Chapter 3

AUTOMATED TEST SYNTHESIS VIA NETWORK FLOWS: AN INTRODUCTION

3.1 Introduction
This chapter explores reactive test synthesis for discrete decision-making components in autonomous systems. This chapter originated from thinking about how to find a small set of difficult test cases for discrete decision-making behaviors. For safety as well as satisfying system requirements, a full-stack autonomous system must reason over its own state as well as about how the environment might react to its actions. Oftentimes, this involves reasoning over inputs and states that are both discrete and continuous valued, and implementations of autonomous systems accomplish this at various levels of abstraction. In this chapter, we formulate the test synthesis problem, and introduce the concept of a test objective.

This chapter is adapted from:

3.2 Related Work
Due to robustness metrics from their quantitative semantics, signal temporal logic (STL) and metric temporal logic (MTL), are natural paradigms for reasoning over trajectories of low-level continuous dynamics [37, 70]. In many instances, the term testing is used interchangeably with falsification [39]. Falsification is the problem of finding initial conditions and input signals that lead to violation of a temporal logic formula with the goal of finding such failures quickly and for black-box models [36, 41, 71, 72]. Furthermore, the black-box approaches in the related topics of falsification of hybrid systems [36], and simulation-based test generation [38, 42], rely on stochastic optimization algorithms to minimize the robustness of temporal logic satisfaction. Since dense-time temporal logics better encapsulate the range of system behaviors at the with continuous dynamics, these techniques are successful at falsification at the low-level. However, some of the complexity can be attributed to the coupling between continuous dynamics with high-level discrete
decision-making behaviors, a hierarchical approach to test-case generation could be effective.

Typically, high-level choices of autonomous robotic systems exhibit discrete decision-making \cite{73,74}, and LTL specifications are often used to capture mission objectives at higher levels of abstraction. Covering arrays have been used to initialize discrete parameters of the test configuration at the start of the falsification procedure in \cite{36,42,75}, but are not reactive. Furthermore, linear temporal logic (LTL) model checkers for testing has been explored in \cite{71,76-78}, in which counterexamples found via model-checking are used to exactly construct test cases. However, these are usually applied to deterministic systems, thus relying on the knowledge of the system controller, and become inconclusive if the system behavior deviates from the expected model. In this chapter and next, we focus on a framework for testing of high-level specifications in linear temporal logic (LTL) without assuming knowledge of the system controller.

### 3.3 Motivation

Here we adopt a different notion of testing – one that is focused on observing the autonomous agent undertake a certain behavior in its mission. The DARPA Urban Challenge test courses, that mainly comprised of static obstacles and (dynamic) human-driven cars, were carefully designed to observe the agent undertaking certain behaviors \cite{10}. For example, a part of the test course was designed for assessing parking behavior. The static obstacles – barriers blocking the region in front of the parking lot and other parked cars – were placed such that the agent had to repeatedly reverse/pull-in to incrementally adjust its heading angle before successfully parking in the designated spot. The clever placement of static obstacles in this scenario made it a challenging test for the agent, as opposed to an environment in which the agent pulls-in straight into the parking spot. Similarly, carefully designed scenarios with human-driven cars sought to observe other behaviors of the agent. In many, but not necessarily all, of these scenarios, the high-level behavior of the agent can be described as a sequence of waypoints. In the parking lot example, the sequence of waypoints can be characterized as a sequence of agent states, which can be characterized as a product of position and heading angle in the high-level abstraction. As a step towards automatically synthesizing these test scenarios, this chapter asks the following question.

**Problem (Informal):** Given a valid, user-defined sequence of waypoints, a reacha-
bility objective for the mission specification, find a set of possible initial conditions for the agent (if not specified by user) and determine a set of static constraints, characterized by transitions that are blocked/restricted, such that:

i. the agent must visit the sequence of waypoints in order before its goal, and

ii. the test environment is minimally restricted.

3.4 Preliminaries

Automata Theory and Temporal Logic

Definition 3.1 (Finite Transition System). A finite transition system (FTS) is the tuple

\[ TS := (S, A, \delta, S_0, AP, L) \]

where \( S \) denotes a finite set of states, \( A \) is a finite set of actions, \( \delta : S \times A \rightarrow S \) the transition relation, \( S_0 \) the set of initial states, \( AP \) the set of atomic propositions, and \( L : S \rightarrow 2^{AP} \) denotes the labeling function. We denote the transitions in \( TS \) as \( TS.E := \{(s, s') \in S \times S \mid \exists a \in A \text{ s.t. } \delta(s, a) = s'\} \). We refer to the states of \( TS \) as \( TS.S \), and similarly denote the other elements of the tuple. An execution \( \sigma \) is an infinite sequence \( \sigma = s_0s_1 \ldots \), where \( s_0 \in S_0 \) and \( s_k \in S \) is the state at time \( k \). We denote the finite prefix of the trace \( \sigma \) up to the current time \( k \) as \( \sigma_k \). A strategy \( \pi \) is a function \( \pi : (TS.S)^*TS.S \rightarrow TS.A \).

Definition 3.2 (System). The system under test is modeled as a finite transition system \( T_{sys} \) with a singleton initial set, \( |T_{sys}.S_0| = 1 \).

A directed graph \( G = (V, E) \) can be induced from \( T_{sys} \) in which the vertices represent states \( T_{sys}.S \) and the edges represent the transitions \( T_{sys}.E \), and the labeling function assigns propositions that are true at each vertex. For a proposition \( p \in AP \), and vertex \( v \in V \), \( v \vdash p \) means that \( p \) evaluates to \( \text{True} \) at \( v \). A run \( \sigma = s_0s_1 \ldots \) on the graph is an infinite sequence of its nodes where \( s_i \in T_{sys}.S \) represents the system state at time step \( i \).

We introduce the notion of a test harness to specify how the test environment can interact with the system. A test harness is used to constrain a state-action \( (s, a) \) pair of the system in the sense that the system is prevented from taking action \( a \) from state \( s \in T_{sys}.S \). Let the actions \( A_H \subseteq T_{sys}.A \) denote the subset of system actions that can be restricted by the test harness. The test harness \( H : T_{sys}.S \rightarrow 2^{AH} \) maps states of the transition system to actions that can be restricted from that state.
In the examples considered in this thesis, every state of the system has a self-loop transition corresponding to stay-in-place action, though the framework does not require this. Note that in our examples, \( A_H \) does not contain self-loop actions.

In this work, we synthesize tests for high-level decision-making components of the system under test and therefore model it as a discrete-state system. Linear temporal logic (LTL) has been effective in formally specifying safety and liveness requirements for discrete-decision making \[14, 15, 18\]. For our problem, we use LTL to capture the system and test objectives. The reach-avoid fragment of LTL is restricted to the use of logical operators and the next, always, and eventually temporal operators, and can capture a rich set of behaviors such as safety and coverage properties. Every LTL formula can be transformed into an equivalent non-deterministic Büchi automaton, which can then be converted to a deterministic Büchi automaton \[60\].

