Stellaris Universal Expansion Tool

Cameron Arsenault Max Schwartz

April 30, 2020

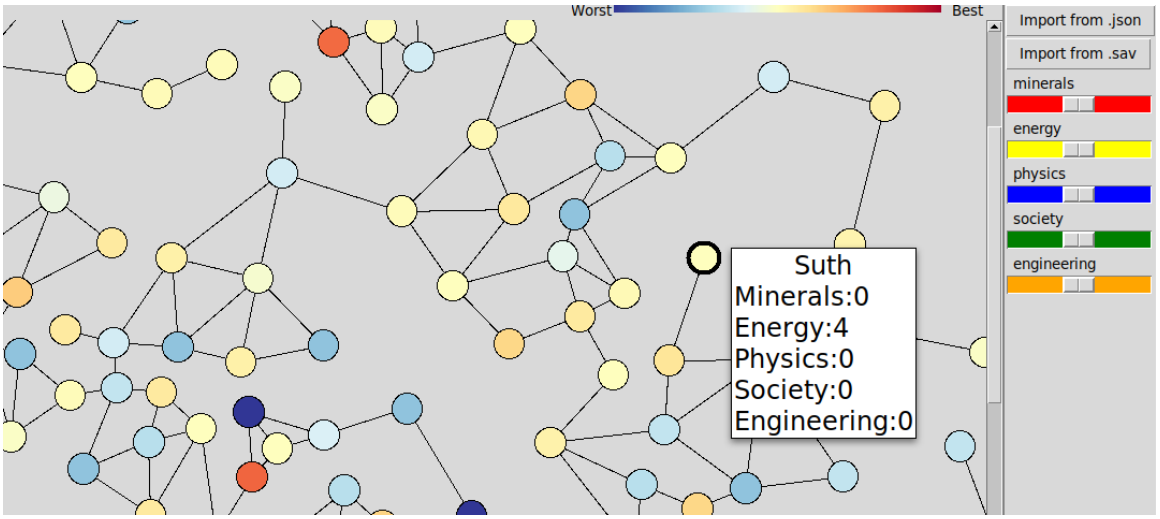

Figure 1: Above is the finished visualization tool. To the left, the galaxy is visualized within the main canvas. The user is able to hover over a system to view more information. To the right is the user interface with import buttons and weight sliders.

1 Introduction

For our Interactive Visualization final project, we created a tool for a game called Stellaris. Max starting playing Stellaris in high school and introduced Cameron to it once they met in college. Before we describe our tool, let us describe Stellaris. Stellaris is a game created by Paradox Interactive where you play the role of a newly interstellar empire. A player starts off inhabiting only a single planetary system. As the game goes on, the player explores and expands to systems around them. The systems that count as being next to each other are any two systems who share a hyperlane. They can easily claim this new system by building an outpost there. The planetary systems are a part of a larger galaxy which can consist of anywhere from six-hundred to one thousand systems. The player's goal is to become the most powerful empire in the game, competing against other players. This can be accomplished by eliminating other players through war, befriending empires and joining a federation, or being the hero of the galaxy while fending off intergalactic threats. In order to accomplish this, the player has to gather resources which each system contains. The main resources in the game consist of minerals, energy, physics, society, and engineering. No matter how a player wants to tackle the game, the start of the game remains very important. Establishing yourself in the galaxy can be instrumental in a player's chances of

victory or defeat. The Stellaris Universal Expansion Tool (SUET) is meant to help players of Stellaris get a strategic foothold in their game. Since veteran players of Stellaris don't need that much assistance when deciding where to expand, we think that our tool will be most helpful for players that are new to the game. Our goal is to visualize which systems are most desirable for players to expand to based on resources that are present in the game.

