Caching Reverse-Geocoded Locations on Smartphones

Thomas Phan, Albert Baek, Abhishek Singh, and Zheng Guo
Samsung Research America - Silicon Valley, San Jose, CA USA

***Abstract*—Photo-driven smartphone usage often requires invoking a server to perform reverse-geocoded location lookup, an approach that incurs network delay and increased power consumption for communication. A solution we consider is to cache the reverse-geocoding results so that the location of subsequent photos taken within some range of previously reverse-geocoded photos can be looked up in the cache without having to access a server. The key challenge, then, is effectively defining and caching the reverse-geocoded regions on the smartphone to enable accurate lookups. We propose three schemes and evaluate their performance over a photo data set taken across 14 cities.**

I. Introduction

Smartphone camera usage has increased dramatically within the last several years, with estimates of millions of photos being taken by smartphones on a daily basis [2]. A popular use case for such photos entails the on-device grouping of photos into albums based on the geolocation (e.g. the city) where each photo was taken [3][7]. Such groupings are enabled by the fact that when a photo is taken, the geocoordinates of the smartphone, stated in terms of latitude and longitude acquired from GPS or cellular trilateration, are embedded into the photo as metadata, such as within the EXIF headers of JPEG images.

A server invocation is then typically made to perform reverse-geocoding, which is the process of translating latitude and longitude coordinates to a physical location. Once the reverse-geocoding process completes for the photos, they can then be grouped together by city to form different albums for the user to browse. However, a problem incurred by this approach is that if the user is out taking photos, such as on a weekend trip, and wants to form groupings, then invoking a server-side process for reverse-geocoding lookup of one or a few photos at a time incurs network delay and increased power consumption over cellular radio (e.g. 3G/4G) or Wi-Fi.

A straight-forward solution is to perform the reverse-geocoding on the phone itself by using city boundary shape data and performing a point-in-polygon test for each photo. The resulting problem is that city boundaries are extremely fine-grained and defined by many points; for example, the city of San Jose, CA, has its official boundary defined by over 19,000 geocoordinate points [8], and determining point inclusion in an N-point polygon requires $O(N)$ time [5]. For a metropolitan region with many closely-packed incorporated cities (such as the San Francisco Bay Area) preloading detailed boundary data on a phone can consume much space. Similarly, smartphone offline mapping apps such as Garmin's Navigon take up about 1.4 GB for the Western USA.

In this work we propose caching reverse-geocoding results to exploit geolocality. After a user takes a photo and the geocoordinate is determined, a geolocation cache is checked to see if the photo is within some range of a previously reverse-geocoded location. For a cache hit, the new photo is tagged with that inferred location. For a cache miss, a server invocation to perform ground-truth reverse-geocoding is performed for the photo, and that resulting location is added to the cache. The key challenge, then, is effectively defining and caching the reverse-geocoded regions on the smartphone to enable high hit rates and accurate geolocation lookups.

To that end we explored three caching schemes: (1) forming convex hulls around geolocations; (2) searching within a circular boundary of a query point; and (3) hashing points to implicit squares within the geocoordinate space. We evaluated the schemes with geolocation metadata from 4000 photos across 14 cities in Silicon Valley. Our results show that the circular scheme produced the highest *accurate cache hit rate*, however it has a slower search than the $O(1)$ hash scheme.

Others have looked at clustering geocoordinates (e.g. [1]), but all rely on server-side reverse-geocoding. To the best of our knowledge, ours is the first to explore client-side caching of such results.

II. Design and implementation

In our work we address the problem of effectively defining and caching reverse-geocoded locations on a phone in order to reduce the need to invoke an online server for such lookups. We assume an LRU cache that holds 1000 geocoordinates of latitude, longitude pairs, with each being an 8-byte double-precision float, resulting in a 16 KB cache. Note that 1000 geocoordinates may be far more than necessary, as we show later. We evaluated three schemes, as shown in Figure 1.

