal{Y}$. Conversely, show that if $\mathcal{Y}$ contains $(p-1)$ segments satisfying conditions 1 and 2 in the statement of Fact 15, then $\mathcal{Y}$ is a solution for $\mathcal{G}$.

Hint: From your course on *graph theory* or on *design and analysis of algorithms*, the spanning tree of a simple graph with p nodes is a subset of the edges of G of size $(p-1)$. □

4.2 Using SAT Solvers

One possible formulation of the Yashi constraints is to use the segment names $\mathcal{X} = \{X_1, X_2, \ldots\}$ as Boolean variables and then write propositional formulas over $\mathcal{X}$ that enforce the selection of a subset $\mathcal{Y} \subseteq \mathcal{X}$ of size $(p-1)$ satisfying conditions 1 and 2 in the statement of Fact 15. The idea is that if Boolean variable X_i is assigned truth-value *true* (resp. *false*), then the corresponding line segment X_i is included (resp. excluded) from the solution.

Moreover, since we want to use a SAT solver to solve the constraints, it is better if these propositional formulas are in CNF, though this is not essential.

Following this proposal we need to write three propositional wff's in CNF:

- φ_{tree}, which is satisfied by a truth-value assignment iff exactly $(p-1)$ of the Boolean variables in $\mathcal{X}$ are assigned *true*.
- $\varphi_{\text{no-crossings}}$, which is satisfied by a truth-value assignment iff no two Boolean variables in $\mathcal{X}$ representing a crossing are simultaneously assigned *true*.
- $\varphi_{\text{no-cycles}}$, which is satisfied by a truth-value assignment iff no subset of Boolean variables in $\mathcal{X}$ representing a cycle are simultaneously assigned *true*.

Using a SAT solver, we can then decide whether the propositional wff $\Phi \triangleq \varphi_{\text{tree}} \wedge \varphi_{\text{no-crossings}} \wedge \varphi_{\text{no-cycles}}$ is satisfiable. The existence of a model (or Boolean interpretation) for Φ solves **Version 1** of the game, and counting the number of models for Φ solves **Version 2** of the game.

For **Version 3**, **Version 4**, and **Version 5**, which involve *optimization*, we need to do more to model the problem and then use an SMT solver (not just a SAT solver).

For "small" instances $\mathcal{G}$ of Yashi, the approach outlined so far is feasible and manageable, but for "large" instances it may not be. Specifically, the wff Φ may be too large to write by hand or for a SAT solver to process in a timely fashion.

Exercise 17 Let $\mathcal{G}$ be an instance of Yashi as in the statement of Fact 15 and consider the wff φ_{tree} whose satisfaction corresponds to the selection of exactly $(p-1)$ segments from the set $\mathcal{X}$ of all segments. Suppose $\left|\mathcal{X}\right| = q$ with $q > p$.

1. Show that if φ_{tree} is written in DNF, *i.e.*, as a disjunction of conjuncts of literals, then φ_{tree} contains $\binom{q}{p-1}$ conjuncts.
2. Show that if φ_{tree} is written in CNF, *i.e.*, as a conjunction of disjuncts of literals (or clauses), then φ_{tree} contains $2^q - \binom{q}{p-1}$ clauses.

These numbers quickly become unmanageable. For example, if $q = 14$ and $p = 12$ (these are the numbers in Example 14), then $\binom{q}{p-1} = \binom{14}{11} = 364$ and $2^q - \binom{q}{p-1} = 2^{14} - \binom{14}{11} = 16384 - 364 = 16020$. Certainly, φ_{tree}, whether in DNF or CNF, is not to be written by hand! □

What is the alternative, if we do not want to encode φ_{tree} as a propositional wff? Luckily, if a subset $\mathcal{Y} \subseteq \mathcal{X}$ satisfies the constraints *no-crossings* and *no-cycles*, we can enforce the requirement that $\mathcal{Y}$ is a spanning tree of $\mathcal{G}$ differently. This alternative consists in writing wff's that prevent *cuts* in $\mathcal{Y}$. [5]

