Flexible Pathfinding System for 3D Environments

Technical Documentation

Introduction

This document provides an in-depth look at the Flexible Pathfinding System tool. The system is built upon **Unity version 2022.3.52f1** and consists of two core modules: Link Generation and Pathfinding Logic. This documentation is intended for developers who aim to modify, or expand the tool. It includes details about the architecture, algorithms, and crucial components used in the system.

Link Generation **2**
- Overview 2
- Data Format - Edge class 3
  - Purpose 3
  - Key Properties 3
  - Key Methods 4
- Data Fromat - LinkData class 4
  - Purpose 4
  - Key Properties 4
  - Key Methods 5
- Key Component: NavLinkGenerator Class 5
  - Purpose: 5
  - Key Properties: 6
  - Key Methods: 7
- Management and Utility Methods: 10
  - Edges: 10
  - Links: 10

Pathfinding System **11**
- Overview 11
- Character Related Structures: 11
  - Player Controller Class 11
  - PhysicalStatsLogic Class 12
- Key Component: NavLinkManager Class 12
  - Purpose 13
  - Key Properties 13
  - Key Methods 13
  - NavRequest Struct 14

Extending and Customizing **14**
- Edge Division and Connection Alignment Adjustments 14
  - Overview: 14
  - Idea of Implementation 15
- Short Edge Grouping Enhancements 15

Overview:.. 15
Supporting Multiple NavMeshes for Diverse Character Types..15
Overview.. 15
Idea of Implementation..15
Asynchronous Pathfinding Request Processing.. 16
Overview:.. 16

Link Generation

Overview

The Link Generation system utilizes Unity's built-in navigation mesh (NavMesh) to dynamically create traversable connections between edges in a 3D environment. This process enhances the navigation capabilities of NavMeshAgents by largely increasing the number of possible routes to every point. The connections generated by the system are either edge-to-edge or edge-to-floor (dropdown) links, each tailored to the user’s settings.

Link Generation system is contained within two components:

- **NavLinkGenerator -** the functional component, responsible for the process of detecting, evaluating and creating navigation mesh links
- **NavLinkManager** - the controller for the system, which stores NavMeshLinks wrapped in **LinkData** class and includes simple management functions.

The process of link generation executed in **NavLinkGenerator** begins by extracting vertex and triangle data from Unity's NavMesh using the NavMesh.CalculateTriangulation() function. The system then identifies edges by analyzing vertex pairs and retains only the outermost edges of the mesh. These edges are converted into instances of a custom **Edge** data format and stored for further processing.

An optional grouping step combines smaller, adjacent edges into groups if their lengths fall below a user-defined threshold. This ensures that even complex, high resolution mesh geometries can be simplified into logical units, as well as gives the possibility of reducing the unwanted density of connections in the scene.

The edge generation is a process which has to occur before link generation can begin. If the edges have been found this process will not be repeated until the level geometry or the edge generation settings has been changed.
Once edges are prepared, the system begins the creation of edge-to-edge connections. It takes each edge and evaluates the path from it to every other edge on the scene. For this type of connections, the system attempts to connect each edge to others within range, applying several constraints:

- Links cannot duplicate existing connections.
- Angles between the edges must remain within user-defined thresholds, avoiding unrealistic or impractical jumps.

- The path of the jump must be clear of obstructions, verified through raycasting or box casting depending on the settings.

For edge-to-floor (dropdown) connections, the system evaluates each edge to check for nearby surfaces below. This process also includes checks for drop distances and collision paths.

All validated connections are stored as instances of the **LinkData** class in the **NavLinkManager**, which encapsulates the link's key properties and provides management functionalities like add, update and delete which are capable of incorporating manually added links to the system.

Data Format - Edge class

The Edge class represents the outermost edges of the navigation mesh and contains essential data and functionality for generating connections between these edges. Below is an explanation of its purpose, key properties, and notable methods.

Purpose

1. Processing edges detected from the NavMesh.
2. Identifying suitable positions for link connections.
3. Identifying suitable positions and directions for collision checks between connections.
4. Storing intermediate calculations.

Key Properties

Basic Edge Data:

- Vector3 start and Vector3 end: Start and end points of the edge in world space.
- float length: The length of the edge, calculated from the start and end points.
- Vector3 edgeSurfaceNormal: The normal vector of the surface the edge is part of.

