GAI-CA2

Colin Zenner, Dominik Koschik, Luca Ahlf, Yan Wittmann

January 2025

1 Motivation and Idea

This project has the goal to demonstrate an AI technique suitable for educational or self-learning contexts. We chose to expand upon the original concept of the "Food Gatherer" model developed by Professor Eckert during our course by introducing more complex mechanics. We wanted to construct this autonomous survival simulation by creating a more complex behavior tree. The result is "Deserted Island Survivor", a game in which the player must navigate a deserted island, manage resources, and ultimately escape by repairing a boat. This document outlines the design, mechanics, and implementation of this project.

1.1 Inspiration: Food Gatherer

The original "Food Gatherer" model served as the foundation for our project. In this model, two entities compete to gather food and water on a small 2D grid. The players are able to move one tile per turn, and can collect food by positioning themselves on a tile that has a berry bush or water from nearby water tiles. Berry bushes regrow automatically after a random amount of turns, water stays always available. The entities lose health if they fail to maintain their levels of hunger and thirst, and the game ends when one entity dies from starvation or thirst, or players can win by collecting a certain amount of food.

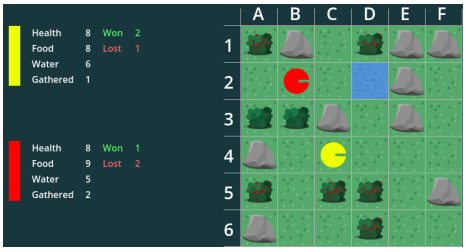

Figure 1: The original Food Gatherer model.

2 Concept and Design Decisions

2.1 Initial Development and Brainstorming

The foundation of our project began with a brainstorming session aimed at understanding the realistic survival needs of a stranded character:

- What does a Person need to survive on a deserted island?
- How can we translate these survival priorities into a behavior tree the character AI can follow?
- How should the character balance his short-term needs (e.g., hunger, temperature) with his long-term goal of building the boat and escape from the island?

2.2 Defining the Characters Needs

To ensure the game felt realistic, we modeled the survival needs the AI would address:

- **Hunger:**
  Hunger was defined as the most critical need. If the hunger value of the character drops below a certain threshold, the AI would prioritize finding and eating food to avoid starvation.
- **Temperature:**
  Environmental conditions can differ on the island depending on where the survivor is exploring. When the character tries to walk on cold terrain, he needs to find a stick that he can use as a torch to use this resistance to keep his temperature stable and not get damaged by the cold.
- **Long-Term Objective (Escape):**
  The overall goal of the character is to escape from the island. To achieve this, the character has to find a set amount of boat parts scattered around the island to build the boat. Only if the character's other critical goals are fulfilled will he work towards this long-term goal.
- **Exploring:**
  If critical needs like hunger are met and there is no boat part in sight, the AI should explore the island in search of boat parts, moving to different locations around the map on his own. The exploration algorithm should act in a smart way, making the AI feel realistic and unpredictable.

2.3 The Behavior Tree Design

To organize our ideas, we used a collaborative Figma board to visualize an early version of this behavior tree (Figure 2). The visual approach helped us

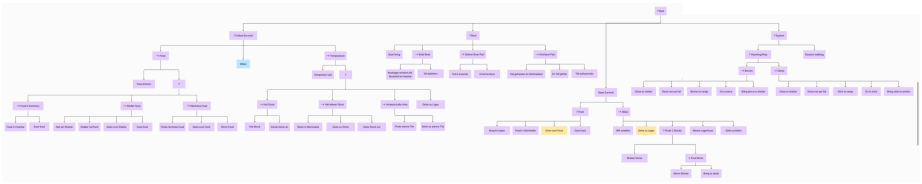

Figure 2: The initial concept for the behavior tree.

to structure the characters' behavior and determine how the AI should react in various situations.

