

DEE Implementation for EGAD

Mark Voorhies

June 20, 2003

Contents

1	Terminology	2
2	Overview	2
3	DeeTest	3
4	DEE objects	4
4.1	DeeSpace	4
4.2	DeeEnergy	5
4.3	DeeTable	6
5	DEE sieves	8
5.1	Simple DEE Singles	8
5.2	Goldstein Singles	9
5.3	Conformational Splitting	9
5.4	Bounding Expressions	10
5.4.1	Bounding Singles	12
5.4.2	Bounding Doubles	12
5.5	Branch and Terminate	12
5.6	Magic Bullet Doubles	13
5.7	Full Goldstein Doubles	14
6	Unification	14

7	Backtrack Methods	14
7.1	Exhaustive Search	14
7.2	Branch and Terminate	15

1 Terminology

In this documentation, “resimer” refers to a specific residue in a specific rotamer conformation at a specific backbone position, *i.e.* one of the discrete choices at a variable position in the design space.

The GMEC (Global Minimum Energy Conformation) is the set of resimer choices that minimizes the energy function. The problem that DEE solves is the determination of the GMEC for a given design space and a given energy function.

When cost estimates are given for a function (*e.g.*, “one round of simple DEE costs $O(n^2p^2)$ ”), p is the number of variable positions and n is the average number of resimer choices at each position. We expect $n < p$ for rotamer repacking and $n > p$ for design calculations.

2 Overview

Dead end elimination (DEE) is a deterministic method for solving rotamer optimization and protein design problems¹. The general protocol used in this implementation is²:

1. Initialization: Initialize a description of the design space (DeeSpace), a table of pairwise energies for the variable positions (DeeEnergy), and an elimination table (DeeTable).
2. Sieving: Apply various elimination criteria in a predetermined order³ to eliminate resimers based on the energy table and the current state of the elimination table.

¹See [5] and [7] for a good overview of the current state-of-the-art in DEE. See [6] for Hellinga’s take on DEE

²See [8] chapter 4 for a discussion of backtrack and sieve methods for exhaustive searching

³Mayo refers to this as the Algorithm Schedule or DEE cycle

3. Unification: Unify positions into “super-rotamers” and go back to the previous step. In Mayo’s implementation, this step is applied until the elimination table converges to a single solution or the memory required for the next unification is prohibitive. In the current implementation, unification is not performed.
4. Backtrack: Finally, a backtrack algorithm is used to extract the solution from the converged elimination table, or to exhaustively search for a solution in a partially converged table.

3 DeeTest

The test client code for the DEE implementation is `DeeTest.cpp`, which is compiled to `bin/DeeTest` relative to the base EGAD directory. `DeeTest` takes an EGAD format input file⁴ defining the design space and prints the GMEC sequence/conformation (in the event of convergence) or an error message if the final design space is too large for the exhaustive search (*c.f.* section 7.1). `ExhaustiveSearch` decides that the space is “too large” if it explores more than `max_nodes` nodes during the backtrack search⁵. The function `DeeInit()` in `DeeTest` allocates memory for an EGAD `PROTEIN`, calls `input_stuff` to read the input file, and calls `generate_lookuptable` to read or generate the lookup table. The resulting `PROTEIN` struct is then used to initialize a `DeeSpace` (4.1) which spawns the `DeeEnergy` (4.2) and `DeeTable` (4.3) to be used for the remainder of the run⁶. The DEE cycle takes up the bulk of `DeeTest.cpp`’s `main()`. Experimenting with different combinations of DEE criteria is encouraged⁷. Presently, logical elimination is applied heavily between other DEE criteria to ensure that each function receives a valid elimination table to operate on. The state of the elimination table is dumped

⁴I haven’t played around with the input file requirements for `DeeTest` much. Using `JOBTYPE MC_GA` works; there may be problems with other jobtypes, *e.g.* `HBUILD`, which do not require a lookup table. The current update of EGAD that Navin is working on will do the correct set-up for `JOBTYPE DEE`.

⁵`max_nodes` defaults to 100,000 and may be changed from the command line (`argv[2]`). For large problems, I use `max_nodes = 1,000,000`, which gives a worst case `ExhaustiveSearch` time of about ten minutes.

