🌟 Hoshi: A* Pathfinding Algorithm in MIPS Assembly

📝 Overview

Hoshi (星, Japanese for "star") is a complete implementation of the A* pathfinding algorithm in MIPS assembly language. This project demonstrates low-level programming concepts by implementing a complex algorithmic solution in assembly language, offering valuable insights into computer architecture and optimization techniques.

🚀 Features

- ✅ **Complete A* algorithm implementation** in pure MIPS assembly
- ✅ **Priority Queue** with efficient insertion and extraction operations
- ✅ **Manhattan/Chebyshev Distance Heuristic** for optimal path calculation
- ✅ **Visual Representation** through bitmap display for step-by-step algorithm progression
- ✅ **Obstacle Detection** with robust path planning around barriers
- ✅ **Path Reconstruction** with visual highlighting of the optimal route

🧠 A* Algorithm: Core Concepts and Implementation

What is A*?

A* (pronounced "A-star") is an informed search algorithm widely used in pathfinding and graph traversal. It efficiently plots a traversable path between multiple nodes by maintaining a priority queue of paths and choosing the lowest-cost path to expand.

Algorithm Overview

The A* algorithm is built on three key components: 1. **g-score**: The actual cost from the start node to the current node 2. **h-score**: A heuristic estimate of the cost from the current node to the goal 3. **f-score**: The sum of g-score and h-score, representing the total estimated cost

The algorithm maintains two sets: - **Open Set**: Nodes to be evaluated (stored in a priority queue) - **Closed Set**: Nodes already evaluated

Algorithm Pseudocode

```
1. Initialize:
   • Open List: Contains nodes yet to be evaluated (start with start node)
   • Closed List: Stores nodes already evaluated (starts empty)
   • Set start.g = 0: Cost from start to start
   • Set start.h = heuristic(start, goal): Estimated cost from start to goal
   • Compute start.f = start.g + start.h
   • Set start.parent = null: No parent yet
```

```
2. Main loop: While open list is not empty:
   • Select current as the node in openList with the lowest f value
   • Goal check:
     ° If current == goal, return the reconstructed path
   • Move current node:
     ° Remove current from openList
     ° Add current to closedList
   • Process neighbors:
     ° For each neighbor of current:
       ▪ Skip if neighbor is in closedList
       ▪ Compute tentative_g = current.g + distance(current, neighbor)
       ▪ If neighbor is not in openList: Add it
       ▪ Else if tentative_g >= neighbor.g: Skip (existing path is better)
       ▪ Otherwise (this path is better):
         ▫ Update neighbor.parent = current
         ▫ Update neighbor.g = tentative_g
         ▫ Update neighbor.h = heuristic(neighbor, goal)
         ▫ Recompute neighbor.f = neighbor.g + neighbor.h

3. If loop ends with no path found:
   • Return failure: No path exists between start and goal

4. Path Reconstruction Function:
   • Start from goal node
   • Trace back using parent links, adding each node to a path
   • Return the path in reverse (from start to goal)
```

Core psudo Implementation

The A* algorithm is implemented in the a_star function, which follows the pseudocode above:

function A_Star(start, goal): // Initialize open and closed lists openList = [start] // Nodes to be evaluated closedList = [] // Nodes already evaluated

```
// Initialize node properties
start.g = 0                // Cost from start to start is 0
start.h = heuristic(start, goal)  // Estimate to goal
start.f = start.g + start.h       // Total estimated cost
start.parent = null              // For path reconstruction
while openList is not empty:
    // Get node with lowest f value - implement using a priority queue
   // for faster retrieval of the best node
    current = node in openList with lowest f value

    // Check if we've reached the goal
    if current = goal:
        return reconstruct_path(current)

    // Move current node from open to closed list
```

```
    remove current from openList
    add current to closedList

    // Check all neighboring nodes
    for each neighbor of current:
        if neighbor in closedList:
            continue  // Skip already evaluated nodes

        // Calculate tentative g score
        tentative_g = current.g + distance(current, neighbor)

        if neighbor not in openList:
            add neighbor to openList
        else if tentative_g >= neighbor.g:
            continue  // This path is not better

        // This path is the best so far
        neighbor.parent = current
        neighbor.g = tentative_g
        neighbor.h = heuristic(neighbor, goal)
        neighbor.f = neighbor.g + neighbor.h

return failure  // No path exists
```

function reconstruct_path(current): path = [] while current is not null: add current to beginning of path current = current.parent return path

📊 Data Structures and Modules

The implementation is organized into several modules, each handling specific aspects of the algorithm:

1. Priority Queue

The priority queue is implemented as a binary min-heap, which ensures efficient extraction of the node with the lowest f-score.

Node Structure in the Priority Queue

```
Offset  Field     Size (bytes)
0       x         4
4       y         4
8       parent    4
12      fScore    4
```

Key Operations:

- **push**: Inserts a node with O(log n) complexity
- **pop**: Extracts the node with lowest f-score with O(log n) complexity

2. Node List

Each node in the grid has specific properties that track its state in the A* algorithm.

