CME 323: Distributed Algorithms and Optimization, Spring 2015
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 7, 4/25/2016. Scribed by Rolland He.

1 Binary Search Trees

Binary search trees (BST's) help abstract away sets. Everything on the left subtree has value less than everything on the right subtree. This property holds through insertion into and deletion from the tree.

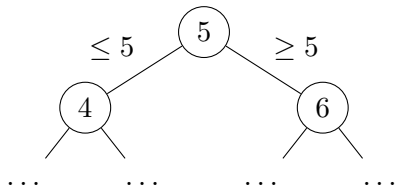

Figure 1: Basic structure of a binary search tree

Balanced BST's

BST's can be balanced – that is, the total height of the tree is $O(\log n)$, with n being the number of nodes. This allows for the height of the tree to be minimal, and therefore leads to operation times (e.g. insertion, deletion). There are a number of data structures that utilize self-balancing in order to keep the tree balanced at all times:

- AVL tree
- Red/Black tree
- AA tree
- etc.

BST Cost Specification

Below are the cost specifications for BST's in Big-O notation:

	Work	Depth
Insert	$\log n$	$\log n$
Delete	$\log n$	$\log n$
Find	$\log n$	$\log n$
Union	$n \log n$	$\log n$
Intersect	$n \log n$	$\log n$

Table 1: Cost specification for binary search trees. n is the number of nodes in the tree.

2 Graph Contractions

2.1 Graph Search Algorithms

We start off with some basic graph theory conventions and ideas. Assume all nodes are labeled with some unique ID number. It is often useful to search or traverse through trees – one example is to find some specific target node. The order in which the nodes are searched determines the type of search. If the nodes are searched in order of increasing depth, then it is a breadth first search. If the search explores down each branch as far as possible before backtracking, then it is a depth first search. If there is some heuristic being used to determine where next to search, it is an A* search.

One perhaps surprising thing to note is that depth first search is not parallel. Consider the following counterexample with threads:

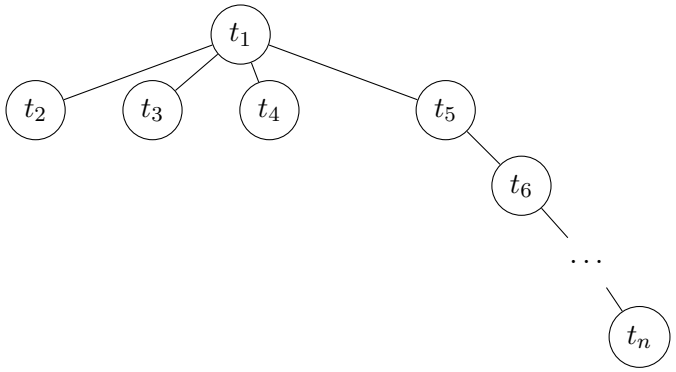

Figure 2: An example of a tree where depth-first search cannot be parallelized

Each thread needs to know where it started, so in order to attempt to parallelize the search, we would create 4 threads for t_2, t_3, t_4, t_5. In the worst case above, it is pretty clear that the depth is still $O(n)$ since the t_5 thread would still have to search through $O(n)$ nodes. Thus, the depth is the same as that of the non-parallelized case, which means that there is no generalized parallel depth first search algorithm.

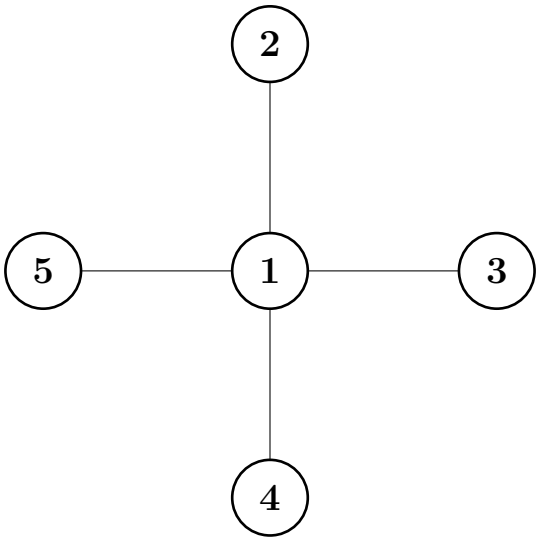

Figure 3: A star graph with 4 satellites, which does not allow for parallelization of our simple random graph contraction attempt above. Note that we can't simultaneously contract node 1 into node 2 and node 1 into node 3, because this will cause race conditions, etc.

2.2 Parallel Graph Contraction Algorithm

For many graph-related problems, it is often useful to contract the graph in order to compute various properties of the graph in parallel.

We begin with a natural attempt to try and parallelize graph contractions. For each edge, we pick a random edge (uniformly) and contract it. Unfortunately, this turns out to not work – if we have a star graph, then we will still have $O(n)$ contractions after parallelization, so parallelization isn't useful here. Therefore, we will need a more clever way to perform parallelize the process.

Consider the following algorithm:

1. Each node flips a fair coin. If the coin comes up head, the node is designated as a "leader"; otherwise, the node is designated as a"follower".

2. Each follower chooses an adjacent leader (if it exists) and that edge is contracted. The follower is then removed and all nodes that the follower is connected to is now connected to the leader in order to maintain connectivity.

