Chapter 1

Risk Propagation

Risk propagation is a message-passing algorithm that estimates an individual's infection risk by considering their demographics, symptoms, diagnosis, and contact with others. Formally, a *risk score* s_t is a timestamped infection probability where $s \in [0, 1]$ and $t \in \mathbb{N}$ is the time of its computation. Thus, an individual with a high risk score is likely to test positive for the infection and poses a significant health risk to others. There are two types of risk scores: *symptom scores*, or prior infection probabilities, which account for an individual's demographics, symptoms, and diagnosis (Menni et al., 2020); and *exposure scores*, or posterior infection probabilities, which incorporate the risk of direct and indirect contact with others.

Given their recent risk scores and contacts, an individual's exposure score is derived by marginalizing over the joint infection probability distribution. Naively computing this marginalization scales exponentially with the number of variables (i.e., individuals). To circumvent this intractability, the joint dis-

tribution is modeled as a factor graph, and an efficient message-passing procedure is employed to compute the marginal probabilities with a time complexity that scales linearly in the number of factor nodes (i.e., contacts).

Let $G = (X, F, E)$ be a *factor graph* where X is the set of variable nodes, F is the set of factor nodes, and E is the set of edges incident between them (Kschischang et al., 2001).

A *variable node* $x : \Omega \to \{0, 1\}$ is a random variable that represents the infection status of an individual, where the sample space is $\Omega = \{\textit{healthy}, \textit{infected}\}$ and

$$x(\omega) = \begin{cases} 0 & \text{if } \omega = \textit{healthy} \\ 1 & \text{if } \omega = \textit{infected}. \end{cases}$$

Thus, $p_t(x_i) = s_t$ is a risk score of the i-th individual.

A *factor node* $f : X \times X \to [0, 1]$ defines the transmission of infection risk between two contacts. Specifically, contact between the i-th and j-th individual is represented by the factor node $f(x_i, x_j) = f_{ij}$, which is adjacent to the variable nodes x_i, x_j. This work and Ayday et al. (2021) assume risk transmission is a symmetric function, $f_{ij} = f_{ji}$. However, it may be extended to account for an individual's susceptibility and transmissibility such that $f_{ij} \neq f_{ji}$. Figure 1.1 depicts a factor graph that reflects the domain constraints.

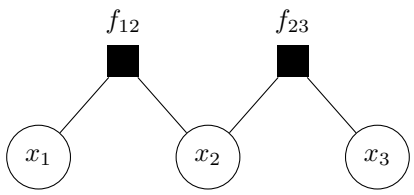

Figure 1.1: A factor graph of 3 variable nodes and 2 factor nodes.

1.1 Synchronous Risk Propagation

Ayday et al. (2021) first proposed risk propagation as a synchronous, iterative message-passing algorithm that uses the factor graph to compute exposure scores. The first input to Risk-Propagation is the set family S, where

$$S_i = \{\, s_t \mid \tau - t < T_s \,\} \in S \tag{1.1}$$

is the set of recent risk scores of the i-th individual. The second input to Risk-Propagation is the contact set

$$C = \{\, (i, j, t) \mid i \neq j, \tau - t < T_c \,\} \tag{1.2}$$

such that (i, j, t) is the *most recent* contact between the i-th and j-th individual that occurred from time t until at least time $t+\delta$, where $\delta \in \mathbb{N}$ is the *minimum contact duration*[1]. Naturally, risk scores and contacts have finite relevance, so (1.1) and (1.2) are constrained by the *risk score expiry* $T_s \in \mathbb{N}$ and the *contact expiry* $T_c \in \mathbb{N}$, respectively. The *reference time* $\tau \in \mathbb{N}$ defines the relevance of the inputs and is assumed to be the time at which Risk-Propagation is invoked. For notational simplicity in Risk-Propagation, let X be a set. Then $\max X = 0$ if $X = \emptyset$.

[1] While Ayday et al. (2021) require contact over a δ-contiguous period of time, the Centers for Disease Control and Prevention (2021) account for contact over a 24-hour period.

1.1.1 Variable Messages

The current exposure score of the i-th individual is defined as $\max S_i$. Hence, a *variable message* $\mu_{ij}^{(n)}$ from the variable node x_i to the factor node f_{ij} during the n-th iteration is the set of maximal risk scores $R_i^{(n-1)}$ from the previous $n-1$ iterations that were not derived by f_{ij}. In this way, risk propagation is reminiscent of the max-sum algorithm; however, risk propagation aims to maximize *individual* marginal probabilities rather than the joint distribution (Bishop, 2006, pp. 411–415).

1.1.2 Factor Messages

A *factor message* $\lambda_{ij}^{(n)}$ from the factor node f_{ij} to the variable node x_j during the n-th iteration is an exposure score of the j-th individual that is based on interacting with those at most $n-1$ degrees separated from the i-th individual. This population is defined by the subgraph induced in G by

$$\{\, v \in X \cap F \setminus \{x_j, f_{ij}\} \mid d(x_i, v) \leq 2(n-1) \,\},$$

where $d(u, v)$ is the distance between the nodes u, v. The computation of a factor message assumes the following.

1. Contacts have a nondecreasing effect on an individual's exposure score.

2. A risk score s_t is *relevant* to the contact (i, j, t_{ij}) if $t < t_{ij} + \beta$, where $\beta \in \mathbb{N}$ is a *time buffer* that accounts for the incubation period, along with the delayed reporting of symptom scores and contacts. The expression

$t_{ij} + \beta$ is called the *buffered contact time*.

3. Risk transmission between contacts is incomplete. Thus, a risk score decays exponentially along its transmission path in G at a rate of $\log \alpha$, where $\alpha \in (0, 1)$ is the *transmission rate*. Figure 1.2 visualizes this decay.

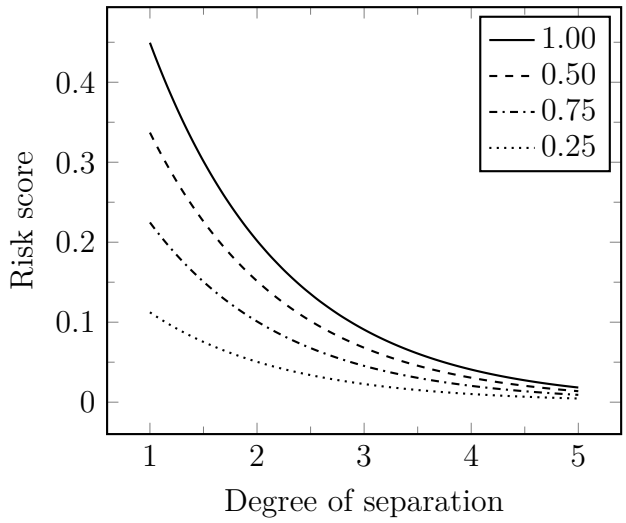

Figure 1.2: Exponential decay of risk scores.

To summarize, a factor message $\lambda_{ij}^{(n)}$ is the maximum relevant risk score in the variable message $\mu_{ij}^{(n)}$ (or 0) that is scaled by the transmission rate α.

Ayday et al. (2021) assume that the contact set C may contain (1) multiple contacts between the same two individuals and (2) *invalid* contacts, or those lasting less than δ time. However, these assumptions introduce unnecessary complexity. Regarding assumption 1, suppose the i-th and j-th individual come into contact m times such that $t_k < t_\ell$ for $1 \leq k < \ell \leq m$. Let Λ_k be the set of relevant risk scores, according to the contact time t_k, where

$$\Lambda_k = \{\, \alpha s_t \mid s_t \in \mu_{ij}^{(n)}, t < t_k + \beta \,\}.$$

Then $\Lambda_k \subseteq \Lambda_\ell$ if and only if $\max \Lambda_k \leq \max \Lambda_\ell$. Therefore, only the most recent contact time t_m is required to compute the factor message $\lambda_{ij}^{(n)}$. With respect to assumption 2, there are two possibilities.

1. If an individual has at least one valid contact, then their exposure score is computed over the subgraph induced in G by their contacts that define the neighborhood N_i of the variable node x_i.

2. If an individual has no valid contacts, then their exposure score is $\max S_i$ or 0, if all of their previously computed risk scores have expired.

In either case, a set C containing only valid contacts implies fewer factor nodes and edges in the factor graph G. Consequently, the complexity of Risk-Propagation is reduced by a constant factor since fewer messages must be computed.

1.1.3 Termination

To detect convergence, the normed difference between the current and previous exposure scores is compared to the threshold $\epsilon \in \mathbb{R}$. Note that $\mathbf{r}^{(n)}$ is the vector of exposure scores in the the n-th iteration such that $r_i^{(n)}$ is the i-th component of $\mathbf{r}^{(n)}$. The ℓ^1 and ℓ^∞ norms are sensible choices for detecting convergence. Ayday et al. (2021) use the ℓ^1 norm, which ensures that an individual's exposure score changed by at most ϵ after the penultimate iteration.

RISK-PROPAGATION(S, C)

1: $(X, F, E) \leftarrow$ CREATE-FACTOR-GRAPH(C)
2: $n \leftarrow 1$
3: **for each** $x_i \in X$
4: $\quad R_i^{(n-1)} \leftarrow$ top K of S_i
5: $\quad r_i^{(n-1)} \leftarrow \max R_i^{(n-1)}$
6: $\quad r_i^{(n)} \leftarrow \infty$
7: **while** $\|\mathbf{r}^{(n)} - \mathbf{r}^{(n-1)}\| > \epsilon$
8: $\quad$ **for each** $\{x_i, f_{ij}\} \in E$
9: $\quad\quad \mu_{ij}^{(n)} \leftarrow R_i^{(n-1)} \setminus \{\, \lambda_{ji}^{(k)} \mid k \in [1\,..\,n-1] \,\}$
10: $\quad$ **for each** $\{x_i, f_{ij}\} \in E$
11: $\quad\quad \lambda_{ij}^{(n)} \leftarrow \max\{\, \alpha s_t \mid s_t \in \mu_{ij}^{(n)}, t < t_{ij} + \beta \,\}$
12: $\quad$ **for each** $x_i \in X$
13: $\quad\quad R_i^{(n)} \leftarrow$ top K of $\{\, \lambda_{ji}^{(n)} \mid f_{ij} \in N_i \,\}$
14: $\quad$ **for each** $x_i \in X$
15: $\quad\quad r_i^{(n-1)} \leftarrow r_i^{(n)}$
16: $\quad\quad r_i^{(n)} \leftarrow \max R_i^{(n)}$
17: $\quad n \leftarrow n + 1$
18: **return** $\mathbf{r}^{(n)}$

1.2 Asynchronous Risk Propagation

RISK-PROPAGATION is an *offline algorithm*, because it requires the contact and health information of all individuals as input. As Ayday et al. (2021) note, this centralization of personal data does not preserve privacy. RISK-

PROPAGATION is also inefficient. Most exposure scores are not likely to change across frequent invocations, which implies communication overhead and computational redundancy. To mitigate this inefficiency, Ayday et al. (2020) suggest running RISK-PROPAGATION once or twice per day. However, this causes substantial delay in updating individuals' exposure scores. In the face of a pandemic, timely information is essential for individual and collective health. In brief, RISK-PROPAGATION offers proof of concept, but is not viable for real-world application.

