An Optimized Interestingness Hotspot Discovery Framework for Large Gridded Spatio-temporal Datasets

Fatih Akdag
Computer Science Department
University of Houston
fakdag@uh.edu

Christoph F. Eick
Computer Science Department
University of Houston
ceick@cs.uh.edu

***Abstract*—We define interestingness hotspots as contiguous regions in space which are interesting based on a domain expert's notion of interestingness captured by an interestingness function. This paper centers on finding interestingness hotspots on very large gridded datasets which are quite common in scientific computing. Mining large gridded datasets with a lot of variables and measurements requires a scalable framework that can process large amounts of data in an efficient way. In our recent work, we proposed a computational framework which discovers interestingness hotspots in gridded datasets using a 3-step approach which consists of seeding, hotspot growing and post-processing steps. In this paper, we significantly improve the efficiency of the framework by utilizing parallel processing and employing more efficient data structures and algorithms. We propose a novel heap-based hotspot growing algorithm which brings down the cost of hotspot growing phase significantly. In addition, we propose a graph-based preprocessing algorithm which decreases the number of hotspots grown by merging some hotspot seeds. Other improvements to the framework involve incremental calculation of interestingness functions, and growing hotspots in parallel. The improved framework is evaluated in a case study for a very large 4-dimensional gridded air pollution dataset in which we find interestingness hotspots with respect to pollutants.**

***Keywords*—Interestingness Hotspot, Spatio-temporal Data Mining, Interestingness Function, Hotspot Discovery, Hotspot Growing Algorithm, Gridded Dataset**

I. INTRODUCTION

We define interestingness hotspots as contiguous regions in space which are interesting based on a domain expert's notion of interestingness which is captured in an interestingness function. Our research centers on finding interestingness hotspots on very large gridded datasets which are quite common in scientific computing: Many scientific disciplines such as optometry, earth and atmospheric sciences, medicine, and ecology produce large amounts of samples relying on spatial grid-structures that identify locations where measurements are taken. This leads to very large gridded datasets with a lot of variables and a large number of measurements recorded, posing challenging problems on how to store, query, summarize, visualize and mine such datasets. Coping with the mentioned challenges requires a scalable computational framework for processing large amounts of data.

Typically, spatial or spatio-temporal clustering algorithms have been used to analyze such datasets; however, in our recent work [1], we proposed an alternative, non-clustering approach to obtain interestingness hotspots, which grows interestingness hotspots from seed hotspots, and then post-processes the obtained hotspots to remove highly-overlapping hotspots.

In this paper, we improve the proposed hotspot discovery framework using the following approaches:

1) A graph-based preprocessing algorithm is introduced that decrease the number of hotspots grown by merging some hotspot seeds.
2) Another preprocessing algorithm is introduced that eliminates hotspot seeds which are included in an already grown hotspot.
3) A heap-based hotspot growing algorithm is proposed that is much more efficient than the previously used hotspot growing algorithm.
4) Incremental (online) calculation of various interestingness functions are provided to improve the runtime complexity of the hotspot growing algorithms.
5) Parallel processing frameworks are used to speedup hotspot growing phase
6) The improved interestingness hotspot discovery framework is evaluated in a case study involving a challenging, very large 4-dimensional gridded air pollution dataset.

The rest of the paper is organized as follows. In Section 2, we describe the hotspot discovery framework. Section 3 provides a brief discussion of our methodology. We present the improvements on the existing framework in Section 4 and experimental evaluation in Section 5. We review the related work in Section 6 and Section 7 gives a conclusion of the paper.

II. Overview Of Interestingness Hotspot Discovery Framework

In this section, we give a brief description of the framework for discovering hotspots in gridded datasets using interestingness functions. For a more detailed discussion of the framework, we refer to the original paper [1].

A. Framework for Spatio-temporal Interestingness Scoping

Interestingness hotspots are contiguous areas in space for which an interestingness function *i* assigns a reward $w \geq 0$, indicating "news-worthy" regional associations. Our goal is to mine spatio-temporal patterns for performance attributes in a predefined space. The scope of an interestingness hotspot is a contiguous spatio-temporal region for which the association is valid; validity is assessed using interestingness functions. More formally, we assume a gridded dataset O is given in which objects $o \in O$ are characterized by: a set of performance attributes P, a set of spatial attributes S, a set of temporal attributes T, a set of continuous attributes M, which provide meta data under which the performance attributes P are analyzed in the spatio-temporal space. Moreover, we assume a spatial neighboring relationship N is given

$$N \subseteq O \times O$$

that describes which objects belonging to O are neighbors. N is usually computed using spatio-temporal attributes $S \cup T$ of objects in O. Finally, we assume that we have an interestingness measure

$$i: 2^O \rightarrow \{0\} \cup \Re^+$$

that assesses the interestingness of subsets of the objects in O by assigning rewards to a particular cluster H. Moreover, we assume an interestingness threshold θ is given that defines which patterns are interesting.

The goal of this research is to develop frameworks and algorithms that find interestingness hotspots $H \subseteq O$; where H is an interestingness hotspot with respect to *i* if the following 2 conditions are met:

1. $i(H) \geq \theta$

2. H is contiguous with respect to N; that is, for each pair of objects (o,v) with $o,v \in H$, there has to be a path from o to v that traverses neighboring objects (w.r.t. N) belonging to H. In summary, interestingness hotspots H are contiguous regions in space that are interesting ($i(H) \geq \theta$).

B. Example Interestingness Functions

The most simplistic interestingness i_p measure we can think of is one that directly uses the value of a single performance attribute p, which is defined as follows:

$$i_p(H) = \frac{\sum_{h \in H} h.p}{|H|} \qquad (1)$$

where $H \subseteq O$ is an interestingness hotspot, |H| denotes cardinality of H and h.p denotes the value for attribute p for cell h in H.

