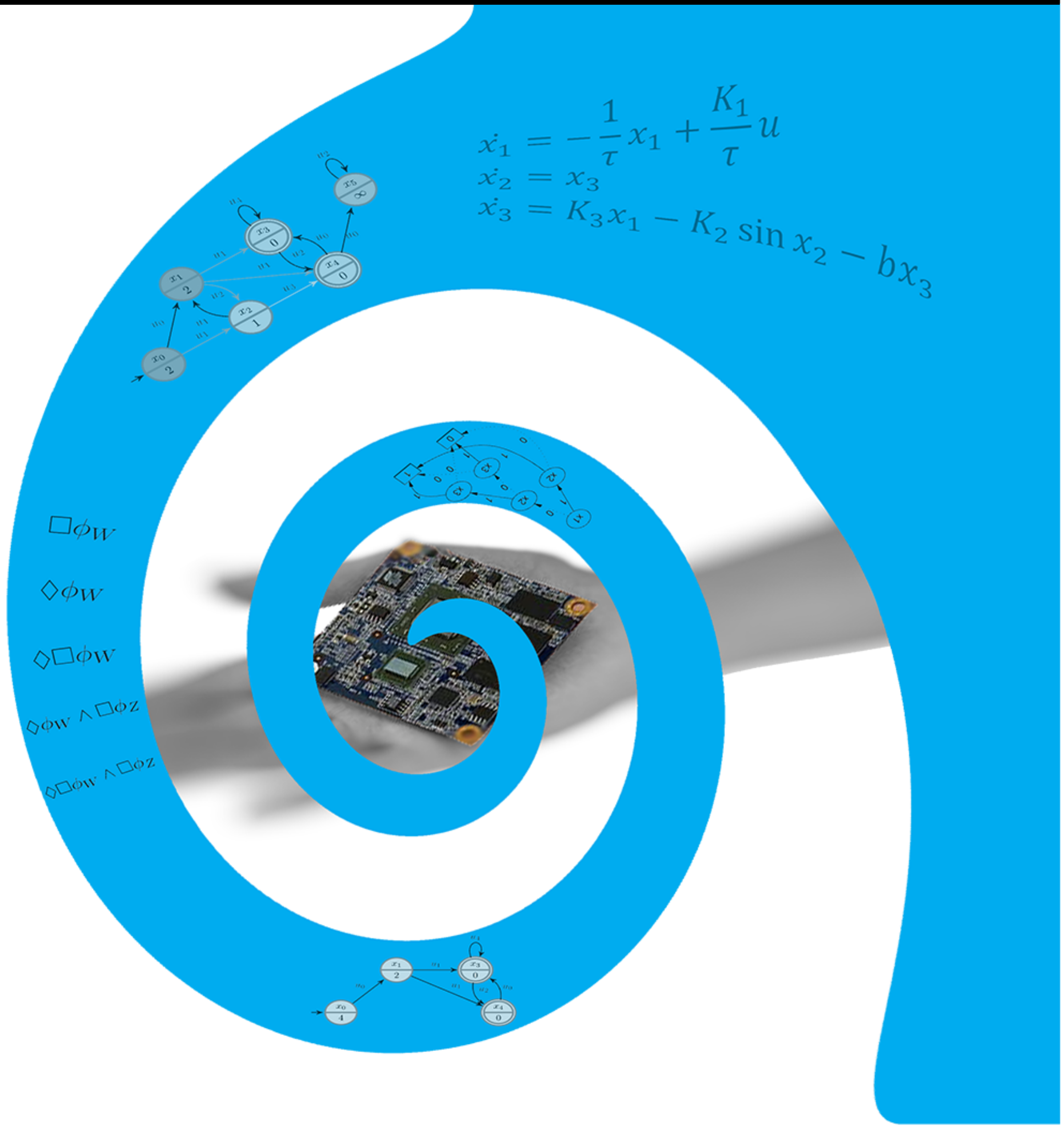
An approach to the automatic synthesis of controllers with mixed qualitative/quantitative specifications.

Athanasios Tasoglou

Master of Science Thesis



$\dot{x_1} = -\frac{1}{\tau}x_1 + \frac{K_1}{\tau}u$
$\dot{x_2} = x_3$
$\dot{x_3} = K_3x_1 - K_2 \sin x_2 - bx_3$
$\Box\phi_W$
$\Diamond\phi_W$
$\Diamond\Box\phi_W$
$\Diamond\phi_W \wedge \Box\phi_Z$
$\Diamond\Box\phi_W \wedge \Box\phi_Z$

TU Delft Delft University of Technology

Delft Center for Systems and Control

An approach to the automatic synthesis of controllers with mixed qualitative/quantitative specifications.

Master of Science Thesis

For the degree of Master of Science in Embedded Systems at Delft University of Technology

Athanasios Tasoglou

October 11, 2013

Faculty of Electrical Engineering, Mathematics and Computer Science (EWI) · Delft University of Technology

*Cover by Orestis Gartaganis

Abstract

The world of systems and control guides more of our lives than most of us realize. Most of the products we rely on today are actually systems comprised of mechanical, electrical or electronic components. Engineering these complex systems is a challenge, as their ever growing complexity has made the analysis and the design of such systems an ambitious task. This urged the need to explore new methods to mitigate the complexity and to create simplified models. The answer to these new challenges? *Abstractions*. An abstraction of the the continuous dynamics is a *symbolic model*, where each "symbol" corresponds to an "aggregate" of states in the continuous model. Symbolic models enable the *correct-by-design* synthesis of controllers and the synthesis of controllers for classes of specifications that traditionally have not been considered in the context of continuous control systems. These include *qualitative* specifications formalized using temporal logics, such as Linear Temporal Logic (LTL). Besides addressing qualitative specifications, we are also interested in synthesizing controllers with *quantitative* specifications, in order to solve optimal control problems. To date, the use of symbolic models for solving optimal control problems, is not well explored. This MSc Thesis presents a new approach towards solving problems of optimal control. Without loss of generality, such control problems are considered as path-planning problems on finite graphs, for which we provide two shortest path algorithms; one deterministic Set-Destination Shortest Path (SDSP) algorithm and one non-deterministic SDSP algorithm, in order to solve problems with quantitative specifications in both deterministic and non-deterministic systems. The fact that certain classes of qualitative specifications result in the synthesis of (maximally-permissive) controllers, enables us to use the SDSP algorithms to also enforce quantitative specifications. This, however, is not the only path towards our goal of synthesizing controllers with mixed qualitative-quantitative specifications; it is possible to use the SDSP algorithms directly to synthesize controllers for the same classes of specifications. Finally, we implement the algorithms as an extension to the `MATLAB` toolbox `Pessoa`, using Binary Decision Diagrams (BDDs) as our main data structure.

Table of Contents

List of Figures

List of Tables

Acknowledgements

First and foremost I offer my sincerest gratitude to my parents Emmanouil and Maria, for their support, encouragement and unwavering belief in me. Without you, I would not be the person I am today.

Secondly, I would like to thank Dr. ir. Manuel Mazo Jr., my daily supervisor, for his continuous and valuable guidance and advice. His willingness to give his time so generously has been very much appreciated. One simply could not wish for a better or friendlier supervisor.

To my friends and roommates, thank you for listening, offering me advice, and supporting me through this entire process. Last but not least, I would like to thank Tesi for her love, constant support and for keeping me sane over the past few months.

Delft, University of Technology
October 11, 2013

Athanasios Tasoglou

Chapter 1

Introduction

The dynamics of processes found in nature or made by humans are traditionally modeled using continuous systems, i.e., by a set of differential equations capturing the time evolution of quantities of interest. Continuous systems are also essential when modeling and designing dynamical or hybrid control systems. Nevertheless, many problems in systems and control are NP-hard and, in some cases, undecidable [3, 4]. They require numerical methods, whose computational complexity often grows rapidly as the state dimension and the number of time steps increase. This growth in the complexity of dynamical systems poses new challenges that fall beyond the traditional methods of control theory. To treat such problems, simpler models can be created that are much easier to analyze and to control, while preserving the essential characteristics of the original model. Such simplified models are called *symbolic* or *discrete-abstraction* of the original model in the sense of Willems [5, 6].

This approach though, introduces the problem of equivalence of systems, which is of great importance to systems and control theory [7]. For this, the notion of bisimulation is a powerful mathematical framework for addressing systems abstraction. A bisimulation relation requires all external behaviors of the simplified system to be equal to the external behaviors of the original one. In the context of control systems, one can safely assume that the trajectories of the *bisimilar* discrete system would include the trajectories of the continuous one. Such a discrete system is said to *simulate* the original system. The work of bisimulation originated in the field of labeled transition systems [8] and was introduced for dynamical and control systems by Haghverdi et al. [9]. While labeled transition systems are purely discrete, the dynamical systems in control theory may consist of both continuous and discrete variables. However, for systems observed over metric spaces, requiring strict equality of observed behaviors is often too strong. Only a small class of continuous or hybrid systems admits bisimilar discrete abstractions [1]. To address this problem, a different approach emerged through the work of [10, 11, 12, 13], where an approximate version of bisimulation is considered. While *exact* bisimulation requires the external behavior of two systems to be identical, the notion of approximate bisimulation relaxes this condition by allowing observations to be simply within a desired precision. This relaxation made it possible to extend the class of systems for which discrete abstractions can be computed, providing a more robust relationship between systems. Whether using exact or approximate bisimulation relations to abstract a system, the goal remains to construct symbolic models with a finite number of states, which is especially useful for controller design.

Symbolic models pose another important property: they allow us to use tools, that have been successfully used by computer scientists in the past decades, for analysis, design and verification of labeled transition systems. In fact, they are especially well suited for automated analysis and design which is becoming increasingly important given the size of nowadays complex control systems. From the analysis point of view, symbolic models provide a unified framework to describe continuous systems as well as hardware and software interacting with the physical environment. But most importantly, they enable the synthesis of controllers for classes of specifications that traditionally have not been considered in the context of continuous control systems.

These include specifications formalized using regular languages, fairness constraints, temporal logics, etc. Particularly Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) [14] are being used as specification languages for complex control systems. Such logics are appealing because they have well defined syntax and semantics, which can be easily used to specify complex behavior. In LTL for instance, it is easy to specify persistent tasks, e.g., "Visit regions A, then B, and then C, infinitely often. Never enter B unless coming directly from D." In addition, off-the-shelf model checking algorithms [14] and temporal logic game strategies [15] can be used to verify the correctness of system trajectories and to synthesize provably *correct-by-design* control strategies [11, 16, 17]. These control strategies basically enforce safety and liveness constraints or more generally, satisfy *qualitative* specifications.

The term qualitative refers to the fact that all the desired trajectories are treated as being equally good, as all undesired trajectories have been precluded from the system. However in many practical applications, this might not be enough. There might also be the need to select the "best" of the remaining trajectories. For that, typically, each trajectory is mapped with a cost stating how good a given trajectory is and defining the *quantitative* properties of the controller. The control design problem then requires the removal of the undesirable trajectories and the selection of the minimum cost trajectory. Although this design objective is very attractive for solving optimal control problems, it has been less considered within the context of discrete abstractions [18, 19, 20, 21, 22, 23].

1-1 Motivation and Related Work

To date, the use of discrete abstractions for solving optimal control problems, which is also our problem of interest, is not well explored. Some early attempts towards solving optimal control problems by using finite state representations, have been reported by Broucke et al [18]. In their work, they recast the continuous optimal control problem to a hybrid optimal control problem. Then they use finite bisimulation relations to transform the (non-linear) hybrid system into a finite automaton and in turn transform the initial problem into synthesizing a discrete supervisor, i.e. a scheme for switching between automaton locations, that will minimize a discrete cost function. The discrete cost function is considered as an approximation of the continuous one and their problem is equivalent to a shortest path problem on non-deterministic graph.

In more recent approaches [22, 24], approximately bisimilar abstractions are being used to design sub-optimal controllers for the fixed-bounded horizon optimal control problem. Approximate (alternating) bisimilar abstractions are especially suited for control problems with a quantitative performance measure, as they give more flexibility in the abstraction process. In particular they have been successfully used, to solve time-optimal control problems [22, 23, 25]. The key contribution of these approaches is to provide upper and lower bounds for the time to reach a target, while satisfying liveness and safety constrains. This way, the resulting approximately time-optimal controllers guarantee certain performance under given qualitative specifications.

In the latest work of Roo and Mazo Jr. [26] more general optimal control specifications are being addressed. More precisely, they provide the underlying theory on how to combine the quantitative properties of a system with the desired qualitative specifications, to solve mixed qualitative-quantitative optimal control problems. Without loss of generality, such control problems are being considered as path-planning problems on finite graphs, most of which can be solved using novel search techniques.

With respect to the aforementioned research, this MSc Thesis aims on providing such techniques, to address the practicality of this matter.

1-2 Thesis Goal and Contribution

The main contribution of this Thesis is to present a novel algorithmic solution for the mixed qualitative-quantitative optimal control problems discussed in [26] and to illustrate a way on how to actually synthesize such controllers. As a first step towards solving these problems is to construct a discrete abstraction of the continuous dynamics of the system and then to synthesize a controller under given safety and liveness constraints. Although it is possible to synthesize more than one controller at this point, we (usually) care about synthesizing a controller that is as permissive as possible. These so called *minimally-restrictive* or equivalently *maximally-permissive* controllers [27], allow us, given a state, to choose an input from a set of possible inputs that are equally good. Hence, we are interested in choosing a sequence of inputs that yield the best trajectory by means of a cost. To achieve that, a cost value is being given to each of the states of the controller's state space. In this way each transition is now characterized by some cost value and the optimal control problem is reduced to finding the shortest path on a finite graph. For that, known algorithms are available from the field of computer science, such as Dijkstra's algorithm [28] for the Single-Source Shortest Path (SSSP) problem or Floyd-Warshall algorithm [29] for the All-Pairs Shortest Path (APSP) problem. But, since we are interested in finding the "all-sources" shortest path to a given set (or the Set-Destination Shortest Path (SDSP)), we propose a modified version of the Floyd-Warshall algorithm to solve the optimal control problem. This approach though, is only well suited for deterministic systems (or graphs) due to the inability of the shortest path algorithms to handle non-deterministic transitions. For this reason, we also present a non-deterministic shortest path algorithm to address optimal control problems in non-deterministic systems.

The implementation of the two algorithms is based on a data structure known as *Binary Decision Diagram (BDD)* [30] and its extension called *Algebraic Decision Diagram (ADD)* [31]. BDDs have been used for years to provide a cogent representation of Boolean functions and to reduce the amount of space and computation required to verify digital circuits. BDDs represent Boolean functions as a directed acyclic graph - essentially a decision tree with re-convergent branches and binary terminal nodes. On the other hand, ADDs have real-valued terminals, and are considered as an efficient mean to represent and perform arithmetic operations on functions from a factored boolean domain to a real-valued range. It has been shown, for example, that for any matrix, the ADD representation is no larger than the corresponding sparse-matrix representation and is often smaller than any other conventional special-case representation, such as a $n \times n$ Walsh matrix [32]. ADDs provide also an ideal form of storing discrete abstractions of systems by offering great compression, while being fairly easy to use and manipulate. This space-optimal representation, can then be used to automatically generate hardware [33] or software [34] implementations of the controller.

To illustrate the feasibility of our proposed approach, we present some simple examples in chapter 4. These examples show the procedure for synthesizing controllers with mixed qualitative and quantitative specifications using our proposed approach. It is important to mention here that the resulting algorithms, serve as an extension to the freely available `MATLAB` toolbox `Pessoa` [2].

1-3 Thesis Structure

The rest of the thesis is structured as follows. Chapter 2 and 3 contain most of the terminology and notation used in this thesis. More precisely chapter 2 provides the fundamental knowledge on some concepts of graph theory, upon which the shortest path algorithms are based and operate. BDDs and ADDs, a special case of directed graphs (digraphs), which serve also as a data structure in the practical implementation, are also extensively covered in this chapter. In this chapter we also present two very important shortest path algorithms; the all-pairs shortest path algorithm of Floyd and Warshall and the single source shortest path algorithm of Dijkstra. Furthermore, chapter 2 covers important notions of systems, simulation relations and composition of systems. Most importantly, a fixed-point algorithm that is used to synthesize controllers with liveness constraints is being reviewed. This algorithm provides a solution to the so called “reachability game” and it is able to handle both deterministic and non-deterministic systems. In fact, the non-deterministic shortest path algorithm presented in this thesis is inspired by this particular algorithm. The last section of chapter two, can be considered as a quick overview on formal languages with a focus on the LTL language, which is used to formulate qualitative specifications.

Chapter 3 and 4 contain the main contribution of this thesis. In chapter 3 the deterministic and non-deterministic set-destination shortest path algorithms are presented. These algorithms play a key role to the solution of optimal control problems. Chapter 4 explains which classes of qualitative specifications can be addressed with the SDSP algorithms and the technique to synthesize controllers with mixed qualitative-quantitative specifications. In this chapter we provide also some examples to illustrate the feasibility of our approach.

Chapter 5 we present our conclusions and discuss potential future work that may be carried out. Appendix A provides the ADD implementation of the algorithms presented in this Thesis.

Chapter 2

Preliminaries

By making use of discrete abstract models, control problems are converted into path-planning problems on finite automata or more generally on finite graphs. In fact, the solutions provided for the optimal control problems in this thesis, are based on finding the shortest path in finite graphs. In particular, for solving the deterministic and non-deterministic shortest path problems presented later on, well-known shortest path algorithms, such as Floyd-Warshall or Dijkstra's algorithm, have been used as a reference. It is therefore important to review some key notions and terminologies from the theory of graphs and to show the theoretical and algorithmic aspects of the two shortest path algorithms.

In this chapter we also cover important notions of systems, simulation relations and composition of systems. Most importantly we review the algorithm that provides a solution to the "reachability game". The idea behind this algorithm is our key component in the non-deterministic set-destination shortest path algorithm. In addition to that and since our goal is to synthesize controllers with mixed qualitative-quantitative specifications, we provide a quick overview on formal languages and especially on the LTL language.

2-1 Directed graphs

Most of the definitions and concepts in this thesis require the notion of *directed graph.* Binary and algebraic decision diagrams are a form of directed graphs, while the shortest path algorithms used to address optimal control problems, operate on directed graphs.

Definition 2.1 (Directed graph [35])**.** A *directed graph* or *digraph* D is an ordered tuple (V, A) consisting of a non-empty finite set V of vertices and a finite set $A \subseteq V \times V$ of *ordered pairs* of distinct vertices called *arcs* or *directed edges.* We call V the vertex set and A the arc set of D.

The two sets V and A can also be denoted as $V(D)$ and $A(D)$ to emphasize that these are vertex and edge sets of a particular graph D. Vertices are sometimes called *points* or *nodes* and arcs are sometimes called *lines*. For an arc (v_1, v_2) the first vertex v_1 is its *tail* and the second vertex v_2 is its *head*. We also say that the arc (v_1, v_2) *leaves* v_1 and *enters* v_2 and that v_1 is adjacent to v_2. The arc (v_1, v_2) is also denoted by v_1v_2.

The cardinality of the vertex set of a directed graph D, $|V|$, is called the *order* of D and is commonly denoted by $n(D)$, or more simply by n when the graph under consideration is clear. The cardinality of the arc set of a graph D, $|A|$, on the other hand represents the *size* of the graph D and is often denoted as $m(D)$ or m. So, a (m, n) graph has size m and order n.

It is customary to define or describe a directed graph by means of a diagram in which each vertex is represented by a point and each edge is represented by an "arrow" line or curve joining adjacent vertices. But, a directed graph can also be described by means of matrices. One such matrix is the *adjacency matrix*.

Definition 2.2 (Adjacency matrix [35]). Let $D = (V, A)$ be a directed graph of order n, the *adjacency matrix* M of D is the $n \times n$ zero-one matrix where,

$$m_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \in A \\ 0 & \text{if } \{v_i, v_j\} \notin A \end{cases}$$

Thus, the adjacency matrix of a directed graph D is a symmetric $(0, 1)$ matrix having zero entries along the main diagonal. For a pair X, Y of vertex sets of a digraph D, we define

$$(X, Y)_D = \{xy \in A(D) : x \in X, y \in Y\}$$

i.e. $(X, Y)_D$ is the set of arcs with tail in X and head in Y. The above definition of a digraph implies that we allow a digraph to have arcs with the same end-vertices, but we do not allow it to contain parallel (also called multiple) arcs, that is, pairs of arcs with the same tail and the same head, or loops (i.e. arcs whose head and tail coincide).

Definition 2.3 (Directed pseudographs and multigraphs [36]). *Directed pseudographs* are directed graphs with parallel loops and arcs. Directed pseudographs without loops are called *directed multigraphs.*

Given a vertex v, the following definition expresses the connection of v compared with its adjacent vertices.

Definition 2.4 (Neighbourhood of a vertex [35]). Given a vertex v of a directed (pseudo)graph $D = (V, A)$ then,

- $N_D^+(v) = \{u \in V \setminus v : vu \in A\}$ is the *out-neighbourhood* of v
- $N_D^-(v) = \{w \in V \setminus v : wv \in A\}$ is the *in-neighbourhood* of v

The vertices in $N_D^+(v)$, $N_D^-(v)$ and $N_D(v)$ are called the *out-neighbors*, *in-neighbors* and *neighbors* of v respectively. The set $N_D(v) = N_D^+(v) \cup N_D^-(v)$ is called the neighbourhood of v.

Since it is possible to have multiple arcs in a directed pseudograph, it is also important to define the number of edges entering and leaving a vertex.

Definition 2.5 (Degree of a vertex [35])**.** Given a vertex v of a directed graph $D = (V, A)$ then,

- $Deg^{+}_{D}(v) = |(v, V)_D|$ is the *out-degree* of v
- $Deg^{-}_{D}(v) = |(V, v)_D|$ is the *in-degree* of v

Note that, a loop at a vertex contributes to both the in-degree and the out-degree. In case of directed multigraphs, the cardinality of $N^{-}_{D}(v)$ and $N^{+}_{D}(v)$ is the same as the in- and out-degree respectively.

When one needs to add some weight or cost in the link between two vertices as it is in the case of the shortest path algorithms, we are referring to *weighted directed pseudographs*[36].

Definition 2.6 (Weighted directed graph [35, 36])**.** A weighted directed graph is an ordered triple $D = (V, A, c)$, consisting of a non-empty finite set V of vertices, a finite set $A \subseteq V \times V$ of directed edges and a map $c : A \longrightarrow \mathbb{R}$ expressing the weight of the link. If we define the weight map as $c : V \longrightarrow \mathbb{R}$, then the graph is referred to as a *vertex-weighted directed graph.*

If a is an element (i.e. a vertex or an arc) of a weighted directed graph $D = (V, A, c)$, then $c(a)$ is called the weight or the cost of a. Similarly to the adjacency-matrix used to represent directed graphs, we might also use a matrix representation for weighted directed graphs.

