# A\* Pathfinding Algorithm Implementation in MIPS Assembly

## Overview

This document provides a comprehensive explanation of a MIPS assembly implementation of the A\* pathfinding algorithm. The implementation includes visualization components to display the algorithm’s progress in real-time, making it suitable for educational purposes and algorithm visualization.

A\* (pronounced “A-star”) is a popular pathfinding algorithm used in computer science and artificial intelligence. It combines the advantages of Dijkstra’s algorithm (which guarantees the shortest path) and greedy best-first search (which uses heuristics to improve performance). A\* is widely used in applications such as video games, robotics, and navigation systems.

## Data Structures

### Movement Directions

The program defines two sets of movement directions:

# Movement directions (4-way: up, right, down, left)  
d4x: .word 0, 1, 0, -1  
d4y: .word -1, 0, 1, 0  
  
# Or 8-way movement (includes diagonals)  
d8x: .word 0, 1, 1, 1, 0, -1, -1, -1  
d8y: .word -1, -1, 0, 1, 1, 1, 0, -1

* d4x and d4y: Define 4-way movement (cardinal directions)
* d8x and d8y: Define 8-way movement (including diagonals)

The program primarily uses 4-way movement for its pathfinding, though 8-way movement is defined and could be used with minor modifications.

### Node Structure

Each node in the grid is represented by a data structure with the following attributes:

.eqv x, 0 # x-coordinate (offset 0)  
.eqv y, 4 # y-coordinate (offset 4)  
.eqv wall, 8 # wall status (offset 8)  
.eqv gScore, 12 # cost from start (offset 12)  
.eqv hScore, 16 # heuristic cost to goal (offset 16)  
.eqv fScore, 20 # total cost (g + h) (offset 20)  
.eqv parent\_x, 24 # parent node x-coordinate (offset 24)  
.eqv parent\_y, 28 # parent node y-coordinate (offset 28)

This structure enables tracking of all necessary information for the A\* algorithm, including: - Position in the grid - Whether the position is a wall/obstacle - Path costs (g, h, and f scores) - Parent node for path reconstruction

## Main Components

### Main Function

The main function initializes the program by: 1. Setting up the stack pointer 2. Clearing the screen 3. Drawing the grid 4. Highlighting the start and goal positions 5. Adding a brief delay 6. Calling the A\* algorithm implementation

### A\* Algorithm Implementation

The A\* algorithm is implemented in the a\_star function, which follows these steps:

1. **Initialization**:
   * Calls initialize\_nodes to set up the node data structures
   * Adds the starting node to the open set with a g-score of 0
   * Calculates the h-score (Manhattan distance to the goal)
   * Calculates the f-score (g-score + h-score)
   * Pushes the start node to the open set (priority queue)
2. **Main Loop**:
   * Continues until the open set is empty or the goal is found
   * Pops the node with the lowest f-score from the open set
   * Checks if the popped node is the goal
   * If not, adds the node to the closed set and processes its neighbors
3. **Neighbor Processing**:
   * For each neighbor in the 4 cardinal directions:
     + Calculates its coordinates
     + Checks if it’s a valid position (within bounds and not an obstacle)
     + Checks if it’s already in the closed set
     + Calculates a tentative g-score
     + Updates the g-score if the tentative score is better
     + Calculates the h-score and f-score
     + Stores the node’s parent for path reconstruction
     + Pushes the neighbor to the open set
4. **Path Reconstruction**:
   * If a path is found, it traces back from the goal to the start using parent pointers
   * Visualizes the final path on the grid

## Helper Functions

### is\_valid\_position

Checks if a given position is valid by verifying: - The position is within the grid boundaries - The position is not an obstacle

### is\_in\_closed\_set

Checks if a node is in the closed set (already evaluated)

### add\_to\_closed\_set

Adds a node to the closed set

### manhattan\_heuristic (referenced but defined in an included file)

Calculates the Manhattan distance between two points, which serves as the heuristic for the A\* algorithm

### Visualization Functions

The code includes visualization functions to display the algorithm’s progress:

* drawGrid: Renders the initial grid
* drawGridNode: Highlights a specific node with a color
* clearScreen: Clears the display

These functions allow for real-time visualization of: - The explored nodes (yellow) - The frontier nodes (gray) - The final path (green) - The start and goal positions

## Implementation Details

### Priority Queue

The open set is implemented as a priority queue (heap) with operations: - push: Adds a node to the open set - pop: Removes and returns the node with the lowest f-score

These operations are defined in the included file node\_list.asm.

### Path Reconstruction

When the goal is found, the algorithm traces back from the goal to the start using the parent pointers stored in each node. This path is then visualized on the grid.

### Delays

The code includes strategic delays to make the visualization comprehensible:

li $a0, 100 # 100 milliseconds delay  
li $v0, 32 # syscall for sleep  
syscall

These delays slow down the algorithm to allow human observation of the search process.

## Performance Considerations

1. **Heuristic Choice**: The Manhattan distance heuristic is used, which is admissible for 4-way movement but can be less effective for 8-way movement.
2. **Open Set Implementation**: The open set is implemented as a priority queue, which allows for efficient selection of the lowest f-score node.
3. **Closed Set Implementation**: The closed set is implemented as a direct-access array, allowing for O(1) lookups.

## Visualization Color Scheme

The code uses a consistent color scheme for visualization: - Start position: Color code 8 - Goal position: Color code 2 (green) - Current node being explored: Color code 5 (yellow) - Nodes in the open set: Color code 9 (gray) - Final path: Color code 3 (green)

## Included Files

The implementation references several included files: - node\_list.asm: Contains priority queue operations - h\_calc.asm: Contains heuristic calculation functions - bitmap\_helper.asm: Contains visualization functions

## Conclusion

This MIPS assembly implementation of the A\* pathfinding algorithm demonstrates both the algorithm’s functionality and its visualization. The real-time visualization helps in understanding how A\* efficiently finds the shortest path by combining the best of Dijkstra’s algorithm and greedy best-first search.

The implementation is well-structured, with clear separation between the core algorithm, data structures, and visualization components. It could be extended to support 8-way movement, different heuristics, or more complex grid environments.