ABSTRACT: TODO!

1. Introduction

A solid understanding of Data Structures and Algorithms (DSA) is an important skill for any student in the field of Computer Science, since they underpin many of the fundamentals of Computer Science. However, as is the case with many conceptually demanding topics, students can find learning DSA challenging early on in their studies[1] due to their inability to correlate DSA concepts with real-world objects and problems. Furthermore, the complexity of DSA means that some students may struggle to engage with the topic, thus requiring more motivation if they are to successfully learn it.

Educational tools have often been proposed that either take advantage of Game-based Learning (GBL) or Algorithm Visualisation (AV) in order to help students better learn concepts in computer science.

Game-based Learning is the use of games with traditional game elements (such as level progression or animation) in order to teach or practice a particular topic[2]. We find that GBL is been quite commonly used in the some fields of computer science, especially programming, however there are currently few examples of games that directly teach basic DSA. Those games that do exist to teach DSA usually revolve around more advanced DSA theory, such as algorithmic complexity. While learning those concepts are also important, there is a lack of games that help students learn about the practical application of DSA to computer science problems.

Another commonly studied educational tool is the concept of Algorithm Visualisation (AV). With AV, we try to abstract away the implementation of a given algorithm or data structure (in terms of, say, memory locations and pointers) and instead present them as static or dynamic diagrams or even analogies. Unfortunately while AVs are commonly seen in DSA teaching, most of these AVs are of the static, box-and-line diagram type[3]. When compared to dynamic visualisations that use analogies, static AVs may be less effective at helping students correlate DSA with problems and implementations.

We therefore propose an educational tool, DeCode, that integrates both GBL and AV in ways that aren't commonly seen in the field of DSA. By using both GBL and AV approaches, we hope to address both issues with learning DSA - the high complexity (with support from AV), and lack of student motivation (with support from GBL)

DeCode is a game-based learning tool for teaching introductory data structures such as Arrays, Lists, Stacks, and Queues alongside basic algorithms that use said data structures. DeCode takes advantage of game-based level design and a 2.5D, animated depiction of data structures. The animation uses the metaphor of cars and parking spaces to better engage the student and help map key DSA concepts to a real-world analogy.

The research, implementation process and evaluation of DeCode will be discussed in this report, along with conclu-

sions and possible future work.

2. Research Intent and Questions

Our research intent revolves around determining whether our implementation of Algorithm Visualisation with Game-based Learning in the same tool offers improved educational value for students new to learning DSA. We therefore want to address the following research questions;

1. Does DeCode help improve students understanding of DSA concepts?
2. Does DeCode offer additional educational value on top of existing teaching methods or tools (such as box-and-line visualisations of DSA)?
3. Is DeCode enjoyable and engaging for students to use (and thus students are more likely to remain motivated)?

3. Literature Review

3.1. Algorithm Visualisation Techniques

One of the first techniques used to improve instruction of DSAs was Algorithm Visualisations. While these were originally limited to drawings on paper and blackboard, these eventually developed into software-generated interactive visualisations as early as 1984, with BALSA[4] as an early example of such. As development progressed, more user friendly visualisation tools were created, with most being freely distributed online as open source software, for use by educators around the world.

Despite this work however, the most common examples of AVs are still Static AVs (whether on a computer or paper), which do not allow the user to interact with the visualisation while using it, or allow analogies to be displayed. Recent research has shown that improved interactivity, explanations[5] and the ability for students to construct their own visualisations can help significantly improve the usefulness of AVs to students, especially those that struggle with traditional learning methods[6].

3.2. Existing AV Work

One of the most prominent algorithm visualisation tools available now is VisuAlgo.net[7], developed at the National University of Singapore. It offers a clean, modern, 2D visualisation of various data structures, and full interactivity for the user to perform operations on the data structure, and see what happens. The disadvantage of this approach is that the visualisations don't naturally lend to analogies to be constructed. We decided to implement the idea of having interactivity for the user to perform various operations on the data structure into our game, since we felt it allows for much better self-directed learning and experimentation.

While most AV tools we identified were designed purely for teaching and demonstration, the TRAKLA2[8] tool was

developed to be used as a student assessment tool. Even though it uses similar, traditional representations of DSA as VisuAlgo, TRAKLA adds the ability for students to view algorithm or DS questions. Students are then able to answer the questions by manipulating the visual representation, and have their answer marked automatically. Students are also able to view and browse through (i.e. step back and forward) the model solution for their question.