Definition 2.7 (Cost adjacency matrix)**.** Let $D = (V, A, c)$ be a weighted directed graph of order n, the *cost adjacency matrix* M of D is the $n \times n$ matrix where,

$$m_{ij} = \begin{cases} c(\{v_i, v_j\}) & \text{if}\{v_i, v_j\} \in A \\ \infty & \text{if}\{v_i, v_j\} \notin A \end{cases}$$

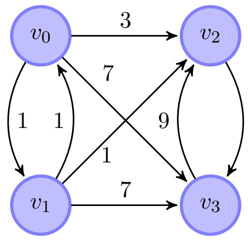
We can also use the cost adjacency matrix to describe vertex-weighted directed graphs. In this case the matrix is defined as follows:

$$m_{ij} = \begin{cases} c(v_j) & \text{if}\{v_i, v_j\} \in A \\ \infty & \text{if}\{v_i, v_j\} \notin A \end{cases}$$

This means that the cost of the arc (v_i, v_j), namely the link between two adjacent vertices, is defined by the head-vertex v_j as $c(v_j)$. We use ∞ to denote that there is no connection between v_i and v_j in the corresponding graph (Figure 2-1).

Before we can speak about shortest path algorithms, it is important to present the notions of *walk, trail* and *path* in a digraph D.

Definition 2.8 (Walk, trail and path in a digraph [36])**.** Let $D = (V, A)$ be a directed graph. A walk in D is an alternating sequence $W = v_1 a_1 v_2 a_2 v_3 ... v_{k-1} a_{k-1} v_k$ of vertices v_i and arcs a_j from D such that the tail of a_i is v_i and the head of a_i is v_{i+1} for every $i = 1, 2, ..., k-1$. We say that W is a walk from v_1 to v_k or an (v_1, v_k)-walk. A *trail* is a walk in which all arcs are distinct and is called a *path* if only the vertices are distinct too. For a trail W, if $v_k = v_1$, then W is a *cycle.*



(a)

$$M_D = \begin{bmatrix} \infty & 1 & 3 & 7 \\ 1 & \infty & 1 & 7 \\ \infty & \infty & \infty & 3 \\ \infty & \infty & 9 & \infty \end{bmatrix}$$

(b)

Figure 2-1: A weighted directed graph $D(V, A, c)$ (a) and its cost adjacency matrix M_D (b).

In this context, whenever we speak about the *length of a walk* W in a weighted (or vertex-weighted) directed pseudograph $D = (V, A, c)$, we mean the weight of that walk, with respect to c. That is, the sum of the weights of the arcs (or vertices) in W. A *negative cycle* in a weighted digraph D is a cycle W whose weight is negative. Note also that a vertex y is *reachable* from a vertex x if D has an (x, y)-walk.

At this point it is also important to review the concept of *connectivity*. This notion is of paramount importance when analyzing the complexity of the shortest path algorithms. Furthermore, it defines the complexity of the BDD data structure.

Definition 2.9 (Strongly connected digraph [36])**.** Let $D = (V, A)$ be a directed graph. The digraph D is *strongly connected* (or, just, *strong*) if, for every pair x, y of distinct vertices in D, there exists an (x, y)-walk and a (y, x)-walk. In other words, D is strong if every vertex of D is reachable from every other vertex of D.

We say that a digraph D is *complete* if, for every pair (x, y) of distinct vertices of D, both xy and yx are in D.

2-2 Dijkstra's algorithm

Dijkstra's algorithm [28] is one the most well known answer to the *single-source shortest path* problem on weighted directed graphs with no negative weights. To compute the shortest path, Dijkstra proposed a greedy algorithm[37], that relies on the property that a shortest path between two vertices contains other shortest paths within it. This optimal-substructure property is the result of the *principle of optimality* [38]. It states that *an optimal path has the property that whatever the initial conditions and control variables (choices) over some initial period, the control (or decision variables) chosen over the remaining period must be optimal for the remaining problem, with the node resulting from the early decisions taken to be the initial condition.*

This optimal-substructure property is a hallmark of the applicability of the greedy method. The following lemma states the optimal-substructure property of shortest paths more precisely.

Lemma 2.1 (Subpaths of shortest paths are shortest paths [39])**.** Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \longrightarrow \mathbb{R}$, let $p = \langle v_1, v_2, ..., v_k \rangle$ be a shortest path from vertex v_1 to vertex v_k and, for any i and j such that $1 \leq i \leq j \leq k$, let $p_{ij} =$

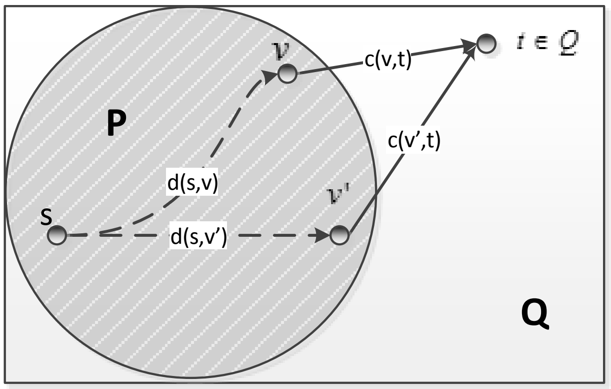


Figure 2-2: Dijkstra's Algorithm in terms of the *optimal-substructure property*.

$\langle v_i, v_{i+1}, ..., v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j . Then, p_{ij} is a shortest path from v_i to v_j.

If x and y are vertices of a weighted directed graph $D(V, A, c)$, then the distance from x to y in D, denoted $d(x, y)$, is the minimum length of a (x, y)-walk, if y is reachable from x, and otherwise $d(x, y) = \infty$. If D has no cycle of negative weight, it follows that $d(x, x) = 0$ for every vertex $v \in V$.

Having the principle of optimality in mind, the shortest path search in a weighted directed graph $D(V, A, c)$ with non-negative weights, from a source node s to some node t results in:

$$d(s,t) = \min_{v \in N_D^-(t)} \{d(s,v) + c(v,t)\} \tag{2-1}$$

In other words, any subpath of an optimal path is itself optimal (otherwise it could be replaced to yield a shorter path). The equation also suggests that the set V can be thought as the union of two distinct sets, P and Q. Let $P \subseteq V$ be the set of nodes where the shortest path has already been computed, i.e. where $d(s, v) \neq \infty$ for all $v \in P$, and $Q = V \setminus P$ the set of the rest of the nodes. For every $v \in Q$ the shortest distance of any path from s to v, that uses only nodes in P is being determined. This procedure is being applied recursively until t is reached (Figure 2-2). Initially, only the node s belongs to the set P, for which $d(s, s) = 0$ and an upper bound on $d(s, v)$ for each node $v \in Q$ is set to ∞.

In many problems, it is not only important to compute the shortest path, but also to keep track of it. For this reason we use the map $\widetilde{E} : V \longrightarrow A$. For each newly added node $t' \in P$, $\widetilde{E}(t') = (t', v)$, where $v \in P$ such that $d(s, t') = d(s, v) + c(v, t')$ and $v \in N_D^-(t')$. Thus, in order to find the shortest path from $s \in P$ to node $t' \in P$ we have to start from node t' and trace the path by going backwards. That is, we have to apply the map $\widetilde{E}$ for the node v, which will bring us one step closer to s. Continuing in the same way, we are able to reconstruct the shortest path from s to t'. The reason why the edges point backwardly, is because this way each node would have always one (or no) pointing edge and thus we know exactly which path to follow to achieve the shortest path. In the other way around, there might exist a node, which is considered as an intermediate node for several shortest paths and as a result we wouldn't know which way to follow from that state to achieve the shortest path towards the desired destination.

For the formal description of Dijkstra's Algorithm, we are using the technique of *relaxation* [39]. For each vertex $v \in V$, we maintain an attribute $\delta_s(v)$, which is an upper bound on the weight of a shortest path from source s to v and is called a *shortest-path estimate.* The process of *relaxing* an edge (u, v) (see Algorithm 1), consists of testing whether one can improve the shortest path to v (found so far) by going through u and, if so, updating $\delta_s(v)$ and $\widetilde{E}(v)$. Note that, in Dijkstra's Algorithm, each edge is relaxed exactly once and $\delta_s(v) = d(s, v)$ for each $v \in P$ when the algorithm terminates. A formal description of Dijkstra's Algorithm is listed in Algorithm 2 and an example is shown in Figure 2-3.

Algorithm 1 Dijkstra's Algorithm - Relax

Description: Relaxes the edge (v, u), i.e. updates δ_s and $\widetilde{E}$.
Input: The edge (v, u) to be relaxed.

```
1: function RELAX($(v, u)$)
2:     if ($\delta_s(v) > \delta_s(u) + c(u, v)$) then
3:         $\delta_s(v) = \delta_s(u) + c(u, v)$
4:         $\widetilde{E}(v) = (v, u)$
5:     end if
6: end function
```

Algorithm 2 Dijkstra's Algorithm

Input: A weighted digraph $D = (V, A, c)$, such that $c(a) \geq 0$ for every $a \in A$, and a vertex $s \in V$.
Output: The shortest distance $d(s, v) = \delta_s(v)$ for every $v \in V$ and the pointer map $\widetilde{E}$.

```
1: function DIJKSTRA($D$)
2:     $P = \{s\}$
3:     $Q = V \setminus s$
4:     $\widetilde{E}(s) = (s, s)$
5:     $\widetilde{E}(Q) = \emptyset$
6:     $\delta_s(s) = 0$
7:     $\delta_s(Q) = \infty$
8:
9:     while $Q \neq \emptyset$ do
10:        $u = \arg\min_{v \in Q}\{\delta(s, v)\}$
11:        $Q = Q \setminus u$
12:        $P = P \cup u$
13:        $\widetilde{E} = \widetilde{E} \cup (u, v)$
14:        for all ($v \in N_D^-(u)$) do
15:            $Relax(u, v, c)$
16:        end for
17:    end while
18: end function
```

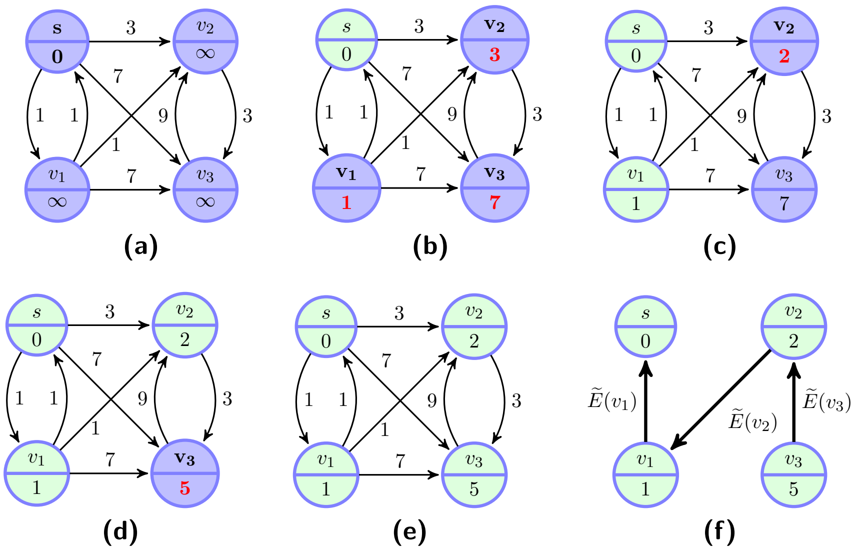


Figure 2-3: An example of Dijkstra's Algorithm in a weighed directed graph $D(V, A, c)$, where $s \in V$ is the source. The blue vertices are in Q; the light green vertices are in P. The number below each vertex is the current value of the parameter δ_s. (a) The initialization of the algorithm. (b)-(e) The status of the algorithm after each successive iteration. After the termination of the algorithm $d(s, v) = \delta_s(v)$. (f) The reversed (bold) edges show the set $\widetilde{E}$. Following these edges from some end point $v \in V$, we can backtrace the shortest path (s, v).

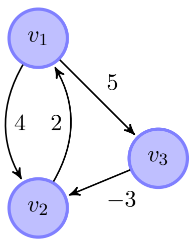
Although greedy methods do not always yield optimal solutions, they are quite powerful and work well for a wide range of problems [39]. In fact, the following theorem shows that Dijkstra's Algorithm - a greedy algorithm - does indeed produce an optimal solution.

Theorem 2.1 (Correctness of Dijkstra's algorithm [39])**.** *Dijkstra's algorithm, run on a weighted, directed graph $D = (V, A, c)$ with non-negative weight function c and source s, terminates with $d(s, u) = \delta_s(u)$ for all vertices $u \in V$.*

Dijkstra's algorithm is usually implemented using a priority queue. The priority queue is basically substituting the operation that returns the node with the minimum δ_s. Instead of finding the minimum δ_s value of all vertices in Q and then return the corresponding vertex, we can use a *min-priority queue*. Each time a δ_s value is being updated through the $Relax()$ function, it is added/updated in the priority queue as well. This way the first element in the queue, is the vertex with the smallest δ_s value. The running time of Dijkstra's algorithm depends on how the min-priority queue is implemented. Typically Dijkstra's algorithm runs in $O(|A| + |V| \log |V|)$, where V is the set of nodes and A the set of arcs.

2-3 Floyd-Warshall algorithm

The Floyd-Warshall Algorithm is a search algorithm that computes the shortest paths between all pairs of vertices of a weighed digraph with positive or negative weights, but no negative cycles. This algorithm is based on *dynamic programming* and was introduced by



$$W = D^0 = \begin{bmatrix} 0 & 4 & 5 \\ 2 & 0 & \infty \\ \infty & -3 & 0 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

(a)

$$D^1 = \begin{bmatrix} 0 & 4 & 5 \\ 2 & 0 & \mathbf{7} \\ \infty & -3 & 0 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 \end{bmatrix}$$

$$D^2 = \begin{bmatrix} 0 & 4 & 5 \\ 2 & 0 & 7 \\ \mathbf{-1} & -3 & 0 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ \mathbf{2} & 0 & 0 \end{bmatrix}$$

$$D^3 = \begin{bmatrix} 0 & \mathbf{2} & 5 \\ 2 & 0 & 7 \\ -1 & -3 & 0 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & \mathbf{3} & 0 \\ 0 & 0 & 1 \\ 2 & 0 & 0 \end{bmatrix}$$

(b)

Figure 2-4: Floyd-Warshall all-pairs shortest path example. (a) Shows the initial state of the algorithm. (b) Shows all steps of the algorithm. The values in bold are the ones that get updated in each step.

Floyd [29] and Warshall [40]. Dynamic programming is suitable for problems having optimal substructure and overlapping sub-problems. It solves problems by combining the solutions to sub-problems. This approach is also a result of the *principle of optimality*. Compared to the greedy method, where the problem is treated top-down, hoping that a locally optimal choice will lead to a globally optimal solution, dynamic programming formulates the problem in a bottom-up fashion and is typically applied to optimization problems.

Given a weighted digraph $D(V, A, c)$, where $V = \{v_1, v_2, ..., v_n\}$ and n being the number of nodes, we denote D^k_{ij}, with $0 \leq k \leq n$, the length of a shortest (i, j)-path in D, that uses vertices that are only in the set $\{v_1, v_2, ..., v_k\}$ as indeterminate points. It is obvious that D^0_{ij} represents only the edge weights of the digraph (cost adjacency matrix). According to the principle of optimality it holds that:

$$D^k_{ij} = \min\{D^{(k-1)}_{ij}, D^{(k-1)}_{ik} + D^{(k-1)}_{kj}\}. \tag{2-2}$$

Observe that a shortest (i, j)-path in D either does not include the vertex v_k, in which case $D^k_{ij} = D^{(k-1)}_{ij}$, or does include it, in which case $D^k_{ij} = D^{(k-1)}_{ik} + D^{(k-1)}_{kj}$. Note also that $D^k_{ii} = 0$ for all $k = 1, 2, .., n$. Furthermore,

$$\begin{aligned} D^k_{ik} &= \min\{D^{(k-1)}_{ik}, D^{(k-1)}_{ik} + D^{(k-1)}_{kk}\} \\ &= \min\{D^{(k-1)}_{ik}, D^{(k-1)}_{ik} + 0\} \\ &= D^{(k-1)}_{ik} \end{aligned}$$

Algorithm 3 Floyd-Warshall's Algorithm

Description: This is an improved version based on the observations $D_{ik}^{k} = D_{ik}^{(k-1)}$, $D_{kj}^{k} = D_{kj}^{(k-1)}$ and therefore it uses a single matrix D for its computations.
Input: The cost adjacency matrix W representing the weights of a digraph $D(V, A, c)$. The number of nodes is n.
Output: Matrix D containing all-pair shortest distances and the pointer array P.

```
1:  function FLOYD-WARSHALL($W, n$)
2:      $D^0 \leftarrow W$
3:      $P \leftarrow 0$
4:      for ($k = 1$ to $n$) do
5:          for ($i = 1$ to $n$) do
6:              for ($j = 1$ to $n$) do
7:                  if ($D[i,j] > D[i,k] + D[k,j]$) then
8:                      $D[i,j] = D[i,k] + D[k,i]$
9:                      $P[i,j] = k$
10:                 end if
11:             end for
12:         end for
13:     end for
14: end function
```

which means that a path from i to k will not become shorter by adding k to the allowed subset of intermediate vertices and

$$\begin{aligned} D_{kj}^{k} &= \min\{D_{kj}^{(k-1)}, D_{kk}^{(k-1)} + D_{kj}^{(k-1)}\} \\ &= \min\{D_{kj}^{(k-1)}, 0 + D_{kj}^{(k-1)}\} \\ &= D_{kj}^{(k-1)} \end{aligned}$$

As in the single source shortest path problem, we are also interested in being able to reconstruct all existing shortest paths. For this reason, we introduce the $n \times n$ *pointer matrix* P, which is used to trace back the shortest path in a similar manner as in the Dijkstra's algorithm.

Definition 2.10 (Pointer matrix)**.** Let $D = (V, A, c)$ be a weighted directed graph of order n, for which we want to compute the all-pairs shortest paths using Algorithm 3. To trace back a (i, j)-path we use the $n \times n$ matrix P where,

$$p_{ij} = \begin{cases} 0 & \text{if } i \text{ and } j \text{ are adjacent} \vee \text{if the } (i,j)\text{-path does not exists} \\ k \in \mathbb{R}^{+} & \text{if } v_k \text{ is adjacent to } j \text{ and indermidiate to the } (i,j)\text{-path} \end{cases}$$

At first, the pointer matrix is initialized to zero. The procedure to find the shortest path between node i and j is to look at the matrix element (i, j), which points to the intermediate node m_1 that the path has to contain. The new path now becomes $((i, m_1), j)$. If the path

(i, m_1) has no intermediate node, i.e. if the matrix element (i, m_1) is zero, then the path is (i, m_1, j). Otherwise the above procedure is applied recursively until there is only a direct link between m_n and i, where n the number of times the above procedure has been applied, which corresponds to the number of intermediate nodes between i and j.

Having the above observations in mind, Algorithm 3 illustrates the Floyd-Wharshall all-pair shortest path algorithm using matrices as a date structure and Figure 2-4 an example of it. Note that, although it is possible to have more than one path from some node $v \in V$ to some node $v' \in V$, the pointer array is only capable of keeping the last updated value. Thus, there might be a case where we discard actual shortest paths.

The running time of the Floyd-Warshall algorithm is determined by the triply nested "for" loops. Because each execution of the "if" statement takes constant time, the algorithm has takes a total time of $O(n^3)$, where n is the number of vertices.

2-4 Algebraic decision diagrams

The vast increase of complexity of digital functions and systems, has made the use of the Boolean Algebra in form of boolean equations, Karnaugh maps, etc., not efficient. Many problems in digital logic design, testing and verification are NP-complete or co-NP-complete. Consequently, all known approaches to performing these operations require an amount of computer time that grows rapidly as the size of the problem increases. This makes it difficult to compare the relative efficiencies of different approaches to representing and manipulating Boolean functions. A variety of methods have been developed for representing and manipulating Boolean functions, but either their representation or their manipulation suffered from the unpleasant property of rapid growth with the number of variables involved. A new approach to these problems had to be explored.

A data structure that gives a solution to the above problems is a special form of digraphs and is called Binary Decision Diagram (BDD). The notion of BDDs was first introduced by Lee [41] and further popularized by Akers [42]. BDDs provide a more efficient method for representing and manipulating Boolean functions than binary decision trees and truth tables do. They represent Boolean functions as a rooted directed acyclic graph. Below is a formal definition of BDDs.

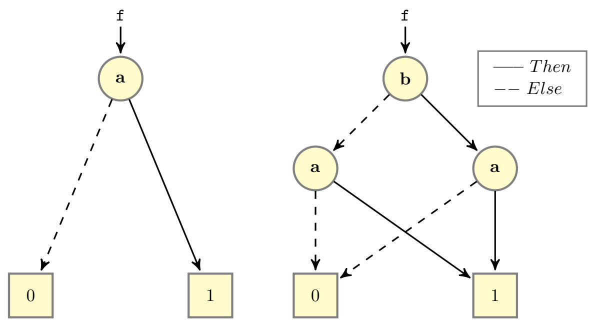
Definition 2.11 (Binary Decision Diagram [42])**.** A Binary Decision Diagram (BDD) is a rooted directed acyclic graph $(V \cup F \cup T, E)$ representing a set of functions $f_i : \{0,1\}^n \rightarrow \{0,1\}$, where:

- F is the set of the function nodes, for which $Deg^-(f) = 0$ and $Deg^+(f) = 1 \; \forall f \in F$.
- V is the set of internal nodes, for which $Deg^-(v) \geq 1$ and $Deg^+(v) = 2 \; \forall v \in V$.
- T is the set of terminal nodes, for which $Deg^-(t) \geq 1$ and $Deg^+(t) = 0 \; \forall t \in T$.
- E is the set of edges connecting the nodes of the graph.

Each (f, v) edge is called *incoming* edge. The two outgoing arcs of a node v are labeled *then*, if v is one and *else*, if v is zero.

a	b	f
0	0	0
0	1	0
1	0	1
1	1	1

(a)



(b) **(c)**

Figure 2-5: A ROBDD representation of a Boolean function f. (a) The truth table of the boolean function f. (b) The ROBDD representation using $a < b$ as a variable ordering. (c) Shows that ROBDDs are sensitive to variable ordering. Reordered as $b < a$.