Another interestingness function considers the correlation of two performance attributes p_1 and p_2; the corresponding interestingness function $i_{corr(p1,p2)}$ is defined as follows:

$$i_{corr(p1,p2)}(H) = \begin{Bmatrix} 0, & if\ |correl(H,p_1,p_2)| < \theta \\ |correl(H,p_1,p_2)| - \theta, & otherwise \end{Bmatrix} \quad (2)$$

where $0 < \theta < 1$ is the interestingness threshold, and $correl(H,p_1,p_2)$ is the correlation of attributes p_1 and p_2 with respect to the grid-cells belonging to hotspot H. This interestingness function is used to find regions in a dataset where the performance attributes p_1 and p_2 are correlated and therefore allows for the identification of regional correlation patterns in spatial datasets which is needed for commercial applications, such as geo-targeting.

Finally, variance interestingness function considers the variance of a performance attribute p and we define the corresponding interestingness function $i_{var\,(p)}$ as follows:

$$i_{var(p)}(H) = \begin{Bmatrix} \theta - variance(H,p), & if\ variance(H,p) < \theta \\ 0, & otherwise \end{Bmatrix} \quad (3)$$

where $\theta > 0$ is the variance threshold, and *variance(H,p)* is the variance of an attribute p with respect to the grid cells that form hotspot H. This interestingness function is used to find regions in a dataset where the performance attribute *p* does not change significantly. The obtained hotspots can be used to generate maps for the performance attribute and for generating prediction models for the performance attribute—similar to regression trees,

C. Computing Interestingness Hotspots

There are two different approaches for finding interestingness hotspots in gridded datasets. First approach is to use clustering techniques for mining gridded datasets. One clustering approach [5] works by dividing the dataset into smaller regions and then merges the pair of neighboring regions which yields the highest overall reward when merged. Merging continues until there are no merge candidates left and returns a set of disjoint regions which have the maximal sum of rewards.

In this research we focus on a hotspot discovery framework, again, we divide the region into smaller regions, however, instead of merging regions, we firstly identify some small regions with high reward as seed regions and then grow these seed regions by adding neighboring objects which increase the reward most when added. Since we grow multiple seed regions, some of them may overlap after growing and we apply post processing to deal with overlapping hotspots. We focus on the latter approach which we believe has a much higher novelty value and more potential to compute "better", more interesting hotspots, as the clustering approach searches for all hotspots in parallel, being forced to make compromises, as switching one sub region from one to another cluster might increase the reward of one cluster but decrease the reward of the other cluster. On the other hand, the main advantage of the clustering approach is that the obtained interesting hotspots are disjoint, whereas the hotspot discovery approach has to deal with overlapping hotspots, which requires a post-processing step that is not necessary for the clustering approach.

III. Overview of Hotspot Discovery Algorithm

In this section, we briefly describe the interestingness hotspot discovery algorithm which works in 3 phases:

1) Find small hotspots with high interestingness
2) Grow the hotspots found in Phase 1
3) Post-process hotspot regions found in Phase 2 by removing highly overlapping hotspots.

A. Phase 1: Seeding Phase

In this phase, we identify small regions with high interestingness in the dataset, which we call "hotspot seeds" and grow these seeds in Phase 2 to obtain larger hotspots. The whole dataset is firstly divided into small sub regions of same sizes and for each region an interestingness value is calculated using the plugin interestingness function. The framework allows user defined seed sizes. We refer to these smaller regions as "seed candidate regions". A "seed interestingness threshold" is used to determine which seed candidate regions will be used to grow larger hotspots. Seed candidate regions having an interestingness value larger than the "seed interestingness threshold" are grown in hotspot growing phase.

B. Hotspot Growing Phase

We proposed a hotspot discovery algorithm in our previous work[1] which grows seed regions by adding a new cell in each step. This method grows hotspots by adding a neighboring grid cell in each step by finding the neighbor which will increase the reward function the most when added to the region. This neighbor is then added into the region and region's neighbors list is updated with the neighbors of the newly added grid cell which are not already a neighbor or already in the region. We memorize the best hotspot obtained so far and its reward value, and update this information when a "hotter" hotspot has been found. We continue adding neighbors as long as the region's interestingness remains positive. Our previous implementation used to stop growing of a hotspot if an improvement cannot be obtained in a predefined number of consecutive steps. We observed that once a hotspot's reward start decreasing, it is quite possible that it start increasing again after a while and it is hard to predict this threshold. Thus, we updated the algorithm to grow the hotspot as long as the hotspot's interestingness remains positive.

Computational complexity of the hotspot growing phase is $O(|H|) \times O(R_H)$ *for each iteration* of a hotspot where |H| is the cardinality of an hotspot and $O(R_H)$ is the runtime complexity of calculating the reward value. In each iteration, reward value with each neighbor is calculated and the number of neighbors is of order O(|H|). As the number of objects increase in each iteration, a hotspot grows from h_0 initial elements to n elements, and if $O(R_H) = O(H)$ the runtime complexity of growing a hotspot is:

$$O\left(\sum_{h=h_0}^{n} h^2\right) = O(n^3) \qquad (4)$$

If the interestingness function is calculated incrementally in O(1) time, then the complexity can be reduced to $O(n^2)$.

Neighborhood definition is a plugin function in the framework. In our current implementation, we define two grid cells as neighbors if only one of the dimensions differ and the difference is 1. Following is the neighborhood definition for a 4D dataset with x, y, z, and t dimensions:

$$Neighbor(o_1, o_2) \Leftrightarrow |o_1.x - o_2.x| + |o_1.y - o_2.y| + |o_1.z - o_2.z| + |o_1.t - o_2.t| = 1 \qquad (5)$$

where o_1 and o_2 are two grid cells and $o_i.x$ corresponds to x dimension value of o_i. Depending on the application domain, various neighborhood definitions can be defined.

