Branch-and-Bound for the Travelling Salesman Problem

Leo Liberti

LIX, École Polytechnique, F-91128 Palaiseau, France
Email:liberti@lix.polytechnique.fr

March 15, 2011

Contents

1 The setting

These notes complement the lecture on Branch-and-Bound for the Travelling Salesman Problem given in the course INF431 (edition 2010/2011). The lecture slides are more informal and attempt to convey the important concepts of the Branch-and-Bound algorithm, whereas these notes provide a formal treatment of the correctness of the Branch-and-Bound algorithm.

1.1 Graphs

Warning: many of the definitions below are not "standardized", i.e. you might find slight variations thereof in different texts by different authors.

1. A *graph* is a pair $G = (V, E)$ where V is a set of *vertices* and E is a set of subsets of V of cardinality either 1 or 2, i.e. the elements of E have the form $\{u, v\}$ where $u, v \in V$ (if $u = v$ then $\{u, v\} = \{v\}$ and the edge is called a *loop*). A *simple* graph has no loops. Given a graph G, we also denote its vertex set by $V(G)$ and its edge set by $E(G)$.

2. A *digraph*, or directed graph, is a pair $G = (V, A)$ where V is a set of *nodes* and $A \subseteq V \times V$ is a set of *arcs*. A simple digraph has no directed loops (i.e. elements of the form (v, v) for some $v \in V$). Given a digraph G, we also denote its node set by $V(G)$ and its arc set by $A(G)$.

3. Given a digraph $G = (V, A)$, the *underlying (undirected) graph* is a graph $G' = (V, E)$ where $E = \{\{u, v\} \mid (u, v) \in A\}$.

4. A graph $H = (U, F)$ is a *subgraph* of a graph $G = (V, E)$ if $U \subseteq V$ and $F \subseteq E$. A subgraph H is *induced* if B contains all edges $\{u, v\} \in E$ such that $u, v \in U$. Induced subgraphs are uniquely defined by their vertex set U, and denoted by $H = G[U]$. An equivalent notion holds for digraphs.

5. A subgraph H is *spanning* in G if $V(H) = V(G)$.

6. Given a graph $G = (V, E)$ and $v \in V$, the *star of* v is the vertex set $\delta(v) = \{u \in V \mid \{u, v\} \in E$. The *degree of* v is $\deg(v) = |\delta(v)|$.

7. Given a digraph $G = (V, A)$ and $v \in V$, the *incoming star of* v (or "instar") is the node set $\delta^-(v) = \{u \in V \mid (u, v) \in A\}$. The *outgoing star of* v (or "outstar") is the node set $\delta^+(v) = \{u \in V \mid (v, u) \in A\}$. The *indegree* of v is $\deg^-(v) = |\delta^-(v)|$ and the *outdegree* of v is $\deg^+(v) = |\delta^+(v)|$.

8. Given a graph $G = (V, E)$, a *path* in G is a subgraph P of G with an order on $V(P)$ such that for all $u, v \in V(P)$ with unit rank difference in the order $\{u, v\} \in E(P)$. The first vertex in the path is called the *source* and the last vertex is called the *destination*. A path is *simple* if each vertex has degree at most two. The *length* of a path P is $|E(P)|$. Equivalent notions hold for digraphs.

9. A graph (resp. digraph) G is *connected* (resp. *strongly connected*) if for each pair $(s, t) \in V^2$ there exists a path from s to t in G.

10. Given a graph $G = (V, E)$, a *cycle* in G is a subgraph C of G where each vertex in $V(C)$ has even degree. A cycle is *simple* if it is connected and every vertex has degree two.

11. Given a digraph $G = (V, E)$, a *circuit* is G is a directed simple path with equal source and destination nodes.

12. A cycle (resp. circuit) is *Hamiltonian* if it is spanning. A Hamiltonian circuit is also called a *tour*.

13. A graph (resp. digraph) is *acyclic* if it does not have a cycle (resp. circuit).

14. A *forest* (resp. *arborescence*) is an acyclic graph (resp. digraph — acyclic digraphs are also called DAGs, which stands for Directed Acyclic Graphs).