In general, when we speak of BDDs we are actually referring to *Reduced Ordered Binary Decision Diagram (ROBDD)* [30]. ROBDDs are a special form of BDDs, where all variables are ordered on previously known ordering and every path visits variables in an ascending order. Furthermore, any two nodes of the BDD differ from each other. ROBDDs ensure the canonical[1] representation of a Boolean function and with this property the equality of functions can be checked without difficulty. Time and space complexity of ROBDDs depend on variable ordering. A simple reordering of variables alone may have a great impact on the size of the diagram. Determining the optimal variable ordering is unfortunately a NP problem. Figure 2-5 shows a BDDs representation of a boolean function and illustrates how variable reordering affects the BDDs.

BDDs take only terminal values of 0 and 1. Nevertheless, one can expand BDDs to allow them to have arbitrary integer terminals and more than two terminal nodes. These kind of BDDs are referred to as *Multi-Terminal Binary Decision Diagram (MTBDD)* [43]. The MTBDDs allow the implementation of symbolic algorithms, which are applicable not only to arithmetic, but also to many algebraic structures. Because of their applicability to different algebras and their foundation in large Boolean algebra, BDDs with multi-terminals are called *Algebraic Decision Diagrams (ADD)* [44, 45].

Definition 2.12 (Algebraic Decision Diagram [44])**.** An Algebraic Decision Diagram ADD is a rooted directed acyclic graph $(V \cup \Phi \cup T, E)$ representing a set of functions $f_i : \{0,1\}^n \to S$, where:

- Φ is the set of the function nodes, for which $Deg^{-}(\phi) = 0$ and $Deg^{+}(\phi) = 1\ \forall \phi \in \Phi$.
- V is the set of internal nodes, for which $Deg^{-}(v) \geq 1$ and $Deg^{+}(v) = 2\ \forall v \in V$.
- T is the set of terminal nodes, for which $Deg^{-}(t) \geq 1$ and $Deg^{+}(t) = 0\ \forall t \in T$.
- E is the set of edges connecting the nodes of the graph.
- S is the finite carrier of the algebraic structure over which the ADD is defined.

[1]BDDs are called canonical when they are unique for a representation of a boolean function given a variable ordering.

Each (f, v) edge is called *incoming*. The two outgoing arcs of a node v are labeled *then*, if v is one and *else*, if v is zero.

The function nodes are in one-to-one correspondence with the f_i's. Every node $v \in V$ has label $l(v) \in \{0, ..., n-1\}$, while each terminal node t is labeled with an element of S, $s(t)$. The label of nodes in E, identifies a variable on which the f_i's depend. The variables of the ADD are ordered, which means if v_j is a descendant of v_i, i.e. $v_i v_j \in E$, then $l(v_i) < l(v_j)$. An ADD represents a set of boolean functions, one for each function node, defined as follows [44]:

1. The Boolean function of a terminal node, t, is the constant function $s(t)$. The constant $s(t)$ is interpreted as an element of a boolean algebra larger than or equal in size to S.
2. The function of a node $v \in V$ is given by $l(v) \cdot f_{then} + l(v)' \cdot f_{else}$, where '$\cdot$' and '$+$' denote boolean conjunction and disjunction, and f_{then} and f_{else} are the functions of the *then* and *else* children.
3. The function of $\phi \in \Phi$ is the function of its only child.

2-4-1 ADD for graph and matrix representations

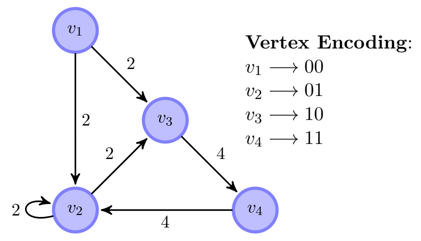
ADDs are a natural symbolic representation of weighted directed graphs, which are in one-to-one correspondence with square matrices. Suppose we have a weighted digraph G with N vertices. Firstly we construct its adjacency matrix M_G by encoding the nodes of the digraph. The ADD representation $A_G(x, y)$ will have $2n$ encoding variables, where $n = |x| = |y| = \lceil log_2(N) \rceil$; $x \in \{x_0, ..., x_n\}$ is a row variable and $y \in \{y_0, ..., y_n\}$ is a column variable. In the adjacency matrix M_G we think of the zero value as a *background*, denoting no entry, or no connection in the corresponding graph. Of course, in other applications, different backgrounds can be used for an efficient ADD representation. An example of an ADD representation of a weighted digraph can be seen in Figure 2-6. We can see that, when the number of vertices of the weighted graph G increases linearly, then the number of (ADD) nodes, i.e. the number of encoding variables in the ADD representation $A_G(x, y)$, grows exponentially.

ADDs can also represent bipartite[2] graphs which are equivalent to rectangular matrices. However, it is important that the ADD matrix is a square one. If this is not the case, then the ADD computations cannot be employed [44]. Therefore, in the case of a rectangular matrix, the matrix must be "padded" with appropriately valued dummy rows and columns, to convert the number of rows and columns in order to convert the matrix to a square one.

The ADD matrix representation can now been seen as a special form of a Boolean function $f(x)$ of n variables, that is, $f(x) : \{0, 1\}^n \to S$, where S the carrier of a Boolean algebra.

As a consequence, all theorems of Boolean algebra can be applied to ADDs. The most important one, is Boole's (or Shannon's) expansion theorem:

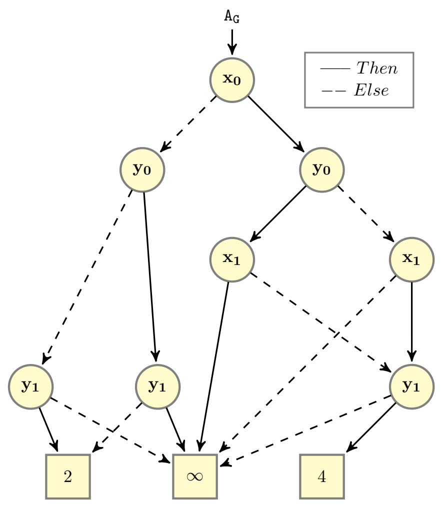
[2]A bipartite graph, also called a bigraph, is a set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent.



(a)

$$M_D = \begin{array}{c} y_0y_1 \\ \begin{array}{cccc} 00 & 01 & 10 & 11 \end{array} \\ \begin{bmatrix} \infty & 2 & 2 & \infty \\ \infty & 2 & 2 & \infty \\ \infty & \infty & \infty & 4 \\ \infty & 4 & \infty & \infty \end{bmatrix} \end{array} \begin{array}{c} \\ \\ 00 \\ 01 \\ 10 \\ 11 \end{array} x_0x_1$$

(b)



(c)

Figure 2-6: ADD representation of a weighted digraph. (a) The directed graph and the vertex encoding. (b) The cost-adjacency matrix. The columns are encoded with the y_0y_1 variables and the rows with the x_0x_1 variables. (c) The resulting ADD that represents the directed graph. The variable ordering is $x_0 < y_0 < x_1 < y_1$.

Theorem 2.2 (Shannon's expansion theorem [44])**.** If $f(x) : \{0,1\}^n \to \{0,1\}$ is a Boolean function, then for all $(x_1, ..., x_n) \in \{0,1\}^n$:

$$f(x_1, x_2, ..., x_n) = x_1' \cdot f(0, x_2, ..., x_n) + x_1 \cdot f(1, x_2, ..., x_n)$$

The function resulting when some argument x_i of function f is replaced by a constant $b \in \{0,1\}$ is called *restriction* of f and is denoted $f|_{x_i=b}$. This notion is also termed as the *cofactor* of f, with $f|_{x_i=0} = f|_{\overline{x}}$ the *negative* cofactor and $f|_{x_i=1} = f|_x$ the *positive* cofactor.

If the Shannon expansion is carried out recursively, it gives a full binary tree with leafs of value 0 and 1. Each internal node represents a function, where its left child is its cofactor with respect to $\overline{x_i}$ for some variable x_i, and its right child is its cofactor with respect to x_i. This tree is called a *Shannon Tree* of f [43] and is represented by the well known *minterm canonical form*[46]:

$$\begin{aligned} f(x_1, ..., x_{n-1}, x_n) = & f(0, ..., 0, 0)x_1' \cdots x_{n-1}'x_n' + \cdots \\ & + f(1, ..., 1, 1)x_1 \cdots x_{n-1}x_n \end{aligned}$$

The values $f(0, ..., 0, 0), ..., f(1, ..., 1, 1)$ are elements of S and they are called the *discriminants* of the function f; the elementary products $x_1' \cdots x_{n-1}'x_n', x_1' \cdots x_{n-1}'x_n, ..., x_1 \cdots x_{n-1}x_n$ are called the *minterms*.

We can see the minterms as a set of arguments describing a path in the graph, starting from the root and the discriminant as the value of the leaf at the end of the path. Therefore, the *Shannon Tree* is basically describing each possible path in the graph from the root. Nevertheless, ADDs have an interesting property, namely *canonicity*. In this regard, minterms with the same discriminant are grouped together. In a graph this can be seen as eliminating redundant vertices and duplicate subgraphs. Therefore, the ADD can be reduced in size without changing the denoted function so that each boolean function corresponds to a unique ADD.

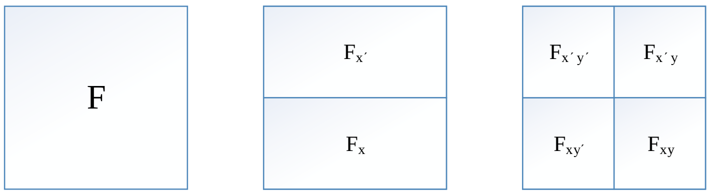


Figure 2-7: Matrix partitioning by cofactoring.

It is now obvious that the cofactors of ADDs play an important role. In figure 2-7 we see an example of a 4×4 matrix cofactoring. We begin to partition the matrix, by cofactoring with respect to the arbitrary selected top variable x_0. Although the order of the variable selection is not predefined, special emphasis is given to top variable cofactoring, since this can be done in $O(n)$ time. Cofactoring with respect to x_0, partitions the matrix in two rectangular sub-matrices. The first (upper) is represented by the else-child $f_{else} = f_{x'_0}$ and the second by $f_{then} = f_{x_0}$. We continue by selecting the next variable y_0 and we end up, in the same way, with 4 square sub-matrices represented by $f_{x'y'}$, $f_{x'y}$, $f_{xy'}$ and f_{xy} respectively. At the end of this recursive "descent", all row and column variables will have been cofactored, and we will finish having a partition with a set of 1×1 matrices. These 1×1 matrices are basically representing the constant terminal nodes, i.e. the leafs of the ADD.

Note that during this recursive procedure, some of the ADD function nodes f do not need to be distinct. In fact recombination, as explained earlier, of identical sub-graphs or nodes, can lead to great efficiencies. Note also that there are just $n = \log_2 N$ row and n column variables, so access to any of the non-zero elements can be attained in $O(n)$ operations.

2-5 Discrete abstractions

Although there are many mathematical models that describe a dynamical phenomenon, we need a model that cuts off the details, while preserving the essential characteristics. Abstraction provides a mean to represent the dynamics of a system and allows for rigorous analysis. Model abstraction is furthermore equipped with relationships explaining how different systems can be related. Different systems might also be combined to create new ones, as in the case of a plant with a controller. This section will review the notions of systems, simulation relations, composition of systems, but most importantly the notions of reachability and the corresponding fixed-point algorithm as an answer to the "reachability games".

Definition 2.13 (System [1])**.** A system S is a sextuple $(X, X_0, U, \longrightarrow, Y, H)$ consisting of:

- a set of states X;
- a set of initial states $X_0 \subseteq X$;

- a set of inputs U;
- a transition relation $\longrightarrow \subseteq X \times U \times X$;
- a set of outputs Y;
- an output map $H : X \to Y$.

A system is called finite-state if X is a finite set. The notation $x \xrightarrow{u} x'$ is used to describe the transition $(x, u, x') \in \longrightarrow$, which captures the evolution of a system. The state x' is called the *u-successor*, or simply *successor*, of state x. On the other hand, x is called the *u-predecessor*, or *predecessor*. The set of *u-successors* is denoted by $Post_u(x)$. Note that, since $\emptyset$, the empty set, is a subset of X, there maybe no *u-successors*. Therefore, we denote $U(x)$ as the set of inputs $u \in U$, for which $Post_u(x) \notin \emptyset$. The system is called *blocking* in case there exists a state $x \in X$ such that $U(x) = \emptyset$. This means that there are no further possible transitions from that state. If $U(x) \neq \emptyset$ then system is called *non-blocking*. Of course there might be the case when a state $x \in X$ has more than one *u-successors*. Such a system is called *non-deterministic*. Similarly, in a *deterministic* system a state $x \in X$ has at most one *u-successor*, which means that for any state $x \in X$ and any input $u \in U$, $x \xrightarrow{u} x'$ and $x \xrightarrow{u} x''$ imply $x' = x''$. If the function $H|_{X_0}$[3] is *one-to-one* and for any state $x \in X$ and any inputs $u, u' \in U$, $x \xrightarrow{u} x'$ and $x \xrightarrow{u'} x''$ with $H(x') = H(x'')$ imply $x' = x''$, then the system is *output deterministic*.

If we want to explicitly refer to the possible sequences of states and outputs that a system can generate, we are referring to the so called *system behaviors*.

Definition 2.14 (Internal behavior [1]). For a system S and given any state $x \in X$, a *finite* internal behavior generated from x is a finite sequence of transitions:

$$x_0 \xrightarrow{u_0} x_1 \xrightarrow{u_1} x_2 \xrightarrow{u_2} ... \xrightarrow{u_{n-2}} x_{n-1} \xrightarrow{u_{n-1}} x_n$$

such that $x_0 = x$ and $x_i \xrightarrow{u_i} x_{i+1}$ for all $0 \leq i < n$. A finite internal behavior generated from x is *initialized* if $x \in X_0$.

An *infinite* behavior generated from x is an infinite sequence of transitions:

$$x_0 \xrightarrow{u_0} x_1 \xrightarrow{u_1} x_2 \xrightarrow{u_2} x_3 \xrightarrow{u_3} ...$$

that satisfies $x_0 = x$ and $x_i \xrightarrow{u_i} x_{i+1}$ for all $i \in \mathbb{N}$. An infinite internal behavior generated from x is *initialized* if $x \in X_0$.

The sequence of outputs that are caused internally are called *external behaviors* and are defined as follows.

Definition 2.15 (External behavior [1]). For a system S given a state x, every finite internal behavior

$$x_0 \xrightarrow{u_0} x_1 \xrightarrow{u_1} x_2 \xrightarrow{u_2} ... \xrightarrow{u_{n-2}} x_{n-1} \xrightarrow{u_{n-1}} x_n$$

[3] $H|_{X_0}$ is the function H, but having its domain restricted, meaning that $dom(H|_{X_0}) \subseteq dom(H)$.

defines a *finite* external behavior through the map H:

$$y_0 \longrightarrow y_1 \longrightarrow y_2 \longrightarrow \dots \longrightarrow y_{n-1} \longrightarrow y_n$$

with $H(x_i) = y_i \in Y$ for all $0 \leq i \leq n$. Similarly, every infinite internal behavior defines an *infinite* external behavior:

$$y_0 \longrightarrow y_1 \longrightarrow y_2 \longrightarrow \dots$$

with $H(x_i) = y_i \in Y$ for all $i \in \mathbb{N}$. The external behavior is *initialized* if the corresponding internal behavior is initialized.

The set of external behaviors that are defined by internal behaviors generated from state x is denoted by $\mathcal{B}_{\mathbf{x}}(\mathbf{S})$ and is called the *external behavior* from state x. If a behavior y is not contained as a prefix in any other behavior from the system, then the behavior y is called *maximal.*

Definition 2.16 (Finite External Behavior)**.** The finite external behavior generated by a system S, denoted by $\mathcal{B}(S)$, is defined by:

$$\mathcal{B}(S) = \bigcup_{x \in X_0} \mathcal{B}_x(S).$$

The external behavior for an *output deterministic* system, is defined only by one corresponding internal behavior.

Definition 2.17 (Infinite External Behavior)**.** The infinite external behavior generated by a system S, denoted by $\mathcal{B}^{\omega}(S)$, is defined by:

$$\mathcal{B}^{\omega}(S) = \bigcup_{x \in X_0} \mathcal{B}^{\omega}_x(S).$$

If a system S is *non-blocking*, then $B^{\omega}_x(S) \notin \emptyset$. However, $B^{\omega}_x(S)$ may be nonempty even if S is a blocking system.

Until now we have reviewed the definition of a system and some notions that allows us to have an insight on the characteristics of a system. Another important characteristic that we are going to review is how a system relates to another one. Bisimulation relation is a powerful mathematical framework for addressing the equivalence of systems. More precisely, we are interested in approximate bisimulation relations, as we are able to address a broader class of systems for which discrete abstraction can be computed.

Definition 2.18 (Approximate Simulation Relation [1])**.** Consider two metric systems S_a and S_b with $Y_a = Y_b$ and let $\varepsilon \in \mathbb{R}^+_0$. A relation $R \subseteq X_a \times X_b$ is an *ε-approximate simulation relation* from S_a to S_b if the following three conditions are satisfied:

1. $\forall x_{a0} \in X_{a0}\ \exists x_{b0} \in X_{b0}$ with $(x_{a0}, x_{b0}) \in R$;
2. $\forall (x_a, x_b) \in R : d(H_a(x_a), H_b(x_b)) \leq \varepsilon$;

3. $\forall (x_a, x_b) \in R : x_a \xrightarrow[a]{u_a} x'_a$ in S_a implies the existence of $x_b \xrightarrow[b]{u_b} x'_b$ in S_b satisfying $(x'_a, x'_b) \in R$.

We say that system S_a is ε-approximately simulated by system S_b or that system S_b ε-approximately simulates system S_a, denoted by $S_a \preceq_S^{\varepsilon} S_b$, if there exists an ε-approximate simulation relation from S_a to S_b.

If $\varepsilon = 0$ in definition 2.18, we say that the relation is an exact relation. Exact or approximate simulation relations describe whether two systems can be considered as equivalent or not in terms of their external behaviors. Nevertheless, when it comes to control systems, we are also interested in similarity relationships that capture the effect that different choices of inputs have on transitions. Such a relation is called *approximate alternating simulation relation.*

Definition 2.19 (Approximate Alternating Simulation Relation [1])**.** Let S_a and S_b be two metric systems with $Y_a = Y_b$ normed vector spaces, and let $\varepsilon \in \mathbb{R}_0^+$. A relation $R \subseteq X_a \times X_b$ is an *ε-approximate alternating simulation relation* from S_a to S_b if the following three conditions are satisfied:

1. $\forall x_{a0} \in X_{a0}\ \exists x_{b0} \in X_{b0}$ with $(x_{a0}, x_{b0}) \in R$;
2. $\forall (x_a, x_b) \in R : d(H_a(x_a), H_b(x_b)) \leq \varepsilon$;
3. $\forall (x_a, x_b) \in R : \forall u_a \in U_a(x_a)\ \exists u_b \in U_b(x_b)$ such that $\forall x'_b \in \mathrm{Post}_{u_b}(x_b)\ \exists x'_a \in \mathrm{Post}_{u_a}(x_a)$ satisfying $(x'_a, x'_b) \in R$.

We say that system S_a is ε-approximately alternatingly simulated by system S_b or that system S_b ε-approximately alternatingly simulates system S_a, denoted by $S_a \preceq_{AS}^{\varepsilon} S_b$, if there exists an ε-approximate alternating simulation relation from S_a to S_b.

As in the case of approximate simulation relation, if $\varepsilon = 0$ in definition 2.19, then the relation is exact. Note that the notion of alternating simulation coincides with simulation in the case of deterministic systems. Also note that every non-deterministic system S_a and its associated deterministic system $S_{d(a)}$ satisfy $S_a \preceq_{AS}^{0} S_{d(a)}$.

A simulation relation that does not only relate states but also inputs, is the *extended alternating simulation relation.* This simulation relation is of paramount importance when addressing problems of control. In these problems the controller S_c is interconnected with the system S_a to be controlled, such that the desired specifications S_b can be met. To render this interconnection possible, certain synchronization constraints are needed and the extended alternating simulation relation is used to describe them. If the interconnection between S_c and S_a is feasible, then we say that S_c is feedback composable with S_a.

Definition 2.20. (Extended Alternating Simulation Relation [1]). Let R be an alternating simulation relation from system S_a to system S_b. The *extended alternating simulation relation* $R^e \subseteq X_a \times X_b \times U_a \times U_b$ associated with R is defined by all the quadruples $(x_a, x_b, u_a, u_b) \in X_a \times X_b \times U_a \times U_b$ satisfying:

1. $(x_a, x_b) \in R$;

2. $u_a \in U_a(x_a)$;
3. $u_b \in U_b(x_b)$, and $\forall x'_b \in \text{Post}_{u_b}(x_b)\ \exists x'_a \in \text{Post}_{u_a}(x_a)$ satisfying $(x'_a, x'_b) \in R$.

The notion of *approximate feedback composition* is now formalized as follows:

Definition 2.21. (Approximate Feedback Composition [1]). Let S_a and S_c be two metric systems with the same output set $Y_a = Y_c$ and let R be an ε-approximate alternating simulation relation from S_c to S_a. The *approximate feedback composition* of S_c and S_a with interconnection relation $F = R^{\varepsilon}$, denoted by $S_c \times^{\varepsilon}_{\mathcal{F}} S_a$, is the system $(X_{\mathcal{F}}, X_{\mathcal{F}0}, U_{\mathcal{F}}, \underset{\mathcal{F}}{\longrightarrow}, Y_{\mathcal{F}}, H_{\mathcal{F}})$ consisting of:

- $X_{\mathcal{F}} = \pi_X(\mathcal{F}) = R$;
- $X_{\mathcal{F}0} = X_{\mathcal{F}} \cap (X_{c0} \times X_{a0})$;
- $U_{\mathcal{F}} = U_c \times U_a$;
- $(x_c, x_a) \xrightarrow[\mathcal{F}]{(u_c,u_a)} (x'_c, x'_a)$ if:
    1. $(x_c, u_c, x'_c) \in \underset{c}{\longrightarrow}$;
    2. $(x_a, u_a, x'_a) \in \underset{a}{\longrightarrow}$;
    3. $(x_c, x_a, u_c, u_a) \in \mathcal{F}$;
- $Y_{\mathcal{F}} = Y_c = Y_a$;
- $H_{\mathcal{F}}(x_c, x_a) = \frac{1}{2}(H_c(x_c) + H_a(x_a))$.

