A. Implementation details

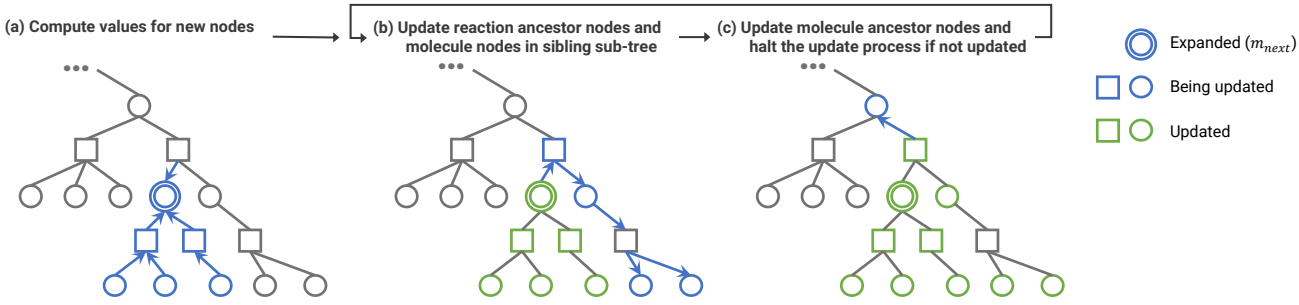

Figure 5. Illustration for the update process. Three phases correspond to line 1-8, line 11-16, and line 17-21 in Algorithm 2.

In this section we describe the algorithm details in the update phase of Retro*. The goal of the update phase is to compute the up-to-date $V_t(m|T)$ for every molecule node $m \in \mathcal{F}(T)$. To implement efficient update, we need to cache $V_t(m|T)$ for all $m \in \mathcal{V}^m(T)$. Note that from Eq (8), we can observe the fact that sibling molecule nodes have the same $V_t(m|T)$, *i.e.* $V_t(m_a|T) = V_t(m_b|T)$ if $pr(m_a|T) = pr(m_b|T)$. Therefore instead of storing the value of $V_t(m|T)$ in every molecule node m, we store the value in their common parent via defining $V_t(R|T) = V_t(m|T)$ if $R = pr(m|T)$ for every reaction node $R \in \mathcal{V}^r(T)$.

In our implementation, we cache $V_t(R|T)$ for all reaction nodes $R \in \mathcal{V}^r(T)$ and cache $rn(v|T)$ for all nodes $v \in \mathcal{V}(T)$. Caching values in this way would allow us to visit each related node only once for minimal update.

Algorithm 2: `Update`$(m_{next}, \{R_i, \mathcal{S}_i, c(R_i)\}_{i=1}^k)$ [5]

1 **for** $i \leftarrow 1$ **to** k **do**
2 **for** $m \in \mathcal{S}_i$ **do**
3 $rn(m) \leftarrow V_m$;
4 $rn(R_i) \leftarrow c(R_i) + \sum_{m \in \mathcal{S}_i} rn(m)$;
5 $V_t(R_i) \leftarrow V_t(pr(m_{next})) - rn(m_{next}) + rn(R_i)$;
6 $new\_rn \leftarrow \min_{i \in \{1,2,\cdots,k\}} rn(R_i)$;
7 $delta \leftarrow new\_rn - rn(m_{next})$;
8 $rn(m_{next}) \leftarrow new\_rn$;
9 $m_{current} \leftarrow m_{next}$;
10 **while** $delta \neq 0$ *and* $m_{current}$ *is not root* **do**
11 $R_{current} \leftarrow pr(m_{current})$;
12 $rn(R_{current}) \leftarrow rn(R_{current}) + delta$;
13 $V_t(R_{current}) \leftarrow V_t(R_{current}) + delta$;
14 **for** $m \in ch(R_{current})$ **do**
15 **if** m *is not* $m_{current}$ **then**
16 `UpdateSibling`$(m, delta)$;
17 $m_{current} \leftarrow pr(R_{current})$;
18 $delta = 0$;
19 **if** $rn(R_{current}) < rn(m_{current})$ **then**
20 $delta \leftarrow rn(R_{current}) - rn(m_{current})$;
21 $rn(m_{current}) \leftarrow rn(R_{current})$;

The update function is summarized in Algorithm 2 and illustrated in Figure 5, which takes in the expanded node m_{next} and the expansion result $\{R_i, \mathcal{S}_i, c(R_i)\}_{i=1}^k$, and performs updates to affected nodes. We first compute the values for new

[5] For clarity, we omit the condition on T in the notations.

reactions according to Eq (7) and (8) in line 1-8. Then we update the ancestor nodes of m_{next} in a bottom-up fashion in line 9-21. We also update the molecule nodes in the sibling sub-trees in line 16 and Algorithm 3.