C. Phase 3: Post-processing algorithms to remove overlapping hotspots

Hotspot growing phase usually creates large number of hotspots with high degree of overlaps. In this phase, we try to remove specific overlapping hotspots in order to eliminate redundant hotspots. To do this, we set an overlap threshold and find an optimal set of hotspots that overlap less than the threshold value while maximizing the total reward. More formally, the post processing problem can be defined as follows: Given a set of hotspots S, and an overlap threshold λ, find a subset $S' \subseteq S$ for which $\sum_{H \in S} i(H)$ is maximal, subject to the following constraint: $\forall H \in S' \forall H' \in S'$ $\lambda \geq overlap(H,H')$ where degree of overlap is measured by:

$$overlap\ (H, H') = \frac{\text{number of grid cells H and H' have in common}}{\text{number of grid cells in the smaller hotspot}} \qquad (6)$$

The degree of overlap of two hotspots is the ratio of the number of grid cells that are shared between both hotspots to the number of grid cells in the smaller hotspot. For example, if one hotspot has 100 grid cells, and another one has 80 grid cells and they share 60 grid cells, then the degree of overlap is 60/80 = 0.75. In definition (6), the number of grid cells in the smaller hotspot is used in the denominator to make sure that hotspots which are completely contained in another hotspot can be eliminated. Alternatively, the total number of grid cells in both hotspots could be the denominator, however, if hotspot A with 1000 grid cells completely contains all grid cells in the hotspot B which has 100 grid cells, then the overlap ratio would be 100/1100 = 0.09, which implies a very low degree of overlap. We overcame this problem by using definition (6).

The proposed preliminary post-processing algorithm in [1] used to iterate over pairs of hotspots that are overlapping to a degree more than λ and remove the hotspot with the lower interestingness. The algorithm was not finding the optimal set of non-overlapping hotspots with the largest total reward value. We replaced this algorithm with a graph-based algorithm that finds the optimal set of non-overlapping hotspots, however it is a subject of another paper which is not published yet.

IV. Improvements

In this section, we present the improvements for each phase of the hotspot discovery framework.

A. *Seeding Phase*

The seeding phase divides the region into smaller regions of same size and finds the ones with the higher interestingness which are grown in phase 2. However, we observed that many of these smaller regions grow to the same (or quite similar) regions. This is not surprising as a large hotspot with high interestingness will usually have smaller sub-regions with very high interestingness. A Case study reported in [1] justifies that the hotspots created by seed regions that grow to the same region are either same or very close to each other in shape and size. Obviously, it is better to eliminate some of these seed regions before growing them. We propose the following technique to reduce the number of seed regions grown:

1) Find neighboring seed regions

2) Create a neighborhood graph of seed regions where each seed region is a node. Create an edge between nodes if the corresponding regions are neighbors and if the union of these regions yields a region with an acceptable reward value. A merge threshold is used to assess if the union of these regions is acceptable.

3) Merge the seed regions connected by an edge starting with the pair that yields the highest reward gain when merged.

4) Update neighborhood graph after the merge operation. Create an edge between the new node and neighbors of the merged nodes using the same procedure.

5) Continue merging seed regions as long as there are nodes to be merged (i.e. edges) in the graph.

This algorithm is similar to MOSAIC [5] clustering algorithm, however is implemented using more efficient data structures, and creates the neighborhood graph using a undirected graph with weighted edges instead of a Gabriel Graph. We use an additional HashSet data structure to keep a list of nodes instead of an Array to ensure minimum add/remove and containment check time complexity as a HashSet data structure has O(1) time complexity for all of these operations. Moreover, we use a Heap data structure [17] to keep the list of edges. A heap is a special tree structure that satisfies the *heap property:* All nodes are either ‘greater than or equal to’ or ‘less than or equal to’ each of its children, according to a comparison predicate. Heaps are mostly used for priority queues and in this problem we need to prioritize the edges that will be processed. The edge connecting the pair of nodes which results the highest reward gain when merged is the root of the heap. Heap data structure has O(log$n$) time complexity for extract-max operation, so less time is spent for finding the next merge candidate; and O(log$n$) time complexity for add operation as we add new edges for the new region after a merge operation. MOSAIC uses an array structure for keeping the clusters and merge candidates which require O(n) time for all of these operations. Furthermore, we require that the merge operation yields a region with an acceptable reward. That is, we do not want to create hotspots with low rewards as a result of merging two seed regions. This is also consistent with the objective of the post-processing algorithm—to get the set of hotspots with the highest total reward. We use a merge threshold μ to define if the union of seed regions is acceptable. If the reward of the new region is higher than the total reward of merged regions multiplied by μ, then the merge is acceptable:

$$merge(s_1, s_2)\ if\ R(\cup(s_1,s_2)) > (R(s_1) + R(s_2)) * \mu \qquad (7)$$

where $R(s_i)$ represents the reward of seed region s_i. Fig. 1 depicts pseudocode for the seed preprocessing algorithm and Fig. 2 depicts the merge procedure which merges seed regions connected by an edge. We assign the reward gain which is calculated by $R(\cup(s_i,s_j)) - (R(s_i) + R(s_j))$ as the weight of an edge (line 10 in Fig. 1). The edge with the highest reward gain is the root of the heap tree and processed first.

```
1:  Create an undirected graph G and HashSet S of seed regions
2:  foreach seed region $s_i$
3:      Add $s_i$ to G as a vertex
4:      Add $s_i$ to S
5:  end foreach
6:  for i = 0 to number of seed regions – 1
7:      for j = i + 1 to number of seed regions
8:          if $s_i$ and $s_j$ are neighbors and $R(\cup(s_i,s_j)) > (R(s_i) + R(s_j)) * \mu$ then
9:              Create an edge e connecting nodes $s_i$ and $s_j$
10:             e.weight = $R(\cup(s_i,s_j)) - (R(s_i) + R(s_j))$
11:             G.AddEdge(e)
12:         end if
13:     end for
14: end for

15: Create a max-Heap H of edges
16: foreach edge e in G.edges
17:     H.enqueue(e, e.weight)
18: end foreach

19: while H has elements
20:     nextEdge = H.dequeue()
21:     if S contains both nodes connected by nextEdge then
22:         Merge(nextEdge)
23:     end if
24: end while
```