2 Related Work

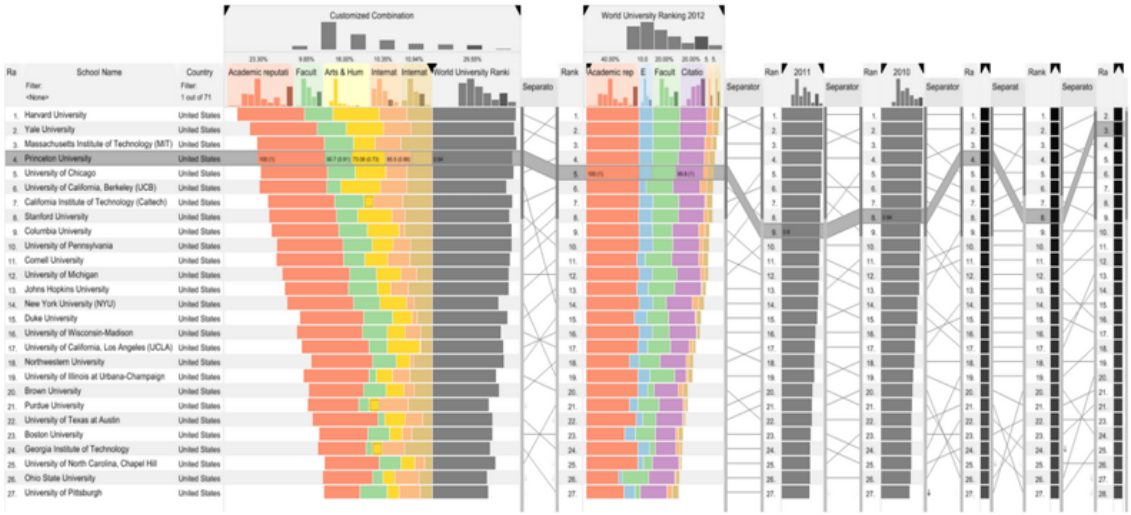

Figure 2: Above is an example of LineUp (Gratzl et al., 2013). The items on the left are the individual items with the bars to their right describing their overall score and ranking. The colored boxes at the top represent the weights of each of these scores which can be changed by clicking and dragging that box to make it larger or smaller.

2.1 LineUp

LineUp: Visual Analysis of Multi-Attribute Rankings written by Samuel Gratzl, Alexander Lex, Nils Gehlenborg, Hanspeter Pfister and Marc Streit describes a tool that they created. This tool ranks items based on multiple data points describing those items with various weights (Figure 2). These data points are normalized and added together to give each item in the list its score which it is then ranked on. A user of LineUp can actively change the weights of each of these data points. This means that if a user finds one metric more important than another, they can make that apparent in the ranking system. LineUp displays to the user how ranks change after a weight adjustment is made. Since every game of Stellaris is different, we thought that it was a necessity for a user to be able to change weights so our application could fit their needs.

2.2 ColorBrewer

ColorBrewer.org: An Online Tool for Selecting Colour Schemes for Maps by Mark Harrower and Cynthia A. Brewer describes a tool for coloring maps. ColorBrewer is a tool

meant to help map makers pick the correct color scheme for their own personal needs. By inputting how many different colors one needs, what medium the map will be presented on, and other considerations like color-blindness or black and white versions, ColorBrewer will present the map maker with different color schemes that fit these requirements. One of the most important inputs is the type of color scheme the user needs. This includes sequential, diverging, or qualitative color schemes. Qualitative color schemes are comprised of all different colors. These colors have no correlation with each other and can potentially represent anything. Sequential color schemes have white on one end and a color on the other with steps between them. These schemes are great at representing a scale from zero to some maximum number. Diverging color schemes have a color on one end, white in the middle, and a color on the other end. Like sequential color schemes, these include color steps transitioning between these colors. These are great for when you have a scale with negative numbers and positive numbers. In this situation, the middle white represents zero, while the deep hues on each end represent either the minimum or maximum of the scale.