Example 18 Consider the Yashi instance $\overline{\mathcal{G}}_2$ in Example 14 again. We do not try to write φ_{tree}. Instead, we formulate constraints that prevent cuts in any proposed solution $\mathcal{Y}$, by examining $\overline{\mathcal{G}}_2$ more closely. These constraints specify enough of the structure of $\overline{\mathcal{G}}_2$ so that a SAT solver can return an answer quickly. But first, we write the wff's $\varphi_{\text{no-crossings}}$ and $\varphi_{\text{no-cycles}}$, both in CNF, which are fairly simple:

$$\varphi_{\text{no-crossings}} \triangleq (\neg X_3 \vee \neg X_{13}) \wedge (\neg X_7 \vee \neg X_{13})$$

$$\begin{aligned}\varphi_{\text{no-cycles}} \triangleq\ & (\neg X_1 \vee \neg X_5 \vee \neg X_6 \vee \neg X_9 \vee \neg X_{10} \vee \neg X_{11}) \\ & \wedge\ (\neg X_1 \vee \neg X_3 \vee \neg X_5 \vee \neg X_6 \vee \neg X_7 \vee \neg X_9 \vee \neg X_{10} \vee \neg X_{14}) \\ & \wedge\ (\neg X_1 \vee \neg X_2 \vee \neg X_5 \vee \neg X_6 \vee \neg X_8 \vee \neg X_9 \vee \neg X_{12} \vee \neg X_{13}) \\ & \wedge\ (\neg X_2 \vee \neg X_8 \vee \neg X_{10} \vee \neg X_{11} \vee \neg X_{12} \vee \neg X_{13}) \\ & \wedge\ (\neg X_3 \vee \neg X_7 \vee \neg X_{11} \vee \neg X_{14})\end{aligned}$$

For each node $\langle i, j\rangle$ we write a wff $\text{NO-CUT}_{\langle i,j\rangle}$ which must be satisfied in order that node $\langle i, j\rangle$ not be cut from every proposed spanning tree (*i.e.*, solution) for $\overline{\mathcal{G}}_2$:

$$\begin{aligned}\text{NO-CUT}_{\langle 2,1\rangle} &\triangleq X_1 \vee X_9 \\ \text{NO-CUT}_{\langle 6,1\rangle} &\triangleq X_1 \vee X_2 \vee X_{10} \\ \text{NO-CUT}_{\langle 8,1\rangle} &\triangleq X_2 \vee X_{13} \\ \text{NO-CUT}_{\langle 6,5\rangle} &\triangleq X_3 \vee X_{10} \vee X_{11} \\ \text{NO-CUT}_{\langle 9,5\rangle} &\triangleq X_3 \vee X_{14} \\ \text{NO-CUT}_{\langle 0,7\rangle} &\triangleq X_4 \\ \text{NO-CUT}_{\langle 2,7\rangle} &\triangleq X_4 \vee X_5 \vee X_9 \\ \text{NO-CUT}_{\langle 4,7\rangle} &\triangleq X_5 \vee X_6 \\ \text{NO-CUT}_{\langle 6,7\rangle} &\triangleq X_6 \vee X_7 \vee X_{11} \vee X_{12} \\ \text{NO-CUT}_{\langle 9,7\rangle} &\triangleq X_7 \vee X_{14} \\ \text{NO-CUT}_{\langle 6,9\rangle} &\triangleq X_8 \vee X_{12} \\ \text{NO-CUT}_{\langle 8,9\rangle} &\triangleq X_8 \vee X_{13}\end{aligned}$$

[5] Recall that a *cut* in a graph with set N of nodes is denoted by a pair $(A, N - A)$ where $A \subsetneq N$. This is the cut that removes all the edges connecting nodes in A with nodes in $N - A$. We call $(A, N - A)$ the *cut specified by* A.

Let N be the set of all nodes in $\overline{\mathcal{G}}_2$. For every cut $(A, N-A)$, we can write a wff NO-CUT$_A$ which must be satisfied so that, in every proposed spanning tree, the nodes in A are reachable from the nodes in $(N-A)$.The following are examples of wff's preventing two-node cuts:

$$\begin{aligned}
\text{NO-CUT}_{\{\langle 2,1\rangle,\langle 6,1\rangle\}} &\triangleq X_2 \vee X_9 \vee X_{10} \\
\text{NO-CUT}_{\{\langle 6,1\rangle,\langle 8,1\rangle\}} &\triangleq X_1 \vee X_{10} \vee X_{13} \\
\text{NO-CUT}_{\{\langle 0,7\rangle,\langle 2,7\rangle\}} &\triangleq X_5 \vee X_9 \\
\text{NO-CUT}_{\{\langle 9,5\rangle,\langle 9,7\rangle\}} &\triangleq X_3 \vee X_7 \\
\text{NO-CUT}_{\{\langle 6,9\rangle,\langle 8,9\rangle\}} &\triangleq X_{12} \vee X_{13}
\end{aligned}$$

