Analysing Open Source Software to Better Understand Long Term Memory Structures in the Human Brain.

Thomas Mullen
tom@tom-mullen.com

Abstract
As AI models become larger, replicating long term memory structures (LTM-S) may produce the same benefits that evolution provided the human brain (efficiency, performance, and extensibility).

At the heart of this paper is the conjecture that software structures are close representations of LTM-S. If this is true, then open source can be considered a huge database of easily searchable LTM-S examples that could assist in a deeper understanding of the same.

The paper proposes a general refactoring algorithm based around two elements of LTM-S, chunks and analogies. The underlying aim is to develop mechanisms and theories to analyse the analogical and chunking structures employed in software.

1. Introduction
The processes of introducing a required behaviour to a code base involves translating the requirements into dependencies amongst a set of variables. Even when a successful dependency graph is identified, software languages allow many structural choices of how to represent that graph. Design principles then provide the guidance to establish which alternative is easiest to understand.

There may be background and context that form part of this process. However, this paper looks specifically at the relationship between the dependency graph and the software structures that the developer alights upon. At the heart is the conjecture that those structures are close representations of the LTM-S of the dependency graph in the developer's brain.

The conjecture that *code structures are representations of long-term memory structures* was stated in Mullen (2009) in a slightly different form. This paper attempts to further validate this conjecture by proposing a refactoring algorithm which is based on elements of LTM-S (specifically, chunking and analogies).

The refactoring algorithm follows this process and has three elements:

- Software Chromatography. The code to be analysed is distilled into two graphs. Both graphs have shared nodes which represent the inherent variables/values/attributes in the code to be refactored. The first graph is a normalised dependency graph (which is inclusive of conditionals) and the second is a structural graph that identifies where each node is represented in the software language elements.
- Candidate Solutions Library. A selection of dependency sub-graphs, each of which is mapped to a number of alternative structural graph solutions.
- Chunking Objective Function. A measurement of simplicity, a function comprised of coupling and cohesion parts.

A more detailed description of the algorithm is provided in section 3.2, but the precis is that the code base to be analysed is firstly distilled into the dependency and structure graphs. Each dependency sub-graph in the library is searched for in the distilled dependency graph. Where a match is found each alternative structural solution is then employed in the distilled structural graph and the change to the chunking objective function is calculated. If the objective function is improved than that refactor becomes a recommendation.

The advantages of LTM-S that evolution has provided the human brain would likely benefit AI models. Webb et al (2023) showed that *"[GPT-3] appears to display an emergent ability to reason by analogy"* so it's possible that the training processes already produces them. However, a better understanding of

LTM-S could assist in recognising analogical structures in AI models and refine the training process to produce them. For example, Holyoak and Thagard (1997) propose that analogies are chosen based on similarity, structure and purpose. Gentner (1983) proposes that the depth of the behaviour function is a driver. Analysing which abstractions/analogies are chosen to be represented by classes in open-source code may help to fill in specific details of the process in choosing primary analogies.

2. Software Structures are LTM-S Conjecture

2.1 LTM-S in Languages

Whenever we understand anything, the endgame is to lay down long term memories. Only then can we utilise that knowledge, build upon it and apply it to other domains (to expand understanding in those areas). If we had a choice of how to represent something and the primary aim was to make it easier to understand, then the closer our representation gets to LTM-S, the less translation is needed when we (or others) try to understand the knowledge it represents.

The same argument could be applied to natural languages. For example, if you needed to explain something to me (especially knowledge that it took time for you to acquire). You would first access the LTM-S in your brain that contains that knowledge. You would then serialize those structures as natural language. I would deserialize those sentences and (hopefully) produce the same LTM-S that exist in your mind. In that way I could achieve the same knowledge without the cost you had to achieve it. Indeed, cognitive linguists analyse natural languages to see if they betray how the mind works.

If natural languages are the serialization of LTM-S, then it could be said that software languages provide a direct access equivalent. So, from a 30,000 foot point of view, the conjecture that *software structures are long term memory structures* makes sense.

It's probably inevitable that at some point AI models will communicate with one another to pass on knowledge without the necessity of repeating costly training. If AI models employ the same LTM-S as in a human brain, then the languages they will use will likely be based upon those structures (potentially a massive parallel version). The more we understand about LTM-S, the better our ability to eavesdrop or, more usefully, influence that knowledge transfer (especially to correct bias or error).