2.3 Stellaris Save Visualizer

The Stellaris Save Visualizer by Michael Kim was a project that we found on Github with similar goals to our application. His project converts a Stellaris save file to a JSON and displays the galaxy in a web application using `d3js`. Each node on the map is a planetary system where owned systems are colored by each empire's color and vacant systems are shown as smaller dots. He shows hyperlanes as black lines between systems and additionally shows wormholes, direct connections between far away systems, as pink lines. The main difference between our project and his is that it mainly aims to purely visualize a save while our project aims to show things to the user that they would not be able to see normally.

3 Design Process

3.1 Data Processing

The first step for creating our visualization was extracting data from a Stellaris save file. Stellaris save files are zip archives with the `.sav` file extension containing two text files: `gamestate` and `meta`. The text file we mainly care about is `gamestate` which contains practically all of the information relating to a save game and is where we pull our data from. The data is structured in a way that looks similar to JSON but Paradox uses their own non-standard file structure. Thankfully, there was a library created for deserializing these files called `pypdx-clausewitz` (Kim, 2019a) which we utilized for converting these files from `.sav` to `.json`. File conversion is done in our `converter.py` file which returns a python dictionary and locally saves a JSON file created from the save. Using this library allowed us to put more of our focus on the visualization aspect of this project. However, it

may have cost us in performance, since we did not write this library. File conversion takes around thirty seconds to one minute depending on the file size, but this only needs to be done once and all subsequent uses of our application can load from the created JSON file.

Once we have the `gamestate` converted into a python dictionary, we then perform some pre-processing steps to allow us to more easily store and manipulate information. The base Stellaris save has very little when it comes to structure, with basically every component being on the top level of the dictionary. This is why we create a more structured `Galaxy` object to allow easier parsing of the data. For our visualization, we care about three things: Systems, Planets, and Deposits. `Systems` store information associated with planetary systems. For each system, we store its name, position in x,y coordinates, hyperlane connections, and planets. A system has multiple planets and for each `Planet`, we store its name and deposits, if any. A `Deposit` tells us detailed information about what is on the planet in it's type. A deposit with type d_minerals_4 would mean the planet had 4 minerals, for example. Once we have processed all systems, planets, and deposits, we attach deposits to planets, creating a sum of the resources that planet contains. We then attach planets to systems, creating a sum of all the resources that system contains. Finally, we attach all the systems to our `Galaxy` object where we store, for each resource, the maximum and minimum values that are contained in a single system.

3.2 Galaxy Visualization

For the visual side of SUET, we looked at many different GUI libraries. We knew we wanted to make an application without being web based, which limited the types of libraries we could use. After looking at libraries such as `kivy`, `pyform`, `PyQt`, and `pyvis`, we eventually landed on using `Tkinter`. `Tkinter` worked for us mostly because of its simplicity. By creating a `Tkinter` canvas, we could create a circle centered at any x,y coordinate. This was helpful given that we had access to system positions in x,y coordinates from the save file. By first drawing the edges connecting the systems together and placing the circles on top, we create a galaxy of circles connected by edges (Figure 3).

In Stellaris, a player can pan around the galaxy as well as zoom in and out to get a closer look at systems. We wanted to have a similar experience with SUET which was intuitive to use. Some example code on stack overflow gave the framework we were looking for (FooBar167, 2018). This code provides a solution for the zooming and panning of a static image in `Tkinter`. By replacing the image with the nodes and edges of our galaxy, we were able to adapt this code to our application.

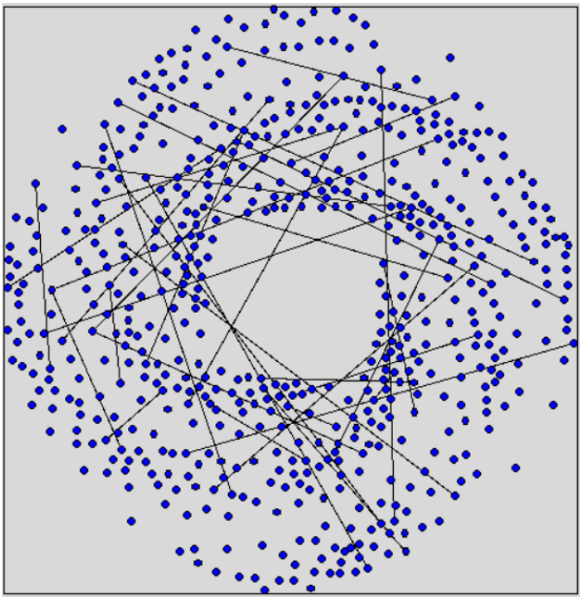

Figure 3: Above is one of the early drafts of the main galaxy view. The nodes at this point were all colored the same, and we randomly assigned edges between them. At this point we were getting correct coordinates for where to place each node.

