Advanced algorithms
Branch & Bound + Metaheuristics Notes

Søren Dahlgaard

June 15, 2011

1 Disposition

1. Motivation - NP-hard problem.
2. Functions f and g
3. Sets P and S.
4. General procedure
   (a) Keep a set of unexplored subproblems. Each with a bound $g(P_i)$.
   (b) View this as a tree and explore subproblems by branching on them
   (c) Requirements of g: $g(P_i) \leq f(P_i)$ and parent-child.
   (d) Lazy vs Eager
5. Bounding function
   (a) $\min_{x \in P} f(x)$ - Relax some requirements.
   (b) $\min_{x \in S} g(x)$ - Modify objective function
   (c) Combine these. We can parameterize. Langrangean relaxation.
6. Selection strategy - BeFS, DFS, IDDFS, etc.
7. Branching strategy - TSP example
   (a) Create permutation picking next vertex each time
   (b) Create subset deciding if e_i is in or not.
8. TSP 1-tree example
9. Time analysis of worst case
   (a) k-SAT.
   (b) Branching vector
   (c) Branching factor. $\tau(1, 2, \ldots, q)$.
10. *Initial incumbent - metaheuristics*
    (a) gradient search all of them.
    (b) Simulated annealing
    (c) Hill climbing
    (d) Tabu search

2 B&B - 1

2.1 Motivation

We have already seen that some problems may very likely be intractable to try and solve completely (NP-Hard problems). We have also seen that we can some times approximate these problems very well, however sometimes this is not possible (TSP) or the approximation is simply not good enough.

One of the most widespread ways to find optimal solutions is using branch and bound.

2.2 Procedure

Let S be the set of feasible solutions to a problem A and let f be a function that associates a solution with its objective value.

Example If A is TSP, then S would be all permutations of V and f would be a function summing the weights of the edges.

We also define P as a set of all potential solutions and a g which is a bounding function.

Example If A is TSP then P could be the family of all sets of subtours in G. The bounding function g will be described later.

The idea is to keep a set of unexplored solutions and then at each step of the algorithm we split the set in subsets (branch) and compute a bound on the best solution in each of these sets (bound). We also keep a "current best" or incumbent, so if one of the subsets S_i has a bound worse than the incumbent, then we discard S_i.

In order to do this we view the problem as a tree with each node representing a set of potentional solutions P_i. Each P_i thus represents a subproblem of the original problem by imposing extra constraints (eg. "edge e_i must be in the tour" for TSP). If P_j is the child of P_i then P_i is a subproblem derived from P_i.

We call all nodse P_i in the tree for live. If some node has a worse bound than the incumbent we kill/discard/fathom it. The same goes for nodes that cannot lead to feasible solutions.

In order to do this we normally require some things of the bounding function g:

- $g(P_i) \leq g(P_i)$ for all nodes in the tree.
- $g(P_i) = f(P_i)$ for all leaves in the tree.
- $g(P_i) \leq g(P_j)$ if P_j is a child of P_i.

2.3 Lazy vs Eager

There are two ways to build the tree. The eager way we create the bound each time we create a node (when branching). In the lazy way we wait until we are picking the node to try and branch on, so we pick each node based on the bound of its parent, then evaluate the node's bound and see if it's better than the incumbent. If it is, we branch.

If we use BFS or DFS it can be a good idea to use the lazy method, because the order in which we visit the nodes isn't determined by their bounds. If we use a BeFS it can still be done with lazy, but it doesn't make much sense.

2.4 The three elements of B&B

As can be seen we use three elements in this approach:

Bounding function: The bounding function is the most important part of the B&B algorithm. If we use a weak bounding algorithm we will end up traversing huge parts of the search tree (there will be many critical nodes). If we use a strong bounding function however we will only visit very few of the nodes in the search tree.
We often have a tradeoff of precision versus time when picking a bounding function. For instance a bounding function satisfying $g(x) = \min f(x)$ over its children will be NP-Hard itself.
There is generally three ways of making lower bounds:

- $\min_{x \in P} f(x)$. This means that we relax some of the requirements on the problem and calculate the optimal solution. For instance we can get a lower bound for the 0/1-Knapsack by calculating the value of the fractional knapsack.
- $\min_{x \in S} g(x)$. This way we calculate the bound for the problem itself, but we modify the objective function. If we do this we will not likely have $f(x) = g(x)$ for leaves. An example of this is to simply use the constrains applied to the P_i we're calculating the bound for. For instance using TSP with a problem P_i defining the extra constraints that some edges MUST be in the tour, then we simply sum the weight of these edges.
- We might also combine the two to get $\min_{x \in P} g(x)$. This might seem weaker, but we can parameterize it to find a good parameter. An example of this is described below.

