Supplementary Information for Manuscript titled "***Selection of cost-effective yet chemically diverse pathways from the networks of computer-generated retrosynthetic plans***" by Tomasz Badowski[1+], Karol Molga[1+], Bartosz A. Grzybowski[1,2]*

[1] Institute of Organic Chemistry, Polish Academy of Sciences, ul. Kasprzaka 44/52, Warsaw 01-224, Poland

[2] IBS Center for Soft and Living Matter and Department of Chemistry, UNIST, 50, UNIST-gil, Eonyang-eup, Ulju-gun, Ulsan, 689-798, South Korea

[+] Authors contributed equally

*Correspondence to: nanogrzybowski@gmail.com

Section S1. Description of the algorithms.

This section provides descriptions of the key aspects of our pathway selection algorithms. In **Section S1.1** we formalize the definitions of a chemical reaction network, of synthetic pathways within it, and the costs of such pathways. In **Section S1.2**, we discuss a procedure for restricting the initial retrosynthetic graph to its subgraph containing all syntheses of the target, and called a solutions' graph. In **Section S1.3**, we outline the algorithm for computing costs of nodes in the solutions' graph and in **Section S1.4**, the algorithm for finding the cheapest pathways in the solutions' graph and the path retrieving part of the algorithm for finding both cheap and diverse pathways. In **Section S1.5**, we focus on the penalization of reactions and on finding the nodes whose costs increase due to such penalization. Finally, in **Section S1.6** we discuss how such costs are recomputed.

S1.1. Definitions.

A chemical network is represented as a finite directed bipartite graph comprised of chemical nodes and reaction nodes. We assume that for each reaction node in the network there is exactly one edge from it to some chemical node, which is called the product of the reaction. Chemical nodes from which there are edges to a reaction node are called substrates of this reaction. We assume that each reaction in the network has at least one substrate. Some chemical nodes in the network are considered to be starting materials and are not products of any reactions in the network.

A synthetic pathway leading to the target chemical node is an acyclic subgraph of the network, containing the target, such that:

(i) each reaction in the pathway has all the same substrates and product as in the network,

(ii) each chemical node in the pathway which is not a starting material is a product of exactly one reaction in the pathway,

(iii) the target is not a substrate of any reaction in the pathway and the remaining chemical nodes in the pathway are substrates of at least one reaction in it.

We define the cost of a synthetic pathway (per millimole of its target) as follows. If the pathway consists of a single node (which is a starting material), we assume its cost to be given and equal to the cost of a millimole of this starting material. Otherwise, the pathway's cost is defined recursively as:

$$cost(S) = fixed\_cost(r) + \sum_{c \in pred(r)} a_{r,c}\, cost\big(subpath(S,c)\big),$$

where r is the only reaction in S producing the target, $fixed\_cost(r)$ is a nonnegative fixed cost of the reaction as discussed in the main text, $pred(r)$ is the set of predecessors of r in S (i.e., it substrates), $subpath(S,c)$ is the only (sub)pathway in S having c as its target, and $a_{r,c} \geq 1$ are some coefficients denoting the number of millimoles of substrate c needed to synthesize one millimole of the product of reaction r. As discussed in the main text, we implemented our algorithms for the special case of $a_{r,c} = 1/yield$, for $yield \in (0,1]$ denoting the average/global yield, though our algorithms also apply to other definition of $a_{r,c}$. The computation of costs of subpathways of S according to the above formula takes place in the topological order of their targets, i.e., the cost of a chemical node being pathway's target is computed only after such costs for all its predecessor chemical nodes in S being the targets have been computed. The assumption $a_{r,c} \geq 1$ ensures that our Dijkstra-like algorithms for computing costs of nodes in the network work, as discussed in **Section S1.3** (see also [26] and references therein for a related definition of a synthesis plan and its cost).

For coefficients $a_{r,c}$ equal to 1 (yield equal to 100% for $a_{r,c} = 1/yield$) and costs of starting materials equal zero, the cost of a pathway is equal to the sum of $fixed\_cost(r)$ over all reaction nodes r in the pathway if the pathway is a tree (i.e., a directed tree rooted in the target) or at least if the subgraph of the pathway induced by its nodes which are not starting materials is a tree. If such a subgraph is not a tree, then $fixed\_cost(r)$ for some reactions r

will appear several times in a sum obtained by unfolding the above recursive formula for $cost(S)$.

S1.2. Restriction of the retrosynthetic graph to its solutions' subgraph.

A chemical node c in a reaction network G is called synthesizable if there exists a synthetic pathway in G of which c is a target. A reaction r in G is called viable if all substrates of r are synthesizable, or equivalently if in G there exists a synthesis pathway of r's product containing r. An algorithm described by pseudocode in **Figure S1** updates the set of previously found synthesizable chemical and viable reaction nodes (together called synthesizable nodes) considering newly discovered synthesizable chemicals. The algorithm performs a DFS-like search of the reaction network, beginning with the newly discovered synthesizable chemicals and using the definition of a reaction being viable and the fact that a chemical node which is not a starting material is synthesizable only if it is product of some viable reaction. This algorithm is similar to the one for finding nodes B-connected to a source node in a hypergraph from [24] (if one identifies reactions with hyperarcs in such hypergraphs analogously as in [26,S1]).

To find all synthesizable nodes in the network, it is sufficient to run the procedure from **Figure S1** with the argument *synthFound* (denoting the initial set of found synthesizable nodes) being an empty set, *newSynthChems* (i.e., the list of newly discovered synthesizable chemical nodes) consisting of all the starting materials in the network, and with the dictionary *numNonsynthSubs* (mapping reaction nodes to the numbers of their substrates not in *synthFound*) initialized to store the total numbers of substrates for each reaction in the network. After the procedure finishes, *synthFound* consists of all the synthesizable nodes in the network.

```
1: procedure UPDATESYNTHFOUND(G, synthFound, newSynthChems, numNonsynthSubs)
   ▷ Updates the set synthFound of synthesizable nodes found considering
   newly discovered synthesizable chemical nodes newSynthChems.
   ▷ Arguments:
   ▷ G: reaction network
   ▷ synthFound: set of previously found synthesizable nodes
   ▷ newSynthChems: list of newly found synthesizable chemical nodes
   ▷ numNonsynthSubs: dictionary mapping reactions to numbers of their
   substrates not in synthFound
2:     Add nodes from newSynthChems to synthFound.
3:     while newSynthChems is nonempty do
4:         chem ← newSynthChems.pop()
5:         for rx ∈ G.successors(chem) do
6:             numNonsynthSubs[rx] ← numNonsynthSubs[rx] − 1
7:             if numNonsynthSubs[rx] = 0 then
8:                 synthFound.add(rx)
9:                 product ← G.successor(rx)
10:                if product ∉ synthFound then
11:                    synthFound.add(product)
12:                    newSynthChems.add(product)
```

Figure S1. Pseudocode of an algorithm for updating the set of synthesizable nodes in a reaction network. Triangles denote comments. The method *pop()* removes and returns the last element from a list. The method *succesors* of a graph returns the set of successor reactions of a chemical node given as the argument (i.e., reactions of which it is a substrate), while the method *successor* returns the single successor product of a given reaction node. The method *add* adds the element given as its argument to a set or a list. For a dictionary *d*, *d[x]* denotes the value stored in *d* corresponding to a key *x*.

We note that if the graph grows with the progress of retrosynthetic searches (cf. main text), it is more effective to update the set of synthesizable nodes each time a new reaction and its substrates are added to the reaction network. In this way, one can immediately find out when the target becomes synthesizable and one can proceed with selecting its synthetic pathways. Such an update of the set of found synthesizable nodes *synthFound* can be realized as follows. After a reaction *rx* and its substrates are added to the network, *rx's* substrates which are new starting materials are added to *synthFound* and information about the number of *rx*'s substrates not in *synthFound* is recorded in the dictionary *numNonsynthSubs*. Next, if

all the *rx*'s substrates are in *synthFound* (i.e., *rx* is viable), *rx* is also added to *synthFound*, and if further the *rx*'s product is not in *synthFound,* the procedure from **Figure S1** is run with the list *newSynthChems* comprising only this product.

If the target belongs to the set of synthesizable nodes found, we further proceed as follows. We find the set of ancestors of the target in the subgraph of the original retrosynthetic graph induced by its synthesizable nodes. We do this without actually computing such a subgraph – we perform a DFS-like search of the original network, starting from the target and exploring nodes which are yet undiscovered synthesizable ancestors of the already visited nodes. Once such a set of ancestors is determined, we compute a subgraph of the original network induced by such ancestors and the target. Note that the resulting subgraph is a reaction network containing all the synthesis pathways of the target present in the original network G (this follows from the fact that all nodes of every synthetic pathway of the target in a reaction network are synthesizable and are either ancestors of the target or the target itself). Thus, we call such a subgraph a solutions' graph. Because the solutions' graph is not larger (and typically much smaller) than the original graph, it uses significantly less memory (e.g., when saved on a disk). It is also not more computationally expensive (and typically cheaper) to perform computation of initial costs on it (discussed **Section S1.3**) or to find nodes whose costs are affected by penalization and to recompute these costs (discussed in **Sections S1.5** and **S1.6**).

Assuming that the number of substrates of each reaction (i.e. its in-degree) in the network G is bounded by a given constant (e.g., in our numerical experiments, all reactions had no more than four substrates), and that the set *synthFound* and dictionary *numNonsynthSubs* are implemented using hash tables, our algorithms for computing the solutions' graph discussed in this section run in time $O(number\ of\ nodes\ in\ G)$.