2.2 Chunking Analogies

If we consider the aspect of software design that simplifies the code structure without changing the behaviour. Then this is the same process as laying down long term memories, i.e. chunking analogies (Mullen, 2009). Specifically,

- Chunking – where memory elements are grouped so that elements within a chunk are strongly related to each other, but loosely related to elements in other chunks. The design principle of low coupling/high cohesion guides the developer to chunk the code.
- Analogies – a mapping of two or more domains that contains attributes and behaviours. This is similar to the class description of the OO paradigm. However, there are other software language structures that can represent analogies (Mullen, 2009).

Software design is a process of identifying the different language structures we can employ to represent the abstractions/analogies of the required behaviour and then choosing whichever provides the best coupling and cohesion (chunking). The similarity between the cognitive psychologists' description of LTM-S and software design principles and languages adds further validation to the *software structures are LTM-S* conjecture.

2.3 So What?

Even if we are prepared to accept at this stage that the conjecture is true, how does that help us?

Evolution has provided the human brain with LTM-S that produce an efficient, performant, and extensible knowledge store. These structures could be just as advantageous to AI models. Understanding LTM-S and the process that creates them would be key to recognising these structures in AI models or guiding the training process to produce them.

If the *software structures are LTM-S* conjecture is true, then open-source software could provide us with a huge searchable database of LTM-S examples. We could analyse these structures with the same aim that cognitive linguists analyse natural languages.

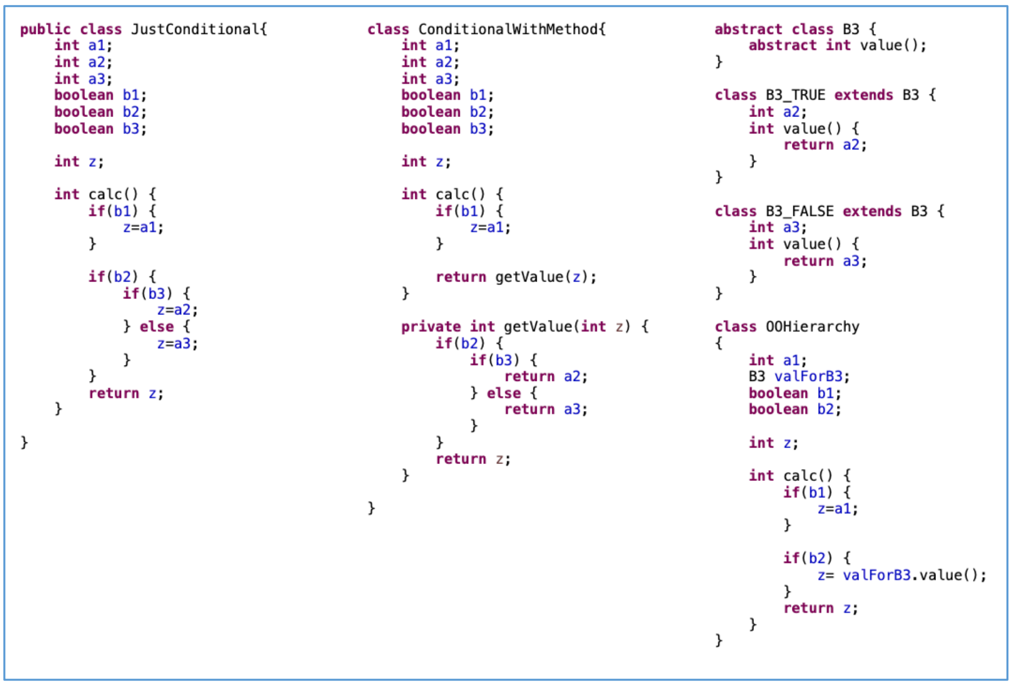
The next section proposes a general refactoring algorithm. It is not expected that this will produce a usable refactoring tool for developers. However, if it can come up with reasonable (or, hopefully, illuminating) refactoring suggestions then that could provide a more detailed validation of the conjecture. In addition, the mechanisms and theories needed for the algorithm could be employed to analyse open source. This would provide a large enough dataset for correlations to be identified and provide further evidence of the links between software structures and LTM-S.

3. Analysing Analogical and Chunking Structures In Open Source Code

Before going into the details of the algorithm it may be beneficial to provide an example of the type of refactoring that the tool is intended to deliver. This is included in the next section with the description of the algorithm following it.