15. A *tree* is a connected forest in a graph. In the context of digraphs, a *tree* is a DAG whose underlying undirected graph is a tree, and which possesses a distinctive *root node* which is the source node to paths to all other nodes. All connected DAGs (and therefore all trees) have a unique root.

16. A node of a tree with degree 1 (resp. a node of a DAG with outdegree 0) is called a *leaf node*.

17. In a DAG, any path from the root to any other node v is a *branch* to v; any node u on a branch to v is an *ancestor* of v. Given two nodes v, w of a DAG, any node u such that there are paths P_v from u to v and P_w from u to w is called a *common ancestor* of v, w; the root is a common ancestor of all nodes in the DAG. If no vertex apart from u is an ancestor of v, w on either P_v or P_w then u is the *nearest common ancestor*.

1.2 Problems

1.2.1 Decision problems

Informally speaking, a decision problem is a question which only admits the answers “yes” or “no”. More formally, it can be considered a function f (called the decision function) defined over the language $\{0,1\}^*$ and mapping into $\{0,1\}$: formal questions (which, in practice, are often about graphs) are encoded into a string in $\{0,1\}^*$, and the answers are either 0 (for “no”) or 1 (for “yes”).

For example, the Hamiltonian cycle problem is as follows.

HAMILTONIAN CYCLE

Given a graph $G = (V, E)$, does it admit a Hamiltonian cycle?

Each time a different graph G is given, the answer may vary. For example, for all disconnected graphs the answer is “no”. For all complete graphs the answer is “yes”. In other words, the problem itself cannot be solved before a concrete graph is provided: we can see a problem as a sentence which is parametrized on G. Any assignment of consistent values to the parameters of a problem is called an *instance* of the problem (which can therefore also be seen as a collection of instances). Formally, an instance is an element of the domain of the decision function.

A simple answer “yes” or “no” is relatively useful: we are usually also interested in a proof of validity of the provided answer. Such a proof is also called *certificate*. For example, the complete graph $K_4 = (\{1,2,3,4\}, \{\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,4\},\{3,4\}\})$ on four vertices is an instance of HAMILTONIAN CYCLE, and the Hamiltonian cycle $P = (\{1,2,3,4\}, \{\{1,2\},\{2,3\},\{3,4\},\{1,4\}\})$ is a certificate for the answer “yes” (the instance is also termed a “yes” instance).

1.2.2 Optimization problems

An optimization problem asks instead for some kind of a “structure” of “optimum cost”: a structure is usually a set of “elements”, each of which is assigned a numerical cost. Such elements can be combined in different ways to generate a set of structures, and the problem asks for the cost-wise best structure in the set. Here, “best” might mean minimum or maximum according to specification. Formally, an optimization problem is a pair of functions f, g where $f : \{0,1\}^* \to \{0,1\}$ is the decision function and $g : \{0,1\}^* \to \mathbb{Q}$ is the cost function, together with an optimization direction in $\{\min, \max\}$. An instance is an element of the domain of f, g. As for decision problems, we are not just interested in the answer (“yes”/“no” and the cost) but in the certificate too, i.e. the proof that the answer holds.

For example, the travelling salesman problem is as follows.

TRAVELLING SALESMAN PROBLEM (TSP)

Given a complete digraph $G = (V, A)$ with arc cost function $d : A \to \mathbb{Q}_+$, find the tour of minimum cost.

Since a complete digraph is uniquely identified by the number of nodes $|V|$, the instance is here a pair (n, D) where $n = |V|$ and $D = (d_{ij})$ is an $n \times n$ matrix with entries in $Q_+ \cup \{\infty\}$: $d_{uv} = d((u,v))$ for all $(u,v) \in A$ and $d_{vv} = \infty$ for all $v \in V$.