S1.3. Computing the initial costs in solutions' graph.

The cost of a chemical node in a reaction network is defined as the cost of its lowest-cost synthetic pathway in the network, while the cost of a reaction node in a network is defined as the cost of the cheapest synthetic pathway of the reaction's product and containing this reaction. Such costs fulfill the following generalized Bellman's equations[24]: for each chemical node c in the network which is not a starting material we have

$$cost(c) = min_{r \in pred(c)}(cost(r))$$

and for each reaction r in the network

$$cost(r) = fixed\_cost(r) + \sum_{c \in pred(r)} a_{r,c}\, cost(c).$$

We compute the costs of all the nodes in the network which are not starting materials using a Dijkstra-like algorithm. The algorithm is similar to the one for finding minimum weight B-paths in weighted hypergraphs described in [24] (see also [26]) for a binary heap used as a priority queue. This algorithm can be used in our case because the fixed costs of reactions we consider are nonnegative and, because $a_{r,c} \geq 1$, the so-called gain-free condition (which guarantees that cycles in the network are nondecreasing) is satisfied[24]. One difference between our algorithm and the one from [24] is that rather than starting from a single source node, our algorithm begins with computing the costs of reactions whose all substrates are starting materials, and pushing the minimum cost of their products and such products themselves onto the priority queue. Assuming boundedness of reactions' in-degrees (i.e., number of substrates in each reaction, see above), our algorithm for computing the costs runs it time $O(nlog(n))$ for n denoting the number of nodes in the solutions' graph.

S1.4. Finding the pathways.

We shall first discuss our algorithm for finding a desired number of the target's minimal-cost syntheses and then the algorithm for finding both economical and diverse routes. The number of pathways found by each of these algorithms is equal to the minimum of a user specified positive integer *k* and the number of all synthesis pathways of the target that exist in the solution's graph *G*.

Our algorithm for finding the minimal-cost pathways constructs them recursively starting from the target. The function used for expanding a pathway selects a reaction's product provided as the function's argument and calls itself recursively for each substrate of this reaction which is not a starting material, given as an argument. During the pathway's construction, the function maintains a set of argument products with which it was called on the recursively processed path from the target (i.e. the argument product is added to this set at the beginning of the function and removed at its end) and a dictionary mapping products with which it was called to the selected reactions yielding such products. The algorithm maintains a directed graph, called a sequence graph, whose nodes are unique integer identifiers representing different sequences of reactions that can be consecutively selected during the recursive construction of pathways. A node n_1 in the sequence graph has an edge to node n_2, only if n_2 is an identifier of a sequence of reactions represented by n_1 followed by a reaction that can be chosen next by the recursive function. For each identifier in the sequence graph,

the last reaction in the corresponding sequence is remembered in a dictionary. The algorithm also keeps a priority queue (implemented using a binary heap) containing sequence identifiers. The score of a sequence identifier in the queue is the minimum possible cost of pathways comprising the reactions from the corresponding sequence. The queue is initialized to contain an identifier of an empty sequence of reactions with a score equal to the cost of the target (computed as discussed in **Section S1.3**).

The algorithm keeps performing the following procedure in a loop until k pathways are returned or the priority queue becomes empty (meaning that all synthetic pathways of the target in G have been returned). First, it pops from the priority queue the sequence identifier with the lowest score v. It then retrieves all the ancestors of the sequence identifier from the sequence graph and, using them and the dictionary mapping identifiers to the last reactions in their sequences, it reconstructs a list of consecutive reactions to make during pathway's construction. Then, the abovementioned recursive function is called with the target and the list of reactions given as arguments. The function tries to construct a pathway containing reactions from the list and with cost equal to v as follows. If the list of reactions is nonempty, the function pops a next reaction to perform from the list. Otherwise, it proceeds as follows. It selects a lowest-cost reaction r_{min} in G producing the argument product p. The further operations made by the function depend on whether it is called with p provided as its argument for the first time during the pathway's construction. If this is the case, then the function finds reactions in G producing p which do not create a cycle (we say that a reaction r creates a cycle if there is a cycle in the subgraph induced by reactions selected so far by the function, r, as well as substrates and products of these reactions). Reactions creating a cycle cannot be chosen during pathway expansion, since, by definition, synthetic pathways cannot contain cycles. To verify if a reaction r creates a cycle, the function checks if any of r's substrates is present in the maintained set of products from the recursively processed path from the target to p. The function adds to the sequence graph new identifiers representing the sequences of reactions selected so far followed by each of the found reactions producing p that does not create a cycle. Each such new identifier corresponding to some last reaction r other than the selected cheapest one r_{min} is also added to the priority queue. The score of such an identifier in the queue is computed as the popped identifier's score v plus a product of $cost(r) - cost(r_{min})$ and the product of coefficients $a_{s,c}$ over the reactions s and their substrates c encountered on the recursively processed path from the target to p (which for $a_{s,c} = 1/yield$ is equal to the inverse yield raised to the power of the number of reactions in such a path). If r_{min} creates a cycle, then the pathway expansion function is terminated without

a success and the algorithm starts expanding another pathway from the beginning (i.e. starting from popping a new identifier from the priority queue). In a situation when the function is called with the argument product p for the second or later time during the pathway's construction, it proceeds as follows. It checks if the selected reaction r_{min} is equal to the previously chosen one r for this product (using the maintained dictionary mapping products to the selected reactions producing them). If not, then the function adds to the sequence graph a new identifier corresponding to choosing r again and to the priority queue this identifier with a score computed identically as discussed above. Furthermore, the function is terminated without a success (by definition, synthesis pathways can contain only one reaction with a given product), and the algorithm starts expanding another pathway. If, on the other hand, $r = r_{min}$, then the function adds an identifier corresponding to choosing this reaction to the sequence graph. When the function called with the target as an argument finishes successfully, a pathway comprised of the selected reactions is returned.

The algorithm for finding economically feasible and diverse pathways performs the following steps in a loop until it stops. It runs a procedure like the one above for finding the lowest-cost pathways until a pathway that was not returned yet is retrieved or the procedure discovers that there are no more synthetic pathways left in G (i.e. the priority queue becomes empty). If such a new pathway is found, it is returned. When the k pathways requested by the user are returned or the procedure discovers that there are no more pathways left in G, the algorithm stops. Otherwise, it penalizes appropriate reactions in the solutions' graph and recomputes the costs of nodes affected by such a penalization as discussed in **Sections S1.5** and **S1.6**. In our implementation, different runs of the procedure for finding lowest-cost pathways in the above loop reuse the same sequence graph (but, of course, the priority queue is reinitialized each time at the beginning of the procedure). Note that in this algorithm, our procedure for finding a sequence of lowest-cost pathways could be replaced by an alternative one (see, e.g., [21] and its discussion in [26]).

S1.5. Penalization of reactions and identification of nodes whose costs increase due to such penalization.

To promote finding diverse pathways, we add a penalty $p > 0$ to (i) fixed costs of reactions from the previously found pathway and (ii) fixed costs of other, similar reactions in the network. We consider a reaction s to be similar to reaction r if s has the same product as r and at least one of the substrates of s belongs to the set of main substrates of r (main substrates are those with at least four carbon atoms or the largest number of carbon atoms).

Let *cost* denote the cost function defined as in **Section S1.3** before penalization of fixed costs of reactions, and *cost'* – after the penalization. We are interested in finding the set S_{incr} of nodes n in the solutions' graph G for which $cost(n) < cost'(n)$, i.e., whose costs increase due to penalization. From the generalized Bellman's equations in **Section S1.3**, S_{incr} satisfies the following two conditions (for S replaced by S_{incr}).

Condition 1. A reaction node r (from G) belongs to S only if it is one of the penalized reactions or some of r's substrates belong to S.

Condition 2. A chemical node c belongs to S only if all reactions r producing c and such that $cost(r) = cost(c)$ form a nonempty subset of S.

The algorithm described by the pseudocode in **Figure S2** finds and returns the smallest set S_{found} satisfying the above two conditions, i.e., such that for any other S satisfying them, we must have $S_{found} \subset S$. In particular, we have $S_{found} \subset S_{incr}$, i.e., all the nodes found by this algorithm incrase costs due to penalization.

```
1: function GETNODESINCREASINGCOST(G, penalizedRxs, cost)
   ▷ Returns a set of nodes in G whose costs increase due to penalization.
   ▷ Arguments:
   ▷ G: solutions' graph
   ▷ penalizedRxs: list of penalized reactions
   ▷ cost: dictionary mapping nodes in G to their costs before penalization
   ▷ set of found nodes in G whose costs increase due to penalization
2:     nodesIncrCost ← set(penalizedRxs)
   ▷ list of reactions whose descendants need to be checked for increasing costs
3:     rxsCheckDescIncr ← penalizedRxs
   ▷ dictionary mapping chemical nodes to the number of their cheapest predecessor reactions which are not known to increase cost
4:     numCheapestRxs ← empty dictionary
5:     while rxsCheckDescIncr is nonempty do
6:         rx ← rxsCheckDescIncr.pop()
7:         product ← G.successor(rx)
8:         if product ∉ nodesIncrCost then
   ▷ if rx is the cheapest predecessor reaction of product
9:             if cost[rx] = cost[product] then
10:                if product ∉ numCheapestRxs then
11:                    numCheapestRxs[product] ← number of predecessor reactions of product with cost equal cost[product]
12:                numCheapestRxs[product] ← numCheapestRxs[product] − 1
   ▷ if all the cheapest reactions leading to product are known to increase cost
13:                if numCheapestRxs[product] = 0 then
14:                    nodesIncrCost.add(product)
15:                    for rx ∈ G.successors(product) do
16:                        if rx ∉ nodesIncrCost then
17:                            nodesIncrCost.add(rx)
18:                            rxsCheckDescIncr.add(rx)
19:    return nodesIncrCost
```

Figure S2. Pseudocode of an algorithm identifying nodes of the solutions' graph whose costs increase due to penalization. For a list *l*, *set(l)* creates and returns a set with the same elements as *l*. The remaining notations used are the same as in **Figure S1**.