Selection strategy: This defines the order in which we pick the nodes in our search tree.

BFS Simply go through the search tree one level at a time. This is however not very smart because it requires a lot of memory and potentially a lot of time. Only use this if we know that the optimal solution is very close to the root (in the tree). *Queue*

BeFS Pick the live node with the lowest bound first. This way we only visit critical nodes - nodes $x \in S$ with $g(x) \leq f(x')$ for optimal $x' \in S$. This can however use just as much memory as a BFS in the worst case and in general uses a lot of memory. *Priority queue*

DFS Simply pick a live node of max depth in the tree. This ensures that at most b nodes per level will be live, where b is the maximal branch factor. So the memory use is $O(bd)$ where d is the depth of the search tree. DFS may however visit a lot of non-critical nodes.

IDDFS Iterative deepening DFS. Combine DFS with a depth limit. This way we explore all levels at a time like in BFS, but we also get the space complexity of DFS. The running time turns out to be the same. In most cases the earlier searches (ie. smaller depths) give good bounds for use in the later searches.

We might also combine DFS with an ordering of the nodes of a level, so instead of just picking any node of maximum depth, we pick the "best" node of maximum depth.

Branching rule:

We also want a good initial incumbent.

2.5 Good TSP bound

Use a #1-tree:

1. Pick a vertex v and remove it from the graph, then create a MST of the remaining graph.
2. Add the two smallest-weight edges incident v to the graph
3. The result is a graph with $|V|$ edges that might not be a tour.

This is obviously a lower bound on the optimal TSP tour. If the returned graph is a tour then it must be optimal.

We now have two choices:

1. Improve the bound we found
2. Branch

The idea is that some vertices will have more than 2 incident edges and thus some will have less than 2. If we choose to branch on a vertex like that we make degree(v) ned nodes for each of those nodes with > 2 incident edges. For each of these nodes we remove an associated edge from the graph. This will however cause some subproblems P_i to overlap.

If we want to imporve the bound, we define a number $\pi(v)$ for each vertex $v \in V$ to be $\pi(v) = \text{degree}(v) - 2$ (for the degree of v in T). For each edge (u, v) we now define a modified cost:

$$c'(u, v) = c(u, v) + \pi(u) + \pi(v)$$

Note that we have

$$\sum_{v \in V} \pi(v) = \sum_{v \in V} \deg(v) - 2|V| = 0$$

Because we have exactly n edges. Any hamilton tour will thus have cost:

$$\sum_{e \in H} c(u, v) + \pi(u) + \pi(v) = \sum_{e \in H} c(u, v) + \sum_{e \in H} \pi(u) + \pi(v) \tag{1}$$

$$= \sum_{e \in H} c(u, v) + \sum_{v \in V} 2\pi(v) \tag{2}$$

$$= \sum_{e \in H} c(u, v) \tag{3}$$

Where $e \in H = (u, v)$ is some edge in the hamilton cycle. Thus we have that all hamilton tours under this new constraint have the same value, though 1-trees will likely cost more - giving a better bound.

2.6 Branching rule

A branching rule is how we branch. For instance the above strategy of creating $\deg(v)$ new nodes for a node v is a branching strategy.

Convergence is ensured if subproblems P_j of P_i are smaller and there's a finite branching factor b.

Other branching strategies could be:

- For TSP start with vertex 1. Each time we branch create a branch for all vertices not yet in the given tour. so first we create the paths: $(1,2),(1,3),(1,4),\ldots,(1,n)$ and then for subproblem $(1,4)$ we will get the problems $(1,4,2),(1,4,3),(1,4,5),\ldots,(1,4,n)$.
  In this case we won't get identical subproblems.
- For TSP look at the edges of E in some order and for each branching make a node whether to include edge e_i or not. This gives us a binary tree but with much bigger depth than the previous one ($|E|$ versus $|V|$).

Neither of these two strategies give overlapping subproblems though the 1-tree branching does. This is not incorrect, but just less efficient.

