# Heuristic Functions Documentation

This document provides a detailed explanation of the heuristic functions used in a pathfinding algorithm implementation. These functions calculate the estimated distance between the current position and the goal position, which is essential for algorithms like A\* to efficiently find the shortest path.

## Table of Contents

1. [Overview](#overview)
2. [Manhattan Heuristic](#manhattan-heuristic)
   * [Function Description](#function-description-manhattan)
   * [Algorithm](#algorithm-manhattan)
   * [Code Explanation](#code-explanation-manhattan)
   * [Register Usage](#register-usage-manhattan)
   * [Edge Cases](#edge-cases-manhattan)
3. [Euclidean Heuristic](#euclidean-heuristic)
   * [Function Description](#function-description-euclidean)
   * [Algorithm](#algorithm-euclidean)
   * [Code Explanation](#code-explanation-euclidean)
   * [Register Usage](#register-usage-euclidean)
   * [Floating-Point Operations](#floating-point-operations)
4. [Usage Guidelines](#usage-guidelines)
5. [Performance Considerations](#performance-considerations)

## Overview

Heuristic functions are used in informed search algorithms to estimate the cost from the current state to the goal. A good heuristic never overestimates the actual cost, making it admissible for algorithms like A\*. This implementation provides two common heuristic functions used in grid-based pathfinding:

1. **Manhattan Distance**: The sum of horizontal and vertical distances, ideal for 4-direction movement.
2. **Euclidean Distance**: The straight-line distance, ideal for 8-direction movement.

## Manhattan Heuristic

### Function Description

The Manhattan distance, also known as taxicab distance or L1 norm, calculates the sum of absolute differences between coordinates. It represents the distance a taxicab would drive in a grid-like street layout where diagonal movement is not allowed.

**Formal Definition**: For two points with coordinates (x₁, y₁) and (x₂, y₂), the Manhattan distance is |x₁ - x₂| + |y₁ - y₂|.

### Algorithm

1. Calculate the horizontal distance: |current.x - goal.x|
2. Calculate the vertical distance: |current.y - goal.y|
3. Return the sum of these distances

### Code Explanation

manhattan\_heuristic:  
 # Manhattan distance (|x1 - x2| + |y1 - y2|)  
 # Inputs:  
 # $a0: current.x  
 # $a1: current.y  
 # $a2: goal.x  
 # $a3: goal.y  
 # Output:  
 # $v0: Manhattan distance  
 sub $t4, $a0, $a2 # t4 = current.x - goal.x  
 bltz $t4, neg\_x # If t4 < 0, branch to neg\_x  
x\_done:  
 sub $t5, $a1, $a3 # t5 = current.y - goal.y  
 bltz $t5, neg\_y # If t5 < 0, branch to neg\_y  
y\_done:  
 add $v0, $t4, $t5 # v0 = |current.x - goal.x| + |current.y - goal.y|  
   
 jr $ra # Return to caller  
neg\_x:  
 sub $t4, $zero, $t4 # t4 = zero - t4 (absolute value)  
 j x\_done # Continue with y calculation  
neg\_y:  
 sub $t5, $zero, $t5 # t5 = zero - t5 (absolute value)  
 j y\_done # Continue with result calculation

The function calculates the Manhattan distance between two points:

1. First, it computes the horizontal distance by subtracting goal.x from current.x.
2. If this difference is negative, it branches to neg\_x to calculate the absolute value by negating the number.
3. Next, it computes the vertical distance by subtracting goal.y from current.y.
4. Again, if this difference is negative, it branches to neg\_y to calculate the absolute value.
5. Finally, it adds the absolute horizontal and vertical distances together to get the Manhattan distance.
6. The result is returned in register $v0.

### Register Usage

* **Input Registers**:
  + $a0: x-coordinate of current position
  + $a1: y-coordinate of current position
  + $a2: x-coordinate of goal position
  + $a3: y-coordinate of goal position
* **Temporary Registers**:
  + $t4: Holds |current.x - goal.x| (absolute horizontal distance)
  + $t5: Holds |current.y - goal.y| (absolute vertical distance)
* **Output Register**:
  + $v0: Returns the final Manhattan distance

### Edge Cases

* The function correctly handles negative differences through the neg\_x and neg\_y branches, ensuring proper absolute value calculation.
* The function works correctly with zero distances (when current and goal coordinates are the same).

## Euclidean Heuristic

### Function Description

The Euclidean distance calculates the straight-line distance between two points. This is the most intuitive measure of distance and is particularly useful for algorithms that allow diagonal movement.

