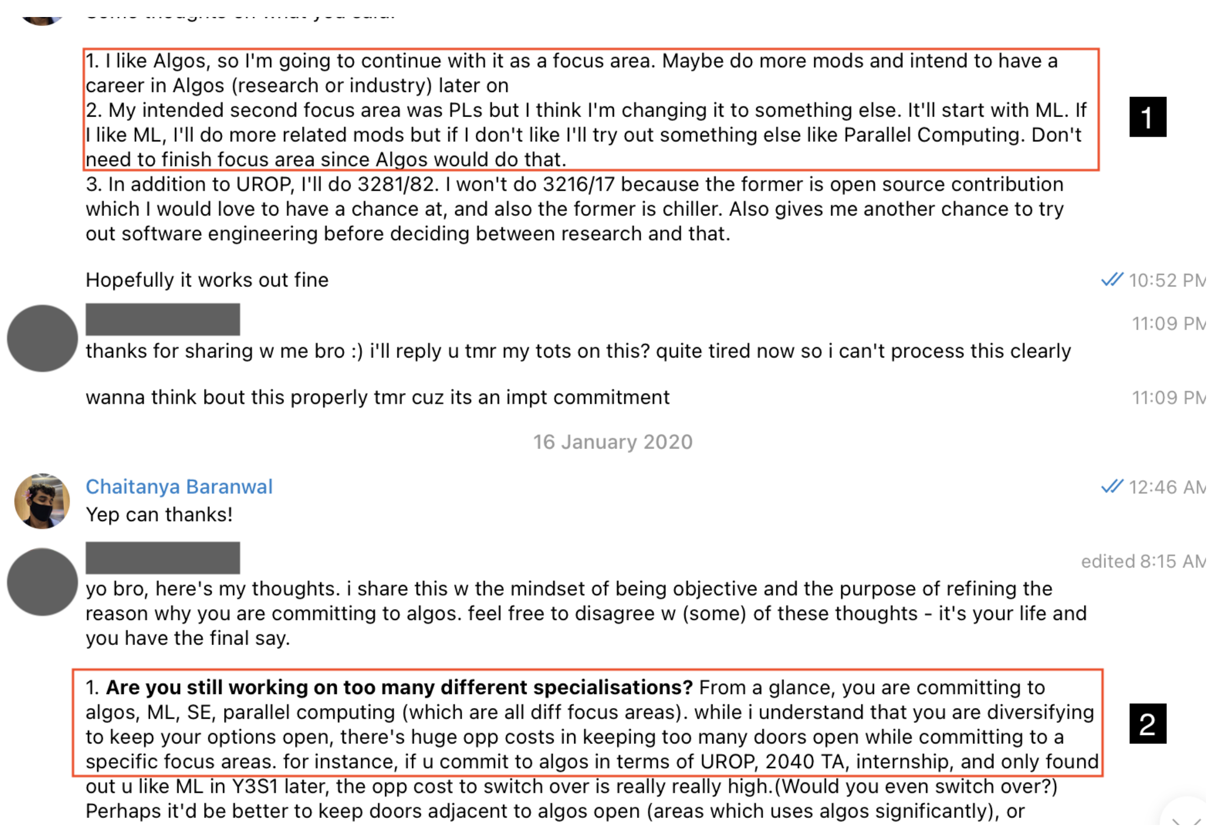
Mastering Contextual Computational Thinking: The Usefulness of a Theoretical Computer Science Education

The question of to what extent my computer science (CS) education is useful feels critical, especially as I join the tech industry after graduation. My 18-year-old freshman-self did not care much about the question – I was just very excited to learn CS in a structured and rigorous manner, hoping to coherently understand the subject. Consequently, I was too focused on intrinsic learning and *mastering technical content*. This fixation on intrinsic learning, coupled with the highly theoretical nature of core modules in the NUS CS curriculum, meant I *pondered little about the real-world applications* of what I was learning. However, I soon realised there exists a tension between theory and what is often required in practice, which made me question the usefulness of my theoretical learning. Reflecting on this tension, I see that learning theory has benefits that surpass the instrumental benefits that seem lacking. Learning theory makes one an effective *contextual computational thinker* – someone with the ability to *learn how to learn* and *apply computational thinking* to real-world contexts.