To address the limitations of RISK-PROPAGATION, Ayday et al. (2021) propose decentralizing the factor graph such that the processing entity (e.g., mobile device or "personal cloud") associated with the i-th individual maintains the state of the i-th variable node and the neighboring factor nodes. Ayday et al. (2021) do not leave key questions unanswered.

1. Is message passing synchronous or asynchronous?
2. How does message passing terminate?
3. Are any optimizations utilized to reduce communication cost?
4. How do processing entities exchange messages over the network?
5. How private is decentralized risk propagation?

What Ayday et al. (2021) propose is a decentralized communication protocol for exchanging information about infection risk. Such message passing amongst stateful processing entities can be formally described using the *actor model* (**??**).

1.2.1 Actor Behavior

The CREATE-ACTOR operation initializes an actor (Agha, 1985). An actor a has the following attributes.

- $a.exposure$: the current exposure score of the individual that this actor represents. This attribute is either a symptom score, a risk score sent by another actor, or the null risk score (see NULL-RISK-SCORE).

- $a.contacts$: a *dictionary* (Appendix D) of contacts. In the context of an actor, a contact is a *proxy* (Gamma et al., 1995) of the actor that represents an individual with which the individual represented by this actor was physically proximal. That is, if the i-th individual interacted with the j-th individual, then $a_i.contacts$ contains a contact c such that $c.key = c.name$ is a name of the j-th actor and $c.t$ is the most recent time of contact. This attribute extends the concept of *actor acquaintances* (Agha, 1985; Hewitt, 1977; Hewitt and Baker, 1977) to be time-varying.

- $a.scores$: a dictionary of exposure scores, such that $s.key$ for an exposure score s is the time interval during which $a.exposure = s$. The null risk score is returned for queries in which the dictionary does not contain a risk score with a key that intersects with the given query interval. Figure 1.3 depicts a hypothetical step function that $a.scores$ represents.

Note that CREATE-ACTOR does not specify a name for the actor. This allows the actor to have multiple names and for them to be specified "out-of-band."

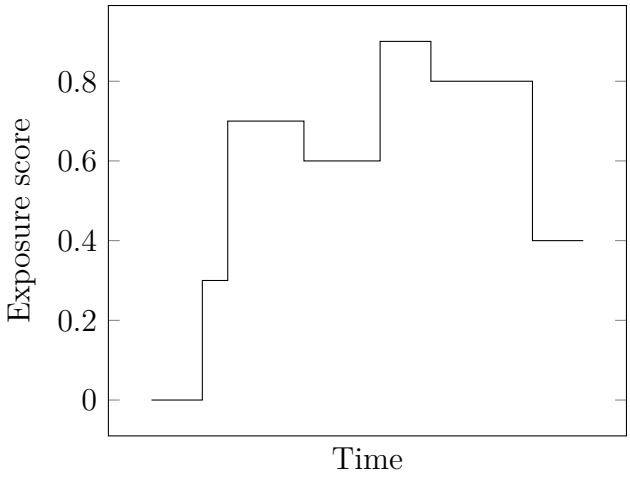

Figure 1.3: Historical exposure scores of an actor.

NULL-RISK-SCORE

1: $s.value \leftarrow 0$

2: $s.t \leftarrow 0$

3: $s.sender \leftarrow$ NIL

4: **return** s

CREATE-ACTOR

1: $a.contacts \leftarrow \emptyset$

2: $a.scores \leftarrow \emptyset$

3: $a.exposure \leftarrow$ NULL-RISK-SCORE

4: **return** a

The interface of an actor is primarily defined by two types of messages: contacts and risk scores. As with Section 1.1, contacts and risk scores have finite relevance. Let the *time-to-live* (TTL) of a message be the remaining time of its relevance. The reference time τ is assumed to be the current time.

The HANDLE-RISK-SCORE operation defines the actions an actor performs

```
RISK-SCORE-TTL(s)
1: return T_s − (τ − s.t)
```

```
CONTACT-TTL(c)
1: return T_c − (τ − c.t)
```

upon receiving a risk score. The key initially associated with the risk score is the time interval for which it is relevant. For the dictionary $a.scores$, MERGE preserves the mapping invariant defined above such that risk scores are ordered first by value and then by time. Thus, $s.key \subseteq [s.t, s.t + T_s)$ for all exposure scores in $a.scores$. The UPDATE-EXPOSURE-SCORE operation describes how $a.exposure$ is updated. The dictionary $a.contacts$ is assumed to only contain unexpired contacts. Additional context is needed before specifying APPLY-RISK-SCORE in detail. For now, it is sufficient to know that the operation uses the risk score to update the state of a contact.

```
HANDLE-RISK-SCORE(a, s)
1: if RISK-SCORE-TTL(s) > 0
2:     s.key ← [s.t, s.t + T_s)
3:     MERGE(a.scores, s)
4:     UPDATE-EXPOSURE-SCORE(a, s)
5:     for each c ∈ a.contacts
6:         APPLY-RISK-SCORE(a, c, s)
```

```
UPDATE-EXPOSURE-SCORE(a, s)
1: if a.exposure.value < s.value
2:     a.exposure ← s
3: else if RISK-SCORE-TTL(a.exposure) ≤ 0
4:     a.exposure ← MAXIMUM(a.scores)
```

For the moment, assume that APPLY-RISK-SCORE is equivalent to computing a factor message (see Section 1.1). Line 2 indicates that the risk score s is copied and updated.

```
APPLY-RISK-SCORE(a, c, s)
1: if c.t + β > s.t
2:     s'.value ← α · s.value
3:     SEND(c.name, s')
```

The problem with APPLY-RISK-SCORE is that it causes risk scores to propagate *ad infinitum*. For asynchronous risk propagation to be scalable and cost-efficient, actors should only send risk scores that offer new information to other actors. Unlike RISK-PROPAGATION, a global convergence test is not available to terminate message passing, so it is necessary to define a local condition that determines if a risk score should be sent to another actor.

The objective of sending a risk score to another actor is to update its exposure score. Based on HANDLE-RISK-SCORE, it is only necessary to send an actor risk scores with values greater than its current exposure score. However, an actor is only privy to the risk scores that it sends. Thus, an actor can associate a *send threshold* for a contact that must be exceeded for a risk score

to be sent to the target actor. To permit a trade-off between accuracy and efficiency of asynchronous risk propagation, let the *send coefficient* $\gamma \in \mathbb{R}$ be a scaling factor that is applied to a risk score upon setting the send threshold.

SET-SEND-THRESHOLD(c, s)

1: $s'.value \leftarrow \gamma \cdot s.value$

2: $c.threshold \leftarrow s'$

Below is the definition of APPLY-RISK-SCORE that incorporates this new aspect of message passing. Assuming a finite number of actors, a positive send coefficient guarantees that a risk score has finite propagation.

APPLY-RISK-SCORE(a, c, s)

1: **if** $c.threshold.value < s.value$ **and** $c.t + \beta > s.t$

2: $\quad s'.value \leftarrow \alpha \cdot s.value$

3: $\quad$ SET-SEND-THRESHOLD(c, s')

4: $\quad$ SEND$(c.name, s')$

The SET-SEND-THRESHOLD operation defines *how* the send threshold is updated, but not *when* it should be updated. The UPDATE-SEND-THRESHOLD operation encapsulates this update behavior. The latter predicate of Line 1 stems from the fact that the send threshold is a risk score and thus subject to expiry. The former predicate is more subtle and will be revisited shortly. The MAXIMUM-OLDER-THAN is the same as MAXIMUM with the additional restriction that the interval key intersects with the query interval $(-\infty, c.t + \beta)$. In this way, the returned risk score is always relevant to the contact. As with APPLY-RISK-SCORE, the risk score retrieved from $a.scores$ is also scaled by

the transmission rate and set as the new send threshold.

```
UPDATE-SEND-THRESHOLD(a, c)
1: if c.threshold.value > 0 and RISK-SCORE-TTL(c.threshold) ≤ 0
2:     s ← MAXIMUM-OLDER-THAN(a.scores, c.t + β)
3:     s'.value ← α · s.value
4:     SET-SEND-THRESHOLD(c, s')
```

The UPDATE-SEND-THRESHOLD is invoked during APPLY-RISK-SCORE, so another modification to the operation is defined below.

```
APPLY-RISK-SCORE(a, c, s)
1: UPDATE-SEND-THRESHOLD(a, c)
2: if c.threshold.value < s.value and c.t + β > s.t
3:     s'.value ← α · s.value
4:     SET-SEND-THRESHOLD(c, s')
5:     SEND(c.name, s')
```

Returning to the first predicate on Line 1 of UPDATE-SEND-THRESHOLD, there are two cases in which the send threshold has a value of 0:

1. when initially assigning the send threshold to be the null risk score; and

2. when no key interval in *a.scores* intersects the query interval, and thus the null risk score again is assigned the send threshold.

Suppose the first predicate is omitted from Line 1. Given that UPDATE-SEND-THRESHOLD is the first statement in APPLY-RISK-SCORE, it is possible that the send threshold is set prior to sending the contact a relevant risk

score. In the worst case, this prevents *any* risk score from being sent to the target actor, thus providing the individual associated with target actor a false sense of security that they are not at risk of being infected. From a broader message-passing perspective, updating the send threshold to a non-null risk score *before* applying the risk score received by the actor may introduce a non-trivial amount of inaccuracy. To summarize, updating the send threshold only when its value is nonzero ensures correct message-passing behavior between actors.

Up until this point, the refinements to Apply-Risk-Score have focused on ensuring that message passing terminates and correctly adjusts over time, as risk scores expire. Prior to concluding this section, a message-passing optimization is introduced. Over a given period of time, an actor may receive several risk scores that are then propagated to multiple contacts. Intuitively, rather than sending multiple risk scores, it would be more efficient to send just the final risk score. To increase the likelihood of that this occurs, Apply-Risk-Score can be modified so that a contact "buffers" a single risk score that is intended to be sent to the target actor. Upon receiving a *flush timeout* message, the actor then "flushes" all contacts by sending the buffered message of each contact to the respective target actor. This is also known as *sender-side aggregation* in which the contact is an *aggregator* of risk scores.

The final iteration Apply-Risk-Score integrates sender-side aggregation. Moreover, Handle-Flush-Timeout clarifies the concept of "flushing" a contact. A flush timeout is assumed to be a periodically occurring "self" message.