⁶The code in `DeeInit()` could be moved to the `DeeSpace` constructor, but I like keeping the input file parsing as a distinct step.

⁷For small problems, commenting out the slow implementation of `BoundingDoubles()` should change run times from hours to minutes.

frequently to track the progress of the run⁸. At the moment, the final GMEC is dumped to stdout. Ideally, DeeTest should use the standard EGAD output functions to report the result.

4 DEE objects

DeeSpace, DeeEnergy, and DeeTable are defined in DeeSpace.h and implemented in DeeSpace.cpp. All of the DEE and backtrack functions are implemented in terms of these three objects. For debugging purposes, a number of range-checking tests are provided in DeeSpace.h and are compiled if the macro MSV_DEE_RANGE_CHECKING is defined. Since these range-checks are on access functions that are typically called in the inner loops of the DEE algorithms, they produce a significant increase in run-time.

4.1 DeeSpace

DeeSpace defines the design space to be optimized. Most importantly, it provides NumPos(), the number of variable positions in the space⁹, and NumResimers(*pos*), the number of resimer choices at position *pos*¹⁰. NumResimers does not change with the status of the elimination table¹¹. The worst-case size of the design space (*i.e.* neglecting information about dead-end pairs) is:

$$\prod_i^{NumPos()} NumResimers(i) \quad (1)$$

MaxResimers() gives the maximum possible value of NumResimers. This is no longer used by any of the DEE algorithms¹² and may be worth dropping.

A second function of DeeSpace is to translate the DEE results to the outside world. To this end, it provides PosID(*pos*), ResimerID(*pos*, *resimer*),

⁸Navin: this can be targeted to a log file in the context of JOBTYP DEE.

⁹Note: after unification, a variable position may actually represent multiple amino acid positions

¹⁰Where $0 \leq pos < NumPos()$, *i.e.* *pos* is an index independent of the external numbering of the backbone positions.

¹¹In principle, we should be able to dynamically resize the design space as all resimers but one are eliminated at a given position. Certainly, this can be done as part of unification. Can resizing also be done on the fly without breaking the DEE algorithms?

¹²Mehagan's original implementation of doubles_max_min used MaxResimers() for static allocation. The current implementation uses a dynamically allocated array.

`ResType(pos, resimer)`, and `ChiAngles(pos, resimer)`, which return strings for the actual backbone position, resimer, residue type, and rotamer conformation for a given $(pos, resimer)$ pair¹³.

Finally, the `DeeSpace` provides functions for generating the remaining DEE objects: `CreateEnergy()` and `CreateTable()`. These functions return pointers to a `DeeEnergy` and a `DeeTable` respectively. The calling function “owns” the returned objects, and is responsible for de-allocating them using `delete`¹⁴.

The current implementation of `DeeSpace` is intended to interface with EGAD and is constructed from an EGAD PROTEIN struct. `DeeTest.cpp` contains code for allocating the PROTEIN and initializing it for the DEE objects.

A `DeeSpace` is passed to most of the DEE algorithms. Since both `DeeEnergy` and `DeeTable` provide pointers to the `DeeSpace` that they were created from, it may be a good idea to stop passing the `DeeSpace` itself.

4.2 DeeEnergy

`DeeEnergy` represents the pairwise energy table. Energies can be accessed via `get(ipos, ired, jpos, jres)`, which behaves like a two-dimensional array of resimer-resimer pairwise energies, and `get(ipos, ired)`, which behaves like a one-dimensional array of self energies (the diagonal of the two-dimensional array). The component of the total protein energy that does not depend on the variable positions is not included in the pairwise or self terms. The `get` functions are used in implementing most of the DEE criteria.