While theoretical learning can spark fascination for abstract concepts, this can come without a clear idea of its applications and real-word instrumentality. One core module which exemplified my fascination for theoretical learning was *CS2040: Data Structures and Algorithms*. "Data structures" refer to abstract mechanisms to store information, and "algorithms" are formal steps to efficiently solve problems using those mechanisms. I was so drawn to algorithms that I went on to pursue research in a sub-discipline and become a Teaching Assistant for the class. I also finished a specialisation in "Algorithms and Theory", and yet I did not concern myself with how algorithmic techniques apply in the real-world outside academia. Fig 1 accurately summarises my approach towards learning algorithms. It follows a lunch conversation, where my friend said that *I am not reflecting much into my decision to specialise in algorithms and theory, given it is hard to find relevant roles in the industry*. Despite his concerns against focusing on theory, I was intent to "continue with it" because I "like algorithms" (box "1"). My fascination originated from intrinsic value and I did not (could not) counter my friend's argument about the lack of practical relevance. More critically, the conversation got me considering *more industry-oriented specialisations* (boxes "1" and "2") like machine learning and parallel computing, revealing that I felt other fields were more practical than algorithms. This shows that I had not considered the applicability of algorithms in the industry, despite my fascination with the subject.



1. I like Algos, so I'm going to continue with it as a focus area. Maybe do more mods and intend to have a career in Algos (research or industry) later on
2. My intended second focus area was PLs but I think I'm changing it to something else. It'll start with ML. If I like ML, I'll do more related mods but if I don't like I'll try out something else like Parallel Computing. Don't need to finish focus area since Algos would do that.
3. In addition to UROP, I'll do 3281/82. I won't do 3216/17 because the former is open source contribution which I would love to have a chance at, and also the former is chiller. Also gives me another chance to try out software engineering before deciding between research and that.

Hopefully it works out fine — 10:52 PM

[redacted] — 11:09 PM
thanks for sharing w me bro :) i'll reply u tmr my tots on this? quite tired now so i can't process this clearly

wanna think bout this properly tmr cuz its an impt commitment — 11:09 PM

16 January 2020

Chaitanya Baranwal — 12:46 AM
Yep can thanks!

[redacted] — edited 8:15 AM
yo bro, here's my thoughts. i share this w the mindset of being objective and the purpose of refining the reason why you are committing to algos. feel free to disagree w (some) of these thoughts - it's your life and you have the final say.

1. **Are you still working on too many different specialisations?** From a glance, you are committing to algos, ML, SE, parallel computing (which are all diff focus areas). while i understand that you are diversifying to keep your options open, there's huge opp costs in keeping too many doors open while committing to a specific focus areas. for instance, if u commit to algos in terms of UROP, 2040 TA, internship, and only found out u like ML in Y3S1 later, the opp cost to switch over is really really high.(Would you even switch over?) Perhaps it'd be better to keep doors adjacent to algos open (areas which uses algos significantly), or

Fig 1: A Telegram conversation with a friend

Witnessing the practical needs of one's discipline can be powerful in bringing to light its *disconnect* with theoretical explorations. The first time I witnessed this disconnect was during an internship after freshman year, where my team developed a digital system for volunteer management. I was struck by the irrelevance of theoretical concepts – none of the problems involved designing algorithms to solve complicated problems. Instead, significant parts of the work involved improving the website design or filling data gaps such as the gender of volunteers. Even easier algorithmic problems like sorting and searching (covered in every Algorithms 101 class) are handled by "libraries"[1] that implement the algorithms for you. This was not an isolated experience – my next internships exhibited the same irrelevance of algorithms in daily work. It took seeing the disconnect from within the industry to seriously question theoretical learning's relevance beyond intrinsic value.

