Memory Management and Efficient Graph Processing in Rust

KEVIN CHEN and ELBERT LIN
Stanford University

1. INTRODUCTION

Graphs are a common data structure in most object-oriented programming languages due to their natural representation of many real-world datasets, but they are also somewhat challenging to implement in the memory-safe language Rust. In this paper we discuss the different approaches we took in writing a graph library and analyze the tradeoffs of each approach.

2. OVERVIEW

Rust is a systems programming language designed to have memory-safe abstractions, but not sacrifice performance in the process (as many compiled and interpreted programming languages do). In this project, we wanted to focus on how the language allowed us to build a performant graph library comparable to one implemented in any low-level programming language, but also allowed us to do so in a relatively ergonomic and typesafe way.

Constructing graphs in Rust is quite challenging due to Rust's strict lifetime and mutability constraints. In most languages we can rely on the garbage collector to collect objects that we no longer care about, but Rust forces us to take into consideration the *lifetime* of our graph objects. In most cases the lifetime of values can be inferred (e.g. when they go out of scope), but the problem is a little more subtle for the case of graphs, in which many objects hold valid pointers to many other objects. *Mutability* is another concern; Rust tries to favor immutable objects so that pointers remain valid, whereas graphs should be able to change at a moment's notice. Thus, there are two main problems that we must solve in order to implement graphs:

(1) How do we track the lifetime of a graph?
(2) How do we track the mutability of a graph?

To solve the first we must essentially determine a method of linking to other nodes within the graph. As graphs are often recursive, we cannot use a value-based structure and must use pointers. One might think to use `Box<...>` as pointers as is typically used for trees or linked lists in Rust, but graphs can be cyclic, and Rust does not allow cyclic ownership.

To solve the question of mutability, we need to consider that graphs must be, at the very least, mutable during initialization. Since graphs are allowed to contain cycles, we cannot simply create the graph using a single statement. Additionally, multiple edges may lead into or out of any node, and must be mutable when initialized as well, which means that edges are not guaranteed to be unique. Because Rust mandates that pointers be unique or immutable, we must use some more advanced techniques to maintain the mutability of the graph.

The simplest but most dangerous way to implement graphs is to use mutable raw pointers, as we would in an unsafe language like C++. With raw pointers, we can control the lifetime and mutability over all nodes and edges; however, this largely defeats the purpose of using a safe language like Rust. Thus, we opted to explore three other methods of memory management:

(1) reference counting
(2) arena allocation
(3) complete ownership and indexing

The implementation and efficiency of each of these approaches are discussed in the next sections.

3. APPROACH

We present three different methods for memory management while implementing graphs in Rust. For simplicity, all our implementations are of directed graphs, allowing parallel edges and cycles to be added. The graphs are all represented using adjacency lists. As a small twist, we allow both nodes and edges to contain data that is specified by the user at compile-time. We will first outline how each one works, then analyze the problem that each solves, and finally discuss the issues that each faces. All graphs were created and tested using a subset of the Twitter social graph data collected by [Kwak et al. 2010].

3.1 Reference Counting

Reference counting is a method that is often used for memory management purposes. Here it provides a way of skirting Rust's invariant of unique ownership, instead giving shared ownership to a value. This allows us to link nodes together and ensure that nodes do not contain dangling pointers. On the other hand, it introduces a small problem with Rust, because values can only be borrowed immutably when references are shared. In order to reintroduce mutability in something immutable, we must use cell types. Specifically, we have chosen to use `RefCell` with reference counting, as it is safer and defers responsibility of following borrow rules to Rust.

To implement reference counting, we use `Rc`, Rust's single-threaded reference-counting pointers, which provides shared ownership of values on the heap. We also use `RefCell`, taking advantage of Rust's built-in mutable memory location with dynamically checked borrow rules.

We have a `struct RcGraph` that holds two vectors, one of reference-counted pointers to `struct Nodes` and another of also reference-counted pointers to `struct Edges`. Each `Node` has a `data` field of generic type `TNode`, and two vectors of reference-counted pointers to `Edges`, one for `Edges` leading to predecessors and one for successors. Similarly, each `Edge` has a `data` field of generic type `TEdge`, but rather than two vectors it has two fields `start` and `end` that are single reference-counted pointers to its start and end `Nodes`, respectively.