Fig. 1. Pseudo-code for preprocessing algorithm

While processing edges in the order of descending weights, it is possible that an edge which is still in the heap might have been removed from the graph by merge procedure as a result of merging one of the connected nodes in a previous step. That edge might still be in the heap as the heap data structure does not support deleting a particular node. In that case, the dequeue operation will return an edge that does not actually exist in the graph. So, we check if both nodes connected by the edge exist in the set of seed regions to make sure that both nodes survive (line 21 in Fig. 1); otherwise we just skip processing that edge.

```
1: Procedure Merge (Edge e)
2:  Set $s_1$ = e.source, $s_2$ = e.target
3:  Merge $s_1$ and $s_2$ by adding all elements in $s_1$ and $s_2$ in a new region $s_{new}$
4:  Add $s_{new}$ into G as a new vertex
5:  Remove e from G
6:  foreach neighbor $s_i$ of $s_1$ connected by edge $e_i$
7:     if $R(\cup(s_i,s_{new})) > (R(s_i) + R(s_{new})) * \mu$ then
8:        Create an edge $e_{new}$ connecting nodes $s_i$ and $s_{new}$
9:        $e_{new}$.weight = $R(\cup(s_i,s_{new})) - (R(s_i) + R(s_{new}))$
10:        G.AddEdge($e_{new}$)
11:        G.RemoveEdge($e_i$)
12:    end if
13:  end foreach
14:  foreach neighbor $s_j$ of $s_2$ connected by edge $e_j$
15:     if $R(\cup(s_j,s_{new})) > (R(s_j) + R(s_{new})) * \mu$ then
16:        Create an edge $e_{new}$ connecting nodes $s_j$ and $s_{new}$
17:        $e_{new}$.weight = $R(\cup(s_j,s_{new})) - (R(s_j) + R(s_{new}))$
18:        G.AddEdge($e_{new}$)
19:        G.RemoveEdge($e_j$)
20:    end if
21:  end foreach
22:  Remove $s_1$ and $s_2$ from the graph G and HashSet S
23:  Add $s_{new}$ into HashSet S
24: end procedure
```

Fig. 2. Pseudo-code for Merge operation

B. *Hotspot Growing Phase*

In this phase, we present four different optimization techniques and a new hotspot growing algorithm.

1) *Eliminate contained seed regions*

As many seed regions grow to the same hotspot, it is possible to anticipate if an un-grown seed region will grow to a hotspot which was already discovered by growing another seed region. Before growing a seed region, we check if it is already contained in an already grown seed region and if so, we do not grow it. In most cases, this optimization work pretty well and eliminates seed regions which would grow to an already discovered hotspot. However, it is still possible that a seed region eliminated by this method could possibly create a hotspot with a higher interestingness if it were grown. In order to minimize this possibility, we sort seed regions in the decreasing order of initial interestingness and grow them in this order. Obviously, a seed region with a higher interestingness value has a higher chance of growing a hotspot with higher interestingness.

However, it is not trivial to decide if a seed region is contained by a hotspot. In most cases, some grid cells in the seed region are not included in a hotspot which covers the seed region. Thus, we use a containment threshold to decide if the seed region will be grown or not. If the containment threshold is set to 0.9, then 90% of the grid cells in the seed region needs to be included in a hotspot to eliminate this seed region. We use thresholds higher than 0.9 in our framework implementation. This threshold can be set to different values depending on the application domain. We use the algorithm in Fig. 3 to calculate if a hotspot contains a seed region.

```
1: Procedure checkContainment(hotspot, seed)
2:    set maxExcludedAllowed = seed size * (1- ContainmentThreshold)
3:    set numNotIncluded = 0
4:    foreach grid cell in seed
5:       if hotspot does not contain cell then
6:           numNotIncluded = numNotIncluded + 1
7:           if numNotIncluded >= maxExcludedAllowed then
8:               return false
9:          end if
10:      end if
11:   end foreach
12:   return true
13:end procedure
```

Fig. 3. Pseudocode for containment check algorithm

Since we use a HashSet to keep list of grid cells in a hotspot, calculating the number of grid cells in the seed region which are included in a hotspot only takes O(|s|) time where |s| is the size of the seed region. We further optimize this procedure and use the algorithm in Fig. 3 which prematurely stops calculating (in line 8) when a minimum number of cells not included in the hotspots is reached. For two completely separated regions, if the containment threshold is set to 0.9, this procedure returns the result in |s|/10 iterations.

2) *Heap-based hotspot growing algorithm*

In this subsection, we introduce a novel heap-based hotspot growing algorithm. In hotspot growing phase, we search the best neighbor among all neighbors in each step, and after each time we add a new neighbor we do this search again. Searching the most fit neighbor each time takes the complexity of hotspot growing algorithm to O(n^2) at least where n is the number of grid cells in the hotspot, assuming reward function is calculated incrementally. However, we observed that the ordering of the neighbors according to their 'fitness' for the region does not change much as the region grows. If a neighbor n_1 increases the reward more than the neighbor n_2 when evaluated in step s_i, this usually means that n_1 is still a better fit to be included in the region for another step s_j. There are some cases in which this does not hold; if the attributes of n_1 and n_2 are very close, as the region grows n_2 may become a better fit for the region. However, such cases occur very rarely and do not affect the final hotspot dramatically as both neighbors are generally either included in or excluded from the hotspot. Thus, it is redundant to evaluate each neighbor in each step of the growing phase. Instead, we use a max-heap data structure to keep the list of neighbors where the neighbor with the highest fitness value is the root of the heap tree. Using a max-heap, instead of searching for the best neighbor in each step, we simply add the root node into the region. When new neighbors are encountered as a result of growing the hotspot, we assign each new neighbor a fitness value by evaluating the reward gain in case the neighbor is added to the region and add it into the heap using the reward gain as the priority. Next, we continue growing the region as long as there are more neighbors and the interestingness of the region is higher than the interestingness threshold.