Algorithm 3: `UpdateSibling`$(m, delta)$

```
1  rn(m|T) ← rn(m|T) + delta;
2  for R ∈ ch(m|T) do
3      for m' ∈ ch(R|T) do
4          UpdateSibling(m', delta);
```

Our implementation visits a node only when necessary. When updating along the ancestor path, it immediately stops when the influence of the expansion vanishes (line 10). When updating a single node, we use a $O(1)$ delta update by leveraging the relations derived from Eq (7) and (8), avoiding a direct computation which may require $O(k)$ or $O(\text{depth}(T))$ summations.

B. Guarantees on finding the optimal solution

Since Retro* is a variant of the A* algorithm, we can leverage existing results to prove the theoretical guarantees for Retro*. In this section, we first state the assumptions we make, and then prove the admissibility (Theorem 1) of Retro*.

The theoretical results in this paper build upon the assumption that we can access $\hat{V}_m$, which is a lowerbound for V_m for all molecules m. Note that this is a weak assumption, since we know 0 is a universal lowerbound for V_m.

As we describe in Eq (6), $V_t(m|T)$ can be decomposed into $g_t(m|T)$ and $h_t(m|T)$, where $g_t(m|T)$ is the exact cost of the partial route through m which is already in the tree, and $h_t(m|T)$ is the future costs for frontier nodes in the route which is a summation of a series of V_ms. In practice we use $\hat{V}_m$ in the summation, and arrive at $\hat{h}_t(m|T)$, which is a lowerbound of $h_t(m|T)$, *i.e.* the following lemma.

Lemma 2 *Assuming V_m or its lowerbound is known for all encountered molecules m, then the approximated future costs $\hat{h}_t(m|T)$ in Retro* is a lowerbound of true $h_t(m|T)$.*

We re-state the admissibility result (Theorem 1) in the main text and prove it with existing results in A* literature.

Theorem 1 ***(Admissibility)** Assuming V_m or its lowerbound is known for all encountered molecules m, Algorithm 1 is guaranteed to return an optimal solution, if the halting condition is changed to "the total costs of a found route is no larger than $\text{argmin}_{m \in \mathcal{F}(T)} V_t(m)$".*

Proof Combine Lemma 2 and Theorem 1 in the original A* paper (Hart et al., 1968). ■

C. Sample search trees and solution routes

In this section, we present two examples of the solution routes and the corresponding search trees for target molecule A and B produced by Retro*.

Solution route for target molecule A/B is illustrated in the top/bottom sub-figure of Figure 6, where a set of edges pointing from the same product molecule to reactant molecules represents an one-step chemical reaction. Molecules on the leaf nodes are all available.

The search trees for molecule A and B are illustrated in Figure 7 and Figure 8. We use reactangular boxes to represent molecules. Yellow/grey/blue boxes indicate available/unexpanded/solved molecules. Reactangular arrows are used to represent reactions. The numbers on the edges pointing from a molecule to a reaction are the probabilities produced by the one-step model. Due to space limit, we only present the minimal tree which leads to a solution.

