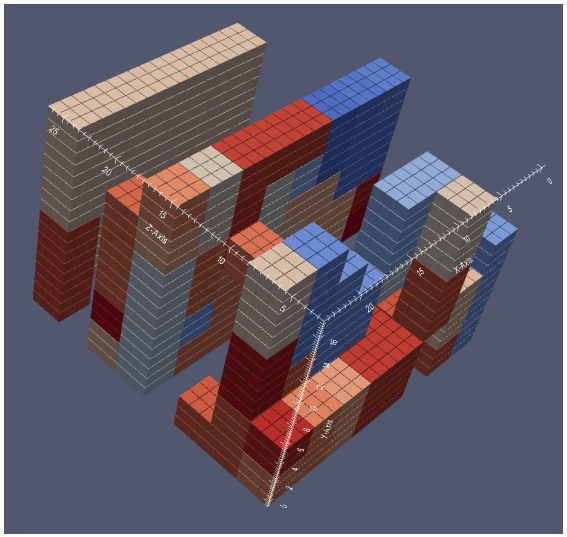
An Optimized Interestingness Hotspot Discovery Framework for Large Gridded Spatio-temporal Datasets

Fatih Akdag and Christoph F. Eick
Department of Computer Science, University of Houston, U.S.A

Talk Organization

1. Introduction
2. Interestingness Hotspot Discovery
3. Optimizations
4. Experimental Evaluation
5. Summary and Conclusion



1. Introduction

- Large amounts of data stored in gridded datasets/raster datasets.
- Hundreds of variables stored in each grid.
- It is challenging to store, query, summarize, visualize, mine such datasets.
- A common problem: How to find interesting contiguous regions in gridded datasets?
- Our approach: Non-clustering approach to obtain interestingness hotspots using plug-in reward functions:
  - Novel hotspot discovery framework and algorithms
  - Capable of identifying much broader class of hotspots than traditional distance based clustering algorithms
  - We find high correlation and low variance hotspots w.r.t. pollutants.
  - Optimized algorithms and efficient data structures.

Talk Organization

1. Introduction
2. Interestingness Hotspot Discovery Framework
3. Optimizations
4. Experimental Evaluation
5. Summary and Conclusion

2. Interestingness Hotspot Discovery Framework

Problem Definition:

Given

1. Dataset O
2. Neighborhood relation $N \subseteq O \times O$
3. Interestingness measure $i: 2^O \rightarrow \{0\} \cup \Re^+$

The goal of this research is to develop frameworks and algorithms that find interestingness hotspots $H \subseteq O$; 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 ($i(H) \geq \theta$).

Example Interestingness Functions

Reminder: Our goal is to find interestingness hotspots H, with respect to interestingness functions i which evaluates
the "*newsworthiness*" of H.

1. Correlation interestingness function:

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

2. Variance interestingness function:

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

where ? is a threshold value, p_1 and p_2 are attributes.

Neighborhood Definition for Grids

Neighborhood Definition 4D-Grid dataset with x, y, z, and t dimensions:

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

where o_1 and o_2 are two grid cells and $o_i.x$ corresponds to x dimension value of o_i.

- Neighborhood definition is a plugin function. Can be changed based on application domain.
- For a grid cell, there is a total of 8 neighbors in 4D and 6 neighbors in 3D according to definition (3).

Hotspot Growing Algorithm

A. Seeding phase:

1. Divide dataset into smaller regions of same size called seed candidate regions
   - For a 4 dimensional dataset with 10×10×10×10 grid cells, dividing the dataset into smaller regions of 2×2×2×2 grid cells yields $(10 / 2)^4 = 625$ such regions.
2. Identify seed regions with high interestingness value and grow them in the next step

B. Growing phase

1. Find the neighbor of the seed region which increases a reward function most when added.
2. Add this neighbor into region and update neighbors list. (A hash set data structure is used to keep neighbors list. Hash set has O(1) runtime complexity for add/remove/contains operations)
3. Continue adding more neighbors (repeat 1&2) as long as the interestingness is positive.

Legacy Growing Algorithm Pseudo Code

```
FUNCTION AddBestNeighborForRegion(region)

        SET bestNewReward = -1
        SET bestNeighbor = null

        FOREACH neighbor of region
                Add Neighbor to region
                SET reward = CalculateReward(region)
                IF reward > bestNewReward  THEN
                        SET bestNewReward = reward
                        SET bestNeighbor = neighbor
                ENDIF
                Remove Neighbor from region
        ENDFOREACH

Add bestNeighbor to region
...
```

Remark: The interestingness of hotspot of size H+1 has to be computed repeatedly with each neighbor in each step.