Convex hull: For a set of points associated with the same city, a bounding hull enclosure can be formed around the outermost points of that set. Consider the left-most portion of Figure 1. Suppose the first photos are taken at points 1, 2, and 3; each is a cache miss because there is no already-cached non-empty convex hull, and so a lookup is needed for all three data points to acquire a ground truth reverse-geocoded location. Assuming all three points are in the same city, a convex hull can then be formed around them. A new query point Q then lies within the enclosing convex hull and is a cache hit, with the point being labelled with the inferred location of the hull. Note that point Q is not added to the cache since it is already inside. Finally, suppose point 4 lies outside the convex hull, resulting in a cache miss and a network lookup; if this point lies in the same city as the other points, then the convex hull can be extended to cover it. If there are h hulls, each with N points on average, then this approach runs in time $O(hN)$.

Radius-formed circle: For a fixed radius r, a bounding circle can be formed around a query point; this simple approach is shown in the center of Figure 1. Suppose points 1 and 2 are already in the cache and another photo is taken at the query point Q. After Q's geocoordinates have been determined, the cache is checked to find the nearest point within distance r of Q, where the distance between two pairs of latitude and longitude coordinates is calculated using a Haversine estimation upon a spherical surface. The result here is point 1, and so the query point Q is assigned the same city as point 1. Note that point Q is not added to the cache since its city location was inferred, not obtained through a ground-

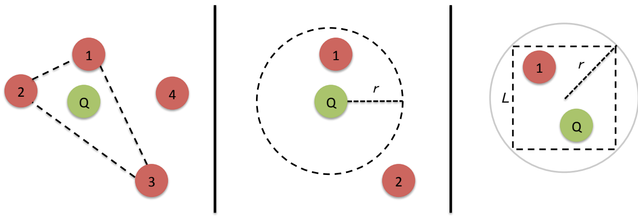

Fig. 1. The three geolocation caching schemes we evaluate in this paper: (left) convex hull; (center) circular boundary; and (right) implicit square hashing.

truth reverse geolocation lookup. If there are N points kept in the cache, then this nearest-neighbor query can be answered in $O(lg\ N)$ on average and $O(N)$ in the worst case if a spatial index such as an R-Tree [4] is used.

Implicit square hashing: In this approach shown in the right side of Figure 1, implicit square boundaries are formed throughout the geocoordinate space. In order to better compare this scheme with the radius-formed circle approach, we define a square boundary based on a radius r that describes a circumscribed circle around the square, where the length of a square side is $L = \sqrt{2} \cdot r$. In turn, this length L in units of meters dictates the needed precision of the cached latitude and longitude points in terms of the number of necessary decimal digits to keep. The two numbers are thus truncated to that number of digits and hashed together in $O(1)$ time to form a 64-bit long integer. In this example, the query point Q is hashed to the same implicit square as point 1, which was already in the cache. Note that point Q is not added to the cache.

III. Experimental results

To evaluate each caching scheme, we looked to simulate user photo-taking behavior and determine the overall cache performance metrics. We downloaded metadata from Flickr.com of photos taken since January 2011 in the Silicon Valley area, covering the bounding box region specified by the latitude and longitude coordinates of (37.3164, -122.2199) to (37.4746, -121.7392), a region that covers approximately 747 kilometers^2 (or 288 miles^2) from western-most Menlo Park to eastern-most San Jose. We specifically queried for photos taken with smartphones, resulting in 4043 photos from 425 users across parts of 14 cities. We evaluated the cache strategies by sequentially iterating over all the photos in order to have the observed relative frequencies more closely approximate the underlying true caching probabilities. To attain deterministic caching results, we sorted all the photos by increasing Unix-epoch timestamp. To get ground-truth reverse-geocoded locations, we used geographical data from OpenStreetMap and the reverse-geocoding functionality of the open-source Nominatim software [6], both of which we installed on a local server.