**Flow Networks**

We will leverage network flows to model the test synthesis problem. Flow networks are used in computer science to model several problems on graphs \[79\]. One of the main contributions of this thesis is in using flow networks for test synthesis.

**Definition 3.3** (Flow Network \[80\]). A flow network is a tuple \( N = (V, E, c, (V_s, V_t)) \), where \( V \) denotes the set of nodes, \( E \subseteq V \times V \) the set of edges, \( c \geq 0 \) represents edge capacity, \( V_s \subseteq V \) the source nodes, and \( V_t \subseteq V \) the sink nodes. On the flow network \( N \), we can define the flow vector \( f \in \mathbb{R}_{\geq 0}^{|E|} \) to satisfy the following constraints: i) the capacity constraint

\[
0 \leq f^e \leq c, \forall e \in E,
\]  

ii) the conservation constraint

\[
\sum_{u \in V} f^{(u,v)} = \sum_{u \in V} f^{(v,u)}, \forall v \in V \setminus \{V_s, V_t\}, \text{ and}
\]

iii) no flow into the source or out of the sink

\[
f^{(u,v)} = 0 \text{ if } u \in V_t \text{ or } v \in V_s.
\]

The flow value on the network \( N \) is defined as

\[
F := \sum_{(u,v) \in E, \ u \in V_s} f^{(u,v)}.
\]
In the following chapters, we will primarily be using the framework of network flows to identify restrictions on system actions, which will be analogous to cuts on a graph. An edge represents a transition the system can make. For this reason, it can suffice to define flow networks that have unit capacity $c = 1$ on all edges. Unless otherwise mentioned, all references to a flow network hereafter will assume unit edge capacities.

**Definition 3.4 (Cut [80]).** Given a graph $G = (V, E)$, a cut of $G$ is the tuple $\text{Cut} := (S, T)$ that partitions the vertices of $G$ into disjoint sets $S \subset V$ and $T \subset V$, that is, $S \cup T = V$ and $S \cap T = \emptyset$. The cut-set $C \subseteq E$ of the cut $\text{Cut} = (S, T)$ is the set of edges that when removed from $G$ results in the disjoint node sets $S$ and $T$:

$$C := \{(u, v) \in E \mid u \in S, v \in T\}. \quad (3.5)$$

The expression $G_{\text{cut}}(C) := (V, E \setminus C)$ refers to the graph resulting from removing the edges in $C$ from $G$. We will use the same expression to refer to any graph from which edges $C$ are removed, even if the set $C$ does not correspond to a Cut (i.e., complete partition of the graph $G$). Any edge that is removed from $G$ is referred to as an edge-cut.

In finding maximum flow, it becomes important to identify edges on the graph through which flow can be pushed through and track edges which have already been saturated. This is the concept of a residual network which is defined below. For a more detailed exposition with illustrations, see pages 726–727 of [80].

**Definition 3.5.** Given a graph $G = (V, E)$ and a flow $f$, the set of residual edges $E_f$ are defined as

$$E_f := E \cup \{(u, v) \mid (v, u) \in E \text{ and } f(v, u) > 0\}. \quad (3.6)$$

The corresponding residual network $G_f = (V, E_f)$ is a flow network with edge capacities $c_f : E_f \rightarrow [0, 1]$ defined as follows:

$$c_f(u, v) = \begin{cases} 
1 - f(u, v) & \text{if } (u, v) \in E \text{ and } f(u, v) < 1, \\
1 & \text{if } (v, u) \in E, \\
0 & \text{otherwise.}
\end{cases} \quad (3.7)$$

Standard algorithms such as Edmonds-Karp [79] use residual networks to find the maximum-flow (and equivalently, minimum-cut) of a single source-sink flow problem in a graph $G = (V, E)$ in $O(|V||E|^2)$ time. Roughly, starting with zero flow,
the Edmonds-Karp algorithm iteratively updates the residual network as flow from the source to target is found. Initially, the residual network is exactly the same as the original graph $G$. Then, the algorithm finds paths from source to target on the residual network until no such paths remain. These paths are known as augmenting paths, and are used to construct a realization of the maximum flow.

**Definition 3.6 (Augmenting Path [80])**. Given a graph $G = (V, E)$ with flow $f$, an *augmenting path* from a source $s \in V$ to target $t \in V$ is a simple path $Path(s, t)$ (i.e., without any cycles) from $s$ to $t$ on the corresponding residual network $G_f$.

The time complexity of the Edmonds-Karp algorithm comes from finding the shortest augmenting path on the residual network in each iteration until no paths remain. If $G$ has a maximum possible flow $F_{\text{max}}$, then the set of augmenting paths $\mathcal{AP} = \{Path_1(s, t), \ldots, Path_{F_{\text{max}}}(s, t)\}$ has cardinality $F_{\text{max}}$ since all edges in $G$ have unit capacities. That is, once an augmenting path is identified, all of the edges in the augmenting path are saturated on the residual network. As a result, any two paths $Path_i(s, t), Path_j(s, t) \in \mathcal{AP}$ are always *edge-disjoint* in the sense that there does not exist any edge $e \in E$ that is in both $Path_i(s, t)$ and $Path_j(s, t)$. The set of augmenting paths found by the Edmonds-Karp algorithm is denoted as the *set of shortest augment paths* $\mathcal{SAP}$. It can be proven that finding the shortest augmenting paths and updating the residual network accordingly results in finding a realization of the maximum flow [80].

**Proposition 3.1.** A set of shortest augmenting paths $\mathcal{AP}$ comprises of edge-disjoint paths.

### 3.5 Test Objective

We begin by considering reachability specifications as mission objectives for the agent under test. For the test itself, we wish to observe a sequence-like behavior of the agent in its attempt to satisfy its mission objectives. Formally, this test behavior can be described by the temporal logic formula given below.

**Definition 3.7 (Test Objective (strict sequence)).** The *test objective* for strict sequenced visit is given by the LTL formula:

$$
\varphi_{\text{test}} := \Diamond(p_1 \land \Diamond(p_2 \land \Diamond(\cdots \land \Diamond(p_n))) \bigwedge_{i=1}^{n-1} (\neg p_{i+1} \cup p_i), 
$$

(3.8)
where \( p_1, \ldots, p_n \in AP \) are propositional formulas. This is a sequence-like formula since the agent has to eventually visit every \( v_i \), but it cannot visit \( v_{i+1} \) before visiting \( v_i \), where \( v_i \models p_i \), for all \( i = 1, \ldots, n \).

The system under test does not have access to the test objective \( \varphi_{test} \). Since LTL formulae cannot be evaluated on finite test runs, the length of the test run depends on the time the agent takes to satisfy its mission objective.