Falloff and Connection Points:

- List<Vector3> connectionPoint: Points along the edge indicating where the links should connect. With current implementation only one point (middle of the edge) is being created, however depending on the needs each edge can have multiple connection points which gives more possibilities for connections but increase computational complexity
- Vector3 falloffDirection: A vector perpendicular to the edge, pointing outward where the mesh ends indicating a steep fall.
- List<Vector3> falloffPoint: A complementary list to connectionPoint. Each falloffPoint contains the position of a connectionPoint shifted by a large enough value to be hovering over the edge in the direction of the falloffDirection vector. From this

position collision checks can be made downwards without worrying about colliding with the geometry the mesh is generated on.

- public bool hasPivotPoint: A flag indicating whether or not any connection points have been found on this edge.

Key Methods

- Main Constructor - public Edge(Vector3 startPoint, Vector3 endPoint, Vector3 surfaceNormal, (NavMeshBuildSettings settings, float correction) navMesh):
    - Calculates edge-specific data such as length, falloffDirection, and default connection/falloff points during initialization.
    - Uses Unity's NavMeshBuildSettings and custom correction to calculate agent-specific parameters (falloffDistance and pivotCheckDistance).
    - Navigation mesh settings are taken in the form of a tuple along a custom correction which patches a common Unity bug where the mesh is generated inside the geometry instead of on top of it.

- CalculateFalloffPivots(float falloffDistance, float pivotCheckDistance, float correction):
    - Determines valid falloff points by testing perpendicular pivots on both sides of the edge.
    - Uses isPivorValid() function to determine which way the edge is facing and to assign proper falloffPoint to each connectionPoint

- IsPivotValid(Vector3 pivot, Vector3 point):
    - Uses raycasts to check whether a given pivot is clear of obstacles and correctly positioned in a valid space - hovering just over the edge of geometry.

Data Fromat - LinkData class

The LinkData class represents individual links recognised by the system. It serves as a wrapper around Unity's built-in NavMeshLink component, providing additional metadata and functionality to evaluate and update links.

Purpose

- Storage and management of links present in the scene.

Key Properties

- Vector3 Start and Vector3 End: Computed properties that provide the world-space start and end positions of the link, including offsets.

- float length: The straight-line distance between the link's start and end points, kept in memory for faster path calculation.

- NavMeshLink linkComponent: A reference to Unity's built-in NavMeshLink component.

- Vector3 linkObjectPosition: The position of the NavMeshLink object in world space, used to calculate offsets for the start and end points.

- int ogCostModifier: Stores the original cost modifier of the link, enabling the system to revert changes if needed.

- bool wasGenerated: Tracks whether the link was auto-generated by the system or added manually.

Key Methods

- Constructor - public LinkData(Vector3 origin, NavMeshLink _linkComponent, bool generated)
  - Assigns the NavMeshLink reference and position and calculates length based on those parameters.
  - Stores the original cost modifier for potential reversion.

- RecalculateParameters():
  - Updates the link's object position and length.
  - Should be used when links are moved to align with changes in the scene.

- RevertCost():
  - Resets the cost modifier of the NavMeshLink to its original value.
  - Is used with a weight-based version of pathfinding system (disabled in current implementation - see)

- LinkActivate(bool activate):
  - Enables or disables the NavMeshLink object.
  - Is used with an activation-based version of pathfinding system (see)

Key Component: NavLinkGenerator Class

The NavMeshLinksGenerator is the central class responsible for creating and managing navigation links in the system. It acts as a controller that processes edges, evaluates conditions for link generation, and manages link objects within the Unity scene.

Purpose:

1. Processing the Unity NavMesh data (NavMeshSurface) to detect and store edges as a custom data class Edge.
2. Generating and configuration of navigation links (NavMeshLink) based on edge properties and user-defined settings.
3. Storing adjustable parameters for customizing edge detection, link generation, and raycast logic.

Key Properties:

Object References and Prefabs (can be automatically assigned - see Auto-Assignment Logic)

- **NavMeshSurface**: The primary component for accessing baked NavMesh data.
- **NavLinkManager**: Manages organization and updating of generated links.
- **standardLinkPrefab** and **dropDownLinkPrefab**: Prefabs for standard and dropdown links, respectively.
- Transform **generatedLinksGroup**: References a scene object which will act as a container for all generated links.