For the criteria that depend on bounding energies (currently `BoundingSingles`), `pair_energy(ipos, ired, jpos, jres)` and `pair_energy(ipos, ired)` are used instead. These energies correspond to dividing the fixed component of the energy evenly among the self energies, then dividing the self energies evenly

¹³The current implementation for `PosID` is incomplete: it returns the backbone ID used internally by EGAD without the conversion for chain ID and insertion codes. `ResimerID` uses EGAD’s internal representation, which just means indexing from 1 instead of from 0. `ResType` appears to be completely implemented. `ChiAngles` is completely implemented but differs from EGAD’s output in the `.out` and `.pdb` files in that it: 1) reports an extra decimal place and 2) only reports the existing chi angles for each rotamer (instead of printing 0.0 for non-existent chi angles)

¹⁴The previous DEE implementation used reference counting to garbage collect these objects. I’m not sure if the added complexity is worth it for objects that don’t tend to get copied, but it may be worth adding back in.

among the pairwise energies to give¹⁵:

$$E_{pair}(i_r) = 0 \quad (2)$$

$$E_{pair}(i_r, j_s) = \frac{E(i_r) + E(j_s) + 2E_{fixed}/p}{2p - 2} + \frac{E(i_r, j_s)}{2} \quad (3)$$

such that:

$$E_{total} = \sum_i^p \sum_{j \neq i}^p E_{pair}(i_r, j_s) = \sum_i^p \sum_{j \geq i}^p E(i_r, j_s) \quad (4)$$

The conversion for `pair_energy` is currently done at run-time, making this type of energy look-up expensive. It would be a good idea to make the `pair_energy` value the internal representation used by `DeeEnergy` and have `get()` return the same value (I'm pretty sure this wouldn't screw anything up).

`seq_energy(begin, end)` is a template function that calculates the energy for a complete sequence. `begin` and `end` should be iterators¹⁶ over a sequence of resimer indices. `seq_energy` includes all interaction energies in the protein, including E_{fixed} in equation 3¹⁷.

Finally, the `Space()` function provides a pointer to the `DeeSpace` that the `DeeEnergy` is based on, for use in interpreting `DeeEnergy` values in the context of the external representation of the design space.

4.3 DeeTable

`DeeTable` represents the dead-end elimination table. `get(ipos, ires, jpos, jres)` and `get(ipos, ires)` access this table as a two-dimensional or one-dimensional array as for the `DeeEnergy` object. `get(ipos, ires, jpos, jres)` is true for uneliminated pairs and false for dead-end pairs. Likewise, `get(ipos, ires)` is true for uneliminated resimers and false for eliminated resimers.

Note: This representation means that (eliminated.get(ipos, ires, jpos, jres) == false) for dead-ending pairs. This may seem counter-intuitive!!!

¹⁵See [4] for dividing the self energies among the pairwise energies. The division of the fixed component is used to make it easier to set the reference energy for bounding in terms of the total energy of known sequence/confomations.

¹⁶*C.f.* [10] chapter 19 for a discussion of C++ iterators

¹⁷Note: implementing `seq_energy` in terms of `pair_energy()` instead of `get()` gives agreement only to three decimal places. It would be worthwhile to track down the source of this rounding error.

The initial state of the table is true for all elements ($ipos, ires, jpos, jres$) except for elements where ($ipos = jpos$) and ($ires \neq jres$), *i.e.* it is not possible to have two different resimers at the same position. Attempts to access these impossible positions result in a core dump¹⁸.

Pairs of resimers can be flagged as dead–ending by calling `eliminate($ipos, ires, jpos, jres$)`¹⁹ and single resimers can be eliminated by calling `eliminate($ipos, ires$)`²⁰. Note that there is no function for uneliminating an already eliminated pair; this may be useful for efficient internal representations of the elimination table.

Two elimination criteria are provided directly by the elimination table²¹. `LogicalSingles()` eliminates any resimers that are dead–end with all remaining choices at another position; it returns the number of resimers eliminated this way. Each iteration of elimination is $O(n^2p^2)$, and `LogicalSingles()` iterates until no further eliminations can be performed. `LogicalPairs()` flags as dead–end any resimer pair that is incompatible with all remaining choices at any position. Each iteration of this criterion is $O(n^3p^3)$. `LogicalPairs()` iteratively applies the singles and pairs criterion until no further eliminations are possible and returns the number of eliminated pairs. It may be useful to move the iteration components of these functions to the client code and have `DeeTable` provide only the inner loops (*i.e.*, one round of singles elimination, one round of pairs elimination).