In the following section we will review how to solve control problems with liveness constraints. The fixed point algorithm used to solve the so called reachability games is also the source of inspiration for solving shortest path problems in non-deterministic systems.

2-6 Reachability games

Some controller designs are forced to satisfy certain specifications, in order to achieve a desirable behavior of the system. A common specification needed in applications, is the one that requires the trajectories of the controlled system to enter some target set W, which corresponds to some target set W of outputs, in finite time. This *reachability* control problem can been seen as a game, in which the controller S_c must eliminate all states (and only these states) that do not guarantee that the controlled system will eventually enter the target set W.

Definition 2.22 (Reachability games [1]). Let S_a be a system satisfying $Y_a = X_a$ and $H_a = 1_{X_a}$, and let $W \subseteq X_a$ be a set of states. The reachability game for system S_a and specification set W asks for the existence of a controller S_c such that:

- S_c is feedback composable with S_a;

- for every maximal behavior $y \in \mathcal{B}(S_c \times_{\mathcal{F}} S_a) \cup \mathcal{B}^{\omega}(S_c \times_{\mathcal{F}} S_a)$ there exists $k \in \mathbb{N}_0$ such that $y(k) = y_k \in W$.

A reachability game is considered to be solvable when S_c exists.

The above definition states that the only goal of the composed system is to simply reach the desired set W. That is, in finitely many steps in case of a any finite behavior $y_0 y_1 ...$, and in case of any finite behavior $y_0 y_1 ... y_k$ to visit W before or when reaching a blocking state.

To solve the reachability problem the following *fixed-point* operator is used [1]:

$$G_W : 2^{X_a} \longrightarrow 2^{X_a}$$

for any specification set $W \subseteq X_a$. A fixed-point of a function $f : X \longrightarrow X$ is an element $x \in X$ satisfying $f(x) = x$. The G_W operator is defined by:

$$G_W(Z) = \{x_a \in X_a \mid x_a \in W \text{or} \exists u_a \in U_a(x_a), \emptyset \neq Post_{u_a}(x_a) \subseteq Z\}. \tag{2-3}$$

The fixed-points of the above operator form a minimal set of states $x_a \in Z$, which ensure that the controlled system will reach the desirable state W, if $Z \cap X_{a0} \neq \emptyset$. Note that the inclusion $Z \subseteq Z'$ implies $G_W(Z) \subseteq G_W(Z')$ for any $W \subseteq X_a$ and thus guaranteeing the existence of a unique minimal fixed-point of G_W.

Among the several possible solutions, the following *maximally permissive* controller is considered [1]:

$$S_c = (X_c, X_{c0}, U_a, \underset{c}{\longrightarrow}) \tag{2-4}$$

defined as:

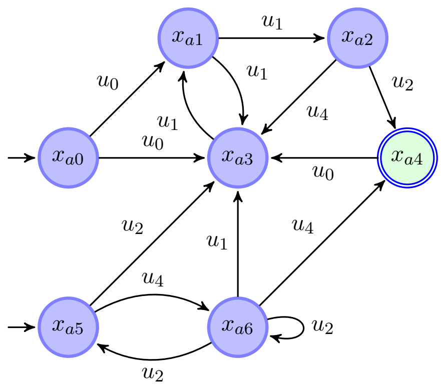
- $X_c = Z$;
- $X_{c0} Z \cap X_{a0}$;
- $x_c \underset{c}{\xrightarrow{u_a}} x_c'$ if there exists a $k \in \mathbb{N}$ such that $x_c \notin G_W^k(\emptyset)$ and $\emptyset \neq Post_{u_a}(x_c) \subseteq G_W^k(\emptyset)\}$,

and where $Post_{u_a}(x_c)$ refers to the u_a-successors in S_a. Moreover, one can easily verify that the relation defined by all the pairs $(x_c, x_a) \in X_c \times X_a$ with $x_c = x_a$ is an alternating simulation relation from S_c to S_a. The solution of reachability games can be fully characterized in terms of the fixed-points of G_W.

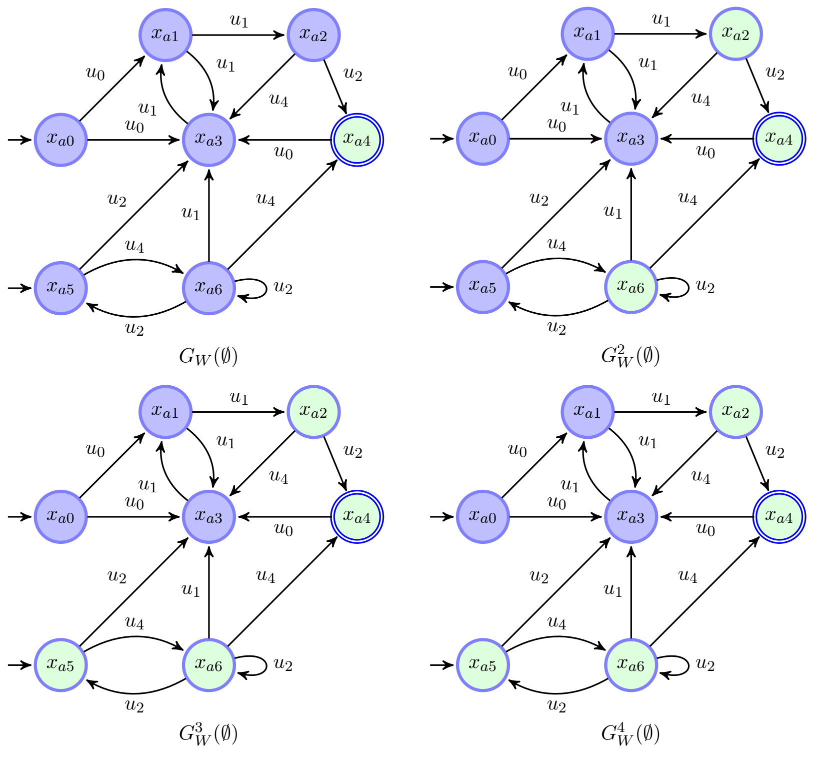
Theorem 2.3 ([1]). *Let S_a be a system with $Y_a = X_a$ and $X_a = 1_{X_a}$, and let $W \subseteq X_a$ be a set of states. The reachability game for S_a and specification set W is solvable iff the minimal fixed-point Z of the operator G_W satisfies $Z \cap X_{a0} \neq \emptyset$. Moreover, Z can be obtained as[1]:*

$$Z = \lim_{i \to \infty} G_W^i(\emptyset)$$

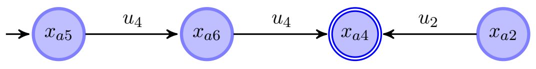
When $Z \cap X_{a0} \neq \emptyset$, a solution to the reachability game is given by the controller (2-4)*.*



(a)



(b)



(c)

Figure 2-8: The figure illustrates an example of a reachability game for system S_a in (a). The specification set is $W = \{x_{a_4}\}$ and $\{x_{a0}, x_{a5}\} \in X_0$. The stages of computing the fixed-point set Z using the operator G_W are illustrated in (b). The resulting controller S_c is depicted in (c). Source [1].

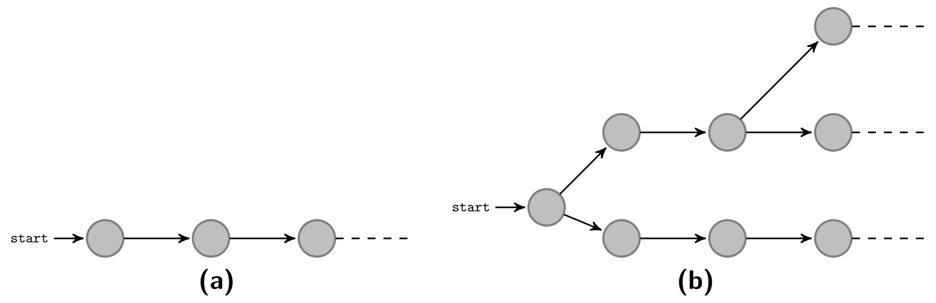


Figure 2-9: Typical models of time of temporal logics. (a) Linear structure. (b) Branching structure.

The computation of the fixed-point Z begins with the empty set and by applying the operator $G^i_W(\emptyset)$ for $i = 1$. In this first step, all the states that belong to the *Reach*-set are being collected, and thus $Z = G^1_W(\emptyset) = W$. The process then works backwards from the specification set, by adding states $x_a \in Xa$ to Z such that $\exists u_a \in U_a(x_a)$ where $\emptyset \neq Post_{u_a}(x_a) \subseteq Z$, until all states have been covered. Of course if $Z \cap X_{a0} = \emptyset$ for $i \to \infty$, the reachability game is not solvable. Figure 2-8 illustrates how the image of G_W is constructed.

Theorem 2.3 can be generalized to the case where the initial states of S_a cannot be initialized by the controller. This generalization consists in replacing $Z \cap X_{a0} \neq \emptyset$ with the stronger condition $X_{a0} \subseteq Z$ guaranteeing that no initial state of S_a is eliminated in the composition with the controller [1].

2-7 Linear temporal logic

Temporal logic formulae describe orderings of events in time without introducing time explicitly. They are often classified according to whether time is assumed to have a linear or a branching structure (Figure 2-9). The meaning of a temporal logic formula is determined with respect to a labeled state transition graph or a Kripke structure.

Definition 2.21 (Kripke Structure [14])**.** Let $P = \{p_0, p_1, \ldots\}$ be a set of atomic propositions[4]. A *Kripke structure M over P* is a four tuple $M = (S, S_0, R, L)$ where

- S is a finite set of states.
- $S_0 \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is a transition relation that must be total, that is for every state $s \in S$, there is a state $s^{'} \in S$ such that $R(s, s^{'})$.
- $L : S \to 2^P$ is a function that labels each state with the set of atomic propositions true in that state.

[4]In logic, an atomic proposition is a type of declarative sentence which is either true or false and cannot be broken down into other simpler sentences.

A relation R is total when for all $a, b \in X$, $aRb \vee bRa$. Then R is left-total iff for each $s \in S$ there exists $t \in T$ such that $(s, t) \in R$. That is, iff every element of S relates to some element of T. A *path* in the Kripke structure M from a state s is an infinite sequence (since R is total) of states $\pi = s_0 s_1 s_2 ...$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$. The *word* on the path π is the sequence of sets of the atomic propositions $w = L(s_1)L(s_2)L(s_3), ...$, which is an $\omega - word$ over alphabet 2^P. Since R is *left-total*, it is always possible to construct an infinite path through the Kripke structure. A *deadlock* state can be modeled by a single outgoing edge back to itself. The set of all atomic propositions that are true in $s \in S$ is $L(s)$.

Linear Temporal Logic (LTL) is a subset of the powerful logic called CTL* [47]. It was first introduced by Pnueli in 1977 [48], and it is a logic for specifying temporal properties for reactive and concurrent systems. In LTL, formulas are composed of *temporal operators* and *logical operators* [14]. The alphabet of LTL is defined as follows:

Definition 2.22 (The alphabet of LTL [14])**.** The alphabet of LTL is composed of:

- *Atomic proposition symbols*, such as p, q, r,...etc.
- *Logical connectives*: $\vee$ (or) and $\neg$ (not)
- *Temporal connectives*: $\mathbf{X}$ or "$\bigcirc$" and $\mathbf{U}$ or $\mathcal{U}$

The set of LTL formulae is defined as follows:

Definition 2.23 (The syntax of LTL [48])**.** Let P be the set of atomic proposition names. The syntax of path formulas is given by the following rules:

- if $p \in P$, then p is a path formula
- if ϕ and ψ are path formulas, then $\neg\phi$, $\phi \vee \psi$, $\mathbf{X}\phi$ and $\phi\mathbf{U}\psi$ are path formulas.

We define the semantics of LTL with respect to a Kripke structure M. We will use π_i to denote the path that starts from state s_i.

Definition 2.24 (The semantics of LTL [48])**.**

- $M, i \models p \iff p \in L(\pi(i))$
- $M, i \models \neg\phi \iff M, i \not\models \phi$
- $M, i \models \phi \vee \psi \iff M, i \models \phi$ or $M, i \models \psi$
- $M, i \models \bigcirc\phi \iff M, i + 1 \models \phi$
- $M, i \models \phi\mathbf{U}\psi \iff \exists k \in \mathbb{N} : M, k \models \psi$ and $M, j \models \phi \; \forall j : i \leq j < k$

If ϕ is a path formula, the notation $M, i \models \phi$ means that "ϕ is true at time instant i in the Kripke structure M". Since the modalities we have defined only talk about future time-points within a Kripke structure, it is not difficult to argue that a formula is satisfiable iff in some Kripke structure it is satisfied at the initial point.

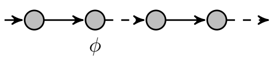
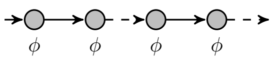
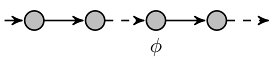
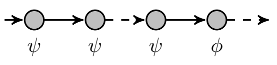
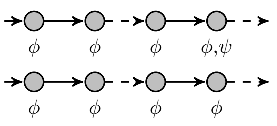
Text	Symbol	Explanation	Diagram
Unary Operators			
$\mathbf{X}\phi$	$\bigcirc\phi$	*neXt*: ϕ has to hold at the next state	
$\mathbf{G}\phi$	$\Box\phi$	*Always*: ϕ has to hold on the entire subsequent path.	
$\mathbf{F}\phi$	$\Diamond\phi$	*Eventually*: ϕ has to hold (somewhere on the subsequent path).	
Binary Operators			
$\psi\mathbf{U}\phi$	$\psi\mathcal{U}\phi$	*Until*: ψ has to hold *at least* until ϕ, which holds at the current or a future position.	
$\psi\mathbf{R}\phi$	$\psi\mathcal{R}\phi$	*Release*: ϕ has to be true until and including the point where ψ first becomes true; if ψ never becomes true, ϕ must remain true forever.	 ϕ ϕ ϕ ϕ,ψ ϕ ϕ ϕ ϕ

Table 2-1: LTL Syntax illustrated.

Proposition 2.1. Let ϕ be a path formula. Then ϕ is satisfiable iff there exists a Kripke structure M such that $M, 0 \models \phi$.

As usual, we introduce constants $\top$ and $\bot$ representing "true" and "false". We can write $\top$ as, for instance, $p_0 \vee \neg p_0$, where $p_0 \in P$ and $\bot$ as $\neg\top$. We can also generate normal Boolean connectives like $\wedge$ ("and"), $\rightarrow$ ("implication") and $\equiv$ ("equivalence") from the connectives $\neg$ and $\vee$ in the usual way - for instance, $\phi \rightarrow \psi = \neg\phi \vee \psi$.

We also introduce two derived modalities based on $\phi\mathcal{U}\psi$. We write $\Diamond\phi$ for $\top\mathcal{U}\phi$ and $\Box\phi$ for $\neg\Diamond\phi$. The modality $\Diamond$ is read as "eventually" while the modality $\Box$ is read as "always" or "globally". It is not difficult to verify the following facts:

- $M, i \models \Diamond\phi \iff \exists k \geq i : M, i \models \phi$.
- $M, i \models \Box\phi \iff \exists k \geq i : M, i \models \phi$.

Less formally, the semantics of the temporal operators are explained bellow and illustrated in Table 2-1.

- **X** or "$\bigcirc$" ("next time") requires that a property holds in the second state of the path.
- **F** or "$\Diamond$" ("eventually" or "in the future") is used to assert that a property will hold at some state on the path.
- **G** or "$\Box$" ("always" or "globally") specifies that a property holds at every state on the path.

- **U** or $\mathcal{U}$ ("until") is used to combine two properties: First, it holds if there is a state on the path where the second property holds, and second at every preceding state on the path, the first property holds.
- **R** or $\mathcal{R}$ ("release") is the logical dual of the **U** operator. It requires that the second property holds along the path up to and including the first state where the first property holds. However, the first property is not required to hold eventually.

By combining the above operators new temporal modalities can be obtained, such as "$\Box\Diamond$" for "infinitely often" or "$\Diamond\Box$" for "eventually always".

LTL is especially suited for expressing and verifying important properties of symbolic controllers, such as *safety* and *reachability* [22, 25, 23, 26]. In a transition system, such as a symbolic controller, a state is called *reachable* if there is a computation path from a defined initial state leading to this state. Reachability is one of the most important properties of transition systems in connection with safety properties. Suppose that u is a formula which expresses an undesirable property of a transition system. States satisfying u are usually called *unsafe* or *bad*. Naturally, one would like to know whether the system is safe. Reachability of a state satisfying u can be expressed as the existence of a path satisfying $\Diamond u$. Then the safety of the system can be expressed as non-reachability of a state satisfying u, i.e. $\Box\neg u$. The transition system is safe if this property is held on all computation paths.

Chapter 3

Set-destination shortest path problems

The algorithms we reviewed in the previous chapter for solving shortest path problems are not well-suited for our task. In particular, we are interested in finding the shortest path from all vertices to a destination-set of vertices, or the so called *Set-Destination Shortest Path (SDSP)*, for both deterministic and non-deterministic systems. In the case of deterministic systems, we will present an algorithm that uses the outcome of the Floyd-Warshall algorithm as an intermediate step to compute the set-destination shortest path. Unfortunately none of the well known shortest path algorithms is able to handle non-deterministic systems. For that, we will present an algorithm that is inspired by the Dijkstra's algorithm and based on reachability fixed-point algorithm.

Before we can present these algorithms, it is important to update the definition of a system, in order to include a cost map, that will allow us to define the cost of each transition.

Definition 3.1 (System)**.** A system S is a septuple $(X, X_0, U, \longrightarrow, Y, H, C)$ consisting of:

- a set of states X;
- a set of initial states $X_0 \subseteq X$;
- a set of inputs U;
- a transition relation $\longrightarrow \subseteq X \times U \times X$;
- a set of outputs Y;
- an output map $H : X \to Y$.
- a cost map $C : X \to \mathbb{R} \cup \{+\infty\}$.

We can now define the cost of a transition as:

Definition 3.2 (Transition Cost)**.** Let $S(X, X_0, U, \longrightarrow, Y, H, C)$ be a system, the cost of a transition is given by the operator $C_T : X \times X \longrightarrow \mathbb{R} \cup \{+\infty\}$ defined as:

$$C_T(x, x') = C(x') \text{ iff } \exists u \in U : (x, u, x') \in \longrightarrow$$

The above definition implies that the transition cost is only defined by a state x and its u-successor x'. If $(x, u, x') \in \longrightarrow$ and $(x, u', x') \in \longrightarrow$ are two transitions, then the transition cost is the same, namely $C_T(x, x') = C(x')$. We do not care about the input, as long as there is a transition in $\longrightarrow$. Having this in mind, we can treat our system as a *vertex-weighted directed graph* $D(X, \longrightarrow, C)$. In fact our complete algorithmic solution to the optimal control problems, treat the systems as vertex-weighted directed graphs. Nevertheless, the deterministic SDSP algorithm operates on any weighted-directed graph, that has no negative cycle and the non-deterministic SDSP algorithm operates on any weighted-directed graph, that has no negative weight.

3-1 Deterministic systems

The problem in which, given a *deterministic* system $S(X, X_0, U, \longrightarrow, Y, H, C)$, we want to find the shortest path from any initial state $x \in X$ to some set $W \subseteq X$, can been seen as a special case of the *all-pairs shortest path* problem. Therefore, in our approach we will first apply the Floyd-Warshall algorithm and then solve the *set-destination* shortest path problem.

To be concise with the definition of the system and any other following definition, we will assume that the cost-adjacency matrix and the outcome of the Floyd-Warshall algorithm is not expressed in terms of matrices but in terms of maps. Note that $\mathbb{R}_0^+$ is the set of all positive real numbers including zero.

Definition 3.3 (Cost of adjacent states)**.** Let $S(X, X_0, U, \longrightarrow, Y, H, C)$ be a finite deterministic system. The map $A_c : X \times X \longrightarrow \mathbb{R} \cup \{+\infty\}$ denotes the cost of a transition.

- $A_c(x, x') \in \mathbb{R}$ iff $x' \in Post_u(x)$ or
- $A_c(x, x') \in \{+\infty\}$ otherwise.

The shortest path cost and the pointer array are now treated as:

Definition 3.4 (Shortest path cost)**.** Let $S(X, X_0, U, \longrightarrow, Y, H, C)$ be a finite deterministic system. The map $C_{FW} : X \times X \longrightarrow \mathbb{R}_0^+ \cup \{+\infty\}$ denotes the shortest path cost between two states of the system.

- $C_{FW}(x, x') \in \mathbb{R}_0^+$ iff there exists a path from x to x' or
- $C_{FW}(x, x') \in \{+\infty\}$ otherwise.

Note that the Floyd-Warshall algorithm does not support negative cycles.

Definition 3.5 (Shortest path pointer map)**.** Let $S(X, X_0, U, \longrightarrow, Y, H, C)$ be a finite deterministic system. The map $P_{FW} : X \times X \longrightarrow X \cup \{\emptyset\}$ denotes the shortest path cost pointer map.

- $P_{FW}(x, x') = y \in X$ implies that y is adjacent to x' and intermediate in the (x, x')-path and

- $P_{FW}(x,x') = \emptyset$ iff $x' \in Post_u(x)$ or $C_{FW}(x,x') = +\infty$.

We will start by assuming that the Floyd-Warshall algorithm has already been applied. We consider the set-destination shortest path as the tuple (X, d_W), where X is the set of the states of the system S and $d_W : X \longrightarrow \mathbb{R}$ the map representing the shortest path cost to the set $W \subseteq X$. We can now define d_W as:

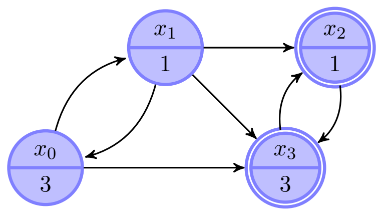
$$d_W(x) = \min_{w \in W}\{C_{FW}(x,w)\} \tag{3-1}$$

where $x \in X$ and $w \in W$. The above definition states that, in order to find the shortest path from an initial state $x \in X$ to the target set W, one has to pick the minimum shortest path cost value C_{FW}, defined by state x and all $w \in W$. In other words, since there might be more than one path from state x to the target set W, we are interested in the one, that has the minimum cost C_{FW}.