To conclude the specification of actor behavior, the Handle-Contact

```
APPLY-RISK-SCORE(a, c, s)
1: UPDATE-SEND-THRESHOLD(a, c)
2: if c.threshold.value < s.value and c.t + β > s.t
3:     s'.value ← α · s.value
4:     SET-SEND-THRESHOLD(c, s')
5:     if c.name ≠ s.sender
6:         c.buffered ← s'
```

```
HANDLE-FLUSH-TIMEOUT(a)
1: for each c ∈ a.contacts
2:     if c.buffered ≠ NIL
3:         SEND(c.name, c.buffered)
4:         c.buffered ← NIL
```

operation indicates how an actor responds when a new contact or contact with a newer contact time is received. Similar to HANDLE-RISK-SCORE, expired contacts are not processed. The MERGE operation for *a.contacts* differs from how its used for *a.scores*. Specifically, if a contact with the same key already exists, its contact time is updated to that of the new contact; all other state of the previous contact is maintained.

1.2.2 Message Reachability

1.2.3 Mobile Crowdsensing

Mobile crowdsensing (MCS) is a "sensing paradigm that empowers ordinary citizens to contribute data sensed or generated from their mobile devices" that

HANDLE-CONTACT(a, c)

1: **if** CONTACT-TTL$(c) > 0$
2: $\quad c.threshold \leftarrow$ NULL-RISK-SCORE
3: $\quad c.buffered \leftarrow$ NIL
4: $\quad c.key \leftarrow c.name$
5: $\quad$ MERGE$(a.contacts, c)$
6: $\quad s \leftarrow$ MAXIMUM-OLDER-THAN$(a.scores, c.t + \beta)$
7: $\quad$ APPLY-RISK-SCORE(a, c, s)

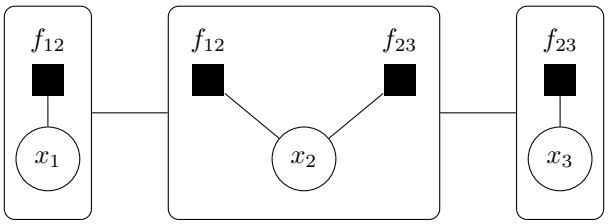

Figure 1.4: One-mode projection onto the variable nodes in Figure 1.1.

is aggregated "in the cloud for crowd intelligence extraction and human-centric service delivery" (Guo et al., 2015). Over the past decade, substantial research has been conducted on defining and classifying MCS applications (Capponi et al., 2019; Guo et al., 2015, and references therein). While not discussed in previous work Ayday et al. (2020, 2021), ShareTrace is a MCS application. The following characterization of ShareTrace assumes the four-layered architecture of a MCS application Capponi et al. (2019), which offers a comprehensive set of classification criteria that To offer a clear comparison, Table 1.1 follows the same structure as Capponi et al. (2019). Some aspects of the architecture, namely sampling frequency and sensor activity, are marked according to how ShareTrace is described in previous work, rather than how it would function

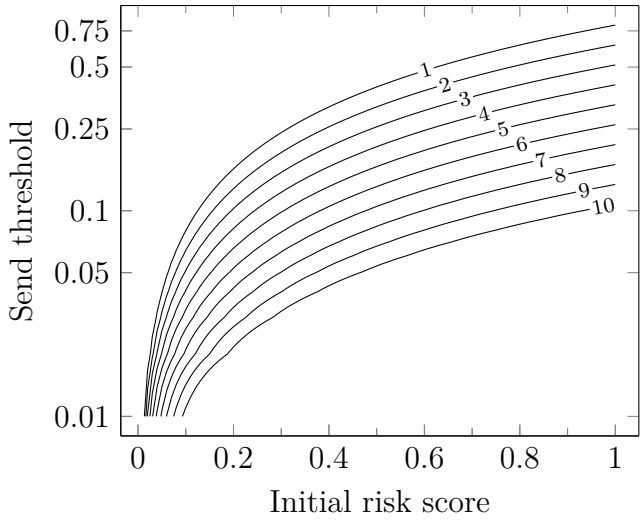

Figure 1.5: Estimated message reachability. Contour lines are shown for integral reachability values. Given an initial risk score and message reachability, a contour line indicates an upper bound on the permissible send threshold.

to optimize for energy efficiency. More detail is provided below in this regard. When classifying ShareTrace as an MCS application, the following description is helpful:

> *ShareTrace is a decentralized, delay-tolerant contact-tracing application that estimates infection risk from proximal user interactions and user symptoms.*

1.2.3.1 Application Layer

Application Tasks

Task Scheduling *Proactive scheduling* allows users to decide when and where they contribute sensing data, while *reactive scheduling* requires that "a user receives a request, and upon acceptance, accomplishes a task" Capponi

et al. (2019). ShareTrace follows proactive scheduling where the sensing task is to detect proximal interactions with other users. Naturally, the scheduling of this task is at the discretion of the ShareTrace user.

Task Assignment *Centralized assignment* assumes that "a central authority distributes tasks among users." Conversely, with *decentralized assignment*, "each participant becomes an authority and can either perform the task or forward it to other users. This approach is very useful when users are very interested in specific events or activities" Capponi et al. (2019). The latter naturally aligns with ShareTrace in which each user is responsible for their own interactions that are both temporally and spatially specific.

Task Execution With *single-task execution*, MCS applications assign one type of task to users, while *multi-task execution* assigns multiple types of tasks Capponi et al. (2019). ShareTrace only involves the single task of sensing proximal interactions. Alternatively, ShareTrace could be defined more abstractly as sensing infection risk through interactions and user symtpoms. In this case, ShareTrace would follow a multi-tasked execution model, where the task of sensing user symptoms is achieved through user reporting or bodily sensors.

Application Users

User Recruitment *Volunteer-based recruitment* is when citizens can "join a sensing campaign for personal interests...or willingness to help the com-

munity," while *incentive-based recruitment* promotes participation and offers control over the recruitment rate…These strategies are not mutually exclusive and users can first volunteer and then be rewarded for quality execution of sensing tasks" Capponi et al. (2019). ShareTrace assumes volunteer-based recruitment. However, as a decentralized application (dApp) that aligns with the principles of self-sovereignty, an incentive structure that rewards users with verifiable, high-quality data is plausible.

User Selection *User-centric selection* is when "contributions depend only on participants['] willingness to sense and report data to the central collector, which is not responsible for choosing them." *Platform-centric selection* is "when the central authority directly decides data contributors…Platform centric decisions are taken according to the utility of data to accomplish the targets of the campaign" Capponi et al. (2019). ShareTrace employs user-centric selection, because the purpose of the application is to passively sense the user's interactions and provide them with the knowledge of their infection risk.

User Type A *contributor* "reports data to the MCS platform with no interest in the results of the sensing campaign" and "are typically driven by incentives or by the desire to help the scientific or civil communities." A *consumer* joins "a service to obtain information about certain application scenario[s] and have a direct interest in the results of the sensing campaign" Capponi et al. (2019). For ShareTrace, users can be either a consumer or a contributor but are likely biased toward the former.

1.2.3.2 Data Layer

Data Management

Data Storage *Centralized storage* involves data being "stored and maintained in a single location, which is usually a database made available in the cloud. This approach is typically employed when significant processing or data aggregation is required." *Distributed storage* "is typically employed for delay-tolerant applications, i.e., when users are allowed to deliver data with a delayed reporting" Capponi et al. (2019). For sensing human interaction, ShareTrace relies on distributed storage in the form of Dataswift Personal Data Accounts. Moreover, ShareTrace is delay-tolerant, so distributed storage is most appropriate. However, for reporting the population-level risk distribution, it is likely that centralized storage would be used.

Data Format *Structured data* is standardized and readily analyzable. *Unstructured data*, however, requires significant processing before it can be used Capponi et al. (2019). ShareTrace deals with structured data (e.g., user symptoms, actor URIs).

Data Dimensionality *Single-dimension data* typically occurs when a single sensor is used, while *multi-dimensional data* arises with the use of multiple sensors. ShareTrace data is one-dimensional because it only uses Bluetooth to sense nearby users.

Data Processing

Data Pre-processing *Raw data output* implies that no modification is made to the sensed data. *Filtering and denoising* entail "removing irrelevant and redundant data. In addition, they help to aggregate and make sense of data while reducing at the same time the volume to be stored" Capponi et al. (2019). ShareTrace only retains the actor URIs that correspond to valid contacts (i.e., lasting at least 15 minutes).

Data Analytics *Machine learning* (ML) *and data mining analytics* are not real-time. They "aim to infer information, identify patterns, or predict future trends." On the contrary, *real-time analytics* consist of "examining collected data as soon as it is produced by the contributors" Capponi et al. (2019). ShareTrace aligns with the former category since it aims to infer the infection risk of users.

Data Post-processing *Statistical post-processing* "aims at inferring proportions given quantitative examples of the input data. *Prediction post-processing* aims to determine "future outcomes from a set of possibilities when given new input in the system" Capponi et al. (2019). ShareTrace applies predictive post-processing via risk propagation.

1.2.3.3 Communication Layer

Communication Technology

Infrastructured Technology *Cellular* connectivity "is typically required from sensing campaign[s] that perform real-time monitoring and require data

reception as soon as it is sensed." *WLAN* "is used mainly when sensing organizers do not specify any preferred reporting technologies or when the application domain permits to send data" at "a certain amount of time after the sensing process" Capponi et al. (2019). Infrastructured technology is also referred to as the *infrastructured transmission paradigm* Ma et al. (2014). ShareTrace does not require cellular infrastructure, because it is delay-tolerant and thus only requires WLAN.

Infrastructure-less Technology *Infrastructure-less technologies* "consists of device-to-device (D2D) communications that do not require any infrastructure…but rather allow devices in the vicinity to communicate directly." Technologies include *WiFi-Direct*, *LTE-Direct*, and *Bluetooth* (Capponi et al., 2019). Infrastructure-less technology is also called the *opportunistic transmission paradigm* (Ma et al., 2014). ShareTrace uses Bluetooth because of its energy efficiency and short range.

Data Reporting

Upload mode With *relay uploading*, "data is delivered as soon as collected." *Store-and-forward* "is typically used in delay-tolerant applications when campaigns do not need to receive data in real-time" (Capponi et al., 2019). Because ShareTrace is delay-tolerant, it uses store-and-forward uploading.

Methodology *Individualized sensing* is "when each user accomplishes the requested task individually and without interaction with other participants." *Collaborative sensing* is when "users communicate with each other, exchange data[,] and help themselves in accomplishing a task or delivering information to the central collector. Users are typically grouped and exchange data exploiting short-range communication technologies, such as WiFi-[D]irect or Bluetooth…Note that systems that create maps merging data from different users are considered individual because users do not interact between each other to contribute" (Capponi et al., 2019). The methodology of sensing is similar to the *sensing scale* which is typically dichotomized as *personal* (Ganti et al., 2011; Lane et al., 2010) (i.e., individualized) and *community* (Ganti et al., 2011) or *group* (Lane et al., 2010) (i.e., collaborative). ShareTrace is inherently collaborative, relying on mobile devices to exchange actor URIs to estimate infection risk. Thus, collaborative sensing is used.

Timing *Timing* is based on whether devices "should sense in the same period or not." *Synchronous timing* "includes cases in which users start and accomplish at the same time the sensing task. For synchronization purposes, participants communicate with each other." *Asynchronous timing* occurs "when users perform sensing activity not in time synchronization with other users" (Capponi et al., 2019). ShareTrace requires synchronous timing, because contact sensing inherently requires synchronous communication between the involved devices.