The `Space()` function provides a pointer to the generating `DeeSpace` object, as for `DeeEnergy` in the previous section.

The `dump()` function provides a diagnostic dump of the elimination table. It lists the resimer choices at undetermined positions, the resimer identity at determined positions, and an error message for positions where all resimers have been eliminated. An upper bound on the size of the conformational

¹⁸The rationale for this core dump is that making this impossible `get()` call is the result of faulty logic in the client code.

¹⁹Calling `eliminate` with $ipos = jpos$ currently results in a core dump since pair elimination and resimer elimination are fundamentally different operations.

²⁰When a resimer is eliminated, all pairs with this resimer are flagged as dead–ending. Some of the code takes advantage of this by assuming (`eliminated.get(i,r,j,s)`) implies (`eliminated.get(i,r)`).

²¹There are two reasons for making these criteria member functions of `DeeTable`. 1) They depend only on the state of the elimination table (*i.e.*, they do not require the energy table) and 2) they provide a validity check on the state of the elimination table (*i.e.* any client can use these functions to check if it is being passed or has produced a bad elimination table).

space is reported as:

$$N = \prod_i^p n_i = EXP[\sum_i^p \ln(n_i)] \quad (5)$$

where p is the number of positions and n_i is the number of uneliminated resimers at i . Note that this estimate does not account for dead-ending pairs.

Additional functions that should be added include save, load, and copy functions and a better estimator for the size of the uneliminated design space.

5 DEE sieves

DEE sieves perform eliminations on a DeeTable based on the state of that table and energies in a DeeEnergy. For this reason, most of the sieve functions take the same parameters, *viz.*:

```
int DeeSieve(const DeeEnergy&, DeeTable&, const DeeSpace&, FILE *)
```

where the return value is the number of eliminated resimers or dead-ended pairs and the file pointer is an optional message stream. The final two parameters should probably be dropped (DeeEnergy and DeeTable both provide pointers to the appropriate DeeSpace, and the value of the optional message stream is debatable).

Some sieves also require an upper bound on the total energy. At the moment, this energy is obtained from an independent MC_GA run using EGAD.

Given the common interface, it may be useful to make the DEE sieves objects derived from a common base class²².

5.1 Simple DEE Singles

The original DEE criterion is[1]:

$$E(i_r) - E(i_t) + \sum_{j \neq i} \min_s E(i_r, j_s) - \sum_{j \neq i} \max_s E(i_t, j_s) > 0 \quad (6)$$

²²This would be useful for applications that dynamically modify the DEE cycle

That is, a resimer i_r may be eliminated if, for some resimer i_t at the same position, the worst case energy of i_t with all other positions j is better than the best case energy of i_r with the same positions.

This criterion is not implemented in the current code, but the doubles version is included as part of the magic bullet doubles code (*c.f.* section 5.6).

The cost of one round of simple singles elimination is $O(n^2p^2)$.

5.2 Goldstein Singles

The Goldstein DEE criterion is[2]:

$$E(i_r) - E(i_t) + \sum_{j \neq i} \min_s [E(i_r, j_s) - E(i_t, j_s)] > 0 \quad (7)$$

That is, a resimer i_r may be eliminated if, for some resimer i_t at the same position, there is no way to choose resimers at the remaining positions such that i_r produces a lower energy than i_t . By moving $E(i_t, j_s)$ inside the min operator relative to equation 6, we increase the elimination potential but also increase the cost of the calculation to $O(n^3p^2)$.

Goldstein singles elimination is declared as `GoldsteinSingles(energy, eliminated, space, message_stream)` in `GoldsteinSingles.h` and is implemented in `GoldsteinSingles.cpp`.

5.3 Conformational Splitting

In conformational splitting, the conformational background (represented by the sum over j in Goldstein singles) is partitioned based on the rotamer choice at S splitting positions. If i_r can be eliminated by some resimer i_{t1} on the first partition, another resimer i_{t2} on the second partition, and so on for every partition, then i_r can be eliminated. For $S = 1$, this gives[7]:

$$E(i_r) - E(i_t) + E(i_r, k_v) - E(i_{t,v}, k_v) + \sum_{j \neq i, j \neq k} \min_s [E(i_r, j_s) - E(i_t, j_s)] > 0 \quad (8)$$

for all k_v .