**Formal Definition**: For two points with coordinates (x₁, y₁) and (x₂, y₂), the Euclidean distance is √((x₁ - x₂)² + (y₁ - y₂)²).

### Algorithm

1. Calculate the differences: (current.x - goal.x) and (current.y - goal.y)
2. Convert these differences to floating-point numbers
3. Square each difference
4. Add the squared differences
5. Calculate the square root of the sum
6. Convert the result back to an integer (rounded to nearest)
7. Return the integer result

### Code Explanation

euclidean\_heuristic:  
 # Inputs:  
 # $a0: current.x  
 # $a1: current.y  
 # $a2: goal.x  
 # $a3: goal.y  
 # Output:  
 # $v0: Euclidean distance (rounded to nearest integer)  
 # Calculate (x1 - x2)  
 sub $t4, $a0, $a2  
   
 # Calculate (y1 - y2)  
 sub $t5, $a1, $a3  
   
 # Convert (x1-x2) to float  
 mtc1 $t4, $f1  
 cvt.s.w $f1, $f1  
   
 # Convert (y1-y2) to float  
 mtc1 $t5, $f2  
 cvt.s.w $f2, $f2  
   
 # Square both  
 mul.s $f3, $f1, $f1 # (x1-x2)^2  
 mul.s $f4, $f2, $f2 # (y1-y2)^2  
   
 # Add  
 add.s $f5, $f3, $f4 # (x1-x2)^2 + (y1-y2)^2  
   
 # Take square root  
 sqrt.s $f0, $f5 # Result in $f0  
   
 # Convert float result to integer (round to nearest)  
 cvt.w.s $f0, $f0 # Convert to word format  
 mfc1 $v0, $f0 # Move to integer register for return  
   
 # Restore return address and return  
 jr $ra # Return to caller

The function calculates the Euclidean distance between two points:

1. It calculates the differences in both x and y coordinates between current and goal positions.
2. It converts these integer differences to floating-point format for precise calculation.
3. It squares both differences using the floating-point multiplication operation.
4. It adds the squared differences together.
5. It takes the square root of the sum using the sqrt.s instruction.
6. It converts the floating-point result back to an integer, rounding to the nearest whole number.
7. The final integer result is placed in register $v0 for return.

### Register Usage

* **Input Registers**:
  + $a0: x-coordinate of current position
  + $a1: y-coordinate of current position
  + $a2: x-coordinate of goal position
  + $a3: y-coordinate of goal position
* **Temporary Integer Registers**:
  + $t4: Holds (current.x - goal.x)
  + $t5: Holds (current.y - goal.y)
* **Floating-Point Registers**:
  + $f1: Floating-point version of (current.x - goal.x)
  + $f2: Floating-point version of (current.y - goal.y)
  + $f3: Holds (current.x - goal.x)²
  + $f4: Holds (current.y - goal.y)²
  + $f5: Holds (current.x - goal.x)² + (current.y - goal.y)²
  + $f0: Holds the square root result (final distance)
* **Output Register**:
  + $v0: Returns the final Euclidean distance as an integer

### Floating-Point Operations

The Euclidean heuristic function uses the MIPS Floating-Point Unit (FPU) for accurate distance calculation:

* mtc1: Moves an integer value to a floating-point register
* cvt.s.w: Converts a word (32-bit integer) to single-precision floating-point
* mul.s: Performs floating-point multiplication
* add.s: Performs floating-point addition
* sqrt.s: Calculates the square root of a floating-point value
* cvt.w.s: Converts a single-precision floating-point to word (integer)
* mfc1: Moves a value from a floating-point register to an integer register

## Usage Guidelines

### When to Use Each Heuristic

* **Manhattan Distance**: Best for grid-based pathfinding where movement is restricted to horizontal and vertical directions only (4-directional movement).
* **Euclidean Distance**: Best for applications where movement can occur in any direction, including diagonals (8-directional movement).

### Choosing the Right Heuristic

The choice between Manhattan and Euclidean heuristics depends on the allowed movement in your application:

* If diagonal movement is not allowed or costs more than orthogonal movement, use Manhattan distance.
* If diagonal movement is allowed and costs the same as moving horizontally or vertically the same total distance, use Euclidean distance.

## Performance Considerations

* **Manhattan Heuristic**: More computationally efficient as it only uses integer operations.
* **Euclidean Heuristic**: Less efficient due to floating-point operations and the square root calculation, but provides more accurate distance estimates when diagonal movement is allowed.

For performance-critical applications, consider: 1. Using Manhattan distance when possible, as it’s faster to compute. 2. Pre-computing distances for common scenarios. 3. Using lookup tables for square roots if the range of possible distances is limited.