1.2.3.4 Sensing Layer

Sensing Elements

Sensor Deployment *Dedicated deployment* involves the use of "non-embedded sensing elements," typically for a specific task. *Non-dedicated deployment* utilizes sensors that "do not require to be paired with other devices for data delivery but exploit the communication capabilities of mobile devices" (Capponi et al., 2019). ShareTrace relies on non-dedicated deployment since it relies on Bluetooth that is ubiquitous in modern-day mobile devices.

Sensor Activity *Always-on sensors* "are required to accomplish mobile devices['] basic functionalities, such as detection of rotation and acceleration… Activity recognition [i.e., context awareness]…is a very important feature that accelerometers enable." *On-demand sensors* "need to be switched on by users or exploiting an application running in the background. Typically, they serve more complex applications than always-on sensors and consume a higher amount fo energy" (Capponi et al., 2019). ShareTrace uses Bluetooth, which may be considered on-demand. While energy efficient, users do control when it is enabled. Ideally, ShareTrace would also use always-on sensors to enable Bluetooth with context awareness (i.e., that the user is carrying or nearby the device).

Acquisition *Homogeneous acquisition* "involves only one type of data and it does not change from one user to another one," while *heterogeneous acquisition* "involves different data types usually sampled from several sensors"

(Capponi et al., 2019). ShareTrace is homogeneous, because all users sense the same data from one type of sensor.

Data Sampling

Sampling Frequency *Continuous sensing* "indicates tasks that are accomplished regularly and independently [of] the context of the smartphone or the user['s] activities." *Event-based sensing* is "data collection [that] starts after a certain event has occurred. In this context, an event can be seen as an active action from a user or the central collector, but also a given context awareness" (Capponi et al., 2019). ShareTrace sensing is continuous but would ideally be event-based to conserve device energy.

Sensing Responsibility When the *mobile device* is responsible, "devices or users take sampling decisions locally and independently from the central authority…When devices take sampling decisions, it is often necessary to detect the context [of the] smartphones and wearable devices…The objective is to maximize the utility of data collection and minimize the cost of performing unnecessary operations." When the *central collector* is responsible, they make "decisions about sensing and communicate them to the mobile devices" (Capponi et al., 2019). Given the human-centric nature of the ShareTrace sensing task, mobile devices are responsible.

User involvement *Participatory involvement* "requires active actions from users, who are explicitly asked to perform specific tasks. They are re-

sponsible to consciously meet the application requests by deciding when, what, where, and how to perform sensing tasks." *Opportunistic involvement* means that "users do not have direct involvement, but only declare their interest in joining a campaign and providing their sensors as a service. Upon a simple handshake mechanism between the user and the MCS platform, a MCS thread is generated on the mobile device (e.g., in the form of a mobile app), and the decisions of what, where, when, and how to perform the sensing are delegated to the corresponding thread. After having accepted the sensing process, the user is totally unconscious with no tasks to perform and data collection is fully automated…The smartphone itself is context-aware and makes decisions to sense and store data, automatically determining when its context matches the requirements of an application. Therefore, coupling opportunistic MCS systems with context-awareness is a crucial requirement" (Capponi et al., 2019). Earlier works on MCS refer to user involvement as the *sensing paradigm* (Ganti et al., 2011; Lane et al., 2010; Ma et al., 2014). ShareTrace is opportunistic, ideally with context-awareness.

Application	Task	Scheduling	Proactive Reactive	•
		Assignment	Centralized Decentralized	 •
		Execution	Single task Multi-tasking	•
	User	Recruitment	Voluntary Incentivized	•
		Selection	Platform-centric User-centric	 •
		Type	Consumer Contributor	• •
Data	Management	Storage	Centralized Distributed	 •
		Format	Structured Unstructured	•
		Dimension	Single dimension Multi-dimensional	•
	Processing	Pre-processing	Raw data Filtering and denoising	 •
		Analytics	ML and data mining Real-time	•
		Post-processing	Statistical Prediction	 •
Communication	Technologies	Infrastructured	Cellular WLAN	• •
		Infrastructure-less	LTE-Direct WiFi-Direct Bluetooth	 •
	Reporting	Upload mode	Relay Store and forward	 •
		Methodology	Individual Collaborative	 •
		Timing	Synchronous Asynchronous	•
Sensing	Elements	Deployment	Dedicated Non-dedicated	 •
		Activity	Always-on On-demand	• ○
		Acquisition	Homogeneous Heterogeneous	•
	Sampling	Frequency	Continuous Event-based	• ○
		Responsibility	Mobile device Central collector	•
		User involvement	Participatory Opportunistic	 •

Table 1.1: ShareTrace classification using the four-layered architecture of a mobile crowdsensing application Capponi et al. (2019). Always (•); with context-awareness (○).

Appendix A

Previous Designs and Implementations

Before working on ShareTrace, I did not have experience developing distributed algorithms. The approach proposed in Chapter 1 is my *fifth* attempt[1] at defining a performant implementation of risk propagation that satisfies the requirements of being decentralized and online. The prior four attempts offered valuable learnings that guided me toward the proposed approach. In fact, the approach defined in Appendix A.4 is the topic of published work Tatton et al. (2022). To document my efforts in developing this thesis, prior designs and implementations are provided in this appendix.

[1] In completing this thesis, I have learned two important rules: define requirements clearly and recognize when those requirements are (not) satisfied. The time and effort needed for this thesis could have been significantly reduced had I adhered to these two simple rules.

A.1 Thinking Like a Vertex

The first iteration of risk propagation Tatton utilized Apache Giraph **?**, an open-source version of the iterative graph-processing library, Pregel Malewicz et al. (2010), which is based on the bulk synchronous parallel model of distributed computing Valiant (1990). Giraph follows the *"think like a vertex" paradigm* in which the algorithm is specified in terms of the local information available to a graph node McCune et al. (2015).

Risk propagation was implemented as defined in Ayday et al. (2020, 2021), using the factor graph representation of the contact network. However, because the Google/Apple API does not permit remotely persisting ephemeral identifiers, the implementation assumed that user geolocation data would be analyzed to generate the factor nodes in the factor graph (Appendix B.3). Figure A.1 describes the high-level architecture. Callouts 1, 2, and 4 were implemented using a fan-out design pattern in which a *ventilator* Lambda function divides the work amongst *worker* Lambda functions.

Several factors prompted me to search for an alternative implementation.

1. *Dependency management incompatibility.* A major cause for redesigning the implementation was the dependency version conflicts between Giraph and the other libraries. In spite of several attempts (e.g., using different library versions, using different versions of Giraph, and forcing specific transitive dependency versions) to resolve these conflicts, a lack of personal development experience and stalled progress prompted me pursue other approaches of implementation.

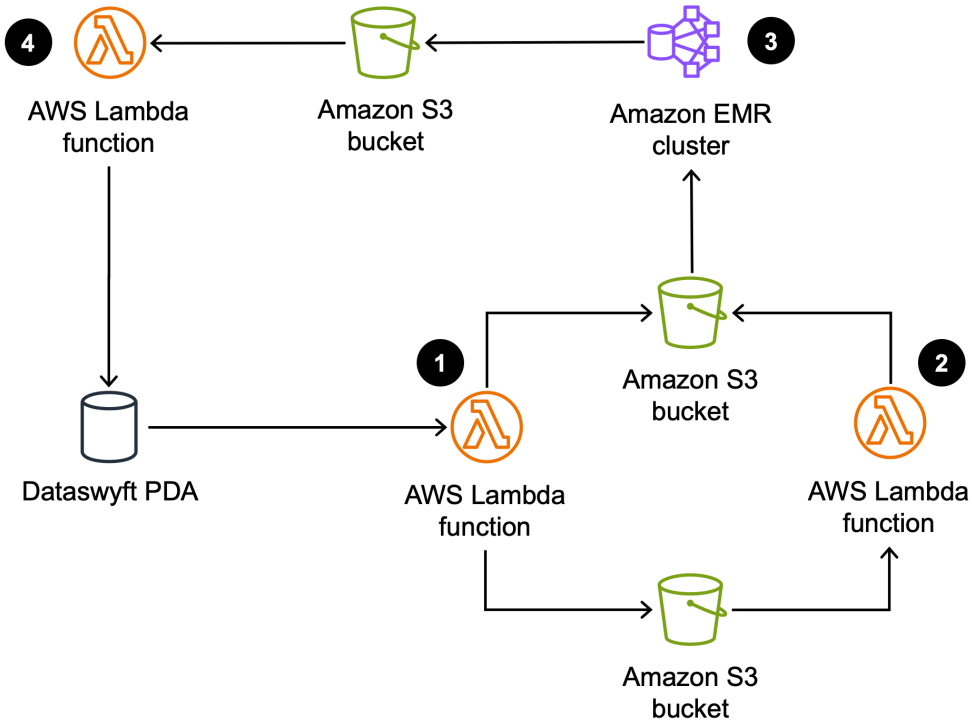

Figure A.1: ShareTrace batch-processing architecture. (1) An AWS Lambda function retrieves the recent risk scores and location data from the Dataswyft Personal Data Accounts (PDAs) of ShareTrace users. Risk scores are formatted as Giraph nodes and stored in an Amazon Simple Storage Service (S3) bucket. Location data is stored in a separate S3 bucket. (2) A Lambda function performs a contact search over the location data and stores the contacts as Giraph edges in the same bucket that stores the Giraph nodes. (3) Amazon Elastic MapReduce (EMR) runs risk propagation as a Giraph job and stores the exposure scores in an S3 bucket. (4) A Lambda function stores the exposure score of each user in their respective PDA.

2. *Implementation complexity.* For a relatively straightforward data flow, the architecture in Figure A.1 corresponds to over 4,000 lines of source code. In retrospect, AWS Step Functions could have been used to orchestrate the workflow, including the fan-out design pattern, which would have simplified the Lambda function implementations. Regarding the implementation of risk propagation, one-mode projection onto variable nodes (first used in Appendix A.4) would have simplified the implementation since it avoids multiple node types, multiple message types, and enables the encapsulation of state and message-passing behavior.

3. *External data persistence.* One of the core tenets of Dataswyft is that the user fully controls the access to their data. However, as shown in Figure A.1, user data is stored in S3 buckets. While it is possible to encrypt S3 objects at rest and enforce a short data retention policy, data persistence to any extent is suboptimal for *privacy-preserving* digital contact tracing.

A.2 Subgraph Actors

In an attempt to simplify the design in Appendix A.1, I rewrote risk propagation using Ray, a Python library that "aims to provide a universal API for distributed computing" Team (2022). While it claims to support actor-based programming, Ray only offers coarse-grained concurrency, with each actor being mapped to a machine processor. As with Appendix A.1, I assumed the factor graph representation of the contact network. To achieve parallelism, the

factor graph is partitioned amongst the actors such that each actor maintains a subset of variable nodes *or* factor nodes. The graph topology is stored in shared memory so that all actors can efficiently access it. The lifetime of this design was brief for the following reasons.