Hotspot Post-processing

C. Post-processing step: remove specific overlapping hotspots:

Input: a set of hotspots H,

an overlap threshold λ

Find a subset $H' \subseteq H$ for which

$\sum_{h \in H} i(H)$ is maximal,

subject to the following constraint: $\forall h \in H' \forall h' \in H'$ $\lambda \geq overlap(h,h')$

overlap(h,h') = (number of grid-cell h and h' have in common)/(total number of grid-cells in h and h')

- We developed a graph-based post-processing algorithm which uses the maximum weight independent set algorithm to find the optimum solution. It is not published yet.

Talk Organization

1. Introduction
2. Interestingness Hotspot Discovery Framework
3. Optimizations
4. Experimental Evaluation
5. Summary and Conclusion

Optimization 1: Merge Seeds

- Many of the seed regions grow to the same regions
- Eliminate seed regions by merging some
- Procedure:
  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.

Optimization 1: Merge Seeds

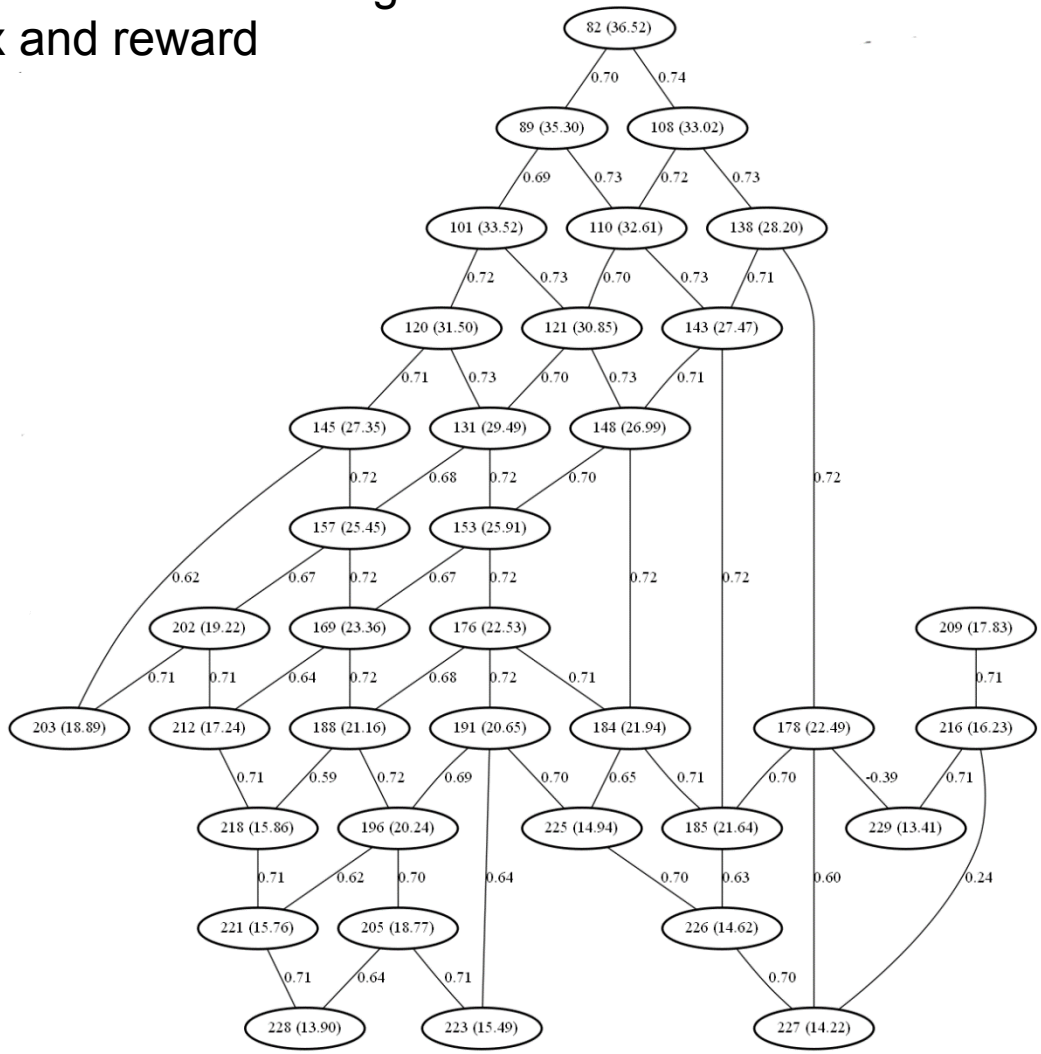
```
1:  Create an undirected graph G and HashSet S of seed regions
2:  foreach seed region s_i
3:      Add s_i to G as a vertex
4:      Add s_i to S
5:  end foreach
6:  for i = 0 to number of seed regions – 1
7:     for j = i + 1 to number of seed regions
8:        if s_i and s_j are neighbors and R(U(s_i,s_j))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

Seed Neighborhood Graph

Edge weights = reward increase when merged
Vertices show seed index and reward



Optimization 2: Eliminate contained seeds

- Before growing a seed, check if any grown hotspot already contains it and if so, do not grow it.
  - 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.
  - Stop calculating (in line 7-8) 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