3.3 User Interface

The user interface for SUET was also made in `Tkinter`. The first step and the most important buttons were the import buttons. If the user already has the JSON formatted data, then it can easily be read in once given the file location. If the user only has the `.sav` data we must first process it into a `.json`. We wanted a button for each because the time saved from already having the JSON would be significant enough to choose uploading a `.json` over a `.sav`. `Tkinter` has pre-built buttons which we used for both the `.json` and `.sav` options. We used `easyGUI` in order to open a file explorer upon clicking one of these buttons. Once the user selects their file, the file location is sent to our program where we either translate it to `.json` using the method described in "Data Processing" or we open the `.json` and begin drawing with the data. When the user first opens SUET, these two buttons are the only two that are available (Figure 4). This is because we do not have any data to show the user, and must wait until there is data to show. After the user gives the data these buttons migrate to the right side of the screen to be with the other user interface tools. If the user has other data to upload after uploading this initial data, they are free to click on one of these buttons and upload new data following the same process.

Figure 4: Above is how SUET first opens up for a user. We set a default size for this window since the buttons would be lost if they were not inside of a larger window. After clicking either of the buttons a file browser is opened which lets the user select their save file.

One of the key parts of our tool was the ability for a user to change the weights of a resource freely. We took much inspiration for this feature from LineUp (Gratzl et al., 2013). We considered allowing the user to manually input weights between zero and one, but felt that would not be interesting or engaging enough for a user. Instead we used `Tkinter` scales which we thought would be more intuitive when it comes to making some resources more important than others. A user can click on an arrow which is pointing at the scale and drag it to the left or right, left meaning less important and right meaning more. Initially, we wanted to show the user the current weight they are giving to each resource with a number from zero to one above each scale. We eventually removed this for simplicity of weight normalization. After we removed the number, we made it so that the scale is between one and one thousand and normalized within our code. We used scales that are pre-built in `Tkinter` and colored each one differently depending on the resource's in-game color. Minerals being red, energy being yellow, physics being blue, society being green and engineering being orange.

The last part of the user interface that we finished was the action of hovering over a given system and getting it's resource information. We thought it would be difficult for a user to understand why a system was getting the score it was if they were unable to see the individual values of a system. By using a `Tkinter TopLevel` function, we could make a floating text box above a system once a user hovers over it. This box displays the system name as well as all of the individual resource values. Once a user is finished looking at the system information, by simply moving their cursor off of the system the box will disappear.

3.4 Desirability Score

To convey to the user which systems are good, we had to create a score to quantify the desirability of a system. The first component of the desirability score came from the user-defined weights. From the sliders we get weights for each resource which when added together, sum to 1. For the second component of the score, which would be multiplied

by the weights, we went through a couple different iterations. Originally, we multiplied the total number of resources in each system by their respective weights. We decided that this gave an unfair bias to certain resources that may be more common, essentially already giving a weight to each resource based on its frequency. Since we wanted each resource to count the same when all the weights were equal, we decided that we should try doing a divergence from an average. This came with some difficulties since it introduced negative values and also had the potential for unfair biasing with some resources having minimums and maximums that are further away from the mean than others. We decided that we would normalize the divergences using these minimum and maximum values. Later, we discovered that this produced the same result as normalizing the number of resources in a system with the minimum and maximum values of that resource in the galaxy. Therefore, we decided to use the normalization of the number of resources.

