# 🌟 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 Implementation in MIPS Assembly

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

a\_star:  
 # Function setup  
 addi $sp, $sp, -4  
 sw $ra, 0($sp)  
   
 # Initialize nodes and data structures  
 jal initialize\_nodes  
   
 # Setup start node  
 lw $s0, start\_x  
 lw $s1, start\_y  
   
 # Set g-score of start node to 0  
 move $a0, $s0  
 move $a1, $s1  
 li $a2, 0  
 jal set\_g\_score  
   
 # Calculate h-score using heuristic  
 lw $a2, goal\_x  
 lw $a3, goal\_y  
 jal chebyshevDistance # or manhattanDistance  
   
 # Calculate f-score (g + h)  
 move $a2, $v0  
 move $a0, $s0  
 move $a1, $s1  
 jal set\_f\_score  
   
 # Push start node to open set  
 move $a0, $s0  
 move $a1, $s1  
 li $a2, 0 # parent (none for start)  
 move $a3, $v0 # f-score  
 jal push  
   
 # Main loop  
a\_star\_loop:  
 # Check if open set is empty  
 la $t0, heapSize  
 lw $t0, 0($t0)  
 beqz $t0, a\_star\_no\_path  
   
 # Pop node with lowest f-score  
 jal pop  
   
 # Check if popped node is goal  
 la $t0, extracted\_node  
 lw $t1, 0($t0) # x  
 lw $t2, 4($t0) # y  
 lw $t3, goal\_x  
 lw $t4, goal\_y  
   
 beq $t1, $t3, goal\_check\_y  
 j not\_goal  
   
goal\_check\_y:  
 beq $t2, $t4, a\_star\_found\_path  
   
not\_goal:  
 # Add current node to closed set  
 move $a0, $t1  
 move $a1, $t2  
 jal add\_to\_closed\_set  
   
 # Process neighbors (4 directions)  
 li $s0, 0 # direction counter  
   
process\_neighbors:  
 li $t0, 4  
 beq $s0, $t0, a\_star\_loop # If all neighbors processed  
   
 # Calculate neighbor coordinates  
 la $t0, d4x  
 la $t1, d4y  
 sll $t2, $s0, 2  
 add $t0, $t0, $t2  
 add $t1, $t1, $t2  
 lw $t2, 0($t0) # dx  
 lw $t3, 0($t1) # dy  
   
 la $t0, extracted\_node  
 lw $t4, 0($t0) # current.x  
 lw $t5, 4($t0) # current.y  
   
 add $t6, $t4, $t2 # neighbor.x  
 add $t7, $t5, $t3 # neighbor.y  
   
 # Check if valid position  
 move $a0, $t6  
 move $a1, $t7  
 jal is\_valid\_position  
 beqz $v0, next\_neighbor  
   
 # Check if in closed set  
 move $a0, $t6  
 move $a1, $t7  
 jal is\_in\_closed\_set  
 bnez $v0, next\_neighbor  
   
 # Calculate tentative g-score  
 move $a0, $t4  
 move $a1, $t5  
 jal get\_g\_score  
   
 addi $t0, $v0, 1 # tentative\_g = current.g + 1  
   
 # Get neighbor's current g-score  
 move $a0, $t6  
 move $a1, $t7  
 jal get\_g\_score  
   
 # Compare tentative\_g with neighbor.g  
 bge $t0, $v0, next\_neighbor # Skip if not better  
   
 # Update neighbor's scores  
 move $a0, $t6  
 move $a1, $t7  
 move $a2, $t0 # new g-score  
 jal set\_g\_score  
   
 # Calculate h-score  
 move $a0, $t6  
 move $a1, $t7  
 lw $a2, goal\_x  
 lw $a3, goal\_y  
 jal chebyshevDistance  
   
 # Calculate new f-score  
 add $a2, $t0, $v0 # g + h  
 move $a0, $t6  
 move $a1, $t7  
 jal set\_f\_score  
   
 # Set parent  
 move $a0, $t6  
 move $a1, $t7  
 move $a2, $t4 # parent.x  
 move $a3, $t5 # parent.y  
 jal set\_parent  
   