Heap data structure has O(log$n$) time complexity for extracting the root node (extract-max operation), insertion and deletion operations. Some specialized heap implementations (Binomial heaps and Fibonacci heaps [17]) allow amortized O(1) time complexity for insertion. Figure 4 gives the pseudocode of heap based hotspot growing algorithm. This procedure is called in a loop as long as the interestingness of the region remains positive and there are more neighbors in the heap. The best reward found is memorized (lines 11-14) and when hotspot growing is finished, the hotspot is set back to this state. The runtime complexity of the heap-based hotspot growing algorithm is $O(n\log n)$ as a total of $O(\log n)$ time is spent in each step where n is the number of objects in the hotspot—$O(\log n)$ time is spent for finding the best neighbor and removing it from neighbors list, plus O(1) for adding it to the region and $O(\log n)$ for adding neighbors of newly added object to the neighbors list. Before adding the neighbors of the newly added object to the heap—there are at most 8 neighbors according to the neighborhood definition (5)—we need to ensure that the neighbor is not contained in the region or in the neighbors list. To optimize this containment check time complexity, we keep the objects in the region in a HashSet. Furthermore, in the implementation of max-heap data structure, we put all elements in the heap into an additional HashSet data structure and manage them together to minimize containment check operation time complexity.

```
1:  Procedure AddNextNeighbor(region)
2:      set bestNeighbor = Heap.dequeue()
3:      add bestNeighbor to region
4:      set newReward = CalculateReward(region)
5:      foreach neighbor n of bestNeighbor
6:          if n is not in region and n is not in the neighbors list then
7:              set fitness = CalculateFitness(region, n)
8:              Heap.enqueue(n,fitness)
9:          end if
10:     end foreach
11:     if newReward > region.alltimeBestReward then
13:         set region.alltimeBestReward = newReward
12:         set region's alltimeBestGridCells = region.currentGridCells
14:     end if
15: end procedure
```

Fig. 4. Pseudocode for heap-based hotspot growing algorithm

This optimization requires choosing a good fitness measure for each neighbor of the growing hotspot. New neighbors are evaluated with a different state of the growing hotspot. Thus, it is very important to put the new neighbors in a correct order in the list of neighbors ordered by decreasing fitness values. A sample fitness function is given in the experimental evaluation section.

3) *Incremental calculation of interestingness functions*

As mentioned in Section 3, calculating the interestingness function incrementally dramatically decreases the runtime complexity of hotspot growing phase. In this subsection, we show how to calculate interestingness functions incrementally. Incremental calculation means that as new data comes, a function's output when applied on a dataset is calculated without going over the previous data. For example, count or sum of elements in a dataset can be calculated incrementally by just increasing the value of a variable, instead of counting or adding all numbers again. It should be noted that some functions cannot be calculated incrementally. Aggregate functions are categorized as distributive, algebraic or holistic [18]. Distributive functions can be computed in a distributive manner by partitioning the data into subsets, and aggregating the result of applying the function to each subset. Sum, count, min, max are some example distributed functions. Algebraic functions can be computed by applying an algebraic function on a constant number of distributive functions. For example, average function is an algebraic function as it can be computed by sum/count. Holistic functions cannot be computed by applying an algebraic function on a constant number of distributive functions, thus all data needs to be processed together. As a result, unlike distributive and algebraic functions, holistic functions cannot be calculated incrementally. Median and rank are examples to holistic functions. In this section we describe a methodology for implementing a given algebraic or distributive function incrementally, using variance function as an example.

Given an empty set of objects R, we are supposed to calculate the variance of attribute a incrementally while adding new objects into R where each object has an attribute a. It is shown in the literature [19, 20] that variance can be calculated incrementally by updating the mean and sum of squared differences (M_2) with each new value x using the following equations:

$$\text{mean}_n = \text{mean}_{n-1} + (x - \text{mean}_{n-1}) / n \text{ , and}$$

$$M_2 = M_2 + (x - \text{mean}_{n-1}) \times (x - \text{mean}_n)$$

Then the variance can be calculated in O(1) time by:

$$\text{variance}(R, a) = M_2 / (n-1)$$

where n is the number of objects in the dataset. We use the same equation in our code to calculate variance in O(1) time when a new grid cell is added to a region. From a software design perspective, we create a new class named "StatsCalculator" which keeps a reference to n, mean and M_2. Each region has its own copy of StatsCalculator object. Each time we add a new object to a region, we call the procedure "Add(x)" in StatsCalculator class, which updates these values using x. When we need to retrieve the variance value for the region, we call "variance()" procedure which just returns M_2/ (n-1).

Calculating correlation function incrementally is similar but more complicated. We will not present the details as space is limited and refer to literature [20].

4) *Parallel processing of hotspot growing phase*

Hotspot growing phase can be easily processed in parallel by assigning a seed region to be grown for each processor. In our implementation we use a shared memory parallel programming approach. The framework was implemented using .NET Framework, and we use TPL (task parallel library) which is Microsoft's shared memory parallel programming implementation for the .NET framework. A parallel for loop is used in our implementation which assigns seed regions to multiple threads as threads become available. The framework itself optimizes the number of threads created to complete the loop as fast as possible while trying to optimize system resources and assigns tasks to threads when threads become available.

V. Experimental Evaluation

In this section, we present experiments in which we evaluate the effectiveness of the algorithms we propose in this paper. We evaluated our framework on a very large 4-dimensional gridded air pollution dataset. We find high correlation hotspots and low variance hotspots with respect to pollutants in the dataset. The air pollution dataset [6] provided by EPA (Environmental Protection Agency) contains 132 air pollutant observations divided to 4km by 4km square grids on latitude and longitude dimensions, and 27 layers of various heights on altitude dimension. The total area is covered by 84 grid cells on latitude dimension which correspond to the columns of the grid, and 66 grid cells on longitude dimensions which correspond to the rows of the grid. The concentrations of air pollutants and meteorological variables are recorded in each hour for each grid cell. Storing all observations for a day in raw binary format takes 1.8 GB memory space. For the case study, we use a subset of grid cells covering the city of Houston which includes 26 columns, 19 rows and 27 layers (13338 grid cells). One day of air pollution data covering the city of Houston includes 320112 observations for each of 132 air pollutants in the dataset.

In the next subsections, we will measure the improvements obtained by each optimization. We ran all test on a Windows computer with 4 processors.