Normalizing the number of resources in each system based on the minimum and maximums for the galaxy allowed us to treat all resources equally before their weights were applied. Our final desirability score, d, is defined as follows for each system:

$$d = \sum_{i=0}^{n} normalize(resources[i]) * weights[i]$$

where n is the number of resources, $resources$ is a list of total resources for each resource type, and $weights$ are the associated weights of each resources (defaulted to $1/n$). The desirability scores for every system needs to be updated every time weights are changed. For a future performance improvement, we would store the output of $normalize(resources[i])$ so that we only do that calculation once for every system.

3.5 Node Coloring

Now that we had a desirability score, we needed to color nodes based on that score. Both sequential and diverging color schemes could fit our data set. We had to decide which would be a better scheme for our visualization. After making the decision that it would be best to see a system's desirability in relation to a galactic mean, we decided to use a diverging color scheme. Using `Matplotlib` color maps for coloring our nodes, we chose their *coolwarm* color map using blue to represent systems below the mean and red to represent systems above the mean. We later observed that it was hard to tell the difference between certain shades of red and blue especially towards the center of the spectrum (Figure 5). We turned to ColorBrewer in search of a better scheme and decided on the *RdYlBu* diverging color scheme since it was relatively similar to what we were currently using but with easier to distinguish shades. We also found it to be easier on the eyes compared to the *RdBu* scheme, since we though the dark reds and blues at either end were a too intense (Figure 6).

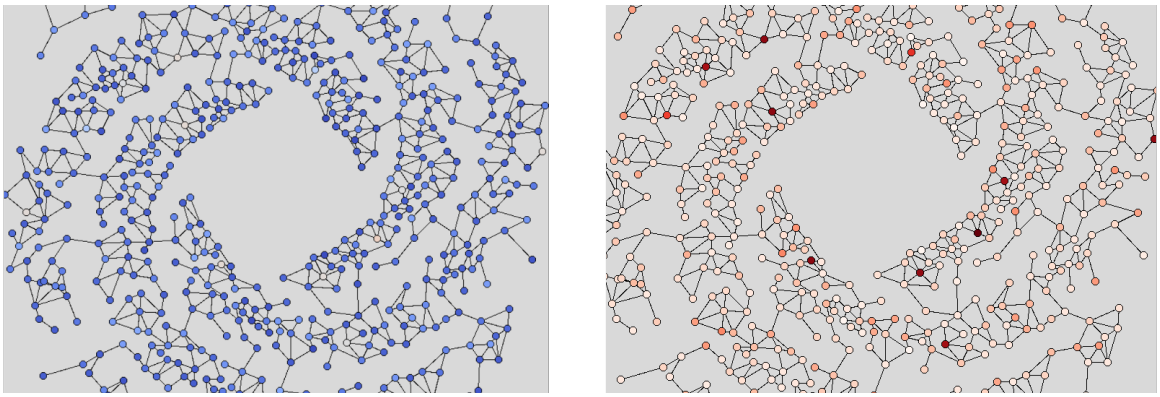

Figure 5: The left shows the *coolwarm* diverging color map before normalization of any kind was applied to the desirability score. The right shows the *Reds* sequential color map after normalization was applied to the desirability score but before using Two Slope Normalization. We decided on using the *coolwarm* color map before later switching to using the *RdYlBu* map.

Figure 6: Two color maps based on the ColorBrewer specifications. We decided to use the *RdYlBu* color map since we though the dark reds and blues in the *RdBu* color map were too intense.

Our color map took in a value between 0 and 1 and returned a color in RGB which we converted to hex. Initially, we passed our desirability score to our color map, since our desirability scores were guaranteed to be between 0 and 1, giving us an absolute coloring based on our desirability score. This posed a problem, however. When we first ran our application this way, almost all of the colors were blue. This was because the maximum score in the galaxy was just over 0.5, colored slightly red, and the mean score was around 0.09. We decided that it would be more useful if we instead normalized our desirability score based on the minimum and maximum scores in the galaxy. This made a big difference, but still did not solve the problem completely. Because a few systems had very high scores and many systems had very low scores, most of the colors were still blue. In order to solve this, we decided to use a Two Slope Normalization to set a custom midpoint in the color map (Figure 7). To best represent our systems scores in relation to the average, we decided to calculate the median score in the galaxy, which was often slightly lower than the mean. We used the `TwoSlopeNorm` function and *RdYlBu* color map for our final coloring of the galaxy displaying a key at the top of the application for users to reference (Figure 8). We notably reversed the color map so that blue would still represent bad systems and red would represent good systems.

