Appendix of "*Computational Protein Design Using AND/OR Branch and Bound Search*"

Yichao Zhou[1], Yuexin Wu[1], and Jianyang Zeng[1,2,⋆]

[1] Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, 100084, P. R. China
[2] MOE Key Laboratory of Bioinformatics, Tsinghua University, Beijing, 100084, P.R. China

A1 Introduction to Traditional Branch-and-Bound Search

Suppose we try to find the global minimum value of the energy function $E(r)$, in which $r \in R$ and R is the conformational space of the rotamers. The BnB algorithm executes two steps recursively. The first step is called *branching*, in which we split the conformational space R into two or more smaller spaces, i.e., R_1, R_2, ..., R_m, where $R_1 \cup R_2 \cup \cdots \cup R_m = R$. If we are able to find $\hat{r}_i = \arg\min_{r \in R_i} E(r)$ for all $i \in \{1, 2, \ldots, m\}$, we can compute the minimum energy conformation in the conformational space R by identifying one of $\hat{r}_i$ that has the lowest energy.

The second step of BnB is called *bounding*. Suppose the current lowest energy conformation is $\overline{r_i}$. For any sub-space R_j, if we can ensure that the lower bound of the energy of all conformations in R_j is greater than $E(\overline{r_i})$, we do not need to further search this sub-space, that is, we can prune the whole sub-space R_j safely. The lower bound of the energy of the conformations in a given space usually can be computed based on some heuristic functions.

The BnB algorithm generally performs the branching step recursively until the current conformational space contains only one single conformation. The space generated from the branching step can form a *BnB search tree*, in which the union of sub-spaces represented by the children of a node covers the whole conformational space of this node. In the protein design problem, we can split a conformational space by assigning a particular rotamer in a residue. For each node in the search tree, the bounding step is applied to prune some branches and thus shorten the search time. Fig. A1 shows an example of the traditional branch-and-bound (BnB) search tree.

In order to traverse the BnB search tree, a queue Q is often used to store the nodes to be expanded. Initially, Q only contains the node representing the whole conformational space. Also, we maintain a global variable u to store the current lowest energy value. Initially, u can be initialized to the energy of any conformation. In practice, we often use a stochastic local search algorithm to generate a local optimal conformation so that it can be used to prune more nodes at the beginning of the search process. BnB can be executed by looping the following steps until Q becomes empty:

1. Extract the conformational space R from Q;
2. If R only contains a single conformation, update u using the energy of this conformation and then restart the loop;

⋆ Corresponding author: Jianyang Zeng. Email: zengjy321@tsinghua.edu.cn.

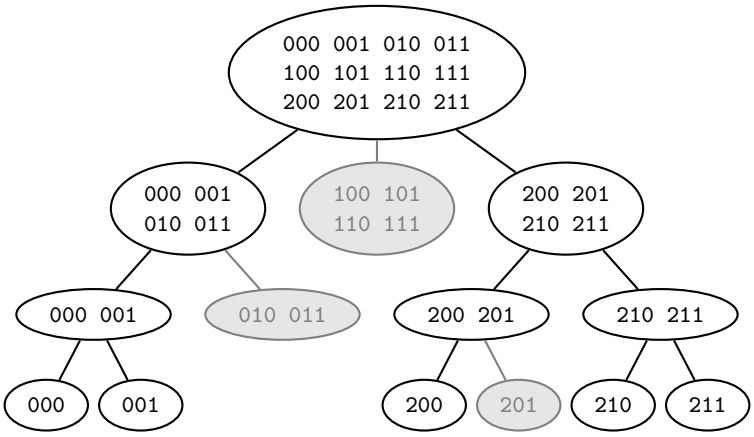

Fig. A1. An example of a branch-and-bound search tree, which contains three mutable residues. The first residue has three allowed rotamers while the other two only have two allowed rotamers. The coding of a conformation is given by three integers, each of which is the index of the rotamer in the corresponding residue. Each tree node represents a conformational space. For each tree node, we split its conformational space by determining a particular rotamer in a residue. The shaded nodes are pruned in the bounding steps because the lower bound of the energy values in these nodes given by the heuristic function is greater than one of the optimal conformations in its siblings. A brute-force search for the full conformational space requires the computation of twelve conformations to guarantee the GMEC solution, while BnB only needs to compute five of them.

3. Otherwise, split R into $R_1, R_2, \ldots, R_m$ by fixing a particular rotamer of a residue;
4. For each $i \in \{1, 2, \ldots, m\}$, compute the lower bound of the energy for all nodes in space R_i. If it is smaller than u, push R_i to Q.