**Example 3.1.** Consider the gridworld in Figure 3.1 on which the agent can transition between states (up, down, left, right) with the mission specification of reaching some goal state (formalized as \( \varphi_g = \Diamond g \)). Of the many possible paths the agent can take to meet its objective, we are interested in observing it navigate to the goal while restricted to a class of paths described by the test specification \( \varphi_{test} = \Diamond (p_1 \land \Diamond p_2) \). How would we constrain actions of the agent in certain states, such that it navigates through the sequence of waypoints before reaching the goal? Furthermore, is it possible to synthesize these constraints such that the sequence flow value from \( p_1 \) to \( g \) is maximized?

![Figure 3.1: Left: Unrestricted gridworld labeled by propositional formulas. Right: A test environment synthesized by our algorithm where the transitions (2, 1) \( \rightarrow \) (2, 2), (3, 1) \( \rightarrow \) (3, 2), (2, 2) \( \rightarrow \) (3, 2), and (2, 2) \( \rightarrow \) (2, 3) blocked. Red semi-circle patches illustrate one-way constraints, that is, transition from state \( u \) to state \( v \) is restricted, but \( v \) to \( u \) is allowed, if arch of the semi-circle is in the grid corresponding to \( u \) along the transition from \( u \) to \( v \).](image)

**Problem Statement**
Now, we formalize the test environment synthesis problem. We limit our focus to *static* test environments, by which we mean that the test environment does not react to the actions of the agent during the test, leaving the reactive test synthesis problem for later in this chapter and the next chapter.
Definition 3.8 (Test Graph). Given a labeled directed graph $G = (V, E)$, a mission/agent specification $\varphi_{sys} = \Diamond p_{n+1}$, a test specification $\varphi_{test}$ (equation (3.8)), a test graph $G_{cut}(C) = (V, E \setminus C)$ is the directed graph obtained by removing set of edges $C$ from the original graph $G$. On $G_{cut}(C)$, a run $\sigma$ starting from state $v_1$ will satisfy the specification, (3.9)

Definition 3.9 (Minimally Restricted Test Graph). A test graph $G_{cut}(C)$ is minimally restricted if the total sequence flow value from $v_1 \models p_1$ to $v_{n+1} \models p_{n+1}$ on $G_{cut}(C)$ are maximized.

Remark 3.1. In this chapter, the definition of a minimally restricted graph does not relate to the actual number of cuts in the cut-set $C$, but only whether the flow on $G_{cut}(C)$ is maximized.

Problem 3.1. Given a system specification $\psi_{sys} = \Diamond v_{n+1}$, a labeled directed graph $G = (V, E)$ induced by the non-deterministic transition model $T$ of the agent, a test specification $\varphi_{test} = \Diamond (p_1 \land \Diamond (p_2 \land \Diamond (\cdots \land \Diamond p_n)))$, static constraints $C \subseteq E$ such that on the resulting test graph $G_{cut}(C) = (V, E \setminus C)$ is a minimally restricted test graph.

Here we aim to find a cut that maximizes the flow from a waypoint $p_i$ to its consecutive waypoint $p_{i+1}$, while eliminating any flow to waypoints $p_j$ (for $j > i + 1$) for all $i = 1, \ldots, n$. In other words, some flows need to be cut while other flows should be maximized. The problem of constructing a minimally restricted test graph for observing a sequence-like specification can be cast as the following optimization,

$$\max_{C \subset E} \quad f_{G_{cut}(C)}$$
$$\text{s.t.} \quad f_{G_{cut}(C)}(v_i, v_{i+1}) \leq f_{G_{cut}(C)}(v_i, v_{j}) \quad \forall i = 1, \ldots, n - 1,$$
$$f_{G_{cut}(C)}(v_i, v_j) = 0 \quad \forall i = 1, \ldots, j - 2, \forall j = 3, \ldots, n, \quad (3.10)$$

where the variables $C \subset E$ are the set of edges to be restricted resulting in the unit-capacity graph $G_{cut}(C) = (V, E \setminus C)$, the scalar $f_{G_{cut}(C)}$ represents the total flow on $G_{cut}(C)$ from $v_1$ to $v_n$, the scalars $f_{G_{cut}(C)}(v_i, v_j)$ represent the total flow from source $v_i$ to sink $v_j$. The problem data is the original graph $G = (V, E)$ and the sequence nodes $v_1, \ldots, v_n$. Solving this optimization directly will require constructing an integer linear program (ILP), for which constructing the constraint set is not straightforward. Furthermore, it would require solving the integer program with $|E|$ number of integer variables.
3.6 Algorithm for Synthesizing Static Test Environments

Let $G = (V, E)$ be a directed graph, with unit capacity on every edge, induced by the transition system $T_{sys}$ of the system under test. Assuming that the test environment has complete freedom to “block” any transition in the graph $G$ allowed by the test harness $H$, Algorithm 4 returns a set of edges, $C \subseteq E$, of the graph $G$ that must be removed before the test run. First, we make following assumptions on $G$.

Let $d_G(v_1, v_2)$ denote the length of the shortest path from vertex $v_1$ to vertex $v_2$ on graph $G$.

**Assumption 3.1.** For each $i \in \{1, \ldots, n + 1\}$, let $v_i$ denote the vertex $v \in V$ s.t $v \vdash p_i$. Assume $|v_i| = 1$, for all $i = \{1, \ldots, n + 1\}$.

Informally, Assumption 3.1 states that every propositional formula, $p_1, \ldots, p_{n+1}$, has a single vertex in $G$ associated with it.

**Assumption 3.2.** There exists a set of edges $C \subseteq E$ such that the modified graph obtained by removing these edges, $G_{cut}(C) = (V, E \setminus C)$, is such that

$$d_{G_{cut}(C)}(v_1, v_{n+1}) > \cdots > d_{G_{cut}(C)}(v_n, v_{n+1}) > d_{G_{cut}(C)}(v_{n+1}, v_{n+1}) = 0. \quad (3.11)$$

Assumption 3.2 is equivalent to the statement that by removing some edges (or restricting certain transitions) from the original graph $G$, there exists some set of initial conditions $Q_0$ for which the only path(s) to the goal $g$ is through the behavior $\varphi_{test}$. This assumption is imperative since there might be instances for which it is impossible to construct a test graph. For example, in the following simple labeled graph (Figure 3.2), it is impossible to construct a test graph for the test specification $\varphi_{test} = \lozenge (p_1 \land \lozenge p_2)$. Once the system is in state $v_1$, it can directly proceed to the goal state $v_g$ without visiting $v_2$. For instances such as this one, a reactive test environment is necessary.

![Figure 3.2: An invalid configuration of propositional formulas for test specification $\varphi_{test} = \lozenge (p_1 \land \lozenge p_2)$](image)

**Overview of the Approach:** At a high-level, we identify all edge disjoint path combinations through the sequence specification to find edge restrictions. This can be seen as a brute-force approach to solving the problem. As we will discuss in the
following chapter, this problem is NP-hard, and therefore, it is not possible to find a
polynomial-time algorithm to solve this problem if we assume that P is not NP. Later
in this chapter and next, we will cover a more efficient optimization formulations
that can capture a wide-range of specification types with faster runtimes.