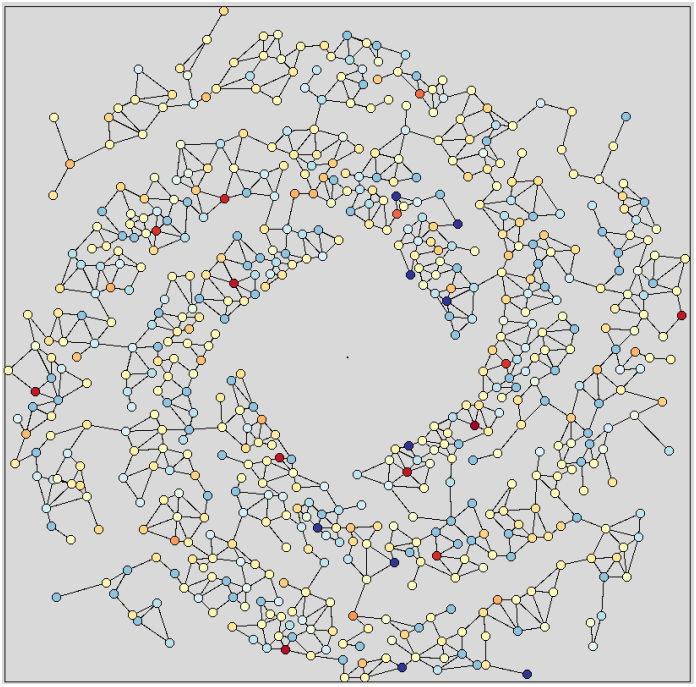

Figure 7: The final coloring using the *RdYlBu* color map based on the ColorBrewer specifications and using a Two Slope Normalization of the desirability score. Here, blue indicates a low desirability score, yellow indicates a score close to the median, and red indicates a high score. Hyperlane connections between systems are shown using black lines.

Figure 8: The key for the color map showing an increase in desirability from blue to red.

4 Final Design

The Stellaris Universal Expansion Tool (SUET) allows for a user to better plan their galactic expansion within a game of Stellaris (Figure 1). From opening SUET, the user can choose to either select their save data or a JSON file previously created by our application. The user is then shown a view of the galaxy with individual planetary systems connected by hyperlanes. The user is able to click and drag to pan along with zooming in and out with the scroll wheel. A user can get a close look at a cluster of systems this way, or zoom out to view the whole galaxy. Each system is colored based on the desirability score which takes in to account the five main resources in the game: minerals, energy, physics, society, and engineering. The user can modify the weights for each resource with sliders. After weights have been changed, the coloring will update for all of the systems. By referencing the color key at the top right of the visualization, the user can find systems that are *warmer* in color compared to other systems, and therefore have a higher score. The user can continue modifying weights until they have a good idea on where to expand, at which point they can resume their game of Stellaris and work towards expanding to their desired system.

5 Division of Work

For this project, Cameron worked mainly on the front-end of the application, implementing the user interface, and drawing the galaxy. Max worked mainly on the back-end doing file conversion, data processing, and the desirability score. Both members worked on creating the coloring for the systems and on the overall design process.

6 User Feedback

After giving the presentation for SUET we were given some peer feedback as well as professor feedback. There were many recommendations concerning the user interface. A broad but important bit of feedback was that the user interface as a whole looked outdated. This may be because `Tkinter` was created in 1999 and has not received many visual updates since. Any of the changes we made while using `Tkinter` would seem outdated. Another common bit of feedback was that the colors used for the scales were a bit confusing. More specifically, about the minerals and physics scales. These scales are red and blue respectively, and we use these same colors when coloring the systems based on their score. A user may see a blue node and think it has a lot of physics when it is in fact a bad system. This could be unclear especially to those who are not familiar with the in game resource colors.