Since a tour is a spanning subgraph of G, it is uniquely identified by its arc set S. The “elements” here are the arcs of the digraph, and the “structures” are tours. In the TSP setting, we shall define a *solution* any set of arcs. A solution is *feasible* if it is a tour and *infeasible* otherwise. A solution is *optimal* for the TSP if it is feasible and has minimum cost amongst all feasible solutions.

2 Branch-and-Bound

The Branch-and-Bound algorithm explicitly explores a large search space by recursively branching on disjunctions of possible values assigned to various decisions. Some of the recursions can be avoided if it can established that no subsequent recursive call contains a good solution. Typically, with optimization problems, recursions are avoided by employing a bound to the optimal value: if a good solution was already identified and the current bound is worse, there is no point in further recursion.

In the rest of the section we shall discuss a pure branching algorithm first, and add the bounding part as a modification yielding better performance.

2.1 A pure branching algorithm

Extend the arc cost function $d : A \to \mathbb{Q}_+$ to act on solutions S as follows:

$$d(S) = \sum_{(u,v)\in S} d_{uv}. \tag{1}$$

Let $\mathcal{P}(A)$ be the set of all subsets of A. Define the following functions:

- $\tau : \mathcal{P}(A) \to \{0,1\}$ which maps a solution S to 1 if S is a tour, and to 0 otherwise;
- $\phi : \mathcal{P}(A)^2 \to \{0,1\}$ which maps an arc set pair (Y,N) to 1 if Y can be extended to a tour *without using the arcs in* N, and 0 otherwise; specifically:

$$\phi(Y,N) = 0 \quad \text{if } (|Y \cap N| \geq 1) \ \vee \ (Y \cup N = A) \ \vee \ (Y \text{ contains a circuit of length } < n).$$

We claim that Alg. 1 can solve the TSP when called as `treeSearch`$(\varnothing,\varnothing,\varnothing)$.

Algorithm 1 `treeSearch`(Y,N,S)

```
1: if $\tau(Y) = 1$ then
2:   if $d(Y) < d(S)$ then
3:     $S = Y$
4:   end if
5: else
6:   if $\phi(Y,N) = 1$ then
7:     choose $a \in A \smallsetminus (Y \cup N)$
8:     treeSearch($Y \cup \{a\}, N, S$)
9:     treeSearch($Y, N \cup \{a\}, S$)
10:  end if
11: end if
```

2.1 Lemma
During the execution of Alg. 1, Y,N,S are always subsets of A.

Proof. Alg. 1 calls itself only if it reaches Lines 8-9, in which case it also reaches Line 7. Since a is an arc in $A \smallsetminus (Y \cup N)$, and since Y,N are updated in the recursive calls, both are always subsets of A. □

Let $\mathcal{G} = (\mathcal{V},\mathcal{A})$ be the following digraph, generated by the execution of Alg. 1: each node of $\mathcal{V}$ is the triplet (Y,N,S) in memory at Line 11, and there is an arc $((Y,N,S),(Y',N',S'))$ whenever Y',N',S' are the arguments of a recursive call from `treeSearch`(Y,N,S). For a node $\alpha = (Y,N,S)$ of $\mathcal{V}$ we write $Y = Y_\alpha$, $N = N_\alpha$, $S = S_\alpha$. Because $\mathcal{G}$ is the representation of a recursion structure, we sometimes refer to nodes in an outgoing star as *subnodes*.

Remark

From now on, we deal with two separate digraphs: the complete digraph $G = (V, A)$ where we look for an minimum cost circuit, and the digraph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ representing the recursive calls in the execution of `treeSearch`$(\varnothing, \varnothing, \varnothing)$. Particular care must be paid not to confuse the two digraphs, for we shall discuss nodes and circuits in both G and $\mathcal{G}$.