3.1 Example

The three code examples in Fig 1 represent the same behaviour with respect to the relationship between z and its constituent elements a1, a2, a3, b1, b2 and b3. In all the examples, the same labels are used (z, a1, etc.) but in real code they would be different labels and may have additional dependencies amongst them, and with other elements. The proposal is to represent the dependency relationship in a normalised graph so that the isomorphic nature can be easily identified.



```
public class JustConditional{
    int a1;
    int a2;
    int a3;
    boolean b1;
    boolean b2;
    boolean b3;

    int z;

    int calc() {
        if(b1) {
            z=a1;
        }

        if(b2) {
            if(b3) {
                z=a2;
            } else {
                z=a3;
            }
        }
        return z;
    }

}
```

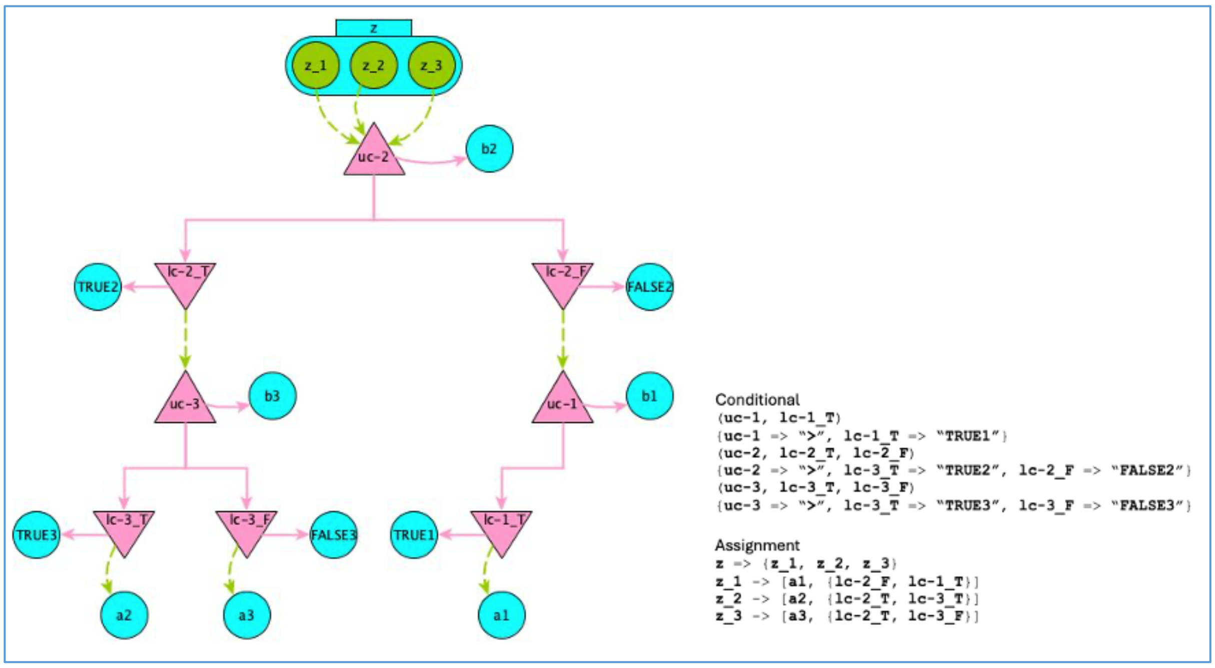
```
class ConditionalWithMethod{
    int a1;
    int a2;
    int a3;
    boolean b1;
    boolean b2;
    boolean b3;

    int z;

    int calc() {
        if(b1) {
            z=a1;
        }

        return getValue(z);
    }

    private int getValue(int z) {
        if(b2) {
            if(b3) {
                return a2;
            } else {
                return a3;
            }
        }
        return z;
    }

}
```

```
abstract class B3 {
    abstract int value();
}

class B3_TRUE extends B3 {
    int a2;
    int value() {
        return a2;
    }
}

class B3_FALSE extends B3 {
    int a3;
    int value() {
        return a3;
    }
}

class OOHierarchy
{
    int a1;
    B3 valForB3;
    boolean b1;
    boolean b2;

    int z;

    int calc() {
        if(b1) {
            z=a1;
        }

        if(b2) {
            z= valForB3.value();
        }
        return z;
    }
}
```

Figure 1 – Three different code structures that provide the same behaviour between z and its dependencies a1, a2, a3, b1, b2, b3.

Fig 2 shows the resulting dependency graph for all the examples in Fig 1. There are three main composite elements to the graph;- conditional elements; pure functional sub-graphs; and assignments. This is discussed in more detail in Appendix A.

