Vassar College | URSI Symposium | 2021

TEMPORAL REASONING ALGORITHMS

JESSICA FLEISCHER, FRANCESCA LUCCHETTI, MUHTASIM MIRAZ, BENJAMIN PRUD'HOMME
LUKE HUNSBERGER

INTRODUCTION

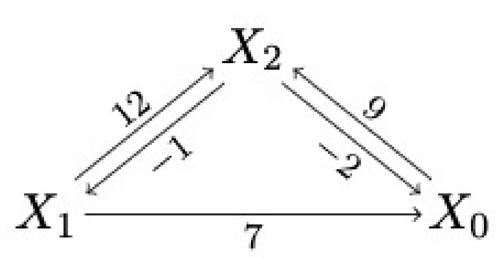
Temporal networks are data structures for representing and reasoning about temporal constraints on real events and activities. The most basic kind of temporal network is a Simple Temporal Network (STN) which can accommodate such constraints as release times, deadlines, precedence constraints, and duration constraints. The primary goal of this project was to create a public repository that will include implementations of several useful algorithms for manipulating STNs. Researchers interested in using STNs as a temporal reasoning tool will be able to use this repository for their needs.

SIMPLE TEMPORAL NETWORKS

A **Simple Temporal Network (STN)** has two main components: a set $\mathcal{T}$ of real-valued variables called **time-points,** and a set $\mathcal{C}$ of **edges** imposing constraints on the distance between pairs of time-points.
These constraints have the form $Y - X \leq \delta$, where X and Y are time-points, and δ is a real number.
The **Simple Temporal Problem** is that of determining if an STN is **consistent** - that is, whether there is a set of values for all time-points that satisfies all edge constraints.
A consistent STN has an $n \times n$ matrix **D** called the **distance matrix.**
For each pair of time-points X and Y, the entry $D(X,Y)$ contains the length of the shortest path from X to Y in the STN graph.

$$\begin{aligned} \mathcal{T} &= \{X_0, X_1, X_2\} \\ \mathcal{C} &= \{X_2 - X_1 \leq 12, \quad X_2 - X_0 \leq 9, \\ & \quad X_1 - X_2 \leq -1, \quad X_0 - X_2 \leq -2, \\ & \quad X_0 - X_1 \leq 7\} \end{aligned}$$