To add a node, we simply create a node and add the necessary wrappers around it:

```
let node = Rc::new(RefCell::new(Node::new(data)));
```

We then clone this reference (incrementing its count) and push it onto the graph's `nodes` vector. To add an edge, we similarly create and push the edge onto the graph's `edges` vector, but given the start and end nodes we must first clone those pointers to give to the edge. Once we have the edge, we must now clone it as well, and push it onto the start and end nodes' vectors of successors and

predecessors, respectively. To do these last two pushes, we need to borrow the nodes mutably. For example,

```
start_node.borrow_mut().successors
  .push(edge.clone());
```

Using reference counting with Rust's built-in `RefCell` allows us to maintain memory safety (there is no `unsafe` code whatsoever), and also provides a fair amount of flexibility. The graph is fully mutable, and nodes can easily be reused independently outside of the graph. We let Rust track lifetimes for us through reference counts, and also let Rust track mutability dynamically, thus taking full advantage of the safety guarantees provided by the language.

This comes at a cost of efficiency and ergonomics, however, as Rust needs to check references and mutability requirements at runtime rather than compile time. Users must also borrow references rather than operating on them directly. There is also a very large risk of leaking memory when using reference counting. If cycles are introduced, then counts may never fall to zero and the graph would never be deallocated. One method to avoid this would be to use `Weak` pointers, though it would mean that references would need to first be `downgraded`, and then `upgraded` whenever the node or edge was needed. Another method, which we (unsuccessfully) attempt in our code, is to manually break cycles by dropping the references whenever the graph goes out of scope, but this requires extra work that other methods may be able to avoid.

3.2 Arena Allocation

Many of the approaches (such as reference counting) have the flaw that it is difficult to reason about the lifetimes about certain objects. For instance, reference counting still allows for the subtle error of maintaining undesirable references in the case of cycles. In addition, reference counting can often be slow because each object has to keep track of a atomic reference counter.

3.2.1 *Lifetimes.* Using memory arenas for graph allocation is motivated by the idea that everything in the graph should have the same lifetime. Most real-world applications maintain the graph, along with its edges and nodes, as a single unit; there are very few cases where you might want to drop a large number of nodes all at once. This means that when we need to deallocate the graph, we can drop it all at once. The natural and efficient way to do this is to allocate a large "arena" of memory to store all the graph objects into. Then, deallocation is basically free; all of the memory is marked as freed at once, so this can be as simple as a pointer move. Furthermore, we get the benefit of cache locality since all graph objects that might be linked would likely be colocated during the initial graph creation.

However, this implies that we can't drop individual nodes or edges; we can allocate additional memory as necessary, but deallocation involves deallocating everything all at once. This makes it a little trickier to perform methods like `remove_node`; we have to take the node out of the graph without actually deallocating the memory (or invalidating any references). One method that could be used to resolve this would be to set a `valid` flag on the graph objects; this delegates the responsibility of determining if a node or edge is still valid to the consumer of the graph.

We initially considered Rust's `Vector` type, which can contain many objects of the same type (and more importantly, lives on the stable branch). However, Rust prevents us from using references into a vector because the vector can be reallocated without warning, so we were unable to use this approach. We instead made use of Rust's `TypedArena`, which efficiently allocates contiguous chunks of memory that are aligned to the size of the allocated objects. More importantly, references into the arena never change, so Rust allows us to hand these references out to be used to access the actual object (so long as the references are never used past the lifetime of the graph).

Ergonomics-wise, we found it significantly more verbose to express lifetimes when using the `TypedArena`. We needed to explicitly indicate to Rust that all edges and nodes (along with their associated metadata) contained within the same graph have the same lifetime. That is, when the graph is dropped, the lifetimes of the node and edge arenas (and the objects contained within) should come to an end as well. This meant that we had to annotate all of our objects with explicit lifetimes; for example, we had to bind the lifetime of the start and end node to the directed edge struct as follows:

```
pub struct Edge<'a, TEdge: 'a, TNode: 'a> {
  data: TEdge,
  start: &'a Node<'a, TNode, TEdge>,
  end: &'a Node<'a, TNode, TEdge>,
}
```

This ended up causing us a lot of difficulty when working out how to implement actual graph methods, which also had to take in explicit lifetime parameters.