Edge Detection and Configuration Settings

- Parameters float **min / maxEdgeLength** and int **min / maxGroupSize** control how edges are detected and grouped. Grouping is attempted for edges of length below minEdgeLength.

Link Generation Settings

- Distance related parameters: float **maxEdgeLinkDistance**, float **shortLinkDistance** and float **maxDropDownLinkDistance** define the spatial range for connecting edges. Links which are too long will not be generated, links which are shorter will be generated at more flexible angles (permissive angles).

- Angle restriction parameters:

    - For Dropdown (edge-to-floor) links: **dropDownSteepnessModifier** defines the direction Y parameter of dropdown links. **dropDownLinkAngles** include potential rotation in the Y axis for searching dropdown links.

    - For Standard (edge-to-edge) links: angle restrictions are divided into **strict** and **permissive** categories (for normal and shorter links accordingly). **AngleRestrictionForward** and **AngleRestrictionUpward** ensure links adhere to realistic traversal constraints.

Raycasting Settings

- bool **enableJumpArcRayCasts** toggles advanced collision checks for links using a box cast shaped by the agent's size and jump arc.
- float **maxjumpArcHeight** configures the height of the jump arc for collision checks.

Internal Variables

- List<Edge> **edges** contains a list of all valid edges and grouped edges in the scene.
- Vector3[] **vertices** and int[] **pairIndices** are internal representations of the NavMesh geometry used for edge detection.

- (NavMeshBuildSettings settings, float correction) **navMesh** holds all settings of a current nav mesh as well as a custom correction which patches a common Unity bug where the mesh is generated inside the geometry instead of on top of it. The value of the correction can be set by adjusting float **navMeshCheckCorrection**.
- custom struct **EdgeParameters** holds the information on previously set parameters regarding edge detection.

Key Methods:

Auto-Assignment Logic

- **bool AutoAssign()**
  - Ensures that critical components and prefabs are properly configured.
  - Finds and necessary parameters in the scene, in components or in files by using **FindAssetPathByName()** function, depending on the field and assigns them to appropriate fields from Object References and Prefabs category.

Edge Detection Logic

Edge detection is a first step towards generating links. It processes the geometry of the baked NavMesh to identify valid edges and organize them for further analysis.

- **GetCurrentNavMeshSettings()**
  - Retrieves the settings for the current agent type used in the NavMeshSurface and assigns navMeshCheckCorrection as a correction value.

- **FindEdges()**
  - This is the main method for detecting edges in the NavMesh.
  - Uses NavMesh triangulation to extract vertices and triangle connections from the NavMesh and identifies outermost edges by checking shared edges between triangles.
  - Steps:
    i. Extract vertices and triangle indices from NavMesh using NavMesh.CalculateTriangulation() method.
    ii. Uses the **PairToEdge()** method to evaluate all potential edges and leave only edges of NavMesh segments.
    iii. Groups short edges into lists using **GroupShortEdges()** and merges each list into one representative edge using **GroupRepresentative()**.
    iv. (Optional) Instantiates debug point prefabs at relevant points for each edge for visual inspection.
    v. Removes all invalid edges that have no found connection points or could not be grouped.

- **PairToEdge(int n1, int n2, int n3)**

    - Evaluates 3 pairs of vertices forming a triangle on the NavMesh.
    - Uses these edges to calculate the normal of the triangle surface.
    - Adds each edge as the dedicated Edge class to the list and removes duplicates (outermost edges would not have any duplicates as they are included in no more than 1 triangle)

- **List<List<Edge>> GroupShortEdges()**
    - Finds all edges shorter than the minEdgeLength and groups only those which are connected, forming clusters of short edges.
    - Discards clusters smaller than minGroupSize and returns the list of valid groups.

- **Edge GroupRepresentative(List<Edge> group)**
    - Takes a group of edges and chooses the best representative by how well its falloffDirection aligns with the average falloffDirection of the group.
    - Returns a new Edge based on the best representative but shifts its falloffDirection by summing it with average falloffDirection.

Link Generation Logic

The Link Generation process is responsible for creating navigation links between detected edges. It begins by validating the current edge data and proceeds to create links between eligible edges and dropdowns.

- **CheckCreateConditions()**
    - Ensures the edge data is up-to-date before link generation
    - Recalculates edges by calling FindEdges() if:
        i. No edges exist
        ii. User-configured parameters for edge detection have changed
        iii. Vertices of NavMesh have changed. Verified by VerticesChanged() method
    - In case none of the above conditions are met the edges are not recalculated.