2.2 Lemma

If ν is an ancestor of μ in $\mathcal{G}$ then $Y_\mu \supset Y_\nu$ and $N_\mu \supset N_\nu$.

Proof. Directly by Lines 8-9. □

2.3 Lemma

$\mathcal{G}$ has no circuits.

Proof. By Lines 7-9, each non-leaf node $(Y, N, S) \in \mathcal{V}$ has two subnodes $(Y \cup \{a\}, N, S)$ and $(Y, N \cup \{a\}, S)$, where $a \notin (Y \cup N)$. In particular, by Lines 6-7, each arc $((Y, N, S), (Y', N', S')) \in \mathcal{A}$ is such that $|Y' \cup N'| = |Y \cup N| + 1$. Now suppose there is a circuit in $\mathcal{G}$ with ordered node set $((Y_0, N_0, S_0), \dots, (Y_{k-1}, N_{k-1}, S_{k-1}))$: then $|Y_i \cup N_i| < |Y_{i+1 \pmod k} \cup N_{i+1 \pmod k}| + 1$ for all $i \in \{0, \dots, k-1\}$, which implies $|Y_0 \cup N_0| < |Y_0 \cup N_0|$, a contradiction. □

2.4 Lemma

In any node $\omega \in \mathcal{V}$, we have that Y_ω, N_ω uniquely determine S_ω.

Proof. Consider nodes $\alpha, \beta \in \mathcal{V}$ with $Y_\alpha = Y_\beta \wedge N_\alpha = N_\beta$ $(*)$. Let γ be their nearest common ancestor, and consider the paths P_α and P_β from γ to α and β. Since each node in the paths (apart perhaps from α, β) have subnodes, τ is always 0 at Line 1, so S is never updated at Line 3. Thus $S_{\alpha'} = S_{\beta'} = S_\gamma$ for α' the parent node of α and β' the parent node of β. Now, since $Y_\alpha = Y_\beta$ and $N_\alpha = N_\beta$, if α has subnodes, β also has subnodes; in this case $\tau(Y_\alpha) = \tau(Y_\beta) = 0$ and $S_\alpha = S_{\alpha'} = S_{\beta'} = S_\beta$. Otherwise $\tau(Y_\alpha) = \tau(Y_\beta) = 1$ which implies $S_\alpha = Y_\alpha = Y_\beta = S_\beta$. □

By Lemma 2.4, we can also refer to nodes of $\mathcal{G}$ simply as pairs (Y, N).

2.5 Lemma

At termination of a call `treeSearch`$(\varnothing, \varnothing, \varnothing)$ *to Alg. 1, the set $\{Y_\alpha \mid \alpha \in \mathcal{V} \wedge \alpha \text{ is leaf}\}$ contains all tours of the given instance.*

Proof. Let $\bar{S} \subset A$ be a tour and let $<$ be the order on A used to choose a in Line 7. Consider the branch from the root of $\mathcal{G}$ obtained by taking left subnodes (Line 8) whenever $a \in \bar{S}$ at Line 7 and right subnodes (Line 9) whenever $a \notin \bar{S}$ at Line 7. The branch exists in $\mathcal{G}$ since any proper subset Y of $\bar{S}$ is not a tour and does not contain circuits of length $< n$, so $\tau(Y) = 0$ and $\phi(Y, N) = 0$; furthermore, by construction it contains a node α where $Y_\alpha = \bar{S}$ and $N_\alpha = \{a \in A \mid a \notin \bar{S} \wedge \exists b \in \bar{S}\ (a < b)\}$. Since $\tau(Y_\alpha) = 1$, by Line 1 this is the last node of the branch, i.e. a leaf node. □

2.6 Theorem

A call `treeSearch`$(\varnothing, \varnothing, \varnothing)$ *to Alg. 1 solves the TSP.*