A. *Merging Seed Regions*

We run hotspot discovery algorithms on air pollution data to find low variation hotspots with different parameter settings. We run test on different timeframes and merge thresholds and measured the number of seed regions found, number of seed regions grown and the total reward and average variance of hotspots obtained after eliminating overlapping hotspots. Table 1 lists the parameters applied, and Table 2 lists the results for each test. We use the variance interestingness function defined in (3) and set the interestingness threshold to 0.65 and following reward function is used for evaluating the quality of a region R:

$$\varphi(R) = interestingness(R) \times \text{size}(R)^{\beta} \qquad (8)$$

where $\beta>1$ is a parameter determining the degree preference for larger regions. We set β to 1.01 as we prefer smaller hotspots with high interestingness to larger hotspots with low interestingness. We set seed size to 3x3x3x1 for Test 1 and Test 2 and to 3x3x3x3 for Test 3 as Test 3 dataset is 4-dimensional. Seed threshold was set to 0.65 for the Test 1 ad Test 2 and 0.5 for Test 3.

TABLE I. TEST PARAMETERS

Test#	Dataset Date	Timeframe	Merge Threshold
1	2013-09-01	12am (1hr)	0.96
2	2013-09-01	12am (1hr)	0.60
3	2013-09-01	6am-6pm (12hr)	0.96

TABLE II. TEST RESULTS

	Test 1	Test 2	Test 3
Seeds Found	27	27	233
Seeds Grown	23	17	79
Hotspots Found	8	7	9
Hotspots found without merging	8	8	9
Average hotspot variance	0.275	0.290	0.219
Average hotspot variance without merging	0.269	0.269	0.207
Total reward	1203	1163	63049
Total reward without merging	1206	1206	63544

Merging seed regions affect the number of seed regions grown. It is expected that the reward of hotspots may not be as high when some seed regions are merged as the merge operation causes some grid cells to be included in the merged region which would not be included otherwise. The test results show that the loss of total reward and the variance increase are not significant. As shown in Test 2 results, decreasing the merge threshold obviously causes more loss in the total reward (loss is 43 compared to 3 in Test 1) as less compatible regions are merged. The runtime of the seed preprocessing phase in all tests were much lower than a second, taking at most 0.2 seconds in Test 3 in which the number of seed regions reduced from 233 to 79 after 154 successive merge operations were applied. Figure 5 shows some of the seed regions which were computed by preprocessing seed regions. As shown in the figure, there are many large regions which are composed of smaller seed regions.

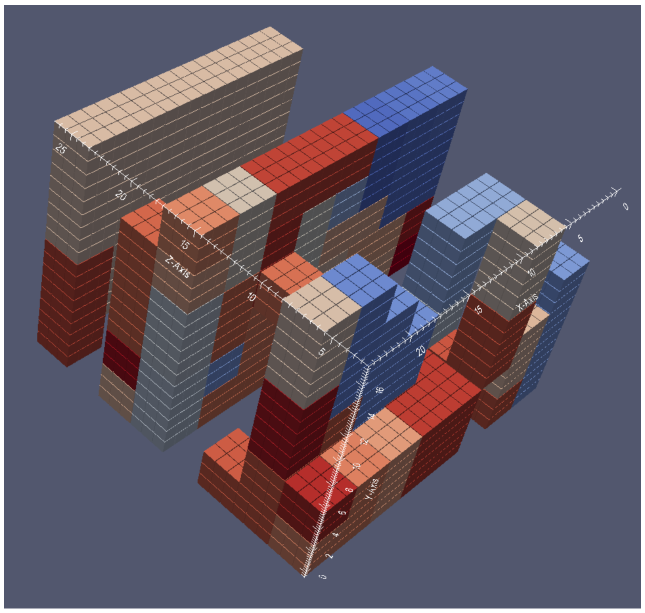

Fig. 5. Merged seed regions.

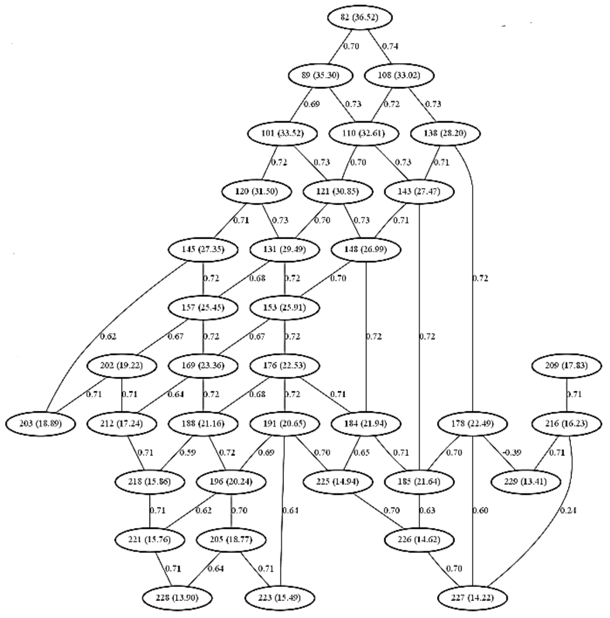

Fig. 6. Neighborhood graph of a set of seed regions

Figure 6 shows a part of the neighborhood graph for a set of seed regions that were merged by the preprocessing algorithm. A grid structure is clearly visible in the neighborhood graph. Vertices show each seed regions' index and reward in parenthesis. Edge weights (edge weight=reward gain, not gain ratio) are given next to each edge. There is an edge between two vertices if their merge is acceptable (i.e. gain ratio is larger than merge threshold).

The results show that preprocessing algorithm succeeds in decreasing the number of seed regions dramatically without affecting the final set of hotspots discovered when a large merge threshold is used.

B. *Eliminate contained seed regions*

In this subsection, we will assess the effectiveness of eliminating seed regions that are contained by an already grown hotspot. We ran Test 1 described in the previous section without running the seed preprocessing algorithm, only to see the effects of this optimization by itself. When we set the containment threshold to 0.9, number of seed regions grown decreased from 27 to 17 as 10 hotspots were eliminated by this optimization. After the post-processing phase, only 8 hotspots survived and the total reward of those hotspots was 1205, which would be 1206 if no optimization would be done.

Next, we ran the same test by firstly preprocessing seed regions and then eliminating contained seeds; the first step decreased the number of seed regions to 23 as in Section 5.1 and out of these 23 regions another 6 of them were eliminated by this optimization as some seed regions were included in the grown hotspots. The total reward was still 1203 as in Section 5.1. This optimization is definitely very effective in eliminating redundant seed regions.