```
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
```

Optimization 3: Heap-based Growing Algorithm

- Instead of searching for the best neighbor in each step, put them into a heap (priority queue) based on their fitness value and choose the one with the best fitness in each step.
- O(lgn) time algorithmic complexity for extract-max(), and insert() operations.
- Fitness calculation affects the quality of hotspot.
- Sample fitness function for variation interestingness:

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

Where R is the reward function, H is the hotspot and |H| is hotspot size.

Heap-based Growing Algorithm

```
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
```

Optimization 4: Incremental calculation of interestingness functions

- Algebraic and distributive aggregate functions can be calculated incrementally: Sum, count, average, variance, correlation
- Holistic functions cannot be calculated incrementally. (median, rank etc.)
- Incremental versions of correlation and variance interestingness functions were created
- Calculating the new reward when an object is added to a region only takes O(1) time.

Optimization 5: Parallel processing of hotspot growing phase

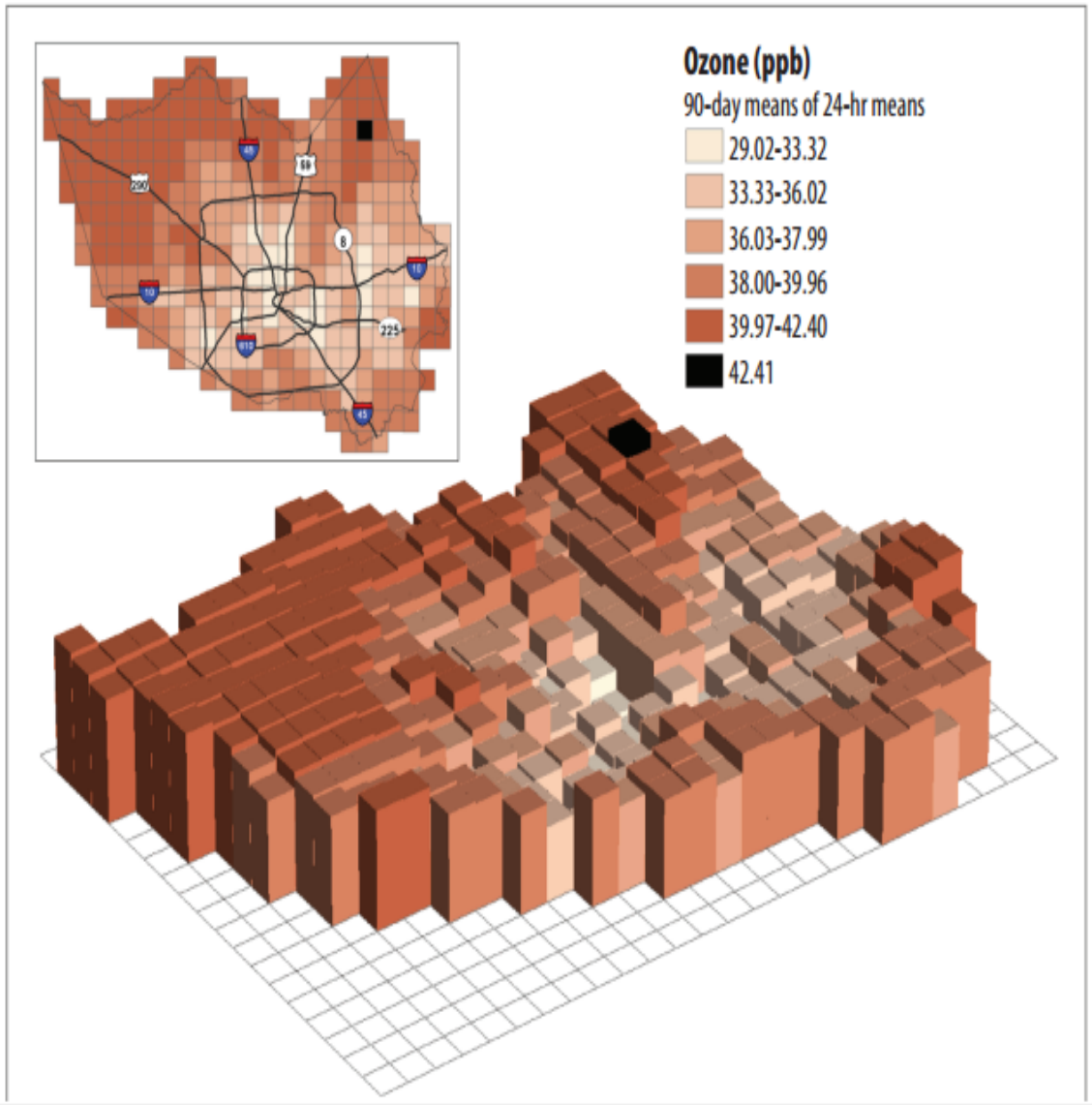
- Process each seed in a thread.
- We use shared memory programming model.
- .NET framework Task Parallel Library used.

Talk Organization

1. Introduction
2. Interestingness Hotspot Discovery Framework
3. Optimizations
4. Experimental Evaluation
5. Summary and Conclusion

Grid-based Air Pollution Dataset

C. Ozone