We are also interested in knowing for which $w \in W$ we acquire the minimum cost. For that, we define the operator $P_W : X \longrightarrow 2^W \cup \{\emptyset\}$, also referred to as *"pointer map"*, as:

$$P_W(x) = \{w | w \in W : +\infty \neq C_{FW}(x,w) = d_W(x)\} \tag{3-2}$$

The above definition states that, if there exists a shortest path from a state $x \in X$ to the target set W, then there exists $w \in P_W(x)$, such that $C_{FW}(x,w) = d_W(x)$. Otherwise $P_W(x) = \emptyset$. Note also that, given a state $x \in X$ there might be more than one $w \in P_W(x)$ such that $C_{FW}(x,w) = d_W(x)$. Figure 3-1 illustrates the special case of the all-pair shortest path problem.



$$C_{FW} = \begin{bmatrix} 4 & 1 & 2 & 3 \\ 3 & 4 & 1 & 3 \\ \infty & \infty & 4 & 3 \\ \infty & \infty & 1 & 4 \end{bmatrix}$$

$$d_W = \{2, 1, 3, 1\}$$

$$P_W(x) = \{\{x_2\}, \{x_2\}, \{x_3\}, \{x_2\}\}$$

Figure 3-1: Example of the deterministic SDSP algorithm (Algorithm 4), where $W = \{x_2, x_3\}$ is the target set. We assume that the Floyd-Warshall algorithm has already been applied.

The pointer map P_W is different than the pointer map P_{FW} used in Floyd-Warshall algorithm, as it cannot be used as a guide to reconstruct the desired shortest path, but rather as a way of knowing which destination, i.e. which $w \in W$, results in the shortest path. For example, node x_1 in Figure 3-1 has a transition to all the nodes in the target set W, but only state $P_W(x_1) = x_2 \in W$ results in the shortest path. To to be able to reconstruct the shortest path using both P_{FW} and P_W consider the following operator $N : X \times X \longrightarrow X \cup \{\emptyset\}$, which is defined as:

$$N(x,y) = \begin{cases} N(x, P_{FW}(x,y)) & \text{if } P_{FW}(x,y) \notin \emptyset \wedge C_{FW}(x,y) \in \mathbb{R}_0^+ \\ y & \text{if } P_{FW}(x,y) \in \emptyset \wedge C_{FW}(x,y) \in \mathbb{R}_0^+ \\ \emptyset & \text{if } P_{FW}(x,y) \in \emptyset \wedge C_{FW}(x,y) = +\infty \end{cases} \tag{3-3}$$

If $(x, P_W(x))$ is a path, the operator $N(x, P_W(x))$ will return the state $x' \in Post_u(x)$, i.e. the adjacent state to x. If we apply N for all states $x \in X$ we will construct a system whose states will only have transitions that lead towards the target set W.

Algorithm 4 presents a formal description of the algorithm that solves the set-destination shortest path problem for deterministic systems. To compute the running time of the SDSP algorithm we will assume that the operations on sets, such as addition, subtraction or comparison, take constant time and that n is the number of states. The update of the pointer map P_W in line 4, is of constant time. If we assume that the target set has all states $x \in X$, to compute the minimum value of all $w \in W$ (line 3) we would need n comparisons. Since we are seeking the minimum for all $x \in X$ (line 2), Algorithm 4 takes $O(n^2)$ time. Of course, before we apply Algorithm 4, we compute the Floyd-Warshall algorithm that has a running time of $O(n^3)$. Thus, we conclude that the deterministic SDSP algorithm takes a total time of $O(n^3)$, in the worst case. Note here that the complexity of the deterministic SDSP may vary, depending on the underlying data structures.

Algorithm 4 Deterministic SDSP Algorithm

Description: Given a system $S(X, X_0, U, \longrightarrow, Y, H, C)$ and a desired target set W, this algorithm computes the *set-destination* shortest path. It assumes that the operator C_{FW} is already defined by running the Floyd-Warshall algorithm.
Input: The system S, the operator C_{FW} and the target set $W \subseteq X$.
Output: The vector $d_W(x)$ for all $x \in X$ containing the shortest path cost value and the pointer map P_W.

```
1: function D-SDSP($S, C_{FW}, W$)
2:     for all ($x \in X$) do
3:         $d_W(x) = \min_{w \in W}\{C_{FW}(x, w)\}$
4:         $P_W(x) = \{w | w \in W : C_{FW}(x, w) = d_W(x)\}$
5:     end for
6: end function
```

3-2 Non-deterministic systems

As in the case of deterministic systems, we seek the shortest path from all states to a target set $W \subseteq X$. Since, the solution applied in deterministic systems cannot be used as it is, we use a different approach, in which the shortest path problem can be considered as a combination of the reachability game problem [1] (see section 2-6) and the single-source shortest path problem described by Dijkstra [28] (see section 2-2). This means that for the non-deterministic case, we first extract the set $R \subseteq X$ which guarantees that, the target set W we will be eventually reached, and then we find the shortest-pessimistic path from all $x \in R$ to W using a variation of Dijkstra's algorithm.

The term "pessimistic" refers to the fact that, given a state with only non-deterministic transitions, we consider the one that yields the *maximum*-shortest path distance. If a state has also deterministic transitions, then we pick the minimum distance of all deterministic transitions to compare it with the maximum distance of all non-deterministic transition, in order to find the shortest path distance. This way, we guarantee that the shortest path distance is upper-bounded.

Let $S(X, X_0, U, \longrightarrow, Y, H, C)$ be a non-deterministic system with no negative weights, i.e. $C \geq 0$ only. To make our approach more generically applicable, we consider the function $c : X \times U \times X \longrightarrow \mathbb{R}_0^+$ instead of C, to describe the cost of a transition $x \xrightarrow{u} x'$, where $(x, u, x') \in\longrightarrow$.

Each state of the system can have one or more deterministic transitions, one or more non-deterministic transitions or a combination of the two. If some or all of these transitions end up in a target set $A \subseteq X$, we consider the set $U_{S_A}(x) \subseteq U(x)$ as the set of all possible inputs that guarantee the transition to the set A. This state then belongs to the set $X_{S_A}(x) \subseteq X$, which represents the states that guarantee that if $u \in U_{S_A}(x)$ implies $Post_u(x) \subseteq A$. The fixed point operator $X_S : X \longrightarrow 2^X$ that is used to construct the set $X_{S_A}(X)$ is defined as:

$$X_{S_A}(x) = \{x \in X \mid \exists u \in U(x) : \emptyset \neq Post_u(x) \subseteq A\} \tag{3-4}$$

This operator is almost the same as one that solves the reachability game [1]. The only difference is that the target set W has no effect on the choice of states. We are interested in finding the set of nodes that are directly connected to the set A. The operator $U_S : X \longrightarrow 2^U$ is defined as:

$$U_{S_A}(x) = \{u \in U(x) \mid x \in X_{S_A} : \emptyset \neq Post_u(x) \subseteq A\} \tag{3-5}$$

which is a way of filtering the "in-valid" inputs, i.e. the inputs that do not enable a transition to the target set A. We shall call every transition that is forced by an input $u \in U(x)$ as a *u-transition.*

The complete solution of the non-deterministic shortest path problem requires the computation of the shortest path cost map $\mathsf{d}_W : X \longrightarrow \mathbb{R}_0^+ \cup \{+\infty\}$ defined as:

$$\mathsf{d}_W(x) = \begin{cases} c \in \mathbb{R}_0^+ & \text{if there exists a shortest path from x to the set } W \\ +\infty & \text{if does not exists a shortest path from x to the set } W \end{cases} \tag{3-6}$$

and the pointer map $\mathsf{P}_W : X \longrightarrow 2^U \cup \emptyset$, defined as:

$$\mathsf{P}_W(x) = \begin{cases} U_W \subseteq U(x) & \text{if } \mathsf{d}_W(x) \in (0, \infty) \\ \emptyset & \text{if } \mathsf{d}_W(x) = 0 \ \vee \ \mathsf{d}_W(x) = +\infty \end{cases} \tag{3-7}$$

which is used to trace back the shortest path. If $\mathsf{d}_W(x) = +\infty$ for some $x \in X \setminus W$, then there exists no shortest from x to W and $\mathsf{P}_W(x) = \emptyset$. In case that $x \in W$, then $\mathsf{d}_W(x) = 0$ and $\mathsf{P}_W(x) = \emptyset$. In any other case, $\mathsf{d}_W(x) \in (0, \infty)$ and $\mathsf{P}_W(x) \notin \emptyset$, pointing to the "valid" input(s) U_W to be used in order to reach the target set W.

Algorithm 5 describes the procedure for finding the shortest path value d_W and constructing the pointer map P_W for a given target set W. This implementation is based on the Dijkstra's single-source shortest path algorithm. The key differences are two; firstly the non-deterministic transitions are taken into account and secondly the algorithm starts from the end point, namely the target set W and propagates backwardly. If the set R represents the set where the shortest path has been already computed, the algorithm terminates if all states have been treated, i.e. if $Q = \emptyset$,. Note that, states that cannot be added to the R set, do not belong to the reach-set, i.e. are not part of the reachability game solution. Initially the set R is empty.

The set R, $\mathsf{P}_W(X)$ and the cost function $\mathsf{d}_W(X)$ are initialized at the beginning of the algorithm. Then the algorithm uses the fixed point operator X_{S_R} to extract the states from which the transition to the set R is certain. These states form the $X_R \subset X$ set. Of all the $x \in X_R$, the state with the minimum shortest-path estimate is being selected. For this state, the relaxation process, $Relax()$, updates the costs of the states, $x \in X_R$, in the case that the estimate of the shortest path from x to W is improved. The procedure of “relaxing” is similar to the definition used in [39].

The different part is the special treatment of the non-deterministic transitions. If given a state $x \in X_R$ and an input $u \in U_{S_R}(x)$, more than one u-successors exist, then the worst case scenario is taken into account. That is, the shortest distance estimate from x is updated with the maximum cost of all u-transitions. The complete solution of the non-deterministic shortest path problem is described in Algorithm 5.

Algorithm 5 Non-Deterministic SDSP Algorithm

Description: Given a non-deterministic (or deterministic) system $S(X, X_0, U, \longrightarrow, Y, H, C)$ and a desired target set W, this algorithm computes the *set-destination* shortest path.
Input: The system S and the target set W.
Output: The shortest path value $\mathsf{d}_W(x)$ and the pointer set $\mathsf{P}_W(x)$ $\forall x \in X$.

```
function ND-SDSP($S, W$)
    $\mathsf{d}_W(W) = 0$
    $\mathsf{P}_W(X) = \emptyset$
    $R = \emptyset$
    $Q = X$
    while ($Q \neq \emptyset$) do
        $x = \min\{\mathsf{d}_W(x) \mid x \in Q\}$
        if ($\mathsf{d}_W(x) \neq \infty$) then
            $R = R \cup x$
        end if
        $X_R = X_{S_R}(Q)$
        if ($X_R \neq \emptyset$) then
            for all ($x \in X_R$) do
                $Relax(x)$
            end for
        end if
        $Q = Q \setminus x$
    end while
    return $\mathsf{d}_W, \mathsf{P}_W$
end function
```

Algorithm 5 is considered a greedy algorithm. Although greedy strategies do not always yield optimal results, the following theorem shows that the non-deterministic set-destination shortest path algorithm does indeed compute the optimal result.

Algorithm 6 Relax

Description: The process of updating the shortest path estimate $d_W(x)$ and the pointer set P_W of the input state.

Input: The state x to be relaxed.

```
 1: function RELAX($x$)
 2:     $\mathsf{P}_{temp} = \emptyset$
 3:     for all ($u \in U_{S_R}(x)$) do
                                                            ▷ Deterministic case.
 4:         if ($|Post_u(x)| == 1$) then
 5:             if ($\mathsf{d}_W(x) \geq \mathsf{d}_W(x') + c(x,u,x')$) then
 6:                 $\mathsf{d}_W(x) = \mathsf{d}_W(x') + c(x,u,x')$
 7:                 $u_{temp} = u$
 8:             end if
                                                        ▷ Non-deterministic case.
 9:         else
10:             $max\_cost = \max\{\mathsf{d}_W(x') + c(x,u,x') \mid x' \in Post_u(x)\}$
11:             if ($\mathsf{d}_W(x) \geq max\_cost$) then
12:                 $\mathsf{d}_W(x) = max\_cost$
13:                 $u_{temp} = u$
14:             end if
15:         end if
16:                                                     ▷ Update the pointer set.
17:         if ($\mathsf{P}_{temp} = \emptyset$) then $\mathsf{P}_{temp} = u_{temp}$
                                    ▷ More than one input can give the same cost.
18:         else
19:             $\mathsf{P}_{temp} = \mathsf{P}_{temp} \cup u_{temp}$
20:         end if
21:     end for
22:     $\mathsf{P}_W(x) = \mathsf{P}_{temp}$
23: end function
```

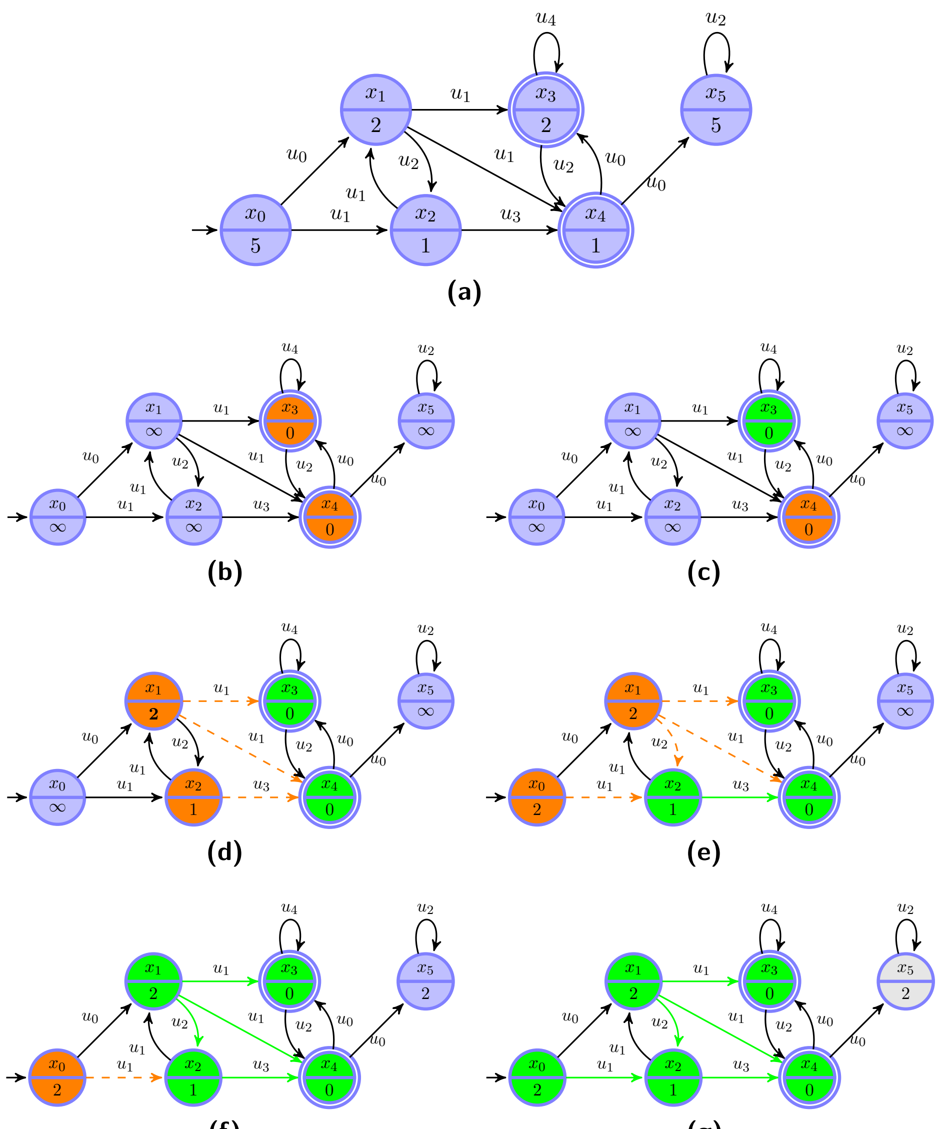


Figure 3-2: Example of the *non-deterministic SDSP* algorithm. The set $W = \{x_3, x_4\}$ is the target set. (a) Depicts the system S_a. The numbers below the states represent the cost of each state. (b)-(g) These are the steps of the non-deterministic SDSP algorithm. The numbers represent the d_W value of each state now. The states in orange are the states that are being *relaxed*. The green states belong to the resolved set R, i.e. to the set where the SDSP has already been computed, and the green inputs to the P_W pointer set.

Theorem 3.6 (Correctness of Algorithm 5)**.** Let $S(X, X_0, U, \longrightarrow, Y, H, C)$ be a system and let $W \subseteq X$ be a set of states. Algorithm 5 for system S and specification set W gives the optimal solution to the non-deterministic set-destination shortest path problem.

Proof. The proof is similar to the proof of the correctness of Dijkstra's algorithm. First, we note the following fact about the states in R, which is an immediate consequence of the reachability game: we know that there exists a path from each $x \in R$ to the target set W and the algorithm takes only these states into consideration to solve the shortest path problem.

Now, we only need to prove that the algorithm returns the optimal path in terms of the minimum (pessimistic) cost. If all the states had - only - deterministic transitions then by the proof of Dijkstra's algorithm, the values of $\mathsf{d}_W(x)$ are the shortest path distances to W. Therefore, we may think of a state x with non-deterministic transitions as a state with *only* deterministic transitions, if for all possible u-successors $x_i = Post_{u_i}(x)$, of each non-deterministic transition, we only consider one u-successor $x_i = Post_{u_i}(x)$ for which $\mathsf{d}_W + c(x, u_i, x_i)$ is maximum.

By the above argument, one can reduce the proof to the standard proof of Dijkstra's algorithm [39]. □

The running time of the non-deterministic SDSP algorithm, depends on the choice of the data structures. We will assume that the operations on sets, such as addition, subtraction or comparison, take constant time and that n is the number of states. To compute the minimum in each iteration, for all $x \in Q$ (line 7), we would need in the worst case, i.e. when $Q = X$, n operations. In total though, since we iterate over all $x \in X$, we would need $n(n-1)/2$ operations and thus the *min* operator needs $O(n^2)$ time. In the same sense, the operator X_{S_R} in line 11 takes the same time. Continuing with the "for" loop in line 13, we can consider that in the worst case the set W consists only of one state and at each iteration the operator X_{S_R} is returning the maximum possible states. In other words we assume that the operator will return $(n-1), (n-2), \ldots, 1$ states respectively in each iteration. This means that the "for" loop in line 13 needs $n(n-1)/2 - n$ operations, which correspond to a $O(n^2)$ time. The rest of the operations are of constant time ($O(1)$) and thus the total operations needed for Algorithm 5 are:

$$\frac{n(n-1)}{2} + \frac{n(n-1)}{2} + \left(\frac{n(n-1)}{2} - n\right) * T_{Relax} \tag{3-8}$$

Continuing on with the $Relax()$ function (Algorithm 6), the "for" loop of the function will repeat it self in the worst case $n-1$ times and that is when $|R| = n-1$. In that case, the *max* operation in line 5 might need up to $n-1$ operations, if we consider that the last remaining state would have only non-deterministic transitions. The rest of the operations take constant time. It is obvious now that the $Relax()$ function takes $O(n^2)$ time and as a result of this, the SDSP is runs in $O(n^4)$ total time.

To provide a more realistic estimation on the running time of the SDSP algorithm, we will consider the degree of a weighted directed graph. More precisely, if k is the maximum out-degree of some state, then the "for" loop of the $Relax()$ function will have a maximum of k iterations. The same holds also for the *max* operator. Thus, the total running time of the SDSP Algorithm is now $O(n^2k^2)$. Note that this running time depends directly on the connectivity of the graph representing the discrete abstraction. A strong connectivity results in a higher running time and a complete graph result in a running time of $O(n^4)$.

Notes

Since the deterministic systems can been seen as a subset of the non-deterministic ones, Algorithm 5 can been used to solve the shortest path problem for systems with deterministic behavior. Nevertheless, there is an important difference between the approach used in the deterministic systems and the approach used in the non-deterministic systems.

In the case of deterministic systems, we are computing the shortest path from all states to all states, using the Floyd-Warshall Algorithm and then we apply the operator (3-2) to get the set-destination shortest path. By keeping the result of the Floyd-Warshall Algorithm, if we want to compute the shortest path to a new set W', we only have to apply operator (3-2). Thus, this method enables us to compute the shortest path for different target sets, with higher computational efficiency.

However, this is not the case for Algorithm 5, since it is dependent of the target set W; if W changes, we have to apply Algorithm 5 again, in order to get the shortest paths for the new set.

Chapter 4

Optimal control problems

In this section we present the complete algorithmic approach for solving control problems with mixed qualitative-quantitative specifications. Our initial idea is to exploit the fact that, there are more than one "good" trajectories available when synthesizing *maximally-permissive* controllers, which allows us to address also quantitative specifications. Nevertheless, we will show that it is possible to aim directly for a controller with mixed qualitative-quantitative specifications, using merely the shortest path algorithms of the previous section.

4-1 Shortest path algorithms for mixed qualitative-quantitative specifications

At first, we show that the set-destination shortest path algorithms can be used successfully to enforce both liveness and safety constraints, while satisfying quantitative specifications. There are several kinds of qualitative specifications that can be enforced using only the SDSP algorithms. Below, we define five kinds of qualitative specifications:

1. **Stay:** trajectories start in the target set W and remain in W. This specification corresponds to the LTL formula $\Box\phi_W$, where ϕ_W is the predicate defining the set W.

2. **Reach:** trajectories enter the target set W in finite time. This specification corresponds to the LTL formula $\Diamond\phi_W$.

3. **Reach and Stay:** trajectories enter the target set W in finite time and remain within W thereafter. This specification corresponds to the LTL formula $\Diamond\Box\phi_W$.

4. **Reach while Stay:** trajectories enter the target set W in finite time while always remaining within the constraint set Z. This specification corresponds to the LTL formula $\Diamond\phi_W \wedge \Box\phi_Z$, where ϕ_Z is the predicate defining the set Z.

5. **Reach and Stay while Stay:** trajectories enter the target set W in finite time and remain within W thereafter while always remaining within the constraint set Z. This specification corresponds to the LTL formula $\Diamond\Box\phi_W \wedge \Box\phi_Z$.