Usually Q is implemented by a LIFO queue (i.e., stack), so that the energy of current best conformation u can be updated as early as possible to prune more branches. In this case, the BnB algorithm runs in a depth-first-search mode. Another benefit of this mode is that memory used by the BnB algorithm is only proportional to the depth of the search tree, namely the number of mutable residues in the protein design problem. On the other hand, other search strategies, such as A* search, can store an exponential number of nodes in memory in the worst case. Alg. A1 gives the pseudocode of an implementation of the BnB search algorithm.

A2 Details of AND/OR Branch-and-Bound Search

Alg. A2 provides the pseudocode of the AOBB search algorithm. For simplicity, we leave out the code of constructing solution trees and only describe the procedure of computing $v(\cdot)$ values. For each OR node x, we use $c(x)$ to store the pointer to the child with the best $v(\cdot)$ value if the sub-tree rooted at x has been fully explored (Line 13), or the pointer to the child whose sub-tree is currently being visited (Line 6), or a null pointer if x has not been visited (Line 1).

The bounding step can also be performed in AOBB to prune unpromising branches. The heuristic function $h(x)$ returns a lower bound of $v(x)$, which is used to compute

Alg. A1 Traditional branch-and-bound search

```
1: u ← ∞                                                    ▷ Initialize u to infinity.
2: procedure BRANCH-AND-BOUND(R)
3:     if |R| = 1 then                                      ▷ Termination condition
4:         Let r be the conformation in R
5:         u ← min(u, E(r))
6:         return r
7:     end if
8:     if h(R) > u then                                     ▷ h(·) is a heuristic function
9:         return null                                      ▷ Bounding step
10:    end if
11:    Split R to R_1, R_2, ..., R_m by fixing a rotamer of a particular residue
12:    for i ← 1, t do
13:        r_i ← BRANCH-AND-BOUND(R_i)
14:    end for
15:    return arg min_{r_i} E(r_i)
16: end procedure
```

Alg. A2 An implementation of AND/OR branch-and-bound search

```
1: Initialize c(x) to null for all x
2: function AOBB(x)
3:   if x is an OR node then
4:     v(x) ← +∞
5:     for all y ∈ child(x) do
6:       c(x) ← y
7:       call AOBB(y)
8:       if e(y) + v(y) < v(x) then
9:         c_0 ← y
10:        v(x) ← e(y) + v(y)
11:      end if
12:    end for
13:    c(x) ← c_0
14:  else if x is an AND node then
15:    v(x) ← 0
16:    for all y that is x's ancestors do
17:      if TREE-HEURISTIC(y) > v(y) then
18:        mark x as pruned
19:        return +∞
20:      end if
21:    end for
22:    for all y ∈ child(x) do
23:      v(x) ← v(x) + AOBB(y)
24:    end for
25:  end if
26:  return v(x)
27: end function
28: function TREE-HEURISTIC(x)
29:   if x is an AND node then
30:     s ← 0
31:     for all y ∈ child(x) do
32:       s ← s + TREE-HEURISTIC(y)
33:     end for
34:     return s
35:   else
36:     if c(x) = null then
37:       return h(x)
38:     end if
39:     t ← TREE-HEURISTIC(c(x))
40:     return e(c(x)) + t
41:   end if
42: end function
```

the heuristic value of a incomplete solution tree. When performing the bounding step for an AND node x, we examine all the OR ancestors nodes of x. For any OR ancestor y, if the heuristic value for the current incomplete solution tree rooted at y (computed by TREE-HEURISTIC(y)) is worse than $v(y)$ computed from another explored branch y, we can safely prune the current sub-tree rooted at x (Lines 16-21 of Alg. A2). The function TREE-HEURISTIC(x) computes the heuristic value for the current incomplete solution tree rooted at x using a method similar to that of Equation (2) of the origi-

nal paper, except that when it meets an unexplored nodes, it returns $h(x)$ as a lower bound.

In practice, we can maintain data structures $c(\cdot)$ and $v(\cdot)$ carefully to free the memory occupied by useless nodes so that the whole AOBB search algorithm can still run in bounded memory. Therefore, space complexity of the AOBB search is still $O(n)$, where n is the number of mutable residues. The time complexity of AOBB in the worst case is $O(n * p^d)$, where p is the number of rotamers per residue and d is the depth of the pseudo-tree, as the size of the AOBB search tree is $O(p^d)$ and for each tree node we need to compute TREE-HEURISTI(x), whose time complexity is $O(n)$ assuming $h(x)$ can be computed in $O(1)$ time.

A3 Correctness for Finding Suboptimal Solutions

In this section, we provide the proof of Theorem 1 in the paper, which states the correctness of our merge algorithm for AND nodes. We re-state that theorem in Theorem A1 and the pseudocode of the merge operation of AND nodes in Alg. A3.