We will show that under the additional **Assumption 1** below, we also have $S_{incr} = S_{found}$, i.e., this algorithm returns exactly the nodes whose costs increase due to penalization.

Assumption 1: For each reaction r in G and its substrate c, $cost(c) < cost(r)$. This assumption holds, e.g., if all the fixed costs of reactions are positive or if the costs of

starting materials are positive and $a_{r,c}$>1 (which for $a_{r,c} = 1/yield$ is equivalent to $yield <$ 100%).

Let us now make **Assumption 1**. We will show that $S_{incr} \setminus S_{found}$is empty, which (along with $S_{found} \subset S_{incr}$) implies that $S_{incr} = S_{found}$. Let *C* be the set of nodes in $S_{incr} \setminus S_{found}$ with minimum value of *cost*, i.e., for

$$m = \min\big(cost(n): n \in S_{incr} \setminus S_{found}\big),$$

we have

$$C = \big\{n \in S_{incr} \setminus S_{found}: cost(n) = m\big\}.$$

We will show that *C* is empty, which will imply that $S_{incr} \setminus S_{found}$ is empty.

Assume, aiming at a contradiction, that for some reaction node *r,* $r \in C$. Then, due to **Condition 1** for $S = S_{incr}$ (and the fact that *r* cannot be penalized since $r \notin S_{found}$), some substrate *c* of *r* must belong to S_{incr}. From **Assumption 1**, for this substrate it holds that $cost(c) < cost(r)$. We must have $c \in S_{found}$ as otherwise it would hold $c \in S_{incr} \setminus S_{found}$ and $cost(c) < cost(r) = m = \min\big(cost(n): n \in S_{incr} \setminus S_{found}\big)$, which is impossible. Thus, from **Condition 1** for $S = S_{found}$, we must also have $r \in S_{found}$. We received a contradiction with $r \in C \subset S_{incr} \setminus S_{found}$. Thus, *C* cannot contain any reaction nodes.

Assume now, again aiming for a contradiction, that for some chemical node *c,* $c \in C$. Then, from **Condition 2** for $S = S_{incr}$, all reactions *r* of which *c* is a product and for which $cost(r) = cost(c)$, fulfill $r \in S_{incr}$. For such reactions we must have $r \in S_{found}$, as otherwise we would have $r \in S_{incr} \setminus S_{found}$ and from $cost(r) = cost(c) = m$, it would hold $r \in C$, which we just proved to be impossible. Therefore, from **Condition 2** for $S = S_{found}$, we have $c \in S_{found}$, which is in contradiction with $c \in C \subset S_{incr} \setminus S_{found}$. Thus, *C* also cannot contain any chemical nodes, i.e. it is indeed empty.

Assuming the boundedness of reactions' in-degrees, and that dictionaries and sets used in the algorithm described by pseudocode in **Figure S2** are implemented using hash tables, this algorithm runs it time $O(number\ of\ nodes\ in\ solutions'graph)$.

We note that if **Assumption 1** does not hold, then, instead of finding nodes according to the above algorithm and recomputing their costs as discussed in **Section S1.6**, one can recompute the cost of all nodes which are not starting materials from scratch as discussed in **Section S1.3**.

S1.6. Recomputing the costs which increase due to penalization.

To recompute the costs of nodes whose costs increase (due to penalization), we use a Dijkstra-like algorithm similar to the one described in **S1.4**. The algorithm starts with computing the new costs (i.e. after penalization) of penalized reactions whose substrates do not increase their costs. Then, it finds chemical nodes whose cost increases and which are products of at least one reaction with known new cost and pushes the minimum new costs of such products and the products themselves onto the priority queue.

Section S2. Performance experiments.

Our algorithms were implemented in Python (without any parallelization) and run on a computer with AMD Opteron 6380 processors with 2.5 GHz clockspeed. In all performance tests, 80% yield was used. For clofedanol and NAr_3, fixed reaction cost \$1/mmol was used, while for AMG641 it was set to \$20/mmol. For reactions requiring protections, additional penalty (AMG641: \$40/mmol; NAr_3: \$1/mmol; Clofedanol: \$2/mmol) was added. For each molecule, we saved retrosynthesis graphs of various sizes from a single search and ran the full path selection algorithm on them either with (i) no diversity penalty or (ii) with such penalty equal to 10,000. CPU times of various stages of the algorithm and of finding the consecutive pathways were recorded. In **Figure 5** in the main text, only the times t_n of computing the cost of solutions' graph and finding first n pathways were considered for graphs which contained at least 100 different synthesis pathways of the target.

The table below summarizes information about CPU times of all the stages of the full algorithm for selecting 100 pathways from the largest retrosynthetic graphs for each molecule. The time to find synthesizable chemical nodes and viable reactions in the graph (using the procedure from **Figure S1** with *newSynthChems* consisting of starting materials as discussed in **Section S1.2**, which could be executed after the retrosynthetic graph was constructed as opposed to the alternative updating approach) is denoted as t_{synth}; the time to find the ancestors of the target in the subgraph induced by synthesizable nodes is $t_{ancestors}$; the time to restrict the retrosynthetic graph to the solutions' subgraph induced by the target and such ancestors as $t_{subgraph}$; and the time to compute the initial costs in the solutions' graph as t_{icost}. Since these parts are identical with and without diversity penalties applied, the table lists averages of CPU times for both of these scenarios. The time of finding paths is denoted as t_{paths} and of penalizing reactions, finding the nodes changing costs, and recomputing their costs as t_{rcost} (for p equal to zero such operations are not performed and thus their CPU time is zero). The sum of CPU times of all stages is denoted as t_{total}.

Molecule	p	t_{synth} [s]	$t_{ancestors}$ [s]	$t_{subgraph}$ [s]	t_{icost} [s]	t_{paths} [s]	t_{rcost} [s]	t_{total}
Clofedanol	0	0,033	0,032	0,088	0,113	0,045	0	0,311
	10000					0,081	0,289	0,635
AMG641	0	0,014	0,015	0,038	0,047	0,022	0	0,136
	10000					0,073	0,159	0,346
NAr_3	0	0,007	0,008	0,019	0,028	0,024	0	0,086
	10000					0,068	0,216	0,345

As seen in the Table, the total CPU time with $p = 0$ is less than 0.32 sec and less than 0.65 sec for p = 10,000. Note that t_{rcost} for experiments in which nonzero penalties were used was in all cases much smaller than $99 \cdot t_{icost}$, which demonstrates that finding and recomputing only the costs of nodes increasing due to penalization is much faster than recomputing the costs of all nodes from scratch.

Section S3. Differences with prior approaches.

S3.1. Comparison with Chematica's early path-selection algorithms. Previous versions of Chematica included a rudimentary path-selection algorithm described briefly in the SI Section S6.3 of our 2018 *Chem* publication [20]. This prior method differed from the current one in the several important ways – both in terms of unrealistic chemical assumptions and also much less efficient algorithms, together translating into chemically sub-optimal solutions being found and into painfully long path retrieval times. These differences are detailed below:

S.3.1.1. Chemical differences. Previous version of the algorithm used a different, less realistic definition of cost of a synthetic pathway. The cost of a pathway was based on the grams of starting materials rather than millimoles and, more importantly, did not take into account **reaction yields**, and it was assumed that each step produces one gram of the product from one gram of each of reaction's substrates:

$$cost(S) = fixed\_cost(r) + \sum_{c \in pred(r)} cost\big(subpath(S, c)\big),$$

This formulation translated into unrealistic cost estimates – for instance, a ten step linear pathway would score on par with a convergent 5+5 synthesis starting from the same number of similarly priced materials, although it is evident that in practice, the latter route is significantly more economical. The new implementation, taking into accounts yields and per-millimole conversions is much more chemical and can discriminate between such cases (see also main-text Figure 2).

Next, the penalties assigned to avoid repetition of **similar reactions** are now improved to select really diverse pathways. As detailed in Section **S1.5**, we penalize reactions that were already present in the previously found pathways and also those that use similar reactions. We consider a reaction s to be similar to reaction r if s has the same product as r and *at least one* of the substrates of s belongs to the set of main substrates of r (main substrates are those with at least four carbon atoms or the largest number of carbon atoms). In contrast, in the previous version of the algorithm, reaction *s* was considered to be similar to reaction *r* if it had the same product and the substrate with the highest number of carbon atoms (for several substrates having the same, largest number of carbon atoms, the one with the lexicographically longest SMILES string[S2] was considered). This condition for similarity was narrower in scope than the new one and, consequently, resulted in smaller set of reactions being penalized. For example, analogous steps marked in grey in Figure 9a and blue in Figure

9b in the main text are similar according to the new definition, but not according to the previous one.