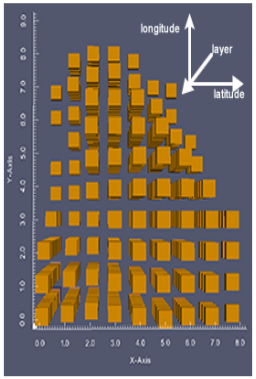
Grid-based Pollution Dataset

Huge amounts of gridded data are generated every day by Environmental Protection Agency (EPA). We just use a subset of the air pollution data covering the Houston Metropolitan area.

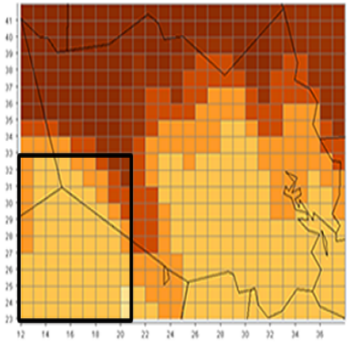
- 4x4km squares in the longitude/latitude space
- This dataset contains 4 dimensions (longitude, latitude, altitude, and time (as measurements are taken every hour)) including 84 columns, 66 rows, 27 layers and 24 hours for each day. This corresponds to 3.6 million grid cells for a day (1.8 GB) and 1.3 billion grid cells for a year;
- Each grid cell contains 132 air pollutant densities as attributes, and these observations are typically extended by adding meteorological and other types of observations for a particular analysis task, such as humidity, temperature, wind speed and solar radiation.
- The city of Houston covers about 15% of the dataset.

Case Study: Low Ozone Variance Regions

- Use a 3D grid for the Houston Metropolitan area for Sept. 1, 2013, noon: 26x19x27=13,338 grid cells
- Reward function: $\varphi(R_i)$ = *interestingness*(R_i) x size$(R_i)^{\beta}$ where β parameter determines preference for larger regions, where i is the variance interestingness function.
  - Seed size was set to 3x3x3 (432 seed candidates)
  - Use β=1.01
  - Variance threshold = $1x10^{-3}$ ppmV
  - Results: 47 hotspots identified
  - Many of the hotspots are overlapping
  - Paraview was used for visualization

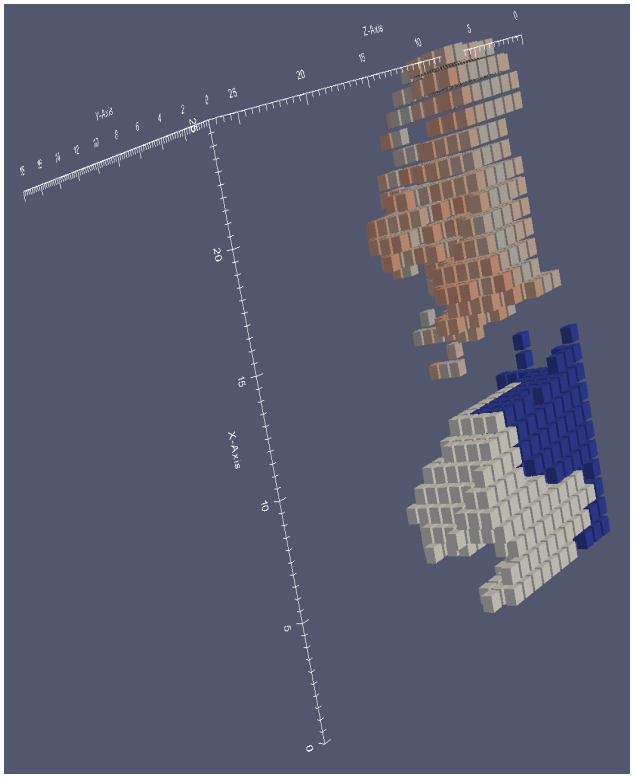


a) Hotspot 1



b) Location of Hotspot 1 in Houston Map