Other feedback suggests new features and things that would help a user more when making the decision of where to expand. One bit of feedback suggests a feature which allows for a user to select a system. From this system, our application would highlight a path to the nearby system with the highest. This would make it easier for a user to make a decision. They can choose this suggested path, or find another system that they wish to claim. However, deciding what path to take could be difficult based on other factors in Stellaris that we are not currently representing. A similar feature which was suggested would be to highlight a system that would be recommended for a newer player. This may not be the optimal system, but perhaps one that could help out the player in case they make any mistakes during the game.

7 Discussion

We received some many ideas from the user feedback. Some comments mentioned that SUET looked outdated, which is in part due to how `Tkinter` looks and partly because we did not do a lot of customization to the default look of the widgets. There is a lot of work that can be done to modernize our visualization, including finding a sleeker GUI package or by making a web application. In terms of colors for the scales and the systems, there was some confusion on what these colors meant. For us, it was intuitive that the colors under the sliders representing the resource had no relation to the coloring used by the nodes. This is because we have played the game extensively, but new players, which are our target audience, might not understand this. We would like to look for alternative

methods of showing the resources, perhaps using the resource symbols from the game to show them more clearly to newer players.

There were a couple bits of feedback suggesting ease of use features which could help newer players. One of these features was to allow a user to select a system and showing the path from it to a nearby system with a high score. There would need to be some limit on how far the path could go from its starting point since it does not make sense for a user to travel to the other side of the galaxy just to get more energy, for instance. The other new feature suggestion was to highlight a node which is recommended for newer players. It would be difficult to decide what is recommended compared to what is optimal. This feature would likely be overshadowed by the path highlighting feature, since SUET would likely need your starting position in order to recommend a system anyways.

In terms of future work there are many things that we could add to this to improve it. One of the most important things would be to allow for an anti-cheat mode. Currently, we present information about the entire galaxy. It is unlikely that in a Stellaris game that a player gets information about every single system. This could be seen as cheating, and we definitely would want to give users a choice between cheating or not cheating. This would entail hiding system information for the user until they have explored that system, and only showing information based on visited systems. To make this feature more convenient to the user, we would like to port our visualization over to a Stellaris game modification, where a user could toggle a ”lens” over their game.

Another worthwhile change to be made would be expanding our desirability calculations to include more game features. Some features we have yet to cover would be habitable worlds, which are integral to expansion, strategic resources, such as terraforming gases, and wormholes. This would allow us to better represent all the features of the game and accommodate more user preferences.

For our Interactive Visualization final project we made the Stellaris Universal Expansion Tool. This tool allows for new players to make better strategic decisions in the early game when it comes to the question of ”Where do I expand first?” By leveraging changeable attribute weights, we allow users to customize how they want to play the game, and how our tool can help them.

Academic References

Gratzl, Samuel et al. (2013). "LineUp: Visual Analysis of Multi-Attribute Rankings." In: *IEE Transactions on Visualization and Computer Graphics 19, no. 12*. ISSN: 2277–86. DOI: 10.1109/tvcg.2013.173.

Harrower, Mark and Cynthia A. Brewer (June 2003). "ColorBrewer.org: An Online Tool for Selecting Colour Schemes for Maps." In: *The Cartographic Journal* 40.1, pp. 27–37. DOI: 10.1179/000870403235002042.

Coding References

FooBar167 (Jan. 2018). *Tkinter canvas zoom + move/pan*. https://stackoverflow.com/a/48137257.

Kim, Michael (2019a). *Clausewitz*. https://github.com/PyPDX/clausewitz.

— (2019b). *Stellaris Save Visualizer*. https://github.com/PyPDX/stellaris-save-vid.