1 Branch and Bound - Making Backtracking Practical

In optimization problems, narrowing the search space is essential in finding the optimal solution efficiently. We have seen examples such as (1) only generating legal partial solutions, (2) imposing ordering on choices so that no redundant paths through the search tree are considered. In addition, there are cases where we can determine if no solutions in a subtree are good enough to be optimal. The idea is to create a function that returns the best possible cost of any solution derived from the current partial solution if this cost is not better that the best known solution, the branch doesn't contain an optimal solution and can be pruned. Of course, it is generally not possible to determine the best possible cost of a solution derived from a given partial solution without actually searching the subtree (o.w. we could just pass the root of the search tree and get back our answer). Instead, we have a function return an estimate of the best possible cost of any solution derived from the current partial solution. If the estimated cost is not better that the best known solution, the branch doesn't contain an optimal solution and can be pruned. We call this type of function a *bound* function, and in order to ensure that an optimal solution is actually discovered, the bound function must be

1. *Safe* (optimistic about quality of solutions)
    (a) If our objective is to maximize, then the bound function must return a value for a partial solution that is at least as large as the actual value of any partial solution in its subtree. If we are too optimistic, we waste time (don't prune early enough). If we are pessimistic, we may miss solutions.
2. *Based only on the partial solution / subproblem*
    (a) Cannot consider how future choices play out - we are trying to avoid searching the subtree
3. *Relatively Efficient to compute*
    (a) Bound function will be evaluated for every partial solutoin/ subproblem generated. Should be fast enough to actually save time.

The practicality of the bound function depends heavily on

1. *The quality of the bound*
    (a) tighter bound = earlier pruning
2. *The initial solution value*
    (a) You need to arrive at a solution quickly in order for your bound to allow you to prune
    (b) The quality of your solution will also control how early branches are pruned - the quicker you find a good solution, the more likely you are to satisfy Bound(current solution) $\leq$Value(Best Solution so Far).
3. *Time to compute bound*
    (a) better quality estimates are generally more expensive
    (b) expensive bound may not save enough work via pruning to be worthwhile

Implementation Patterns

Recall the Implementation pattern for finding an optimal solution -

Algorithm 1 Find optimal solution

```
1. solution solve(partial solution : $P$, subproblem : $S$, best solution so far $P^*$):
2. If the partial solution $P$ is complete:
    (a) return the better of $P$ and $P^*$
3. Else:
    (a) for each legal *next-choice*:
        i. generate a partial solution $P^{'}$ and subproblem $S^{'}$ with that choice made
        ii. result = solve(new partial solution : $P^{'}$, new subproblem $S^{'}$)
        iii. if result is a solution, and it is better than $P^*$, the best solution so far:
            A. update $P^*$ to result
    (b) return best solution so far, $P^*$
```

To convert this into a branch and bound style template, we just add checks for bound/prune opportunities.

Algorithm 2 Find optimal solution (for maximization)

```
1. BBsolve(partial solution : $P$, subproblem : $S$, best solution so far $P^*$):
2. If FinishedQ($P$, $S$) : #partial solution complete? Partial solution invalid? No more choices to be made?
    (a) return the better of $P$ and $P^*$ #Compare Value(P) to Value(P*)
3. If Bound ($P$) $\leq$Value ($P^*$):
    (a) Return $P^*$ #prune this branch of the search tree - we can not find a solution better than $P^*$ by traversing this branch
4. Else:
    (a) #Optionally : order the next choices and remove any that are no longer valid
    (b) for each legal *next-choice*:
        i. generate a partial solution $P^{'}$ and subproblem $S^{'}$ with that choice made
        ii. result = solve(new partial solution : $P^{'}$, new subproblem $S^{'}$)
        iii. if result is a solution, and it is better than $P^*$ :
            A. update $P^*$ to result
    (c) return best solution so far, $P^*$
```

Pruning

Clever pruning tactics are essential for there to be any hope in finding a solution to certain problems. For example, in the 8-Queens problem, the goal is to find all possible placements of 8 queens on an 8×8 chessboard such that for each pair (x, y) of queens, x cannot capture y. There are 178,462,987,637,760 possible configurations of 8 queens on an 8×8 chessboard, but by using pruning tactics, we can greatly reduce the amount of configurations we need to consider. Consider this problem from a more of the output approach - our next choice is the position of the next queen.

1. *Impose Ordering on choices*

    (a) There are only 16,777,216 placements with one queen per row

2. *Prune if no legal solution is reachable*

   (a) Only 15,720 placements when conflicts with previously placed queens are considered

3. *Narrow the search space*

   (a) Exploit symmetry - e.g. only place the first queen in rows 1-4, cutting the search space in half. To recover all solutions, consider "flipping" each generated solution about the x-axis. Of course, more clever techniques can also be applied.

