Implementation and Exploration of Rust-based Graph Library

Rao Zhang
Electrical Engineering Department
Stanford, California
zhangrao@stanford.edu

ABSTRACT

Rust is a safe systems programming language. It introduces the concept of ownership, used for memory management to prevent segment fault, resulting from improper memory manipulation. However, this mechanism to guarantee memory safety may bring about obstacles for usage of shared memory resources. Graph is such an example with shared nodes and edges mutually connected to each other. In this report, the author will explore patterns of Rust language by iterations on a graph library, discuss several designs and implementations of the library based on Rust language, and analyze memory issues caused by graph representations and Rust memory management strategies.

KEYWORDS

Rust, Graph processing library, Memory management

ACM Reference Format:
Rao Zhang. 2017. Implementation and Exploration of Rust-based Graph Library. In *Proceedings of CS242 (Programming Languages).* ACM, Stanford, CA, USA, Article none, 6 pages. https://doi.org/none

1 BACKGROUND

Graph is a common data structure, pervasively used in various areas, especially for topology and network science. Social networks, computer networks, road networks, all of them can be abstracted to graph representations. A graph can be represented by its nodes and edges. If there is a direction associated with each edge, this graph is called directed graph. If there is a weight associated with each edge, this graph is called weighted graph.

Since graph is ubiquitous in our daily lives, graph libraries are developed to represent a graph and provide convenient interfaces to solve problems related to graphs, with different languages. In this report, the author will focus on the implementations of a graph library written in Rust language.

Rust is a systems programming language [1]. It is safe, concurrent and practical [2]. The Rust system is designed to be memory safe, and it does not permit null pointers, dangling pointers, or data races in safe code [3]. Rust introduced a new concept, ownership. Ownership is Rust's most unique feature, and it enables Rust to make memory safety guarantees without needing a garbage collector [4]. Ownership follows three rules [5]. (1) Each value in Rust has a variable that's called its owner. (2) There can only be one owner at a time. (3) When the owner goes out of scope, the value will be dropped. Along with these rules, ownership can be only "borrowed" or "moved", which guarantees the uniqueness of allocated memory for the value. Finally, a borrow checker performed during compile time eliminates all these memory issues, such as memory leaks.

However, these rules and checks somethings seem too strict, so that they make this language not so flexible in its applications. Graph is such a data structure that multiple edges connecting to one node, which means it is possible that multiple reference shares the same value in some implementations. In this sense, Rust's strict borrow checker makes graph library difficult to handle. This becomes the greatest challenge in this project. Hence, the author is going to explore features of Rust language, practices various graph implementations with different internal representations and try to discover memory issues as well as find solutions to these issues.

The rest of this paper is organized as follows. In section 2, the author will elaborate the iterations of implementations of the graph library. In each iteration, the features of Rust language and analysis of memory management issues will be explained, in the context of design of the graph library. The author will also theorize the ways to get around such problems. In section 3, experience and lessons learned during development on usability of Rust language are documented. In section 4, the conclusion of this report will be drawn.

2 APPROACHES

The author implements the graph library in an iterative way. He gradually improves and completes the library during exploration and practice of the language. In this section, the author will tell his thoughts of designs or/and solutions to memory issues in chronological order.

2.1 The First Iteration

As shown in Figure 1, a graph consists of nodes and edges. Each node has a unique identifier, which is an i32 type value here. Each edge has a direction, which is uniquely determined by its start node and finish node.

The first iteration of the graph library is derived from assignment 6 starter code. In the starter code, the graph is a directed graph without weight, and represented by an adjacent list with the following data types and structures.