Our behavior tree forms the central logic of the game, structuring the character's actions hierarchically to ensure critical needs are always prioritized over secondary tasks. Our behavior tree enables dynamic and reactive AI by constantly evaluating conditions and re-prioritizing tasks based on the character's current state. We designed the tree with four top-level selectors, each corresponding to a category of task sequences that the character could perform if certain conditions are met. These selectors were organized hierarchically to reflect the urgency of various needs:

1. **Critical Survival (highest priority):**
   This behaviour describes an "emergency" state of the survivor that requires urgent actions. For this the survivor checks if his food and temperature parameter are below a critical threshold. If this is true he will try to increase the temperature or saturation with items he currently carries, that are located next to him or that he stashed in his encampment.
   for example: Is Hunger Critical? If yes, it will check in this order:
   - Is there food in inventory? If so, eat it.
   - Is the shelter close and does it have food? If so, go to it and eat it.
   - Otherwise, it will find the closest food and eat it.
2. **Boat (Second priority):**
   The long-term goal of the simulation is for the character to escape the island by building a boat. This selector handles tasks relating the boat-building process as well as the search for boat parts. If all boat parts have been collected, he will pickup the boat and get to the next pier.
   - Boat finished?
   - if no, build boat with delivered boat part?
   - if no, Boat part in inventory to deliver?
   - if no, search for boat parts in vision range.
3. **Base Survival (Third priority):**
   This selector deals with maintaining the characters general well-being when he is not under immediate threat. It ensures the character remains

adequately prepared for achieving his long-term goal without permanently switching between achieving this goal and being in a critical state. Since the Survivor is a human being, this behavior contains also his need to sleep, if it's nighttime.

- Is hunger sufficiently low (but not critical)?
- if yes, search for food in vision range, take it and eat it.

- Is nighttime ?
- if yes, go back to camp, find 2 sticks to light a fire either from stash or from surrounding areas and go to sleep.

4. **Explore (Lowest Priority)**
   When all other needs are fulfilled, the AI enters exploration mode. This ensures the character continues to discover new areas of the island by walking into a random direction in search for boat parts. In exploration mode, the character also has the possibility to prepare for his adventures by delivering items like berrys or sticks to his shelter to stash them for later use.

2.4 Behaviour Tree Flow

A couple of examples of the adaptive system prioritizing factors depending on the situation in our behavior tree:

- If the character is exploring the island and suddenly enters a state of critical hunger, the behavior tree immediately shifts focus to the critical Survival selector, interrupting exploration.
- Similarly, if the character encounters a boat part while addressing base survival needs or is exploring, the tree re-prioritizes the Boat selector to collect and deliver boat parts.

This hierarchy prevents the AI from getting "stuck" in irrelevant tasks and ensures the balance of short-term survival with completing the long-term objective.

3 Implementation details

Our implementation focused on creating a robust behavior tree system for character's AI. While the theoretical structure supported modularity and dynamic decision-making, its practical implementation revealed several challenges. A problem we faced regularely that demanded to be solved was enabling the character to dynamically respond to higher-priority needs while he was in the middle of executing a lower-priority task. This led to the creation of the `SUCCESS_STOP` status.

3.1 Dynamic Prioritization Using `SUCCESS_STOP`

3.1.1 Limitations of Traditional Task Statuses

The core challenge emerged from a pattern we wanted to follow: allowing task interruption for urgent needs. We tried achieving this using the more conventional status indicators:

1. **SUCCESS**: A Task was completed successfully.
2. **FAILURE**: A task could not be completed.
3. **RUNNING**: A Task is still in progress.

The RUNNING state would be the most obvious choice at first glance. However, we soon realized that it created problematic execution locks. For instance, a character collecting boat parts couldn't interrupt this task when starving. Similarly, new environmental opportunities (e.g., nearby spawned food) couldn't redirect pathfinding until the current tasks was completed.

Using SUCCESS would mark tasks as completed, even if they were only partially executed. For example, if the character was moving towards an item, the task would already be marked as a success despite the character still moving to the item and they would fail to pick up the item.