(a) A sample STN



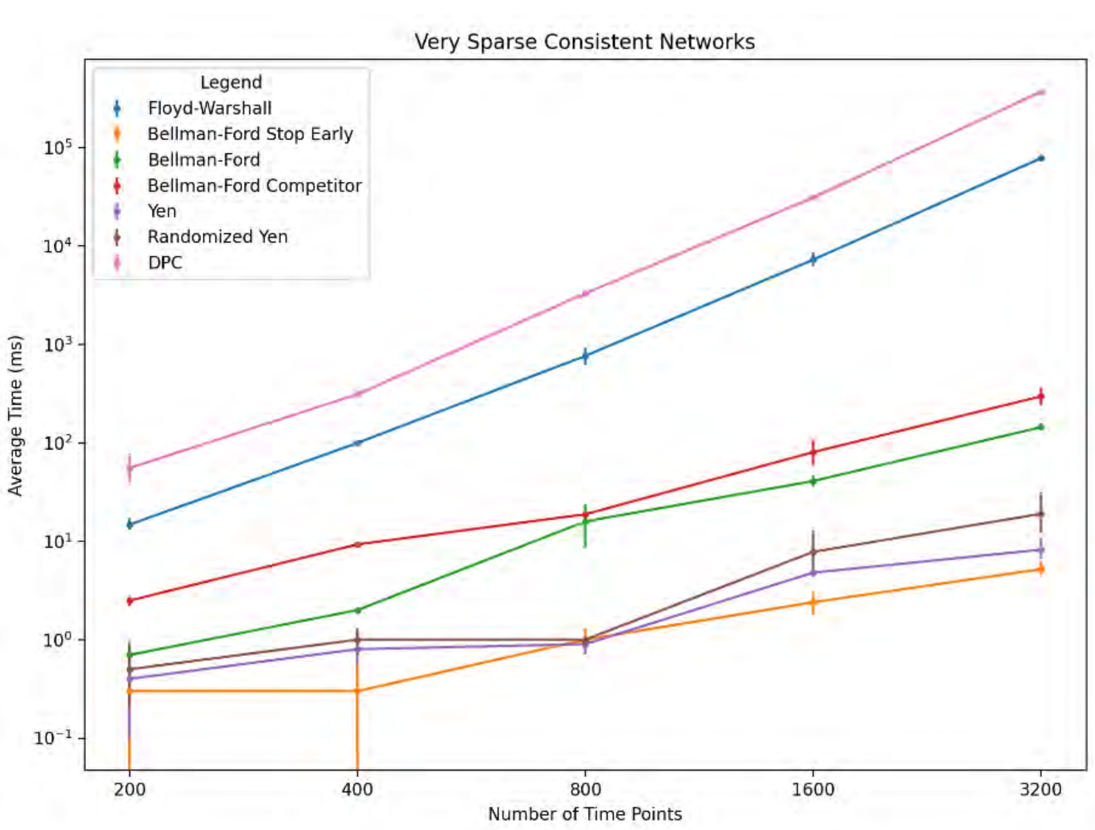
(b) Its graph

D	X_0	X_1	X_2
X_0	0	8	9
X_1	7	0	12
X_2	-2	-1	0

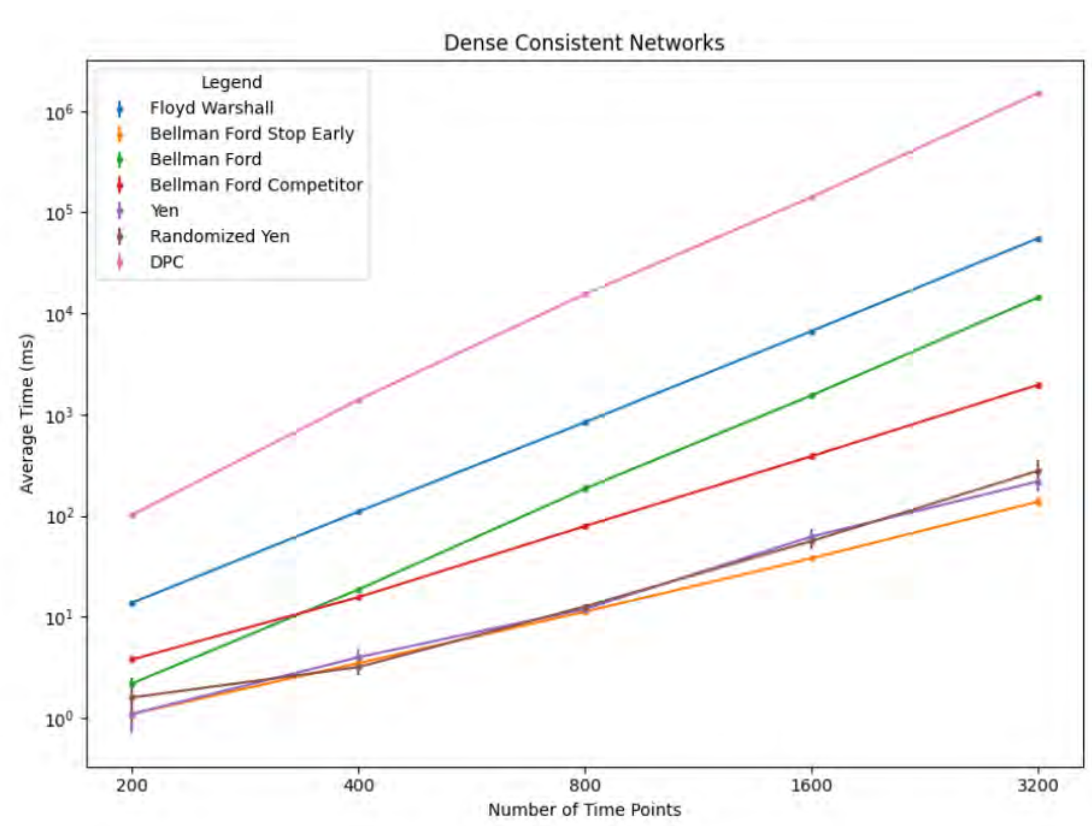
(c) its distance matrix

METHODOLOGY

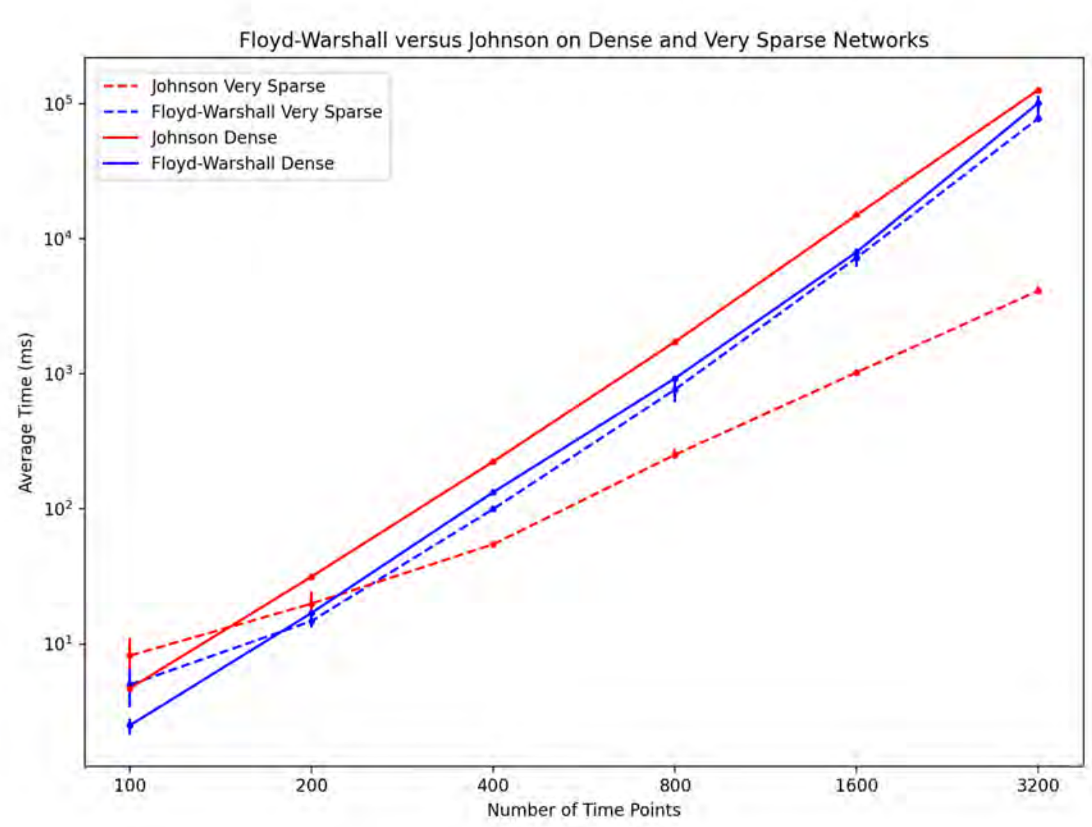
We used Java to code all of the algorithms and a random STN generator. In order to test our code, we created benchmarks of random STNs, varying the number of time-points. For each time-point amount, we varied the number of edges to be very sparse, sparse, or dense and whether it is consistent or inconsistent. We then used the benchmark STNs to test and compare the timing of the algorithms. In order to do comparisons, we grouped some of the algorithms into different groups: consistency checkers, distance matrix generators, shortest path algorithms, and solution generators. Many of these groups had some overlapping algorithms. For example, Bellman-Ford is a shortest path algorithm, a consistency checker, and a solution generator. We then used Python to generate graphs comparing the amount of time that different algorithms took to run on STNs with different densities, consistency statuses, and numbers of time-points.



(d) Very Sparse Consistency Checking



(e) Dense Consistency Checking



(f) Floyd-Warshall vs. Johnson

RESULTS

The consistency-checking algorithms for solving the **Simple Temporal Problem** are displayed in graphs (d) and (e). On both very sparse and dense networks, a "stop early" version of Bellman-Ford is our fastest algorithm, followed closely by Yen and Randomized Yen, which have similar performances due to the randomized generation of the networks. A surprising result