- nodes: HashMap<i32, T>
- edges: HashMap<i32, Vec<i32> >
- count: i32

where "nodes" HashMap maps a node index (i32 type) to node value (T type), "edges" HashMap maps a node index (i32 type) to a vector of node indexes (i32 type), simulating an adjacent list. For each entry in "edges", the key is a start node and the value is a vector of finish nodes. The node index is the unique identifier for a node. Newly added node is assigned an index incremented by one from the previous largest index. Taking into account node removal

Programming Languages, December 2017, Stanford, CA USA

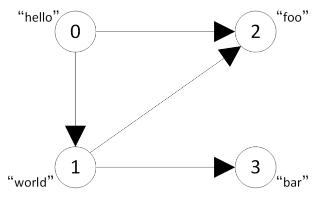

Figure 1: A sample directed graph.

and edge removal, a counter is needed to keep track of this largest index, which is "count" field in the adjacent list representation. Figure 2 shows an example of the internal data structure of this graph representation.

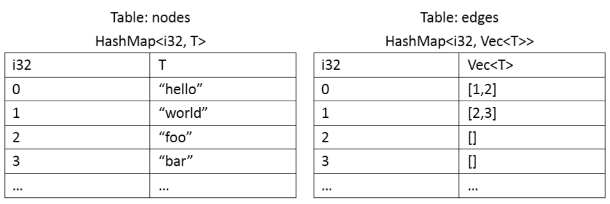

Table: nodes
HashMap<i32, T>

i32	T
0	"hello"
1	"world"
2	"foo"
3	"bar"
...	...

Table: edges
HashMap<i32, Vec<T>>

i32	Vec<T>
0	[1,2]
1	[2,3]
2	[]
3	[]
...	...

Figure 2: Internal data structure of the first iteration.

Since the starter code only implements the following methods,

- add_node: add a node
- add_edge: add an edge
- value: get the value of a specific node
- neighbors: get all neighbors' indexes of a specific node
- connected: check the connectivity between two nodes

which is far away from complete to form a graph library, the author complemented the library with the following methods,

- remove_node: remove a specific node
- remove_edge: remove a specific edge
- get_nodes: get all nodes' indexes
- get_edges: get all edges denoted by nodes pair
- clear: clear the graph
- size: get the graph's size by node
- is_empty: check whether the graph is empty

where remove_node not only removes the node, but also removes all edges connected to that node. Besides, the author implemented `std::fmt::Debug` trait, which works together with `println!` macro to format the output, showing the internal data in a clear way.

All the work in the first iteration aims to add more convenient methods and human readable outputs, to make the starter code more like a graph library.

2.2 The Second Iteration

The second iteration is only a small upgrade on the first iteration. In the first iteration, "edges" in the graph is represented by a HashMap, each entry of which is a mapping from node index (i32 type) to a vector of node indexes (i32 type). Since the complexity of element look-up, removal operation of a vector is O(n), the author therefore replaced this vector structure to a HashSet structure, which has O(1) complexity of accessing and deleting an entry. The entries in the HashSet is the same as the values in the vector, which are the finish nodes. Hence, the graph is now represented by an adjacent list with the following data types and structures.

- nodes: HashMap<i32, T>
- edges: HashMap<i32, HashSet<i32> >
- count: i32

Figure 3 shows an example of the internal data structure of this graph representation.

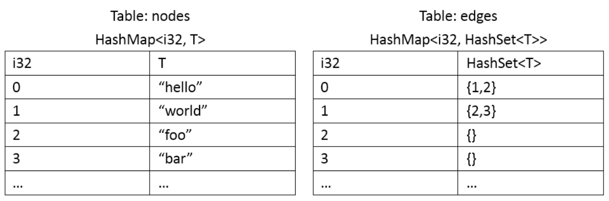

Table: nodes
HashMap<i32, T>

i32	T
0	"hello"
1	"world"
2	"foo"
3	"bar"
...	...

Table: edges
HashMap<i32, HashSet<T>>

i32	HashSet<T>
0	{1,2}
1	{2,3}
2	{}
3	{}
...	...

Figure 3: Internal data structure of the second iteration.