Examining the disconnect reveals that real-world work is often about highly *human-centric problems*, which theoretical learning does not concern with. As mentioned in my internship report excerpted below, a lot of time was spent clarifying user requirements, rather than designing an algorithm to find the shortest path or sort a large collection of data efficiently. Knowing how to code was sufficient, and theoretical knowledge was not necessary. For instance, our team spent more than a week discussing what options to include in a "mood" bar (happy, sad,

[1] A library is code written by someone else to use directly without knowledge about internal implementation. Example: to efficiently sort a bunch of numbers, just type `sort(numbers)`.

sick, etc) for the volunteers, and a significant amount of time testing it with volunteers. As we will see, theory helps in many contexts and does not help in many others – this was an example of the latter. Regardless, all I could think during that period was why I spent so much time learning concepts that are "fundamental" but not practically relevant. The experiences made me question why little attention was given to user experience and design thinking, which are important components of solving human-centric problems.

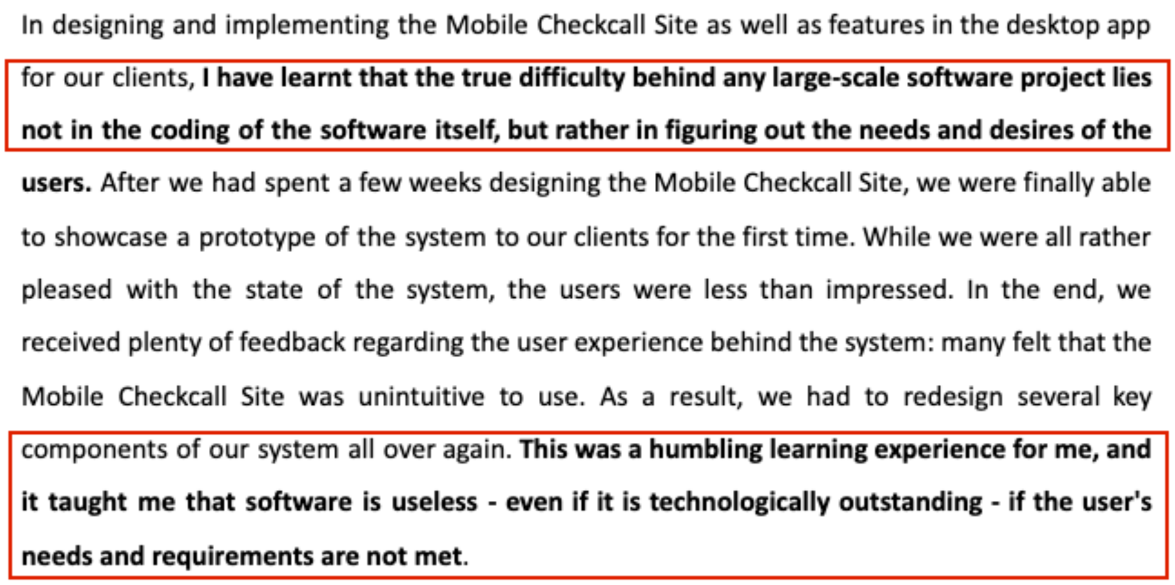
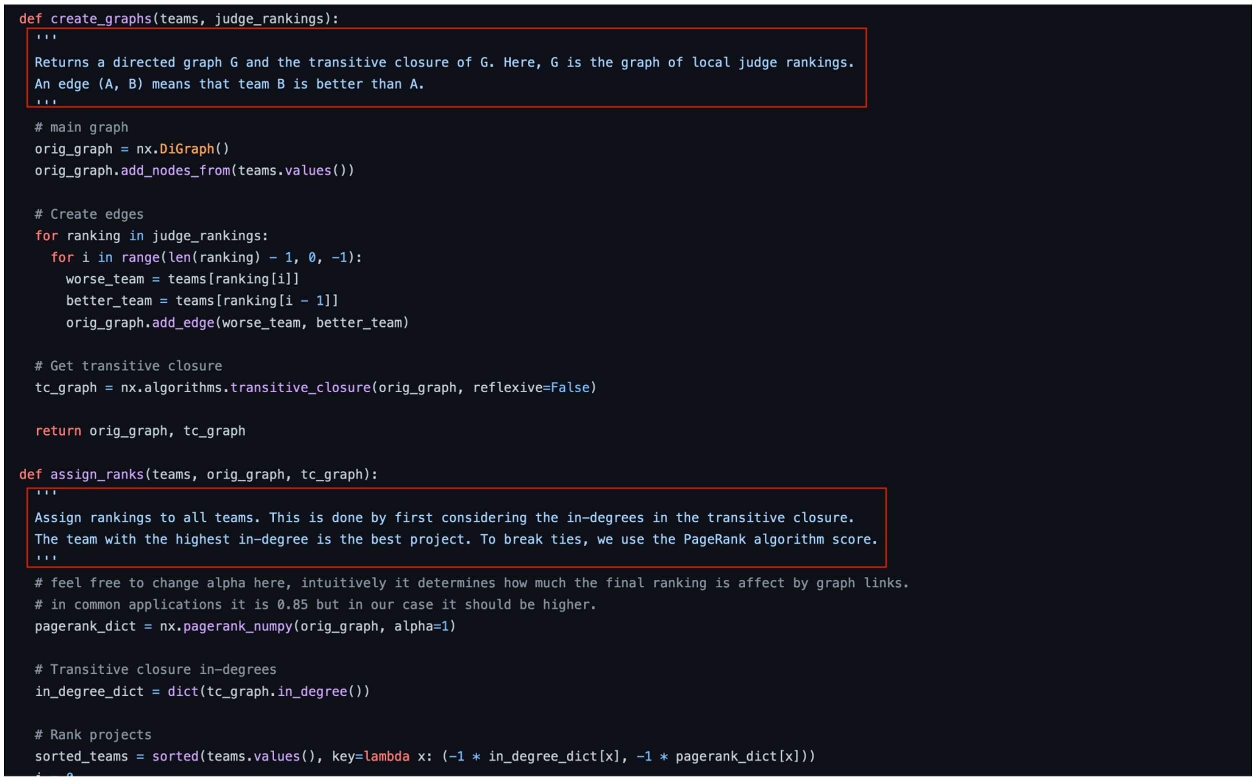