3.2.2 *Mutability.* In the reference-counted version of our graph, we implemented mutability by using `RefCell`. This time, we opted to use `UnsafeCell` for efficiency (since we're already using an efficient arena). This is the only legal way to obtain a mutable pointer to the vector of edges:

```
UnsafeCell<Vec<&'a Edge<'a, TEdge, TNode>>>
```

Rust also forces us to enter an unsafe block every time we want to add an edge:

```
unsafe {
  (*start_node.successors.get()).push(edge);
  (*end_node.predecessors.get()).push(edge);
}
```

This completely bypasses the Rust type-safe compiler and brings us closer to an implementation we would see in an unsafe language.

3.3 Indexing

Indexing provides a closer fit to Rust's ownership model. We can maintain unique ownership by allowing the graph to own all the data it contains, and let other processes access that data with `Copy`able indices rather than borrowed references. Lifetime and mutability are all managed through the graph data structure, which can be viewed as one single unit, just like any other data structure. This means that Rust can easily track the graph as a whole, and ensure that no borrow or mutability rules are violated.

With this method, rather than using pointers (either directly or borrowed) we instead use integer indices to indirectly represent pointers to nodes and edges. We have chosen `i64` as our index type, which assumes that graphs will not contain more than 2^{64} nodes or edges (as you will most certainly run out of memory space needed to hold all that data long before reaching that many). To prevent users from providing invalid indices, we alias the type to `TIndex`.

Like before, we have a `struct IdxGraph`, a `struct Node`, and a `struct Edge`. As we are using integer indices, we can actually use `HashMaps` rather than vectors, since equality and hashing are supported for integers. Thus, the `IdxGraph` has two `HashMaps`, one for indices to `Nodes` and one for indices to `Edges`, in addition

to an index counter `cur_idx`, the `Node`s have their `data` and two `HashMap`s of indices to `Edge`s leading to predecessors and successors, and the `Edge`s have their `data` and two fields `start` and `end` that are indices to that edge's start and end `Node`s.

To add a node, we create a new node and assign to it the graph's current index, then insert the node into the graph's `nodes` `HashMap`, keyed by the index. We then increment the index. There is no need for any clones here, as the integer index is inherently `Copy`able, and the `Node` remains entirely within the graph's ownership. Similarly, to add an edge we create it and assign it to the current index, insert it into the `edges` `HashMap`, and increment the index. We also then (mutably) obtain the nodes from the graph with the indices given for the start and end nodes, and insert the edge, keyed by its index, into the nodes' `HashMap`s successors and predecessors, respectively. Once again, no cloning is needed. Example code for getting and inserting the edge index into a node is shown below:

```
self.nodes.get_mut(&start_node).unwrap()
  .successors.push(edge_idx);
```

As we can see, implementing the graph using indices is straightforward and compact, as we are working entirely within Rust's lifetime and mutability rules. The graph structure contains and owns all the data related to itself, only allowing users to mutate it through `&mut self` methods in the graph, with indices indicating which portions of the graph to modify. This unique ownership also assists lifetime elision, as all the nodes and edges only live as long as the graph itself lives.

On the other hand, giving the graph unique ownership reduces the flexibility of this implementation, as all methods must be done through the graph. Nodes cannot be used outside of the graph, which would make operations such as graph union difficult. Also, there is a slight risk of indices being reused or impersonated, though we have attempted to eliminate this risk by aliasing with an associated type. Had we used vectors, we would have ran the risk of dangling indices as well, where removal of nodes could cause indices to point to empty locations. With `HashMap`s, however, we can simply check if the index exists as a key, and if not we simply return `None`.

4. METHODOLOGY

All benchmarks were run on a Lenovo x220 laptop running Ubuntu 17.10, with a 2.60GHz 4-core processor and 8GB of RAM. Our binaries were configured at the highest optimization level and using the system allocator as described below. Note that our results might differ greatly from if they were run on a dedicated server with the same specifications, since background processes and power throttling may have contended with our binary's CPU time.