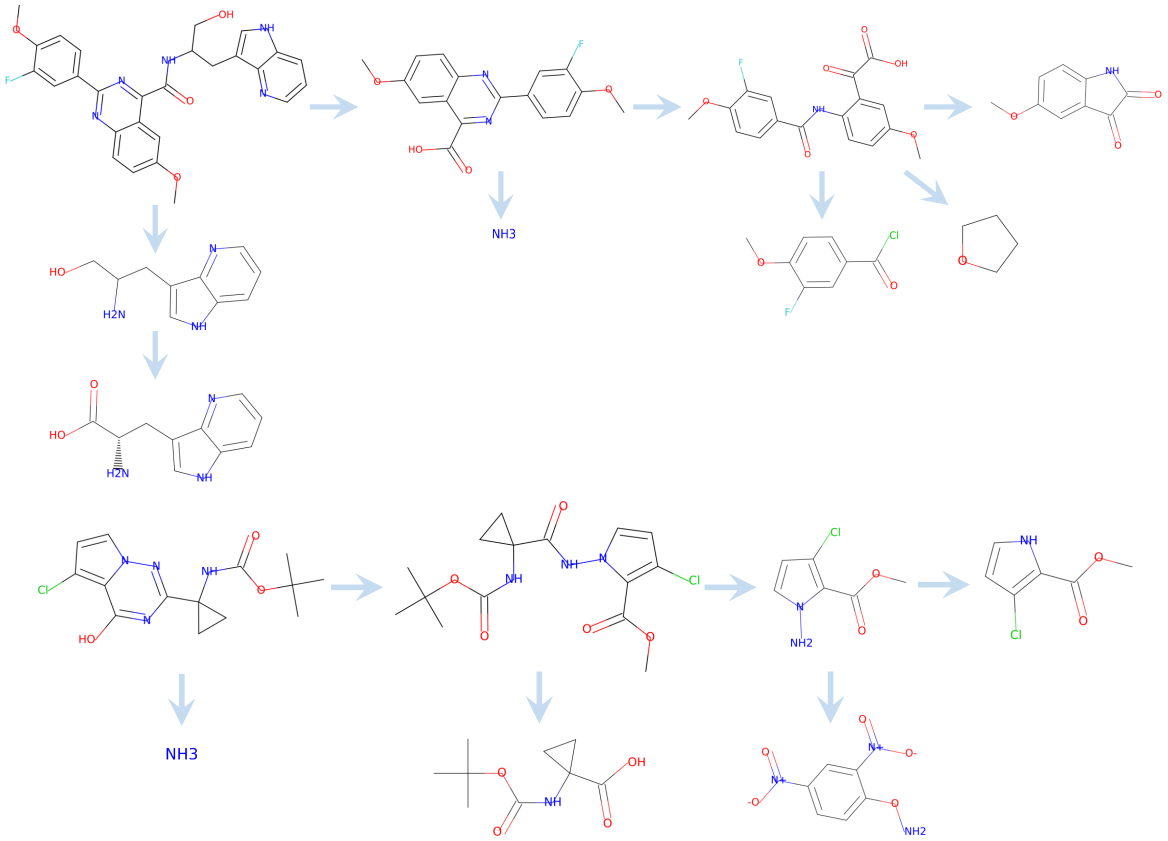

Figure 6. Top/bottom: solution route produced by Retro* for molecule A/B. Edges point from the same product molecule to the reactant molecules represent an one-step chemical reaction.

D. Retro* for hierarchical task planning

As a general planning algorithm, Retro* can be applied to other machine learning problems as well, including theorem proving (Yang & Deng, 2019) and hierarchical task planning (Erol, 1996) (or HTP), etc. Below, we conduct a synthetic experiment on HTP to demonstrate the idea. In the experiment, we are trying to search for a plan to complete a target task. The tasks (OR nodes) can be completed with different methods, and each method (AND nodes) requires a sequence of subtasks to be completed. Furthermore, each method is associated with a nonnegative cost. The goal is to find a plan with minimum total cost to realize the target task by decomposing it recursively until all the leaf task nodes represent primitive tasks that we know how to execute directly. As an example, to travel from home in city A to hotel in city B, we can take either `flight`, `train` or `ship`, each with its own cost. For each method, we have subtasks such as home $\rightarrow$ airport A, `flight`$(A \rightarrow B)$, and airport $B \rightarrow$ hotel. These subtasks can be further realized by several methods.

As usual, we want to find a plan with small cost in limited time which is measured by the number of expansions of task nodes. We use the optimal halting condition as stated in theorem 1. We compare our algorithms against DFPN-E, the best performing baseline. The results are summarized in Table 2 and 3.

Time Limit	15	20	25	30	35
Retro*	.67	.91	.96	.98	1.
Retro*-0	.50	.86	.95	.98	.99
DFPN-E	.02	.33	.74	.93	.97

Table 2. Success rate (higher is better) vs time limit.

As we can see, in terms of success rate, Retro* is slightly better than Retro*-0, and both of them are significantly better than DFPN-E. In terms of solution quality, we compute the approximation ratio (= solution cost / ground truth best solution cost)

Alg	Retro*	Retro*-0	DFPN-E
Avg. AR	1	1	1.5
Max. AR	1	1	3.9

Table 3. AR = Approximation ratio (lower is better), time limit=35.

for every solution, and verify the theoretical guarantee in theorem 1 on finding the best solution.