In addition to the dependency graph, a partner structural graph details how the different elements are represented in the language. This contains the same variable and expression nodes that are in the dependency graph but are placed in the package/class/method structure where they are defined in the code. Fig 3 shows the structure graph for the middle code snippet in Fig 1.



```
Conditional
(uc-1, lc-1_T)
{uc-1 => ">", lc-1_T => "TRUE1"}
(uc-2, lc-2_T, lc-2_F)
{uc-2 => ">", lc-3_T => "TRUE2", lc-2_F => "FALSE2"}
(uc-3, lc-3_T, lc-3_F)
{uc-3 => ">", lc-3_T => "TRUE3", lc-3_F => "FALSE3"}

Assignment
z => {z_1, z_2, z_3}
z_1 -> [a1, {lc-2_F, lc-1_T}]
z_2 -> [a2, {lc-2_T, lc-3_T}]
z_3 -> [a3, {lc-2_T, lc-3_F}]
```

Figure 2 – Normalised dependency graph for all the three code examples in Fig 1

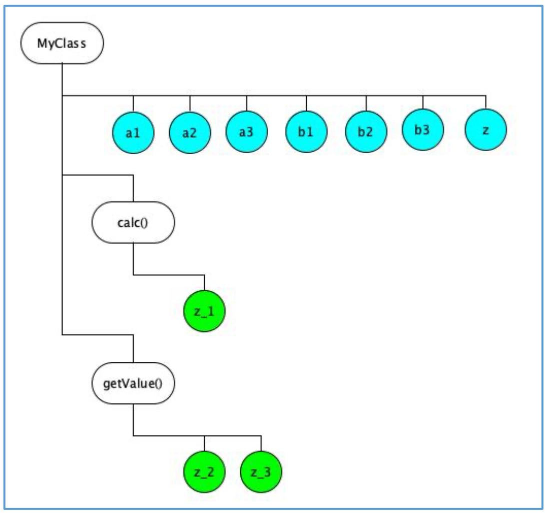
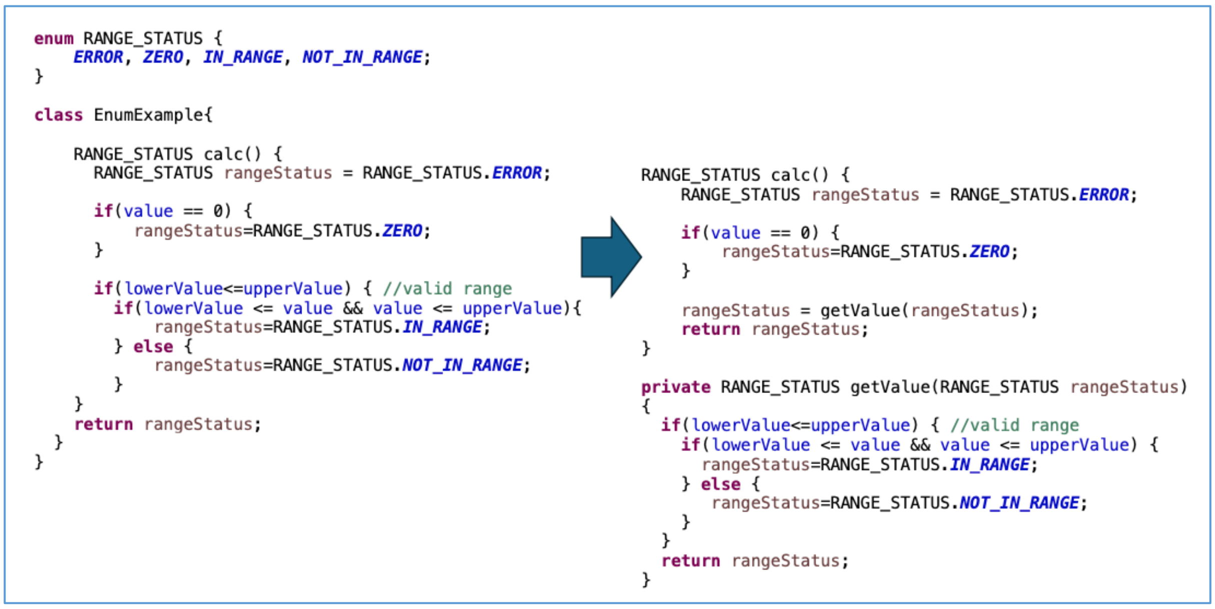