Returning FAILURE for an interrupted task would incorrectly signal that the task could not be completed. This could result in the AI running multiple different behaviors in the same turn, which is not only against our philosophy, but these behaviors could contradict each other, leading to a deadlock.

3.1.2 The `SUCCESS_STOP` Status

We introduced `SUCCESS_STOP` to enable a temporary task suspension for higher priorities. This status allows tasks like `GO_TO` to indicate:

- A partial success in reaching the goal
- Causing an intentional execution halt in the parent task
- Avoidance of immediate re-evaluation

In practice, all it does is to tell the parent task (a sequence or selector) to not run the next child, just as it would with `RUNNING`, but instead of propagating the `RUNNING` status upwards, it will overwrite both the child's and parent statuses to `SUCCESS` and then return to the behavior above.

This seemingly simple change led to us being able to model much more complex interactions and instant reactions to changes in the world state.

3.2 Cached Pathfinding

The `SUCCESS_STOP` status introduced in section 3.1.2 comes with a couple of challenges when used in a task that uses pathfinding. Since we are not using `RUNNING`, the path previously calculated might no longer be valid, as other tasks may have had the chance to run in the meantime. That means, we can't just store a global path in the blackboard that the behavior generally uses to walk along. So, we will theoretically have to recalculate the path every game tick from anew.

But, calculating paths is an expensive operation. Our solution combines caching with dynamic path validation:

How the caching mechanism works

1. If a task requests a path from a starting point to an end point, the navigation algorithm first checks in the cache whether this task has ever calculated a path between two points before. If not, it simply calculates one using the provided parameters and stores the resulting path in the cache using a unique key dedicated to the specific task requesting the path (if any way found at all).

2. If a path was found in the cache, it has to be validated as to whether it is still up-to-date. A path becomes invalid if:

   - The target location is no longer on the path.
   - The character is no longer on the path (it must have moved between executions).
   - Any position along the path can no longer be walked across due to map changes.

3. If a cached path is invalid, it is recalculated and stored in the cache using the task's key,

This process works very well in combination with `SUCCESS_STOP`, allowing tasks to dynamically adjust paths in response to environmental changes, such as new spawning food, without having to recalculate identical paths again and again.

3.3 Behavior Tree Visualization

To provide a clear and intuitive visualization of our behavior tree, we implemented a dynamic graph using Godot's GraphEdit. It offers a real-time, interactive view of the tree structure, highlighting active tasks and their latest status and their custom descriptions. Figure 3 shows the visualized tree as generated in the simulation.