S.3.1.2. Algorithmic differences. Our new selection algorithms are much more time and memory efficient. In particular, for realistic networks of solutions, they now execute in a fraction of a second vs. thousands of seconds in the previous version of Chematica. To achieve these improvements, most of algorithm's routines have been thoroughly changed; some of the key changes are in the modules responsible for:

(i) Updating synthesizable nodes. A more efficient algorithm for updating the set of synthesizable nodes (discussed in Section **S1.2**) is now implemented. Notably, to verify if a reaction is viable, we now check if the number of its substrates not yet found to be synthesizable, maintained in a dictionary *numNonsynthSubs,* is equal to zero (as in line 7 of Figure S1). Before, this was achieved by iteration over all of reaction's substrates and checking if they are synthesizable.

(ii) Extraction of the solution's graphs from the entire network of nodes visited during retrosynthetic searches. In the previous version of the algorithm for finding the solutions' graph, the subgraph of the original retrosynthetic network induced by synthesizable nodes was computed before the ancestors of the target in this subgraph were found. As discussed in Section **S1.2**, in the current version, such ancestors are found using a DFS-like search of the original network without the time-consuming computation of this subgraph.

(iii) Computing and re-computing of costs. Previously, to compute the costs in the solutions' graph, the algorithm began with finding a graph of strongly connected components of the solutions' graph. Then, such components were visited in the topological order and costs of nodes within each of these components were calculated using a Dijkstra-like algorithm. This approach was also used to recompute the costs of all nodes in the whole solutions' graph after penalization of the fixed costs of reactions.

In the new version, we compute initial costs of nodes in the solutions' graph using a Dijkstra-like algorithm (discussed in Section **S1.3**), which is not only faster than the previously used approach but also much simpler to implement. Furthermore, in the algorithm for finding diverse pathways, we find and recompute only the costs changing due to penalization, which is typically significantly faster than recomputing all costs from scratch (see sections **S1.5, S1.6,** and **S2**).

(iv) Retrieval of the minimal-cost and diverse pathways. In the old and new implementation, this algorithm tried to further expand parts of pathways corresponding to

different sequences of reactions chosen in the initial stage of pathway construction. The information about such sequences was stored in a list of tuples consisting of a list of nodes visited up to a given point during pathway expansion, the list of substrates to be expanded (consisting of unexpanded substrates of visited reactions which were not starting materials), the so-called "accumulated costs" (equal to sums of fixed costs of visited reactions and costs of visited starting materials), and the "total costs", equal to sums of the accumulated costs and the computed costs (in the solutions' graph) of substrates to be expanded. Such a list consumed much more memory and time to construct than the priority queue with sequence identifiers and scores (corresponding to the abovementioned total costs) as well as the sequence graph used to reconstruct the sequences of reactions from such identifiers, both of which are used in our new algorithm.

In the old algorithm, before the pathway expansion phase, the total and accumulated costs of all elements of the list of tuples were recomputed (using the information about the visited nodes and substrates to be expanded in the tuples) and the element with the minimum total cost (corresponding to the part of the pathway to be further expanded) was found in the list and removed from it. Such recomputing of total and accumulated costs was very time consuming but was needed in cases when the costs of visited reactions or substrates to be expanded changed as a result of penalization and recomputing of costs in the solutions' graph. Also, in the pathway expansion process, as long as the list of substrates to expand was nonempty, the algorithm proceeded as follows. A chemical node p was popped from this list and reactions from the solutions' graph producing this chemical that did not create a cycle were found (see Section **S1.4** for the definition of reactions creating a cycle) by computing subgraphs of the solutions' graph induced by visited nodes and reactions, and by checking if such subgraphs were directed acyclic graphs. Then, the cheapest reaction r_{min} producing p was found and the tuples corresponding to reactions not creating a cycle other than r_{min} were computed (using accumulated cost to compute their total costs) and added to the list of tuples. If r_{min} created a cycle, then pathway expansion was terminated and the algorithm moved on to expanding another pathway. Otherwise, the algorithm selected r_{min} as the next reaction during the pathway expansion and updated the accumulated cost, set of visited nodes, and substrates to expand. Once the list of substrates to expand became empty, the pathway corresponding to visited nodes was returned. Then, if fewer than the required number of pathways were returned, the algorithm penalized appropriate reactions and recomputed the costs in solutions' graph (as discussed above) and moved on to identify another pathway. Unlike our current algorithm, this old implementation did not have any mechanisms ensuring that the found

pathways had only one reaction with a given product. Thus, it sometimes returned as "pathway" graphs containing several reactions producing a given chemical.

In our new algorithm, there is no need to recompute the scores in the priority queue after the costs in solutions' graph change due to penalization – this is so because after the costs change, a new priority queue is constructed. The new algorithm is also much more efficient in checking if a reaction creates a cycle (see Section **S1.4**).

Finally, we note that with our implementation, we added the possibility of saving a solutions' graph *during* retrosynthetic search to later load it and select pathways from it multiple times under different scenarios (i.e., different costs of reactions, average yields, magnitudes of imposed diversity penalties). In the previous version of Chematica, only the diversity penalties could be changed during retrosynthetic search using the "select diverse" slider in the Chematica's main window. This affected the diversity of the next set of pathways selected from the continuously expanding retrosynthetic graph. The cost of reactions, however, was fixed and specified by the user *before* search – any change in this parameter required the user to restart the entire, slow retrosynthetic search.

S3.2. Comparison with other relevant works in the area.

In this Section, we narrate briefly other publications in which problems and algorithms related to our work have been addressed, albeit not in the context of chemically realistic retrosynthetic design or even not in the context of chemistry at all.

S.3.2.1. Differences from methods for finding the best K synthesis plans [26] and K shortest hyperpaths [21]. In reference [26], the authors reformulate the problem of finding the K lowest-cost synthesis plans in a reaction network in terms of the problem of finding K lowest-cost hyperpaths in a hypergraph. They also apply an algorithm from ref [21] (for the special case of the latter problem for acyclic hypergraphs) to find K synthesis plans with the lowest total weight of starting materials, assuming fixed reaction yields. Unfortunately, they consider a completely unrealistically simple mathematical model of a reaction network, in which the molecules are represented as carbon skeletons and reactions rely on forming bonds between arbitrary carbon atoms of different substrates to join them, or between the atoms of the same substrate to form rings. Even the authors themselves admit that real reactions can differ significantly from the ones in their model and the "skeleton plans" resulting from their model

may not correspond to any feasible syntheses. There is also no mention in their work of any selection based on synthetic diversity.

In contrast, we demonstrate that our algorithm is applicable to realistic, large reaction networks, possibly containing cycles, from which it can rapidly select chemically viable syntheses. In fact, the synthetic examples we provide are the first demonstration of computer-generated plans that are not only chemically correct but also scored realistically against (simultaneously!) prices of the starting materials, reaction operation costs, and yields, and selected according to synthetic diversity criteria.

Down to some more technical detail, we note that the authors of ref [26] mention – but do not demonstrate – that a more complicated version of the algorithm from [21] *could* also be applied to more general reaction networks, like the ones admitting cycles (though, as opposed to our work, they do not provide sufficient conditions for the algorithm to be applicable to such networks). They also suggest that this algorithm could be used with more general synthesis plan costs, having the recursive form of so called "additive weighting functions" (see [21]), e.g., allowing to consider fixed-reaction-costs and costs of consumed starting materials similar as in our work. Note, however, that even if implemented, the algorithm from [21] is expected to be much slower than the version of our algorithm for finding lowest-cost pathways (both run on solutions' graphs similar as in our performance experiments). To show this, consider the following argument. Recall that for a given solutions' graph, our algorithm for selecting a given number of the lowest-cost pathways first computes the initial cost of nodes in the graph (as discussed in Section **S1.3**), and then finds the pathways in it (see Section **S1.4**). Note also that, in all our performance experiments in Section **S2** for computing 100 lowest-cost pathways on the largest solutions' graphs, the time t_{icost} of computing the initial costs was higher than the time t_{paths} of finding all the pathways. The algorithm from [21] requires the computation of costs of nodes in a modified graph using a Dijkstra-like method from [24] (i.e. similar as in our work) at least once for each pathway found. This is the case both for the slower and the improved versions of this algorithm called, respectively, *Yen* and *LBYen* in [21]. Furthermore, the CPU time of both versions of the algorithm in numerical experiments in [21] was roughly proportional to the number of such cost computations made. Thus, even if the algorithm from [21] required only one computation of costs for each of the 100 pathways found in our solutions' graphs, it can still be expected to run much slower than our algorithm (i.e., at least 50 times slower).

S.3.2.2. Differences from methods for finding dissimilar paths in graphs.

There has been some work in the non-chemical literature on the problem of finding dissimilar but possibly short paths between a given origin and a destination in weighted graphs – for instance, in the context of finding spatially dissimilar paths in transportation networks [27,S3]. We note that this problem is significantly less general than considered in our work. First, a weighted directed graph can be identified only with a reaction network in which graph's nodes are represented by chemical nodes and its edges, by unary reactions with fixed costs equal to the edges' weights. Furthermore, for the network containing a single starting material whose cost is zero and for yield equal to 100%, synthetic pathways of a target in the network have cost equal to the length of the corresponding paths from origin to destination in the graph setting.