1. *Poor performance.* Communication between Ray actors requires message serialization. Moreover, partitioning the factor graph into subsets of factor and variable nodes results in maximal interprocess communication. Unsurprisingly, this choice of partitioning manifested in slow runtime performance.

2. *Design complexity.* Not using a framework, like Giraph, meant that this implementation required more low-level code to implement actor functionality and message passing. Regardless of the performance, the overall design of this implementation was poorly organized and overthought.

A.3 Driver-Monitor-Worker Framework

Based on the poor runtime performance and complexity of the approach taken in Appendix A.2, I speculated that centralizing the mutable aspects of risk propagation (i.e., the current value of each variable node) would improve both metrics. With this is mind, I designed the *monitor-worker-driver* (MWD) *framework*, which draws inspiration from the *tree of actors* design pattern Team (b). Figure A.2 describes the framework.

For risk propagation, the driver creates the factor graph from the set of risk scores S and contacts C, stores the factor graph in shared memory, sets

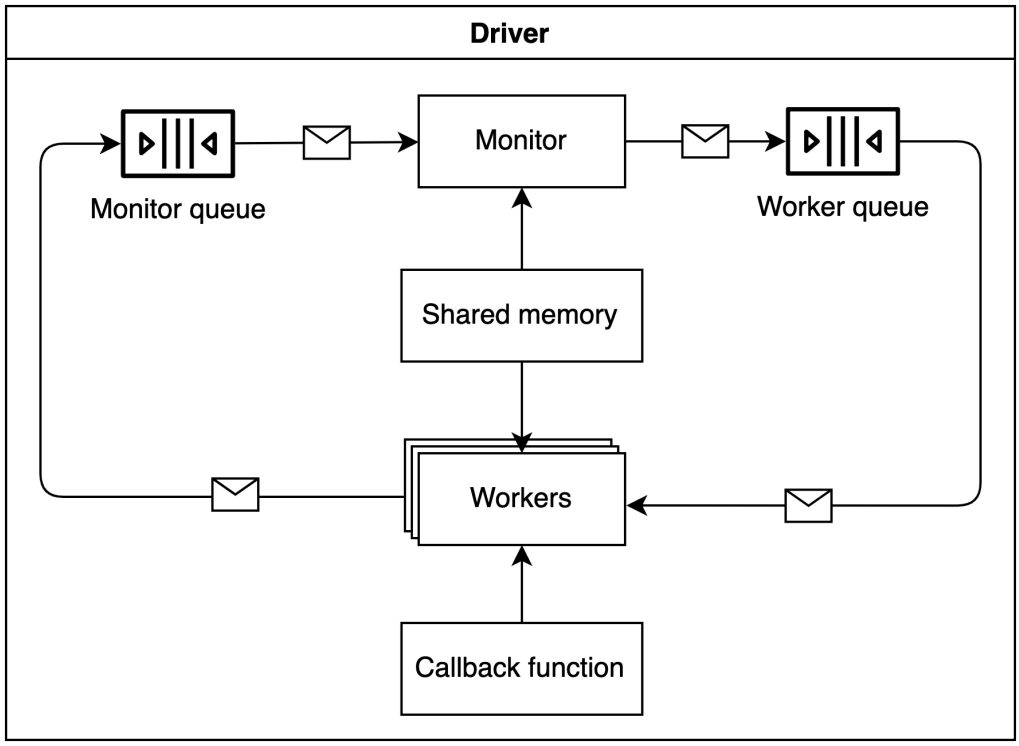

Figure A.2: Monitor-worker-driver framework. The *monitor* encapsulates the mutable state of the program and decides which messages are processed by workers. A *worker* is a stateless entity that processes the messages that the monitor puts in the *worker queue*. Worker behavior is defined by a *callback function*, which typically depends on the message contents. The side effects of processing a message are recorded as new messages and put in the *monitor queue*. This cycle repeats until some termination condition is satisfied. Any immutable state of the program can be stored in *shared memory* for efficient access. The *driver* is the entry point into the program. It initializes the monitor and workers, and then waits for termination.

the initial state of the monitor to be the maximum risk score of each user, and puts all risk scores in the monitor queue. During message passing, the monitor maintains the maximum risk score (i.e., exposure score) for each variable node.

The MWD framework was the first approach that utilized the send coefficient to ensure the convergence and termination of message passing. However, because the MWD-based implementation assumed the factor graph representation of the contact network, the send coefficient was applied to both variable and factor messages.

Compared to the approach in Appendix A.2, this implementation provides a cleaner design and less communication overhead. However, what prompted me to consider (yet another) an alternative implementation was its scalability. Because the monitor processes messages serially, it is a bottleneck for algorithms in which the workers perform fine-grained tasks. Indeed, the Ray documentation notes that the parallelization of small tasks is an anti-pattern because the interprocess communication cost exceeds the benefit of multiprocessing Team (a). Unfortunately, the computation performed by factor nodes and variable nodes is too fine-grained, so the scalability of the MWD framework was demonstrably poor.

A.4 Projected Subgraph Actors

Appendix B

Evaluation

B.1 Experimental Design

Risk propagation requires a partitioning or clustering algorithm, as described in Algorithm **??**. We configured the METIS graph partitioning algorithm Karypis and Kumar (1998) to use k-way partitioning with a load imbalance factor of 0.2, to attempt contiguous partitions that have minimal inter-partition connectivity, to apply 10 iterations of refinement during each stage of the uncoarsening process, and to use the best of 3 cuts.

B.1.1 Synthetic Graphs

We evaluate the scalability and efficiency of risk propagation on three types of graphs: a random geometric graph (RGG) Dall and Christensen (2002), a benchmark graph (LFRG) Lancichinetti et al. (2008), and a clustered scale-free graph (CSFG) Holme and Kim (2002). Together, these graphs demonstrate

some aspects of community structure Fortunato (2010) which allows us to more accurately measure the performance of risk propagation. When constructing a RGG, we set the radius to $r(n) = \min\left(1, 0.25^{\log_{10}(n)-1}\right)$, where n is the number of users. This allows us to scale the size of the graph while maintaining reasonable density. We use the following parameter values to create LFRGs: mixing parameter $\mu = 0.1$, degree power-law exponent $\gamma = 3$, community size power-law exponent $\beta = 2$, degree bounds $(k_{\min}, k_{\max}) = (3, 50)$, and community size bounds $(s_{\min}, s_{\max}) = (10, 100)$. Our choices align with the suggestions by Lancichinetti et al. (2008) in that $\gamma \in \mathbb{R}_{[2,3]}$, $\beta \in \mathbb{R}_{[1,2]}$, $k_{\min} < s_{\min}$, and $k_{\max} < s_{\max}$. To build CSFGs, we add $m = 2$ edges for each new user and use a triad formulation probability of $P_t = 0.95$. For all graphs, we remove self-loops and isolated users.

The following defines our data generation process. Let p be the probability of a user being "high risk" (i.e., $r \geq 0.5$) Then, with probability $p = 0.2$, we sample $L + 1$ values from the uniform distribution $\mathbb{U}_{[0.5,1)}$. Otherwise, we sample from $\mathbb{U}_{[0,0.5)}$. This assumes symptom scores and exposure scores are computed daily and includes the present day. We generate the times of these risk scores by sampling a time offset $t_{\text{off}} \sim \mathbb{U}_{[0\text{s};86{,}400\text{s}]}$ for each user such that $t_d = t_{\text{now}} + t_{\text{off}} - d$ days, where $d \in \mathbb{N}_{[0,L]}$. To generate a contact times, we follow the same procedure for risk scores, except that we randomly sample one of the $L + 1$ times and use that as the contact time.

We evaluate various transmission rates and send tolerances:

$$(\gamma, \alpha) \in \{0.1, 0.2, \ldots, 1\} \times \{0.1, 0.2, \ldots, 0.9\}.$$

For all γ, α, we set $n = 5,000$ and $K = 2$.

To measure the scalability of risk propagation, we consider $n \in \mathbb{N}_{[10^2,10^4]}$ users in increments of 100 and collect 10 iterations for each n. The number of actors we use depends on n such that $K(n) = 1$ if $n < 10^3$ and $K(n) = 2$ otherwise. Increasing K for our choice of n did not offer improved performance due to the communication overhead.

B.1.2 Real-World Graphs

We analyze the efficiency of risk propagation on three real-world contact networks that were collected through the SocioPatterns collaboration. Specifically, we use contact data in the setting of a high school (Thiers13) Fournet and Barrat (2014), a workplace (InVS15), and a scientific conference (SFHH) **?**. Because of limited availability of large-scale contact networks, we do not use real-world contact networks to measure the scalability of risk propagation.

To ensure that all risk scores are initially propagated, we shift all contact times forward by t_{now} and use $(t_{\text{now}} - 1 \text{ day})$ when generating risk scores times. In this way, we ensure the most recent risk score is still older than the first contact time. Risk score values are generated in the same manner as described in Section B.1.1 with the exception that we only generate one score. Lastly, we perform 10 iterations over each data set to obtain an average performance.

B.2 Results

B.2.1 Efficiency

Prior to measuring scalability and real-world performance, we observed the effects of send tolerance and transmission rate on the efficiency of risk propagation. As ground truth, we used the maximum update count for a given transmission rate. Fig. B.1 indicates that a send tolerance of $\gamma = 0.6$ permits 99% of the possible updates. Beyond $\gamma = 0.6$, however, the transmission rate has considerable impact, regardless of the graph. As noted in Section 1.2.2, send tolerance quantifies the trade-off between completeness and efficiency. Thus, $\gamma = 0.6$ optimizes for both criteria.

Unlike the update count, Fig. B.1 shows a more variable relationship with respect to runtime and message count. While, in general, transmission rate (send tolerance) has a direct (resp. inverse) relationship with runtime and message count, the graph topology seems to have an impact on this fact. Namely, the LFRG displayed less variability across send tolerance and transmission rate than the RGG and CSFG, which is the cause for the large interquartile ranges. Therefore, it is useful to consider the lower quartile Q_1, the median Q_2, and the upper quartile Q_3. For $\alpha = 0.8$ and $\gamma = 0.6$, risk propagation is more efficient with $(Q_1, Q_2, Q_3) = (0.13, 0.13, 0.46)$ normalized runtime and $(Q_1, Q_2, Q_3) = (0.13, 0.15, 0.44)$ normalized message count.