3.3. Existing GBL Work

Despite a large volume of existing work on both Game Based Learning and Algorithm Visualisation, there's a surprisingly low number of papers that implement both GBL and AV to teach DSA. One of the few papers we identified was Park and Ahmed[9] in which they design a video game using the Unity game engine that visualises various data structures. The game uses real world analogies in order to visualise these in a 3D environment, such as showing the stack data structure as a stack of crates. The player can perform operations on these data structures in order to complete game objectives. Vegh[10] presents another similar game that's centred around algorithms rather than data structures. The game uses a real word analogy of crates to demonstrate sorting algorithms (in this case, sort by mass). The game also logs the actions of players, allowing educators to investigate them and determine the student's line of thought.

3.4. Other Education Research

Most tools that teach basic DSA usually don't cover some intricacies of working with them, which can result in students being confused on concepts that might otherwise be regarded as basic. Izu, et al.[11] presents an example of students struggling with array manipulation and loop iterations where students are unable to recognise which direction they should loop in, in order to shift an array to the left or right. They conclude that this is the result of an unintentional oversight in course materials favouring upward loops.

Meanwhile, there has also been research into correlations between students ability to explain and trace code and their abilities to write code. Teague and Corney's work[12] presents a study of such a relationship where the ability of students to explain a simple code block that swaps two elements is correlated with their ability to explain a simple sorting algorithm that relies on swapping out-of-order elements (similar to BubbleSort).

4. Goals

In general, we had three broad goals during the development of DeCode;

1. Teach introductory DSA concepts
2. Be an engaging tool for the user
3. Offer a highly visually-appealing visualisation

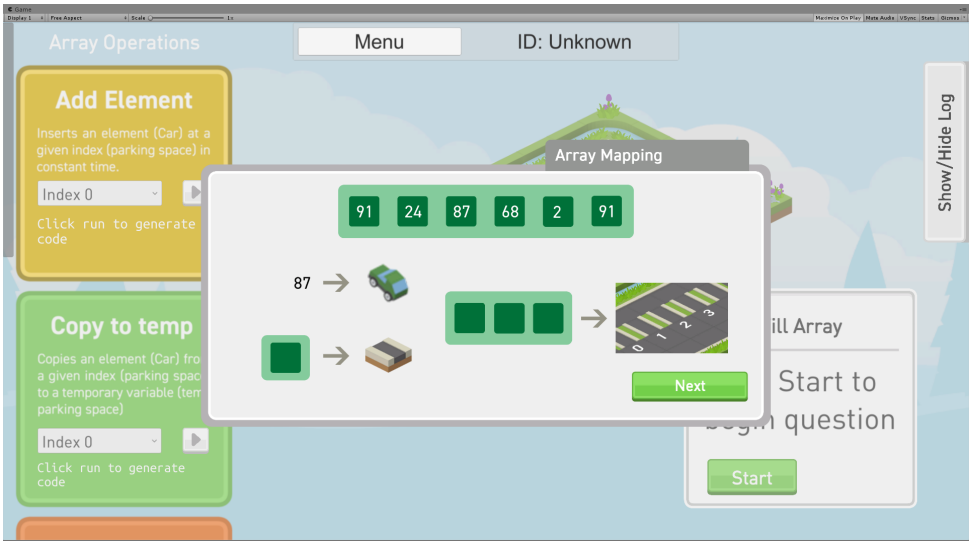

Figure 1: Help screen explaining our analogy

Additionally, from our literature review, we were also able to identify ideas from multiple pieces of existing work that we would like to integrate into DeCode.

- An *interactive* algorithm visualisation with explanations, since they've been shown to be more useful to students[5].
- Ask students DSA questions, and allow them to answer them by manipulating the visualisation, such as in TRAKLA2[8].
- Use a game environment, with game objects as real-world analogies to better demonstrate DSes, such as in Park and Ahmed[9].
- Log students interactions with the tool, to allow educators to better understand student's line of thought, similar to Vegh[10].
- Explicitly introduce loop directions and array shifting, since Izu, et al[11] showed that students are commonly confused by this.
- Measure students abilities to explain and trace code instead of just their performance in performing DS operations, in order to see if we can reproduce the relationships between the two that Teague and Corney demonstrated in their work[12].

5. Analogy/Metaphor

Using an analogy in our visualisation can allow us to make DeCode more visually appealing, while also improving educational value[9][5]. However, coming up with an analogy that works for all of our use cases proved to be quite difficult - if something cannot be adequately explained using your analogy, then it quickly falls apart. The analogy should

also be relatable for almost all users of the application and not be overly complex. Eventually, we settled on the idea of cars and carparks as our analogy.

