Chapter 1

Time Stone

1.1 Extracts from wiki

"The Time Stone was one of the six Infinity Stones... ...the user is able to physically control and redirect the flow of time, and can specifically select the exact area to manipulate without affecting those outside its selected range. The stone can alter targets as small as an apple or as wide in scope as the timeline of the universe... ...Due to the selective nature of the Time Stone's power, it could be used to individually alter the timeline of individual objects or events, reversing them to a previous state or sending the object forward into a future state. This occurs regardless of any potential breaches in causality... ...the Stone could send objects forward to a potential future state that does not necessarily have to occur in the current timeline."

1.2 Abilities

- Can alter an arbitrary space.
- Can reverse such a space to a previous state.
- Can send the space forward into a future potential state (that does not necessarily have to occur in the current timeline.)
- Does not cause issues in causality.

1.3 Definition and Consequences (1st Attempt)

A TS is a tuple T = (R, q_{end})

- R is a deterministic run of some automata which is assumed to halt.
- q_{end} the final step of the run.

When activated, the TS alters the automaton, sending it forward in time to q_{end}. This process is assumed to take no computational power and is a constant $O(1)$ in time and memory complexity.

We define a new MCU complexity class, which in this case defines the number of times the TS must be activated.

1.3.1 $P_{TS} = EXPTIME_{TS}$

Given an arbitrary TM in P, we activate the TS, sending it forward in time to q_{end}. Therefore, its running time has been reduced to $O(1)$.

Given an arbitrary TM in EXPTIME, we activate the TS, sending it forward in time to q_{end}. Therefore, its running time has been reduced to $O(1)$.

Clearly, $P_{TS} = EXPTIME_{TS}$.

1.3.2 Halting Problem

In our formal definition, we state the given input TM is assumed to halt. However, this is a very weak assumption to make.

Let's attempt to bypass this requirement and instead see what happens when we run the TS on a TM which does not necessarily halt.

When activated, the TS alters the TM, sending it forward in time to q_{end}. This process is assumed to take no computational power and is a constant $O(1)$ in time and memory complexity.

Clearly if the TM does halt, it is sent forward in time to q_{end} as normal. However, what would occur if the TM was an infinite loop? Answering anything other than "the TM is sent forward in time to q_{end} as normal" inherently contradicts the halting problem, as if we define this new behaviour in any way that differs from the original, as a consequence we have created a device which determines if a given TM will halt.

Therefore, this definition cannot exist.

1.4 Definition and Consequences (2nd Attempt)

A TS is a tuple T = (R, c)

- R is a deterministic run of some automata which is assumed to halt.
- c is a constant multiplier, and accepts the input language of $\mathbf{N}$.

When activated, the TS alters the automaton, speeding up its operation by a factor of c. This process is assumed to take no computational power and is a constant $O(1)$ in time and memory complexity.

1.4.1 $P_{TS} = NP_{TS}$

We simulate an arbitrary non-deterministic TM that is assumed to halt on a deterministic TM. [1]

The non-deterministic TM can be represented as a tree of non-deterministic choices, where a TM running a BFS simulate non-deterministic behaviour deterministically by iterating through each node at every level of the tree until the TM halts.

As we traverse the BFS tree, the number of nodes at each level increases by a upperbound factor of n, where n is the maximum number of non-deterministic choices at some step in the non-deterministic TMs code.

At each level h of the tree, if we activate the TS to speed up the TM by a factor of $n * h$, clearly, we will run through the entire level less than or equal to the time it would take to run a single step.

Therefore, it will now run in the same time as a true non-deterministic TM.

Therefore, $P_{TS} = NP_{TS}$.

1.4.2 Halting Problem

We get any arbitrary TM and modify it such that at every step, an output is given to the TS to double its own multiplier speed. I.e., the first step will take 1 second to compute, the next step will take 0.5 seconds to compute, 0.25s, 0.125s ...

As we know that $1+0.5+0.25+0.125... \leq 2$, after 2 seconds of computation, we would have ran an infinite number of steps on this TM and will observe two possibilities:

1. The TM will have halted as it has completed some N amount of steps.
2. The TM will continues running as it is in an infinite loop.