Some of the wff's preventing three-node cuts:

$$\begin{aligned}
\text{NO-CUT}_{\{\langle 0,7\rangle,\langle 2,1\rangle,\langle 2,7\rangle\}} &\triangleq X_1 \vee X_5 \\
\text{NO-CUT}_{\{\langle 0,7\rangle,\langle 2,7\rangle,\langle 4,7\rangle\}} &\triangleq X_6 \vee X_9 \\
\text{NO-CUT}_{\{\langle 2,1\rangle,\langle 6,1\rangle,\langle 8,1\rangle\}} &\triangleq X_9 \vee X_{10} \vee X_{13} \\
\text{NO-CUT}_{\{\langle 6,1\rangle,\langle 6,5\rangle,\langle 8,1\rangle\}} &\triangleq X_1 \vee X_3 \vee X_{11} \vee X_{13}
\end{aligned}$$

Some of the wff's preventing four-node cuts:

$$\begin{aligned}
\text{NO-CUT}_{\{\langle 0,7\rangle,\langle 2,1\rangle,\langle 2,7\rangle,\langle 4,7\rangle\}} &\triangleq X_1 \vee X_6 \\
\text{NO-CUT}_{\{\langle 2,1\rangle,\langle 6,1\rangle,\langle 8,1\rangle,\langle 6,5\rangle\}} &\triangleq X_3 \vee X_9 \vee X_{11} \vee X_{13}
\end{aligned}$$

Some of wff's preventing five-node and six-node cuts:

$$\begin{aligned}
\text{NO-CUT}_{\{\langle 2,1\rangle,\langle 6,1\rangle,\langle 8,1\rangle,\langle 0,7\rangle,\langle 2,7\rangle\}} &\triangleq X_5 \vee X_{10} \vee X_{13} \\
\text{NO-CUT}_{\{\langle 6,1\rangle,\langle 8,1\rangle,\langle 6,5\rangle,\langle 9,5\rangle,\langle 9,7\rangle\}} &\triangleq X_1 \vee X_7 \vee X_{11} \vee X_{13} \\
\text{NO-CUT}_{\{\langle 6,7\rangle,\langle 6,9\rangle,\langle 8,9\rangle,\langle 9,5\rangle,\langle 9,7\rangle\}} &\triangleq X_3 \vee X_6 \vee X_{11} \vee X_{13} \\
\text{NO-CUT}_{\{\langle 2,1\rangle,\langle 6,1\rangle,\langle 8,1\rangle,\langle 6,5\rangle,\langle 9,5\rangle\}} &\triangleq X_9 \vee X_{11} \vee X_{13} \vee X_{14} \\
\text{NO-CUT}_{\{\langle 2,1\rangle,\langle 6,1\rangle,\langle 8,1\rangle,\langle 0,7\rangle,\langle 2,7\rangle,\langle 4,7\rangle\}} &\triangleq X_6 \vee X_{10} \vee X_{13} \\
\text{NO-CUT}_{\{\langle 2,1\rangle,\langle 6,1\rangle,\langle 8,1\rangle,\langle 6,5\rangle,\langle 9,5\rangle,\langle 9,7\rangle\}} &\triangleq X_7 \vee X_9 \vee X_{11} \vee X_{13}
\end{aligned}$$

We can write other constraints of the form NO-CUT$_A$, but the preceding already specify much of the structural properties of $\mathcal{G}$ and suffice for a quick execution by a SAT solver that returns a correct answer. [6]

The script that encodes the preceding wff's in the syntax of Z3 is shown in Figure 4. You can copy this script into a file named `yashi1`, say, and then test it by executing the command '`z3 yashi1`', assuming you have installed Z3 on your computer. Or you can load the script into an on-line editor and test it through your Web browser, by clicking here or here. □

[6] We purposely avoid writing a constraint NO-CUT$_A$ for every $\varnothing \subsetneq A \subsetneq N$. If we did, we would have to write as many as $(2^{12} - 2) = 4094$ of them, since there are $p = 12$ nodes in this example. We have written enough of them to get a correct answer which, true enough, we can check by hand in this example.

4.3 Using SMT Solvers

If we want to play versions of Yashi other than **Version 1**, we cannot limit ourselves to SAT solving, because the other versions involve some arithmetic. We need to use a SMT solver. We illustrate this with the Yashi instance already considered in Examples 14 and 18.