3. Repeat the above steps until no edges remain.

In the above algorithm, we don't run into the same problem we had before of trying to contract the same node in parallel because each node contracts into another at most once during each iteration. Therefore, it is clear that this algorithm does indeed parallelize graph contractions.

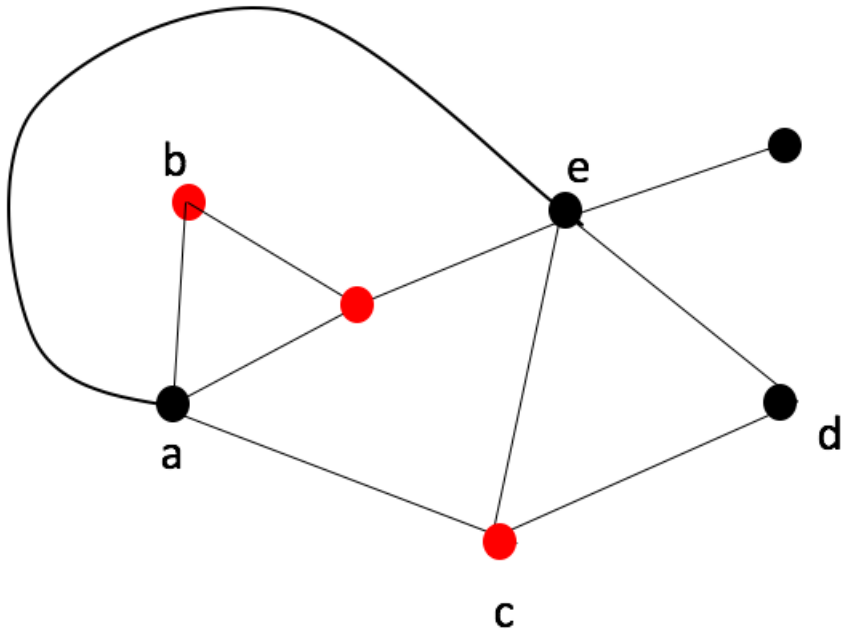

Figure 4: An example round of the parallel algorithm. Each black dot represents a follower node and each red dot represents a leader node. We could, for instance, have the following contractions: a contracts into b, d contracts into c, and e contracts into c. Note that each follower node that is adjacent to a leader node has been removed, and contracted into a randomly selected adjacent leader node. Furthermore, taking a look at the contraction of a into b, we must add e and c to the neighborhood list of b, since those were originally adjacent to a. This is done so that the connectivity properties of the original graph are preserved. The same idea is applied for the other 2 contractions as well.

2.3 Analysis of Algorithm

With this algorithm, we need only start with a neighborhood list (list of adjacent nodes) for each node, which is stored as a BST. Any well ordering for the edges will suffice, as long as it is consistent.

In the first step, where we flip the coins and find the leaders, only $O(n)$ work is necessary, since we spend constant time for each node. Moreover, the depth is $O(1)$ because we can perform the designation of roles for each node individually.

In the next step, we contract the follower nodes into the leader nodes, which involves looking through each edge of the follower node and adding it to the neighborhood list of the leader node. In the worst case scenario, we have to add every single edge ($O(m)$ total edges) to some neighborhood list (which in the worst case has height $O(\log n)$). Therefore, the total work is $O(m \log n)$. We can easily parallelize this work by adding each edge simultaneously because they do not depend on each other. Therefore, we can achieve $O(\log n)$ depth.

Thus, the total work of the algorithm is $O(n + m \log n)$ and the total depth is $O(\log n)$.

With this algorithm, we can also obtain a useful lower bound on the expected number of nodes

removed at each iteration:

$$\mathbf{Pr}(\text{A gets removed}) = \mathbf{Pr}(\text{A is a follower}) \cdot \mathbf{Pr}(\text{One of A's neighbors is a leader}) \geq \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4} \quad (1)$$

Hence, with each iteration, at least a quarter of the nodes are expected to be removed/contracted.

3 Optimization

3.1 Overview

The purpose of optimization is to minimize some objective function F. Formally stated, we wish to solve the following problem:

$$\min_{\mathbf{w}} F(\mathbf{w}), \text{ where } F(\mathbf{w}) = \sum_{i=1}^{n} F_i(\mathbf{w}, x_i, y_i)$$
$$x_i \in \mathbb{R}^d, y_i \in \mathbb{R} \text{ for } i = 1, \ldots, n$$
$$\mathbf{w} \in \mathbb{R}^d$$

Here, the x_i's can be interpreted as the input data, the y_i's as the output data, and $\mathbf{w}$ as the computed optimal weights. F_i is some objective function that is suitable for the problem at hand. A few commonly used examples include:

- Least squares
- Support vector machine (SVM)
- Exponential function (logistic regression)
- Forward/backward algorithm (neural network)

In optimization, there are 2 main dimensions that can be scaled:

1. n – number of training points ("data parallelism")
2. d – model size ("model parallelism")

Optimization has aspects that are suited for distributed computing like regularization and hyper-parameter tuning, but these things are quite straightforward and not particular interesting from an algorithmic or distributed design perspective.

References

[1] T. Cormen, C. Stein, C. Leiserson, R. Rivest. *Introduction to Algorithms.* The MIT Press, 2009.

[2] G. Blelloch, B. Maggs. *Parallel Algorithms.* CMU, 2015.