2.3 The Third Iteration

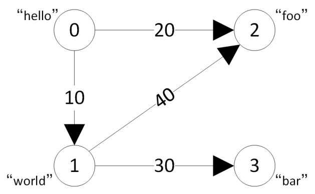

Figure 4: A sample weighted directed graph.

In the third iteration, the graph is still represented by an adjacent list, but converted to a weighted graph, with the following data types and structures. A sample is shown in Fig 4.

- nodes: HashMap<i32, T>
- edges: HashMap<i32, HashMap<i32, i32> >
- count: i32

The difference of internal data structure between the second iteration and the third iteration is that the author associated a weight to each edge in the graph. The association is implemented by another HashMap, which maps the finish node to a weight (i32 type). Therefore, as the finish node is known, the weight can be obtained by looking up the HashMap. An example of this representation is shown in Fig 5.

Since we have more information associated to the graph, we can do more interesting things related to the weight. The author implemented another two methods to play with,

- weight: get the weigh of a specific edge
- dijkstra: find the shortest path between two nodes

which makes the graph library more handy and complete.

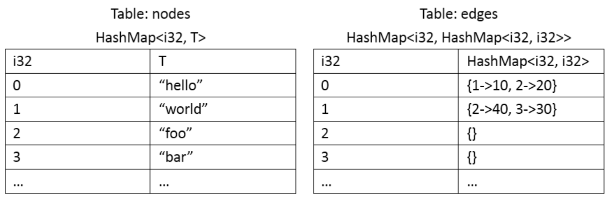

Table: nodes
HashMap<i32, T>

i32	T
0	“hello”
1	“world”
2	“foo”
3	“bar”
…	…

Table: edges
HashMap<i32, HashMap<i32, i32>>

i32	HashMap<i32, i32>
0	{1->10, 2->20}
1	{2->40, 3->30}
2	{}
3	{}
…	…

Figure 5: Internal data structure of the third iteration.

2.4 The Forth Iteration

The forth iteration has a fundamental change in the internal representation of the weighted directed graph. The graph is now represented by an adjacent matrix, with the following data types and structures.

- nodes: Vec<i32, T>
- edges: Vec<Vec<i32> >
- count: i32

where each entry is represented by an element in the "edges" matrix. The value of the element is the weight of that edge. For example, as shown in Fig 6, there are two edges going out from node 0. They are from node 0 to node 1 with weigh 10 and from node 0 to node 2 with weight 20, respectively. The adjacent matrix representation

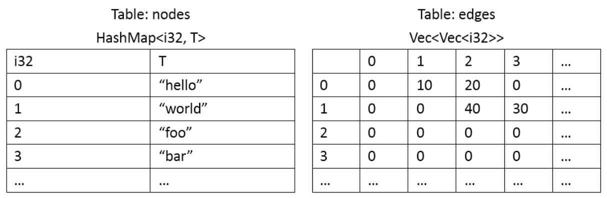

Table: nodes
HashMap<i32, T>

i32	T
0	“hello”
1	“world”
2	“foo”
3	“bar”
…	…

Table: edges
Vec<Vec<i32>>

	0	1	2	3	…
0	0	10	20	0	…
1	0	0	40	30	…
2	0	0	0	0	…
3	0	0	0	0	…
…	…	…	…	…	…

Figure 6: Internal data structure of the fourth iteration.

is as common as the adjacent list for a graph in graph analysis, however, it is not a good candidate for our graph library, because its high space and time complexity in the graph processing.

To store such a graph, the "edges" needs $O(n^2)$ space in this representation. If the graph is a sparse graph, which mean the number of edges is $O(n)$, then the matrix would be a sparse matrix. In this situation, this representation wastes a lot of space in storing edges. In fact, most graphs in our lives are sparse graphs. Hence, it is not a good idea to store graph in this representation.

Besides, this representation also has high time complexity in graph processing, such as adding or removing a node. The node addition and removal operation needs to look up the index of the node in "nodes", and then change the scale of the matrix in "edges", which requires O(n) operations, is also too expensive to be efficient.