The algorithm for finding short but dissimilar pathways in graphs that is most related to the approach in our work is a so-called Iterative Penalty Method (IPM) (see [27] and references therein). It relies on the repetitive application of finding the shortest path (e.g., using the Dijkstra algorithm) and then adding penalties, e.g., to the edges from such a path.

An approach analogous to IPM in the context of our reaction networks could rely on repetitively computing the costs in the network (or efficiently re-computing only the changing costs), finding the lowest-cost pathway, and penalizing the fixed costs of reactions from this pathway. One of the differences between our algorithm and such an IPM analogue is that, after finding a pathway, we penalize not only the reactions from this pathway but also appropriately defined similar reactions (for example, pairs of analogous reactions marked in blue and grey in **Figure 9a** and **9b** in the main text are similar according to our definition though not identical). Another important difference is that our algorithm does not return the lowest-cost pathway in the graph with recomputed cost, but the lowest-cost pathway not returned before (using our method for generating the consecutive lowest-cost pathways until a new pathway is discovered). This ensures that our method cannot return the same pathway several times and that it returns all the existing pathways when their total number is not higher than the number of pathways requested by the user. The IPM-like algorithm, on the other hand, can return repeated pathways and may never return some existing pathways no matter for how many iterations it is run. An IPM version for finding K distinct diverse paths was used in ref [27] whereby, when a repeated path is found, the algorithm rejects it (but applies penalties to its edges) and goes to another iteration of the method**.** Note that a similar idea could be used in the IPM analogue for reaction networks. Unfortunately, such an algorithm

will never finish in the case when K is greater than the number of existing pathways in the network (which the user does not know a priori when specifying K) and even in some cases when there exist at least K distinct paths in the graph. To illustrate this, consider a simple reaction network in **Figure S3** below and assume that the fixed costs of all reactions, diversity penalty, and yield are all equal to 1, as well as that the cost of the only starting material is zero. This network contains three pathways: p_1 containing reactions r_1 and r_4 and with cost 2, p_2 with reactions r_2 and r_5 and cost also 2, and p_3 with reactions r_2, r_3, and r_4 and cost 3.

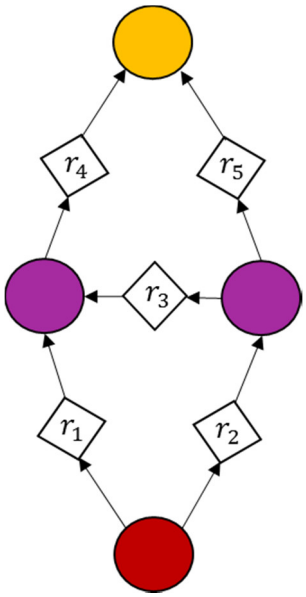

Figure S3. An example of an extremely simple reaction network used to compare our algorithm against an IPM-type approach. Red node is the starting material, violet nodes are intermediates, and the yellow dot is the target molecule.

For this network, both our and the IPM-like algorithm could first return pathway p_1 and then p_2. When queried for more pathways, our algorithm would next return pathway p_3 and then discover that there are no more pathways left in the network. The IPM analogue, on the other hand, would again return pathway p_1, then again p_2, and so on, never returning pathway p_3. Thus, if the technique of rejecting repeated pathways were used, when queried for three or more pathways, the IPM-like algorithm would get stuck in an infinite loop. The same problems can occur with the original IPM algorithm in the graph setting (e.g., the above example can be easily reformulated in the directed graph setting) and an approach similar to ours could be used to overcome them, i.e., instead of finding the shortest path in the penalized graph, one could generate a sequence of shortest paths (e.g., using Yen' algorithm[S4]) until a

new path is found or the algorithm discovers that there are no more paths left in the graph. However, to our knowledge, this has never been done in the literature.

Supplementary references.

S1 P. Carbonell, D. Fichera, S. B. Pandit and J.-L. Faulon, *BMC Syst. Biol.*, 2012, **6**, 10.

S2 D. Weininger, *J. Chem. Inf. Model.*, 1988, **28**, 31–36

S3 H. Liu, C. Jin, B. Yang and A. Zhou, *IEEE Trans. Knowl. Data Eng.*, 2018, **30**, 488–502.

S4 J. Y. Yen, *Manage. Sci.*, 1971, **17**, 712–716.

Section S4. Details of Chematica's syntheses of triarylamine.

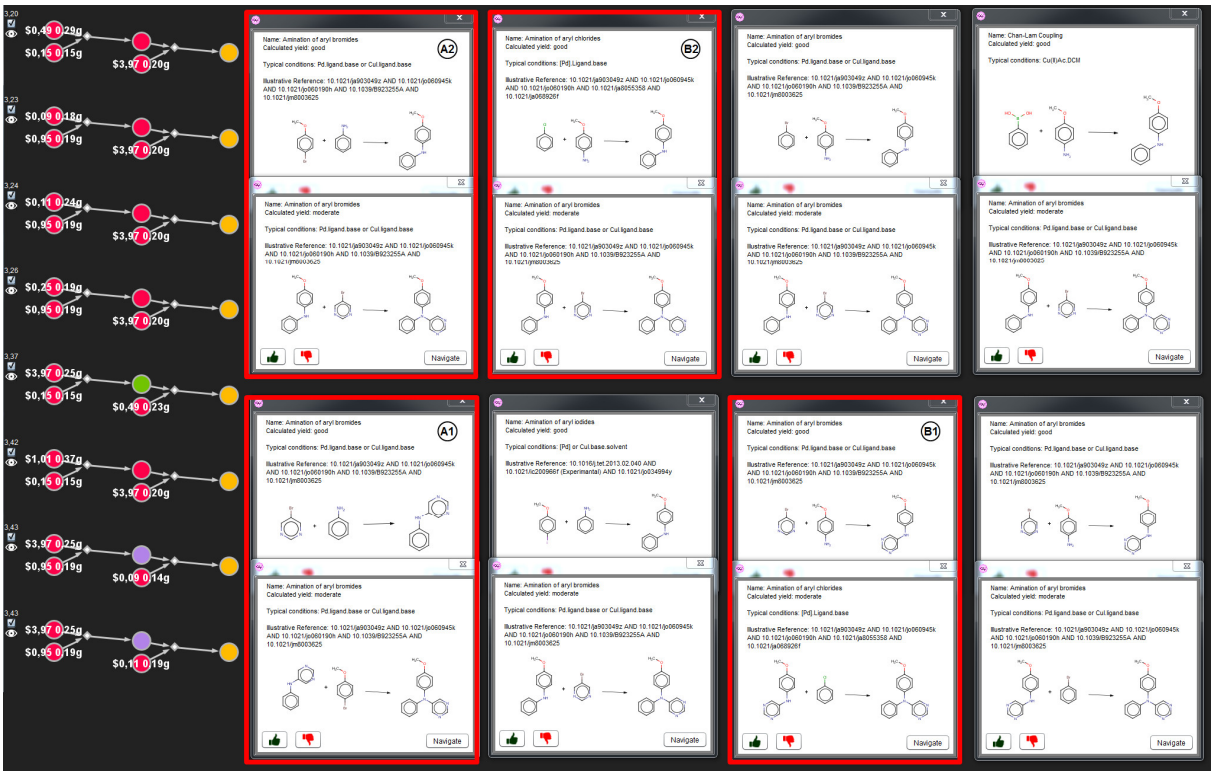

Figure S4. Details of top ten synthetic pathways obtained for triarylamine with *RxC* = \$1/mmol, *Y* = 80%. Pathways depicted in **Figure 6c,d** are marked with red frames.

Section S5. Details of Chematica's syntheses of Clofedanol.

