Heaps and Branch and Bound

COMP 215 Lecture 11

Priority Queues

- A class of data structures that allows efficient access to the item in a set with the largest (or smallest) key.
- Many possible implementations:
  - Sorted list
    - Cost to create a queue with N items?
    - Cost to retrieve the largest item?
    - Cost to enqueue an arbitrary item?
  - Binary search tree
    - Costs?
  - Heap...

Heaps

- In a complete binary tree
  - all internal nodes have two children.
  - all leaves have the same depth, *d*.
- In an essentially complete (or left-complete) binary tree
  - the tree is complete down to level *d*-1.
  - at level *d* all nodes are as far to the left as possible.
- Heaps are binary trees that
  - are left complete.
  - have the heap property – every node has a key that is greater than or equal to they keys of its children.
- Let's look at a heap...

Heaps

- We need to figure out:
  - Given *N* items, how do we create a heap?
  - How do we dequeue and maintain the heap property?
  - How do we enqueue and maintain the heap property?
- A useful routine will be sift-down.
  - Assume we know that every node *except the root* has the heap property.
  - Swap the root with the larger of its two children.
  - Repeat until no more swaps are necessary.
  - Let's try it...

Heap Creation

- Start with a left complete binary tree with keys distributed arbitrarily.
- We can recursively build our heap from the leaves up.
  - All nodes at level *d* (leaves) are valid heaps trivially.
  - All nodes at level *d*-1 are either leaves, and therefore valid heaps, or have two heap children.
  - We can merge two heaps by calling sift-down on their parent.
  - Continue merging until we reach the root.

Heap Creation Cost

- In the worst case, every node that is sifted-down ends up as a leaf.
- So, the worst case number of comparisons is related to the total height of all the nodes in the tree.

- Depth of a binary tree is equal to $\lfloor lg(n) \rfloor$.
- Let's consider the case where *n* is a power of 2.
- Depth is lg(n). There is exactly one node at that depth.
  - We will first sum the heights disregarding that node- dealing with a complete binary tree of depth *d*-1.

Heap Creation Cost

- We take the following sum:

$$\sum_{j=0}^{d-1} 2^j (d-j-1) \quad = \quad 2^d - d - 1$$

# of nodes at level j

height of nodes at level j

- Adjusting for the single node at level d:

$$2^d - d - 1 + d \;=\; 2^d - 1 \;=\; n - 1$$

- It takes two comparisons for each parent-child swap so the total number of comparisons is: $2(n\text{-}1)$.

Removing An Item From the Heap

- Simple:
  - Delete the key at the root.
  - Move the key at the “last” leaf to the root.
  - Call sift-down on the root.
- Worst case $\Theta(lg(n))$ comparisons.

Adding an Item to the Heap

- Also simple:
  - Add a new leaf (with the the new key) to the next available position.
  - Call sift-up.
- Sift-up:
  - Compare current key with parent key.
  - If larger, swap.
  - Continue until no more swaps can be made.
- Worst case $\Theta(lg(n))$ comparisons.
- Average case $\Theta(1)$ comparisons. Why?

Branch and Bound

- The Backtracking 0-1 Knapsack algorithm we saw last week was an example of branch and bound.
  - *bound* – an upper limit on the quality of any solution reachable from the current node.
  - *branch* – beyond that node if the bound is greater than the best solution so far.
- Branch and bound is only appropriate for optimization problems.

DFS BFS Review

- The branch and bound algorithm we saw last week visited nodes according to a depth first search.
  - Easy recursive implementation.
  - Low space requirements. Why?
- We could have used breadth first search.
  - Visit all nodes at level d before visiting any nodes at level d+1.
  - Requires a queue to keep track of the order that nodes need to be visited.
  - $\Theta(n)$ space overhead.

BFS Pseudocode

```
void BFS (tree T){
   queue Q;
   node u, v;

   v = root of T;
   visit v;
   enqueue(v);
   while(!empty(Q)) {
      v = dequeue(Q);
      for (each child u of v) {
         visit u;
         enqueue(Q,u);
      }
   }
}
```

BFS Branch and Bound

```
double BFS_branch_and_bound (tree T){
   queue Q;
   node u, v;
   double best;

   v = root of T;
   enqueue(v);
   best = value(v);
   while(!empty(Q)) {
      v = dequeue(Q);
      for (each child u of v) {
         if (value(u) > best)
            best = value(u);
         if (bound(u) > best)
            enqueue(Q,u);
      }
   }
   return best;
}
```

Why Bother With BFS?

- Maybe you have some reason to believe that good solutions are high in the search tree.
- We've been discussing it because it leads us to Best First Search.
- Simple modification to breadth first search.
  - Instead of storing yet-to-be visited nodes in a queue, store them in a priority queue ordered by bound.
  - Nodes with highest bounds are visited first.
  - The hope is that good answers will be found early on, so many nodes will never be expanded.

Best First Pseudocode

```
double BFS_branch_and_bound (tree T){
    priority_queue PQ;
    node u, v;
    double best;

    v = root of T;
    best = value(v);
    enqueue(v);
    while(!empty(PQ)) {
        v = dequeue(PQ);
        if (bound(v) > best) { //best might have changed
            for (each child u of v) {  //since v enqueued
                if (value(u) > best)
                    best = value(u);
                if (bound(u) > best)
                    enqueue(PQ,u);
            }
        }
    }
    return best;
}
```

Traveling Salesperson

- What would the search tree look like for the traveling salesperson problem?
- Things to consider:
  - Possible solutions exist only at the leaves.
  - In order to develop a branch and bound algorithm, we need a way to compute a bound.
  - In this case the bound should be a *lower* bound instead of an *upper* bound.

Bounding TSP

- A valid tour leaves each vertex exactly once.
- The shortest tour can be no shorter than the sum of the lengths of the shortest edges leaving each vertex.
- I.e. the overall bound is:

$$\sum_{i=1}^{n} \min_{j \neq i} (W[i][j])$$

- Let's look at an example...

Bounding TSP

- After we have added a vertex to our possible tour, the new bound is
  - The length of the tour so far, plus the sum of the minimum length edges leaving the remaining vertices that don't connect to nodes included in the tour so far. (whew.)
- That's it. That's all we need to know to understand the algorithm.
- Will it be able to handle graphs that are not completely connected?
- Note that there are other possible ways to find bounds in this problem.