 # Push neighbor to open set  
 move $a0, $t6  
 move $a1, $t7  
 move $a2, $a2 # parent reference  
 move $a3, $a2 # f-score  
 jal push  
   
next\_neighbor:  
 addi $s0, $s0, 1  
 j process\_neighbors  
   
a\_star\_found\_path:  
 # Print success message  
 la $a0, path\_found\_msg  
 li $v0, 4  
 syscall  
   
 # Reconstruct path  
 lw $a0, goal\_x  
 lw $a1, goal\_y  
 jal constructPathProcedure  
   
 j a\_star\_exit  
   
a\_star\_no\_path:  
 # Print failure message  
 la $a0, no\_path\_msg  
 li $v0, 4  
 syscall  
   
a\_star\_exit:  
 lw $ra, 0($sp)  
 addi $sp, $sp, 4  
 jr $ra

## 📊 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

push:  
 # Check if the heap is full  
 la $t0, heapSize  
 lw $t1, 0($t0)  
 la $t2, maxHeapSize  
 lw $t3, 0($t2)  
 beq $t1, $t3, heap\_full  
   
 # Store the new node at the end of the heap  
 la $t4, heap  
 mul $t5, $t1, 16 # Each node is 16 bytes  
 add $t6, $t4, $t5 # Address of new node  
   
 # Store node data  
 sw $a0, 0($t6) # x  
 sw $a1, 4($t6) # y  
 sw $a2, 8($t6) # parent  
 sw $a3, 12($t6) # fScore  
   
 # Increment heap size  
 addi $t1, $t1, 1  
 sw $t1, 0($t0)  
   
 # Bubble up to maintain heap property  
 addi $a0, $t1, -1 # Index of the new node  
 jal bubble\_up  
   
 jr $ra  
   
pop:  
 # Check if the heap is empty  
 la $t0, heapSize  
 lw $t1, 0($t0)  
 beqz $t1, heap\_empty  
   
 # Extract the root node  
 la $t2, heap # Heap base address  
 la $t3, extracted\_node # Where to store extracted node  
   
 # Copy root node to extracted\_node  
 lw $t4, 0($t2)  
 lw $t5, 4($t2)  
 lw $t6, 8($t2)  
 lw $t7, 12($t2)  
 sw $t4, 0($t3) # x  
 sw $t5, 4($t3) # y  
 sw $t6, 8($t3) # parent  
 sw $t7, 12($t3) # fScore  
   
 # Move the last element to the root  
 addi $t1, $t1, -1 # Decrement heap size  
 sw $t1, 0($t0)  
   
 beqz $t1, pop\_done # If heap is now empty, we're done  
   
 # Calculate address of last node  
 mul $t4, $t1, 16 # Each node is 16 bytes  
 add $t5, $t2, $t4 # Address of last node  
   
 # Copy last node to root  
 lw $t6, 0($t5)  
 lw $t7, 4($t5)  
 lw $t8, 8($t5)  
 lw $t9, 12($t5)  
 sw $t6, 0($t2) # x  
 sw $t7, 4($t2) # y  
 sw $t8, 8($t2) # parent  
 sw $t9, 12($t2) # fScore  
   
 # Bubble down to maintain heap property  
 li $a0, 0 # Start at root  
 jal bubble\_down  
   
pop\_done:  
 jr $ra

### 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

initialize\_nodes:  
 # Function setup  
 addi $sp, $sp, -4  
 sw $ra, 0($sp)  
   
 # Initialize screen and variables  
 jal clearScreen  
 la $s0, nodes # Base node address  
 la $s1, map\_data # Map data pointer  
 lw $s2, map\_width # Grid dimensions  
 lw $s3, map\_height  
   
 # Process each cell in grid  
 li $s4, 0 # y counter  
row\_loop:  
 beq $s4, $s3, \_done  
 li $s5, 0 # x counter  
   
col\_loop:  
 beq $s5, $s2, next\_row  
   
 # Calculate memory addresses  
 mul $t0, $s4, $s2  
 add $t0, $t0, $s5  
 sll $t0, $t0, 2  
 add $t1, $s1, $t0 # map\_data address  
   
 mul $t0, $s4, $s2  
 add $t0, $t0, $s5  
 lw $t2, node\_size  
 mul $t0, $t0, $t2  
 add $t2, $s0, $t0 # node address  
   