Taking the expensive storage and operations into consideration, the author rescinded this practice just after implementing two methods, add_node and add_edge.

2.5 The Fifth Iteration

Until the forth iteration, the author didn't meet many memory management issues, because the "nodes" and "edges" in the graph representation are not so "closely connected". They are either connected by key-value pairs in a HashMap or by index in a vector. In the fifth iteration, the author decided to make the "nodes" and "edges" closely mutually connected, which means nodes point to edges and edges point to nodes. To achieve this target, the author changed the internal representation. The graph is now represented by a star representation with the following data types and structures, as shown in Fig 7.

- nodes: HashMap<String, Rc<RefCell<NodeType> > >
- edges: HashSet<Rc<ArcType> >

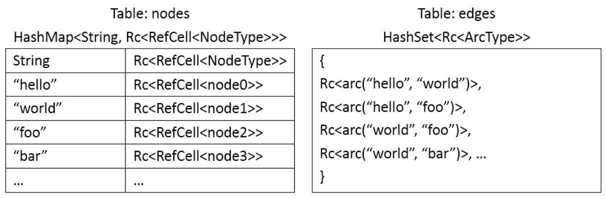

Table: nodes
HashMap<String, Rc<RefCell<NodeType>>>

String	Rc<RefCell<NodeType>>
“hello”	Rc<RefCell<node0>>
“world”	Rc<RefCell<node1>>
“foo”	Rc<RefCell<node2>>
“bar”	Rc<RefCell<node3>>
…	…

Table: edges
HashSet<Rc<ArcType>>

```
{
Rc<arc(“hello”, “world”)>,
Rc<arc(“hello”, “foo”)>,
Rc<arc(“world”, “foo”)>,
Rc<arc(“world”, “bar”)>, …
}
```

Figure 7: Internal data structure of the fifth iteration.

In this representation, a node is uniquely identified by its name, in a String type. The HashMap "nodes" maps a node's name to a smart pointer to a node instance. Each node in the graph is represented by a NodeType structure, whose internal data contains three fields,

- name: String
- into: HashSet<Rc<ArcType> >
- goto: HashSet<Rc<ArcType> >

where "name" field is the same as the key field of "nodes" HashMap, "into" field is a HashSet of smart pointers to incoming edge instances and "goto" field is a HashSet of smart pointers to outgoing edge instances. The reason why a NodeType instance keeps redundant "name" information is to provide a shortcut to get its key in the "nodes" HashMap. When the author was implementing get_nodes method, he realized that a RefCell is not allowed to implement its Hash trait, therefore he decided to use node keys to represent a node, instead. The HashSet "edges" contains smart pointers to edges in the graph. Each edge in the graph is represented by an ArcType structure, whose internal data contains three fields, as well,

- weight: i32
- start: Rc<RefCell<NodeType> >
- finish: Rc<RefCell<NodeType> >

where "weight" field is the weight of this edge, "start" field is a smart pointer to the start node of this edge and "finish" field is a smart pointer to the finish node of this edge.

Why is a node represented by a smart pointer of Rc<RefCell<T> > type to a NodeType instance, but an edge represented by a smart pointer of Rc<T> type to an ArcType instance? Should the smart pointer's type be the same? Let's first explain the difference of Rc<RefCell<T> > pointer and Rc<T> pointer. Rc<T> pointer enables multiple immutable accesses to share the same memory resource [6]. This is not a "borrow", but more like an "own". All of these smart pointers own this memory resource. Once a clone occurs on the Rc<T> pointer, the reference count is incremented by one. Once all copies of Rc<T> expire, which means reference count becomes zero, this block of memory resource is released. However, Rc<RefCell<T> > pointer not only allows multiple immutable accesses to data, but also permits one "mutable borrow", which means one and only one of the copies is able to modify the interior data with type T. This pattern is brought by RefCell<T>, called interior mutability pattern [7]. Additionally, this mutability pattern uses run-time checker instead of the compile-time checker [7].