**Definition 3.10** (Sequence Path). Given a graph $G$ and atomic propositions char-
acterizing the sequence specification, $p_1, \ldots, p_n$, where $Path_G(v_i, v_{i+1})$ represent
a simple path from $v_i$ to $v_{i+1}$, for all $i = 1, \ldots, n$. The **sequence path** from $v_1$ to
$v_{n+1}$ can be constructed from the individual path segments as $Path_G(v_1, v_{n+1}) =
Path_G(v_1, v_2), \ldots, Path_G(v_n, v_{n+1})$. The sequence path $Path_G(v_1, v_{n+1})$ is **valid**
if it does not have a cycle involving two or more path segments. That is, if there
are no edges $(u, w), (w, v) \in E$ such that edge $(u, w) \in Path_G(v_i, v_{i+1})$ and $(w, v) \in Path_G(v_j, v_{j+1})$ for some $i + 1 \leq j \leq n + 1$, except for the case in
which both $w = v_{i+1}$ and $j = i + 1$. In other words, except for the sequence nodes
$v_1, \ldots, v_n$ that link individual segments, there are no common nodes linking an ear-
lier path segment to a later segment. Observe that existence of a valid sequence path implies that Assumption 3.2 is true.

**Finding Combinations of Augmenting Paths**

Maximum flow realizations on a graph need not be unique; there can exist more
than one set of augmenting paths to capture maximum flow between a source and
target. For some graph $G$, we will denote the maximum flow from source $s$ to target
t as $F_{\text{max}}(s, t)$ and a set of augmenting paths by $\mathcal{AP}_G(s, t)$ or $\mathcal{SAP}_G(s, t)$ for the set
of shortest augmenting paths. Let $\mathcal{AP}_G(s, t)$ correspondingly denote the **set of sets of augmenting paths**, and let $\mathcal{SAP}_G(s, t)$ correspondingly denote the **set of sets of shortest augmenting paths**. Note that $\mathcal{AP}_G(s, t)$ captures all possible realizations
of maximum flow from $s$ to $t$ on $G$, and $\mathcal{SAP}_G(s, t) \subseteq \mathcal{AP}_G(s, t)$. Intuitively,
since the shortest path need not be unique, the set $\mathcal{SAP}_G(s, t)$ could have multiple
elements.

By definition, on a test graph $G_{\text{cut}}$, the maximum sequence flow value will be
bounded as follows:

$$f_{G_{\text{cut}}}(v_1, v_{n+1}) \leq \min_{i=1, \ldots, n} f_{G_{\text{cut}}}(v_i, v_{i+1}).$$  \hspace{1cm} (3.12)

On a minimally restricted test graph, the total sequence flow $f_{G_{\text{cut}}}(v_1, v_{n+1})$ is max-
imized. For each $1 \leq i \leq n$, note that $\mathcal{SAP}_G(v_i, v_{i+1})$ is finite since the num-
ber of edges in $G$ are finite, but is combinatorial in the number of edges since
it requires enumerating simple paths from $v_i$ to $v_{i+1}$. The total number of augmenting path combinations from $v_1$ to $v_{n+1}$ that can realize the maximum flow $f_{\text{cut}}(v_1, v_{n+1})$ will be at most $\prod_{i=1}^{n} |AP_G(v_i, v_{i+1})|$. Not every augmenting path combination might lead to a valid test graph since there could exist a combination of augmenting paths that violates equation (5.2) by resulting in an invalid sequence path $\text{Path}(v_1, v_n)$. Consider the simple example of the $3 \times 3$ grid in Figure 3.4. The combination of sequence flows $(AP_G(v_1, v_2), AP_G^2(v_2, v_3), AP_G(v_3, v_4))$ will give us $f_{\text{cut}}(v_1, v_{n+1}) = 1$, but the combination of $(AP_G(v_1, v_2), AP_G^1(v_2, v_3), AP_G(v_3, v_4))$ does not have any valid sequence paths.

![Figure 3.3](image1.png)

Figure 3.3: Left: $3 \times 3$ grid for the sequence specification with atomic proposition $p_1$, $p_2$, and $p_3$. Right: Illustrated with cuts that route the flow from $p_1$ to $p_4$.

![Figure 3.4](image2.png)

Figure 3.4: In this $3 \times 3$ grid, the left and right figures illustrate two different augmenting path combinations. Each grid shows a realization of the flow for each pair of nodes: red: $(p_1, p_2)$, gold: $(p_2, p_3)$, and blue: $(p_3, p_4)$. The main difference between the two figures is in the flow from $p_2$ to $p_3$. In both figures, the augmenting paths characterizing the flow from $p_1$ to $p_2$ and $p_3$ to $p_4$ are the same: $AP_G(v_1, v_2) = \{P_1, P_2\}$ characterizes the maximum flow from $p_1$ to $p_2$, and $AP_G(v_3, v_4) = \{P_5, P_6\}$ characterizes the maximum flow from $p_3$ to $p_4$. On the left, $AP_G^1(v_2, v_3) = \{P_3, P_4\}$, and on the right, $AP_G^2(v_2, v_3) = \{P'_3, P'_4\}$. It is possible to form a sequence path on the right with $\text{Path}_G(v_1, v_4) = P_1, P'_3, P'_6$, but not on the left. This sequence path is exactly illustrated in Fig. 3.3(right).
Algorithm 4: Restrict Transitions

**Input:** $\phi_{\text{test}}, \phi_a, G = (V, E, L)$.

**Output:** $C \subseteq E$.

$p \leftarrow \{p_1, \ldots, p_n, p_{n+1}\}, V_p \leftarrow \{v_1, \ldots, v_n, v_{n+1}\} \quad \triangleright v_i \vdash p_i$

$P_{\text{cut}} \leftarrow \text{Find-Cut-Paths}(G, p)$

$C = \{\}$

if Assumption 3.3 then

$flg \leftarrow 1$

while $P_{\text{cut}} \neq \emptyset$ do

$E \leftarrow \text{Edges in } P_{\text{cut}}$

$A, P_{\text{keep}}, |A|, F_{\text{max}} \leftarrow \text{Sequence-Flows}(G, p, flg = 0) \quad \triangleright \text{Combinations of sequence flows}$

for all $j = 0, \ldots, |F|$ do

$A_f \leftarrow A(j)$

$P_{\text{keep}} \leftarrow P_{\text{keep}}(j) \quad \triangleright \text{Selecting a combination } (S_1, \ldots, S_n)$

$MC_{\text{keep}} \leftarrow \text{Min-Cut-Edges}(G, p, P_{\text{keep}})$

$D_{\text{keep}} \leftarrow \text{diag}(A_{\text{keep}})^{\frac{1}{2}}$

for all $A_f \in A_f$ do

$D_f \leftarrow \text{diag}(A_f)^{\frac{1}{2}}$

$A_{\text{cut}}, A_{\text{keep}}, D_{\text{keep}} \leftarrow \text{ILP-params}(P_{\text{cut}}, P_{\text{keep}}, MC_{\text{keep}})$

$x^*, f^*, b^* \leftarrow \text{ILP}(A_{\text{cut}}, A_{\text{keep}}, D_{\text{keep}}, A_f, D_f) \quad \triangleright \text{Call to ILP (3.10)}$

if $\mathbb{1}^T f^* = F_{\text{max}}$ then

$C_{\text{new}} \leftarrow \{e_i | x_i^* = 1\}$

$C \leftarrow C \cup C_{\text{new}}$

break \quad \triangleright \text{Breaking out of both for loops}$

$G \leftarrow G \setminus C_{\text{new}}$

$P_{\text{cut}} \leftarrow \text{Find-Cut-Paths}(G, p)$

To avoid this issue, the algorithm searches through all combinations of sequence flows before constructing the input to the ILP (3.10). Since this is an expensive computation, a further assumption on the input graph and set of propositions can ease this bottleneck.