Proof. By Lemma 2.1, Y, N are always subsets of A; by Lemma 2.3, $\mathcal{G}$ is a DAG, each node of which is determined by Y, N by Lemma 2.4. Since the number of subsets of A is finite, the algorithm terminates. By Lemma 2.5, the leaves of $\mathcal{G}$ contain all the tours, so by Lines 2-3, S contains the tour of minimum cost at termination. □

2.1.1 Complexity

2.7 Lemma
All nodes of $\mathcal{G}$ with indegree greater than 1 are leaves.

Proof. Let $\alpha \in \mathcal{V}$ with indegree > 1: this means that $\mathcal{A}$ contains two arcs (β, α), (γ, α) with $\beta \neq \gamma$. By Lemma 2.2, $Y_\alpha \supset Y_\beta$, $N_\alpha \supset N_\beta$, $Y_\alpha \supset Y_\gamma$ and $N_\alpha \supset N_\gamma$. Now let θ be the nearest common ancestor of β, γ; by Line 7 there must be $a \in A \smallsetminus (Y_\theta \cup N_\theta)$ such that $a \in Y_\beta$ and $a \in N_\gamma$. Thus, $a \in Y_\alpha \cap N_\alpha$, which means that $\phi(\alpha) = 0$, so α has no subnodes. $\square$

By Lemmata 2.4 and 2.7, each non-leaf node in $\mathcal{V}$ corresponds to a choice of Y, N such that $\tau(Y) = 0 \wedge \phi(Y, N) = 1$, i.e. pairs (Y, N) such that Y contains no circuit, $Y \cap N = \varnothing$ and $Y \cup N \subsetneq A$. We first evaluate the number of pairs (Y, N) of subsets of A such that $Y \cap N = \varnothing$ and $Y \cup N \subsetneq A$; for each choice of $Y \subsetneq A$ there are all possible choices of $N \subsetneq A \smallsetminus Y$. Since $|A| = n(n-1)$, we have:

$$\begin{aligned} & \sum_{h=0}^{n(n-1)} \binom{n(n-1)}{h} \sum_{\ell=0}^{n(n-1)-h-1} \binom{n(n-1)-h}{\ell} \\ = & \sum_{h+\ell<n(n-1)} \binom{n(n-1)}{h} \binom{n(n-1)-h}{\ell} \\ = & \sum_{h+\ell<n(n-1)} \frac{n!(n-1)!}{(h\ell)!(n(n-1)-h-\ell)!}. \end{aligned}$$

To this total, we have to take away the number of circuits of i nodes for $1 < i < n$, i.e. $\sum_{1<i<n}(i-1)!$.

Amongst the leaf nodes there are the circuits of any length $\leq n$: there are $\sum_{1<i\leq n}(i-1)!$ of them. Further, it is easy to show that in a binary tree the number of leaf nodes is $t+1$ where t is the number of non-leaf nodes; and by Lemma 2.7 this is an upper bound for the case of $\mathcal{G}$. All in all, we have

$$2\left(\sum_{h+\ell<n(n-1)} \frac{n!(n-1)!}{(h\ell)!(n(n-1)-h-\ell)!} + (n-1)!\right) + 1 \qquad (2)$$

nodes.

As concerns worst-case complexity, a coarse analysis bounds the number of pairs of subsets of A by $2^{2n(n-1)}$, so we obtain an overall bound $O(2^{2n^2} + (n-1)!)$.

2.2 Enters the bound

In order to improve on the practical performance of Alg. 1, we introduce a technique to compute a lower bound $\bar{d}$ to the optimal cost $d(S)$ of the best circuit $S \subset A$ in a subgraph $\mathcal{G}_\alpha$ rooted at a given node α of the $\mathcal{G}$. If this bound is worse than the cost $d(S')$ best circuit S' found so far (the *incumbent*), i.e. $\bar{d} \geq d(S')$, then there is no need for branching at this node: the subgraph of $\mathcal{G}$ rooted at α cannot contain a better circuit than the incumbent. The node α is said to be *pruned by bound*. This important insight, by the way, also implies that Thm. 2.6 will hold for a `treeSearch` algorithm modified this way.