We randomly selected a sample of 1,000,000 edges and 1,257,934 nodes from the Twitter dataset and constructed a graph, then allowed the graph to go out of scope and the program to terminate. Because we took a random sample from a much larger data set (to allow the graph to fit into memory), the graph ended up being a lot more sparse than we would have liked. However, there were plenty of cycles and the sparseness served as an interesting pathological example for our adjacency list implementation.

4.1 Memory Leaks

Although Rust is memory-safe, memory leaks can still occur. Rust only prevents us from using invalid memory; it doesn't prevent us from not using (or freeing) valid memory. However, Rust does make an effort to limit the number of places where this can occur - usually in `unsafe` blocks, native C interops, or Rust constructs that directly leverage `unsafe` under the hood.

We initially attempted to use LLVM/Clang's `LeakSanitizer` and `MemorySanitizer` to check for memory leaks and uninitialized memory. These sanitizers are supposed to be much faster than Valgrind at finding memory leaks; however, we weren't able to get our code (or their example snippets) to sanitize on our basic setup.

We turned back to the classic Valgrind to check our implementation for memory leaks, and again to perform basic heap profiling. We ran into significant issues on this front - Rust stable uses a build of `jemalloc` that no longer supports Valgrind profiling. Instead, we had to switch over to the nightly Rust toolchain and utilize the `alloc_system` crate in order to compile our program using the native system's `malloc` and `free`.

We found that of our three initial implementations, the reference-counted implementation was the only one that leaked memory. This was likely due to improper deallocation in the case of cycles, but we didn't have time to investigate further.

4.2 Memory Usage and Efficiency

To profile heap usage, we used the Massif tool included with Valgrind. This allowed us to determine when and where memory was being allocated. This tool generates snapshots of memory usage in memory-based intervals, so we were able to extract a time-series graph of how much memory was being used and reallocated. Massif also gave us enough information to determine where in our code the memory was being allocated, but we didn't use this information for any significant purpose.

5. RESULTS AND DISCUSSION

We were sadly unable to prevent memory leaks in our reference-counted implementation, but the other two implementations worked as intended. We had attempted to break all cycles by manually and recursively dropping all references, but it appears that we may have missed some edges cases. As such, our reference counting would not scale well to larger graphs in a world where nodes are repeatedly dropped and added.

We can see some interesting trends to the figures obtained through Massif. The arena allocation implementation was the fastest, then reference counting, then indexing. The reference-counted implementation is likely slower due to the use of `RefCell` over `UnsafeCell`, as checks must be made during runtime about a node's mutability. The index implementation takes nearly twice as long to process the graph, which we hypothesize is due to using `HashMap`s rather than vectors, where the overhead of look-up and insertion is more than that of pushing onto a vector.

We also note that the arena allocation version had the lowest total memory heap consumption, then reference counting, then indexing. We see that both the arena and index implementations do not reallocate, due to how Rust implements `TypedArena`s and `HashMap`s under the hood. They also have some interesting jumps at around the same number of processed nodes and edges (which we can infer from the fraction of current time taken over total time taken). This is likely due to Rust requesting similar amounts of memory once the previous requested space has been filled. On the other hand, the reference-counted implementation must reallocate, when vectors require more space when pushed to. Also of note is that while reference counting and arena allocation use roughly the same amount of memory, indexing used almost 100MiB more. We attribute this to our use of `HashMap`s, though we have not reimplemented and retested our index implementation with vectors to confirm this.

While working on our implementations, we noticed that Rust could benefit and improve its ergonomics if certain lifetime and mutability rules were adjusted to fit some common object-oriented situations. For example, we often needed elements to remain mutable while we initialized them while keeping multiple references to those elements, but later we no longer needed to modify any data, and could return to normal Rust behavior. If some solution or compromise could be achieved, objects could be more freely initialized while remaining safe.

In addition, our graphs essentially all had one lifetime tied to the main graph structure. With current Rust requirements, while it was not an issue with the others, we had to repeat the lifetimes for our arena allocation implementation. It would greatly improve the ergonomics of the code if somehow the lifetime could be tied to the module, rather than having to indicate to the Rust compiler each time for each method we wished to implement within that module.