Example 19 We use the same Yashi instance $\overline{\mathcal{G}}_2$ as in Example 18. For **Version 3** we need to formulate a new constraint which, together with the constraints already formulated in Example 18, require that a solution be of minimum-length.

Let Φ be the conjunction of all the wff's defined in Example 18. These are written over variables $\mathcal{X} = \{X_1, \ldots, X_{14}\}$ of type `Boolean`. Call θ the new constraint that we need to formulate. We choose to write θ over $\mathcal{X}$ and two new variables of type `Integer`, *length* and *temp*:

$$
\begin{array}{lll}
\theta \triangleq \big(\mathsf{eq}\ \mathit{length} & & \\
\quad \big(\textbf{let} & \mathit{temp} = & \\
& (\textbf{if } X_1 \textbf{ then } 4 \textbf{ else } 0) + (\textbf{if } X_2 \textbf{ then } 2 \textbf{ else } 0) + (\textbf{if } X_3 \textbf{ then } 3 \textbf{ else } 0) + & \\
& (\textbf{if } X_4 \textbf{ then } 2 \textbf{ else } 0) + (\textbf{if } X_5 \textbf{ then } 2 \textbf{ else } 0) + (\textbf{if } X_6 \textbf{ then } 2 \textbf{ else } 0) + & \\
& (\textbf{if } X_7 \textbf{ then } 3 \textbf{ else } 0) + (\textbf{if } X_8 \textbf{ then } 2 \textbf{ else } 0) + (\textbf{if } X_9 \textbf{ then } 6 \textbf{ else } 0) + & \\
& (\textbf{if } X_{10} \textbf{ then } 4 \textbf{ else } 0) + (\textbf{if } X_{11} \textbf{ then } 2 \textbf{ else } 0) + (\textbf{if } X_{12} \textbf{ then } 2 \textbf{ else } 0) + & \\
& (\textbf{if } X_{13} \textbf{ then } 8 \textbf{ else } 0) + (\textbf{if } X_{14} \textbf{ then } 2 \textbf{ else } 0) & \\
\quad \textbf{in} & (\mathsf{minimize}\ \mathit{temp})\ \big)\ \big) &
\end{array}
$$

We use two arithmetical primitive operations, 'eq' and '$\mathsf{minimize}$' in the following sub-wff's of θ:

- '$\big(\mathsf{eq}\ \mathit{length}\ (\mathsf{minimize}\ \mathit{temp})\big)$' which returns *true* if the integer values assigned to *length* and ($\mathsf{minimize}$ *temp*) are equal, and *false* otherwise.
- '($\mathsf{minimize}$ *temp*)' which minimizes the value of *temp*; specifically, it chooses an assignment of truth-values to the Boolean variables in $\mathcal{X}$ that minimizes the integer assigned to *temp*.

Next, we invoke a SMT solver with arithmetic capabilities to solve the combined constraint $\Phi \wedge \theta$.

In Figure 5 we show how to encode θ in the syntax of Z3. To test it, you can create a new file named `yashi2`, which extends the script (from Figure 4) encoding Φ by inserting the script (from Figure 5) encoding θ, right after '`(assert yashi-solution)`' and before '`(check-sat)`'. □

With the arithmetic already used in Example 19 to play **Version 3** of Yashi, we can in fact considerably simplify the wff Φ. This is illustrated in the next example.

Example 20 Once more we consider the Yashi instance $\overline{\mathcal{G}}_2$. For **Version 3** of Yashi we show that, with a little arithmetic, we can replace the constraint Φ by a simpler constraint Φ':

$$\Phi' \triangleq \varphi_{\text{no-crossings}} \wedge \varphi_{\text{no-cycles}}$$

where $\varphi_{\text{no-crossings}}$ and $\varphi_{\text{no-cycles}}$ are defined in Example 18. The wff Φ', if satisfied, enforces the two requirements: there are 'no crossings' and there are 'no cycles'. It omits from Φ all sub-wff's of the

form NO-CUT$_A$. According to Fact 15, to get a solution, it suffices to define a new wff, call it θ', which enforces the additional requirement that '11 segments are selected'.

