slidenotes last edited on March 2, 2005

1 Introduction

1.1 Intro Slide

- Paul Komarek, from Carnegie Mellon University
- Should anyone have any questions not answered by this talk, or by my paper in the proceedings, my email address is on the slide
- The webpage on the slide has a copy of the talk and my paper
- I'll put this slide up at the end for anyone who didn't get a chance to write this information down

1.2 Pgh Weather Slide

- Data sets getting larger, but we want to use same old machine learning algorithms as before
- Don't want to sample (perhaps we are looking for rare events)
- I will present a fast, flexible, generic data structure that supports counting queries on large data sets, called AD-trees
- Consider this example data set, each row represents one day's weather. There are 3 attributes, each with 2 possible values
- For many machine learning algorithms, we want to make counting queries on this dataset, that is "How many row have T=5F and S=No"

1.3 Datacube representation

- For performance reasons, we don't want to read through the entire data set each time a counting query is made
- We want counting queries to be FREE, or as close as possible
- One way to avoid this is pre-caching the answers to counting queries is some time-efficient manner
- One way to store these precached queries is a Datacube
- The cube has one axis for each attribute, and each block stores one count
- Point to block with Temp=5F, Humidity=100%, Storming=No
- If you want to know how many rows have Temp=5F, Storming=No, you have to marginalize out humidity by summing two existing counts.
- If you are doing a lot of marginalizing, you probably want to store these sums.

1.4 Datacube with Wildcards

- The black cube in the back, right is the cube from the previous slide
- The purple blocks store marginalizations of HUMIDITY. For instance, this purple block is the sum of these two black blocks
- Adding together two of the purple blocks in this direction marginalizes TEMPERATURE. The result is stored in the blue blocks
- Finally, summing the blue blocks marginalizes STORMING, and this result is stored in the white block. This represents the most generic query, namely "How many rows are there in the data set?"

1.5 Full AD-tree

- Another way to organize the counting queries is an AD-tree (All Dimensions)
- White block is root, storing result of most generic query
- If query is "T=5F, H=100%, S=No", you find the answer by starting at the root, ...
- The blocks that store the counts are called AD-nodes
- The ovals that choose a value for an attribute are called Vary nodes
- Even datasets with a moderate number of attributes have too many different queries to make this Full AD-tree practical. Fortunately there is a lot of redundancy in a Full AD-tree

1.6 Andrew's AD-trees

- Moore and Lee, and Anderson and Moore, discovered several types of redundancies
- Most important redundancy comes from complementarity. Suppose I know how many people in world, and how many have brown hair. If I want to know how many people *don't* have brown hair...
- Removing this redundancy alone took a Full AD-tree with 10^{38} AD-nodes, representing counting queries on a medical dataset, down to 10^{5} nodes
- This data structure has provided several orders of magnitude speed up for several popular machine learning algorithms
- Normally it is built statically, with one pass over the data set, effectively by making counting queries in the "right" order to speed things up. The results are stored in the tree, providing answers to a client algorithm in constant time *once the tree has been built*

- Despite the discovered redundancies, however, these *static AD-trees* are often very large, and can take some time to build
- Since most applications don't require every possible counting query, much of this time and space may be is wasted
- Furthermore, if the underlying data set changes (perhaps gaining a new row), this static structure would require possibly exponential time to update all of its AD-nodes
- CAT ANECDOTE: When working with Catapillar corporation, we received a data set with 49 attributes and 3.5 million rows. 10 of the atts had arity ¿ 100, 4 atts had arity ¿ 1000, and one att had arity ¿ 250,000. We tried building a static AD-tree, but aborted after memory usage ¿ 2GB!
- For these reasons, I am introducing a different implementation of AD-trees which is much more agressive about not wasting time or space...

2 Dyanamic AD-tree explanation

2.1 Paul's AD-trees

- As you can see in the slide, it starts with just a root node, representing the most generic query
- As a client makes counting queries, it computes the answer by reading from the dataset.
- For instance, "T=5F, H=100%, S=No", the Dyanamic AD-tree must build enough of the AD-tree to reach the AD-node for this query (show on Full AD-tree slide), thus creating some AD-nodes along the way (circle on Full AD-tree)
- I call this implementation of the AD-tree structure a "Dynamic AD-tree"
- *In practice, we take advantage of complementarity and other redundancies in Dynamic AD-trees*. However, the description of Dynamic AD-trees is easier if we ignore these techniques
- This isn't a great solution on its own, because we'll have to *read through the entire dataset for each counting query made* –this is exactly what AD-trees are trying to avoid!