Alg. 2 finds such a lower bound, based on choosing the best "next node" for each node in V. We remark that this algorithm provides a bound which varies in function of the node element Y, N. Notationally, for a set of arcs S on the node set V we define $\Lambda^-(S) = \{v \in V \mid \exists u \in V \ (u, v) \in S\}$ and $\Lambda^+(S) = \{v \in V \mid \exists u \in V \ (v, u) \in S\}$. Line 7 is reached if no "next node" can be chosen for a given u due to incompatibilities between Y and N: if such is the case, no circuit can ever be found with such Y, N and the subgraph need not be explored.

Algorithm 2 `lowerBound`(Y, N)

```
 1: $R = Y$
 2: for $u \in V \smallsetminus \Lambda^-(Y)$ do
 3:    if $\{(u,v) \notin N \mid v \in V\} \neq \varnothing$ then
 4:       $a = \mathrm{argmin}\{d_{uv} \mid (u,v) \notin N\}$
 5:       $R = R \cup \{a\}$
 6:    else
 7:       return $\infty$
 8:    end if
 9: end for
10: return $d(R)$
```

2.8 Lemma
Alg. 2 called with arguments Y, N gives a lower bound to the optimal cost of the circuits encoded in the nodes of the subgraph of $\mathcal{G}$ rooted at the node $\alpha \in \mathcal{V}$ with $Y_\alpha = Y$ and $N_\alpha = N$.

Proof. Let $\alpha \in \mathcal{V}$, and let $\lambda =$ `lowerBound`(Y_α, N_α). If no node of the subgraph $\mathcal{G}_\alpha$ of $\mathcal{G}$ rooted at α encodes a tour, then the theorem trivially holds because by convention we set $d(\varnothing) = \infty$, so assume at least one node β of $\mathcal{G}_\alpha$ encodes a tour S. Then by Line 1 in Alg. 1 $Y_\beta = S$, so Y_β is a tour. Since α is an ancestor of β, by Lemma 2.2 $Y_\beta \supset Y_\alpha$ and $N_\beta \supset N_\alpha$. Thus, we can write $S = Y_\alpha \cup S'$ for some arc set S'. By Line 7 in Alg. 1, $S \cap N_\beta = \varnothing$, so that $S \cap N_\alpha = \varnothing$ and hence $S' \cap N_\alpha = \varnothing$. By Lines 4-5 in Alg. 2, $R = Y_\alpha \cup R'$ for some arc set R' with $R' \cap N_\alpha = \varnothing$. Also, because S is a tour, $\Lambda^-(Y_\alpha)$ and $\Lambda^-(S')$ partition $\Lambda^-(S) = V$. By Line 2 in Alg. 2, $\Lambda^-(R') = V \smallsetminus \Lambda^-(Y_\alpha)$, which implies $\Lambda^-(R') = \Lambda^-(S')$. Furthermore, by Line 4 in Alg. 2, each arc $(u, v) \in R'$ is chosen as having minimum cost over all the arcs in $\delta^+(u) \smallsetminus N_\alpha$. Thus, $d(R') \leq d(S')$. Since $S = Y_\alpha \cup S'$ and $R = Y_\alpha \cup R'$, $d(R) \leq d(S)$ as claimed. □

More in general, one might see each property of the sought structure (such as tours) as a constraint imposed on simpler structures (such as arc sets S). Since all tours H have $\Lambda^-(H) = V$, the set $\mathcal{L}$ of all arc sets S such that $\Lambda^-(S) = V$ has fewer constraints than the set $\mathcal{H}$ of all tours. In other words, $\mathcal{L}$ is a *relaxation* of $\mathcal{H}$, or, more precisely, $\mathcal{L} \supset \mathcal{H}$. In general, optimal cost structures in relaxation sets provide bounds for the optimal cost structures of interest, as

$$\min_c \mathcal{L} \leq \min_d \mathcal{H} \leq \max_d \mathcal{H} \leq \max_c \mathcal{L} \tag{3}$$

for all cost functions $c : \mathcal{L} \to \mathbb{R}$, $d : \mathcal{H} \to \mathbb{R}$ such that $c(S) = d(S)$ for all $S \in \mathcal{H}$. Lemma 2.8 can also be seen as a corollary of this result.