6. FUTURE WORK

In our experiments, we initially wanted to run our graph library over the entire dataset. This would've given us a much better idea of how our implementations would hold up against real-world data (since our sampled dataset was a lot more sparse than it should've been). However, cloud instances that could hold the dataset in memory were expensive and development cycles would've been slower, so we opted to just run the examples on our local machines.

Furthermore, all of our implementations varied wildly in terms of implementation. We would've liked to have a more controlled environment, and only changed one aspect at a time when running tests. However, given the number of implementation details we wanted to explore, we just didn't have the time to go through all the possible combinations of features. We settled on just three for this paper, but it would be interesting to explore the others.

Finally, we initially wanted to create a library with a common API and different backends, but the different implementation constraints made this extremely difficult. For example, accessing nodes in the index-based graph involved taking your ID and requesting the node data from the graph, whereas the arena-based graph just used a dereferenced pointer. We decided that it was best to have a separate API for each graph implementation, but it might be worth revisiting this decision in the future.

7. CONCLUSION

Each of the methods presented have their own advantages and disadvantages that arose as a consequence of Rust's lifetime and mutability requirements. Reference counting leaked memory, but maintains memory safety and mutability requirements through Rust itself. Arena allocation was the most efficient, but required explicit lifetimes and the responsibility of the programmer to ensure that all code remains safe. Indexing allowed us work within Rust's lifetime and mutability rules without using any fancy tricks, but turned out to be the least efficient method.

When we started this project, we found that Rust was more difficult to use than any other language we had used previously. Ownership and moves were foreign concepts to us, and we were constantly fighting the Rust compiler. Towards the end of the project, however, we realized that we were fighting the compiler less and less, likely due to a deeper understanding of ownership and lifetimes. Although we acknowledge that Rust has a steep learning curve, it was almost fun to use towards the end.

APPENDIX

A. FIGURES

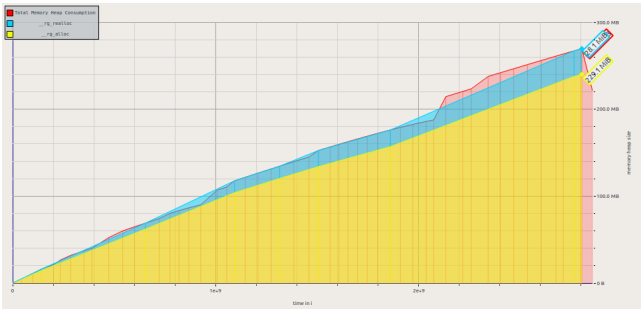

Fig. 1. Reference counting

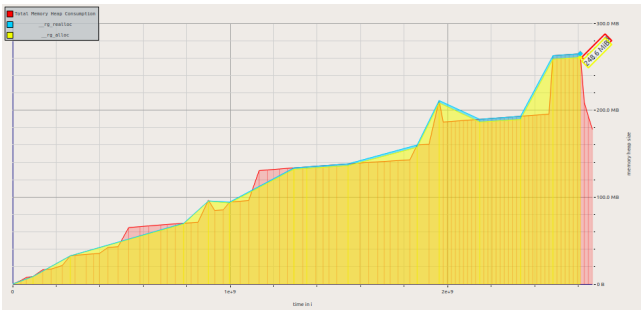

Fig. 2. Arena allocation

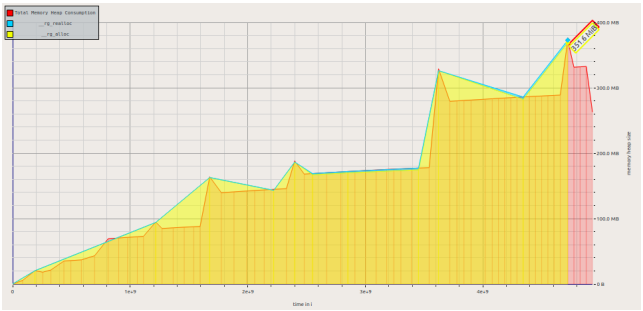

Fig. 3. Indexing

B. CONTRIBUTIONS

Equal work was performed by both project members. All code produced for this project can be found at https://github.com/kvchen/salus.

REFERENCES

Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW '10: Proceedings of the 19th international conference on World wide web*. ACM, New York, NY, USA, 591–600. DOI:http://dx.doi.org/10.1145/1772690.1772751