Alg. A3 Merge operation for AND nodes

```
1:  procedure MERGE-AND(x, y)
2:      b ← (1, 1, ..., 1)
3:      Let Q be a priority queue
4:      PUSH(Q, (∑_{i=1}^{t} v_{b_i}(y_i), b))
5:      for i ← 1 to k do
6:          (s, b) ← POP-MINIMUM(Q)
7:          a_i ← b
8:          v_i(x) ← ∑_{j=1}^{t} v_{b_j}(y_j)
9:          for j ← 1 to t do
10:             b' ← b
11:             b'_j ← b_j + 1
12:             PUSH(Q, (∑_{p=1}^{t} v_{b'_p}(y_p), b'))
13:         end for
14:     end for
15:     return a
16: end procedure
```

Theorem A1. *Alg. A3 guarantees the correctness of finding the k best solutions.*

Proof. It is sufficient to prove that in the ith iteration, the element with the ith smallest value is in the priority queue. This can be proved by contradiction.

Let i be the first round that the element with the ith value is not in the priority queue. Suppose $a_i = (a_{i1}, a_{i2}, \ldots, a_{it})$. Because $i \neq 1$, there exists $j \in \{1, 2, \ldots, t\}$ such that $a_{ij} > 1$. Sequence $s = (a_{i1}, \ldots, a_{i,j-1}, a_{ij} - 1, a_{i,j+1}, \ldots, a_{it})$ must have not been expended. Otherwise, according to Line 12, sequence a_i will be pushed to the priority queue, which contradicts the assumption. On the other hand, because $v_j(y)$

is monotone with respect to j, the value of sequence s is smaller than the value of sequence a_i. This means that sequence s should be expanded before sequence a_i and thus must have already been expanded, which gives the contradiction. ■

Also, we provide the implementation of the merge operation for OR nodes in Alg. A4. The correctness of this algorithm is self-evident, so we omit its proof here.

Alg. A4 Merge operation for OR nodes

```
1: procedure MERGE-OR(x, y)
2:     w ← (v_1(y_1) + e(y_1), ..., v_k(y_1) + e(y_1), ..., v_k(y_t) + e(y_t))
                                   ▷ concatenate v_i(y_j) + e(y_j) for all i and j
3:     SORT(w)
4:     for i ← 1 to k do
5:         v_i(x) ← w_i
6:     end for
7: end procedure
```

A4 Full Comparison Results on Core Redesign

Table A1 lists the full comparison results between A*-based and AOBB-based algorithms on the protein core redesign problem. The meanings of all labels are the same as those of Table 1, which are described in Section 3.1 of the paper.

PDB	Space size	# of A* states	# of AOBB states	OSPREY time	cOSPREY time	AOBB time
1I27	2.03e+20	OOM	29	OOM	OOM	< 1
1L9L	2.37e+19	OOM	1,599,481	OOM	OOM	2,885
1LNI	2.98e+13	OOM	3	OOM	OOM	< 1
1MWQ	9.28e+17	OOM	3	OOM	OOM	< 1
1OAI	3.27e+21	OOM	31	OOM	OOM	< 1
1PSR	1.94e+22	OOM	20,310	OOM	OOM	29
1R6J	3.45e+25	OOM	3,296,587	OOM	OOM	9,875
1T8K	6.32e+20	OOM	581,917	OOM	OOM	1,888
1TUK	1.73e+19	OOM	188,042	OOM	OOM	723
1UCR	6.69e+19	OOM	25	OOM	OOM	< 1
1UCS	1.09e+20	OOM	1,118	OOM	OOM	3
1ZZK	3.44e+15	OOM	255	OOM	OOM	< 1
2BT9	5.40e+21	OOM	3,643,732	OOM	OOM	9,592
2BWF	5.54e+22	OOM	517,258,245	OOM	OOM	1,467,951
2HS1	6.35e+16	OOM	100,117	OOM	OOM	161
2O9S	3.53e+17	OOM	3	OOM	OOM	< 1
2R2Z	7.47e+20	OOM	3	OOM	OOM	< 1
2WJ5	1.47e+20	OOM	140,412,110	OOM	OOM	728,506
3FIL	2.62e+21	OOM	3	OOM	OOM	< 1
3G21	4.59e+21	OOM	197,869	OOM	OOM	441
3JTZ	6.61e+22	OOM	5,074	OOM	OOM	8
3I2Z	4.61e+20	OOM	OOT	OOM	OOM	OOT
2RH2	1.29e+22	OOM	OOT	OOM	OOM	OOT
1IQZ	7.11e+17	18,337,117	90,195	1,824,235	40,217	117
2COV	1.14e+10	43,306	3	317	21	1
3FGV	6.44e+12	3,073,965	3	59,589	5,091	0
3DNJ	5.11e+12	569,597	4,984	7,469	570	3
2FHZ	1.83e+18	14,732,913	3,972	3,475,716	70,783	13

Table A1. The full comparison result between A*-based and AOBB-based search algorithms on the core redesign problem