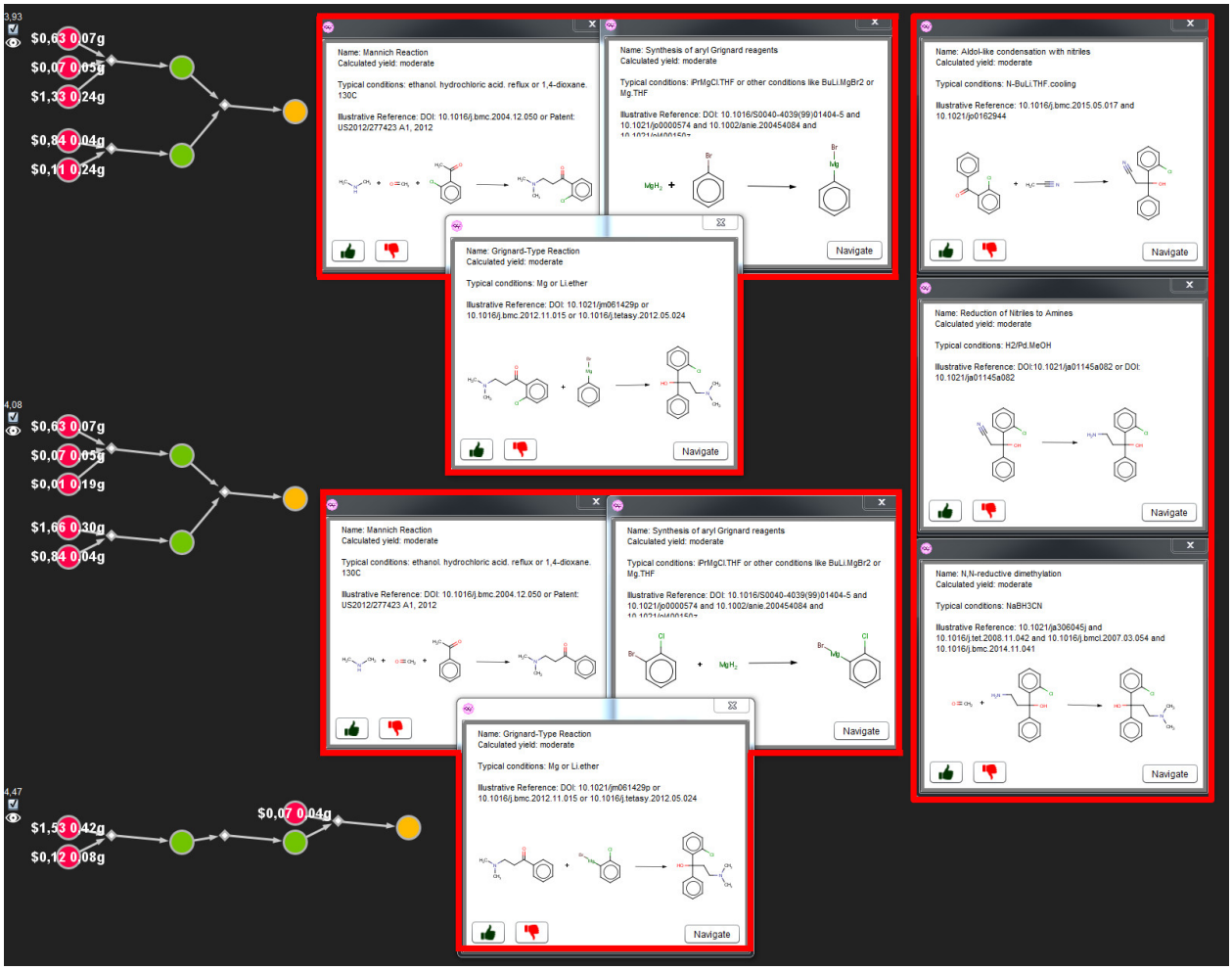

Figure S5. Details of top three synthetic pathways obtained for clofedanol. Paths are arranged in the order obtained with *RxC* = $1/mmol, Y= 80%.

Section S6. Details of Chematica's syntheses of AMG641 with different *RxC-Y* settings.

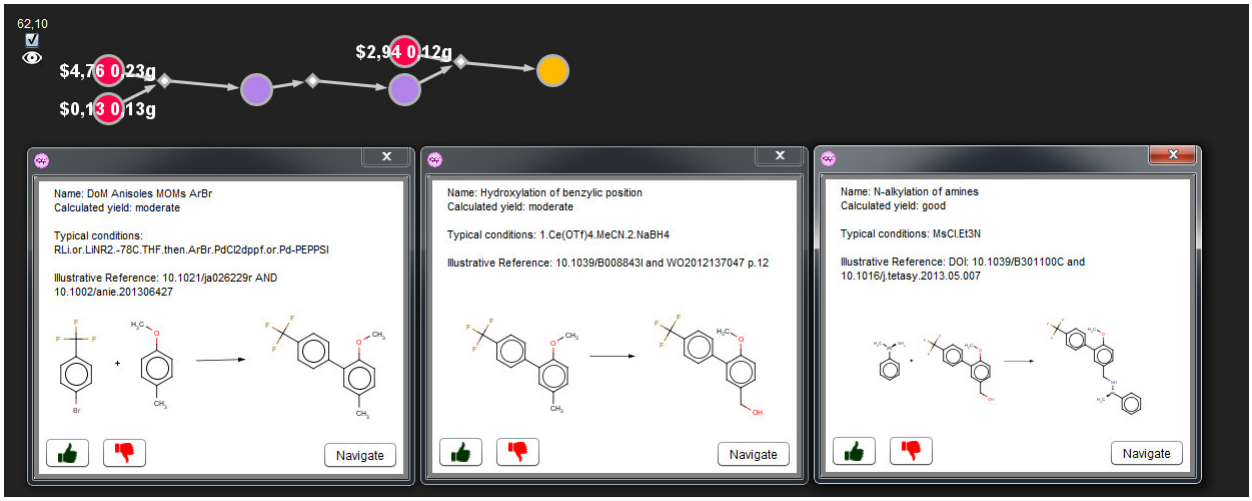

Figure S6. Details of the top-scoring synthetic pathway obtained for AMG641 with *RxC* = \$20/mmol, *Y* = 99% discussed in **Figure 8a**.

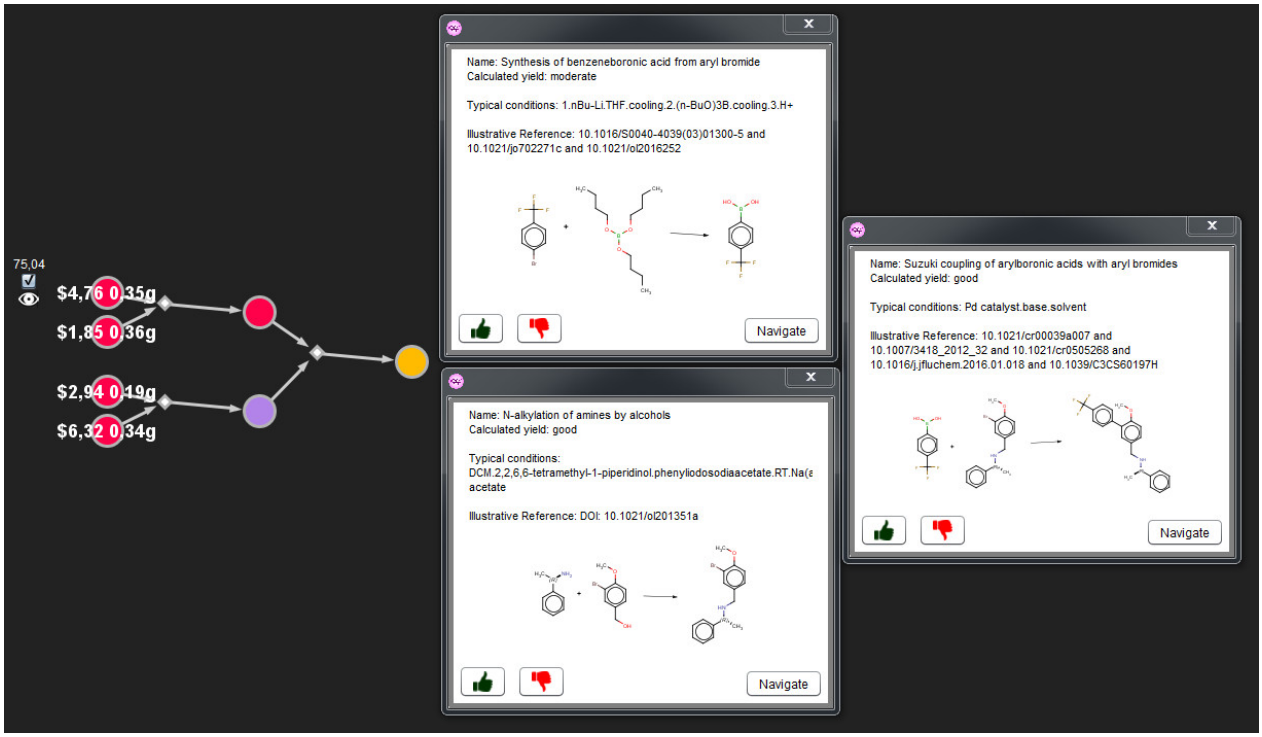

Figure S7. Details of the top-scoring synthetic pathway obtained for AMG641 with *RxC* = \$20/mmol, *Y* = 80% discussed in **Figure 8b**.

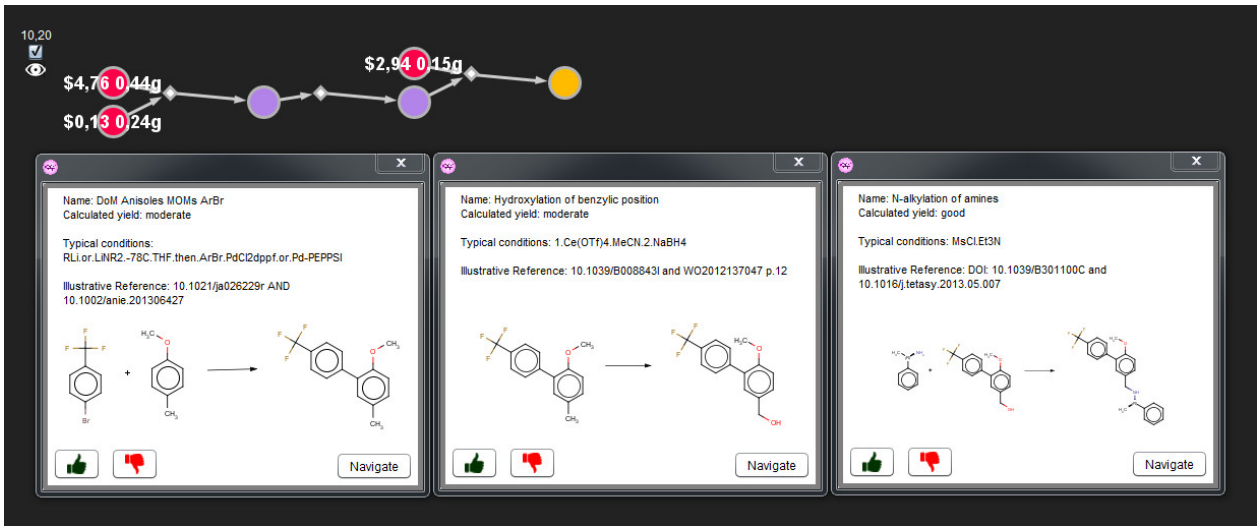

Figure S8. Details of the top-scoring synthetic pathway obtained for AMG641 with *RxC* = \$2/mmol, *Y* = 80% discussed in **Figure 8c**.