Node Structure

```
Offset  Field     Size (bytes)
0       x         4
4       y         4
8       wall      4
12      gScore    4
16      hScore    4
20      fScore    4
24      parent_x  4
28      parent_y  4
```

Key Operations:

- **initialize_nodes**: Sets up the grid based on map data
- **set_g_score/get_g_score**: Manages cost from start
- **set_f_score/get_f_score**: Manages total estimated cost

3. Bitmap Display

The bitmap module manages visualization, providing a graphical representation of the A* algorithm's execution.

Display Constants

```
.eqv    displayWidth, 16       # Width of the display in units
.eqv    displayHeight, 16      # Height of the display in units
.eqv    gridCellWidth, 2       # Width of each grid cell
.eqv    gridCellHeight, 2      # Height of each grid cell
.eqv    gridWidth, 8           # Width of the grid in cells
.eqv    gridHeight, 8          # Height of the grid in cells
.eqv    bitmapBaseAddress, 0x10040000  # Memory address of bitmap
```

Key Operations:

- **clearScreen**: Initializes the display
- **drawGridNode**: Renders a single node with specific color based on its state
- **drawGrid**: Renders the entire grid

4. Heuristic Functions

The A* algorithm uses heuristic functions to estimate the cost from any node to the goal.

Available Heuristics:

- **Manhattan Distance**: Sum of horizontal and vertical distances
- **Chebyshev Distance**: Maximum of horizontal and vertical distances

```
manhattanDistance:
    # Calculate |x1-x2| + |y1-y2|
    sub     $t0, $a0, $a2       # x1-x2
    abs     $t0, $t0            # |x1-x2|

    sub     $t1, $a1, $a3       # y1-y2
    abs     $t1, $t1            # |y1-y2|

    add     $v0, $t0, $t1       # |x1-x2| + |y1-y2|
    jr      $ra

chebyshevDistance:
    # Calculate max(|x1-x2|, |y1-y2|)
    sub     $t0, $a0, $a2       # x1-x2
    abs     $t0, $t0            # |x1-x2|

    sub     $t1, $a1, $a3       # y1-y2
    abs     $t1, $t1            # |y1-y2|

    # Find maximum
    bge     $t0, $t1, max_is_x
    move    $v0, $t1
    j       chebyshev_return

max_is_x:
    move    $v0, $t0

chebyshev_return:
    jr      $ra
```

5. Path Reconstruction

Once the A* algorithm finds a path, it traces back from the goal to the start using parent pointers.

🔍 Implementation Details

Memory Management

The implementation uses a consistent pattern to locate nodes in memory:

```
# Calculate node address from (x,y) coordinates
lw      $t0, map_width          # Load grid width
mul     $t1, $a1, $t0           # t1 = y * width
add     $t1, $t1, $a0           # t1 = y * width + x
lw      $t2, node_size          # Load node size in bytes
mul     $t3, $t1, $t2           # t3 = index * node_size
la      $t4, nodes              # Load base address
add     $t4, $t4, $t3           # t4 = base + offset
```

This efficiently implements the formula: &nodes[y * width + x] to convert 2D coordinates to memory addresses.

Register Usage Strategy

The implementation follows a consistent register allocation strategy: - $s0-$s7: Preserved across function calls, used for loop variables and important data - $t0-$t9: Temporary calculations, not preserved across calls - $a0-$a3: Function arguments - $v0-$v1: Function return values - $ra: Return address register, preserved on stack when making nested calls

Stack Management

Proper stack management is critical for function calls and recursion:

```
# Function prologue
addi    $sp, $sp, -4       # Allocate stack space
sw      $ra, 0($sp)        # Save return address

# Function body
# ...

# Function epilogue
lw      $ra, 0($sp)        # Restore return address
addi    $sp, $sp, 4        # Deallocate stack space
jr      $ra                # Return
```

🎨 Visualization

The visualization uses a consistent color scheme: - **Color 0** (White): Background/free space - **Color 1** (Black): Walls/obstacles - **Color 2** (Green): Goal node - **Color 3** (Green): Final path - **Color 5** (Yellow): Current node being explored - **Color 8** (Cyan): Start node - **Color 9** (Gray): Nodes in the open set

This color coding makes it easy to understand the algorithm's progress visually.

🔧 Optimization Techniques

1. **Priority Queue**: O(log n) operations for managing the open set
2. **Register Usage**: Strategic register allocation minimizes memory access
3. **Closed Set**: Efficient tracking of evaluated nodes
4. **Memory Access**: Calculates addresses efficiently to minimize overhead
5. **Data Organization**: Node structures organized for efficient access patterns

📊 Performance Analysis

Operation	Time Complexity
Node Extraction	O(log n)

Operation	Time Complexity
Node Insertion	O(log n)
Path Reconstruction	O(p) where p is path length
Heuristic Calculation	O(1)
Overall Algorithm	O(E log V) where V is number of nodes and E is number of edges

Movement Directions

The implementation supports both 4-way and 8-way movement:

```
# 4-way movement (up, right, down, left)
d4x:                .word       0, 1, 0, -1
d4y:                .word       -1, 0, 1, 0

# 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
```

The default implementation uses 4-way movement for simplicity and clarity.

Educational Value

This implementation offers several educational insights: 1. **Low-level Programming**: Direct memory management and register allocation 2. **Algorithm Implementation**: From theory to practical assembly code 3. **Data Structures**: Priority queue and grid management 4. **Visualization**: Real-time algorithm execution display 5. **Optimization**: Balancing readability with performance