Cars would represent the values that we would want to store in our data structure, with different types/colours of cars representing different data values that we would like to store. Parking spaces represent locations in memory where these values can be stored, and contiguous blocks of memory (such as arrays) are represented as carparks. To store a value into a memory location, we use the analogy of parking a car into a parking space. The parking spaces are labelled with their index where appropriate.

To help users get better accustomed to our analogy, we show a help screen (Figure 1)at the start that links the analogy to a more classic box-and-line diagram visualisation of data structures.

5.1. Limitations

One of our primary goals with this analogy was to have minimal complexity, which meant keeping it to only two main elements (cars and parks). Unfortunately, this meant that there were a few things that ended up being inadequately explained by our analogy.

Firstly, the user is able to view the contents of the whole carpark at a time, whereas in reality it takes time to check the contents of each memory location. This made some algorithms such as sorting algorithms more awkward to explain since in reality comparing various indexes makes up a significant portion of the running time of such algorithms. Initially we intended to add some sort of character to the game (such as a driver) to represent the current state of execution - so, as an example, the driver would walk around the carpark and look at cars in order determine their value, then get in one and drive it when array elements are being moved/copied. However, we found that this overly complicated the visualisation, and actually made it harder for users to keep track of what's going on. We therefore decided to go without this, even if it did mean our analogy was less accurate.

The second issue was trying to display pointers with our analogy - we simply couldn't come up with something that would fit well that represented a pointer without being overly complex. Signs and other things pointing to a specific parking spot (memory location) would be too complex for students to grasp. Ultimately, we decided to omit most data structures that use pointers, such as Linked Lists since we would be unable to represent them satisfactorily. The only DS that had a pointer was our optimized implementation of Queue, for which we used a red flag to mark the parking space pointed to by a pointer. This only worked since we only had one pointer (having multiple, different coloured flags would be confusing for something like a Linked List).

The final issue that we identified was how we would deal with copying of values in the data structure. For example, copying the value from index 0 to index 1 in an array would mean *duplicating* the car that stores the value in index 0

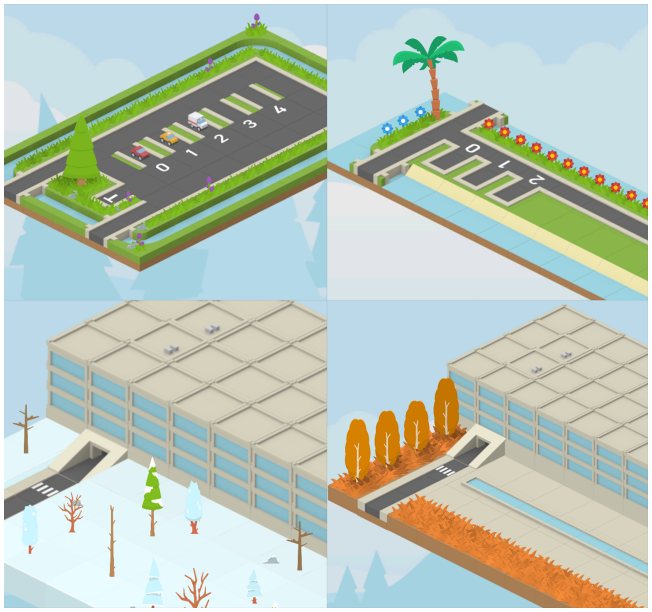

Figure 2: 4 seasons for the 4 levels in DeCode

and driving it to index 1, while leaving the original in index 0. This breaks the analogy's link to the real world since one cannot simply duplicate a car in reality. In the end we kept this in the game since we were unable to come up with a way to represent this more realistically. Even if this one part isn't realistic, we feel students will still be able to understand the intent of the analogy.

6. Design and Technology

6.1. Design

One of the main goals of DeCode was to design a highly visually appealing visualisations, in order to draw users in and differentiate our visualisation from other, more standard, 2D visualisations. One of the best ways to achieve a high quality visualisation would be to use custom 3D models for our analogies, however we this would've been impractical given the time constraints we were under. Instead, we elected to use existing city assets from Kenney's Isometric City Pack[13]. This asset pack offered most of the assets that we needed for our analogy (city buildings, streets, parking spaces and cars) with a consistent design across them. The assets that weren't available (such as labelled trucks for sorting and indexed parking spaces) were created manually by us, with care taken to ensure they match the design of the Kenney pack.