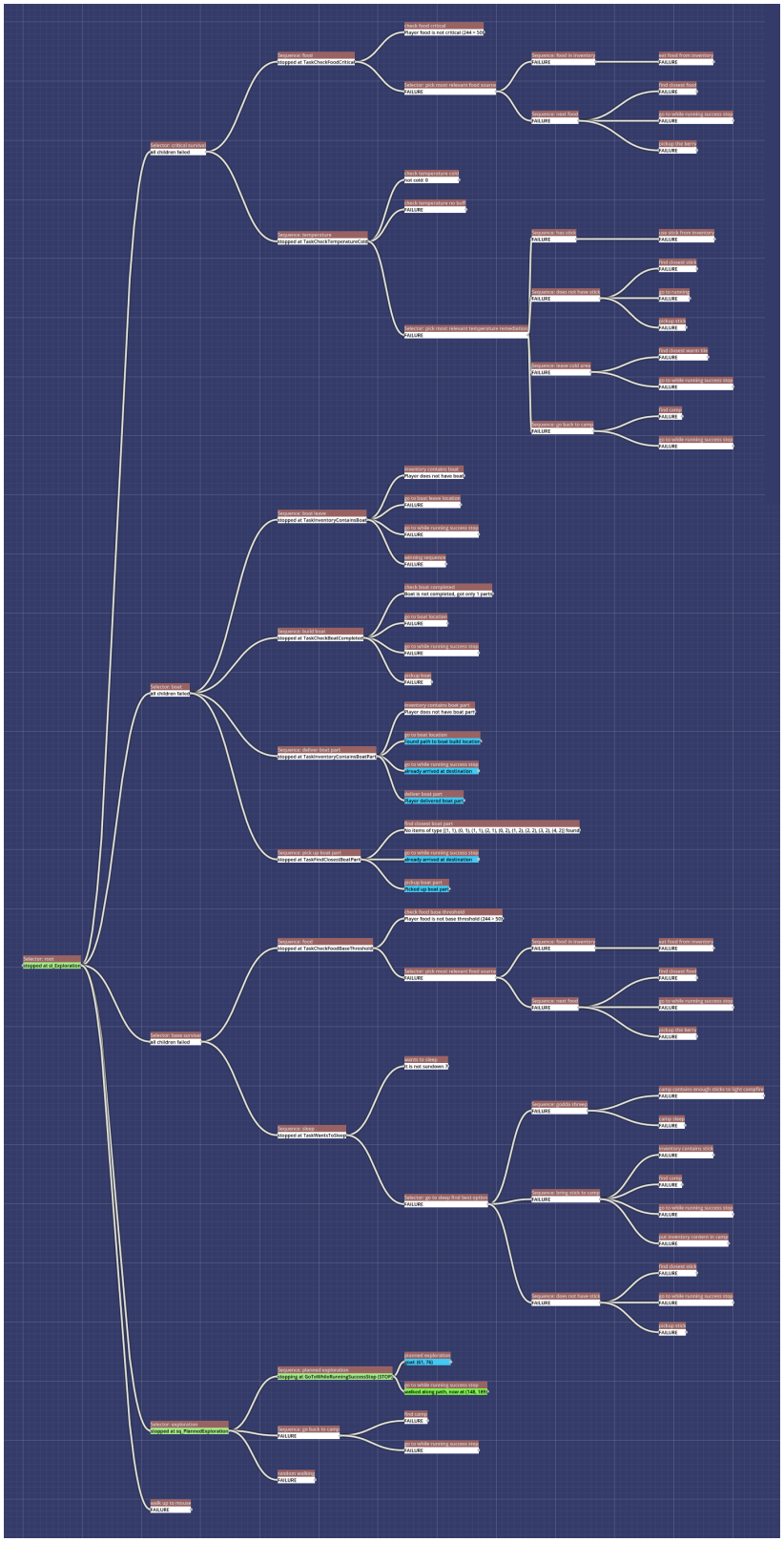

Figure 3: The visualized tree, as layouted and seen in the simulation.

3.3.1 Tree Layout Algorithm

Coming up with this algorithm proved a challenge, even though it's final algorithm may look simple. the issue was arranging them in a way such that they do not overlap and do not cause any connections to cross lines, which is always possible, since a tree is planar. Figure 4 shows an example of a resulting node distribution that the algorithm creates.

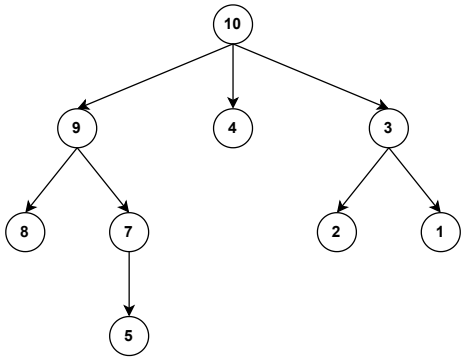

Figure 4: Resulting graph and post-order traversal.

The algorithm we developed determines the node coordinates by recursively iterating through the nodes using post-order traversal (see numbers on nodes in figure 4), processing them in this order. This means that all child nodes of a parent node are called first, then the parent after that. Knowing this, we can guarantee that when processing a node, that all child nodes have already been positioned.

Since our tree is built growing from left-to-right, we can easily determine the x-coordinate of the nodes: $node_x = node\_depth * scale_x + offset_x$.

The y-coordinate is more challenging. For it, the algorithm differentiates the two scenarios of whether it processes a leaf, or a parent node. These two scenarios are described in the following paragraphs alongside the figure 5.