2.2 How Dynamic AD-trees are grown

- To illustrate the problem, consider the following sequence of queries
- "H=100%": this requires a complete read of the data set
- followed by "H=100%, S=Yes": a second complete read of the data set is made

- If we had kept track of which rows had “H=100%” in the previous query, we could have read through just those rows to answer our more specific query “H=100%, S=Yes”
- Thus we make row caches (draw by hand) accelerate queries which are specializations of queries already made. Not just children, but grandchildren and further able to access row caches higher up (if necessary)
- If we kept a row cache for every AD-node in the tree, even with the elimination of redundancy we’d still require far too much memory
- So row cache usage is tracked in a priority queue. If the size of all row caches grows larger than a preset limit, the row caches at the “least-important” end of the *row cache queue* are destroyed.
- We currently mange the row cache queue with a simple Least Recently Used policy
- A little thought suggests this can be improved. We haven’t done much testing in this area yet
- ROW CACHES NOT ENOUGH
- It turns out that row caches, as described above, aren’t really enough to make Dynamic AD-trees competitive with Static AD-trees
- For instance, suppose our next query was “H=99%, S=Yes”. For illustration, we are still ignoring the effects of complementarity

2.3 How Dynamic AD-trees are Grown (slide 2)

- We need to identify which rows have “H=99%, S=Yes”, which is a subset of those for which “H=99%”
- If we don’t know which rows have “H=99%”, we have to read through the entire data set again
- Now, when we made the 1st of these 3 queries, we identified those rows for which “H=100%”. In effect, we were partioning the data set according to the value of the HUMIDITY attribute
- Even though nobody asked for “H=99%” at the time, it is probably a good idea not to throw this partitioning information away, so we store it. In this case, we keep the list of rows for which “H=99%” in the same place that the “H=99%” AD-node would be, if such a query were made
- Since this isn’t really an AD-node, but looks a lot like one, we call it a skinny AD-node
- For memory reasons skinny AD-nodes are tracked in a priority queue, just as with row caches

2.4 Additional Dynamic AD-tree Benefit

- Dynamic AD-trees make it easy to perform amortized updates on nodes of AD-tree
- This can be used to stay in sync with data sets that grow over time–for instance, if you get new Pgh weather data each week
- The same could be done for Static AD-trees, but it is a hassle.

3 Performance

3.1 Some points up front

3.1.1 Test with `e29.fds`

- For the performance testing of this Dynamic implmentation of AD-trees, we used a relatively small data set
- We chose a small one partially because of scarce cpu time, but also because Static AD-trees get very large, very quickly, and sometimes can't be built in the memory available

3.1.2 Query Pattern slide

- We tested performance using an exhaustive rule learner, decision tree learner, and Bayes net structure finder
- We made comparisons to Static AD-trees, as well as specialized algorithms such as an efficient implementation of the ID3 decision tree algorithm
- Each of the learning algorithm run on Dynamic AD-trees has a different pattern of making queries
- Because the row caches and skinny AD-nodes work like any caching structure, the pattern of accesses (here, queries) affects overall performance
- (explain slide for Rule Learner Pattern)

3.2 Rule Learner

3.2.1 Rule Learner perf, unltd mem, standard rule learner only

- bottom axis is number of attributes allowed in LHS of rule
- vertical axis is time required to find best rule, in seconds
- Note that the traditional (non AD-tree) rule learner was able to find the best rules with one or two attributes in their LHS

- If we asked for the best rules when 3 or more attributes were allowed in the LHS, this traditional specialized implementation required more than 1200 seconds and was aborted

3.2.2 Rule Learner perf, unltd, incl. Stat AD-tree

- (slide with traditional and Static AmD-tree)
- The light blue line with hollow bullets is for "one-off" performance: a Static AD-tree is built seperately for each separate run of the rule learner
- Note that "one-off" runs are denoted "single" in my slides
- The purple line with solid bullets is for amortized performance: Static AD-tree built once and all desired runs of the rule learner are made. This is the intended use of Static AD-trees. Each bullet represents (rule time) + (fraction of Static AD-tree build time)
- Note that amortized runs are denoted as "multi" in my slides
- You can think of the area under the curves as related to the total time for all given rule learner runs