E. Related Works

Reinforcement learning algorithms (without planning) have also been considered for the retrosynthesis problem. Schreck et al. leverages self-play experience to fit a value function and uses policy iteration for learning an expansion policy. It is possible to combine it with a planning algorithm to achieve better performance in practice.

Learning to search from previous planning experiences has been well studied and applied to Go (Silver et al., 2016; 2017), Sokoban (Guez et al., 2018) and path planning (Chen et al., 2020). Existing methods cannot be directly applied to the retrosynthesis problem since the search space is more complicated, and the traditional representation where a node corresponds to a state is highly inefficient, as we mentioned in the discussion on MCTS in previous sections.

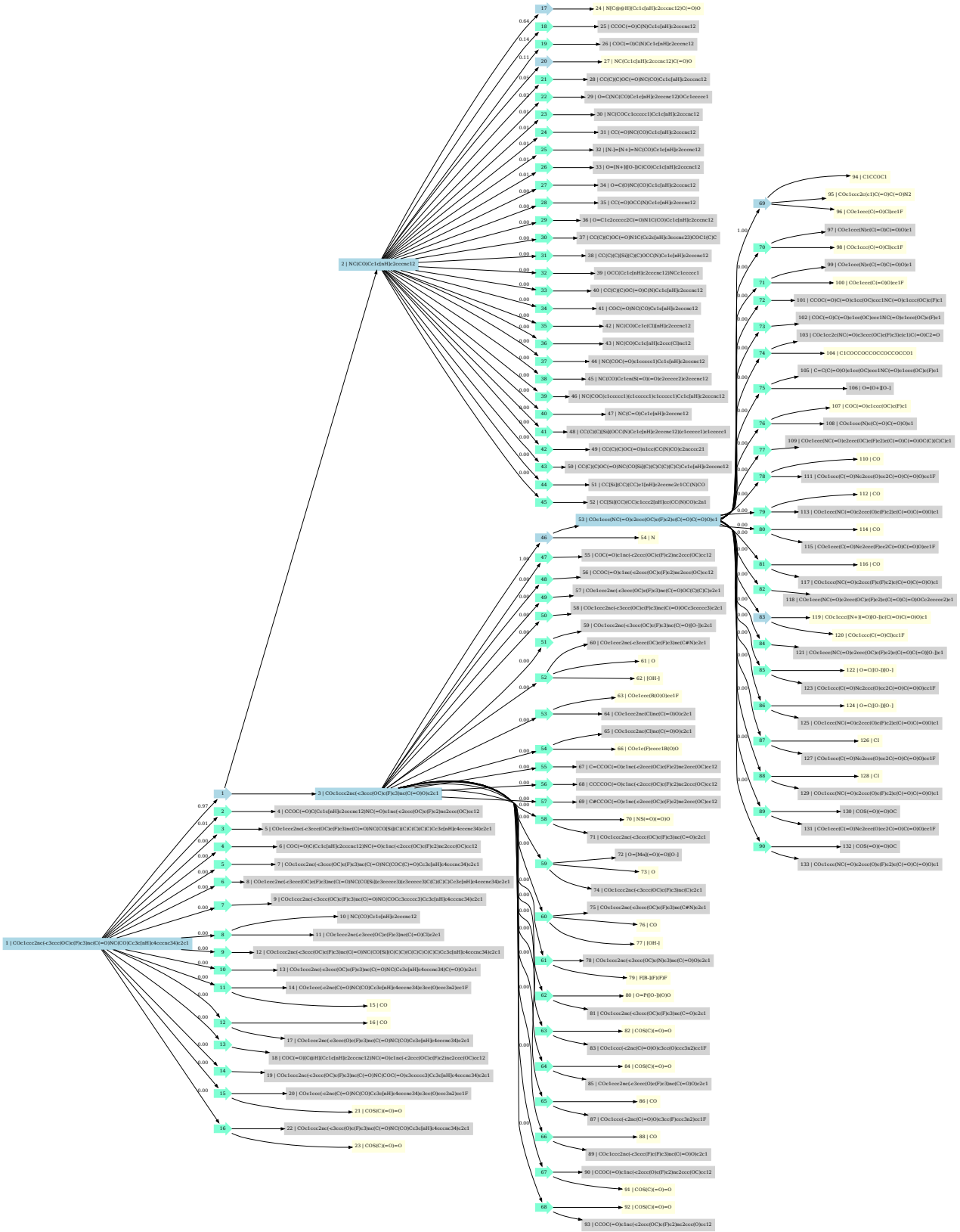

Figure 7. Search tree produced by Retro* for molecule A. Reactangular boxes/arrows represent molecules/reactions. Yellow/grey/blue indicate available/unexpanded/solved molecules. Numbers on the edges are the probabilities produced by the one-step model.

