CISC 610-90- O-2018/Late Fall Assignment 5

Youwei Lu

a) CompareKnapsack

To solve the knapsack problem, the backtracking algorithm tries to traverse all possible solutions, except avoid non-promising paths, but branch and bound only looks for the optimal result.

Therefore, the major differences between the two is the bound function. The bound in backtracking must be better than the current best candidate, but the bound in brand and bound algorithm is a kind of priority list, and we try the priority list to find a leaf as soon as possible.

Based on the purposes, another difference is obvious: the searching way. Backtracking usually use depth-first search, and branch and bound however, for it looks for one leaf only, uses breadth-first search.

a.1 Time Complexity

For backtracking, in the worst case there are $2^n - 1$ left children nodes, and the time complexity to traverse them is $O(2^n)$. The bound function needs $O(n)$, and in the worst case, there are $2^n - 1$ right children nodes, so the time used is $O(n2^n)$. Therefore, in the worst case, the time complexity of backtracking algorithm is $O(n2^n)$.

For branch and bound, the bound function is a heap, which needs time $O(n)$, and in the worst case there are $2^{(n+1)} - 2$ nodes, and if unfortunately need to visit all of them, the time complexity of branch and bound algorithm should be $O(n2^n)$.

a.2 Space complexity

Backtracking algorithm only needs to store the solution space tree, which is $O(n)$, and the recursion reaches at most n, so the space complexity of

backtracking is $O(n)$.

For branch and bound, in the worst case, we have to visit and compare all $2^{(n+1)} - 2$ node, and the space complexity is $O(2^n)$.

a.3 Conclusion

Both algorithms have same time complexity, yet branch and bound algorithm consumes more space. Unless we have some info about the optimal result before hand, generally it's better to use backtracking algorithm for knapsack problem.

b) Mergesort

The Mergesort algorithm based on linked list is implemented in this assignment, and its performance is compared with the one completed in assignment 3 based on array.

5 different numbers of random test data are compared, from small set (10) to big set (100000), and the runtime is recorded for each set. As the data is randomized, the runtime is different in each run. One batch test result is shown in Table. 1.

Test Numbers	Runtime (ns) in Assignment 3	Runtime (ns) in Assignment 5
10	8222	13272
100	149420	65433
1000	2850774	783797
10000	96666016	3328808
100000	2771315573	31155629

Table 1: Runtime for random test data.

Though the theoretical time complexity is $O(n \log n)$ for both algorithms, from the table we can see that the merge sort with linked list takes less time. That's because with linked list, it reduces the need of rearranging sorted sequence in contiguous array slots.

c) CompareSearch

- Binary search tree: The major advantages is searching, which can be averagely $O(logn)$, and $O(n)$ in worst case (and balanced binary search tree is used to guarantee the $O(logn)$ time complexity), and disadvantages include insertion and deletion, which is averagely $O(logn)$, and $O(n)$ in worst case (same with searching, but worse than other searching methods). In contrary, linked lit is good at insertion and deletion, but very slow in searching. Examples of situations: 1. database with emphasize on searching, 2. As basic structure for other abstract data types like collection and Associative Array.

- Interpolation search: This search algorithm is very fast. On average the interpolation search's time complexity is about $O(log(log(n)))$ (if elements are uniformly distributed), and in the worst case it can make up to $O(n)$ comparisons. Disadvantage: it only works in sorted array. Examples of situations: 1. Array with uniformly distributed elements, 2. dictionary search.

- B-tree: A commonly good tree that makes sure all searching, deletion, and insertion have the same $O(logn)$ time complexity, even in worst case. Disadvantage: It stores and visits all nodes from root to leaf, which consumes a lot of storage, so usually we don't use it to save the data, but the index of data. Examples of situations: 1. disk file management, 2. database index.

- Selection: the advantage is to find the element we want without sorting, and average time complexity is $O(n)$. Disadvantage: the worst time complexity is $O(n^2)$, worse than some sorting algorithms. Examples of situations: 1. nearest neighbor problem, 2. shortest path problem.

- Hashing: Advantage: super fast. All operations, searching, insertion, and deletion, only takes only $O(1)$ time. Disadvantage: very hard to sort and expand. Examples of situations: 1. encrypting, 2. spelling check.

d) CompareDesign

d.1 Backtracking versus Branch and Bound

The most reliable method to solve a problem is listing all candidate solutions. Theoretically, we can find the solution(s) by checking all candidates.

Both backtracking algorithm and branch and bound algorithm traverse the solution state space, and using conditions and bound functions to remove unpromising candidates (or a group of candidates) that don't need to be visited.

The major differences between the two is the bound function. The bound in backtracking must be better than the current best candidate, but the bound in brand and bound algorithm is a kind of priority heap, and we try the priority list to find a leaf as soon as possible.

Their purposes are different, as the backtracking is used to find all targets, but branch and bound only looks for the optimal result. Based on the purposes, another difference is obvious: the searching way. Backtracking usually use depth-first search, and branch and bound however, for it looks for one leaf only, uses breadth-first search.

a comparison of backtracking algorithm and branch and bound algorithm is shown in Table. 2.

Algorithms	Search Approach	Data Structure	Nodes characters	Applications
Backtracking	Depth-First Search	Heap	Visit all promising nodes	Find all targets
Branch and Bound	Breadth-First Search	Queue	Node only visited at most once	Find optimal solution

Table 2: Compare Backtracking and Branch and Bound

For time complexity, the backtracking algorithm's time consuming is proportional to the solution space, and the branch and bound algorithm's time efficiency depends on the location of optimal result. If the optimal result is closer to the root, it may have a better time complexity than backtracking.

For space complexity, the backtracking algorithm depends on the depth of solution space, while branch and bound algorithm depends on the breadth

of solution space. Usually the depth is far less than the breadth ($h << 2^h$), so backtracking algorithm uses less space.

Permutations is a typical example that is best solved by backtracking, but could not be solved by branch and bound algorithm, as we have to find out all possible permutations of inputs, not just a 'best' one.

0-1 knapsack and shortest path can be solved by branch and bound, as usually with less time complexity than backtracking algorithm.

d.2 Relation and compare to D&C, DP, and Greedy

All of these algorithms share a lot of concepts and methods with the two studied above. It is hard (and no need) to treat them independently.

All the algorithms divide the problem into smaller problems (almost all can be done by recursion), and backtracking can be used as the first step in finding dynamic programming or greedy approach. Greedy is also used in building the bound functions in backtracking and branch and bound algorithms.

Backtracking and branch and bound algorithms may not need to visit all possible paths as long as they are non-promising, but divide and conquer divides the problems into smallest parts without checking its validity before 'visit'.

Both backtracking and branch and bound algorithms can be improved by dynamic programming if the branches of the tree overlap, which means previous computation is being repeated.

Backtracking and branch and bound algorithms try to prune the solution tree and avoid non-promising paths, while greedy algorithm guesses the 'best' path.