Again, the consequence of this concept is we now have a device that determines if a given TM will halt.

Therefore, this definition cannot exist.

1.5 Definition and Consequences (3rd Attempt)

A TS is a tuple T = (M, O)

- M is some TM.
- O is a time complexity factor, and accepts the input language of the set $\{O(1), O(n^c), O(c^n), O(n!)\}$

When activated, the TS alters the TM, dividing its time complexity by a factor of O. This process is assumed to take no computational power and is a constant $O(1)$ in time and memory complexity.

1.5.1 Division Behaviour

The set of time complexities defined above are ordinal, and are defined without loss of generality by the following division rules:

- $O(x)/O(x) = O(1)$
- $O(x)/O(> x) = O(1)$
- $O(x)/O(< x) = O(x)$

1.5.2 $P_{TS} = EXPTIME_{TS}$

Given an arbitrary TM in P, we activate the TS with an input of $O(n^c)$. As all problems in P have a time complexity of $O(n^c)$, $O(n^c)/O(n^c) = O(1)$. Therefore, its running time has been reduced to $O(1)$.

Given an arbitrary TM in EXPTIME, we activate the TS with an input of $O(c^n)$. As all problems in P have a time complexity of $O(c^n)$, $O(c^n)/O(c^n) = O(1)$. Therefore, its running time has been reduced to $O(1)$.

Clearly, $P_{TS} = EXPTIME_{TS}$.

1.5.3 Halting Problem and Limitations

As there is an upperbound time complexity of $O(n!)$, and through our definition we defined $O(x)/O(< x) = O(x)$, any problem which exceeds $O(n!)$ complexity will not be solvable in constant time.

Therefore, it is impossible to determine whether a TM will halt or loop forever, as there is no opportunity for such behaviour to differ. Therefore, our definition is robust in this regard.

1.5.4 $O(\infty)$ Extension

Could we extend the capabilities of our definition without contradicting the laws of the halting problem? Let us modify the input language to accept an additional complexity class $O(\infty)$, such that given an infinitely looped TM, we can activate the TS with an input of $O(\infty)$, which we assume runs the TM an infinite amount of times in constant time, after which it then halts.

This extension does not contradict the halting problem. As a reminder, we only have a device that contradicts the halting problem if the behavior of the device is inconsistent. From our definition we have $O(x)/O(x) = O(1)$ and $O(x)/O(> x) = O(1)$, implying that for both problems that infinitely loop and problems which exceed $O(n!)$ complexity, we experience the same behaviour. Therefore, it is impossible to determine if a TM halts or loops from our definition.

Great, the TS now has the capability to run ALL time complexities in constant time. But not so fast. Of course, what "running an infinite loop to completion" still needs to be clarified.

Defining $O(\infty)$ Completion

Any definition must be consistent with programs which do halt. Therefore, the end result of an infinitely looped sequence must be indistinguishable from those designed to halt.

Let's consider a simple example of an infinitely looped sequence that iterates through the natural numbers in one cell of a TM. We activate the TS, sending it to completion.

Once this process is "finished", we cannot define it such that there is simply a ∞ symbol left behind (as this will determine if the iterator indeed ran an infinite number of times), instead, we would need to be left with an infinitely large digit. Corollary, it is impossible to determine if the output digit is indeed infinite, as when reading the series out, we can always simply say "we don't know if the sequence of numbers is infinite, maybe we just need to read n more digits!" This therefore does not contradict the halting problem, which is a good start.

But what if by extension, we created an additional TM which read this infinite sequence, sped it up infinitely, and once completed, it is instructed to store only the *last* digit. Clearly, this would be possible for a finite sequence of digits, so what behaviour would occur with an infinite sequence?

The question is paradoxical in nature, as there exists no last digit of an infinite sequence. As we now have behavior which is inconsistent, our definition is no longer valid and therefore this extension is invalid.

Bibliography

[1] J. Diligent, “How to simulate non-deterministic turing machines on a deterministic turing machine,” 2020. [Online]. Available: https://www.youtube.com/watch?v=u0t7zJi8Fno