To visually separate each of the 4 data structures we were introducing, each data structure has its own seasonal themed variation on the normal assets. Figure 2 shows the design of DeCode across all four seasons (data structures).

UI design was also done by us, with the goal of trying to match the design of the Kenney pack as much as possible. A primary consideration in our UI design was to ensure all UI elements would be usable on a touch device, should a user choose to test our game on such a device. This meant things like array index selection would need to be designed as drop

down boxes instead of text boxes, since popping up a keyboard results in a poor touch experience. We also made sure that all buttons were of sufficient size for touch interaction.

6.2. Technology

7. Data Structures

We chose 4 data structures to cover in our tool based on their importance in the ACM CS2013 curriculum's section on Fundamental Data Structures[14] and our ability to visualise it using our analogy. As mentioned earlier, we opted to skip data structures that require significant use of pointers since we are unable to represent them using our analogy.

7.1. Arrays

Arrays are depicted as a simple carpark of a fixed size, with an extra park representing a temporary variable. Users are asked to fill the empty array and swap values around - this introduces the user to most of the array operations, including add, copying to/from temp and copying from one index to another. Following on, the user is asked to rotate the array using the operations they were just introduced to. We chose to do this since prior work[11] has shown that students are commonly confused by these operations. Finally, we introduce the user to BubbleSort - a common simple sorting algorithm that puts to use all of the operations that they have learned thusfar. BubbleSort is introduced in two steps - firstly, users are given an animated demonstration of the algorithm completely with explanations, after which they are asked to step through each step of the algorithm by manipulating the visualisation. Throughout this, the user is asked basic multi-choice questions about what they just did, except in a psuedocode form. This encourages the user to link their knowledge in the visualisation with code.

7.2. ArrayLists

ArrayLists (also known as Vectors in some languages) are introduced in two parts - first, we expose the user to the benefits of a List - namely, dynamic sizing that allows us to add as many elements as we see fit. The user is asked to add elements to the ArrayList, and the internal array is shown as expanding every time a new element is added. This is visualised by having multiple rows of parking spaces next to each other (see Figure TODO), with each row 1 larger than the last. When a new element is added that exceeds the size of the current row, all the cars are moved to the next one, thus representing a possible naïve implementation of ArrayList. Throughout this process, the user is asked questions on how the performance of this implementation could be improved, in order to get them thinking about the next part.

In the second part, we introduce a more optimized implementation of ArrayList to the user, in order to give them an idea of how ArrayLists are used in the real world. This done using the same array expanding visualisation as before, except we now show how the implementation doubles the size of the internal array each time.

Figure 3: Annotated view of the gameplay features

7.3. Queues

Since Queues are an abstract data type (ADT), we elected to show it in two parts - first, we show the user the interface of such an ADT, where the implementation is a "black box" and the user can only perform enqueue and dequeue operations on said box. We visualise this as a parking garage with one entrance (for enqueues) and one exit (for dequeues). After the user is familiar with these operations, we allow the user to "see inside" the parking garage, thus revealing the implementation to them. Similar to ArrayLists, we first introduce them to a naïve implementation of a Queue (basic array), then proceed to introduce a circular array implementation, where we use a pointer (red flag) to arbitrarily define the start of the array.

7.4. Stacks

Stacks, being another ADT, are also introduced in a similar way to Queues - first a "black box" view where the user can only use the single entrance for push/pop, and then later a "see inside" view where the array-based implementation is revealed to the player.

8. Gameplay Features / User Interface

The majority of the game has a interface similar to that shown in Figure 3. The annotated elements do the following...

1. Shows the user their current game ID, used to track their progress for the evaluation (more in the Evaluation section).

2. The code view allows the user to see a pseudocode representation of the operations that they have done, so that they can better relate the visualisation with code.

