Decision Trees

- Depth first and breadth first still search the same number of nodes, the order is simply different
- If we are willing to settle for “good enough”, then there is a difference in work done by the two search methods

Searching a decision tree

```
def DFSDTreeGoodEnough(root, valueFcn, constraintFcn, stop):
    stack= [root]
    best = None
    visited = 0
    while len(queue) > 0:
        visited += 1
        if constraintFcn(stack[0].getValue()):
            if best == None:
                best = stack[0]
            elif valueFcn(stack[0].getValue()) > valueFcn(best.getValue()):
                best = stack[0]
            if stop(best.getValue()):
                return best
            temp = stack.pop(0)
            if temp.getRightBranch():
                queue.insert(0, temp.getRightBranch())
            if temp.getLeftBranch():
                queue.insert(0, temp.getLeftBranch())
        else:
            stack.pop(0)
    print 'visited', visited
    return best
```

Searching a decision tree

```
def BFSDTreeGoodEnough(root, valueFcn, constraintFcn, stop):
    queue = [root]
    best = None
    visited = 0
    while len(queue) > 0:
        visited += 1
        if constraintFcn(queue[0].getValue()):
            if best == None:
                best = queue[0]
            elif valueFcn(queue[0].getValue()) > valueFcn(best.getValue()):
                best = queue[0]
            if stop(best.getValue()):
                return best
            temp = queue.pop(0)
            if temp.getLeftBranch():
                queue.append(temp.getLeftBranch())
            if temp.getRightBranch():
                queue.append(temp.getRightBranch())
        else:
            queue.pop(0)
    print 'visited', visited
    return best
```

Testing “good enough”

```
def atLeast15(lst):
    return sumValues(lst) >= 15

print 'DFS’
depth = DFSDTreeGoodEnough(treeTest,
sumValues, WeightsBelow10, atLeast15)

print 'BFS’
breadth = BFSDTreeGoodEnough(treeTest,
sumValues, WeightsBelow10, atLeast15)
```

Searching an implicit tree

- Our approach is inefficient, as it constructs the entire decision tree, and then searches it
- An alternative is only generate the nodes of the tree as needed
- Here is an example for the case of a knapsack problem, the same idea could be captured in other search problems

Implicit search for knapsack

```
def DTImplicit(toConsider, avail):
    # return value of solution, and solution
    if toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0][1] > avail:
        result = DTImplicit(toConsider[1:], avail)
    else:
        nextItem = toConsider[0]
        withVal, withToTake = DTImplicit(toConsider[1:], avail -
nextItem[1])
        withVal += nextItem[0]
        withoutVal, withoutToTake = DTImplicit(toConsider[1:],
avail)
        if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    return result
```