It is important to mention here, that we assume that $W \subset Z$. In fact, this assumption holds for every definition and theorem in this chapter. Table 4-1 shows which of these specifications can be enforced by the corresponding deterministic or non-deterministic SDSP algorithm.

The deterministic SDSP algorithm (Algorithm 4) is capable of addressing *"Reach"* and *"Reach while Stay"* specifications as is, assuming that we provide the proper costs to each state of the system. The non-deterministic SDSP algorithm (Algorithm 5) on the other hand, is only capable of delivering *"Reach"* specifications (see Figure 3-2g). However, with a small modification to the initial algorithm we will be able to address *"Reach and Stay"* and *"Reach and Stay while Stay"* specifications, as we will show later. Recall that the SDSP algorithms are used as a mean to synthesize controllers and cannot be used directly for that purpose. We are only using the outcome of the algorithms, i.e. the shortest path cost and the pointer map, to synthesize the controllers. At this point it is important to distinguish the controllers with respect to the algorithm that has been used to synthesize them.

Definition 4.1 (Deterministic SDSP controller)**.** Let $S_a(X_a, X_{a0}, U_a, \underset{a}{\longrightarrow}, Y_a, H_a, C_a)$ and $W \subseteq X$. If there exists a solution to the shortest path problem, i.e. if there exists $x_{a0} \in X_{a0}$ such that $d_W(x_a) \neq \infty$ we define the *deterministic SDSP controller* $S_c(X_c, X_{c0}, U_c, \underset{c}{\longrightarrow})$ as:

- $X_c = \{x_a \in X_a \mid d_W(x) \neq +\infty\}$
- $X_{c0} = X_{a0} \cap X_c$
- $x_c \xrightarrow[c]{u_a} x'_c$ such that $x'_c = N(x_c, P_W(x_c))$ if $P_W(x_c) \neq \emptyset$

Definition 4.2 (Non-deterministic SDSP controller)**.** Let $S_a(X_a, X_{a0}, U_a, \underset{a}{\longrightarrow}, Y_a, H_a, C_a)$ and $W \subseteq X$. If there exists a solution to the shortest path problem, i.e. $X_{a0} \cap R \neq \emptyset$, we define the *non-deterministic SDSP controller* $S_c(X_c, X_{c0}, U_c, \underset{c}{\longrightarrow})$ as:

- $X_c = R$
- $X_{c0} = X_{a0} \cap X_c$
- $x_c \xrightarrow[c]{u_a} x'_c$ such that $u_a \in \mathsf{P}_W(x_c)$

Definitions 4.1 and 4.2 imply that the corresponding composed controller is the initial system S_a, but with the undesired states and inputs removed. Note that the relation defined by all the pairs $(x_c, x_a) \in X_c \times X_a$ with $x_c = x_a$ is an alternating simulation relation from S_c to S_a.

SDSP Controller	Qualitative Specification
Deterministic	“Reach”
	“Reach while Stay”
Deterministic - Revised	“Reach and Stay”
	“Reach and Stay while Stay”
Non-Deterministic	“Reach”
Non-Deterministic - Revised	“Reach and Stay”
	“Reach and Stay while Stay”

Table 4-1: SDSP controller qualitative specifications overview.

4-1-1 Deterministic SDSP controller

With the help of the above definitions, the following theorems show that the deterministic SDSP controller is able to satisfy “Reach” and “Reach while Stay” specifications. For the latter, we have to define the cost adjacency matrix (or the map A_c) in a special way to achieve the desired result, as Theorem 4.2 shows.

Theorem 4.1. Let $S_a(X_a, X_{a0}, U_a, \underset{a}{\longrightarrow}, Y_a, H_a, C_a)$ be a finite deterministic system and $W \subseteq X_a$ the target set. The deterministic SDSP controller S_c is non-blocking and satisfies the LTL formula $\Diamond\phi_W$ in finite time, where ϕ_W is the predicate defining the set W, if S_c is finite and non-empty.

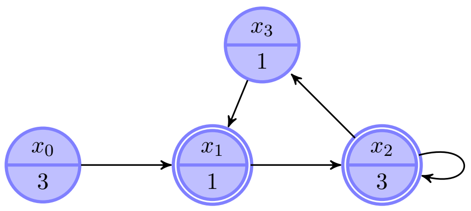
Proof. Let S_c be a finite non-empty controller. It holds from definition 4.1 that there exists a path from all $x_c \in X_c$ to the target set W, since $d_W(x_c) \neq +\infty$ for all $x_c \in X_c$. This means that the controller will eventually steer the system to the target set W in finite time. Since there is always a path from a state $x_c \in X_c$, S_c is considered as non-blocking. □

Theorem 4.2. Let $S_a(X_a, X_{a0}, U_a, \underset{a}{\longrightarrow}, Y_a, H_a, C_a)$ be a finite deterministic system, $Z \subset X_a$ the constrain set, $W \subset Z$ the target set and $Q = X_a \setminus Z$ the set of unsafe states. The deterministic SDSP controller S_c is non-blocking and satisfies the LTL formula $\Diamond\phi_W \wedge \Box\phi_Z$ in finite time, where ϕ_W is the predicate defining the set W and ϕ_Z is the predicate defining the set Z, if:

1. $C(q) = +\infty$ and $A_c(q, X_a) = +\infty$ for all $q \in Q$ and
2. S_c is finite and non-empty.

Proof. Let S_c be a finite non-empty controller. It holds from theorem 4.1 that the LTL formulae $\Diamond\phi_W$ is satisfiable.

If $C(q) = +\infty$ for all $q \in Q$, then all states in Q are precluded as intermediate states to the SDSP problem and thus we guarantee that all shortest paths will not go through the set Q. This however, will not satisfy the LTL formulae $\Box\phi_Z$, since there might exist a transition $(q, u_c, x_c) \in \underset{c}{\longrightarrow}$, where $q \in Q$ and $x_c \in X_c$. Thus, to preclude the states in Q from being considered as initial states, we define $A_c(q, X_a) = +\infty$ for all $q \in Q$. As a result of the previous actions, $d_W(q) = +\infty$ for all $q \in Q$, which means that all the transitions from the states in Q have been eliminated. From definition 4.1 it holds inductively that $Q \cap X_c = \emptyset$ or $Z \cap X_c = X_c$ and thus the LTL formulae $\Diamond\phi_W \wedge \Box\phi_Z$ is satisfiable. Finally, since there is always a path from a state $x_c \in X_c$, S_c is considered as non-blocking. $\square$



$$C_{FW} = \begin{bmatrix} \infty & 1 & 4 & 5 \\ \infty & 5 & 3 & 4 \\ \infty & 2 & 3 & 1 \\ \infty & 1 & 4 & 5 \end{bmatrix}$$

$$d_W = \{1, 3, 2, 1\}$$

$$P_W(x) = \{\{x_1\}, \{x_2\}, \{x_1\}, \{x_1\}\}$$

Figure 4-1: An example illustrating a case where the deterministic SDSP algorithm (Algorithm 4) fails to satisfy "Stay" specifications. $W = \{x_1, x_2\}$ is the target set. From state x_2 there is only a transition outside W.

In general, we can consider that the deterministic SDSP algorithm is able of solving *reachability* and *safety games*[1]. In fact, one can safely assume that all shortest path algorithms can provide solutions to the reachability games. Unfortunately, the deterministic SDSP controller is not able to deliver "Stay" and in turn also "Reach and Stay while Stay" specifications as Figure 4-1 shows. However, a solution to this problem exists, if we consider the fixed point operator $F_W : 2^X \longrightarrow 2^X$ used to solve safety games [1]. If $Z \subseteq W$ is specification set, then the F_W operator is defined as [1]:

$$F_W(Z) = \{x \in Z \mid x \in W \text{ and } \exists u \in U(x) : \emptyset \neq Post_u(x) \subseteq Z\} \tag{4-1}$$

This operator will help us construct the set $W_S \subseteq W$, for which we guarantee that for all $x \in W_S$, there exists $u \in U(x)$ such that $Post_u(x) \in W$. The set W_S can be obtained by iterating F_W as [1]:

$$W_S = \lim_{i \to \infty} F_W^i(W), \tag{4-2}$$

where the set W is also our specification set. It is now obvious, that the image of F_W contains all states that guarantee a transition inside W_S. This however, does not assure us, that there are - *only* - transitions inside W_S. In fact, there might *also* exist an input for which $Post_u(x) \notin W$. As a result of the above we consider the revised deterministic SDSP controller:

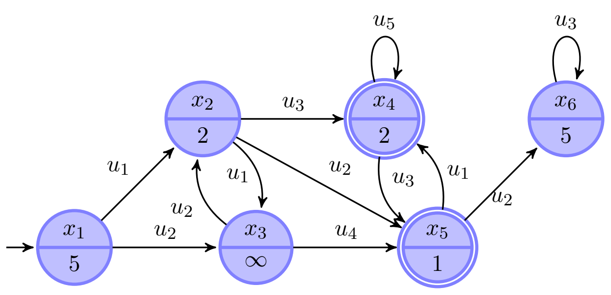
Definition 4.3 (Deterministic SDSP controller - Revised)**.** Let $S_a(X_a, X_{a0}, U_a, \underset{a}{\longrightarrow}, Y_a, H_a, C_a)$ be a finite system; $W \subseteq X$ the target set and $W_S \subseteq W$ the the image of F_W for specification set W. If there exists a solution to the shortest path problem, i.e. if there exists $x_{a0} \in X_{a0}$ such that $d_{W_S}(x_a) \neq \infty$, we define the *deterministic SDSP controller* $S_c(X_c, X_{c0}, U_c, \underset{c}{\longrightarrow})$ as:

- $X_c = \{x_a \in X_a \mid d_{W_S}(x_a) \neq +\infty\}$
- $X_{c0} = X_{a0} \cap X_c$

[1]Control problems enforcing safety specifications are also termed *safety games*, since the controller arises as the solution of a game played against an opponent that tries to prevent the composed system from being safe [1].

- $x_c \xrightarrow[c]{u_a} x'_c$ such that:
  a) $u_a \in \mathsf{P}_{W_S}(x_c)$ if $x_c \in (X \setminus W_S)$ or
  b) $\emptyset \neq Post_{u_a}(x_c) \subseteq W_S$ if $x_c \in W_S$

Note that the above definition implies that the set W_S is first constructed using the F_W operator and then the SDSP algorithm is applied using the new target set W_S as input. The following theorem shows now, that the revised deterministic SDSP controller is capable of enforcing also "Reach and Stay" and "Reach and Stay while Stay" specifications.



(a)

$$A_D = \begin{bmatrix} \infty & 2 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 2 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 2 & 1 & \infty \\ \infty & \infty & \infty & 2 & \infty & 5 \\ \infty & \infty & \infty & \infty & \infty & 5 \end{bmatrix}$$

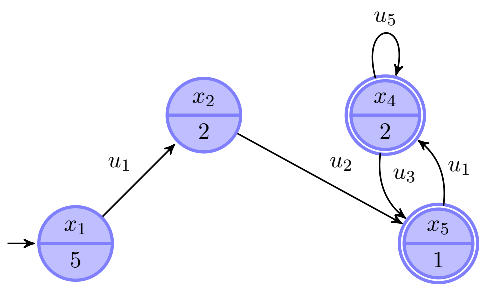
(b)

$$c_{sp} = \begin{bmatrix} \infty & 2 & \infty & 4 & 3 & 8 \\ \infty & \infty & \infty & 2 & 1 & 6 \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 2 & 1 & 6 \\ \infty & \infty & \infty & 2 & 3 & 5 \\ \infty & \infty & \infty & \infty & \infty & 5 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & 0 & 0 & 2 & 2 & 5 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(c)

$$\mathsf{d_W}(X) = \{3, 1, \infty, 1, 2, \infty\} \qquad \mathsf{P}(X) = \{\{x_5\}, \{x_5\}, \{\emptyset\}, \{x_5\}, \{x_4\}, \{\emptyset\}\}$$

(d)



(e)

Figure 4-2: Optimal controller synthesis of a deterministic system S, based on the revised SDSP controller. The set $W_S = \{x_4, x_5\}$ is the target set and the set $Z = \{x_1, x_2, x_4, x_5, x_6\}$ is the constraint set. (a) Depicts the system S, where the numbers under the states represent the cost of each state. (b) The cost-adjacency matrix; used as input for the Floyd-Warshall algorithm. (c) The result of the Floyd-Warshall algorithm. (d) The result of the SDSP algorithm 4. (e) The final controller satisfying qualitative (*"Reach and Stay while Stay"*) and quantitative specifications.

Theorem 4.3. Let $S_a(X_a, X_{a0}, U_a, \underset{a}{\longrightarrow}, Y_a, H_a, C_a)$ be a finite deterministic system; $Z \subset X_a$ the constrain set; $W \subset Z$ the target set and $Q = X_a \setminus Z$ the set of unsafe states. The deterministic SDSP controller S_c is non-blocking and satisfies the LTL formulas $\Diamond\Box\phi_W$ and $\Diamond\Box\phi_W \wedge \Box\phi_Z$ in finite time, where ϕ_W is the predicate defining the set W_S and ϕ_Z is the predicate defining the set Z, if:

1. $C(q) = +\infty$ and $A_c(q, X_a) = +\infty$ for all $q \in Q$ and
2. S_c is finite and non-empty.

Proof. Let S_c be a finite non-empty controller. It holds from theorems 4.1, 4.2 that the LTL formulas $\Diamond\phi_W$ and $\Box\phi_Z$ respectively are satisfied. It is also obvious from definition 4.3, that for all $x_c \in W_S$, $Post_{u_c}(x_c) \in W$ since S_c is non-blocking and there exists (only) inputs $u_c \in U(x_c)$ for which $(x_c, u_c, Post_{u_c}(x_c)) \in \underset{c}{\longrightarrow}$. From all the above, $\Diamond\Box\phi_W$ and $\Diamond\Box\phi_W \wedge \Box\phi_Z$ are satisfiable. □

4-1-2 Non-deterministic SDSP controller

In the case of the non-deterministic SDSP controller, only "Reach" specifications can be addressed, as we discussed earlier. To overcome this inefficiency, we provide a modified version of Algorithm 5 and as in the case of deterministic systems we also provide a revised version of the non-deterministic SDSP controller.

Definition 4.4 (Non-deterministic SDSP controller - Revised).
Let $S_a(X_a, X_{a0}, U_a, \underset{a}{\longrightarrow}, Y_a, H_a, C_a)$ be a finite system; $W \subseteq X_a$ the target set and $W_S \subseteq W$ the image of F_W for specification set W. If there exists a solution to the shortest path problem, i.e. $X_{a0} \cap R \neq \emptyset$, we define the *revised non-deterministic SDSP controller* $S_c(X_c, X_{c0}, U_c, \underset{c}{\longrightarrow})$ as:

- $X_c = R$
- $X_{c0} = X_{a0} \cap X_c$
- $x_c \underset{c}{\xrightarrow{u_a}} x'_c$ such that $u_a \in \mathsf{P}_{W_S}(x_c)$

The modified version of Algorithm 5 is illustrated formally in Algorithm 7. One difference when comparing this algorithm with the original one, is the decision to - *only* - relax states whose cost value is not infinity, i.e. $C(x) \neq \infty$. For the states with a cost of $+\infty$, namely the states in $Q = X \setminus Z$, it holds that $\mathsf{P}_W(q) = \emptyset$ for all $q \in Q$. Another difference lies in the initialization of the algorithm, where we keep all the inputs from the states $x \in W_S$ such that $Post_u(x) \in W_S$. As stated in the previous section, we are applying the algorithm for the new set W_S and not for the initial target set W.

Theorem 4.5. Let $S_a(X_a, X_{a0}, U_a, \underset{a}{\longrightarrow}, Y_a, H_a, C_a)$ be a finite system and $W \subseteq X_a$ the target set. The non-deterministic SDSP controller S_c satisfies the LTL formula $\Diamond\Box\phi_W$ in finite time, where ϕ_W is the predicate defining the set W, if S_c is finite and non-blocking.

Algorithm 7 Non-Deterministic SDSP Algorithm - Revised

Description: This algorithm is a revised version of Algorithm 5 in order to also solve *"Reach and Stay"* and *"Reach and Stay while Stay"* specifications. Given a system $S(X, X_0, U, \longrightarrow, Y, H, C)$ and the set W_S, this algorithm computes the *set-destination* shortest path.
Input: The system S and the target set W_S.
Output: The shortest path value $\mathsf{d}_{W_S}(x)$ and the pointer set $\mathsf{P}_{W_S}(x)$ $\forall x \in X$.

```
 1: function ND-SDSP($S, W_S$)
 2:     $\mathsf{d}_W(X \setminus W_S) = +\infty$
 3:     $\mathsf{d}_{W_S}(W_S) = 0$
 4:     $R = \emptyset$
 5:     $Q = X$
 6:     for all ($x \in X$) do
 7:         if ($x \in W_S$) then
 8:             $\mathsf{P}_{W_S}(x) = \{u \in U(x) \mid Post_u(x) \in W_S\}$
 9:         else
10:             $\mathsf{P}_{W_S}(x) = \emptyset$
11:         end if
12:     end for
13:     while ($Q \neq \emptyset$) do
14:         $x = \min\{\mathsf{d}_{W_S}(x) \mid x \in Q\}$
15:         if ($\mathsf{d}_{W_S}(x) \neq \infty$) then
16:             $R = R \cup x$
17:         end if
18:         $X_R = X_{S_R}(Q)$
19:         if ($X_R \neq \emptyset$) then
20:             for all ($x \in X_R$) do
21:                 if ($C(x) \neq \infty$) then
22:                     $Relax(x)$
23:                 end if
24:             end for
25:         end if
26:         $Q = Q \setminus x$
27:     end while
28:     return $\mathsf{d}_{W_S}, \mathsf{P}_{W_S}$
29: end function
```

Proof. Let S_c be a finite non-blocking and non-empty controller. Initially, we consider all states $x_c \in R$ for which $\mathsf{d}_W(x_c) \neq 0$, i.e. all $x_c \in (R \setminus W)$. We know that for all $x_c \in R$, $\mathsf{d}_W(x_c) \neq +\infty$ and thus, for all these states there exists a path to W and $\Diamond\phi_W$ is true for all $x_c \in (R \setminus W)$. This means that the controller will eventually steer the system to the target set W in finite time.

From the definition 4.4 it holds that for all $w \in W$, $Post_{u_c}(w) \in W$. For the purpose of contradiction, suppose that there exists a transition $(w, u_c, x_c) \in \xrightarrow[c]{}$, such that $w \in W_S$ and $x_c \in (X_c \setminus W)$. From definition 4.4 it holds that if $(w, u_c, x_c) \in \xrightarrow[c]{}$, then $x_c \in W_S$, which violates our assumption. Furthermore, it holds that $Post_{u_c}(w) \neq \emptyset$ for all $w \in W_S$, since S_c is non-blocking and non-empty. Thus, for all $w \in W$, $Post_{u_c}(w) \in W$ and $\Box\phi_W$ is true for all $w \in W$.

We can now safely assume that the LTL formulae $\Diamond\Box\phi_W$ is satisfiable. $\square$

Theorem 4.6. Let $S_a(X_a, X_{a0}, U_a, \xrightarrow[a]{}, Y_a, H_a, C_a)$ be a finite deterministic system, $Z \subset X_a$ the constrain set, $W \subset Z$ the target set and $Q = X_a \setminus Z$ the set of unsafe states. The non-deterministic SDSP controller S_c satisfies the LTL formula $\Diamond\Box\phi_W \wedge \Box\phi_Z$ in finite time, where ϕ_W is the predicate defining the set W and ϕ_Z is the predicate defining the set Z, if:

1. $C(q) = +\infty$ for all $q \in Q$ and
2. S_c is finite, non-blocking and non-empty.

Proof. We only have to show that the LTL formulae $\Box\phi_Z$ holds everywhere, since we know from Theorem 4.5 that $\Diamond\Box\phi_W$ is true. This proof is similar to the proof of Theorem 4.2.

Let S_c be a finite non-blocking and non-empty controller. If $C(q) = +\infty$ for all $q \in Q$, then all states in Q are precluded as intermediate states to the set-destination shortest path problem and thus we guarantee that all shortest paths will not go through the set Q. Furthermore, from the revised SDSP algorithm (Algorithm 7), it holds that for all $x_a \in X_a$ such that $\mathsf{d}_{W_S}(x_a) = +\infty$, the maps $\mathsf{d}_{W_S}(x_a)$ and $\mathsf{P}_{W_S}(x_a)$ are never being updated, since the $Relax(x_a)$ function is *never* being called. It is obvious now that, for all $q \in Q$, $\mathsf{d}_{W_S}(q) = +\infty$ and $Q \notin R$. Thus, from definition 4.2 it holds inductively that $Q \cap X_c = \emptyset$ or $Z \cap X_c = X_c$ and thus the LTL formulae $\Box\phi_Z$ is satisfiable.

We can now safely assume that the LTL formulae $\Diamond\Box\phi_W \wedge \Box\phi_Z$ is satisfiable. $\square$

Figure 4-3 illustrates the flexibility of the revised non-deterministic SDSP algorithm to address qualitative and quantitative specifications.

The question is now, what is the is shortest path to a computationally more efficient solution? Aiming for a *maximally-permissive* controller and then apply the SDSP algorithms *or* apply directly the SDSP algorithms to synthesize controllers with mixed qualitative-quantitative specifications?