All of the optimizations in our framework are optional, so they can be used together or one can be chosen over another one. In this case, the end result did not change dramatically. Both algorithms succeeded in choosing the right set of seed regions to be eliminated.

C. *Heap-based hotspot growing algorithm*

In this subsection, we evaluate the heap-based hotspot growing algorithm and compare it with the previously proposed hotspot growing algorithm. We chose three sample hotspots from test cases in Section 5.1, ran the legacy hotspot growing algorithm and the new heap-based algorithm while calculating the reward function incrementally. We used the following fitness function for a neighbor n_i of hotspot H:

$$fitness\ (n_i) = (R(H \cup n_i) - R(H)\ ) / |H| \qquad (9)$$

where $R(H \cup n_i)$ represents the new reward when n_i is added to the hotspot and |H| is the hotspot size. This fitness function is used with variance interestingness function defined in (3) and reward function defined in (8). When there are more objects in the hotspot, adding a neighbor effects the reward much less compared to a neighbor that is added when the hotspot was smaller. This is due to the nature of variance function which measures the average of the squared differences from the mean, and calculated by dividing the squared differences to the number of elements; thus dividing the reward gain to hotspot size is required to fairly compare the neighbors. Table 3 lists the test results.

TABLE III. **TEST RESULTS OF COMPARING HEAP-BASED GROWING ALGORITHM AND TRADITIONAL GROWING ALGORITHM**

Hotspot #	**1**	**2**	**3**
Final hotspot size (heap-based)	15242 cells	790 cells	3121
Final hotspot size (legacy)	15641 cells	832 cells	3165
Runtime (heap-based)	5.9 seconds	23 ms	0.243 sec
Runtime (legacy)	170 seconds	525 ms	8.8 sec

Table 3 shows that the efficiency of heap-based hotspot growing algorithm is significantly higher than the traditional algorithm. This is expected as heap-based hotspot growing algorithm has $O(n\log n)$ complexity compared to $O(n^2)$ complexity of the legacy hotspot growing algorithm. Moreover, the number of cells in the grown hotspots are almost same for both algorithms which shows that the employed fitness measure did work well.

D. *Incremental (online) calculation of interestingness functions*

In this section we evaluate the performance gain by calculating the variance interestingness incrementally. First 2 columns in Table 4 was reported in our previous work in runtime analysis section. The growing time with the incremental calculation using legacy and new hotspot growing algorithms are added in the 3rd and 4th columns.

TABLE IV. HOTSPOT GROWING PHASE EXECUTION TIMES FOR THE 3D DATASET USING CORRELATION INTERESTINGNESS FUNCTION

hotspot size (grid cells)	Growing time – non-incremental (sec)	Growing time – incremental (sec)	Using heap
450	2	0.69	0.001
1251	25	1.47	0.058
2082	111	3.34	0.098
3933	705	7.1	0.3

Growing times of the hotspot growing algorithm using the incremental reward calculation algorithm is dramatically faster than the non-incremental reward calculation as the runtime complexity decreases from $O(n^3)$ to $O(n^2)$. Moreover, using the heap-based growing algorithm along with incremental calculation makes the growing time orders of magnitude faster (0.3 seconds compared to 705 seconds for 3933 grid cells).

E. *Parallel Processing of Hotspot Growing Phase*

In this subsection we present the runtime improvements obtained by growing hotspots in parallel. We grew all 79 seed regions that were found in Test 3 in Section 5.1 using sequential and parallel processing and recorded the growing times for each hotspot and total growing time for all hotspots. We ran the test 5 times and took the average of runtimes. We used a computer with 4 processors and as shown in Table 5, parallel processing speedup is 977/354 = 2.75 and parallel efficiency is 2.75/4 = 0.6875. Although the total running time improves using parallel processing, it takes about 15 times slower to grow a hotspot as too many threads are running at the same time sharing system resources.

TABLE V. PARALLEL AND SEQUENTIAL GROWING TIMES

	Sequential	Parallel
Average growing time of a hotspot	12.2 sec.	184 sec.
Total growing time of all hotspots	977 sec.	354 sec.

VI. RELATED WORK

Kulldorff [13] introduced basic spatial scan statistics to search spatiotemporal circular regions occurring within a certain time interval which obtains cylinder-shaped hotspots by growing cylinders from a point of origin by increasing the radius and height of the cylinder. Iyengar [9] extended basic spatial scan statistics by using flexible square pyramid shapes instead of cylinders for spatiotemporal clusters that can grow, shrink or move over time.

There are many clustering algorithms capable of computing spatio-temporal hotspots. However, in a clustering approach all hotspots are obtained in a single run of the clustering and the obtained clusters are always disjoint in contrast to hotspot growing approaches. Many of these algorithms extend DBSCAN [8] for performing spatiotemporal clustering. Wang et al. [15] proposed two spatiotemporal clustering algorithms called ST-DBSCAN and ST-GRID. ST-DBSCAN introduces the second parameter of temporal neighborhood radius in addition to the spatial neighborhood radius in DBSCAN. ST-GRID is a grid-based clustering approach, which operates on the spatiotemporal observations in a 4D-gridstructure. Birant [2] et al. also improved DBSCAN for spatiotemporal clustering and applied it to discover spatiotemporal distributions of physical seawater characteristics in Turkish seas. A density factor is assigned to each cluster for detecting some noise points when clusters of different densities exist. The density factor of a cluster captures the degree of the density within a particular cluster. Joshi [11] et al. proposed a spatiotemporal polygonal clustering algorithm, called STPC, which extends the DBSCAN algorithm to cluster spatiotemporal polygons by redefining the neighborhood of a polygon as the union of its spatial and temporal neighborhoods. When calculating spatial neighbors for a polygon, the temporal aspect was reduced to a fixed interval or time instance and was therefore constant. Moreover, the spatial dimension was instead held to a constant space when calculating temporal neighbors of a polygon.