3.2.3 Rule Learner perf, unltd mem, incl Static and Dynamic

- The magenta and red lines represent Dyanmic AD-tree performance when the row caches and skinny AD-nodes are allowed as much memory as they want
- The magenta line show "one-off" performance for dynamic AD-trees
- Since Dynamic AD-trees only build as much of the tree as is necessary, they are faster than Static AD-trees for "one-off" applications
- Given enough memory, Dyanmic AD-trees compete with Static AD-trees even for amortized performance (the forté of Static AD-trees).
- The AD-tree built in the Dynamic implementation on behalf of the rule learner occupied less than 3 MB

3.2.4 Limited Memory Run explanation slide

- explain slide, much is repeat of previous material
- For example, if we make a "32 MB limited memory" run, the row caches are allowed about 19 MB, the skinny AD-nodes about 3 MB, and the remaining 10 MB is reserved for the AD-tree
- Since we know that the tree only required 3 MB (from the previous tests), we are being extremely conservative here

3.2.5 Rule learner perf, limited memory, amortized/multi runs

- All runs in this slide are amortized (i.e. “multi”)
- The purple line with solid triangles is for Static AD-trees, and is the same line we saw on the previous slides
- The red lines with squares represent various limited memory configurations
- The red line with solid, round bullets (at the bottom) is for Dyanmic AD-trees given unlimited memory
- Notice that once enough of the AD-tree is built by the Dynamic implementation, each rule leaner run is very fast

3.2.6 Rule learner perf, limited memory, “one-off”/single runs

- explain the slide, note 1200 second limit
- Note that we’re still faster than our best non-AD-tree algorithm (the short green line in previous slides)
- TRANSITION TO NEXT SLIDE
- Our next slide shows what happens if you data set is growing between successive runs of the rule learner

3.2.7 Rule learner perf, grow

- suppose every week you get more data
- with Static AD-trees you must rebuild your AD-tree each time the data set changes
- with Dynamic AD-trees you have a choice of rebuilding or not rebuilding
- explain slide, Dyanmic runs are unlimited memory

3.3 Decision Tree Learner

3.3.1 Decision tree learner query pattern slide

- explain slide

3.3.2 Decision Tree perf, amortized/multi runs

- horizontal axis is the maximum number of nodes allowed in the decision tree
- vertical axis is time in seconds
- green line is for an efficient implementation of the ID3 decision tree algorithm

- all runs on this slide, except for the ID3 algorithm which doesn't use AD-trees, are for amortized runs
- The purple line is for Static AD-trees
- The red lines with squares are Dynamic AD-tree runs with limited memory
- The red line with filled circular bullets represents a Dynamic AD-tree run given unlimited memory
- Note that AD-trees of both types are clearly superior in amortized settings, for both time and memory use
- Note that, given about the same memory than our efficient ID3 implementation requires, Dyanmic AD-trees are more efficient than Static AD-trees

3.3.3 Decision Tree perf, "one-off"/single runs

- All runs on this slide are "one-off" runs
- The green line, now in the middle of the slide, is the same ID3 line as on the last slide
- The light blue line with hollow triangles is "one-off" performance of Static AD-trees
- The magenta lines are limited memory, "one-off" Dyanmic AD-tree runs
- The magenta line with hollow, round bullets is for unlimited memory, "one-off" Dyanmic AD-tree runs
- Again, note that Dynamic AD-trees are competitive on both speed *and* memory with ID3

3.4 Bayes Net Structure Finder

3.4.1 Bayes Net query pattern slide

- explain slide
- this is bad news for Dyanmic AD-trees!

3.4.2 Bayes Net perf, unlimited memory

- horizontal axis is number of itereations allowed for stochastic hill-climber
- vertical axis is time in seconds
- all Dyanamic AD-tree runs on this slide are given unlimited memory
- Note that Dynamic AD-trees are competitive in both the "one-off" *and* amortized settings!

- This is explained when we see how much memory is being used by the Dynamic AD-trees...
- TRANSITION
- when we limit the memory allowed for row caches and skinny AD-nodes...

3.4.3 Bayes Net perf, limited memory, amortize/multi runs

- when memory is limited, Dynamic AD-trees can't cope with mostly-random query pattern of Bayes net structure finder
- all limited memory Dyanmic AD-tree runs time-out with more than 200 iterations

4 Finalé

4.1 Summary slide

- explain slide

4.2 Future Directions Slide

- explain slide