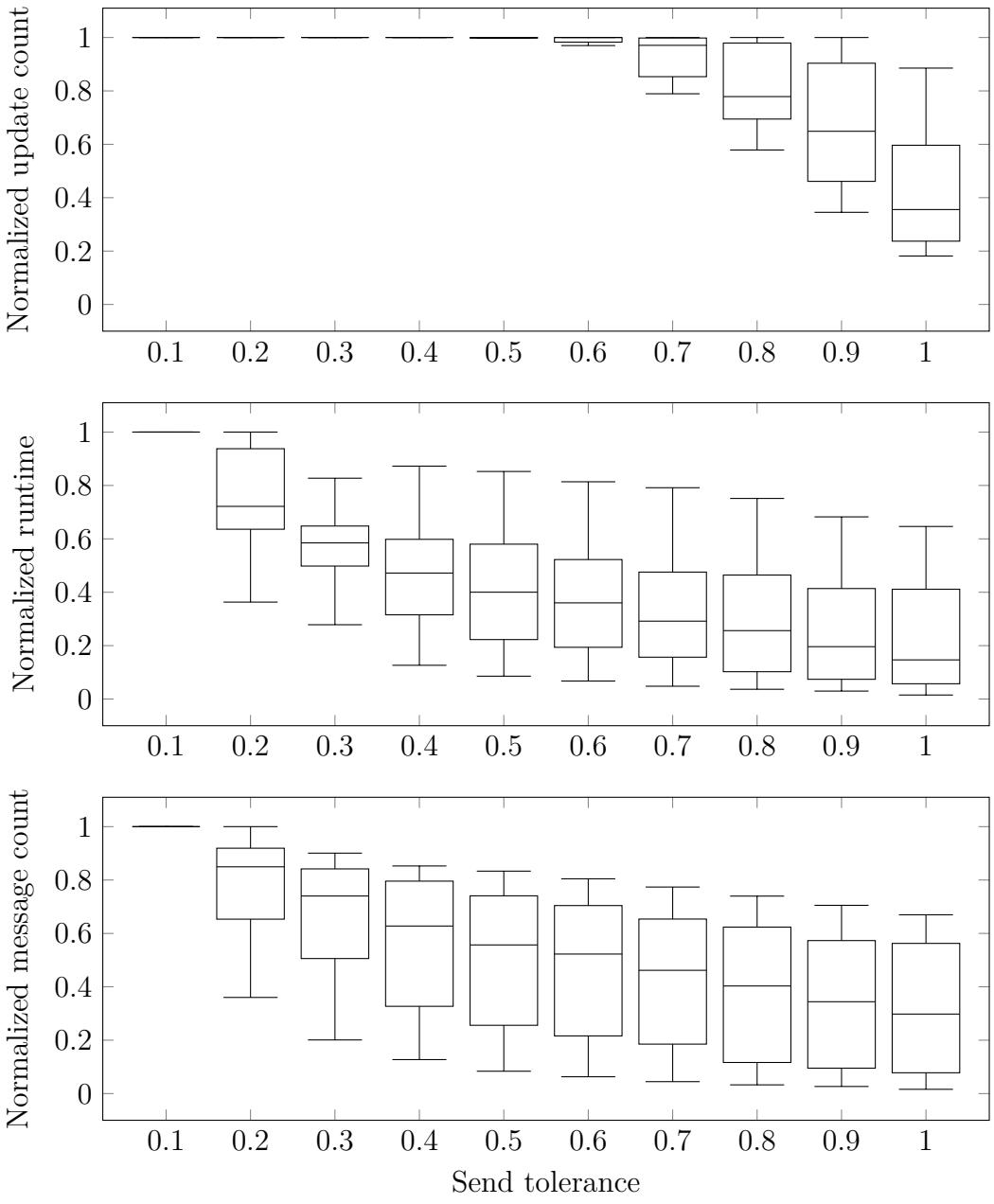

Figure B.1: Effects of send tolerance on efficiency. All dependent variables are normalized across graphs and transmission rates.

B.2.2 Message Reachability

To validate the accuracy of (**??**), we collected values of (**??**) and (**??**) for real-world and synthetic graphs. For the latter set of graphs, we observed reachability while sweeping across values of γ and α.

To measure the accuracy of (**??**), let the *message reachability ratio* (MRR) be defined as

$$\text{mrr}(u) := \frac{m(u)}{\hat{m}(u)}. \tag{B.1}$$

Overall, (**??**) is a good estimator of (**??**). Across all synthetic graphs, (**??**) modestly underestimated (**??**) with quartiles $(Q_1, Q_2, Q_3) = (0.71, 0.84, 0.98)$ for the (B.1). For $\alpha = 0.8$ and $\gamma = 0.6$, the quartiles of (B.1) were $(Q_1, Q_2, Q_3) = (0.52, 0.77, 1.12)$ and $(Q_1, Q_2, Q_3) = (0.79, 0.84, 0.93)$, respectively. Table B.1 provides mean values of (B.1) for both synthetic and real-world graphs. Fig. B.2 indicates that moderate values of γ tend to result in a more stable MRR, with lower (higher) γ underestimating (resp. overestimating) (**??**). With regard to transmission rate, (B.1) tends to decrease with increasing α, but also exhibits larger interquartile ranges.

Because (**??**) does not account for the temporality constraints (**??**) and (**??**), it does not perfectly estimate (**??**). With lower γ and higher α, (**??**) suggests higher MR. However, because a message is only passed under certain conditions (see Algorithm **??**), this causes (**??**) to overestimate (**??**). While (**??**) theoretically is an upper bound on (**??**), it is possible for (**??**) to underestimate (**??**) if the specified value of (v) overestimates the true value of (v). When computing (B.1) for Fig. B.2, we used the mean (v) across all users v,

so $\mathrm{mrr}(u) > 1$ in some cases.

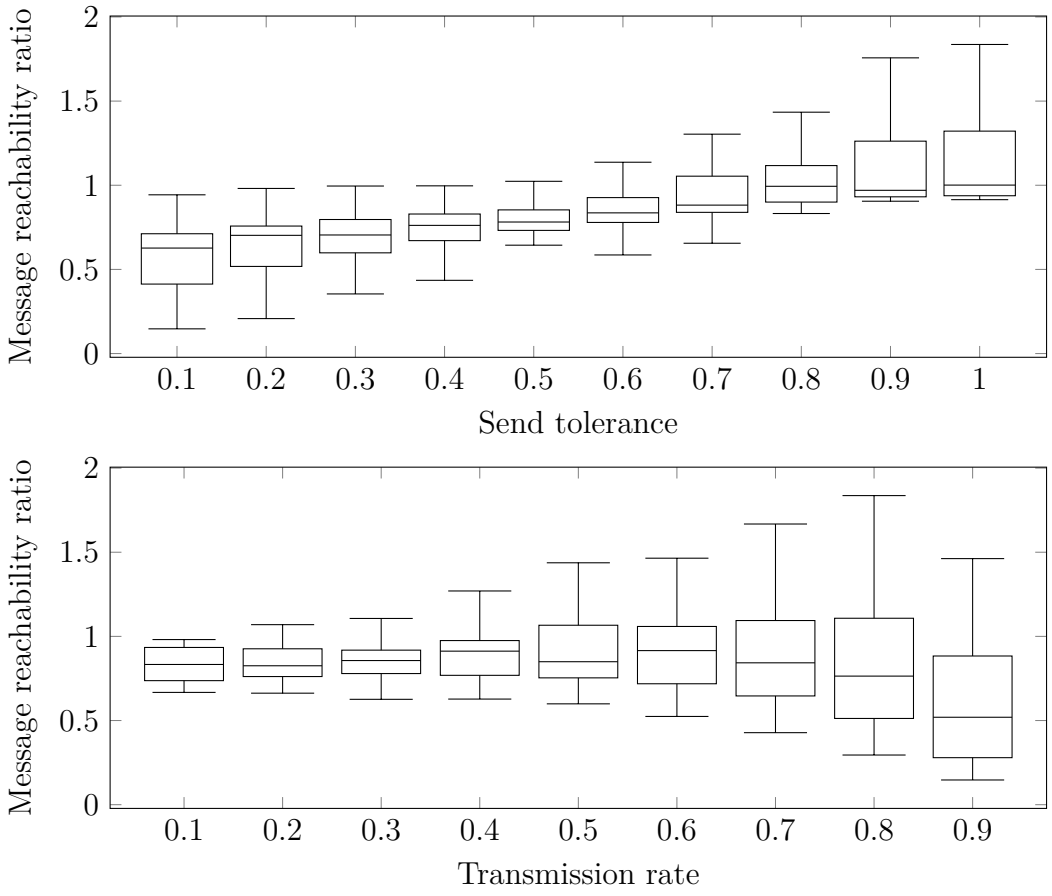

Figure B.2: Effects of send tolerance and transmission rate on the message reachability ratio. Independent variables are grouped across graphs.

B.2.3 Scalability

Fig. B.3 describes the runtime behavior of risk propagation. The runtime of CSFGs requires further investigation. A linear regression fit explains ($R^2 = 0.52$) the runtime of LFRGs and RGGs with a slope $m = (1.1 \pm 0.1) \cdot 10^{-3}$ s/contact and intercept $b = 4.3 \pm 1.6$s ($\pm 1.96 \cdot \mathrm{SE}$).

Setting	$\mathrm{mrr}(u) \pm 1.96 \cdot \mathrm{SE}$
Synthetic	
LFR	0.88 ± 0.14
RGG	0.74 ± 0.12
CSFG	0.90 ± 0.14
	$\mathbf{0.85 \pm 0.08}$
Real-world	
Thiers13	0.58 ± 0.01
InVS15	0.63 ± 0.01
SFHH	0.60 ± 0.01
	$\mathbf{0.60 \pm 0.01}$

Table B.1: Message reachability ratio for synthetic and real-world graphs ($\alpha = 0.8$, $\gamma = 0.6$). Synthetic (real-world) ratios are averaged across parameter combinations (resp. runs).

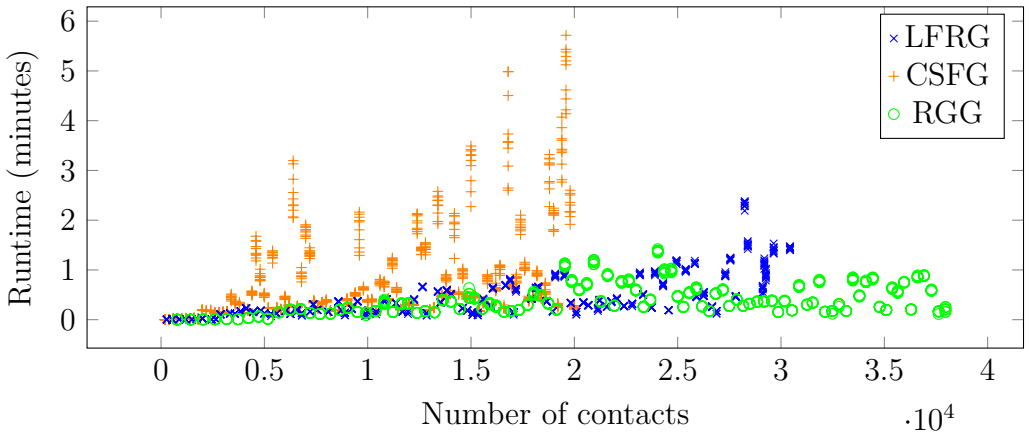

Figure B.3: Runtime of risk propagation on synthetic graphs containing 100–10,000 users and approximately 200–38,000 contacts.