Figure 3 – Structure graph for the second code example in Fig 1

To illustrate a refactoring example, consider the code in the left-hand column of Fig 4, for which the dependency graph is in Fig 5. There is a subgraph isomorphism with the normalised graph of the three code examples in Fig 2. Specifically with the mapping;- b1-> `(value==0)`; b2 -> `(lowerValue<=upperValue)`; b3 -> `(lowerValue <= input && value <= upperValue)`; a1 -> `ZERO`; a2 -> `IN_RANGE`; a3 -> `NOT_IN_RANGE;` z_1 -> z_ZERO; z_2 -> z_IN ;and z_3-> z_NOT. Consequently, we could employ the structure associated with any of the three examples in Fig 1. The second structure in Fig 1 applied to this code would produce something like that in the right-hand column of Fig 4.



```
enum RANGE_STATUS {
    ERROR, ZERO, IN_RANGE, NOT_IN_RANGE;
}

class EnumExample{

    RANGE_STATUS calc() {
      RANGE_STATUS rangeStatus = RANGE_STATUS.ERROR;

      if(value == 0) {
          rangeStatus=RANGE_STATUS.ZERO;
      }

      if(lowerValue<=upperValue) { //valid range
        if(lowerValue <= value && value <= upperValue){
            rangeStatus=RANGE_STATUS.IN_RANGE;
        } else {
            rangeStatus=RANGE_STATUS.NOT_IN_RANGE;
        }
      }
      return rangeStatus;
    }
}
```

```
RANGE_STATUS calc() {
    RANGE_STATUS rangeStatus = RANGE_STATUS.ERROR;

    if(value == 0) {
        rangeStatus=RANGE_STATUS.ZERO;
    }

    rangeStatus = getValue(rangeStatus);
    return rangeStatus;
}

private RANGE_STATUS getValue(RANGE_STATUS rangeStatus)
{
  if(lowerValue<=upperValue) { //valid range
    if(lowerValue <= value && value <= upperValue) {
      rangeStatus=RANGE_STATUS.IN_RANGE;
    } else {
       rangeStatus=RANGE_STATUS.NOT_IN_RANGE;
    }
  }
  return rangeStatus;
}
```