**Assumption 3.3.** Let $f_i$ be the maximum flow on $G$ from source $v_i$ to target $v_{i+1}$.

Let $SAP_{G}(v_i, v_{i+1}) = \{SAP_{G}^1 = \{P_1, \ldots, P_{f_i}\}\}$, represent the set of sets of shortest augmenting paths that characterizes the flow from $v_i$ to $v_{i+1}$ on $G$. Then, there exists a combination $(SAP_{G}^1, \ldots, SAP_{G}^n)$ on which a maximum sequence flow can be characterized.

In other words, Assumption 3.3 allows us to reason over combinations of shortest augmenting path flows, which is combinatorial in all shortest paths, instead of
combinations of the set of augmenting flows, which exacerbates the combinatorial complexity by enumerating all possible paths. All shortest paths are a subset of all simple paths between two nodes.

3.7 Iterative Synthesis of Constraints

Algorithm 4 details how edge cuts are computed by iteratively solving the following integer linear program.

\[
\begin{align*}
\max_{x \in \mathbb{B}^n, f \in \mathbb{B}^l} & \quad 1^T f \\
\text{s.t.} & \quad 1 \leq A_{\text{cut}} x \\
& \quad A_{\text{keep}} x \leq D_{\text{keep}} b \\
& \quad b \leq A_{\text{keep}} x \\
& \quad D_{f} f \leq A_{f}(1 - b), \\
& \quad A_{f}(1 - b) - D_{f}1 + 1 \leq f,
\end{align*}
\] (3.13)

where \((x, b, f)\) are the optimization variables, and \(A_{\text{cut}} \in \mathbb{B}^{k \times n}\), \(A_{\text{keep}} \in \mathbb{B}^{m \times n}\), \(D_{\text{keep}} \in \mathbb{B}^{m \times m}\), \(D_{f} \in \mathbb{B}^{l \times l}\), \(A_{f} \in \mathbb{B}^{l \times m}\) are problem data described in more detail below.

**Variables:** The variable \(x \in \mathbb{B}^n\), where \(n = |E_{\text{cut}}|\), is the Boolean vector corresponding to edges \(E_{\text{cut}}\) such that for some \(k \leq n\), if \(x_k = 1\), then the corresponding edge is restricted, and \(x_k = 0\) means that it is left in the graph for future iterations. Given \(P_{\text{keep}} = (\text{SAP}^1_G, \ldots, \text{SAP}^n_G) \in \mathcal{P}_{\text{keep}}\), a combination of set of shortest augmenting paths, the variable \(b \in \mathbb{B}^m\) keeps track of whether an augmenting path in some \(\text{SAP}^i_G\) \((1 \leq i \leq n)\) is restricted or not.

For some \(k \leq m\), if \(b_k = 1\), then the corresponding augmenting path in some \(\text{SAP}^i_G\) has minimum-cut edge(s) restricted by the ILP, and \(b_k = 0\) if none of the minimum-cut edges of that augmented path have been restricted. The variable \(f \in \mathbb{B}^l\) is the sequence flow vector for a given sequence flow, \(S_f\), such that \(l = |S_f|\) is the number of edge-disjoint paths constituting the sequence flow.

**Constraints:** The first constraint of the ILP, \(A_{\text{cut}} x \geq 1\), enforces the requirement that each path in \(P \in P_{\text{cut}}\) is restricted. Each row of \(A_{\text{cut}}\) corresponds to a path \(P \in P_{\text{cut}}\). The \(q\)-th row of \(A_{\text{cut}}\) is constructed as follows:

\[
(A_{\text{cut}})_{q,r} = \begin{cases} 
1 & \text{if } E_{\text{cut}}(r) \in P = P_{\text{cut}}(q) \\
0 & \text{otherwise.}
\end{cases}
\] (3.14)
In the second and third constraints, $A_{\text{keep}}x \leq D_{\text{keep}}b$ and $b \leq A_{\text{keep}}x$, is used to determine the variable $b$ from the variable $x$. Each row of $A_{\text{keep}} \in \mathbb{B}^{m \times n}$ corresponds to some path $P \in \text{SAP}_i^G$, and $D_{\text{keep}} \in \mathbb{B}^{m \times m}$ is a diagonal matrix. Suppose the $q$-th row of $A_{\text{keep}}$ corresponds to a path $P \in \text{SAP}_i^G$ for $P_{\text{keep}} = (\text{SAP}_1^G, \ldots, \text{SAP}_n^G)$, and $MC_{\text{keep}}(i)$ is the set of minimum-cut edges on some path in $\text{SAP}_i^G$, then the $q$-th row is constructed as follows:

$$
(A_{\text{keep}})_{q,r} := \begin{cases} 
1, & \text{if } E_{\text{cut}}(r) \in P \cap MC_{\text{keep}}(i). \\
0, & \text{otherwise.}
\end{cases} \quad (3.15)
$$

The $q$-th diagonal entry of $D_{\text{keep}}$ stores the total number of minimum-cut edges in the path corresponding to the $q$-th row of $A_{\text{keep}}$:

$$
D_{\text{keep}} := \text{diag}(A_{\text{keep}}1) \quad (3.16)
$$

These two constraints ensure that for some $q \leq n$, $b_q = 1$ iff at least one minimum-cut edge on the path corresponding to the $q$-th row of $A_{\text{keep}}$ is restricted, and $b_q = 0$ iff none of the minimum-cut edges on the path corresponding to the $q$-th row of $A_{\text{keep}}$ are restricted.