2.7 Initial incumbent

If we can produce a good initial incumbent we can limit the amount of nodes searched with for instance DFS or BFS. This is often done with metaheuristics.

3 Branch & Bound - 2

For $L \in NP$ we can state the problem as:

$$\text{Given } x \text{ find } y \text{ such that } |y| \leq |x|^c \wedge A(x, y)$$

(See NP notes). Thus if we enumerate all y with $|y| \leq |x|^c$ we can find the solution.

Subset problems For each element x we can have either $x \in S'$ or $x \notin S'$. This gives $2^{|S|}$ possibilities.

Permutation problems Given a set of size n there is $n!$ permutations.

Partition problems We need to partition a set of n elements into different groups from size 1 to n. There is n^n possibilities.

3.1 Maximum Independent Set

We want to pick a set $I \subseteq V$ such that for all vertices $u, v \in V$ we have $(u, v) \notin E$. The naive version would thus take $O(2^n)$ time.

The idea is that for each vertex v we must have either $v \in I$ or one of its neighbours $u \in N[v]$ must be in I (otherwise we could add v). Thus we pick a vertex v of minimal degree, then we make a new node in the search tree for each node u in its neighbourhood and assume that this node has to be in the set for that given subproblem. Because one of those vertices MUST be in I it is clear that the algorithm produces the correct answer. In general correctness proofs for branching algorithm follow easily from the fact that the algorithm examines all possibilities.

Running time analysis: The running time of a B&B algorithm is a function on the maximal size of the search tree. The algorithm picks a vertex of minimum degree and recurses on all vertices in its neighbourhood. Thus we get the following recursion formula:

$$T(n) \leq 1 + T(n - d(v) - 1) + \sum_{i=1}^{d(v)} = T(n - d(v_i) - 1)$$

That is: 1 node for the current. One subproblem for v where we have removed $d(v)+1$ nodes (all nodes in $N[v]$) and one subproblem for each $v_i \in N[v]$ where we remove all nodes in $N[v_i]$. Because $d(v_i) \geq d(v)$ we can write:

$$T(n) \leq 1 + (d(v) + 1)T(n - d(v) - 1)$$

Let $s = d(v) + 1$ We then have

$$T(n) \leq 1 + sT(n - s) \leq 1 + s + s^2 + \ldots + s^{n/s}$$

Because the recursion $T(n - s)$ goes down n/s levels. So we have:

$$\begin{aligned} T(n) &= 1 + sT(n - s) && (4)\\ &= 1 + s(1 + sT(n - 2s)) && (5)\\ &= 1 + s + s^2T(n - 2s) && (6)\\ &= 1 + s + s^2(1 + sT(n - 3s)) && (7)\\ &= 1 + s + s^2 + s^3(n - 4s) && (8)\\ &= \ldots && (9) \end{aligned}$$

This is a geometric series with the value

$$\frac{1 - s^{n/s+1}}{1 - s} = O(s^{n/s}) = O(3^{n/3})$$

Note. 3 is the integer value that maximizes the formula, otherwise e would be better. Also note that we ignore the polynomial factors associated with each node.

3.2 General time analysis

A general way of analyzing branching algorithms is an open problem. In general we want to find the smallest α such that the running time is $O(\alpha^n)$.

To find this we use a branching vector: Let $r \geq 2$ be the amount of branches and let $t_i, i = 1..r$ be integers $n \geq t_i > 0$ such that branch number i is a subproblem of size $n - t_i$. Then we call $b = (t_1, \ldots, t_r)$ the branching vector. Because $r \geq 2$ we can do with just looking at the amount of leaves in the search tree because they make up at least half the tree. So we get:

$$T(n) \leq T(n - t_1) + T(n - t_2) + \ldots + T(n - t_r)$$

A solution to this is of the form c^n with c being the complex root of:

$$x^n - x^{n-t_1} - \ldots - x^{n-t_r} = 0$$

We call this c for α or the branching factor. We denote this $\tau(t_1, \ldots, t_r)$ for a branching vector. For instance $\tau(2, 2) = \sqrt(2)$. With our restrictions on the branching vector we have:

1. $\tau(t_1, \ldots, t_r) > 1$
2. $\tau(t_1, \ldots, t_r) = \tau(t_{\pi(1)}, \ldots, t_{\pi(r)})$ for some permutation π of $1..r$.
3. $\tau(t_1, \ldots, t_r) < \tau(t'_1, \ldots, t_r)$ if $t_1 > t'_1$.