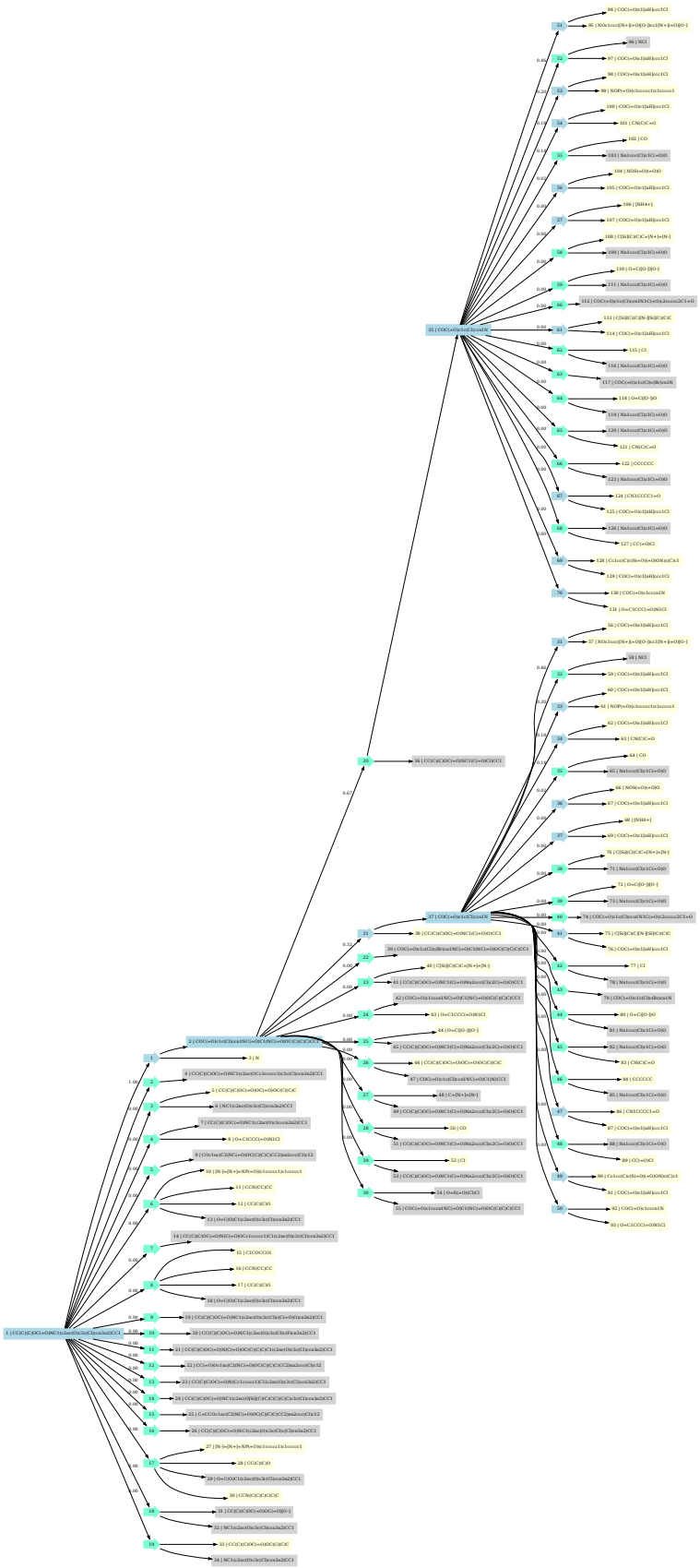

Figure 8. Search tree produced by Retro* for molecule B. Reactangular boxes/arrows represent molecules/reactions. Yellow/grey/blue indicate available/unexpanded/solved molecules. Numbers on the edges are the probabilities produced by the one-step model.