Then, let's analyze the similarity and difference between nodes and edges. First is similarity. Both of a node and an edge must allow multiple pointers to share its data, since a node may have multiple incoming and outgoing edges and an edge is shared by two nodes as its start and finish. Then is difference. An edge, after its declaration, will never be changed because its start node and finish node is fixed and this edge is uniquely determined by these two nodes. However, a node, after its declaration, should be able to change, because there may be more edges added to this graph and connected to this node. In this situation, an edge should maintain immutability and a node must keep mutability. In consequence, we make use of Rc<T> type to represent an edge and Rc<Ref<T> > type to represent a node.

According to the above analysis, the author started to implement the graph library, but stops at add_edge method and he found that it is impossible to implement this method with such data structure and representations. Let's look into the issues in an add_edge method. To better understand what happens in this method, Figure 8 is shown to illustrate the relationship between nodes and edges.

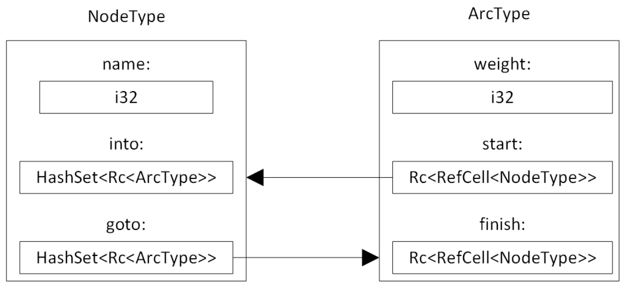

Figure 8: Relationship between nodes and edges.

As shown in Figure 8, a node and an edge are mutually connected to each other. Let's denote them as node N and edge E respectively. Node N is the start node of edge E. Node N contains a HashSet of smart pointers to edges, and one of these pointers is pointing to edge E. Edge E contains two smart pointers to nodes, the start pointer points to node N. Assume node N is already added into the graph, and then the author wants to add edge E. First, he creates edge E with a smart pointer to the start node and another smart pointer to the finish node. Then, he wants to add the smart pointer, which points to edge E, to its "goto" field so he mutably borrows node N and tries to add this pointer. At this time, the program shows it is panicked at twice mutable borrow. After a careful scrutiny, the author understands the cause of this panic. Once node N is mutably borrowed, edge E is mutably borrowed at the same time because edge E "owns" a pointer to node N's data. In this sense, edge E can never be added to node N's "goto" field.

In this example, we see that a mutable borrow of an instance makes all other instances, which have a smart pointer to it, mutable borrowed and cannot be used until the mutable of this instance expires. The language doesn't allow us to make a loop with pointers directly pointing to each other. As we have seen, it is difficult to make a reference circle in Rust, because the reference count of each item in the cycle will never reach 0, and the values will never be dropped [8]. However, is it possible to create reference circles? The answer is yes, but this may lead to memory leak. In this case, the way to create a reference circle is to shrink scale of the range of the RefCell<T>. In other words, we can simply remove the RefCell wrapper on NodeType and add this RefCell wrapper on the inner HashSet inside NodeType, as the example described in the Rust documentation [8]. It is not wise to create reference circles in a specific implementation, although turning a Rc<T> into a Weak<T> could circumvent this![8]. Due to the limit of content, the author does not want to expand the details, he instead decides to break the circle with another HashMap, which brings about the sixth iteration.

2.6 The Sixth Iteration

Since the author has known the reason why he cannot form a reference circle between nodes and edges, the only problem that remains is to break the circle with an appropriate data structure. This time, the author decides to apply a HashMap to edges, using key-value mapping to achieve his goal. The graph is still represented by a star representation but with a HashMap of "edges".