In designing and implementing the Mobile Checkcall Site as well as features in the desktop app for our clients, **I have learnt that the true difficulty behind any large-scale software project lies not in the coding of the software itself, but rather in figuring out the needs and desires of the users.** After we had spent a few weeks designing the Mobile Checkcall Site, we were finally able to showcase a prototype of the system to our clients for the first time. While we were all rather pleased with the state of the system, the users were less than impressed. In the end, we received plenty of feedback regarding the user experience behind the system: many felt that the Mobile Checkcall Site was unintuitive to use. As a result, we had to redesign several key components of our system all over again. **This was a humbling learning experience for me, and it taught me that software is useless - even if it is technologically outstanding - if the user's needs and requirements are not met.**

Fig 2: Excerpt from my internship report

Despite this tension, engaging with theory has benefits that surpass a lack of direct instrumentality – it makes one an effective *contextual computational thinker*. By that I mean someone who can understand user contexts, abstract out relevant information and apply computational techniques to solve problems. This learning came from designing a ranking system for a project fair. Understanding user context was important – I had to understand issues with numerical scoring that are the obvious choice for ranking systems, and how comparative ranking (where a judge compares projects and assigns relative rankings) enables better user experience and more robust results[2]. Nevertheless, it was my theoretical learning that helped me modify a complicated algorithm and apply it to the context of ranking systems, eventually leading to a superior solution. Being a contextual computational thinker comprises two aspects – *learning how to learn* and applying the *spirit of computational thinking* – which I will explore below.