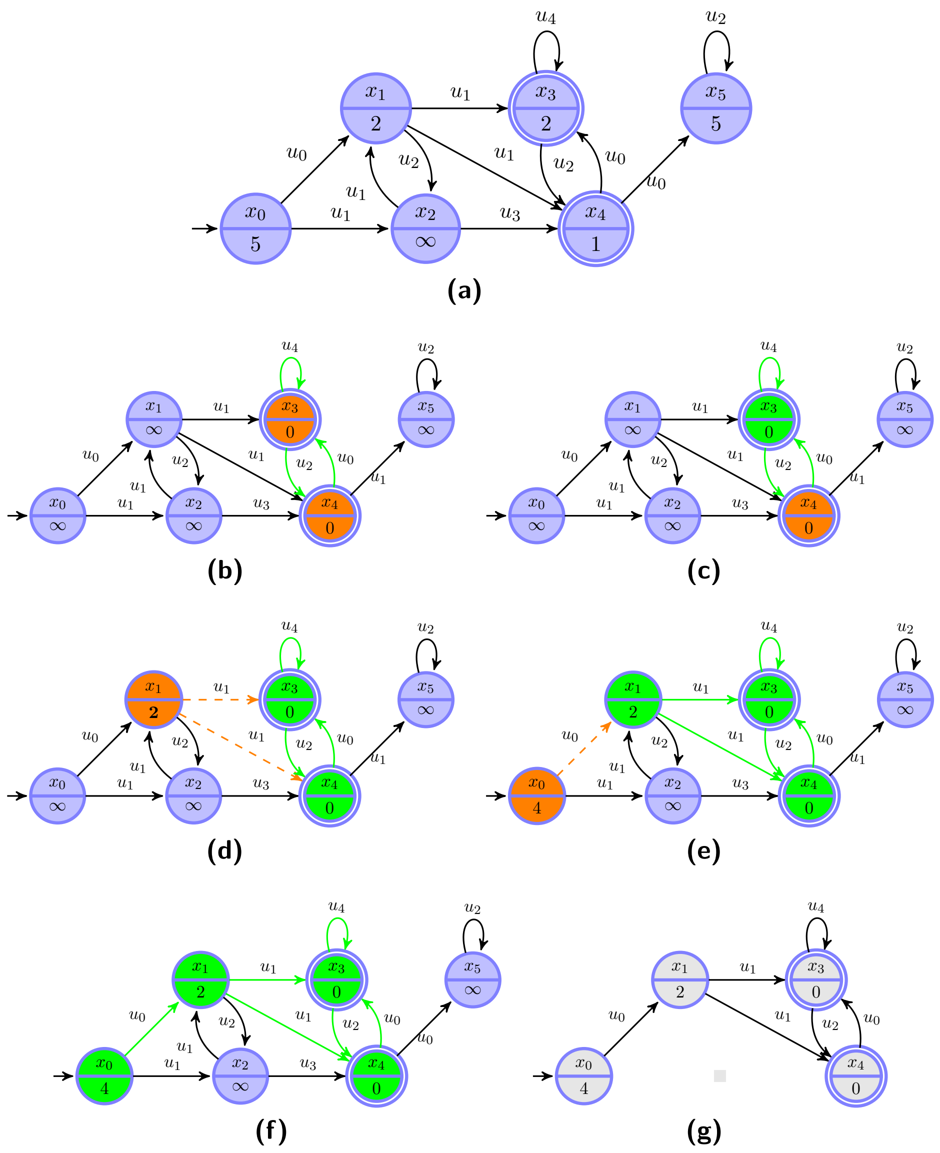


Figure 4-3: Optimal controller synthesis of a non-deterministic system S. The set $W = \{x_3, x_4\}$ is the target set and the set $Z = \{x_0, x_1, x_3, x_4, x_5\}$ is the constraint set. (a) Depicts the system S. The numbers represent the cost of each state. (b)-(f) These are the steps of the non-deterministic SDSP algorithm. *The numbers represent the d_W value of each state now.* The states in orange are the states that are being *relaxed*. The green states belong to the resolved set R, i.e. to the set where the SDSP has already been computed. (g) The final controller satisfying qualitative (*"Reach and Stay while Stay"*) and quantitative specifications.

4-2 Solving optimal control problems

In this section, we present the complete algorithmic approach for solving control problems with mixed qualitative-quantitative specifications. We have shown that it is possible to follow two different directions towards our goal. One option is to first synthesize a *maximally-permisive* controller with qualitative specifications and then use the shortest path algorithms to refine the controller in order to enforce also quantitative specifications. The other option is to directly synthesize a controller with mixed qualitative-quantitative specifications using only the shortest path algorithms.

Below, we present the typical sequence of steps needed to synthesize a controller with mixed qualitative-quantitative specifications.

1. Construct only the discrete abstraction S_a of the continuous system **or** synthesize a *maximally-permissive* controller S_c with the desired qualitative specifications (details here [1, 26]).
2. Define the cost of each state of the system S_a **or** the controller S_c. This can be achieved through the corresponding characteristic functions.
3. Apply the corresponding set-destination shortest path algorithm:
   - In case of deterministic systems:
     (a) Create the cost-adjacency matrix of the system.
     (b) Apply the Floyd-Warshall algorithm.
     (c) Apply the deterministic SDSP (Algorithm 4).
   - In case of non-deterministic systems: Apply the desired non-deterministic SDSP (Algorithm 5 or Algorithm 7).
4. Synthesize a controller **or** refine the controller, based on the results of the SDSP algorithm.

The "refinement" of the controller, which is mentioned in step (4) is basically the process where we assume the system[2] that satisfies our qualitative specifications, witch we then use it as input to the SDSP algorithm to synthesize the final controller that enforces also quantitative specifications.

Beyond the problems of optimal controls that have been covered in this thesis, in the latest work of Roo and Mazo Jr. [26] more general optimal control problems are addressed. These include mixed qualitative-quantitative problems, where the qualitative specifications are given by a formulae in the form of $\phi \wedge \Diamond P$, where $P \subseteq H(W)$ is the set of outputs that correspond to a specification set W and ϕ is a formula in the safe-LTL fragment of LTL [49].

It is important at this point to mention that all the algorithms for solving optimal control problem are offered as an extension (Figure 4-4) to the freely available `MATLAB` toolbox `Pessoa` [2]. `Pessoa` currently supports the synthesis of controllers enforcing "Stay", "Reach", "Reach and Stay" and "Reach and Stay while Stay" specifications.

[2] $S_c \times_{\mathcal{F}} S_a$, where S_c is some maximally permissive controller enforcing the desired qualitative specifications

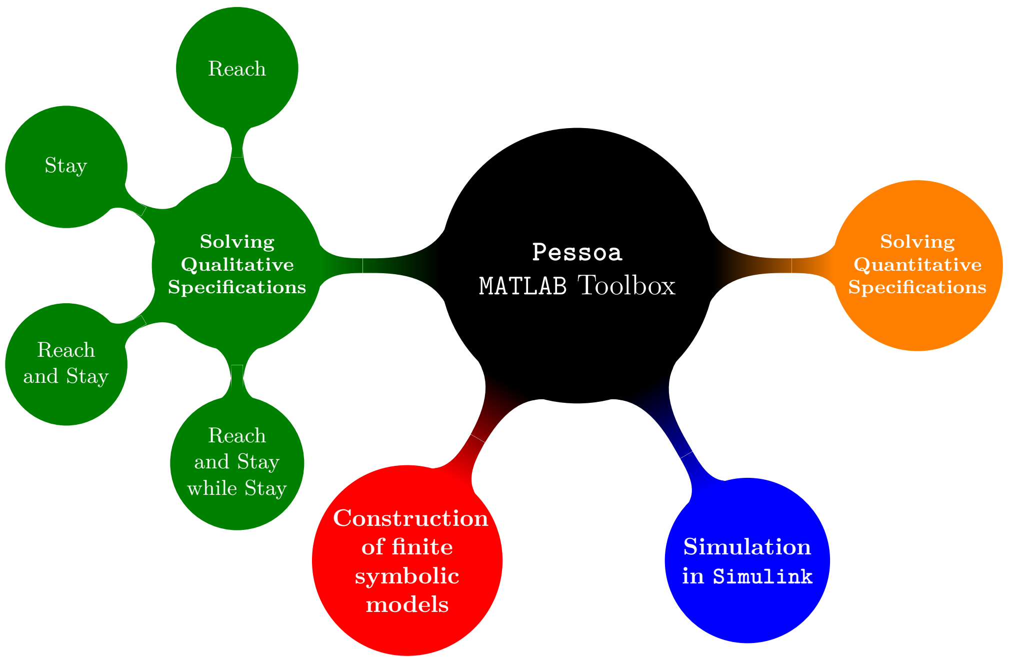

Reach
Stay
Solving Qualitative Specifications
Reach and Stay
Reach and Stay while Stay
Pessoa MATLAB Toolbox
Solving Quantitative Specifications
Construction of finite symbolic models
Simulation in Simulink

Figure 4-4: MATLAB toobox Pessoa for synthesizing symbolic controllers. The figure illustrates the current features of Pessoa [2]. More info at Pessoa's website (https://sites.google.com/a/cyphylab.ee.ucla.edu/pessoa/home).

Although simple, the above specifications already allow Pessoa to solve non-trivial synthesis problems that frequently arise in applications. Pessoa also provides the possibility to simulate the closed-loop behavior in Simulink. For this purpose, Pessoa comes with a Simulink block, implementing a refinement of any synthesized controller. Pessoa together with source code of the SDSP algorithms can be downloaded freely (Open Source Project) from http://code.google.com/p/pessoa/.

Chapter 5

Conclusions and future work

In this thesis we have presented a novel approach on the automatic synthesis of controllers with mixed qualitative-quantitative specifications. To achieve that, the discrete abstraction of the continuous model is considered. Discrete abstractions not only provide *correct-by-design* control strategies, but also enable the synthesis of controllers for classes of specifications that traditionally have not been considered in the context of continuous control systems. These include specifications formalized using regular languages, fairness constraints, temporal logics, etc. We are particularly interested in the Linear Temporal Logic. LTL is very appealing, as it allows us to easily specify *qualitative specifications.* Our goal is however, to design and synthesize controllers that also enforce *quantitative* specifications.

Towards our goal we have proposed two approaches. In the first one, we suggest synthesizing a *maximally-permissive* controller under given safety and liveness constraints. Then apply the corresponding set-destination shortest path algorithm and use the outcome as information to refine the controller, such that it also satisfies the desired quantitative specifications. In the second approach, we suggest using the SDSP algorithms directly, to synthesize controllers with mixed qualitative-quantitative specifications. But, the question is, if it is computationally more efficient to apply directly the SDSP algorithms to solve control problems with mixed qualitative-quantitative specifications or not. Unfortunately we cannot directly provide such an answer, since it is first necessary to analyze the complexity of the techniques used to synthesize controllers with qualitative specifications. Due to time constraints such an attempt was not made.

Regardless which approach we choose follow, our problem can be treated as a path-finding problem on a finite graph. To solve such problems, we have presented two shortest path algorithms that allows us to find the shortest path in the case of deterministic and non-deterministic systems. Since non-deterministic systems can been seen as a superset of deterministic ones, the solution provided for non-deterministic systems can also be applied for both types. However, in the case of deterministic systems, if we want to apply the SDSP algorithm again for a different target set, the deterministic SDSP algorithm is more efficient compared to the non-deterministic SDSP algorithm, as we do not have to re-apply the Floyd-Warshall

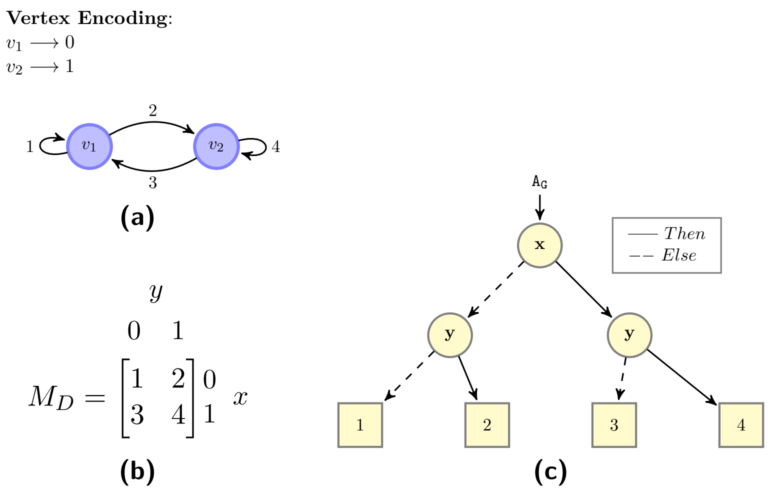


Figure 5-1: An example to illustrate when the worst case complexity of an ADD can occur. The ADD data structure is not offering any compression in this case. Operations on the ADD would take also more time compared to the array data structure. (a) A *complete* directed graph and the vertex encoding. (b) The cost-adjacency matrix, where all values are different. (c) The resulting ADD that represents the directed graph. The variable ordering is $x < y$.

algorithm. Note also that the deterministic SDSP algorithm runs in $O(n^3)$, while the non-deterministic SDSP algorithm in $O(n^2k^2)$. It is clear now, that when we are dealing with deterministic systems, it is preferable to use the deterministic SDSP algorithm, in order to achieve better computation times. But, whether applying the non-deterministic SDSP or the deterministic SDSP algorithm, the computation time is greatly affected by the connectivity of the digraph representing the discrete abstraction and by the number of different weight values. More precisely, the non-deterministic SDSP algorithm and the ADD data structure are directly affected by this, as the complexity of the former depends on the maximum out-degree of the digraph and the complexity of the latter rises with the level of connectivity and the variance in the weights. A comparison of the two shortest path algorithms is presented in Table 5-1.

Besides providing a theoretical framework on solving optimal control problems, our goal was also to implement the algorithms, in order to investigate and to illustrate the practicality and feasibility of our approach. The key component in our implementation was the use of the BDD/ADD data structure. ADD's provide an ideal form of storing discrete abstractions by offering great data compression, while being fairly easy to use and manipulate. However, if the cost adjacency matrix or the all-pairs shortest path cost matrix has a high density[1], the use of ADDs can drastically affect the performance of the SDSP algorithms when compared to other data structures, such as array data structures. Note that the density of the cost adjacency matrix correlates with the connectivity of the weighted directed graph that is used to represent the discrete abstraction. The stronger a graph is connected, the higher the complexity of the ADD structure tends to be, if of course the matrix-elements have values that differ from each other (Figure 5-1). Higher complexity in the ADD structure, results in higher computation times, as simple operations on ADDs need more time to complete.

[1]When a highly large number of elements differ from each other.

SPDP Algorithms	
Deterministic	**Non-Deterministic**
If the target set W changes we do *not* need to compute Floyd-Warshall algorithm again	If the target set W changes we need to apply it again.
Can be used *only* for deterministic systems	Can be used *also* for deterministic systems
$O(n^3)$, n: number of states	$O(n^4)$ or $O(n^2k^2)$, k: max out-degree
Can be used to enforce: 1. "Reach" 2. "Reach and Stay" 3. "Reach and Stay while Stay" qualitative specifications	Can be used to enforce: 1. "Reach" 2. "Reach and Stay" 3. "Reach and Stay while Stay" qualitative specifications

Table 5-1: SDSP Algorithms comparison.

Unfortunately, this is the "Achilles' heel" of the ADD data structure and has to be taken into consideration. In fact, we believe that the ADD data structure might not be a good choice for the implementation of the SDSP algorithms. However, this should not be considered as a concrete conclusion, as it needs further investigation.

5-1 Future work

By now, it is clear that our main focus in this thesis is the theoretical foundation of the SDSP algorithms and their implementation using ADDs, with our ultimate goal to synthesize controllers with mixed qualitative-quantitative specifications. Although this goal was achieved, due to time constraints the techniques we have presented have not been analyzed to determine their limitations and their performance. Nevertheless, some limitations have been more or less revealed to us.

We have seen that a system $S(X, X_0, U, \longrightarrow, Y, H, C)$, can be treated as a vertex-weighted directed graph. This means that each node is assigned with a cost and thus the cost of each transition is determined by the head node, i.e. the cost of $(x, u, x') \in\longrightarrow$ is $C(x')$. As a result of this, for all $x \in X$ such that $Post_u(x) = x'$ the transition cost is the same, namely $C(x')$. In other words the input has no impact on the cost of a transition. Consequently, to make the cost map more flexible, one can also consider the input to determine the cost of a transition.

Another limitation in our approach, is the inability of the native Floyd-Warshall algorithm to capture transitions that yield the same shortest path cost. This affects directly the deterministic SDSP algorithm and as an immediate consequence, multiple trajectories that yield the same cost are not taken into consideration. This however, is not the only drawback of the Floyd-Warshall algorithm. The algorithm is not the optimal all-pairs shortest path algorithm to date with respect to time-complexity. It has a time complexity of $O(n^3)$, while other approaches as the one of Pettie [50] has a time-complexity of $O(mn + n^2 \log \log n)$, where

n is the number of vertices and m the number of edges. Of course, an even more efficient approach is to implement the all-pairs shortest path algorithm in parallel, for which known algorithms already exist [51]. In general, we might say that there is room for optimization in every algorithm presented in this thesis. The implementation of parallel algorithms would have a great impact on the computation time when solving optimal control problems.

Optimization can also be considered with respect to the ADD (or BDD) data structure. ADDs have to be evaluated and compared with simpler data structures, such as array data structures, to check whether they are suitable for our purpose or not. The sole purpose of the ADD data structure is its characteristic to offer great compression when storing discrete abstractions and to accelerate simple operations if the underlying boolean function is not dense[2]. Unfortunately, this characteristic may fade out in the case of the SDSP algorithms, where the matrices that represent the shortest path costs tend to become dense with a great variance in the values that they store.

Beyond the performance optimizations, one can also think about optimizations in terms of expanding the classes of qualitative specifications that can be addressed. We have seen that we can use the SDSP algorithms to synthesize controllers that enforce "Stay", "Reach", "Reach and Stay" and "Reach and Stay while Stay" specifications. Although simple, the above specifications are a very important class of specifications, as such specifications are required in many applications. Even so, one might wish for more complex specifications. Discrete abstractions offer such a possibility, but the algorithms presented in this thesis are not able to provide solutions. Towards that direction, one can start seeking answers in [26].

[2]A boolean function is not dense if $\log_2 N \ll n$, if $f : D \longrightarrow \{0, 1\}$ is of n and $N = |D|$ [52].

Appendix A

ADD Implementation

As we have seen, the algorithms for solving shortest path problems, presented in previous sections, are used to help us solve optimal control problems. Besides constructing a theoretical framework, we also care in investigating how these algorithms can be implemented to solve actual control problems and in turn illustrate the feasibility of our approach. BDDs (and ADDs) are being used as the main data structure for storing discrete abstractions and symbolic controllers. Below we present an implementation of the algorithms presented in this thesis using BDDs.

A-1 Floyd-Warshall Algorithm

We have seen that the ADD data structure provides an efficient way to reduce the data size of matrices, especially when it comes to sparse matrices. Nevertheless, the Floyd-Wharshall all-pair shortest path algorithm operates on raw matrices and cannot be applied directly to graphs that are represented as ADDs. The boolean, arithmetic and abstraction operations on ADDs, allows us to transform the Floyd-Wharshall algorithm so that it can be applied on an ADD [44].

The key step for that is to express equation (2-2) using the corresponding ADD operations. So, considering a digraph $D(V, A, c)$, for every $k \in \{1, 2, .., |V|\}$, we want to compute the $\min(W^k(x, y), c^k(x) + r^k(y))$, where $W^1(x, y)$ is the cost adjacency matrix, $c^k(x)$ and $r^k(y)$ the k'th column and row respectively. This equation can be expanded to a recursive definition using the outer of $\min(W^k(x, y), c^k(x) + r^k(y))$ and the top variable between $W(x, y)$, $r^k(y)$ and $c^k(x)$:

$$\begin{aligned}\min(W^k(x, y), c^k(x) + r^k(y)) =& v \cdot \min(W^k_v(x, y), c^k_v(x) + r^k_v(y)) \\ &+ v' \cdot \min(W^k_{v'}(x, y), c^k_{v'}(x) + r^k_{v'}(y))\end{aligned}$$

This shows that the above procedure iterates over every vertex (or every element of the cost adjacency matrix) in order to find the $\min(W^k(x, y), c^k(x) + r^k(y))$. Algorithm 9 describes the procedure for computing the outer sum and Algorithm 8 illustrates the complete Floyd-Wharshall algorithm using ADDs.

Algorithm 8 Floyd-Warshall's Algorithm using ADDs

Input: The cost adjacency matrix W as an ADD, representing the weights of a digraph $D(V, E, c)$.

Output: Matrix D containing all-pair shortest distances and the pointer array P. Both represented as ADDs.

```
1: function FLOYD-WARSHALL($W$)
2:     $D \leftarrow W$
3:     $P \leftarrow 0$
4:     for ($k = 1$ to $k = |V|$) do
5:         $R = Extract\_Row(D, k)$
6:         $C = Extract\_Column(D, k)$
7:         $S = Outer\_Sum(D, R, C)$
8:         $D = S[0]$
9:         $P = S[1]$
10:    end for
11:    return $D, P$
12: end function
```

Deterministic Set-destination Shortest Path Algorithm

The BDD-implementation of Algorithm 4, which is used for deterministic systems, is pretty straightforward. The only difference when compared to the theoretical approach, is the way we construct the $n \times 1$ pointer array P_W. For the shake of simplicity we follow a slightly different approach.

Let $S(X, X_0, U, \longrightarrow, Y, H)$ be a system and $W \subseteq V$ the target set for which we want to find the set-destination shortest path. While the map P_W (see Definition 3-2) is used to point the state $w = P_W(x)$ of the target set W, for which the path $x, w)$ is the shortest one, the pointer array P_W does not explicitly store this information for all nodes, but rather for the nodes for which the element (x, w) in the array P returns zero.

More precisely, let $X' \subseteq X$ be the set of nodes that are adjacent to some nodes of $w' \in W' \subseteq W$ and for which each (x', w')-path is the shortest path in the all-pairs shortest path problem. In this case, the pointer array P will have the zero value for all (x', w') elements. Recall that the zero value for some element (i, j) of P states that i and j are adjacent or that there is no (i, j)-path. It is now clear that we have to update all (x', w') elements of the pointer array P, now denoted as P_W, to point to the corresponding node $w' \in W'$ for which the (v', w')-path is the shortest one (in the set-destination shortest path problem). For the rest of the nodes $x \in X \setminus X'$, if there exists a node $w \in W$ such that $d_W(x) \neq \infty$, the elements $(x, 1)$ of P_W either get the value of the corresponding (x, w) elements of the P array or zero otherwise. Figure 3-1 illustrates how the pointer array P_W is constructed. A formal description of the BBD-implementation of Algorithm 4 is presented in Algorithm 10.

The *Replace_Zero*() function in Algorithm 10 is simple function that substitutes the zero-valued terminal node of a given ADD with the corresponding k-valued terminal node. This

Algorithm 9 Outer Sum procedure

Input: The cost adjacency matrix W as an ADD, representing the weights of a digraph $D(V,E,c)$.