We can write θ' in the way θ is written in Example 19: θ' counts the number of segments selected, instead of the sum of their lengths, and is satisfied if that number is 11:

$$
\begin{array}{l}
\theta' \triangleq \Big(\mathsf{eq}\ \ 11 \\
\qquad \big(\textbf{let}\ \textit{segment-count}\ = \\
\qquad\quad (\textbf{if}\ X_1\ \textbf{then}\ 1\ \textbf{else}\ 0)\ +\ (\textbf{if}\ X_2\ \textbf{then}\ 1\ \textbf{else}\ 0)\ +\ (\textbf{if}\ X_3\ \textbf{then}\ 1\ \textbf{else}\ 0)\ + \\
\qquad\quad (\textbf{if}\ X_4\ \textbf{then}\ 1\ \textbf{else}\ 0)\ +\ (\textbf{if}\ X_5\ \textbf{then}\ 1\ \textbf{else}\ 0)\ +\ (\textbf{if}\ X_6\ \textbf{then}\ 1\ \textbf{else}\ 0)\ + \\
\qquad\quad (\textbf{if}\ X_7\ \textbf{then}\ 1\ \textbf{else}\ 0)\ +\ (\textbf{if}\ X_8\ \textbf{then}\ 1\ \textbf{else}\ 0)\ +\ (\textbf{if}\ X_9\ \textbf{then}\ 1\ \textbf{else}\ 0)\ + \\
\qquad\quad (\textbf{if}\ X_{10}\ \textbf{then}\ 1\ \textbf{else}\ 0)\ +\ (\textbf{if}\ X_{11}\ \textbf{then}\ 1\ \textbf{else}\ 0)\ +\ (\textbf{if}\ X_{12}\ \textbf{then}\ 1\ \textbf{else}\ 0)\ + \\
\qquad\quad (\textbf{if}\ X_{13}\ \textbf{then}\ 1\ \textbf{else}\ 0)\ +\ (\textbf{if}\ X_{14}\ \textbf{then}\ 1\ \textbf{else}\ 0) \\
\qquad \textbf{in}\quad \textit{segment-count}\ \ \big)\ \Big)
\end{array}
$$

Remark: This is not the most economical way of writing θ'; it is here written to show the similarity with θ.

We now invoke a SMT solver with arithmetic capabilities to solve the combined constraint $\Phi' \wedge \theta \wedge \theta'$ for **Version 3** of Yashi. (Note that we can also solve $\Phi' \wedge \theta'$ for **Version 1** of Yashi, in a much more succint way than in Example 18.)

In Figure 6 we show an encoding of $\Phi' \wedge \theta \wedge \theta'$ in the syntax of Z3. To test it, you can create a new file named `yashi3`, say, and then execute the command '`z3 yashi3`'. □

4.4 Using SMT Solvers with an API

For **Version 2** of Yashi we have to count the number of solutions – which means that, if we encode the Yashi constraints as propositional wff's, we have to count the number of distinct models of such wff's, *i.e.*, interpretations or truth-value assignments satisfying them. This is not a trivial problem. In fact, a SAT or SMT solver is not well equipped to do this kind of counting.

One way of bypassing the difficulty is to use an API (Application Program Interface) which can support model generation and keep a count of distinct models generated. The idea is to generate a model, and then to add constraints in order to prevent the same model from being generated in a subsequent satisfiability check. This process is iterated, with new constraints added after a new model is generated at every iteration, until there are no more models.