Leafs are the first to be processed due to the traversal method. A global counter *leaves* is used here, which counts the amount of leaf nodes added to the graph, which is incremented after each processed leaf node. This counter can be used as the node index on the y-axis: $node_y = leaves * scale_y + offset_y$. This can be seen in figure 5, where each green leaf node is one tick further up on the y-axis.

Parent nodes are only processed after their child nodes are already positioned on the graph. This means that we can use their coordinates to position the parent nodes relative to them. To be specific: the average y-coordinate of all child nodes is calculated and used as it's own: $node_y = \frac{\sum_{i=1}^{n} \text{child}_{i,y}}{n}$. This is shown

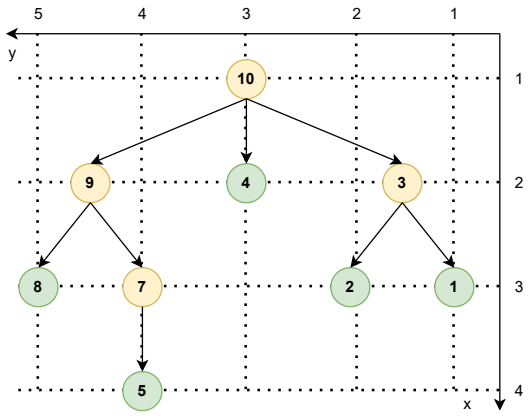

Figure 5: Node positioning on the graph.

in the figure, where each parent node is centered above all it's child nodes.

This is done until all nodes up to the root node have been positioned.

3.3.2 Real-Time visualization updates

Since the game will be used in a self-learning or presentation-style context, it is important for it to update automatically, to allow the guests to more easily follow the happenings of the simulation.

- **Task Statuses:** The visualization dynamically updates to show the current state and a custom description of each task in the tree. This is simply done by updating the content of the first label.
- **Colored Nodes:** Based on the status of the task, a node is colored differently to immediately make it obvious to a viewer which path was chosen.
- **Auto-scrolling:** The leaf task that has been selected to run by returning `SUCCESS` will automatically scrolled to in the graph view. the visualization centers the view on this node to allow more easily demonstrating the execution flow of the tree.

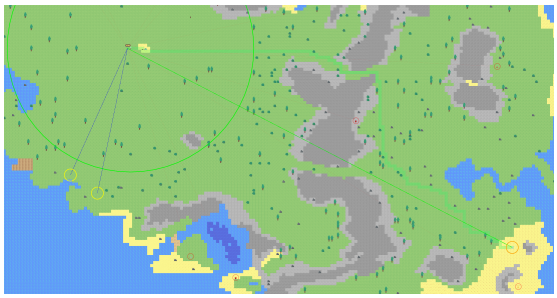

Figure 6: The character explores the island in search of a new boat part.

4 Conclusion & iExpo Feedback

Figure 6 is a screenshot of the game, as the character explores the island.

The development of Deserted Island Escape proved to be an ambitious but rewarding experience. During the project, we experienced first-hand how complex a behavior tree of this size can become. As our implementation progressed, we encountered limitations that required innovative workarounds, such as the `SUCCESS_STOP` status described previously.

The presentation of our work at the iExpo provided us with valuable and mostly positive and encouraging feedback. We were able to demonstrate our simulation to many guests, professors, and fellow students alike.
While the overall system was well received, some feedback pointed towards the games balancing. In our presentation, the survivor never actually struggled to survive. Since our primary focus was on demonstrating the AI's functionality rather than fine-tuning, balancing was not a top priority at the time. This was an intentional trade-off we decided to take, due to the short amount of time we had, to put our efforts into making sure that our simulation could be understood by anyone, using easy-to-understand visualizations.

This project significantly deepened our understanding of AI in video games. The difficulties we encountered in implementing dynamic task prioritization, optimizing pathfinding, and planning visualizations specifically for such a use-case greatly enhanced our understanding of these concepts and will surely be of great use in future projects.