- nodes: HashMap<String, Rc<RefCell<NodeType> > >
- edges: HashMap<String, ArcType>

In the HashMap, the key (in String type) generated from the start node and finish node of the edge, which uniquely determine this edge. In this representation, there is no smart pointer pointing from a node to an edge, but a key associated with the edge, so storing only one copy of edges is enough. The Rc<ArcType> is no longer needed. Instead, only ArcType is used. Hence, each node is represented by a NodeType structure with three internal data fields.

- name: String
- into: HashSet<ArcType>
- goto: HashSet<ArcType>

Similarly, each edge in the graph is represented by an ArcType structure. The only difference is that there is one more "nickname" field, which is exactly the same as its key in the "edge" HashMap. This redundancy is the same as a NodeType, to provide a shortcut to its key in the HashMap.

- nickname: String
- weight: i32
- start: Rc<RefCell<NodeType> >
- finish: Rc<RefCell<NodeType> >

Again, an example of this representation is shown in Figure 9. Two HashMaps stand for "nodes" and "edges", respectively. After this modification, let's look into the implementation of add_edge. Firstly, the author initialized an edge with two pointers, one pointing to the start node, the other pointing to the finish node. Then the author mutably "borrowed" the start node and add the key of that edge into his "goto" HashSet. This time, the node finds its outgoing edges by a HashSet of keys in String type, instead of edge pointers, and then look up the edge instance by its key. Hence, reference circle does

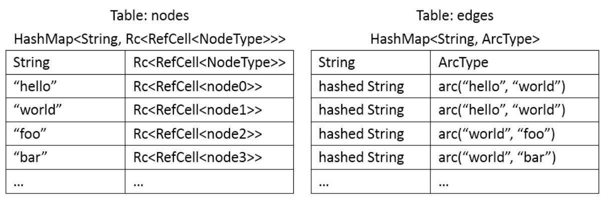

Table: nodes

HashMap<String, Rc<RefCell<NodeType>>>

String	Rc<RefCell<NodeType>>
"hello"	Rc<RefCell<node0>>
"world"	Rc<RefCell<node1>>
"foo"	Rc<RefCell<node2>>
"bar"	Rc<RefCell<node3>>
...	...

Table: edges

HashMap<String, ArcType>

String	ArcType
hashed String	arc("hello", "world")
hashed String	arc("hello", "world")
hashed String	arc("world", "foo")
hashed String	arc("world", "bar")
...	...

Figure 9: Internal data structure of the sixth iteration.

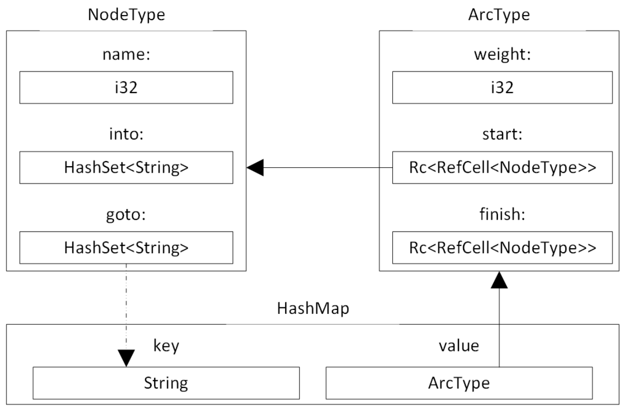

Figure 10: Relationship between nodes and edges.

not exist in the graph any more. The current relationship between nodes and edges is shown in Figure 10 Now there is no problem with reference circles, so the author implemented other methods in the same way as the third iteration. The graph library is now equipped with the same interfaces as the third iteration, but in a different representations and internal data structures.

2.7 Default behavior and Testing

To implement a graph library, the default behavior of improper operations on the graph should be considered, such as try to add the same node twice or try to get an nonexistent node. Usually, we have the following three solutions.

- Throw an exception
- Return an option or a bool value
- Ignore

The first solution is the most strict which will stop the whole program. The second one is relatively mild since only improper manipulation on a returned None type can cause program crash. The third one is the least strict since it ignores the improper operation itself. Referring to the default behavior of other graph libraries, the author developed this graph library in a combined strategy. All "get" methods return an option to let user determine what to do next, but all other methods throw exceptions which makes it faster to discover logic bugs.