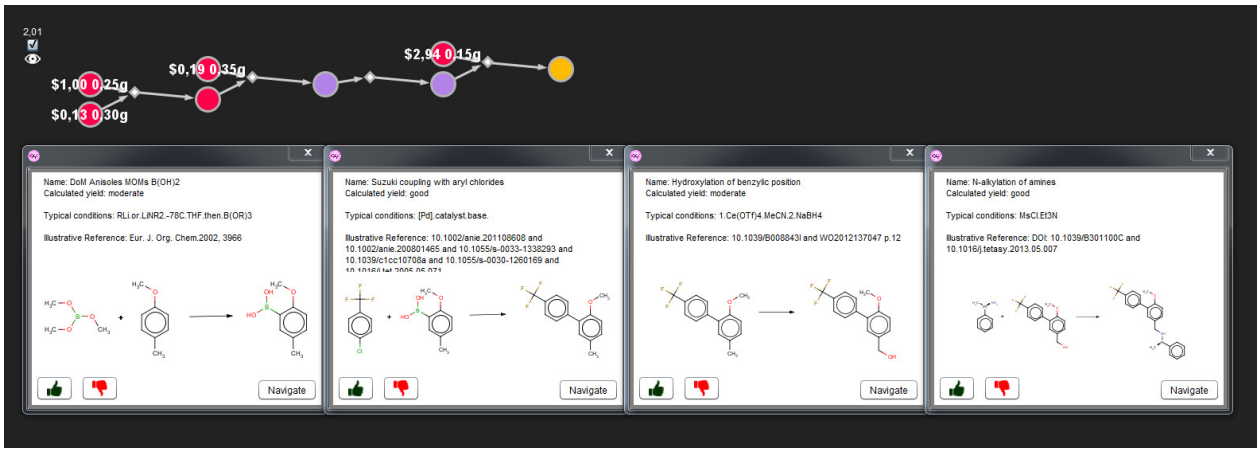

Figure S9. Details of the top-scoring synthetic pathway obtained for AMG641 with *RxC* = \$0.2/mmol, *Y* = 80% discussed in **Figure 8d**.

Section S7. Details of Chematica's syntheses of AMG641 with different *P* settings.

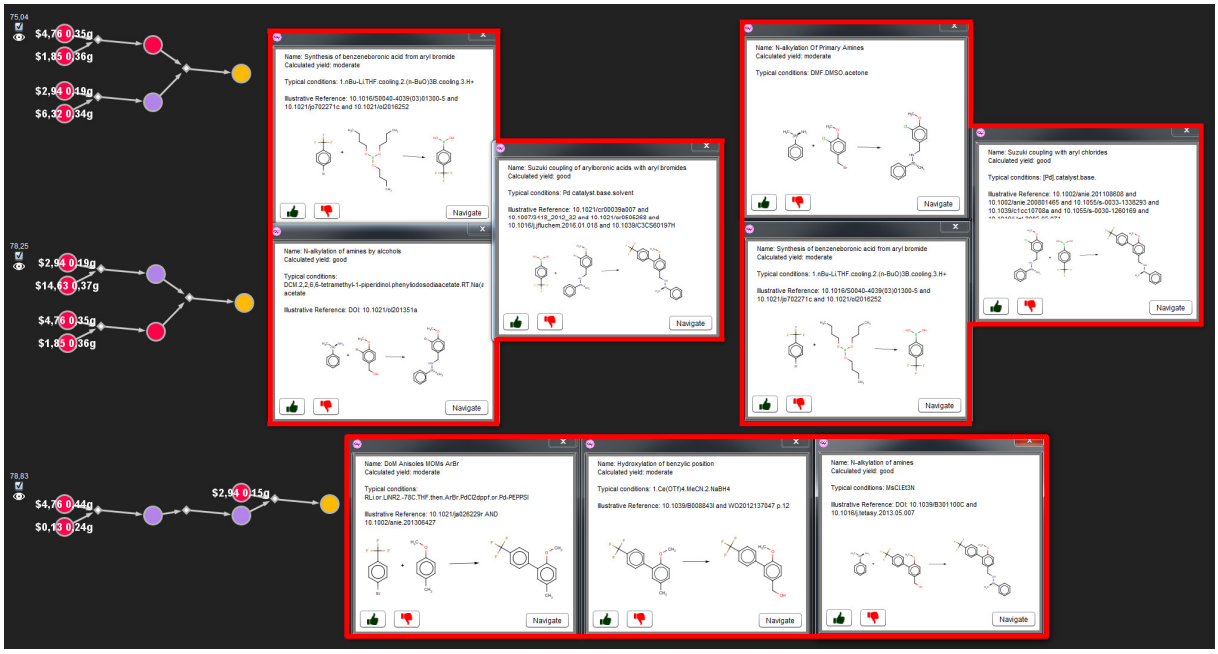

Figure S10. Details of the top three synthetic pathways obtained for AMG641 with *RxC* = \$20/mmol, *Y* = 80%, *P* = 0 discussed in **Figure 9a**.

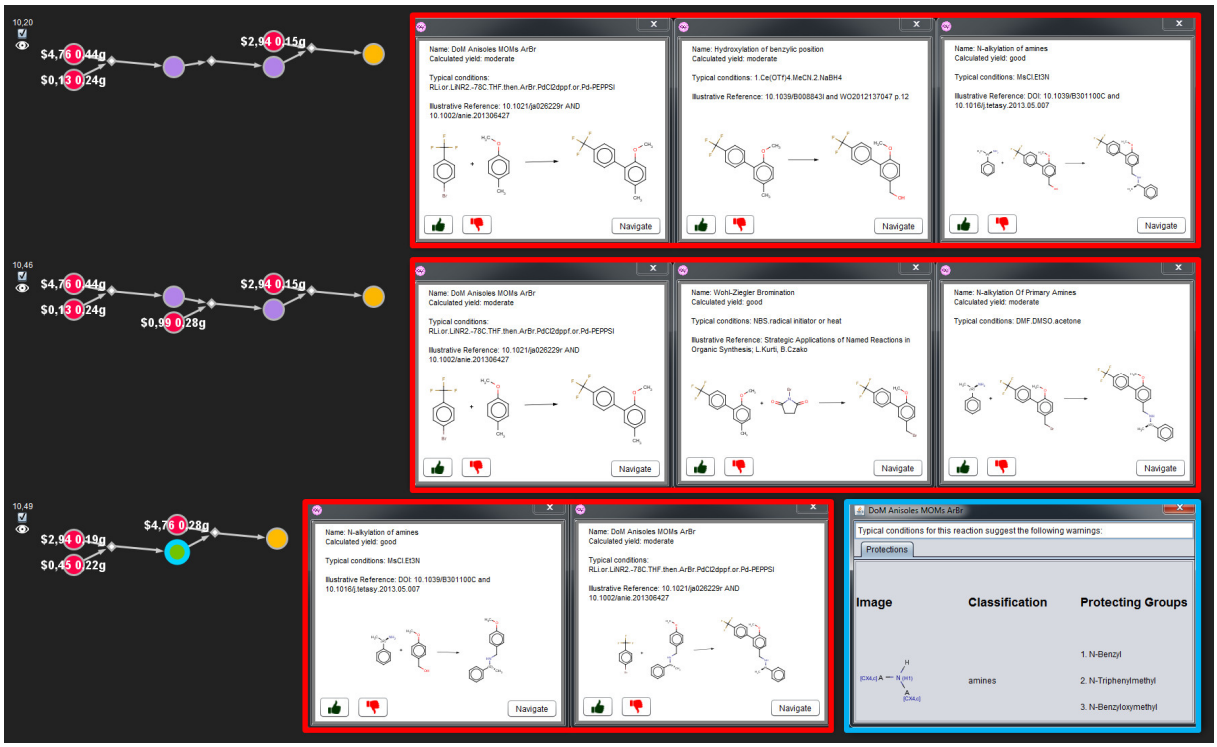

Figure S11. Details of the top three synthetic pathways obtained for AMG641 with *RxC* = \$2/mmol, *Y* = 80%, *P* = 0 discussed in **Figure 9b**. Chematica's proposed protecting group for the last step is shown in blue frame.