Figure 4 – Example refactor

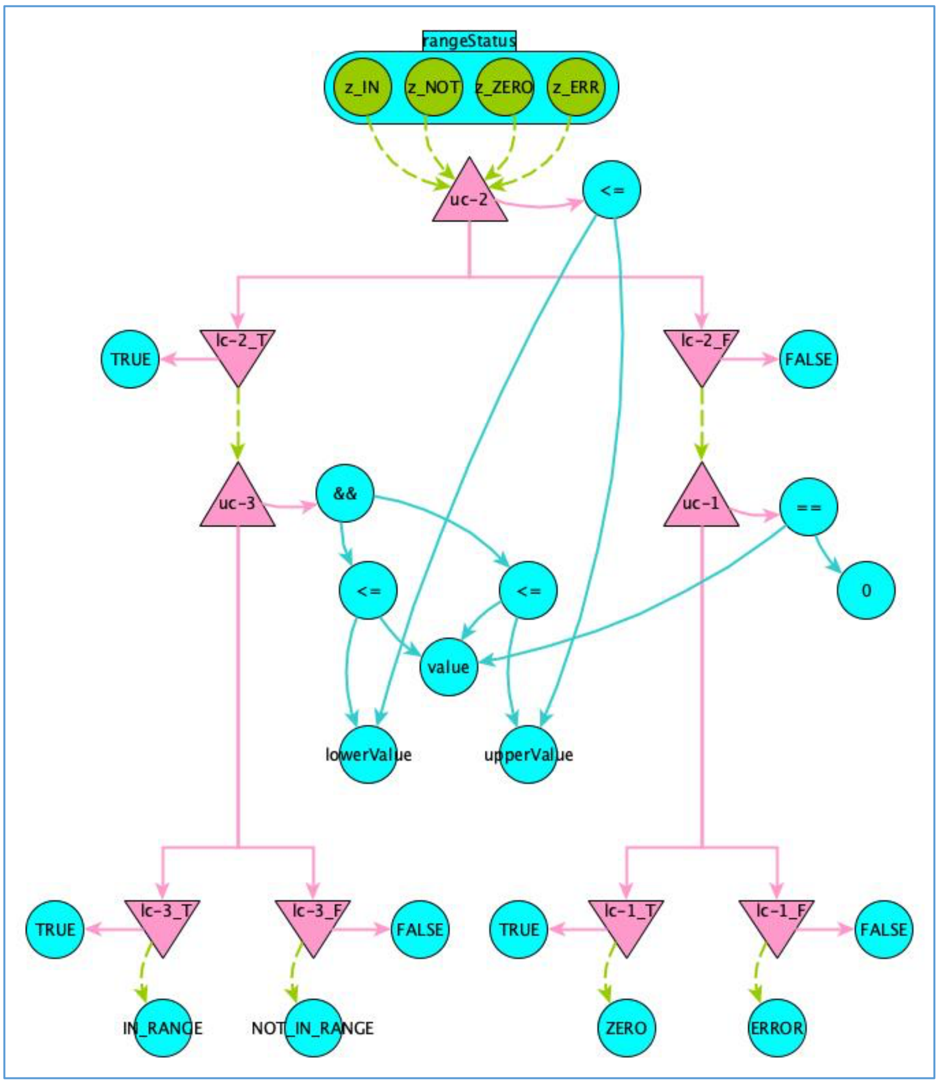


Figure 5 –Normalised dependency graph for the code example in Fig 4

3.2 Proposed Algorithm

The refactoring algorithm has three elements

- Software Chromatography. Where the code to be analysed is distilled into the dependency and structural graphs.
- Candidate Solutions Library. A selection of dependency sub-graphs and their potential structural graph solutions. An example entry in the library would be the dependency graph shown in Fig 2 paired with the three alternative structural graphs that represent the code examples in Fig 1.
- Simplicity objective function. A single value function to measure the simplicity of the solution. The proposal is to use a chunking penalty function (made up from coupling and cohesion penalty functions). This is detailed in Appendix B, but is comprised of distance functions on the structural and dependency graphs.

Definitions:

$\mathbf{F_0}$ the dependency graph of the code to be analysed.
$\mathbf{S_0}$ the structural graph of the code to be analysed.
$\mathbf{G}$ (with elements $\mathbf{g_i}$) the set of dependency sub-graphs in the candidate solutions library.
$\mathbf{T_i}$ (with elements $\mathbf{t_{ij}}$) the set of alternative structural graphs for the dependency sub-graph $\mathbf{g_i}$.

The steps of the algorithm are as follows:

1. Distil the code that is to be analysed and produce the dependency and structural graphs $\mathbf{F_0}$ and $\mathbf{S_0}$.
2. For Each dependency sub-graph $\mathbf{g_i}$ in the candidate solutions library, $\mathbf{G}$:
   a. Search for an isomorphism of $\mathbf{g_i}$ in $\mathbf{F_0}$.
   b. For each finding of the candidate sub-graph, loop through the alternative structural graph solutions $\mathbf{t_{ij}}$ that are stored with $\mathbf{g_i}$ in the candidate solutions library.
      i. Calculate the change to the chunking penalty function when $\mathbf{S_0}$ is modified to employ the alternative structure $\mathbf{t_{ij}}$. If this is an improvement, then the refactor of $\mathbf{t_{ij}}$ to $\mathbf{S_0}$ is presented as a recommendation.

One question remains as to how we build the candidate solutions library. This could be achieved by performing the distillation process across open-source code and identifying dependency sub-graphs using a clustering algorithm. Further filtering could be employed based on the number of alternative structural solutions for a dependency sub-graph. It is expected that a similar process would be useful when we attempt to analyse open source for LTM-S.

3.2 Short Term Memory Capacity Limit

The capacity limit of short-term memory (Miller, 1956) has a significant influence on our ability to understand. It would be natural to assume that any algorithm which represents human understanding should incorporate those limits. The proposed algorithm doesn't explicitly mention short-term memory capacity limits but does model them implicitly in two ways.

Firstly, the candidate solutions library contains structures from existing code. A reasonable assumption is that at least one person (the developer of that code) must have understood it. Since short-term memory is the gateway to long term memory then the structures in the library have successfully passed through someone's short term memory and so, taken in isolation, are within the capacity limits.

Secondly, there is an argument that a function which could describe the limit is likely to be combinatorial in nature. As more and more elements are added to short term memory the possibility of cognitive overload becomes non-linearly more likely. The cohesion penalty function is made up from pairs of siblings in the structural graph, which is $O(n^2)$. This may or may not be sufficient to model the capacity limit so could need to be revisited.