B.3 Location-Based Contact Tracing

- Motivation: Google/Apple API prevents exporting Bluetooth EphIDs
- Other location-based contact tracing approaches

B.3.1 System Model

The system model is very similar to previous work Ayday et al. (2020, 2021) and designs. The only difference is that user geolocation data is collected instead of Bluetooth ephemeral identifiers. Figure B.4 shows the modified dataflow[1]. *Geohashing* is a public-domain encoding system that maps *geographic coordinates* (i.e., latitude-longitude ordered pairs (Sickle, 2004, p. 5)) to alphanumeric strings called *geohashes*, where the length of a geohash is correlated with its geospatial precision Morton (1966). To offer some basic privacy, a user's precise geolocation history is obfuscated on-device by encoding geographic coordinates as geohashes with 8-character precision which corresponds to a region of 730m^2.

B.3.2 Contact Search

a temporally ordered sequence of timestamped geolocations. It is assumed that

1. geolocation histories are not recorded on a fixed schedule, and
2. a user remains at a geolocation until the next geolocation is recorded.

[1]See **??** (p. **??**) for the definition of a dataflow diagram.

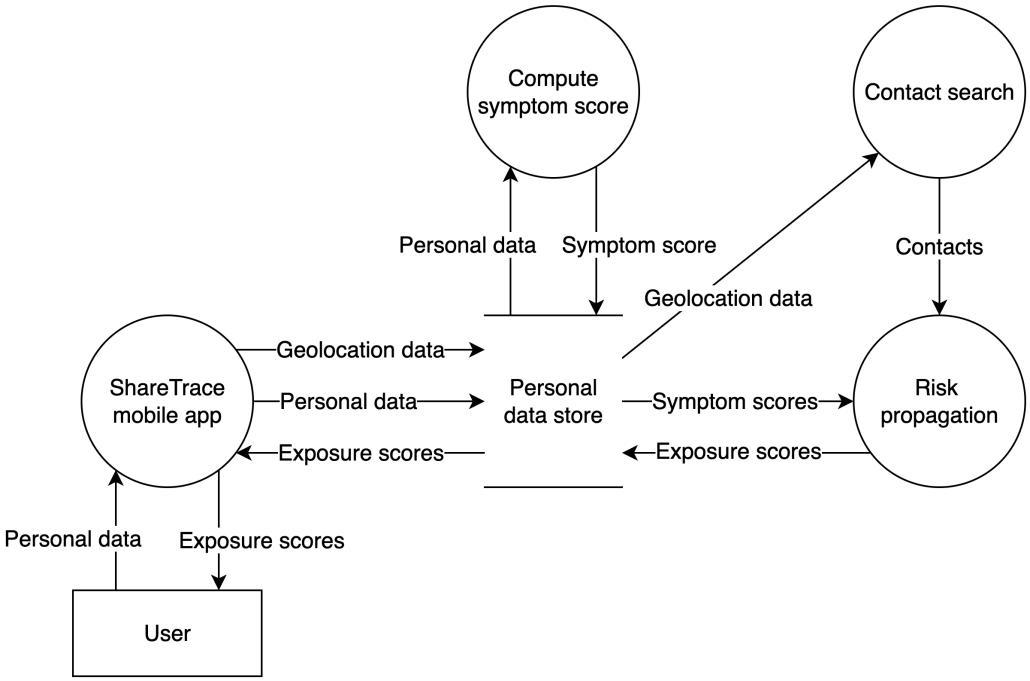

Figure B.4: Location-based ShareTrace dataflow. This requires that risk propagation is executed in a centralized setting since all user geolocation data is needed to construct the contact network.

Finding the most recent contact between two users from their aligned geolocation histories is similar to finding the last k-length common substring between two strings, where each symbol represents a timestamped geolocation. The difference lies in how the start and end of the contact time interval is defined. By assumption 2, the start (end) of a contact time interval is defined as the earlier (ref. later) timestamp of the two first (ref. last) timestamped geolocations in the sequence where the two histories differ. Figure B.5 provides a visual example.

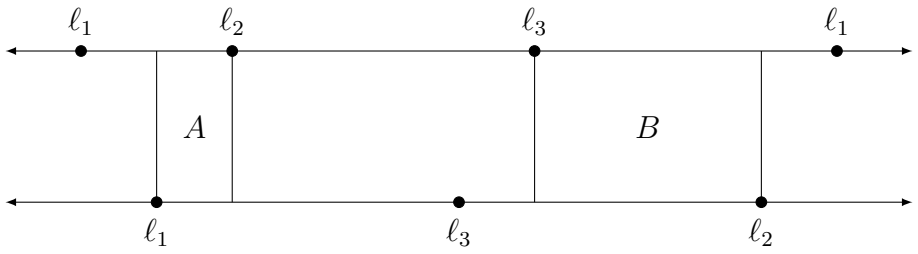

Figure B.5: Contact search with two geolocation histories. Each line denotes time, increasing from left to right. A point ℓ_i is a geolocation and occurs relative in time with respect to the placement of other points. Region B defines the contact interval as it is of sufficient duration and occurs after A.

B.3.2.1 Naive Contact Search

A naive approach to finding all contacts amongst a set of geolocation histories $\mathcal{H}$ is to compare all unordered pairs. For a given pair of aligned geolocation histories, the idea is to maintain a pointer to the previous and current index in each history, advancing the pair of pointers whose geolocation occurs later in time. Once a common geolocation is found, all pointers are advanced together until the geolocations differ. If the sequence is δ-contiguous, where a sequence of timestamped geolocations S is *δ-contiguous* if $S.length \geq \delta$ and then it is recorded. The latest such sequence is used to define the contact between the two users. Because only the most recent time of contact is of interest, the procedure can be improved by iterating in reverse and then terminating once a sequence is found. Regardless, this approach takes $\Theta(n^2)$ time, where $n = |\mathcal{H}|$, because all $\frac{n(n-1)}{2}$ unique pairs must be considered.

B.3.2.2 Indexed Contact Search

While the **for** loop in NAIVE-CONTACT-SEARCH is *embarrassingly parallel* (Herlihy and Shavit, 2012, p. 14), the naive approach is neither scalable nor efficient. It can be improved by observing that it is necessary, but not sufficient, that a pair of ϵ-proximal geolocations exists between two geolocation histories for a contact to exist. Therefore, the geolocation histories $\mathcal{H}$ can be indexed into a spatial data structure $\mathcal{I}$ Mahmood et al. (2019); Mokbel et al. (2003); **?** and then only consider the geolocation-history pairs that share at least one ϵ-proximal geolocation pair. This approach is described by the INDEXED-CONTACT-SEARCH operation.

Line 2 executes a fixed-radius near-neighbors search (FR-NNS) Bentley (1975); Brin (1995) for each geolocation in the spatial index $\mathcal{I}$. Formally, given a set of geolocations $\mathcal{L} \subseteq \mathbb{L}$, a metric d, and a distance ϵ, the *fixed-radius near-neighbors* of a geolocation $\ell \in \mathcal{L}$ is defined as the subset of ϵ-proximal geolocations Brin (1995),

$$\mathcal{N}(\ell) = \{\ell' \in \mathcal{L} \mid d(\ell, \ell') \leq \epsilon\}$$

Note that the set of neighbors $\mathcal{N}(i)$ of user i corresponds to the geolocations that are ϵ-proximal to *any* of the geolocations in their geolocation history H_i,

$$\mathcal{N}(i) = \bigcup_{\ell \in H_i} \mathcal{N}(\ell).$$

On line 3, the operation UNIQUE-USERS maps these near-neighbors back to

the associated users, removing any duplicates that may arise from mapping multiple geolocations to the same user. Finally, line 4 maps the set of users $\mathcal{U}$ back to their geolocation histories and runs Naive-Contact-Search on the resultant subset.

```
Indexed-Contact-Search(ℋ)
1: 𝓘 ← Spatially-Index(ℋ)
2: 𝒩 ← Fixed-Radius-Near-Neighbors(𝓘, ϵ)
3: 𝒰 ← Unique-Users(𝒩, ℋ)
4: return Naive-Contact-Search({H_i ∈ ℋ | i ∈ 𝒰})
```

To carry out FR-NNS, one approach is to use a *ball tree*, a complete binary tree that associates with each node a hypersphere that contains a subset of the data (Kibriya and Frank, 2007; Omohundro, 1989; **?**). Any metric can be used to perform FR-NNS on a ball tree. However, because geolocation is represented as geographic coordinates, metrics that assume a Cartesian coordinate system may be unsuitable. One of the simplest geometric models of the Earth is that of a sphere. Given two geographic coordinates, the problem of finding the length of the geodesic[2] between them is known as the *inverse geodetic problem* (**?**). Assuming a spherical Earth, the solution to the inverse problem is to find the length of the segment that joins the two points on a great circle[3].

The *haversine*, or the half "versed" (i.e., reversed) sine, of a central angle θ is defined as

$$\operatorname{hav}\theta = \frac{\operatorname{vers}\theta}{2} = \frac{1-\cos\theta}{2} = \sin^2\frac{\theta}{2}. \tag{B.2}$$

[2]The *geodesic* is the shortest segment between two points on an ellipsoid (Lu et al., 2014).
[3]The *great circle* is the cross-section of a sphere that contains its center (Lu et al., 2014)

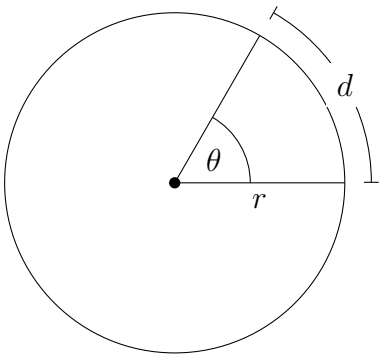

Figure B.6: Central angle of a great circle.

The great-circle distance d between two points can be found by inversing B.2 and solving,

$$d(\ell_i, \ell_j) = 2 \cdot \arcsin \sqrt{\sin^2 \frac{\phi_i - \phi_j}{2} + \cos \phi_i \cdot \cos \phi_j \cdot \sin^2 \frac{\lambda_i - \lambda_j}{2}},$$

where $\ell_i = (\phi_i, \lambda_i)$ is a latitude-longitude coordinate in radians (Brummelen, 2013, pp. 157–162).

The choice of the great-circle distance was primarily driven by its readily available usage in the scikit-learn **?** implementation of a ball tree. If such an approach for discovering contacts were to be used in practice, more advanced *geodetic datum* (Lu et al., 2014, pp. 71–130) could be used to provide better geospatial accuracy. Moreover, by projecting geodetic coordinates onto the plane, metrics that assume a Cartesian coordinate system could be used instead (Lu et al., 2014, pp. 265–326).

The running time of SPATIALLY-INDEX is $\Theta(|\mathcal{H}| \cdot \log |\mathcal{H}|)$ Omohundro (1989). Assuming the ball tree is balanced[4], the running time of FIXED-

[4]The ball-tree implementation provided by scikit-learn **?** ensures the tree is balanced.