Elimination is automatic on partitions k_v where (i_r, k_v) is a dead-end pair.

Conformational splitting singles elimination with $S = 1$ is declared as `SplitSingles(energy, eliminated, space, message_stream)` in `SplitSingles.h` and is implemented in `SplitSingles.cpp`. The cost is $O(n^3p^2)$.

SplitSingles also implements “split flags” [5]. This means that, for any partition k_v where some $i_{t,v}$ eliminates i_r , we can flag (i_r, k_v) as a dead-end pair.

Conformational splitting singles elimination with $S = 2$ is declared as `SplitSingles2(energy, eliminated, space, message_stream)` in `SplitSingles.h` and is implemented in `SplitSingles.cpp`. The cost is $O(n^4 p^3)$. The implementation strategy is to attempt elimination for a first splitting position, descending to a second splitting position only for partitions k_v (corresponding to first position resimers) that can’t be eliminated. For sub-partitions h_w (corresponding to second position resimers) that can be eliminated, (i_r, h_w) is flagged as dead-ending as above. Currently, the two levels of elimination in `SplitSingles2` are hard-coded as two loops. This should be recoded as a recursive function²³ that can descend to an arbitrary splitting depth. The cost for an arbitrary splitting depth should be [5] $O(n^{2+s} pq)$ where

$$q = \frac{(p-1)!}{s!(p-1-s)!} \quad (9)$$

5.4 Bounding Expressions

Given a set of resimer choices at a fixed set of positions f , we can calculate a lower bound on the energy of any complete sequence containing these resimers. The most accurate bound can be produced by an backtrack search⁷ over the remaining positions $i \notin f$. A less expensive, but less accurate, bound is given by [4]:

$$E_{bound} = 2 \sum_{i \in f} \sum_{j \in f, j \neq i} E_{pair}(i_r, j_s) + \sum_{i \notin f} \min_r \left[2 \sum_{j \in f} E_{pair}(i_r, j_s) + \sum_{j \notin f, j \neq i} \min_s E_{pair}(i_r, j_s) \right] \quad (10)$$

This bounding expression is implemented by the following functions:

- `BoundingEnergy(energy, f)` calculates the full expression. It is declared in `BoundingEnergy.h` and implemented in `BoundingEnergy.cpp`. The remaining functions are declared and implemented in `BoundingEnergy.cpp`. *energy* is the pairwise energy table. *f* is the set of fixed resimers, represented as a `FixedSequence` (defined in `FixedSequence.h`). `FixedSe-`

²³Ideally, we want the iterative equivalent of a recursive function, *c.f.* [9] pg. 109 for an example.

quence provides specialized iterators that iterate over only the uneliminated choices at each position.

- `fixed_bound_energy(energy, f)` calculates

$$E_{fix} = 2 \sum_{i \in f} \sum_{j \in f, j \neq i} E_{pair}(i_r, j_s) \quad (11)$$

- `var_bound_energy(energy, f)` calculates

$$E_{var} = \sum_{i \notin f} \min_r \left[2 \sum_{j \in f} E_{pair}(i_r, j_s) + \sum_{j \notin f, j \neq i} \min_s E_{pair}(i_r, j_s) \right] \quad (12)$$

- `single_bound_energy(energy, f, r)` calculates

$$E_{var,fix} = 2 \sum_{j \in f} E_{pair}(i_r, j_s) \quad (13)$$

r is the current resimer choice at i , represented as a `resimer_iterator` (defined in `FixedSequence.h`).

- `double_bound_energy(energy, f, r)` calculates

$$E_{var,var} = \sum_{j \notin f, j \neq i} \min_s E_{pair}(i_r, j_s) \quad (14)$$

This term is invariant for a given set of fixed positions[4] (it is slightly dependent on the resimer choices at f , since resimers j_s that are dead-end with resimers in f can be excluded from the minimization. Neglecting this dependence gives a bounding energy that is less than or equal to the correct bounding energy). Therefore, the complexity of the bounding calculation can be greatly reduced by precomputing this term for all i_r each time f changes.