Another popular clustering algorithm SNN [7] (Shared Nearest Neighbor) has also been extended for spatio-temporal clustering by researchers. Oliveira et al. [14] proposed an algorithm called 4D+SNN which allows the integration of space, time and semantic attributes into the clustering process. This algorithm is able to deal with different data sets and different discovery purposes as the user has the ability to weight the importance of each dimension in the discovery process. Furthermore, two other spatiotemporal clustering algorithms, called Spatiotemporal Shared Nearest Neighbor clustering algorithm (ST-SNN) and Spatiotemporal Separated Shared Nearest Neighbor clustering algorithm (ST-SEP-SNN) [16], were proposed to cluster overlapping polygons that can change their locations, sizes and shapes over time. Both ST-SNN and ST-SEP-SNN are also generalizations of the SSN clustering algorithm.

All spatio-temporal clustering algorithms that were discussed so far compute clusters based on only the distance information. A new group of clustering algorithms has been introduced in the literature which find contiguous clusters by maximizing plug-in interestingness functions similar to the approach used in this paper. These algorithms are capable of considering non-spatial attributes in objective functions that drive the clustering process. They maximize the sum of the rewards for each cluster based on a cluster interestingness function to compute clusters. CLEVER [3,4] is a k-medoids-style [12] clustering algorithm which exchanges cluster representatives as long as the overall reward grows, whereas MOSAIC[5] is an agglomerative clustering algorithm which starts with a large number of small clusters, and then merges neighboring clusters as long as merging increases the overall interestingness. We used an algorithm similar to MOSAIC for pre-processing the seed regions.

VII. Conclusion

In this paper, we optimized our computational framework for mining very large gridded spatio-temporal datasets. Our framework grows hotspots from seed regions using plugin interestingness and reward functions. Our approach is quite different from traditional hotspot discovery algorithms. To the best of our knowledge, this is the only hotspot discovery algorithm in the literature that grows seed regions using a reward function. We claim that the proposed framework is capable of identifying a much broader class of hotspots, which cannot be identified by traditional distance-based clustering algorithms. We plan to compare our framework with other approaches in a future work.

In this paper, we presented very efficient preprocessing algorithms to eliminate redundant seed regions. The experimental evaluation section revealed that the preprocessing algorithms succeeded in eliminating redundant seed regions without affecting the quality of the discovered hotspots. Moreover, we presented a new heap-based hotspot growing algorithm which improved the runtime efficiency of the hotspot growing phase significantly. Parallel processing of hotspot growing phase along with incremental calculation of interestingness functions dramatically reduced the total time required to discover hotspots. The experimental evaluations show that the total speedup is in the orders of magnitude.

On the other hand, our framework is extensible. The proposed algorithms are general and can be applied to various kinds of data such as point sets and polygons. We will work on adapting our framework to work with different types of spatial data rather than just gridded datasets.

References

[1] F. Akdag et al. “A computational framework for finding interestingness hotspots in large spatio-temporal grids” in *Proceedings of the 3rd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. 2014, pp. 21-29.

[2] D. Birant and A. Kut. “ST-DBSCAN: An algorithm for clustering spatial–temporal data”. *Data & Knowledge Engineering* 60.1. 2007, pp. 208-221.

[3] Z. Cao et al. “Analyzing the composition of cities using spatial clustering.” *in Proceedings of the 2nd ACM SIGKDD International Workshop on Urban Computing*. ACM, 2013.

[4] C. S. Chen et al. “Design and Evaluation of a Parallel Execution Framework for the CLEVER Clustering Algorithm” in *PARCO*. 2011, pp. 73-80.

[5] J. Choo et al. “MOSAIC: A proximity graph approach for agglomerative clustering” in *Data Warehousing and Knowledge Discovery*. Springer Berlin Heidelberg. 2007, pp. 231-240.

[6] EPA. (2015, June 30), *Air Data Web Site*. [Online] Available: http://www.epa.gov/airquality/airdata/

[7] L. Ertöz et al. “Finding clusters of different sizes, shapes, and densities in noisy, high dimensional data” in *SDM*. 2003, pp. 47-58.

[8] M. Ester et al. “A density-based algorithm for discovering clusters in large spatial databases with noise” in *KDD*, 1996, vol. 96, no. 34, pp. 226-231.

[9] V. S. Iyengar. “On detecting space-time clusters.” in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2004, pp. 587-592.

[10] R. Jiamthapthaksin et al. “GAC-GEO: a generic agglomerative clustering framework for geo-referenced datasets” in *Knowledge and Information Systems*, 2011, 29(3), pp. 597-628.

[11] D. Joshi et al. “Spatio-temporal polygonal clustering with space and time as first-class citizens” in *GeoInformatica*, 2013, 17(2), pp 387-412.

[12] L. Kaufman, and P. Rousseeuw. “Clustering by means of medoids” in *Statistical Data Analysis Based on the L1 Norm and Related Methods*, North-Holland, Amsterdam, 1987, pp. 405–416.

[13] M. Kulldorff. “A spatial scan statistic” in *Communications in Statistics-Theory and Methods*, 1997, 26(6), pp. 1481-1496.

[14] R. Oliveira et al. “4D+SNN: A Spatio-Temporal Density-Based Clustering Approach with 4D Similarity” in *Data Mining Workshops (ICDMW), IEEE 13th International Conference on Data Mining Workshops,* 2013, pp. 1045-1052.

[15] M. Wang et al. „Mining spatial-temporal clusters from geo-databases”. In *Advanced Data Mining and Applications,* Springer Berlin Heidelberg, 2006, pp. 263-270.

[16] S. Wang et al. “New Spatiotemporal Clustering Algorithms and their Applications to Ozone Pollution” in *Proc. 8th International Workshop on Spatial and Spatio-Temporal Data Mining*, IEEE, 2013.

[17] T. H. Cormen et al. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2009.

[18] S. Shekhar and S. Chawla. “Spatial databases: a tour”. Upper Saddle River, NJ: prentice hall, 2003.

[19] B. P. Welford. “Note on a method for calculating corrected sums of squares and products” in *Technometrics,* 1962, pp. 419-420.

[20] P. Pébay. "Formulas for robust, one-pass parallel computation of covariances and arbitrary-order statistical moments" in *Sandia Report SAND2008-6212, Sandia National Laboratories*. 2008.