RADIUS-NEAR-NEIGHBORS to find the ϵ-proximal neighbors of all geolocations in the spatial index $\mathcal{I}$ is $\Theta(k\,|\mathcal{I}| \cdot \log|\mathcal{I}|)$, where k is the dimensionality of a tree element (i.e., $k = 2$ for geographic coordinates). The running time of UNIQUE-USERS is $\Theta(|\mathcal{N}|)$. While the running time of NAIVE-CONTACT-SEARCH is $\Theta\left(|\mathcal{U}|^2\right)$, it is likely that $|\mathcal{U}| << |\mathcal{H}|$. Regardless of the input,

$$|\mathcal{H}| \geq |\mathcal{I}| \geq |\mathcal{U}|$$

since $|\mathcal{H}| = |\mathcal{I}|$ only if all geolocations in $\mathcal{H}$ are distinct, and $|\mathcal{I}| = |\mathcal{U}|$ only if each user has exactly one geolocation that is distinct from all other users. Dependent upon the input, however, is $|\mathcal{N}|$. In the worst case, $|\mathcal{N}| \in O\left(|\mathcal{I}|^2\right)$ if each geolocation is ϵ-proximal to all other geolocations. This implies that the overall worst-case running time of INDEXED-CONTACT-SEARCH is $O(|\mathcal{H}|^2)$. In practice, the running time depends on the geohash precision as well as the geospatial density and mobility behavior of the user population.

B.3.3 Closing Remarks

Appendix C

Pseudocode Conventions

Pseudocode conventions are mostly consistent with Cormen et al. (2022).

- Indentation indicates block structure.
- Looping and conditional constructs have similar interpretations to those in standard programming languages.
- Composite data types are represented as *objects*. Accessing an *atttribute* a of an object o is denoted $o.a$. A variable representing an object is a *pointer* or *reference* to the data representing the object. The special value NIL refers to the absence of an object.
- Parameters are passed to a procedure *by value*: the "procedure receives its own copy of the parameters, and if it assigns a value to a parameters, the change is *not* seen by the calling procedure. When objects are passed, the pointer to the data representing the object is copied, but the attributes of the object are not." Thus, attribute assignment "is visible

if the calling procedure has a pointer to the same object."

- A **return** statement "immediately transfers control back to the point of call in the calling procedure."
- Boolean operators **and** and **or** are *short circuiting*.

The following conventions are specific to this work.

- Object attributes may be defined *dynamically* in a procedure.
- Variables are local to the given procedure, but parameters are global.
- The "$\leftarrow$" symbol is used to denote assignment, instead of "=".
- The "=" symbol is used to denote equality, instead of "==", which is consistent with the use of "$\neq$" to denote inequality.
- The "$\in$" symbol is used in **for** loops when iterating over a collection.
- Set-builder notation $\{ x \in X \mid \textsc{Predicate}(x) \}$ is used to create a subset of a collection X in place of constructing an explicit data structure.

Appendix D

Data Structures

Let a *dynamic set* X be a mutable collection of distinct elements. Let x be a pointer to an element in X such that $x.key$ uniquely identifies the element in X. Let a *dictionary* be a dynamic set that supports insertion, deletion, and membership querying (Cormen et al., 2022).

- $\textsc{Insert}(X, x)$ adds the element pointed to by x to X.
- $\textsc{Delete}(X, x)$ removes the element pointed to by x from X.
- $\textsc{Search}(X, k)$ returns a pointer x to an element in the set X such that $x.key = k$; or $\textsc{nil}$, if no such element exists in X.
- $\textsc{Merge}(X, x)$ adds the element pointed to by x, if no such element exists in X; otherwise, the result of applying a function to x and the existing element is added to X.
- $\textsc{Maximum}(X)$ returns a pointer x to the maximum element of the totally ordered set X; or $\textsc{nil}$ if X is empty.

Bibliography

Gul Agha. *Actors: A model of concurrent computation in distributed systems.* PhD thesis, MIT, 1985. URL `http://hdl.handle.net/1721.1/6952`.

Erman Ayday, Fred Collopy, Taehyun Hwang, Glenn Parry, Justo Karell, James Kingston, Irene Ng, Aiden Owens, Brian Ray, Shirley Reynolds, Jenny Tran, Shawn Yeager, Youngjin Yoo, and Gretchen Young. Sharetrace: A smart privacy-preserving contact tracing solution by architectural design during an epidemic. Technical report, Case Western Reserve University, 2020. URL `https://github.com/cwru-xlab/sharetrace-papers/blob/main/sharetace-whitepaper.pdf`.

Erman Ayday, Youngjin Yoo, and Anisa Halimi. ShareTrace: An iterative message passing algorithm for efficient and effective disease risk assessment on an interaction graph. In *Proc. 12th ACM Con. Bioinformatics, Comput. Biology, Health Inform.*, BCB 2021, 2021.

Jon Louis Bentley. A survey of techniques for fixed radius near neighbor searching. Technical report, Stanford University, 1975.

Christopher M. Bishop. Pattern recognition and machine learning. In M. I.

Jordan, Robert Nowak, and Bernhard Schoelkopf, editors, *Inf. Sci. Stat.* Springer, 2006.

Sergey Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 574–584, 1995.

Glen Van Brummelen. *Heavenly Mathematics: The Forgotten Art of Spherical Trigonometry*. Princeton University Press, 2013.

Andrea Capponi, Claudio Fiandrino, Burak Kantarci, Luca Foschini, Dzmitry Kliazovich, and Pascal Bouvry. A survey on mobile crowdsensing systems: Challenges, solutions, and opportunities. *IEEE Commun. Surv. Tut.*, 21(3): 2419–2465, 2019. doi:10.1109/COMST.2019.2914030.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, fourth edition, 2022.

Jesper Dall and Michael Christensen. Random geometric graphs. *Phys. Rev. E*, 66, 2002. doi:10.1103/PhysRevE.66.016121.

Centers for Disease Control and Prevention. Quarantine and isolation, 2021. `https://www.cdc.gov/coronavirus/2019-ncov/your-health/quarantine-isolation.html`.

Santo Fortunato. Community detection in graphs. *Phys. Rep.*, 486, 2010. doi:10.1016/j.physrep.2009.11.002.

Julie Fournet and Alain Barrat. Contact patterns among high school students. *PLoS ONE*, 9, 2014. doi:10.1371/journal.pone.0107878.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995.

Raghu K. Ganti, Fan Ye, and Hui Lei. Mobile crowdsensing: current state and future challenges. *IEEE Commun. Mag.*, 49(11):32–39, 2011. doi:10.1109/MCOM.2011.6069707.

Bin Guo, Zhu Wang, Zhiwen Yu, Yu Wang, Neil Y. Yen, Runhe Huang, and Xingshe Zhou. Mobile crowd sensing and computing: The review of an emerging human-powered sensing paradigm. *ACM Comput. Surv.*, 48(1): 1–31, 2015. doi:10.1145/2794400.

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977. doi:10.1016/0004-3702(77)90033-9.

Carl Hewitt and Henry Baker. Laws for communicating parallel processes. Working paper, MIT Artificial Intelligence Laboratory, 1977. URL `http://hdl.handle.net/1721.1/41962`.

Petter Holme and Beom Jun Kim. Growing scale-free networks with tunable clustering. *Phys. Rev. E*, 65, 2002. doi:10.1103/PhysRevE.65.026107.

George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20, 1998. doi:10.1137/S1064827595287997.

Ashraf M. Kibriya and Eibe Frank. An empirical comparison of exact nearest neighbour algorithms. In *Knowledge Discovery in Databases: PKDD 2007*, pages 140–151, 2007.

Frank R. Kschischang, Brendan J. Frey, and Hans A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Trans. Inf. Theory*, 47, 2001. doi:10.1109/18.910572.

Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Phys. Rev. E*, 78, 2008. doi:10.1103/PhysRevE.78.046110.

Nicholas D. Lane, Emiliano Miluzzo, Hong Lu, Daniel Peebles, Tanzeem Choudhury, and Andrew T. Campbell. A survey of mobile phone sensing. *IEEE Commun. Mag.*, 48(9):140–150, 2010. doi:10.1109/MCOM.2010.5560598.

Zhiping Lu, Yunying Qu, and Shubo Qiao. *Geodesy: Introduction to Geodetic Datum and Geodetic Systems*. Springer, 2014.

Huadong Ma, Dong Zhao, and Peiyan Yuan. Opportunities in mobile crowd sensing. *IEEE Commun. Mag.*, 52(8):29–35, 2014. doi:10.1109/MCOM.2014.6871666.

Ahmed R. Mahmood, Sri Punni, and Walid G. Aref. Spatio-temporal access methods: a survey (2010 - 2017). *Geoinformatica*, 23:1–36, 2019.

Grzegorz Malewicz, Matthew H Austern, Aart J C Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

Robert McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surveys*, 48, 2015. doi:10.1145/2818185.

Cristina Menni, Ana M Valdes, Maxim B Freidin, Carole H Sudre, Long H Nguyen, David A Drew, Sajaysurya Ganesh, Thomas Varsavsky, M Jorge Cardoso, Julia S El-Sayed Moustafa, Alessia Visconti, Pirro Hysi, Ruth C E Bowyer, Massimo Mangino, Mario Falchi, Jonathan Wolf, Sebastien Ourselin, Andrew T Chan, Claire J Steves, and Tim D Spector. Real-time tracking of self-reported symptoms to predict potential COVID-19. *Nat. Med.*, 26, 2020. doi:10.1038/s41591-020-0916-2.

Mohamed F. Mokbel, Thanaa M. Ghanem, and Walid G. Aref. Spatio-temporal access methods. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 26:1–7, 2003.

G.M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, International Business Machines, 1966.

Stephen M. Omohundro. Five balltree construction algorithms. Technical report, International Computer Science Institute, 1989.

Jan Van Sickle. *Basic GIS coordinates*. CRC Press, 2004.

Ryan Tatton. sharetrace-giraph GitHub repository. `https://github.com/cwru-xlab/sharetrace-giraph`.

Ryan Tatton, Erman Ayday, Youngjin Yoo, and Anisa Halimi. Sharetrace: Contact tracing with the actor model. In *2022 IEEE International Conference on E-health Networking, Application & Services (HealthCom)*, pages 13–18, 2022. doi:10.1109/HealthCom54947.2022.9982762.

The Ray Team. Anti-pattern: Over-parallelizing with too fine-grained tasks harms speedup, a. `https://docs.ray.io/en/latest/ray-core/patterns/too-fine-grained-tasks.html`.

The Ray Team. Pattern: Using a supervisor actor to manage a tree of actors, b. `https://docs.ray.io/en/latest/ray-core/patterns/tree-of-actors.html`.

The Ray Team. Ray v2 architecture. Technical report, 2022. `https://docs.google.com/document/d/1tBw9A4j62ruI5omIJbMxly-la5w4q_TjyJgJL_jN2fI`.

Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), 1990. doi:10.1145/79173.79181.