The fourth and fifth constraints, $D_{\text{f}}f \leq A_{\text{f}}(1 - b)$ and $f \geq A_{\text{f}}(1 - b) - D_{\text{f}}1 + 1$, determine the flow value for a given set of sequence flow paths, $S_{\text{f}}$. Suppose the $q$-th row of the matrix $A_{\text{f}} \in \mathbb{B}^{l \times m}$ corresponds to some sequence flow path $P = (P_1, \ldots, P_n) \in S_{\text{f}}$. Let $R = (r_1, \ldots, r_n)$ denote the indices of the paths $P_1, \ldots, P_n$ according to the ordering of the paths constituting all $\text{SAP}_i^G$ that is consistent with the construction of $A_{\text{keep}}$ and $D_{\text{keep}}$. Then, the $q$-th row of $A_{\text{f}}$ is defined as follows:

$$
(A_{\text{f}})_{q,r} := \begin{cases} 
1, & \text{if } r = r_i \text{ for some } 1 \leq i \leq n. \\
0, & \text{otherwise.}
\end{cases} \quad (3.17)
$$

The $q$-th diagonal entry of matrix $D_{\text{f}} \in \mathbb{B}^{l \times l}$ stores the total number of ones in the $q$-th row of $A_{\text{f}}$:

$$
D_{\text{f}} := \text{diag}(A_{\text{f}}1). \quad (3.18)
$$

The fourth constraint ensures that if any of the constituent paths, $P_1, \ldots, P_n$, in the $q$-th sequence flow path $P = (P_1, \ldots, P_n) \in S_{\text{f}}$ (for $1 \leq q \leq l$), is restricted, then the flow value, $f_q = 0$. The last constraint ensures that if none of the constituent paths, $P_1, \ldots, P_n$, in the $q$-th sequence flow path $P = (P_1, \ldots, P_n) \in S_{\text{f}}$ (for $1 \leq q \leq l$), are restricted, then the flow value, $f_q = 1$. 
**Parameters:** The parameters used to construct the problem data for the ILP (3.13) are the set of paths that need to be restricted, $P_{\text{cut}}$, the set of paths whose combination constitutes sequence flow and should not be restricted, $P_{\text{keep}}$, and the set of minimum-cut edges, $MC_{\text{keep}}$, on the paths constituting $P_{\text{keep}}$. The set $P_{\text{keep}} = \{(\text{SAP}_G^1, \ldots, \text{SAP}_G^n) | \text{SAP}_G^i \in \text{SAP}_G^i \}$ is a set of all combinations of shortest augmenting paths in the sequence. For a given combination of sets of augmenting paths, $P_{\text{keep}} = (\text{SAP}_G^1, \ldots, \text{SAP}_G^n)$, with the cardinality of $\text{SAP}_G^i$ being denoted as follows: $k_i := |\text{SAP}_G^i|$, and $m := \Sigma_{i=1}^{n} k_i$. In $P_{\text{keep}}$, suppose a combination of augmenting paths, $S_f = \{P = (P_1, \ldots, P_n) \mid P_i \in \text{SAP}_G^i\}$, represents a sequence flow, then a matrix $A_f \in \mathbb{B}^{|S_f| \times m}$ can be constructed to represent the sequence flow $S_f$. This construction is outlined in the descriptions of Constraints of the ILP. An instance of $P_{\text{keep}}$ can have several sequence flows, $S_f$, and correspondingly, several matrices, $A_f$, all of which are collectively denoted by $A_f$. The set of all such $A_f$ is denoted by $A$, which has cardinality $|A| = |P_{\text{keep}}|$, since each $A_f$ corresponds to an instance of $P_{\text{keep}}$. The maximum sequence flow value is given by $F_{\text{max}}$.

**Cost Function:** The cost function computes the maximum sequence flow value. Algorithm 4 does not proceed to the next iteration of $P_{\text{cut}}$ until it finds the set of static constraints that return the maximum possible sequence flow value, $F_{\text{max}}$. To guarantee completeness of Algorithm 4, we need to prove that the cuts synthesized in prior iterations do not preclude feasibility of further iterations with regards to assumption 3.2. See Section 3.8 for complexity of the subroutines in Algorithm 4.

**Sub-routines of Algorithm 4**

The MIN-CUT-EDGES sub-routine takes as input a graph $G$, a list of propositions $\{p_1, \ldots, p_n\}$, and for each $1 \leq i \leq n$, a non-empty set of shortest augmenting paths for the source-sink pair $(v_i, v_{i+1})$. This sub-routine returns as output the set of minimum cut-set on those augmenting paths, which is then used in constructing the problem data for the ILP.

The SEQUENCE-FLows sub-routine takes as input a graph $G$, a list of propositions $\{p_1, \ldots, p_n\}$, and a parameter to indicate if Assumption 3.3 holds. It then computes the combination of all augmenting flows (or all shortest augmenting flows) that can result in a non-zero sequence flow from $v_1$ to $v_{n+1}$. It returns as output the set of all sets of matrices that capture sequence-flow paths, $A$, a set of $P_{\text{keep}} = \{(\text{SAP}_G^1, \ldots, \text{SAP}_G^n) | \text{SAP}_G^i \in \text{SAP}_G^i \}$, the total number of combinations, $|A|$, and the maximum possible sequence flow value, $F_{\text{max}}$, which is determined when $A$ is
The sub-routine \textsc{Find-Bypass-Paths} takes as input a graph \( G \) and list of propositions, \( \{p_1, \ldots, p_n\} \), and uses the Edmonds-Karp algorithm to find bypass paths for every source-sink pair \((v_i, v_j)\), where \( i + 1 \leq j \leq n + 1 \). Specifically, this sub-routine finds a set of shortest augmenting paths from \( v_i \) to \( v_j \), and to ensure that they are bypass paths, the sub-routine is applied on \( G_{ij} = G = (V \setminus V_k, E \setminus E(V_k)) \), where:

\[
V_k := \{v_k \mid 1 \leq k \leq n + 1 \text{ and } k \neq i, k \neq j\},
\]

and the edges associated with \( V_k \) are denoted by \( E_k \):

\[
E_k := \{(u, v) \in E \mid u \in V_k \text{ or } v \in V_k\}.
\]

All of these augmenting paths are collectively returned as the output \( P_{\text{cut}} \), and the edges constituting these cuts are denoted by \( E_{\text{cut}} \). Note that \( P_{\text{cut}} \) does not return all simple paths from \( v_i \) to \( v_{j>i+1} \), but just a set of edge-disjoint paths. As a result, transitions are iteratively restricted until \( P_{\text{cut}} \) is empty.

### 3.8 Characteristics of the Algorithm

**Lemma 3.1.** In a graph \( G = (V, E) \), let \( \mathcal{P} \) represent a maximal set of sequence flow paths from \( v_1 \) to \( v_n \). Let \( \mathcal{P}_{\text{cut}} \) be the set of paths that need to be restricted, with the edges constituting the paths in \( \mathcal{P}_{\text{cut}} \) denoted by \( E_{\text{cut}} \subset E \). Then, the set of constraint edges \( C \subseteq E_{\text{cut}} \) can be found such that \( C \) does not constrain any path in \( \mathcal{P} \).

**Proof.** A path \( P_{\text{cut}} \in \mathcal{P}_{\text{cut}} \) can be restricted by removing at least one of its constituent edges. The number of edges of \( P_{\text{cut}} \) that are not in some path \( P \in \mathcal{P} \) is non-zero, since otherwise it would imply that \( P_{\text{cut},i} \in \mathcal{P} \), and would not need to be restricted. The set \( C \) can simply be chosen by selecting one or more edges on every \( P_{\text{cut}} \in \mathcal{P}_{\text{cut}} \) that are not a part of some path in \( \mathcal{P} \).