- **bool VerticesChanged()**
    - Compares the current NavMesh vertices with the previously stored ones. If their count or positions differ, this method returns true.

- **CreateLinks() -** The core function for generating links
    - Calls CheckCreateConditions() to ensure edge data is valid and updates edge parameters future comparisons.

    - Prepares the generatedLinksGroup (a container for generated links) and ensures the standardLinkPrefab is assigned.
    - Iterates through each edge and compares it with every other edge:
        i. Skips self-comparisons and checks if a link between the edges already exists by calling LinkExists()
        ii. Displays a progress bar to indicate the process status.
        iii. Using ValidConnectionExists() checks for valid connections between the edges.
            1. If a connection is valid, it instantiates a link prefab at the edge's connection. (All link prefabs should have their origin at the start point)
            2. Sets the endpoint of the link to the corresponding target edge's connection point.
            3. Updates the link and organizes it under the generatedLinksGroup.
            4. Adds the new link to the navLinkManager.
    - Calls AddDropDownLink() to attempt to generate dropdown links for each edge.

  Note: The loop is wrapped in a try-finally block to ensure the editor progress bar is cleared after execution.)

- **bool ValidConnectionExists(Edge edge, Edge targetEdge, out int beginIndex, out int endIndex)** - core function for link validation
    - Determines whether a valid navigation link can be created between two edges. Loops through each pair of their falloff points, checking spatial, angular, and collision constraints.
    - Spatial Constraints:
        i. distance of the link can not exceed maxEdgeLinkDistance.
    - Angular Constraints:
        i. (for links shorter or equal to shortLinkDistance) evaluates both edges for alignment using permissiveAngleRestrictionUpward - angle relative to edge surface normal and permissiveAngleRestrictionForward - angle relative to the edge falloffDirection projected onto XZ plane.
        ii. (for links longer than shortLinkDistance) makes the same calculations this time using standardAngleRestrictionUpward and standardAngleRestrictionForward to filter connections.
    - Collision Constraints (custom box cast):
        i. If enableJumpArcRayCasts is enabled, performs bidirectional customized box casts to simulate jump arcs collisions.
        ii. The box shape is dependent on connection steepness, character dimensions and maxjumpArcHeight.
        iii. To avoid detecting collisions with the terrain the edges are a part of the cast is performed in a segment from 0.1 to 0.9 of the whole connection (indicated by 0.1 * distance shift in box center, as well as 0.8 modifier on the maxDistance parameter of the cast)
    - Collision Constraints (standard)

        i. Performs a bidirectional raycast to check for obstacles on the way of the connection
    - If all requirements are fulfilled for the link made between given connection points of each edge the method returns true and outputs the indices of the successful connection points.

Management and Utility Methods:

Edges:

In NavMeshLinksGenerator class, apart from previously mentioned optional step IV in FindEdges() there is a method to showcase edge connection point placement and their directions:

- **int HighlightEdgeDirections()**
    - Draws debug lines for each connection point on the edges. The lines extend in the direction of the edge's falloff vector.
    - Returns the number of edges highlighted.

Links:

Link management is handled by the controller class **NavLinkManager** and includes following functionalities:

- **AddLink(LinkData newLink)**
    - Adds following LinkData into the navLinks list

- **(int, int) UpdateLinks()**
    - Updates the navLinks list by removing entries whose linkComponent is null and recalculating parameters for existing links.
    - Scans all NavMeshLink components in the scene and checks whether they are already tracked in the navLinks list. If not, it creates a new LinkData entry for each untracked link with wasGenerated property set to false.
    - Returns a tuple containing the count of newly recognized links and the count of deleted ones.

- **DeleteAllLinks()**
    - Purges all links that were generated by the system (where wasGenerated == true) by iterating through the navLinks list in reverse, ensuring safe removal of elements.
    - Depending on whether the editor is in play mode or edit mode, the links are destroyed using either Destroy or DestroyImmediate.

- **DeleteLink(NavMeshLink delete)**
    - Removes links containing specified link instance as well as all links without any link instance.

Additionally every link instance generated by the system is wrapped with a SmartNavMeshLink Class which adds additional deleting functionalities.

- **SmartNavMeshLink Class**
  - Extends NavMeshLink to add cleanup functionality when a link is destroyed.
  - In OnDestroy() method attempts to remove the link from the navLinks list of the NavLinkManager using DeleteLink(NavMeshLink delete) method.