The return type for all of the bounding energy functions is `InfDouble` (defined in `InfDouble.h`). An `InfDouble` is a double plus an infinity flag. If the infinity flag is false, the `InfDouble` is treated as a normal double. If the infinity flag is true, then

- The `InfDouble` is `+infinity` if the double value is greater than zero.

- The InfDouble is -infinity if the double value is less than zero.
- The InfDouble is undefined if the double value is zero.

The String() member function of InfDouble gives a string representation²⁴. All operators on InfDoubles are defined as member functions, so the InfDouble needs to be on the left hand side of mixed expressions (*e.g.* *InfDouble + double* will work, *double + InfDouble* won't).

It is possible that regular doubles provide all of the features of InfDoubles. It would be good if this is the case, as it would give a big speed up.

5.4.1 Bounding Singles

BoundingSingles(*energy, eliminated, space, E_ref, message_stream*) applies equation 10 to each resimer, eliminating resimers where the bounding energy is greater than *E_ref*.

OptBoundingSingles(*energy, eliminated, space, E_ref, message_stream*) attempts to implement the precalculation of equation 14. I am not sure if it is working yet. Last time I looked, it produced fewer eliminations than BoundingSingles, which is expected. Ideally, the precalculation should be implemented in terms of the functions called by BoundingEnergy.

5.4.2 Bounding Doubles

BoundingDoubles(*energy, eliminated, space, E_ref, message_stream*) applies equation 10 to each pair of resimers, flagging as dead-end pairs where the bounding energy is greater than *E_ref*. This is currently the slowest step in the DEE cycle since it does not use precalculation.

OptBoundingDoubles(*energy, eliminated, space, E_ref, message_stream*) attempts to implement the precalculation of equation 14. It is definitely a flawed implementation as it can produce incorrect eliminations.

5.5 Branch and Terminate

Equation 10 is also useful in the context of searching the remaining sequences in a partially converged elimination table. See section 7.2 for more details.

²⁴May be preferable to write a casting function?

5.6 Magic Bullet Doubles

When the Goldstein criterion (*c.f.* section 5.2) is extended to pairs of resimers [1], the cost increases from $O(n^3p^2)$ to $O(n^5p^3)$. In Magic Bullet doubles[3], this cost is reduced to $O(n^4p^3)$ by using only one “magic bullet” resimer pair to attempt eliminations at each position. This strategy is implemented as `MagicBulletDoubles(energy, eliminated, space, message_stream)` in `MagicBulletDoubles.h` and `MagicBulletDoubles.cpp`. Several additional functions are defined in `MagicBulletDoubles.cpp` that should be useful for other variations of Goldstein Doubles²⁵:

- `doubles_simple(i_pos, j_pos, best_pair, m, eliminated, message_stream)` applies the doubles version of the simple DEE criterion (*c.f.* 5.1) to eliminate resimer pair m with the magic bullet pair $best_pair$ for the pair of positions i_pos, j_pos . It returns true if m can be eliminated by this criterion.
- `doubles_goldstein(i_pos, j_pos, best_pair, m, eliminated, message_stream)` applies the doubles version of the Goldstein criterion (*c.f.* 5.1) to eliminate resimer pair m with the magic bullet pair $best_pair$ for the pair of positions i_pos, j_pos . It returns true if m can be eliminated by this criterion.
- `doubles_max_min(i_pos, j_pos, elim_count, energy, eliminated, space, message_stream)` returns a list of uneliminated resimer pairs for the pair of positions i_pos, j_pos and calculates best and worst case energies for these pairs with the remaining positions. It is non-const with respect to the elimination table to allow for logical elimination during the search over resimer pairs. Any such elimination happens during the call to `add_max_min` and causes `elim_count` to be incremented.
- `add_max_min(max_min_list, i_pos, j_pos, r, s, energy, eliminated, space, message_stream)` calculates the best and worst case energy for the resimer pair i_r, j_s and appends this value to `max_min_list`. If the pair can be logically eliminated, no value is appended; in this case, `add_max_min` is allowed to flag i_r, j_s as dead-ending in the elimination table (this code is currently commented out); the number of pairs eliminated in this way is given as the return value (presently, the return value is always zero).