is that the Bellman-Ford competitor based on the AddToFeasible algorithm performs better than Bellman-Ford itself on dense networks of around or over 400 time-points. Another surprising result was the performance of Directional Path Consistency (DPC), which surpassed even Floyd-Warshall's complexity of $O(n^3)$. We concluded that this was a consequence of the new edges inserted by the DPC algorithm into the network. As we cannot predict how many edges DPC will add into a random network, the algorithm may perform better or worse than expected.
Graph (f) shows a comparison of the performances of Floyd-Warshall's and Johnson's algorithms on very sparse and dense networks. Both of these algorithms generate distance matrices, with the difference that Johnson works by combining two other algorithms, Dijkstra and Bellman-Ford. We calculate the complexity of Johnson's algorithm from the n iterations of Dijkstra within its body: $O(n^2 \log n + mn)$. In sparse graphs where the number of edges m is much less than the maximum value n^2, Johnson's algorithm should run faster than Floyd-Warshall's. This is confirmed by our results, which show that Johnson performs worse than Floyd-Warshall on denser graphs.

ALGORITHMS

Floyd-Warshall
Generate Solution
DpcDispatch
Bellman-Ford
Bellman-Ford Ext
Dijkstra
Dijkstra Ext
Johnson
Bellman-Ford Ext
Dijkstra
Dijkstra Ext
Johnson
Yen
Randomized Yen
AddToFeasible
AddToFeasible Ext
DPC
Chleq
BFCompetitor

CITATIONS

Floyd, 1962; Warshall 1962; Bellman, 1958; Ford & Fulkerson, 1962; Yen, 1970; Johnson, 1977; Ramalingam et al., 1999; Dechter et al., 1991; Planken, 2013; Chleq, 1995; Cormen et al., 2001