Pathfinding System

Overview

The Pathfinding System integrates physical statistics of agents into navigation, enabling path calculation over a NavMesh that includes dynamically generated links. The system operates under the NavLinkManager class, which acts as a centralized hub for managing link traversal and customization.

The NavLinkManager processes requests from character controllers, and takes character parameters, such as jump height, drop height, and jump distance in order to filter out links that an agent cannot traverse. In current implementation only the subset of links within the agent's capabilities is used during pathfinding. Every other link is disabled for the duration of calculating path for the agent.

With this method the system allows to create multiple subtypes of agents which can move differently on the same NavMesh. The implementation of path requests is structured in the form of a queue in order to allow potential multithreading solutions.

Character Related Structures:

Player Controller Class

Purpose:

1. Showcasing an example implementation of a character controller that integrates with the Pathfinding System.
2. Facilitating pathfinding requests using mouse input.

Key Properties:

- MouseButton: Enum representing which mouse button triggers a pathfinding request.
- PhysicalStatsLogic: A reference to the agent's physical statistics, used to filter valid links during pathfinding.
- Camera mainCamera: A reference to the camera from which screen-space mouse clicks are projected into world-space rays.

Key Methods:

- **Start()**
    - Initializes the reference to the agent's PhysicalStatsLogic component if not manually assigned. Displays a warning if the component is missing.
- **Update()**
    - Detects mouse input for the specified button and projects a ray from the camera through the mouse position.
    - Upon detecting a surface, it calls NavLinkManager.Instance.RequestPath() with the agent's physical stats and the hit point as the destination.

PhysicalStatsLogic Class

Purpose:

1. Storing physical attributes of the agent
2. (Optional) Providing additional safeguards for the path traversal by validating links dynamically.

Key Properties:

- **float maxJumpHeight**: The maximum height the agent can jump.
- **float maxJumpDistance**: The maximum horizontal distance the agent can jump.
- **float maxDropDistance**: The maximum height the agent can safely drop.
- **NavMeshAgent agent**: A reference to the Unity NavMeshAgent component responsible for movement.

Key Methods:

- **Update()**
    - Monitors agent's traversal of navigation mesh links (isOnOffMeshLink).
    - (Optional) Validates links during traversal using QuickLinkValid() method.
- **bool QuickLinkValid()**
    - Validates whether the link the agent is currently traversing is within its physical capabilities. Returns true if valid, false otherwise.
- **StopAndRepath()**
    - Stops the agent's current movement, resets its path, and repositions it at the link's starting point.

Key Component: NavLinkManager Class

Aside from links management functionalities the NavLinkManager class serves as the central controller for the Pathfinding System. It processes pathfinding requests, evaluates agent-specific physical statistics, and ensures only valid links are used during path calculations.

Its structure can facilitate asynchronous request processing, however this functionality is disabled in current implementation.

Purpose

- Management and storage of NavMeshLinks included in the scene.
- Facilitating pathfinding requests, dynamically enabling or disabling links based on agent capabilities.

Key Properties

- **static NavLinkManager Instance**
    - Singleton instance of the NavLinkManager. Ensures a single point of control accessible for all NavMeshAgents in the scene.
- **List<LinkData> navLinks**
    - Stores information on links included in the current NavMesh.
- **Queue<NavRequest> requestQueue**
    - Stores pathfinding requests from multiple agents, ensuring sequential processing.
- **bool isProcessing**
    - Indicates whether the manager is currently processing a pathfinding request.

Key Methods

- **Awake()**
    - Ensures the singleton pattern.
- **RequestPath(PhysicalStatsLogic character, Vector3 destination, Action<bool> onPathCalculated = null)**
    - Accepts a pathfinding request from a specific agent, including his physical capabilities and desired destination. Takes an optional delegate function which will be executed upon finding a path to a point.
    - Forms a **NavRequest** and stores it in the queue.
    - Automatically begins processing by calling ProcessNextRequest() if no other requests are currently being handled.

- **ProcessNextRequest()**
    - Processes the next request in the queue, if any. Proceeds with synchronous processing by calling ProcessPathSync().
    - Automatically continues processing subsequent requests.