We also have

1. $\tau(k, k) \leq \tau(i, j)$ for $i + j = 2k$.
2. $\tau(i, j) > \tau(i + \epsilon, j - \epsilon)$ for $i < j$ and $0 < \epsilon < (j - i)/2$.

A problem may have several branching vectors, so we have to find the "worst" one when calculating the upper bound. We can also add branching vectors, so if we branch (i, j) and immediately branch (k, l) on the i-branch. Then we get the branching vector $(i + k, i + l, j)$ - Remember that we ignore the polynomial cost of the i-node.

3.3 k-SAT

If there is $|L|$ variables we can have at most $2|L|$ literals (both the variable and its negation). Each clause has at most k literals, so in total there can be at most

$$m \leq \sum_{i=1}^{k} \binom{2n}{i}$$

unique clauses.

We use the following branching rule. Let F be the formula we are checking. Let c be some clause in F. Let $(l_1, \ldots, l_q)$ be the $q \leq k$ literals in c. Create the following q branches:

1. $l_1 = \textsc{True}$
2. $l_1 = \textsc{False}, l_2 = \textsc{True}$

3. ...

4. $l_1 = l_2 = \ldots = l_{q-1} = \text{FALSE}, l_q = \text{TRUE}$

Let $T(n)$ be the running time for n literals, then we get the branching vector $(1, 2, \ldots, q)$. It turns out that $\tau(1, 2, \ldots, q)$ is the largest real root of:

$$x^{q+1} - 2x^q - 1 = 0$$

For each F we thus have a branching rule for each clause. The clause with fewest literals will give the best branching.

We will now show that we can do this in a way that always branches on a clause with $q \leq k - 1$ except for the first time. To this we use something called "autarks". An autark is an assignment of truthvalues t such that for any clause $c \in F$ if t sets one literal of c then it MUST set one of the literals to true. That is. All clauses that t reduces, it satisfies. If F' is the subproblem remaining when setting F to t, then we have that whether F' is satisfiable or not is independant of t, so we have F' is satisfiable iff F is satisfiable. Thus if we have added enough constraints to create an autark we don't need to branch, but rather recurse (this is a reduction rule rather than a branching rule). Otherwise we know that there must be a clause for which our t has set a literal to false, but none to true, so it must have $q \leq k-1$ literals. Thus for 3-CNF we need to solve:

$$x^3 - 2x^2 - 1 = 0$$

which gives a running time less than $O(1.6181^n)$.

3.4 Maximum independant set

If minimum degree of $v \in G$ is ≥ 3 we pick a vertex of minimum degree. Otherwise we pick a vertex of maximum degree. We then either branch or reduce.

Domination rule For any $v, w \in V$ if $N[v] \subseteq N[w]$ we can make a MIS without w. This is obvious. Consider $w \in I$ then no neighbours of v can be in I, thus $I \setminus \{w\} \cup \{v\}$ is an independant set of the same size.

2.6 If no MIS contains v then ALL MIS contain at least two vertices from $N[v]$. Assume for contradiction that an MIS doesn't contain anything from $N[v]$, then we could add v. If it only contains one element from $N[v]$ we could remove that element and add v. Contradiction

2.7 Let $M(v)$ be all vertices at distance 2 from v such that $N[v] \setminus N[w]$ is a clique (this is called mirror vertices). Then a MIS either contains v or none of its mirrors.
Proof: If $w \in M(v)$ we have $N[v] \setminus N[w]$ is a clique and thus only one element from it can be in a MIS. Thus there must be one element from $N[w]$ in the MIS.

Simplical If $N[v]$ is a clique then we have $\alpha(G) = 1 + \alpha(G \setminus N[v])$, where $\alpha(G)$ is the size of a MIS for G.

Disconnected graphs Let G be a disconnected graph, then we can calculate the MIS for each of the connected components and add them.

...

The idea is that the worst case branching is $(1, 6)$ and that one is guaranteed to branch nicely on the 1-branch, giving a better branching.