2.2.1 The relaxed solution

On top of returning the lower bound $\bar{d}$, Alg. 2 can also returned a *relaxed solution* R such that $\bar{d} = d(R)$. This solution can be used profitably to speed up the search by replacing the test $\tau(Y) = 1$ at Line 1 in Alg. 1 with $\tau(R) = 1$. All the results in Sect. 2.1 hold anyhow: if R is a tour (i.e. $\tau(R) = 1$) at a node α of $\mathcal{G}$ then there must be a node β in the subgraph $\mathcal{G}_\alpha$ of $\mathcal{G}$ rooted at α with $Y_\beta = R$ by Lemma 2.5. Thus, the effect of this test replacement is simply that of finding circuits earlier on in the search.

2.3 Feasible nodes

If it can be established that a node $\alpha \in \mathcal{V}$, independently of its associated lower bound, is such that Y_α cannot be extended to a tour without using the arcs in N_α, then by Lemma 2.2 it is clear that the subgraph $\mathcal{G}_\alpha$ of $\mathcal{G}$ rooted at α cannot contain any tour, and therefore α can be discarded. Such nodes are said to be *pruned by infeasibility.* By constrast, all other nodes in $\mathcal{G}$ are called *feasible nodes.* With reference to the TSP, the main feasibility test is encoded in the definition of ϕ and implemented at Line 6 of Alg. 1.

2.4 A good order for processing nodes

It is well known that a recursive procedure such as Alg. 1 explores generates a depth-first order on the exploration of the nodes of $\mathcal{G}$. On the other hand, intuitively the node order has an impact on the size of $\mathcal{G}$ (and hence on the CPU time taken by Branch-and-Bound): if we were able to identify the optimal circuit S^* immediately, then a lot more nodes would be pruned by bound, as a lower value for $d(S^*)$ would make the inequality $\bar{d} \geq d(S^*)$ true for comparatively mode nodes α with lower bound equal to $\bar{d}$.

The most commonly employed strategy for locating rooted subgraphs of $\mathcal{G}$ likely to contain good TSP optima is that of processing nodes with lowest lower bound first. Specifically, if $\bar{d}$ is the lower bound computed by Alg. 2 at the current node α, the subnodes of α are stored along with $\bar{d}$ in a priority queue Q ordered according to values of $\bar{d}$.

2.5 The Branch-and-Bound algorithm for the TSP

Algorithm 3 `branchAndBound`(n, d)

```
Require: set $Q$ of nodes $(Y, N, S, c')$ ordered by increasing $c' \in \mathbb{R}$
  Let $Q = \{(\varnothing, \varnothing, \varnothing, -\infty)\}$
  while $Q \neq \varnothing$ do
    Let $\alpha = (Y, N, S, c') = \mathrm{argmin}_{c'} Q$; let $Q = Q \smallsetminus \{\alpha\}$
    if $\phi(\alpha) = 1$ then
      Let $R =$ lowerBound$(\alpha)$
      if $d(R) < d(S)$ then
        if $\tau(R) = 0$ then
          if $Y \cup N \neq A$ then
            choose $a \in A \smallsetminus (Y \cup N)$
            Let $\beta = (Y \cup \{a\}, N, S, d(R))$; let $\gamma = (Y, N \cup \{a\}, S, d(R))$
            Let $Q = Q \cup \{\beta, \gamma\}$
          end if
        else
          Let $S = R$
        end if
      end if
    end if
  end while
  return $S$
```