**Proposition 3.2.** Let \( G_m = (V, E_m) \) denote the graph for which the \( m \)-th iteration of the ILP (3.13) synthesizes new cuts \( C_m \subset E_m \). Then, Assumption 3.2 is satisfied on \( G_{m+1} = (V, E_m \setminus C_m) \).

**Proof.** In the first iteration, from Assumption 3.2 we know there exists at least one test graph \( G' = (V, E \setminus C) \) that satisfies equation (3.11). Assume that the \( m \)-th iteration graph \( G_m = (V, E_m) \) also satisfies Assumption 3.2. We will show
by induction that the graph resulting from the the \((m + 1)\)-th iteration, \(G_{m+1} = (V, E_m \setminus C_m)\), also satisfies Assumption 3.2. By construction, Algorithm 4 chooses a combination of set of shortest augmenting paths \((\text{SAP}_G^1, \ldots, \text{SAP}_G^n)\), such that there exists a non-empty set of sequence flow paths \(\mathcal{F} = \{(P_1, \ldots, P_n) | P_i \in \text{SAP}_G^i\}\) such that the simple path from \(v_1\) to \(v_n\) characterized by \(\Gamma = (P_1, \ldots, P_n) \in \mathcal{F}\) does not form an \(ij\)-cycle for some \(i < j \leq n\). This implies that on the subgraph comprising of the edges in \(\Gamma\), equation (3.11) is satisfied.

If the maximum possible sequence flow in a minimally restricted test graph is \(F_{\text{max}}\), then we can find a combination \((\text{SAP}_G^1, \ldots, \text{SAP}_G^n)\) such that for each \(i = 1, \ldots, n\), there exists a subset of edge-disjoint paths carrying flow \(F_{\text{max}}\):

\[
S_i = \{P_i^1, \ldots, P_i^{F_{\text{max}}}\} \subseteq \text{SAP}_G^i,
\]

from which we can construct the set of sequence flow paths:

\[
\mathcal{F}' = \{(P_i^1, \ldots, P_n^{F_{\text{max}}}) | P_i^k \in S_i, 0 \leq k_i \leq F_{\text{max}}\} \subseteq \mathcal{F}.
\]

By construction of the input variables to the ILP (3.13), the constraints of ILP (3.13) require that the sequence flow variable \(f\) has at least one element that is 1. This is possible only if there exists a set of edges \(C_m\) that constrain \(A_{\text{cut},m}\) such that there exists at least one sequence path \(P \in \mathcal{F}\) that does not have any of its minimum-cut edges restricted, which is true as shown in Lemma (3.1). Therefore, the new graph \(G_{m+1} = (V, E_m \setminus C_m)\) satisfies Assumption 3.2. □ □

**Theorem 3.1.** Under Assumption 3.2, Algorithm 4 is complete and returns a test graph \(G'\) from Definition 3.8 that satisfies equation (3.11).

**Proof.** Consider iteration \(m\) of the outer while loop in Algorithm 4 and let the graph at the \(m\)-th iteration be \(G_m = (V, E_m)\). Denote \(V_p = \{v_i | v_i \vdash p_i, \forall 1 \leq i \leq n+1\}\). Let \(F_{\text{max}}^{i,j}\) denote the maximum flow value from \(v_i\) to \(v_j\) on \(G_{ij} = (V \setminus (V_p \setminus \{v_i, v_j\}), E_m)\), for some \(i, j\) such that \(1 \leq i < j - 1 \leq n\). That is, \(G_{ij}\) is a copy of \(G_m\), but with nodes in \(V_p\), except for source \(v_i\) and sink \(v_j\), removed.

This implies that there is a set \(\text{SAP}_{G_{ij}}^{i,j}\) of \(F_{\text{max}}^{i,j}\) edge-disjoint paths that characterize the maximum flow from \(v_i\) to \(v_j\) on \(G_{ij}\). Let \((P_{i,j}(k))_m\) be the set of all simple paths from \(v_i\) to \(v_j\) that share an edge with the \(k\)-th path in \(\text{SAP}_{G_{ij}}^{i,j}\). Let \((MC_{i,j})_m\) be the set of minimum-cut edges on the paths in \(\text{SAP}_{G_{ij}}^{i,j}\) and let \((E_{i,j})_m \subseteq E_m\) be the set of all edges on some path from \(v_i\) to \(v_j\) on \(G_{ij}\). Clearly, \((MC_{i,j})_m \subseteq (E_{i,j})_m\).
For every $m \geq 1$, we can claim that $|(E_{i,j})_{m+1}| < |(E_{i,j})_m|$ because edges are removed to constrain SAP$_{G_{ij}}^{i,j}$ in the $m$-th iteration. Let $\tilde{m}$ be the number of iterations for $G_{ij}$ to become disjoint. In the worst-case, edges continue to be restricted until iteration $\tilde{m}$ at which $(E_{i,j})_{\tilde{m}} = (MC_{i,j})_{\tilde{m}}$, at which point constraining edges to cut (SAP$_{G_{ij}}^{i,j}$)$_{\tilde{m}}$ results in a cut separating $v_i$ and $v_j$. Thus, $\tilde{m}$ has to be finite for every such $i, j$.

At the same time, from Proposition 3.2, the synthesized cuts are such that Assumption 3.2 is maintained as an invariant. Therefore, when the last set of paths SAP$_{G_{ij}}^{i,j}$ are restricted, the final test graph $G'$ is such that $d_{G'}(v_1, v_{n+1}) > \ldots > d_{G'}(v_n, v_{n+1})$.

In addition to Assumption 3.2, if Assumption 3.3 holds, Algorithm 4 can be modified by a parameter setting. The proof of Theorem 3.1 still holds.

**Lemma 3.2.** On the test graph $G'$, any test run $\sigma$ starting from state $v_1$ will satisfy the specification (5.2).

**Proof.** From Assumption 3.1 there is only one node in $G'$ for each proposition in characterizing the test specification (5.2), and node satisfying proposition $p_i$ is labeled as $v_i$. For every $i \in \{1, \ldots, n\}$, $v_i$ is the only state in test graph $G'$ that is successor to all states $v$ on paths $\text{Paths}(v_{j<i}, v_{n+1})$ for which $d_{G'}(v, v_{n+1}) = d_{G'}(v_i, v_{n+1}) + 1$. This is true by construction of the ILP constraints. All paths in the set $\text{Paths}(p_{j<i}, g)$ on the test graph $G'$ must pass through $v_i$.