4 Metaheuristics

The metaheuristics we look at are almost all based on gradient ascent from mathematics. In gradient ascent we want to maximize the value of $f(x)$ over all possible values of x. We do not necessarily have a way to compute $f(x)$, but we know $f'(x)$ - the slope of $f(x)$. We then keep on computing

$$x = x + \alpha \nabla f'(x)$$

until we find the optimal value of x. Maybe we even know $f''(x)$ also. Then we use Newton's method and add $\alpha(\nabla f'(x))/f''(x)$.

4.1 Hill-climbing

However in algorithms we are rarely able to compute $f'(x)$ and even more rarely $f''(x)$.

We do however often have a way to assess the value of a potential solution x. For instance we can sum the edges of a hamilton cycle to assess a possible TSP tour. We are also often able to tweak a solution a little bit (eg. swapping the spots of two vertices in a hamilton cycle given that the edges exist).

The idea of hill climbing is to tweak the current solution a little bit and use the new solution if it is better. If we use steepest-ascend hill climbing we make $n-1$ tweaks to a solution and pick the best of these n total potential solutions.

Tweaking the solution much will give us better exploration while it might accidentally overshoot the peaks of the different hills and thus never find an optimal solution.

4.2 Global optimization

We often want to make a global optimization algorithm. That is: Given enough time, the algorithm *will* find the global optimal solution. Just using hill climbing we cannot be sure, that we tweak the algorithm enough to avoid local maxima. We might not tweak enough to jump from one hill to the next.

We can fix this by applying random restarts to the hill climbing algorithm. Say when we start climbing up one hill we're given some random amount of time to do this. When this time is up we start at a new random point and climb up this new hill.

For some graphs hill climbing will do nice and for some it won't. We say that a graph where a small change in x-value causes a small change in y-value is smooth. However this is not enough for hill climbing to do well. We must also have an informal gradient. For instance some graphs might be very smooth but unless you land in 2% or the graph you will be lead up to a suboptimal solution. (page 18/20)

In general we can make a heuristic global by the following 4 ideas:

Adjust tweaking Occasionally make large tweaks.

Adjust selection Occasionally go down hills.

Start over Occasionally start over at a random location

Large sample Run the algorithm in parallel.

4.3 Simulated annealing

This is a modified version of hill climbing. We still pick tweaked solutions R over the current solution S if $f(R) < f(S)$. But if not we might still pick R according to a parameter t. We do this with probability:

$$P(t, R, S) = e^{\frac{f(R)-f(S)}{t}}$$

We start out with t being a high number and decrease it each iteration until it is 0.

In beginning this simulates a "random walk" and as t decreases it looks more and more like a hill climbing algorithm. If we set t large enough it is clear that this is a global algorithm.

4.4 Tabu search

Tabu search uses a list of recently considered solutions that we are not allowed to visit again for some time. So when we walk up a hill we HAVE to walk down the other side because we cannot return.

One way of doing this is to keep a list L of some length l, so when we find a new solution R we add it to L and if $|L| > l$ we remove the oldest element.

Note that if we use a real-valued space it is not very likely that we get the same solution twice, so Tabu search is best in discrete space. Also in other cases it might be very easy to stick around the same hill long enough to crawl up it again (eg. a very large search space).

A modified Tabu search stores tweaks rather than solution. For instance in TSP we might swap the spots of v, u in the permutation. We then add v, u to L, and then we aren't allowed to swap their positions again for a while. We can do this eg. by a hash-table where we map each tweak (key) to a timestamp. We must provide this table to the method that creates new potential solutions. This method is a bit different from the rest because we don't consider a solution as an atomic entity, rather it consists of different features that we can taboo.

4.5 Tweaking and neighbourhoods

When tweaking one way is to define a neighbourhood function $N(s) \subseteq S$ for each element $s \in S$. For instance in TSP a neighbourhood function could be to swap any two vertices in the path, leading to a $O(n^2)$ size neighbourhood.

A neighbourhood function also relies on the way we represent a solution. For instance if we represent a solution as a permutation of n vertices it might be easy to swap two vertices. If we represent a solution as a list of edges it might not.

Larger neighbourhoods take longer time to search (in eg. steepest-ascent hill climbing), but they also usually lead to better local optima.

It is also important that the neighbourhood function makes it possible to reach all solutions by going through some sequence of solutions. Note that even though there might be an obvious way from some solution s to s' using the neighbourhood function it doesn't necessarily go through feasible solutions, so the way might not be real.