In each iteration, the author first made unit tests to make every method work as expected, and then created main function to compile the program to a binary executable and tested it against valgrind to make sure no memory leak happens.

3 EXPERIENCE AND LESSONS

In this section, the author wants to talk about his experience and lessons of developing a graph library with Rust as well as some wired issues he has met, to help other new developers get started as soon as possible.

The author learns Rust almost from zero except the content covered in class. Learning a new language always takes time. New grammar, new collections, new concepts (such as ownership, lifetime and trait) and writing unit tests are all challenges to novice. Thanks to the detailed documentation of Rust, this becomes not so terrible. However, some patterns that Rust should behave as other languages differ from these language, which may bring us more confusion.

The first is that Rust doesn't support function overloading. You cannot overload a function with the same name, but you can imitate this behavior with trait. Define traits for function input and output and then implement these traits for intended types or structs, as suggested in this blog article [9].

The second is one issue that the author has not solved yet. Rust does not implement and does not allow to implement Hash trait for RefCell<T: Hash>, even if you have already implemented Hash trait for T. This behavior is wired because the Hash trait is implemented for Rc<T>. These two smart pointers exhibit inconsistency in supporting traits. This is the reason why the author implemented neighbors method with a HashSet of String, but not a HashSet of nodes, in the fifth and sixth iteration.

The third is about safe behavior that is not supported in compile-time. We have a mutable variable "var", and a mutable reference "x" to "var" and then an immutable reference to "x". We want to modify the value of "var". Let's read the following code.

```
let mut var = 5;
let x = &mut var;
let y = &x;
*(*y) += 1;
```

This paragraph of code does not work. The error message is "assignment into an immutable reference". This operation is regarded unsafe by Rust compiler. The way to make it work is to turn "x" into mutable and turn "y" into a mutable reference of "x". We have to make "x" mutable, although we don't modify "x" itself.

```
let mut var = 5;
let mut x = &mut var;
let y = &mut x;
*(*y) += 1;
```

This is the way Rust think safe. In an "unsafe" way, we can use raw pointers combined with "unsafe" key word to achieve this, for example,

```
let mut var= 5;
let x = &mut var as *mut i32;
let y = &x as *const *mut i32;
unsafe{ *(*y) += 1; }
```

This time "y" references an immutable "x", which seems more reasonable, but this is regarded "unsafe" in Rust. It is said this safe feature is related to the implementation of Rust, but obviously this feature makes the language not so flexible. Besides, there is another way to achieve this with Rc<RefCell<T> >, utilizing its interior mutability. The author uses the following code to simulate the above behavior.

```
let var = Rc::new(RefCell::new(5));
let x = &var;
let y = &x;
*y.borrow_mut() += 1;
```

This works but it passes the checker from compile-time to run-time, still not safe as Rust expects. With the limit of time, the author is unable to finish exploring the "unsafe" mechanism of nesting immutable reference and mutable reference, but he thinks that there is still room to improve. In addition, the above code is assisted with automatic referencing and de-referencing [10]. Variable "y" is de-referenced twice to "var" and then calls "borrow_mut" and de-referneced again to access the value of "var".

Then, the author wants to talk about lifetime issues in Rust. Look into the simple code below

```
let mut hashmap = HashMap::new();
let var = "hello";
hashmap.insert(var, 10);
let tmp = hashmap.entry(var).key();
```

You will get an error after compiling, but what is wrong? The error message shows that we created a temporary value in the fourth line and the values is dropped after the semicolon. After carefully reading Rust documentation, the author understands the reason. Because "entry" method creates a new Entry type variable and "key" method only use the reference of this variable, hence "tmp" cannot lives longer beyond this line. The different behavior of "entry" and "key" generates this error. The fix is pretty simple, transfer the ownership to another variable living longer can solve.

```
let mut hashmap = HashMap::new();
let var = "hello";
hashmap.insert(var, 10);
let tmp1 = hashmap.entry(var).
let tmp2 = tmp1.key();
```

This example tells us that, we need to clearly know the behavior of a method, whether it consumes ownership or not, whether it generates ownership or not, whether it returns a mutable reference or not. Hence, we need to pay more attention to the ownership transferring and lifetime expiration during development with Rust.