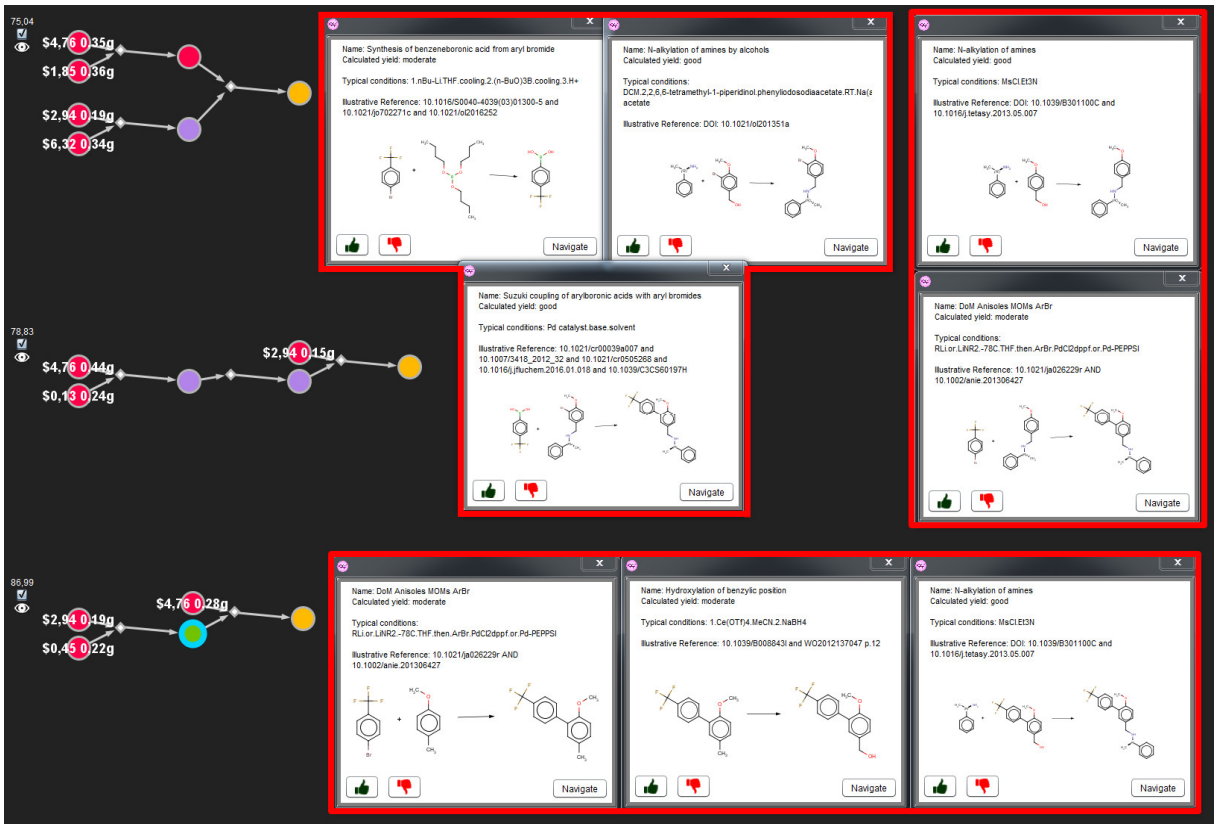

Figure S12. Details of the top three synthetic pathways obtained for AMG641 with Y = 80%, *P* = 10 000 discussed in **Figure 9c,d**. Paths are arranged in the order obtained with *RxC* = $20/mmol, Y= 80%.

Section S8. Details of Chematica's syntheses of the whisky lactone with different *P* settings.

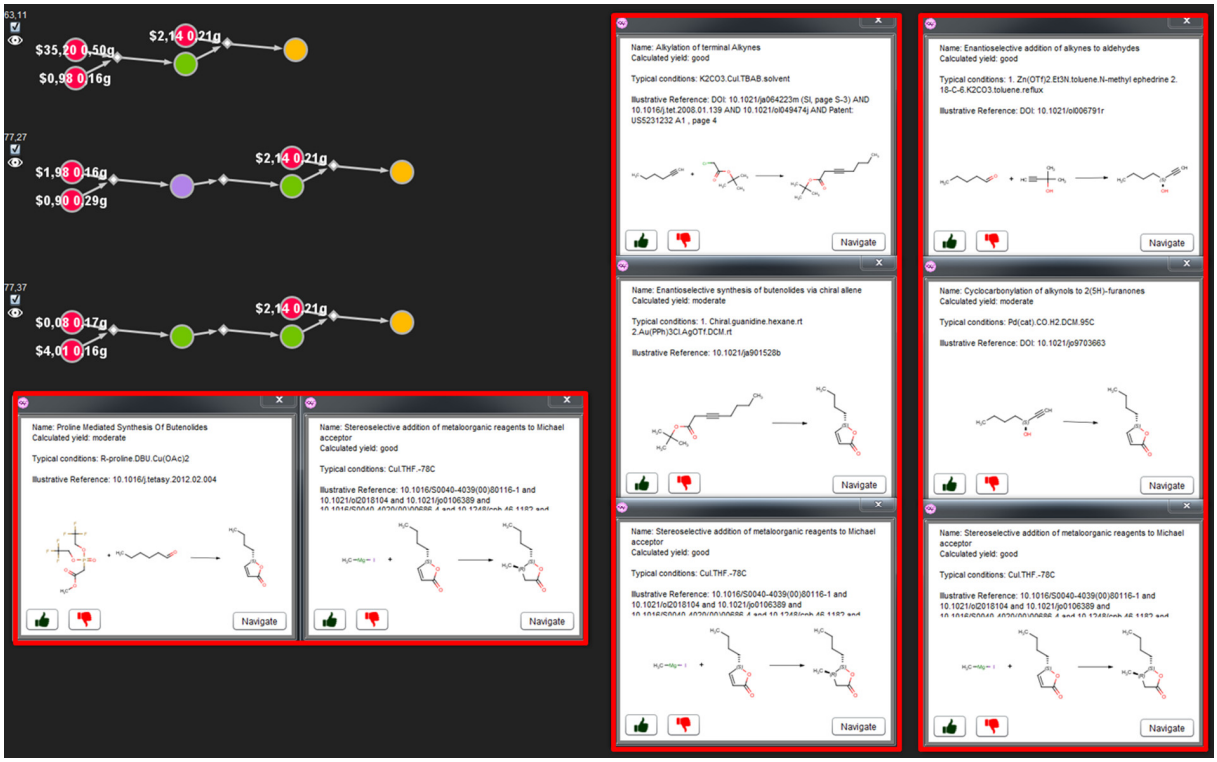

Figure S13. Details of the top three synthetic pathways obtained for whisky lactone with Y = 80%, P = 0 discussed in **Figure 10a**.

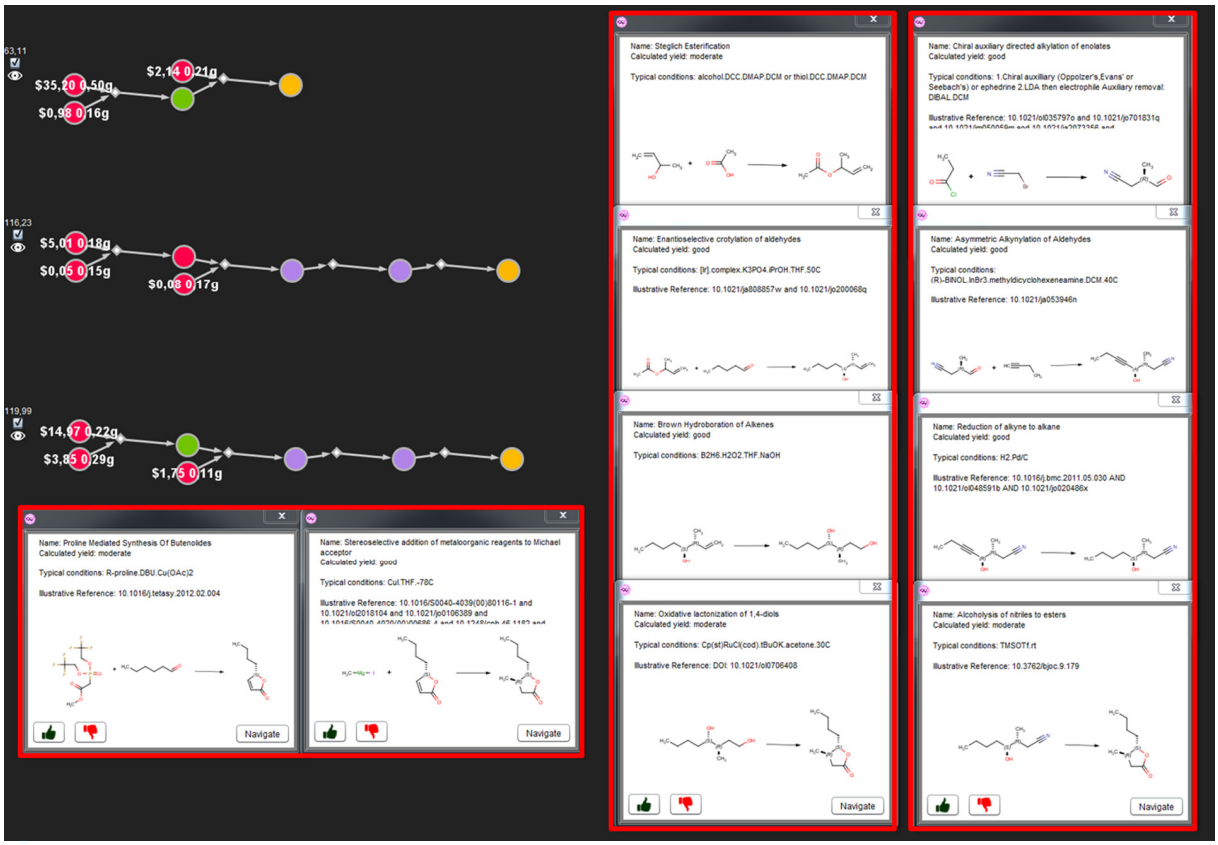

Figure S14. Details of the top three synthetic pathways obtained for whisky lactone with Y = 80%, P = 10 000 discussed in **Figure 10b**.