3. Tasks for the user are introduced here. Users are able to *check* if their answer is correct, or *reset* in order to start

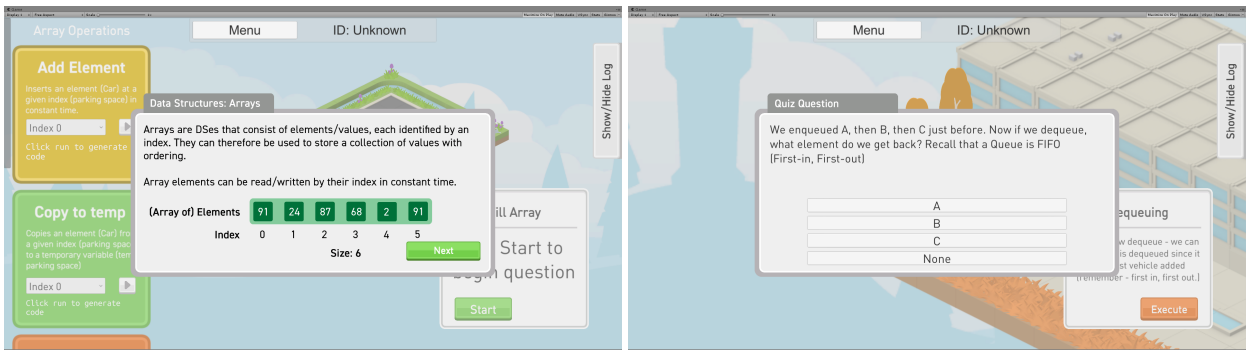

Figure 4: The help and quiz pop-up screens

again.

4. Operations supported by the current data structure are shown here, and users can select the index(es) to perform the given operation on. Generally, not all operations are shown at once since they are gradually introduced to the user

5. The visualisation is shown in the centre of the screen. When the task is introducing something to the user, the visualisation automatically zooms in to focus on what's being shown.

In addition to these elements, DeCode also shows help screens that give users a general introduction to a given data structure, and has quiz screens to ask users specific multi-choice questions, both of which are shown in Figure 4.

9. Evaluation

9.1. Design

9.2. Results

10. Future Work

11. Conclusion

References

[1] B. Pérez-Sánchez and P. Morais, "Learning data structures—same difficulties in different countries?" *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje*, vol. 11, no. 4, pp. 242–247, Nov 2016.

[2] K. Kuk, D. Jovanovic, D. Jokanovic, P. Spalevic, M. Caric, and S. Panic, "Using a game-based learning model as a new teaching strategy for computer engineering," *Turkish Journal of Electrical Engineering and Computer Sciences*, vol. 20, pp. 1312–1331, 12 2012.

[3] M. Esponda-Argüero, "Techniques for visualizing data structures in algorithmic animations," *Information Visualization*, vol. 9, no. 1, pp. 31–46, Mar. 2010. [Online]. Available: http://dx.doi.org.ezproxy.auckland.ac.nz/10.1057/ivs.2008.26

[4] M. H. Brown and R. Sedgewick, "A system for algorithm animation," *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 177–186, Jan. 1984. [Online]. Available: http://doi.acm.org/10.1145/964965.808596

[5] L. Végh and V. Stoffova, "Algorithm animations for teaching and learning the main ideas of basic sortings," *Informatics in Education*, vol. 16, pp. 121–140, 01 2017.

[6] J. Stasko, A. Badre, and C. Lewis, "Do algorithm animations assist learning?: An empirical study and analysis," in *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, ser. CHI '93. New York, NY, USA: ACM, 1993, pp. 61–66. [Online]. Available: http://doi.acm.org/10.1145/169059.169078

[7] "Visualgo.net." [Online]. Available: https://visualgo.net/en

[8] L. Malmi, V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä, and P. Silvasti, "Visual algorithm simulation exercise system with automatic assessment: Trakla2," *Informatics in Education*, vol. 3, pp. 267–288, 10 2004.

[9] B. Park and D. T. Ahmed, “Abstracting learning methods for stack and queue data structures in video games,” in *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, Dec 2017, pp. 1051–1054.

[10] L. Vegh, “Using interactive game-based animations for teaching and learning sorting algorithms,” 04 2016, pp. 565–570.

[11] C. Izu, C. Pope, and A. Weerasinghe, “Up or down?: An insight into programmer’s acquisition of iteration skills,” 02 2019, pp. 941–947.

[12] D. Teague, M. Corney, A. Ahadi, and R. Lister, “Swapping as the ”hello world” of relational reasoning: Replications, reflections and extensions,” in *Proceedings of the Fourteenth Australasian Computing Education Conference - Volume 123*, ser. ACE ’12. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2012, pp. 87–94. [Online]. Available: http://dl.acm.org/citation.cfm?id=2483716.2483727

[13] [Online]. Available: https://www.kenney.nl/assets

[14] A. f. C. M. A. Joint Task Force on Computing Curricula and I. C. Society, *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. New York, NY, USA: ACM, 2013, 999133.