²⁵Mehagan’s original implementation of Magic Bullet Doubles and full Goldstein Doubles is in terms of these functions.

5.7 Full Goldstein Doubles

Full Goldstein Doubles is not yet implemented. There is an initial implementation written by Mehagan for our original DEE package that (I think) uses the q_{rs} and q_{uv} metrics[3].

6 Unification

Unification is not yet implemented. It should be implemented in terms of the DEE objects; *viz.*, `Unify(DeeTable)` should generate a new `DeeSpace` which acts as a filter over the previous one, making an underlying pair of positions appear to be a single super-rotamer.

7 Backtrack Methods

When no further eliminations can be performed²⁶ The eliminated design space is exhaustively searched for the GMEC sequence/conformation. Given enough time, this will always succeed. Since “enough time” may be on the order of centuries, the following algorithms will give up after traversing a maximum number of nodes. The return type for these functions is a pointer to an `ExhaustiveSolution` (defined in `ExhaustiveSearch.h`). Searches that fail without encountering any leaf nodes²⁷ return a 0 (null) pointer. Searches that fail after encountering one or more leaf nodes return an `ExhaustiveSolution` representing the best sequence encountered so far; the success flag in the `ExhaustiveSolution` is set “false” to indicate that this sequence may not be the GMEC²⁸. Searches that run to completion return an `ExhaustiveSolution` representing the GMEC with the success flag set to “true”.

7.1 Exhaustive Search

`ExhaustiveSearch(energy,eliminated,space,max_nodes,message_stream)` implements the general backtrack algorithm, as described in [8] pg. 109.

²⁶Once unification is implemented, this condition will be: when there is not enough memory for further unification.

²⁷*i.e.* the algorithm couldn't find any complete sequences compatible with the elimination table before traversing the maximum number of nodes.

²⁸In practice, it is very unlikely that an incomplete run of `ExhaustiveSearch` will return the GMEC since the search order is not biased by energy.

7.2 Branch and Terminate

Branch and Terminate is not yet implemented. It will be implemented in terms of the code being used for the Bounding Singles implementation (*c.f.* section 5.4).

References

- [1] J. Desmet, M. De Maeyer, B. Hazes, and I. Lasters. The dead-end elimination theorem and its use in protein side-chain positioning. *Nature*, 356:539–542, 1992.
- [2] R. F. Goldstein. Efficient rotamer elimination applied to protein side-chains and related spin glasses. *Biophysical Journal*, 66:1335–1340, 1994.
- [3] D. B. Gordon and S. L. Mayo. Radical performance enhancements for combinatorial optimization algorithms based on the dead-end elimination theorem. *Journal of Computational Chemistry*, 19:1505–1514, 1998.
- [4] D. B. Gordon and S. L. Mayo. Branch-and-terminate: a combinatorial optimization algorithm for protein design. *Structure*, 7:1089–1098, 1999.
- [5] Gordon D. J., Hom G. K., Mayo S. L., and Pierce N. A. Exact rotamer optimization for protein design. *Journal of Computational Chemistry*, 24:232–243, 2003.
- [6] L. L. Looger and H. W. Hellinga. Generalized dead-end elimination algorithms make large-scale protein side-chain structure prediction tractable: Implications for protein design and structural genomics. *J. Mol. Biol.*, 307:429–445, 2001.
- [7] N. A. Pierce, J. A. Spriet, J. Desmet, and S. L. Mayo. Conformational splitting: A more powerful criterion for dead-end elimination. *Journal of Computational Chemistry*, 21:999–1009, 2000.
- [8] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, Inc., 1977.
- [9] R. Sedgewick. *Algorithms*. Addison-Wesley, 1983.

- [10] B. Stroustrup. *The C++ Programming Language*. Addison–Wesley, 3rd edition, 1997.