Figure 2 shows the observed relative frequency for hits in the cache, an approximation of the asymptotic probability $Pr(hit)$. Note that for both the Circle and Hash strategies, a larger radius produces more hits, as expected. Figure 3 shows the relative frequency for accurately-inferred cities given that a hit occurred, approximating $Pr(correct|hit)$. Note here that Convex Hull suffers because cities often jut into the boundary of other cities. Additionally, for Circle and Hash a larger radius will produce more inaccurate results because a broader region increases the chance that the region will include another city's territory. Figure 4 shows the joint relative frequency of both a cache hit and an accurate inference, approximating $Pr(correct, hit) = Pr(correct|hit) \cdot Pr(hit)$. Overall, the

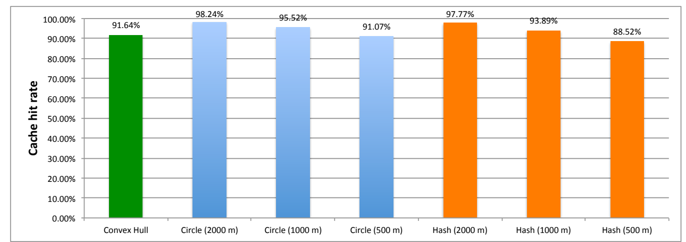

Fig. 2. Observed relative frequency for cache hits. The numbers in parentheses indicate the radius r parameter.

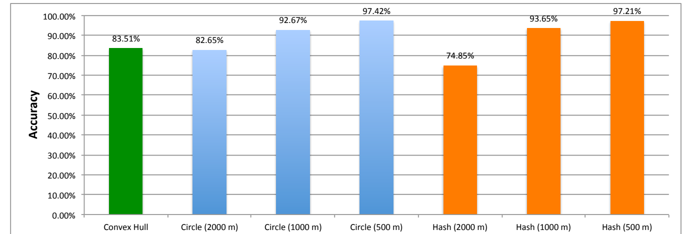

Fig. 3. Observed relative frequency for accurate inferences given a hit.

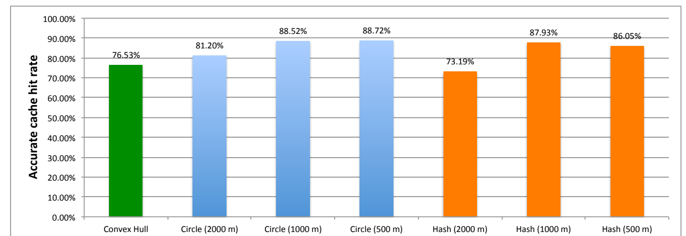

Fig. 4. Observed relative frequency for overall accurate hits.

Circle scheme with a radius of 500m produces the best accurate cache hit rate of 88.72%.

Not shown here is the number of stored points in the caches. After iteration was completed over all the photos, each cache contained less than 500 points for the entire region across 14 cities, which is far less than the 19,000 points needed for defining the boundary of just the city of San Jose.

IV. Conclusion

We explored and evaluated three schemes for defining and caching reverse-geocoded regions on a smartphone client, an approach that reduces the need to invoke a server to perform such lookups. In the future we look to evaluate a larger data set and more thoroughly investigate cache behavior.

References

[1] X. Cao, G. Cong, and C. Jensen. "Mining Significant Semantic Locations from GPS Data," In *Proceedings of VLDB*, 2010.

[2] T. Donegan. "Smartphone cameras are taking over," www.usatoday.com/story/tech/2013/06/06/reviewed-smartphones-replace-point-and-shoots/2373375/

[3] G. Goetz. "Taking, editing, and sharing photos in iOS7," Sept. 21, 2013. gigaom.com/2013/09/21/taking-editing-and-sharing-photos-in-ios7/

[4] A. Guttman. "R-Trees: a Dynamic Index Structure for Spatial Searching," In *Proceedings of ACM SIGMOD*, 1984.

[5] K. Hormann and A. Agathos. "The point in polygon problem for arbitrary polygons," *Computational Geometry*, 20(3), November 2001.

[6] OpenStreetMap. www.openstreetmap.org

[7] S. Perez. "Newly redesigned Cluster makes photo sharing among small groups simpler, more personal," April 23, 2013. techcrunch.com/2013/04/23/newly-redesigned-cluster-makes-photo-sharing-among-small-groups-simpler-more-personal/

[8] San Jose, CA - Official Website, Data Downloads, City Limits, retrieved September 28, 2013. www.sanjoseca.gov/index.aspx?NID=3308