https://stackoverflow.com/questions/13395391/z3-finding-all-satisfying-models

https://github.com/Z3Prover/z3/issues/934

```
(declare-const X1 Bool)       (declare-const X2 Bool)
(declare-const X3 Bool)       (declare-const X4 Bool)
(declare-const X5 Bool)       (declare-const X6 Bool)
(declare-const X7 Bool)       (declare-const X8 Bool)
(declare-const X9 Bool)       (declare-const X10 Bool)
(declare-const X11 Bool)      (declare-const X12 Bool)
(declare-const X13 Bool)      (declare-const X14 Bool)

(define-fun yashi-solution () Bool
   (and (or (not X3) (not X13))                          ; no crossing
        (or (not X7) (not X13))                          ; no crossing
        (or (not X1) (not X5) (not X6) (not X9)
            (not X10) (not X11))                         ; no cycle
        (or (not X1) (not X3) (not X5) (not X6) (not X7)
            (not X9) (not X10) (not X14))                ; no cycle
        (or (not X1) (not X2) (not X5) (not X6) (not X8)
            (not X9) (not X12) (not X13))                ; no cycle
        (or (not X2) (not X8) (not X10) (not X11) (not X12)
            (not X13))                                   ; no cycle
        (or (not X3) (not X7) (not X11) (not X14))       ; no cycle
        (or X1 X9)                                       ; no one-node cut
        (or X1 X2 X10)                                   ; no one-node cut
        (or X2 X13)                                      ; no one-node cut
        (or X3 X10 X11)                                  ; no one-node cut
        (or X3 X14)                                      ; no one-node cut
         X4                                              ; no one-node cut
        (or X4 X5 X9)                                    ; no one-node cut
        (or X5 X6)                                       ; no one-node cut
        (or X6 X7 X11 X12)                               ; no one-node cut
        (or X7 X14)                                      ; no one-node cut
        (or X8 X12)                                      ; no one-node cut
        (or X8 X13)                                      ; no one-node cut
        (or X2 X9 X10)                                   ; no two-node cut
        (or X1 X10 X13)                                  ; no two-node cut
        (or X3 X7)                                       ; no two-node cut
        (or X5 X9)                                       ; no two-node cut
        (or X12 X13)                                     ; no two-node cut
        (or X1 X5)                                       ; no three-node cut
        (or X6 X9)                                       ; no three-node cut
        (or X9 X10 X13)                                  ; no three-node cut
        (or X1 X3 X11 X13)                               ; no three-node cut
        (or X1 X6)                                       ; no four-node cut
        (or X3 X9 X11 X13)                               ; no four-node cut
        (or X5 X10 X13)                                  ; no five-node cut
        (or X1 X7 X11 X13)                               ; no five-node cut
        (or X3 X6 X11 X13)                               ; no five-node cut
        (or X9 X11 X13 X14)                              ; no five-node cut
        (or X6 X10 X13)                                  ; no six-node cut
        (or X7 X9 X11 X13)                               ; no six-node cut
    ))
(assert yashi-solution)

(check-sat)
(get-model)
```

Figure 4: Script `yashi1` for Example 18.

```
(declare-const length Int)

(define-fun temp () Int
  (+ (ite X1 4 0) (ite X2 2 0) (ite X3 3 0) (ite X4 2 0)
     (ite X5 2 0) (ite X6 2 0) (ite X7 3 0) (ite X8 2 0)
     (ite X9 6 0) (ite X10 4 0) (ite X11 2 0) (ite X12 2 0)
     (ite X13 8 0) (ite X14 2 0)
     ))

(minimize temp)

(assert (= length temp))
```

Figure 5: Portion of `yashi2` for Example 19 – to be inserted after '`(assert yashi-solution)`' in `yashi1`.

```
(declare-const X1 Bool)        (declare-const X2 Bool)
(declare-const X3 Bool)        (declare-const X4 Bool)
(declare-const X5 Bool)        (declare-const X6 Bool)
(declare-const X7 Bool)        (declare-const X8 Bool)
(declare-const X9 Bool)        (declare-const X10 Bool)
(declare-const X11 Bool)       (declare-const X12 Bool)
(declare-const X13 Bool)       (declare-const X14 Bool)

(define-fun yashi-solution () Bool
   (and (or (not X3) (not X13))                          ; no crossing
        (or (not X7) (not X13))                          ; no crossing
        (or (not X1) (not X5) (not X6) (not X9)
            (not X10) (not X11))                         ; no cycle
        (or (not X1) (not X3) (not X5) (not X6) (not X7)
            (not X9) (not X10) (not X14))                ; no cycle
        (or (not X1) (not X2) (not X5) (not X6) (not X8)
            (not X9) (not X12) (not X13))                ; no cycle
        (or (not X2) (not X8) (not X10) (not X11) (not X12)
            (not X13))                                   ; no cycle
        (or (not X3) (not X7) (not X11) (not X14))       ; no cycle
    ))

(assert yashi-solution)

(declare-const length Int)

(define-fun temp () Int
  (+ (ite X1 4 0) (ite X2 2 0) (ite X3 3 0) (ite X4 2 0)
     (ite X5 2 0) (ite X6 2 0) (ite X7 3 0) (ite X8 2 0)
     (ite X9 6 0) (ite X10 4 0) (ite X11 2 0) (ite X12 2 0)
     (ite X13 8 0) (ite X14 2 0)   ))
(minimize temp)

(assert (= length temp))

(define-fun segment-count () Int
  (+ (ite X1 1 0) (ite X2 1 0) (ite X3 1 0) (ite X4 1 0)
     (ite X5 1 0) (ite X6 1 0) (ite X7 1 0) (ite X8 1 0)
     (ite X9 1 0) (ite X10 1 0) (ite X11 1 0) (ite X12 1 0)
     (ite X13 1 0) (ite X14 1 0)    ))

(assert (= segment-count 11))

(check-sat)
(get-model)
```

Figure 6: Script `yashi3` for Example 20.