Output: Matrix D containing all-pair shortest distances and the pointer array P. Both represented as ADDs.

```
1: function FLOYD-WARSHALL($W$)
2:     if $(r == \infty)$ or $(c == \infty)$ then
3:         $R = D$
4:         $P = 0$
5:         return $R, P$
6:     end if
7:
8:     if $(IsConstant(c)$ and $IsConstant(r))$ then
9:         $R = c + r$
10:        if $IsConstant(D)$ then
11:            if $R < M$ then
12:                $P = k$
13:            else
14:                $R = D$
15:                $P = 0$
16:            end if
17:        else
18:            $M = Apply(R, D, min)$
19:            $R = M[0]$
20:            $P = M[1]$
21:        end if
22:        return $R, P$
23:    end if
24:
25:    $R = CacheLookup(Outer\_Sum\_tag, D, r, c)$
26:    $P = CacheLookup(Pointer\_Array\_tag, D, r, c)$
27:
28:    $v = Top\_Var(D, r, c)$
29:    $M = Outer\_Sum(D_v, r_v, c_v, k)$
30:    $R_v = M[0]$
31:    $P_v = M[1]$
32:    $M = Outer\_Sum(D_{v'}, r_{v'}, c_{v'}, k)$
33:    $R_{v'} = M[0]$
34:    $P_{v'} = M[1]$
35:    $R = Find\_or\_Create(R_v, R_{v'})$
36:    $P = Find\_or\_Create(P_v, P_{v'})$
37:
38:    return $R, P$
39: end function
```

Algorithm 10 Deterministic SDSP Algorithm - BDD Implementation

Input: The all-pairs shortest path cost array C_{FW} and pointer array P_{FW} of a system $S(X, X_0, U, \longrightarrow, Y, H)$ and the target set $W \subseteq X$.

Output: The vector d_W containing the shortest path cost value for all $x \in X$ and the pointer array P_W.

```
1:  function D-SDSP($C_{FW}, P_{FW}, W$)
2:      $d_W \leftarrow \infty$
3:      $P_W \leftarrow 0$
4:      for $k = 1$ to $k = |W|$ do
5:          $C_{APSP} = Extract\_Column(C_{FW}, w_k)$
6:          $C_P = Extract\_Column(P_{FW}, w_k)$
7:          $C_P = Replace\_Zero(C_P, k)$
8:
9:          $Rslt = Apply\_Min2(d_W, C_{APSP}, P_W, C_P)$
10:         $d_W = Rslt.min$
11:         $P_W = Rslt.P$
12:     end for
13:     return $d_W, P_W$
14: end function
```

way, we make sure to point to the node $w \in W$ for which we get the shortest path. Indeed, given a w_k and for every $x \in X$, if $d_W(x) \neq \infty$ and $P(x, w_k) = 0$, then $P_w(x, 1) = k$. The *Apply_Min*2() function also presented in Algorithm 11 is nothing more than the *Apply*() function with the *minimum* operator [44], but with a small alteration in order to update the pointer array. In each iteration, we compare two columns to compute the shortest path cost. The first column is the k-th column of the C_{FW} array and the second column is the transpose of the current d_W vector. To compute the "minimum" we are basically comparing each row of the first column with the corresponding row of the second column. For each row, if the minimum value originates from the k-th column of the C_{FW} array, we pick the pointer from the k-th column of P. Otherwise, the pointer originates from the corresponding row of the P_W transpose vector, i.e. from itself.

A-2 Non-deterministic Set-destination Shortest Path Algorithm

Algorithm 12 illustrates the ADD-Implementation of the set-destination shortest path algorithm for non-deterministic systems. We initialize the shortest path cost d_W and the pointer map P_W according to the algorithm 5, using the internalization function in line 2. The priority queue, which basically returns in each iteration the state $x = \min\{\mathsf{d}_W(x) \mid x \in Q\}$, is also initialized in the beginning (line 3).

Due to the nature of the data structure we are using, it is convenient to merge the two operators X_S and U_S into the function *operatorXUsr*(). This function is basically searching for all the states that guarantee a transition to R. These states can be considered as "candidate states" for the set R and are the states for which we apply the *Relax*() function. Of course, since *operatorXUsr*() is also implementing the U_S operator, we also store the inputs that are responsible for a transition to R.

Algorithm 11 $Apply - Min$ Function

Description: Given two pairs of ADDs, this algorithm computes the minimum of the first pair and constructs a second ADD whose minterms are chosen from a second pair, depending on the origin of the minimum value.

Input: The ADD-pair F and G and the ADD-pair P_f and P_g.

Output: The minimum of the two ADDs F and G.

```
1: function APPLY-MIN($F, G, P_f, P_g$)
2:     if ($F == \infty$) then
3:         $Rslt.min = G$
4:         $Rslt.P = P_g$
5:         return $Rslt$
6:     end if
7:     if ($G == \infty$) then
8:         $Rslt.min = F$
9:         $Rslt.P = P_f$
10:        return $Rslt$
11:    end if
12:    if ($F == G$) then
13:        $Rslt.min = F$
14:        $Rslt.P = P_f$
15:        return $Rslt$
16:    end if
17:    if ($IsConstant(F)$ and $IsConstant(G)$) then
18:        if $F < G$ then
19:            $Rslt.min = F$
20:            $Rslt.P = P_f$
21:        else
22:            $Rslt.min = G$
23:            $Rslt.P = P_g$
24:        end if
25:        return $Rslt$
26:    end if
27:    if ($Cache\_Lookup(min2\_tag, (F, G, P_f, P_g), Rslt)$) then
28:        return $Rslt$
29:    end if
30:    $v = Top\_Var(F, G, P_f, P_g)$
31:    $Rslt = Apply\_Min2(F_v, G_v, P_{fv}, P_{gv})$
32:    $APSP_v = Rslt.min$
33:    $P_v = Rslt.P$
34:    $Rslt = Apply\_Min2(F_{v'}, G_{v'}, P_{fv'}, P_{gv'})$
35:    $APSP_{v'} = Rslt.min$
36:    $P_{v'} = Rslt.P$
37:    $Rslt.APSP = Find\_or\_Create(APSP_v, APSP_{v'})$
38:    $Rslt.P = Find\_or\_Create(P_v, P_{v'})$
39:    $Cache\_Insert(min2\_tag, (F, G, P_f, P_g), Rslt)$
40:    return $Rslt$
41: end function
```

Algorithm 12 Non-deterministic SDSP Algorithm - ADD Implementation

Input: The all-pairs shortest path cost array $APSP$ and pointer array P of a system $S(X, X_0, U, \longrightarrow, Y, H)$ and the target set $W_S \subseteq X$.

Output: The vector d_W containing the shortest path cost value for all $x \in X$ and the pointer array P_W.

```
 1: function ND-SDSP($S, W_S$)
 2:     $Init(d_W, P_W)$
 3:     $Init(PQ)$
 4:
 5:     while ($Q \neq 0$) do
 6:         $x = PQ.pop()$
 7:         if $Restrict(C_{FW}, x) == \infty$ then
 8:             $R = Apply(R, x, plus)$
 9:         end if
10:         $Q = Apply(Q, x, minus)$
11:         $XUr = operatorXUsr()$
12:         $Relax(XU_{S_R})$
13:     end while
14:     return $d_W, P_W$
15: end function
```

More precisely, in line 3, by applying the restrict operator we filter out the states that do not have a transition to R. This is done by first swapping the variable x and x' in the BDD representation of the set R. In the next step (line 7) we filter also the states that belong to the W set, because these states are not needed.

From line 8 to 11 we expose all states that do not satisfy the reachability game, so that we can use this information to construct the XUr BDD. Initially we expose all inputs of these states by computing the intersection of the BDD representing the system with the BDD of line 7. With the newly created BDD we are able to find which states fail to satisfy the reachability game. These states are the intersection of the newly created BDD with all the states that are not in $R \cup W$. As a result of the above operations, we are able to remove all "bad" x and u in line 11 and use this outcome to create the XUr BDD (line 12).

After the point where the XUr BDD is constructed, the $Relax()$ function is called, in order to "relax" all states in X_{S_R}. In line 3 we extract the d_W value for all $x \in Post_u(x')$ for all $x' \in X_{S_R}$ and in line 4 we extract the d_W for $x \in X_{S_R}$ such that $Post_u(x) \in W$. The operation in line 5 is nothing more that the ADD that contains the cost $c(x, u, x')$ for all $x \in X_{S_R}$ and $x' \in Post_u(x)$. The ADD is in the form of $f(x', u, x)$ and thus we have to switch the variables x and x' (line 6). In line 8 we are basically computing with this single operation both the d_W value for both the deterministic and non-deterministic transitions. Of course in the case of the non-deterministic transitions we have to take the maximum d_W value and in case of deterministic transitions the minimum d_W value, if more the one exist. To achieve that we first have to distinguish the deterministic transitions from the non-deterministic ones, which is done by using the functions $getDeterministic(d)$ and $getNonDeterministic(d)$ respectively. This functions will not be analyzed, as they are too complex and fall beyond the scope of this thesis.

Algorithm 13 *operatorXUsr* Function

Input: The BDD of the System S, the target set W, the set Q and the set of the resolved states R.

Output: The BDD containing the set of states X_{S_R} and the set of inputs U_{S_R}.

```
1: function OPERATORXUSR($S, W, Q, R$)
2:     $Q_{swpd} = SwapVariables(Q, x, x')$
3:     $R_{swpd} = SwapVariables(R, x, x')$
4:     $S_R = Restrict(S, R_{swpd})$
5:     $Sxu_R = KeepXUvariables(S_R)$
6:
7:     $S_{RnW} = Apply(Sxu_R, Apply(W, not), and)$
8:     $Su_{RnW} = Apply(S, S_{RnW}, and)$
9:
10:    $S_{RG} = Apply(Su_{RnW}, Apply(R_{swpd}, not), and)$
11:    $Sxu_{RG} = KeepXUvariables(S_{RG})$
12:    $XU_{S_R} = Apply(Apply(S_{RnW}, Apply(Sxu_{RG}, not), and), Apply(R, not), and)$
13:    return $XU_{S_R}$
14: end function
```

As a next step, we iterate over all states in X to check whether a state belongs to the set X_{S_R} (lines 17 - 23) and if so, we check whether the state has deterministic and/or "valid" non-deterministic transitions (line 27). In lines 27 to 47 we check for each $u \in U_{S_R}$ what is the minimum d_W in case of deterministic transitions and the minimum - maximum d_W for non-deterministic transitions. After we have determined that, we proceed with lines 49 to 69 to compute the actual shortest path estimate d_W and update the pointer map P_W for all the states in X_{S_R}.

Algorithm 14 *Relax* Function

Input: The BDD containing the set X_{S_R} and U_{S_R}.

```
1:  function RELAX($XUr$)
2:      $XUsr_{swpd} = SwapVariables(x, x')$
3:      $D_{W_x} = Apply(XUsr, C_{FW}, and)$
4:      $D_{W_{x'}} = Apply(XUsr_{swpd}, C_{FW}, and)$
5:      $c_{swpd} = Apply(XUsr_{swpd}, C, and)$
6:      $c = SwapVariables(x, x')$
7:
8:      $d = Apply(c_{swpd}, D_{W_{x'}})$
9:      $d_{det} = getDeterministic(d)$
10:     $d_{ndet} = getNonDeterministic(d)$
11:
12:     for ($k = 1$ to $n$) do
13:         $minterm_k = createMinterm(x, k)$
14:         $C_{det} = Restrict(d_{det}, minterm_k)$
15:         $C_{ndet} = Restrict(d_{ndet}, minterm_k)$
16:
17:         if ($C_{det} == \infty$) then
18:             $deterministic \leftarrow false$
19:         end if
20:         if ($C_{ndet} == \infty$) then
21:             $non\_deterministic \leftarrow false$
22:         end if
23:         if (!$deterministic$ and !$non\_deterministic$) then
24:             continue
25:         end if
26:
27:         if ($deterministic$ and $non\_deterministic$) then
28:             $Cdet_{min} = FindMin(C_{det})$
29:             $Cndet_{max} = FindMax(C_{ndet})$
30:             if ($Cdet_{min} < Cndet_{max}$) then
31:                 $DC_W = Cdet_{min}$
32:             else if ($Cdet_{min} == Cndet_{max}$) then
33:                 $DC_W = Cdet_{min}$
34:                 $ndet\_input \leftarrow true$
35:             else
36:                 $DC_W = Cndet_{max}$
37:                 $ndet\_input \leftarrow true$
38:                 $deterministic \leftarrow false$
39:             end if
```

Algorithm 13 *Relax* Function (continued)

```
        else if ($non\_deterministic$) then
            $ndet\_input \leftarrow true$
            $Cndet_{max} = FindMax(C_{ndet})$
            $DC_W = Cndet_{max}$
        else
            $Cdet_{min} = FindMin(C_{det})$
            $DC_W = Cdet_{min}$
        end if

        if ($Restrict(D_{W_x}, minterm_k) \geq DC_W$) then
            if ($deterministic$) then
                $validInput = getTransitionFromValue(C_{det}, Cdet_{min})$
                $validTransition = Ite(minterm_x, validInput, bdd_zero)$
                $D_W = Apply(D_W, Apply(minterm_x, DC_W, times), plus)$
                $P_W = Apply(P_W, validTransition, plus)$
            end if
            if ($non\_deterministic$ and $ndet\_input$) then
                $validInput = getTransitionFromValue(C_{ndet}, Cndet_{max})$
                $validTransition = Ite(minterm_x, validInput, bdd_zero)$
                if ($!deterministic$) then
                    $D_W = Apply(D_W, Apply(minterm_x, not), and)$
                    $P_W = Apply(P_W, Apply(minterm_x, not), and)$
                    $D_W = Apply(D_W, Apply(minterm_x, DC_W, times), plus)$
                end if
                $P_W = Apply(P_W, validTransition, plus)$
            end if
            $deterministic \leftarrow false$
            $non\_deterministic \leftarrow false$
            $ndet\_input \leftarrow false$
        end if
    end for
    return
end function
```

Bibliography

[1] P. Tabuada, *Verification and control of hybrid systems: a symbolic approach.* Springer, 2009.

[2] M. Mazo Jr, A. Davitian, and P. Tabuada, “Pessoa: A tool for embedded controller synthesis,” in *Computer Aided Verification*, pp. 566–569, Springer, 2010.

[3] B. Donald, P. Xavier, J. Canny, and J. Reif, “Kinodynamic motion planning,” *Journal of the ACM (JACM)*, vol. 40, no. 5, pp. 1048–1066, 1993.

[4] V. Blondel and J. N. Tsitsiklis, “Np-hardness of some linear control design problems,” *SIAM Journal on Control and Optimization*, vol. 35, no. 6, pp. 2118–2127, 1997.

[5] J. C. Willems, “Models for dynamics,” in *Dynamics reported*, pp. 171–269, Springer, 1989.

[6] J. C. Willems, “Paradigms and puzzles in the theory of dynamical systems,” *Automatic Control, IEEE Transactions on*, vol. 36, no. 3, pp. 259–294, 1991.

[7] J. van Schuppen, “Equivalences of discrete-event systems and of hybrid systems,” in *Open problems in mathematical systems and control theory*, pp. 251–257, Springer, 1999.

[8] R. Milner, *Communication and concurrency.* Prentice-Hall, Inc., 1989.

[9] E. Haghverdi, P. Tabuada, and G. Pappas, “Bisimulation relations for dynamical and control systems,” *Electronic Notes in Theoretical Computer Science*, vol. 69, pp. 120–136, 2003.

[10] A. Girard and G. J. Pappas, “Approximation metrics for discrete and continuous systems,” *Automatic Control, IEEE Transactions on*, vol. 52, no. 5, pp. 782–798, 2007.

[11] P. Tabuada, “An approximate simulation approach to symbolic control,” *Automatic Control, IEEE Transactions on*, vol. 53, no. 6, pp. 1406–1418, 2008.

[12] M. Ying and M. Wirsing, “Approximate bisimilarity,” in *Algebraic Methodology and Software Technology*, pp. 309–322, Springer, 2000.

[13] T. A. Henzinger, R. Majumdar, and V. S. Prabhu, "Quantifying similarities between timed systems," in *Formal Modeling and Analysis of Timed Systems*, pp. 226–241, Springer, 2005.

[14] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking.* MIT press, 1999.

[15] N. Piterman, A. Pnueli, and Y. SaâĂŹar, "Synthesis of reactive (1) designs," in *Verification, Model Checking, and Abstract Interpretation*, pp. 364–380, Springer, 2006.

[16] P. Tabuada and G. J. Pappas, "Linear time logic control of discrete-time linear systems," *Automatic Control, IEEE Transactions on*, vol. 51, no. 12, pp. 1862–1877, 2006.

[17] B. Yordanov, J. Tumova, I. Cerna, J. Barnat, and C. Belta, "Temporal logic control of discrete-time piecewise affine systems," *Automatic Control, IEEE Transactions on*, vol. 57, no. 6, pp. 1491–1504, 2012.

[18] M. Broucke, M. D. Di Benedetto, S. Di Gennaro, and A. Sangiovanni-Vincentelli, "Theory of optimal control using bisimulations," in *Hybrid Systems: Computation and Control*, pp. 89–102, Springer, 2000.

[19] M. Broucke, M. D. Di Benedetto, S. Di Gennaro, and A. Sangiovanni-Vincentelli, "Efficient solution of optimal control problems using hybrid systems," *SIAM journal on control and optimization*, vol. 43, no. 6, pp. 1923–1952, 2005.

[20] L. Grüne and O. Junge, "Global optimal control of perturbed systems," *Journal of Optimization Theory and Applications*, vol. 136, no. 3, pp. 411–429, 2008.

[21] T. Paschedag, M. Fall, and O. Stursberg, "Optimizing hybrid control trajectories by model abstraction and refinement," in *Analysis and Design of Hybrid Systems*, vol. 3, pp. 298–303, 2009.

[22] Y. Tazaki and J.-i. Imura, "Multiresolution discrete abstraction for optimal control," in *Decision and Control (CDC), 2010 49th IEEE Conference on*, pp. 5905–5910, IEEE, 2010.

[23] M. Mazo Jr and P. Tabuada, "Symbolic approximate time-optimal control," *Systems & Control Letters*, vol. 60, no. 4, pp. 256–263, 2011.

[24] Y. Tazaki and J.-i. Imura, "Finite abstractions of discrete-time linear systems and its application to optimal control," in *17th IFAC world congress*, pp. 10201–10206, 2008.

[25] A. Girard, "Synthesis using approximately bisimilar abstractions: time-optimal control problems," in *Decision and Control (CDC), 2010 49th IEEE Conference on*, pp. 5893–5898, IEEE, 2010.

[26] F. de Roo and M. Mazo Jr, "On symbolic optimal control via approximate simulation relations," in *Conference on Decision and Control (CDC) 2013 (To appear)*, 2013.

[27] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes," *SIAM journal on control and optimization*, vol. 25, no. 1, pp. 206–230, 1987.

[28] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[29] R. W. Floyd, “Algorithm 97: shortest path,” *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[30] R. E. Bryant, “Graph-based algorithms for boolean function manipulation,” *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.

[31] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, “Algebric decision diagrams and their applications,” *Formal methods in system design*, vol. 10, no. 2-3, pp. 171–206, 1997.

[32] M. Fujita, P. C. McGeer, and J.-Y. Yang, “Multi-terminal binary decision diagrams: An efficient data structure for matrix representation,” *Formal methods in system design*, vol. 10, no. 2-3, pp. 149–169, 1997.

[33] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer, “Specify, compile, run: Hardware from psl,” *Electronic Notes in Theoretical Computer Science*, vol. 190, no. 4, pp. 3–16, 2007.

[34] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. M. Sentovich, and K. Suzuki, “Synthesis of software programs for embedded control applications,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 18, no. 6, pp. 834–849, 1999.

[35] G. Chartrand, L. Lesniak, and P. Zhang, *Graphs and digraphs.* CRC Press, 2011.

[36] J. Bang-Jensen and G. Gutin, *Digraphs: theory, algorithms and applications.* Springer, 2009.

[37] G. Brassard and P. Bratley, *Fundamentals of algorithmics*, vol. 524. Prentice Hall Englewood Cliffs, 1996.

[38] R. Bellman, “Dynamic programming, princeton univ,” *Prese Princeton, 19S7*, 1957.

[39] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms.* MIT press, 2001.

[40] S. Warshall, “A theorem on boolean matrices,” *Journal of the ACM (JACM)*, vol. 9, no. 1, pp. 11–12, 1962.

[41] C.-Y. Lee, “Representation of switching circuits by binary-decision programs,” *Bell System Technical Journal*, vol. 38, no. 4, pp. 985–999, 1959.

[42] S. B. Akers, “Binary decision diagrams,” *Computers, IEEE Transactions on*, vol. 100, no. 6, pp. 509–516, 1978.

[43] E. M. Clarke, M. Fujita, P. C. McGeer, K. McMillan, J. C. Yang, and X. Zhao, “Multi-terminal binary decision diagrams: An efficient data structure for matrix representation,” 1993.

[44] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, “Algebraic decision diagrams and their applications,” in *Computer-Aided Design, 1993. ICCAD-93. Digest of Technical Papers., 1993 IEEE/ACM International Conference on*, pp. 188–191, IEEE, 1993.

[45] R. Bahar, G. Hachtel, A. Pardo, M. Poncino, and F. Somenzi, "An add-based algorithm for shortest path back-tracing of large graphs," in *VLSI, 1994. Design Automation of High Performance VLSI Systems. GLSV'94, Proceedings., Fourth Great Lakes Symposium on*, pp. 248–251, IEEE, 1994.

[46] F. M. Brown, *Boolean reasoning: the logic of Boolean equations.* Courier Dover Publications, 2003.

[47] E. A. Emerson and J. Y. Halpern, "“sometimes” and “not never” revisited: on branching versus linear time temporal logic," *Journal of the ACM (JACM)*, vol. 33, no. 1, pp. 151–178, 1986.

[48] A. Pnueli, "The temporal logic of programs," in *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pp. 46–57, IEEE, 1977.

[49] O. Kupferman and M. Y. Vardi, "Model checking of safety properties," *Formal Methods in System Design*, vol. 19, no. 3, pp. 291–314, 2001.

[50] S. Pettie, "A new approach to all-pairs shortest paths on real-weighted graphs," *Theoretical Computer Science*, vol. 312, no. 1, pp. 47–74, 2004.

[51] Y. Han, V. Pan, and J. Reif, "Efficient parallel algorithms for computing all pair shortest paths in directed graphs," in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pp. 353–362, ACM, 1992.

[52] S. Jukna, *Boolean function complexity: advances and frontiers*, vol. 27. Springerverlag Berlin Heidelberg, 2012.

Glossary

List of Acronyms

BDD	Binary Decision Diagram
ADD	Algebraic Decision Diagram
ROBDD	Reduced Ordered Binary Decision Diagram
MTBDD	Multi-Terminal Binary Decision Diagram
SSSP	Single-Source Shortest Path
APSP	All-Pairs Shortest Path
SDSP	Set-Destination Shortest Path
LTL	Linear Temporal Logic
CTL	Computation Tree Logic