4. Conclusion

This paper proposes a mechanism to distil software into two graphs, dependency and structural. The dependency graph is normalised to facilitate recognising isomorphisms of behaviour between different structural implementations. The intention is that this could be useful for the following:

- Better understanding of LTM-S. If we accept the conjecture that *code structures are direct representations of long-term memory*, the open source would be a huge searchable database of LTM-S examples. The ability to recognise code structures that are isomorphic in their behaviour would be key to this search. Software structures are not necessarily the only elements that could be mapped to LTM-S. For example, the same arguments could apply to architectural designs. However, software structures are likely to be more useful, due to the large dataset available with open source and the ability to identify dependency graphs and their associated structural representations.
- Refactoring tool. This paper proposes a work-in-progress algorithm for a general refactoring tool. Analysing open source to build up a library of refactoring candidates and choosing recommendations only when a simplicity function (based on cognitive principles) is improved.
- Code stored as a dependency graph. Most languages in use today require that code is stored in flat files. The files often fit a purpose for defining primary abstractions/analogies or for chunking/cohesion considerations. If the refactoring algorithm, above, could be proved successful then there may be an opportunity for storing the normalised dependency graph. As developers initiate searches on the code, the best structure to represent the dependency sub-graph of the search result would be generated. In most cases, this would likely produce the same results as present in the flat file solution. However, it may allow for a cleaner representation of cross-cutting concerns, as well as reacting to changes to the dependency graph that would otherwise require a significant refactoring.

5. References

M. Fowler, K. Beck, J. Brant, and W. Opdyke (1999). Refactoring: Improving the Design of Existing Code. Addison- Wesley ISBN 0-201-48567-2

D. Gentner (1983). Structure-mapping: A theoretical framework for analogy. In Cognitive Science, 7, pp 155-170.

K. J. Holyoak and P. Thagard. The Analogical Mind (1997). http://cogsci.uwaterloo.ca/Articles/Pages/Analog.Mind.html

Miller, G. A. (1956). "The magical number seven, plus or minus two: Some limits on our capacity for processing information". Psychological Review. 63 (2): 81–97. CiteSeerX 10.1.1.308.8071. doi:10.1037/h0043158. hdl:11858/00-001M-0000-002C-4646-B. PMID 13310704. S2CID 15654531.

Mullen, T. 2009. Writing code for other people: cognitive psychology and the fundamentals of good software design principles. In Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA '09). Association for Computing Machinery, New York, NY, USA, 481–492. https://doi.org/10.1145/1640089.1640126

J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. J. ACM 23, 1 (Jan. 1976), 31–42. https://doi.org/10.1145/321921.321925

Taylor Webb, Keith J Holyoak, and Hongjing Lu (2023). Emergent analogical reasoning in large language models. Published at Nature Human Behaviour (2023) https://doi.org/10.1038/s41562-023-01659-w.

Appendix A : Normalised Dependency Graph

The normalised dependency graph consists of connected parts of three different element constructs. All nodes in the graph have a unique identifier, however for the sake of brevity the examples given here sometimes use the label associated with that element. If we take a simple max function as an example:

```
int max(int i, int j) {
    if (i > j) {
        return i;
    } else {
        return j;
    }
}
```

The normalised dependency graph is in Fig A.1

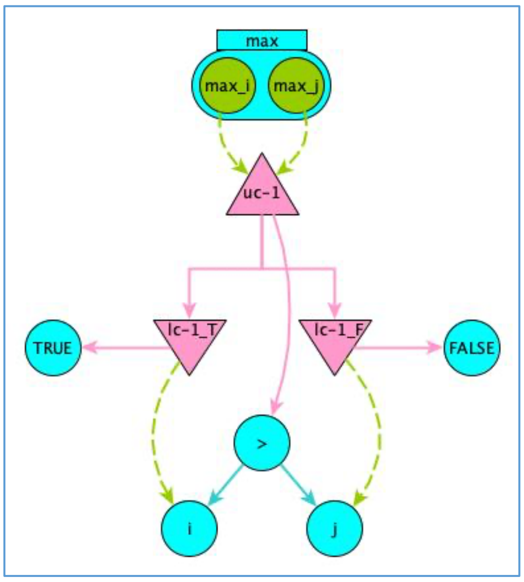


Figure A.1 – Normalised dependency graph for the max function.

The three constructs are

- **Conditional** – consists of the following nodes
    - An upper conditional node (triangle shape pointing up), labelled uc-1 in the max example.
    - Two or more lower conditional nodes (triangle shape pointing down), labelled lc-1_T and lc-1_F in the max example.
        - For conditional statements there will be two lower conditional nodes (representing the true and false cases).
        - Class hierarchies will also be represented by conditional structures with a lower conditional node for each class that inherits from a super class or interface.
        - Switch/case statements will have a lower conditional node for each case element.

