Class Specifications & Module Proposal Company B, Group 19

Authors: Dat, Milo Baran, Devang Sethi, Anurag Senapaty, and Luke Bridges

Classes:

1. **Random** - Dat

Description	A random number generator using the Xoshiro256++ algorithm. Provides methods for generating random values across multiple data types within specified ranges. Supports both seeded generation and time based seeding.
Similar Classes in the Standard Library	std::mt19937 / std::mt19937_64 std::uniform_int_distribution / std::uniform_real_distribution std::bernoulli_distribution std::rand() / std::srand()
Key Functions	Random(uint64_t seed = 0) double GetDouble(double min, double max) float GetFloat(float min, float max) int GetInt(int min, int max) int64_t GetInt64(int64_t min, int64_t max) uint32_t GetUInt(uint32_t min, uint32_t max) bool P(double probability) void Seed(uint64_t seed)
Error Conditions	Invalid ranges (min > max): Automatically swaps values to correct order. Equal min/max: Returns the constant value.
Expected Challenges	Seeding confusion - Users may not understand the difference between seeded (reproducible) vs time-based (non-deterministic) generation.
Compatible with Classes from Other Groups	BehaviorTree (Group 11): Decision making in nodes PathGenerator (Group 11): Add randomness to path finding or generate random routes

2. **WeightedSet** - Luke Bridges

Description	A WeightedSet<T> is a set where each element (of type T) has a (nonnegative, not necessarily integer) “weight”. To put things a bit more formally than the description on the course website, let w1, w2, …, wn be the weights of the elements. Then we should be able to index into the set so that inputting a number in the range [0, w1) gives us the first element, inputting a number in the range [w1, w1 + w2) gives us the second element, and so on. We also have a function GetRandomElement() which picks a random double between 0 and the sum of the weights and returns the element at that index. The probability that an element gets chosen is equal to its weight, so e.g. an element with weight 2 has twice the chance of being selected as an element with weight 1. (Note: above I was treating the elements as ordered. The data structure kind of forces them to be ordered–the element you get by indexing at 0 is the first, the element you get by indexing at the weight of the first element is the second, and so on. When the elements of type T are comparable we might try to make sure that their order in the WeightedSet corresponds to the sorted order. In general, though, we can’t guarantee that this order will mean much of anything, or that it behaves predictably under adding or removing elements. In any case, the order isn’t important for using GetRandomElement(), which is the main function we want to support.) My rough idea of how this will work is that we’ll have a binary tree where each node contains an element, the weight of that element, and the total weight of the whole subtree starting at that node. So, for example, the root node will have an element, the weight of that element, and the weight of all the elements in the tree. Then we’ll try to keep the tree balanced in the sense that the total weight of the left subtree should be close to the total weight of the right subtree. This should make the GetRandomElement() function O(log(n)).

Similar Classes in the Standard Library	In my opinion this is a somewhat unusual data structure which isn't really analogous to any of the STL containers (since it's optimized for GetRandomElement and not for more standard operations like checking membership (which std::set and std::unordered_set are optimized for) or indexing by keys (which std::map and std::unordered map are optimized for)), but there are still some rough equivalents. The closest equivalent (especially if we assume, as discussed in the "note" above, that the elements are ordered) is the std::set. (In fact, per CPPReference, "Sets are usually implemented as Red-black trees" (https://en.cppreference.com/w/cpp/container/set.html), and I'm planning on having a tree-like implementation, so there should be some similarity in the internals too.) In a weird way it's a bit like a vector: if every element has weight 1 then you could index at integers (e.g. GetElement(0) for the first element, GetElement(1) for the second, and so on) and sort of pretend that it's a vector.