Branch and Bound Considerations

- Optimization objective and subproblem value
  - What is the value you are optimizing over?
- Define a Partial Solution
- Define your subproblem
- Next Choice (Extending a subproblem - how to make subproblems bigger)
  - This defines the order of the search space, and is a very important consideration when designing a branch and bound algorithm.
  - More of the input : take / not take next item (branching factor $= 2$, longest path $= n$)
  - More of the output : choose next item to take (branching factor $= n - k$ (k items already considered), longest path $\leq n$)
  - For knapsack - Do you add an item to the current knapsack arbitrarily, or do you choose a new item based on all remaining items weights and values? More generally - do you impose an ordering on your choices to limit redundant paths, or to explore branches that are more likely to lead to good solutions first? It's fine if you don't, but you must acknowledge these considerations.
- How are you bounding the the min/max objective value of a subproblem?
  - For example - in knapsack, a possible upper bound on value can be given by Value(Knapsack Items) + Value(unconsidered items). This is clearly an upper bound on the max value of your current knapsack, though better bounds can be obtained.
- When do you prune a branch of the search tree?
  - The more of your search space that you are able to prune, the less time your algorithm spends considering sub-optimal solutions to your problem.
  - If there is no complete solution reachable from a given partial solution, the branch may be pruned - perhaps you should not consider choices that would lead to an invalid partial solution.
  - For this part, you just need to explain when you stop trying to improve on a partial solution (e.g. in knapsack, if your current knapsack is over - capacity, it's a good time to prune).

Examples

$0 - 1$ **Knapsack**

- Partial Solution
  - A set of items constituting our knapsack P. A set of remaining items S
- Subproblem $\mathsf{Knapsack}\,(P, S, v^*, P^*)$
  - P : set of items in our current backpack

  - S: Set of items not in our knapsack
  - v^*: Best value for any knapsack seen so far
  - P^*: items in best knapsack seen so far
  - Choose item in S to add to P and find best solution with/without it

- Next Choices:
  - Any item in S that is not already in P

- Bound/ Prune
  - Eliminate choices at each step by removing items whose weights are larger than the remaining capacity of the knapsack (be sure to add them back when you backtrack!)
  - **Before we solve, we update v^* and P^* if applicable. **
  - Bound : $v^* >$ Value(current knapsack) + (max (value/weight) of any item)*(remaining capacity)
  - Bound : $v^* >$Value(current knapsack) + (max (value/weight) of item *that can still be added to current knapsack*)*(remaining capacity)
  - If bound holds, then do not continue to recurse, just return P^*
  - You can try to come up with better bounds for the homework.

This is not part of the template, but may be useful to think about :

- Solving Subproblem:
  - $P^{'} = P \cup \{i\}$where i is the next item chosen
  - $S^{'} = S \setminus \{i\}$
  - OR $S^{'} = S \setminus \{i\} \setminus \{j \ : \ Weight\,(P) + w_j \leq W\}$#Narrow search space more by removing items we can't possibly add in future
  - To solve, we just take better of :
    - Knapsack$\left(S^{'}, P^{'}, ...\right)$ #find best solution with i chosen
    - Knapsack$(S^{'}, P, ...)$ #find all solutions without i chosen

- Best branch first:
  - Which of the next choices should we prioritize?
    - Add the highest valued item in S next
    - Add the item in S that maximizes (value/weight) next
    - Add item at random

Graph Coloring - Given $G = (V, E)$, and integer k, find function $c \ : \ V(G) \to [k]$ so that $c(u) \neq c(v)$ for all $(u, v) \in E\,(G)$

- Partial Solution
  - a function c mapping $S \subseteq V(G)$ into $[k]$.

- Subproblem : k-Color(G, k, S, c)
  - $S = \{v \in V(G) \ : \ c(v) \text{ is defined}\}$
  - A partial mapping $c : S \to [k]$
  - For each possible next (legal) color for the next vertex, choose that color and try to color the rest of the graph.

- Next Choices:

    - Each vertex $v \in V(G) \backslash S$

- What's the next vertex?
    - whatever happens to come up next as we iterate through the vertices?
    - one adjacent to the last-colored vertex?
    - one adjacent to any colored vertex ?
    - the one with the fewest color choices available ?
    - the one with the highest degree ?
- Bound Function / Pruning:
    - Is there a vertex with neighbors assigned all k colors?
    - Is there a monochromatic edge? Did we allow ourselves to create a monochromatic edge when making a next choice?

Max Independent Set– largest subset of vertices such that no two in the set are connected by an edge.

- Partial Solution
    - A set of vertices
- Subproblem - MaxIndSet (G, P, S, P^*, v^*)
    - get next choice v, find largest IS with / without this choice
    - $P = P \cup \{v\}$ #current independent set
    - $S = S \backslash \{v\} \backslash \{u \ : \ (u, v) \in E(G)\}$, where v is chosen vertex to add (i.e. eliminate all of $v's$ neighbors) #edges in subgraph we can choose next vtx from
- Next choices:
    - Each of the vertices in S (note : partial solution always legal given our S update)
- Search best branch first:
    - Add vertex in S with lowest degree
    - Add vertex v in S maximizing $|S \backslash \{v\} \backslash \{u \ : \ (u, v) \in E(G)\}|$(favor adding vertex that leaves us with most remaining choices)
- Bound Function:
    - number of vertices picked so far + a maximal independent set of S?
        * Not Safe! This is an under-estimate
    - $|P| + |S|$
        * Safe, but probably a bit too optimistic
    - $|P| + |S| - \min(\deg_{v \in S} v)$
        * Bound to pick some vertex in S, and we can't add any neighbor, so this will give a decent estimate.
    - $|P| + (1 + \sqrt{1 - 8m - 4n + 4n^2})/2$
        * Turan's upper bound for max independent set