 # Initialize node properties  
 sw $s5, x($t2) # Store x coordinate  
 sw $s4, y($t2) # Store y coordinate  
 lw $t5, 0($t1) # Get map value  
 sw $t5, wall($t2) # Store wall status  
   
 # Set default A\* values  
 li $t6, 999  
 sw $t6, gScore($t2) # "Infinity" g-score  
 li $t6, 0  
 sw $t6, hScore($t2) # Init h-score  
 add $t6, $t6, $t6  
 sw $t6, fScore($t2) # Init f-score  
 li $t6, 0  
 sw $t6, parent\_x($t2) # Init parent  
 sw $t6, parent\_y($t2)  
   
 # Visualize the node  
 move $a0, $s5  
 move $a1, $s4  
 move $a2, $t5  
 jal drawGridNode  
   
 # Continue loop  
 addi $s5, $s5, 1  
 lw $t7, nodes\_count  
 addi $t7, $t7, 1  
 sw $t7, nodes\_count  
 j col\_loop  
   
next\_row:  
 addi $s4, $s4, 1  
 j row\_loop  
   
\_done:  
 lw $ra, 0($sp)  
 addi $sp, $sp, 4  
 jr $ra

### 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

drawGridNode:  
 # Save return address  
 addi $sp, $sp, -4  
 sw $ra, 0($sp)  
   
 # Calculate display coordinates  
 li $t5, gridCellWidth  
 mul $t5, $a0, $t5 # baseX = x \* gridCellWidth  
 move $s7, $t5  
   
 li $t6, gridCellHeight  
 mul $t6, $a1, $t6 # baseY = y \* gridCellHeight  
   
 # Calculate cell boundaries  
 addi $t7, $t5, gridCellWidth  
 addi $t8, $t6, gridCellHeight  
   
 # Draw the cell pixel by pixel  
row\_loop\_bitmap:  
 bge $t6, $t8, finish  
 move $t5, $s7  
   
col\_loop\_bitmap:  
 bge $t5, $t7, next\_row\_bitmap  
 move $a0, $t5  
 move $a1, $t6  
 jal drawPixel  
 addi $t5, $t5, 1  
 j col\_loop\_bitmap  
   
next\_row\_bitmap:  
 addi $t6, $t6, 1  
 j row\_loop\_bitmap  
   
finish:  
 lw $ra, 0($sp)  
 addi $sp, $sp, 4  
 jr $ra

### 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.

constructPathProcedure:  
 # Function setup  
 addi $sp, $sp, -4  
 sw $ra, 0($sp)  
   
 # Check if we've reached the start  
 lw $s0, start\_x  
 lw $s1, start\_y  
 bne $a0, $s0, constructPath\_next\_parent  
 bne $a1, $s1, constructPath\_next\_parent  
   
 # If at start node, return  
 lw $ra, 0($sp)  
 addi $sp, $sp, 4  
 jr $ra  
   
constructPath\_next\_parent:  
 # Save current coordinates  
 move $s6, $a0  
 move $s7, $a1  
   
 # Calculate node address  
 lw $t0, map\_width  
 mul $t1, $a1, $t0  
 add $t1, $t1, $a0  
 lw $t2, node\_size  
 mul $t1, $t1, $t2  
 la $t0, nodes  
 add $t0, $t0, $t1  
   
 # Get parent coordinates  
 lw $t3, parent\_x($t0)  
 lw $t4, parent\_y($t0)  
   
 # Recursive call to process parent first  
 move $a0, $t3  
 move $a1, $t4  
   
 # Save current node on stack  
 addi $sp, $sp, -8  
 sw $s6, 0($sp)  
 sw $s7, 4($sp)  
   
 # Process parent  
 jal constructPathProcedure  
   
 # Restore coordinates  
 lw $s6, 0($sp)  
 lw $s7, 4($sp)  
 addi $sp, $sp, 8  
   
 # Visualize this node as part of the path  
 move $a0, $s6  
 move $a1, $s7  
 li $a2, 5 # Path color  
 jal drawGridNode  
   
 # Delay for visualization  
 li $v0, 32  
 li $a0, 100  
 syscall  
   
 # Return  
 lw $ra, 0($sp)  
 addi $sp, $sp, 4  
 jr $ra

## 🔍 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) |
| 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