Key Functions	Key (public) functions (note: error conditions for the “expected” are explained in the section below): ● std::expected<T> GetElement(double index) ○ takes in a number between 0 and the sum of the weights, returns an element based on the process in the description above ● std::expected<T> GetRandomElement() ○ randomly generates a double x in the range [0, sum of weights] and returns GetElement(x) ● bool Insert(T element, double weight): ○ inserts the given element with the given weight. Returns true if successful. If an element equal to the given one already exists, change the weight (a bit like how, if you add an existing key to a map, you still update the value). Could make it return an iterator to the inserted element like the version for UnorderedSet does, but I don’t see us needing that very often. ● bool Remove(T element): ○ removes the element, returning true if it was successful Might also be useful: ● std::expected<double> GetWeight(T element) ○ gets the weight of the given element ● std::expected<std::pair<double, double>> GetIndexRange(T element) ○ gets the lower and upper bounds of the range you can use to get the given element Those are the really important ones, I think. We mainly care about getRandomElement and of course about adding and removing elements, but a lot of functions that we’d usually associate with a “set” class (e.g. a function to test membership) aren’t useful for what we’re doing here. I won’t know exactly what private functions I’ll have until I’ve pinned down the implementation further. Probably some AVL/red-black tree style operations (e.g. rotations) would be useful assuming that I’m implementing this as some kind of balanced tree.

Error Conditions	<ul><li>For GetElement:<ul><li>index out of range [0, sum of weights]</li><li>set empty</li></ul></li><li>For GetRandomElement:<ul><li>set empty</li></ul></li><li>For Insert:<ul><li>weight is negative</li></ul></li><li>For Remove<ul><li>element does not exist</li></ul></li><li>For GetWeight, GetIndexRange<ul><li>element does not exist</li></ul></li></ul>
Expected Challenges	I could see this being a little confusing to use, due to how you can index in (unlike a set, say) but the indexing might behave unpredictably when you add or remove elements. I'd have to design the interface + docs in a way which makes it clear that GetRandomElement is the main intended use case. I remember AVL and red-black trees being the trickiest part of 331, so even reimplementing them without modifications could be a bit difficult, and implementing a similar data structure with a weirder balancing scheme is probably even harder. I'll definitely have to brush up on those.
Compatible with Classes from Other Groups	<ul><li>Not actually another group, but I'd like to use our group's Random class (specifically the GetDouble function) as part of GetRandomElement.</li><li>Maybe the "scripted agents" group could use it alongside ActionMap, e.g. they could have a WeightedSet of function names and use that to randomly select an action to take by calling Trigger(functionNameSet.GetRandomElement()).</li></ul>

3. **DataFileManager**- Devang Sethi

Description	A class to manage a data file where the user may want to periodically update its status. The user can log agents, states, events, timestamps etc in this file. The file will get periodically updated and will keep a log based on timestamps.
Similar Classes in the	Classes from the standard library that might be useful are: std::chrono - For measuring time;

Standard Library	std::vector - For storing column and row values;
Key Functions	DataFileManager(const std::string& filename) - Constructor; void Update() - Trigger stored functions to add a new row and update column values; void AddColumn(std::string& columnName) - Add a new column; void AddRow(const std::vector<std::string>& values) - Add a new row; void LogState(const std::string& stateName, const std::string& stateValue); - Logs the value of a state; void Flush() - Flushes the buffer; void Close() - Close the data file;
Error Conditions	**Programmer Errors:** Passing invalid parameters to functions such as AddColumn or AddRow; Registering two columns with the same name; **Potentially Recoverable Errors:** Calling Update() when the file/schema is blank; File not found; **User Errors:** Attempting to write to a closed file;
Expected Challenges	The schema for the data file is currently unknown. This will have to be discussed with the rest of the company; Making sure files are written to and read correctly without errors (I/O Errors); Structuring the CSV format and managing delimiters and new lines;
Compatible with Classes from Other Groups	This class should be compatible with the Data Log, Action Log, and Timer classes that are being created by Group 20; It can also be compatible with the ErrorManager class being created by Group 21; Within Group 19, this class will be compatible with StateGrid and StateGridPosition;