- **ProcessPathSync(NavRequest request)**
    - Performs synchronous pathfinding using the agent's physical statistics.
    - Iterates through all LinkData objects in the scene.
    - Temporarily disables links that exceed the agent's physical capabilities:
        - Length greater than maxJumpDistance
        - Height differences exceeding maxJumpHeight or maxDropDistance.

    - Uses Unity's NavMeshAgent.CalculatePath() to find a valid path to the destination.
    - If a path is found, it assigns the path to the agent and triggers the callback (onPathCalculated).
    - Re-enables previously disabled links after the calculation.

  Note: The current implementation disables invalid links for the time of path calculation, but an experimental design using weights is commented out for future use in asynchronous path calculations.

- **AutoAssignInstance()**
    - Ensures the Instance is assigned immediately. Intended for use in the editor to reinitialize the manager and enable smartLink functionalities (see Management and Utility Methods -> Links.

NavRequest Struct

The NavRequest struct encapsulates the data required for a pathfinding request.

Fields:

- PhysicalStatsLogic character: Reference to the agent's physical statistic class.
- Vector3 destination: The destination point for the path.
- Action<bool> onPathCalculated: Optional callback invoked upon path calculation. The parameter indicates whether a valid path was found.

Extending and Customizing

The systems included in this project have been built with future extensions in mind allowing for more complex use cases. Below are potential extensions and improvements to the existing system, as well as considerations for implementing them:

Edge Division and Connection Alignment Adjustments

Overview:

The maxEdgeLength parameter in NavLinkGenerator was initially designed to subdivide excessively long edges into smaller segments for increasing the density of the connections. This approach was discarded in the current implementation due to the already high number of links and other possibilities to increase them, like the NavMeshSurface TileSize parameter. However the goal with this extension might lie in more opportunities for link connection alignment.

Idea of Implementation

The system can incorporate a function to detect and align connection points based on angles between segments. The system could evaluate multiple potential connections for each divided edge segment and select the one with the straightest angle for smoother and more natural movement.

The groundwork for this approach exists in the ValidConnectionExists() method, which checks for valid connections between points on an edge. Currently, this method terminates after identifying any valid connection. Modifying this logic to evaluate all possible connections and select the optimal angle would refine connections without increasing their numbers unnecessarily.

Short Edge Grouping Enhancements

Overview:

Short edge grouping is currently managed by the GroupRepresentative() method, which merges edges by selecting a representative edge and slightly adjusting its falloff direction based on the average of the group. Another unused method, MergeEdgeGroup(), attempts to combine edges directly but results in shorter, unpredictably positioned edges that may not align well with the geometry.

Exploring additional approaches for edge grouping could improve this process, potentially by using thresholds or weights.

Supporting Multiple NavMeshes for Diverse Character Types

Overview

Unity's NavMesh system requires baking meshes to account for specific physical parameters, such as agent height and width. The current system is made to include a single NavMesh, which restricts the ability to support a wide range of character sizes and shapes. Adding capabilities to the system which incorporates multiple navigation mesh types and multiple in-built Unity Agent Types could greatly improve the flexibility.

Idea of Implementation

This functionality could be implemented with a layered approach where multiple NavMeshes are baked to accommodate different agent sizes and then stored in the list inside a centralized controller. Since Unity navigation mesh offset changes depending on the agent width, each NavMesh must include its own set of generated links optimized for that specific character size. The system could then run separate instances of the pathfinding system for each NavMesh to ensure accurate and efficient link traversal.

This implementation could unfortunately greatly increase complexity and memory usage, thus potentially a custom implementation of NavMeshSurface class could be made to incorporate these changes efficiently.

Asynchronous Pathfinding Request Processing

Overview:

While the system architecture includes a queue for processing pathfinding requests asynchronously, it currently calculates them sequentially on the main thread only. As tested, Unity's NavMeshSurface methods, in the version 2022.3.52f1 used for this project, do not support asynchronous execution. Attempting to process pathfinding requests this way would require significant changes to how NavMesh data is handled.
The current system could be combined with a custom implementation of NavMeshSurface which is better suited for multithreading.

With such system included, other considerations must be made regarding current implementation of pathfining. Particularly the method used for link filtering is based on link instances being deactivated. This could lead to a situation where links can not be accessed by other agents. A weights based method for this problem has already been explored, however it would require agent to use the validate method: PhysicalStatsLogic.QuickLinkValid(), which unfortunately comes with a downside of producing noticeable jitters and glitches as the execution may be too slow to stop the agent in time.