  The edges in the conditional construct are:
    - Conditional edges (between the upper conditional node and each lower conditional node)
    - An upper conditional value edge and lower conditional value edges (one for each lower conditional node). This is easiest understood by looking at the example where it represents the conditional statement. The value from the upper conditional edge is the boolean expression and the lower conditional edges point to TRUE and FALSE. So for a runtime case the upper conditional value is compared to the lower conditional values to determine which lower conditional branch represents the dependency in that case.
- **Functional** – a sub graph that consists of variables, literal values and operands that combine them. The `i>j` construct in the graph is an example of this.

- **Assignment** – a link between a variable and a functional construct that also contains the lower conditional nodes that must all be satisfied for that dependency. There may be many assignments for a variable (some of which point to the same functional construct). The effective boolean expressions are in disjunctive normal form.

A.1 Shorthand notation

The graph can be described by the following shorthand notation that represent the edges. In each case the examples given are the shorthand notation for the graph of the max function, given in fig x.

- **Conditional** – this consists of two parts.
  - Firstly, an upper conditional node with a set of lower conditional nodes e.g.

    `(uc-1, lc-1_T, lc-1_F)`

    This represents the edges between the upper and lower conditional nodes e.g.

    `{uc-1 => lc-1_T, uc-1 => lc-1_F}`
  - Secondly, the edges between the conditional nodes and the conditional values

    `{uc-1 => “>”, lc-1_T => “TRUE”, lc-1_F => “FALSE”}`
- **Functional** – simply the edges in the functional graph. The example is below. However a functional label, f(i,j), is provided for reasons that will become apparent in section A.2 (Establishing Isomorphisms)

  `f(i,j) -> {“>” => i, “>” => j}`
- **Assignment** – consisting of two parts.
  - The first is the set of assignments for a variable, e.g.

    `max => {max_i, max_j}`
  - the second defines each of these assignments using the value and the set of lower conditionals, e.g.

    ```
    max_i -> [i, {lc-1_T}]
    max_j -> [j, {lc-1_F}]
    ```

A.2 Establishing Isomorphisms

The candidate solutions library contains dependency subgraphs that we need to search for in the dependency graph of the code we are analysing. The intention is to use Ullman's (1976) sub-graph isomorphism algorithm. The number of different node and edge types will be leveraged in the refinement step to aid in the process (e.g. upper conditional nodes will only be mapped to upper conditional nodes).

There will also be different flavours of isomorphisms. This would include:

1. Assignment value as function. In this case the functional constructs are collapsed to a single function node (`f(i,j,…)` as mentioned in the functional shorthand notation in section B.1). The choice would then to be whether we include the function parameter list in the isomorphism. Whilst including the parameter list may lead to a more targeted mapping, not including the list could open up to more refactoring recommendations (with the belief that the chunking objective function would reject many unsuitable refactorings).
2. Pure functional: where the dependency sub-graph would contain a single functional construct. In this case the algorithm would identify refactors such as “Extract Method” and “Introduce Variable” (Fowler et al, 1999).

Appendix B : Chunking Penalty Function

The chunking penalty function consists of two parts;- coupling and cohesion. Both these parts rely on distance functions on the dependency and structural graphs. The initial proposal for these functions is given but analysis of open source may help to refine these.

- Coupling: a reduced form of the dependency graph is used that strips out all the non-variable nodes (but retains any dependency paths). The edges then provide the direct dependencies amongst the variables. Each edge is taken and the penalty is how far away the two nodes are on the structural graph (the further they are away the worse the cohesion). A simple distance function is initially proposed. However, edge weights or a function may need to be applied.
- Cohesion: this applies to elements that are structurally grouped together. Sibling nodes on the structural graph are processed and, for each pair, a metric of the similarity of the pair is used. This is achieved by using a vector that contains the distance of a node (on the dependency graph) to all other nodes. The cosine distance is then used to provide a measure of dissimilarity.

Definitions:

s_{ij} – the distance between nodes i and j in the structural graph.

d_i – a vector giving the distance of node i with each other node on the dependency graph.

$$chunking\ penalty = \sum_{\substack{i,j\ nodes\ of\ an \\ edge\ in\ the\ reduced \\ dependency\ graph}} s_{ij} + \sum_{\substack{i,j\ sibling \\ nodes\ in\ the \\ structural\ graph}} \left(1 - \frac{\underline{d_i}.\underline{d_j}}{\left|\underline{d_i}\right|\left|\underline{d_j}\right|}\right)$$