4. **StateGrid** - Anurag Senapaty

Description	A grid of tiles to represent the campus map we'll work on that tracks position of agents, can measure crowd density, and can affect our "traffic" simulation with regards to other agents. Needs to accurately simulate the map of campus using a tile class that can represent

	things such as paths, walls, stairs, rivers, bridges (Can expand). The tiles need to have a grid to allow multiple agents to exist on them and traverse them. We stick to a 3x3 grid inside the tile that is inside the Campus map for coordinates as opposed to either only one entity per tile or float based coordinates.
Similar Classes in the Standard Library	None
Key Functions	StateGrid: Int width Int height Vector<Vector<Tile>> tiles Stategrid(int width, int height, Vector<Vector<char>>: premadeMap) getTile(int row, int column) Enum Condition: Snowy Wet Damaged Perfect struct MetaData: #Makes it simple for int based coordinates Int movementModifier Condition condition Tile: Int row Int column Char symbol String name MetaData metadata vector<agent> agents Tile(int:row, int column, symbol char, string name, MetaData metaData) addAgent(Agent agent) getAgents() : sharedPointer vector<agents> getDensity()
Error Conditions	Null pointers to tiles or agents for a start
Expected Challenges	Enough complexity but not too much, needs to track relevant information that we will actually use and not too much but also not

	be too simple. Also a grid based on tiles that are based on an even smaller 3x3 grid might be challenging to keep track of. Also weather and dynamic states of paths and such, slipperiness, movement modifiers, danger levels, can pathing take into account density and dynamically change midway through for the agents.
Compatible with Classes from Other Groups	Unsure if the agent and UI teams will interface with this or if we're going to have a wrapper that simplifies the interface, likely the latter, so there should be little interaction outside out group

5. **StateGridPosition** - Milo Baran

Description	Tracks a position and orientation of a single individual agent within a StateGrid object. Furthermore, will allow for agents to move through the grid and determine distance to other positions on the grid.
Similar Classes in the Standard Library	std::pair - simple coordinate storage std::array - orientation storage

Key Functions	Constructor
	StateGridPosition(Point position, Direction direction_facing = Direction::North)
	
	Movements
	bool MoveForward(const StateGrid& grid);
	bool MoveBackward(const StateGrid& grid);
	void TurnLeft();
	void TurnRight();
	void TurnAround();
	bool IsValidMove(const StateGrid& grid);
	
	Distance
	double EuclideanDistance(const StateGridPosition& pos);
	
	Getters for position and direction
	Setter for direction

Error Conditions	Programmer errors: Invalid direction enum, negative coordinate values, Null references to StateGrid or StatePosition Recoverable errors: position given to constructor is out of bounds

	User errors: None, assuming the user has no control over agent movement. If the user can control the position of agents, errors would include invalid movements.
Expected Challenges	Because this class can be implemented in many ways, solidifying details will be challenging. Deciding how to represent the orientation of the agent, what types of movements are allowed and how the class will interface with other classes are all complex design decisions.
Compatible with Classes from Other Groups	PathGenerator - need to coordinate representation of position and valid movement methods Point - may be the way we can represent position WorldPath - WorldPath and StateGridPosition should share similar representations of position

Module Proposal:

Group 19 proposes to design an interaction-heavy simulation of a college campus. The world environment will be a grid-based map of the campus. This world will be populated by various agents. These agents could be students, cars, buses etc. The agents, depending on what they are, can navigate on certain tiles. For example, student agents may navigate on sidewalks, grass etc. Meanwhile, vehicle agents may navigate on streets. Each agent could be programmed to have a certain goal. For example, a student agent might want to travel from one end of the campus to another in order to get to class on time. We will simulate how crowds of students move and interact with each other. We can also gather data about traffic jams and choke points on campus. Obstacles such as rain and snow can be added to study mobility as well. We will also be able to simulate traffic by altering the map of campus (closing streets, opening bridges etc).