Let $\sigma$ denote the test run of the agent starting at $v_1$. We define a metric on the test graph $G'$: $m_t := \min_t d_{G'}(\sigma_t, v_{n+1})$ to be the closest distance to node $v_{n+1}$ in the first $t$ steps of the test run. Note three properties of this metric $m_t$: (a) $m_t \geq 0$, (b) $m_t$ decreases: $m_{t+1} := \min\{\sigma_{t+1}, m_t\} \leq m_t$, and (c) there exists a successor $q_{t+1}$ to $\sigma_t = q_t$ on $G'$ such that $d_{G'}(q_{t+1}, v_{n+1}) = d_{G'}(q_t, v_{n+1}) - 1$ that decreases $m_t$. The metric $m_t$ starts at $m_0 \geq d_{G'}(v_1, v_{n+1})$ and decreases to 0 at the end of the test run. Thus, we can observe that $\sigma \models \Box(p_1 \land \Box(p_2 \land \ldots \land \Box(p_{n+1}))) \land_{i=1}^n \neg p_{i+1} \cup p_i \iff \sigma \models \Box v_{n+1}$.

From Theorem 3.1 and Lemma 3.2, Algorithm 4 synthesizes a test graph $G'$ for the test specification (5.2), solving Problem 3.1.
Lemma 3.3. Consider the test graph $G'$ from Definition 3.8 for the test specification $\sigma$ from (5.2). If Assumption 3.2 holds, Algorithm 4 returns a minimally restricted test graph.

Proof. By construction, the inputs to the ILP (3.13) are constructed based on a maximal set of sequence flow paths from $v_1$ to $v_n$. By Lemma 3.1 at each iteration of the ILP (3.13) from which constraint edges are chosen, the maximum sequence flow value does not decrease at each iteration. Since there are a finite number of edges, there are a finite number of iterations until test graph is found. Therefore, the Algorithm 4 returns a minimally restricted test graph.

Complexity of Subroutines in Algorithm 4
Since Find-Cut-Paths is determining a set of augmenting paths for a single source-sink flow, it has a complexity of Edmonds-Karp algorithm, $O(|V||E|^2)$ time for graph $G = (V, E)$ [79]. The complexity of Min-cut-Edges is $O(|V||E|^3)$ time since it runs a max-flow algorithm for each edge in the worst-case. The main computational bottleneck is in the Sequence-Flows subroutine, which constructs sets of augmenting flows by computing combinations of all simple paths and all shortest paths. In the worst-case, enumerating all simple paths between two nodes is $O(|V|!)$, and enumerating all shortest paths is slightly better in several cases.

3.9 Examples
We illustrate the iterative synthesis of restrictions on a simple graph and a small gridworld, and then show runtimes of Algorithm 4 on random gridworld instances for both the case for which Assumption 3.2 is true, and the case for which Assumptions 3.2 and 3.3 are true.

Simple graph: Consider a simple non-deterministic Kripke structure representing an autonomous agent, shown in Figure 3.5, with propositional formulas labeled adjoining the states. The agent mission objective is to reach $g$ while being restricted to start from state $q_0$. The test environment seeks to restrict transitions such that the agent is prompted to pass through waypoint $w$ in its trajectory to $g$.

Inputs to Algorithm 4 include the labeled graph $G$ induced by the Kripke structure, the agent specification $\Diamond p_3$, the test specification $\Diamond p_2$, and the initial condition constraint $\Diamond p_1$. Algorithm 4 constrains the edges $\{(v_2, v_4), (v_4, v_6)\}$ in the first iteration, and the edges $\{(v_2, v_5), (v_5, v_6)\}$ in the second iteration. Although in this simple example, searching the set of all augmented paths becomes searching over...
all paths, in larger examples discussed below, each augmented path represents a class of paths that share some edge(s) with it.

Figure 3.5: **Left:** Simple Kripke structure representing states that the agent can occupy. The waypoint, $w$, is highlighted in purple to indicate that transitions are restricted corresponding to propositional formula $p_2 = L(w)$. **Right:** A test graph. Dashed edges in red illustrate transitions that have been restricted/removed from the Kripke structure above.

**Simple Gridworld:** In Figure 3.6, we illustrate the iterative synthesis of obstacles in a gridworld instance. Note that this configuration can be synthesized only by considering all sets of augmenting paths between $(p_1, p_2)$ and $(p_2, p_3)$. Since there is no shortest augmenting path from $p_2$ to $p_3$ that does not form a cycle with some (in this example, there is only one) shortest augmenting path flow from $p_1$ to $p_2$, it is imperative to use all sets of augmenting paths in the Sequence-Flows subroutine.

**Random Gridworld Instances:** For the case of setting all augmenting paths in the Sequence-Flows subroutine, we ran 50 random instances each for small gridworlds and propositions and plotted the average runtimes in Figure 3.7a. The number of propositions are limited by the size of the gridworld instances, which is restricted by the combinatorial nature of finding all sets of augmenting paths, and all combinations of sets of augmenting paths.

If we choose initial gridworld instances that satisfy Assumption 3.3, then Algorithm 4 can synthesize static constraints for slightly larger $t \times t$ grid sizes. The average runtimes for 50 random iterations for various grid sizes $t$ is plotted in Figure 3.7b. The small increase to larger grid size is due to the Sequence-Flows subroutine reasoning over shortest augmenting paths, and not all augmenting paths.

The average runtimes increase exponentially with the size of the grid. The number of propositions, denoted by $|P|$, is labeled $n$ if the test specification $\varphi_{test}$ (5.2) is comprised of propositions $(p_1, \ldots, p_n)$. In both Figures 3.7a and 3.7b, the average runtime for fewer propositions is at times higher than the average runtime for more propositions. This can be attributed to the Sequence-Flows subroutine
Figure 3.6: Synthesizing static test environment for $\varphi_{test} = \Diamond (p_1 \land \Diamond p_2) \land \neg p_2 \lor p_1$ and $\varphi_a = \Diamond g$.

Another paradigm for the problem of synthesizing static test environments for sequence behaviors could be multi-commodity network flows, which will be explored later in this chapter. The multi-commodity flow setting typically considers multiple source-sink flows simultaneously drawing from the capacity of each edge, and here we compute separate network flows for every source-sink pair of nodes.

### 3.10 Conclusions

An algorithm to synthesize a static test environment to observe sequence-like behavior in a discrete-transition system was introduced. First, we formulated this test environment synthesis problem as a problem of synthesizing cuts on graphs using concepts of flow networks. Then, we proposed an algorithm which synthesized the cuts iteratively using an integer linear program. We proved that this algorithm is
complete and that the edges restricted by the ILP at each iteration maintain feasibility of the constraint in the next iteration. Finally, we conducted numerical experiments on random gridworld instances to assess the runtime of our algorithm. Simulation results preclude this algorithm from being tractable to larger examples. However, the integer linear program requires reasoning over all possible paths on the transition system in order to identify the set of augmenting paths with the highest flow. This essentially becomes a brute-force approach to finding a set of edge disjoint paths from \( S \) to \( T \) that are routed through the propositions \( p_1, \ldots, p_n \) in a sequence. The poor scalability is due to the exponential number of constraints (in the number of edges of \( T_{sys} \)) in the ILP formulation. To alleviate this, we will present an alternative flow-based formulation in the form of a min-max game with coupled constraints.
Bibliography


176