Finally, the author wants to talk about logs of Rust. Rust is the most friendly language in the aspect of error message. After the compile, you can see that the compiler pinpoints your error with exact lines and character offsets with different marks. What is more important is that it provides you the detailed reason why your program fails and sometimes even a suggestion of modification. This makes the development process becomes smoother without too much frustrations. This is a great advantage of Rust over other languages. The disadvantage is you cannot get such detailed information, when you pass the checker from compile-time to run-time. For example, when you use RefCell<T> smart pointer or "unsafe" key word, everything goes back to C or C++. You have to find the error by yourself. However, this is still an improvement over other language, because if you make a mistake and get an error related to memory safety, you will know that it has to be related to one of the places that you opted into this unsafety [11]. This significantly reduces the search range during debugging. As the documentation suggested, restrain the superpower of unsafe operations, unless you are confident enough or you have to deal with those low-level programmings with interfaces of operating systems.

4 CONCLUSION

In this report, the author explored usage of Rust language by iterations of implementing a Rust-based graph library. Through six iterations, the author tried three different graph representations, adjacent list, adjacent matrix and star representation. In each representation, the author made effort to alter internal data types/structures to discover memory management issues and find solutions to circumvent them. The author explained in detail the design of different implementations of the graph library and tested these implementations in each increment. In addition, the experience , lessons and insights obtained from the development is also provided to people who are unfamiliar with Rust as reference.

5 FUTURE WORK

Based on the work achieved in this project, the author thinks that the future work should focus on improvements of this graph library. The first thing to do is to optimize the usage of reference and clone. Clone is an expansive operation and what the author wants to do is to use reference as much as possible, and replace the occurrences of clone as much as possible. This is a good way to performance improvement. The second thing to do is to refine default behavior of methods, since current implementation makes use of exceptions to pinpoint logic bugs. These exceptions are unfriendly to new users and may let them feel discontinuity during development with this library. The last thing to do is related to the author's interest. He wants to try to make reference circles to truly simulate the behavior of objects and break the circle with Weak<T> pointers. This should be an exciting experiment with Rust!

ACKNOWLEDGMENTS

The author would like to thank Lecturer Will Crichton for delivering interesting and vibrant introductions to Rust language, and TA John Wang Clow for giving insightful critiques on the author's proposal and helpful feedback on the author's checkpoint. Thanks for all teaching staff's hard working on providing such wonderful charms of Rust language to the author.

REFERENCES

[1] https://www.rust-lang.org/en-US/
[2] https://www.rust-lang.org/en-US/faq.html
[3] https://doc.rust-lang.org/book/first-edition/unsafe.html
[4] https://doc.rust-lang.org/book/second-edition/ch04-00-understanding-ownership.html
[5] https://doc.rust-lang.org/book/second-edition/ch04-01-what-is-ownership.html
[6] https://doc.rust-lang.org/book/second-edition/ch15-04-rc.html
[7] https://doc.rust-lang.org/book/second-edition/ch15-05-interior-mutability.html#refcellt-and-the-interior-mutability-pattern
[8] https://doc.rust-lang.org/book/second-edition/ch15-06-reference-cycles.html#reference-cycles-can-leak-memory
[9] https://medium.com/jreem/advanced-rust-using-traits-for-argument-overloading-c6a6c8ba2e17
[10] https://doc.rust-lang.org/book/second-edition/ch05-03-method-syntax.html
[11] https://doc.rust-lang.org/book/second-edition/ch19-01-unsafe-rust.html