[2] A great article about rankings systems: https://www.anishathalye.com/2015/03/07/designing-a-better-judging-system/#fn-1. I stumbled across this after we designed our own ranking system.



```python
def create_graphs(teams, judge_rankings):
  '''
  Returns a directed graph G and the transitive closure of G. Here, G is the graph of local judge rankings.
  An edge (A, B) means that team B is better than A.
  '''
  # main graph
  orig_graph = nx.DiGraph()
  orig_graph.add_nodes_from(teams.values())

  # Create edges
  for ranking in judge_rankings:
    for i in range(len(ranking) - 1, 0, -1):
      worse_team = teams[ranking[i]]
      better_team = teams[ranking[i - 1]]
      orig_graph.add_edge(worse_team, better_team)

  # Get transitive closure
  tc_graph = nx.algorithms.transitive_closure(orig_graph, reflexive=False)

  return orig_graph, tc_graph

def assign_ranks(teams, orig_graph, tc_graph):
  '''
  Assign rankings to all teams. This is done by first considering the in-degrees in the transitive closure.
  The team with the highest in-degree is the best project. To break ties, we use the PageRank algorithm score.
  '''
  # feel free to change alpha here, intuitively it determines how much the final ranking is affect by graph links.
  # in common applications it is 0.85 but in our case it should be higher.
  pagerank_dict = nx.pagerank_numpy(orig_graph, alpha=1)

  # Transitive closure in-degrees
  in_degree_dict = dict(tc_graph.in_degree())

  # Rank projects
  sorted_teams = sorted(teams.values(), key=lambda x: (-1 * in_degree_dict[x], -1 * pagerank_dict[x]))
```

Fig 3: Ranking algorithm code, comments within boxes highlight real-world to computational abstraction

Firstly, good theoretical fundamentals teach someone to *learn how to learn*. Not everything can be taught in school, and a mastery over theory can ensure that learners are ‘generalist’ enough to understand cutting-edge developments by understanding their foundations. The solution I developed was a modification of *PageRank*, an algorithm used to rank web pages in the Google search engine. Even though the *PageRank* paper is more complicated than what I encountered as an undergraduate, I was able and understand its concepts like ‘internet as a graph’ or ‘random walks over graphs’ only because I could build on my theoretical fundamentals. Learning how to learn is important to anyone working in the industry; given the fast pace at which technologies emerge, an education in fundamentals allows one to grasp these and apply them on real-world problems.

The second way in which theoretical fundamentals help, and complement human-centric design, is by infusing a *spirit of computational thinking*. This refers to systematically abstracting out relevant data from a real-world scenario and applying theoretical fundamentals to that data. Learning theory well helped me abstract the real-world scenario into a computational problem, which I could apply algorithmic techniques to. I first had to realise that the *PageRank* algorithm could generate project rankings instead of webpage rankings. I then had to model the projects being judged as a graph, and relative rankings as the edges of that graph, which I would not have been able to do without my strong theoretical knowledge (this cannot be outsourced to a “library” since they libraries implement context-agnostic algorithms, and the context of a ranking system was important here). Thus, people working in the tech industry can design *more effective* *and more robust* solutions to real-world problems by converting user needs to a computational problem and applying relevant fundamentals to solve that problem.

So, do the many software engineers working on non-theoretical user-facing problems benefit from learning theory? I believe so, since they adapt to changing technologies, abstract out relevant data from real-world problems, and design computational solutions to them. I term it as *contextual computational thinking*, since it requires an understanding of real-world context and computational principles. The value of design thinking and user experience is important, but while the user contexts change, fundamentals stay the same. An (imperfect) analogy is to imagine a blacksmith working with a set of tools. While each object wrought is different and catered to user needs, the tools employed are the same, and the same tool can be used for multiple objects. Similarly, the industry requires people that not only *understand the context*, like a blacksmith understands the objects they need to make, but also *have mastery over theoretical concepts*, just like a blacksmith has mastery over their set of tools.

Word count: 1428