A low variance hotspot and its location in the map



Hotspot 1 (gray), Hotspot 2(blue), Hotspot 8 (orange)

Experimental Evaluation

Test parameters:

Test#	Dataset Date	Timeframe	Seed threshold	Merge Threshold	Seed size
1	2013-09-01	12am (1hr)	0.65	0.96	3x3x3
2	2013-09-01	12am (1hr)	0.65	0.60	3x3x3
3	2013-09-01	6am-6pm (12hr)	0.5	0.96	3x3x3x3

Reward function: $\varphi(R_i) = interestingness(R_i) \times size(R_i)^{\beta}$ where β parameter determines preference for larger regions.

- Used variance interestingness function
- Used β=1.01
- Variance threshold = 0.65 x 10^{-3} ppmV

Experimental Evaluation: Merging seeds

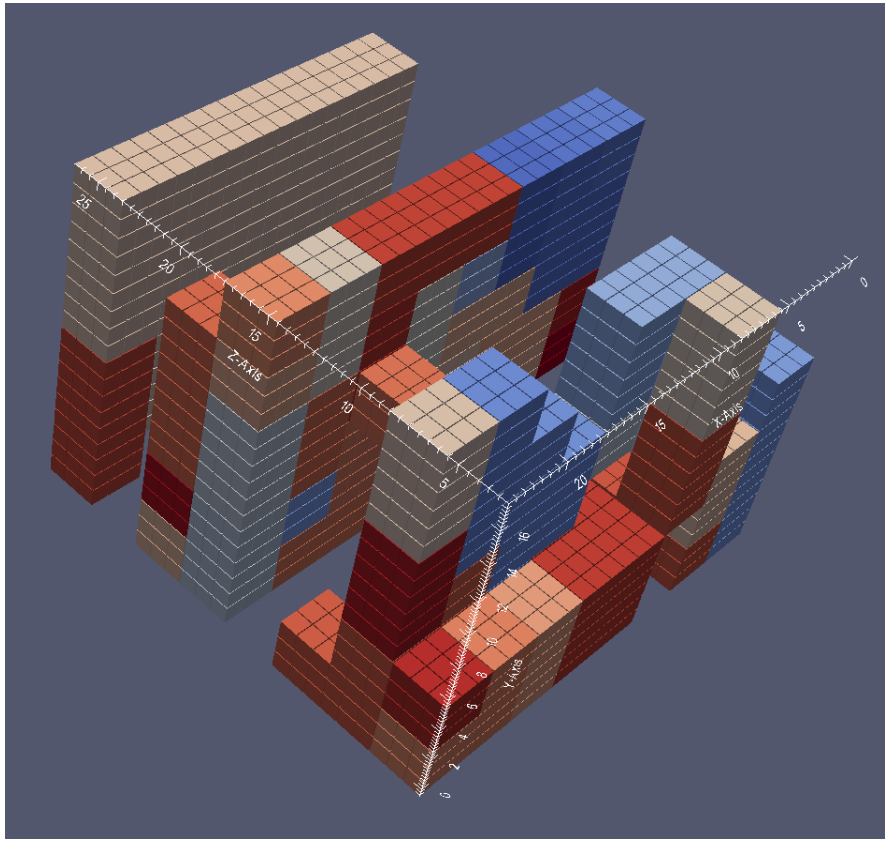
Merging Seed regions:

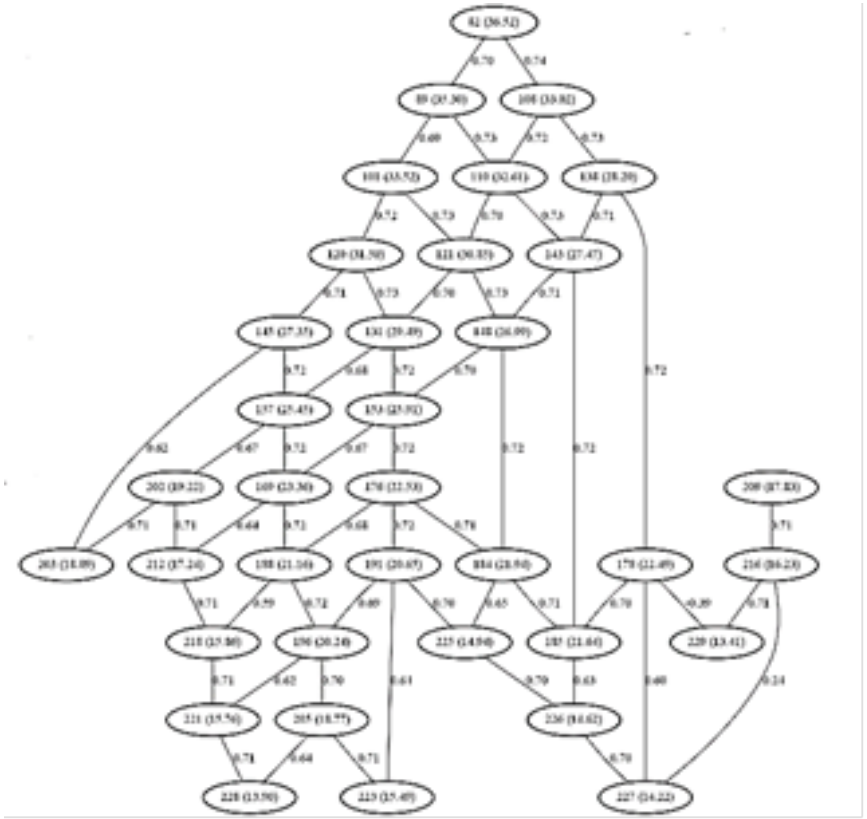
	Test 1	Test 2	Test 3
Seeds Found	27	27	233
Seeds Grown	23	17	79
Hotspots Found after post-processing	8	7	9
Hotspots found without merging	8	8	9
Average hotspot variance	0.275	0.290	0.219
Average hotspot variance without merging	0.269	0.269	0.207
Total reward	1203	1163	63049
Total reward without merging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

- Number for seeds grown decreased, yet hotspot quality almost same.
- 3 times less processing in Test 3. (merging seeds took 0.2 seconds, negligible)

Experimental Evaluation: Merging seeds

Merged seed regions:





Experimental Evaluation: Eliminate contained seeds

Eliminating contained seed regions:

- Containment threshold: 0.9

- Out of 27 seeds in Test 1-2, 10 seeds (37%) eliminated. Total reward decreased from 1206 to 1205.

- Out of 233 seeds in Test 3, 125 seeds (53%) eliminated. Total reward decreased from 63544 to 63049 (1% decrease).

Experimental Evaluation: Heap-based growing

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

- Hotspot size is almost same, processing time is significantly faster (30-40 times in these cases).

Experimental Evaluation: Incremental calculation

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 are orders of magnitude faster for large hotspots.
- Using Heap-based algorithm along with incremental calculation brings a dramatic improvement over legacy algorithm (1000s of times faster growing times).

Experimental Evaluation: Parallel Processing

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

- Used a PC with 4 processors
- Average parallel processing speedup is:

  977 / 354 = 2.75
- Parallel efficiency is 2.75 / 4 = 0.6875

Summary

- 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, compared to other approaches.
- We presented very efficient preprocessing algorithms to eliminate redundant seed regions.
- The proposed heap-based hotspot growing algorithm 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 proposed algorithms are general and can be applied to various kinds of data such as point sets and polygons.

Research Challenges

- Visualization of individual 3D/4D hotspots and visualization of all hotspots
- For some interestingness functions hotspots seem to grow very large
  - Might use a dimensionally growing algorithm for gridded data
- How do we store large grid-based datasets more efficiently?
  - Currently stored in netCDF binary files created for each day