AVL Data Structure for Robot Path Navigation

What is AVL Tree

In the simplest form, AVL tree is a type of binary search tree where the height between parent and child cannot exceed one. Topologically, it has a repeating tri-nodal structure. This height-balance property provides the uniqueness of AVL tree. This tree was named after its inventor Adelson-Velskii and Landis.

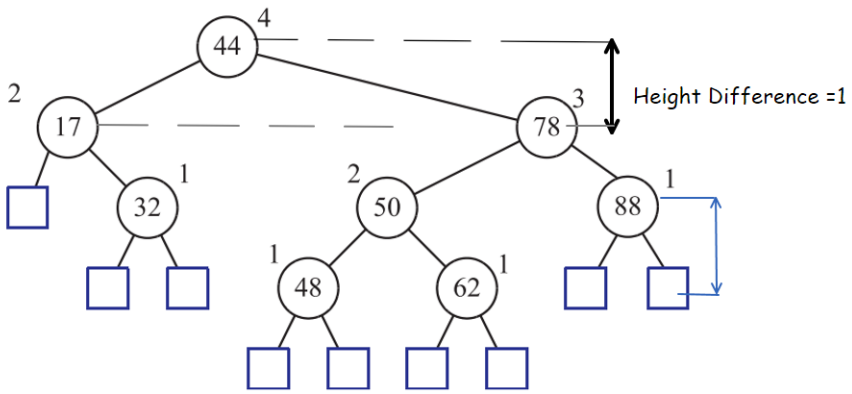


Fig1. Example of an AVL tree.

- An AVL tree is a Binary tree
- It is always balanced.
- Height difference between parent and child should always be 1.

AVL Tree Rotation

To preserve the AVL identity, an AVL tree must conserve two properties

- Binary tree property
- Balance

The main functionality of AVL tree comes into play, when it organizes data while maintaining these properties and this is done through restructuration or rotation. In the process of rotation, an AVL tree changes its structure by transforming parent nodes into child node and vice versa. This self-balancing and restoration of pattern makes AVL an excellent search algorithm for decision making.

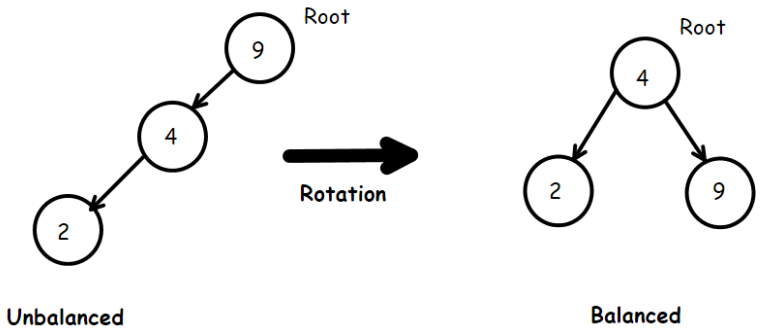


Fig2. Change of structure through rotation.

Application in Robotics

For the sake of simplicity, let's imagine a basic workspace surrounded by wall (Think of an empty room). The robot is a square shaped rigid body, aided with 4 infrared sensors. This robot is capable of performing five actions which are, moving east, moving west, moving north, moving south and stop. The sensors are triggered whenever it faces an obstacle and by synchronizing the sensors to moving actions, it is possible to make this robot autonomous. To this point it is clear that, this synchronization is a decision-making process where AVL tree could be implemented.

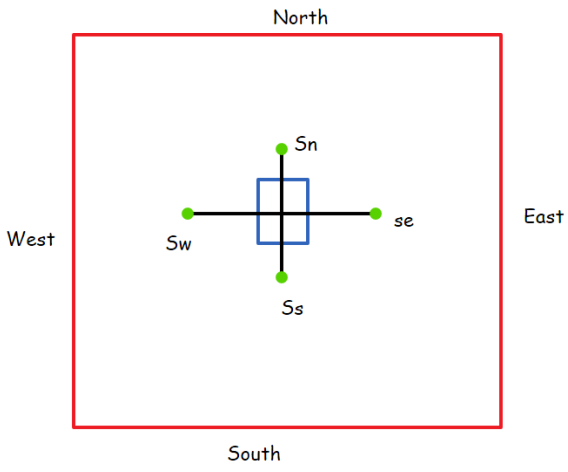


Fig3. A basic robot (blue square) with four sensors (sn, se, ss, sw). Red box is the boundary (wall).

Algorithm Logic

As it was discussed before, an AVL tree must be a balanced binary tree and the algorithm will use that characteristic for decision making. The control architecture of the robot will use two threads where the main thread will run the driver function and a secondary thread will collect sensor output data.

As it's a basic 4-sensor based robot on a 2D plane, using three node AVL tree is sufficient. Each node of the tree has two data field which are *label* and *val*. The *label* keeps track of the sensor located on a specific direction and *val* keeps track of sensor value. When the path is free of obstacle, sensor value will be 0 and 1 otherwise. The data structure has two methods and these are *AVLTreeAdd() and AVLTreeRotate()*. The Add method will receive sensor values as input, will look for first 0 (direction with no obstacle) and return the sensor location and value as a new node. The Rotate method will rotate root and LeftChild node in clockwise direction and then add a new LeftChild using Add method. The program execution steps are given below

- Initialize <root node> with the sensor located in moving direction.
- Keep moving in that direction as long as <root node> == 0
- When <root Node>==1 (there is an obstacle)
    - Call *Robot.Stop()*
    - Call *AVLTree.Rotate()*
    - Initialize New root node with new sensor direction

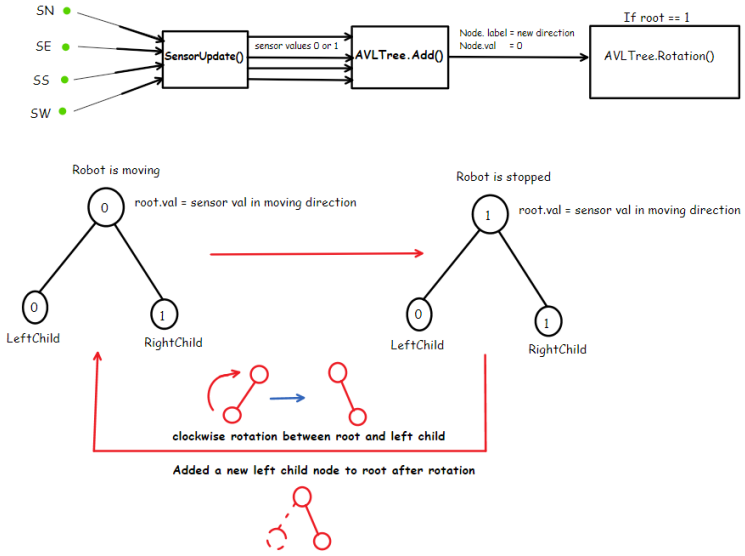
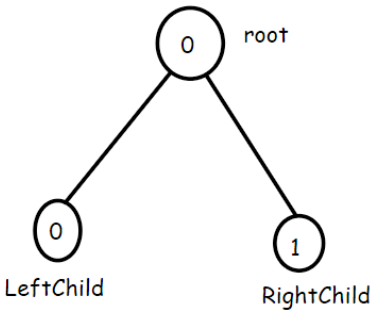


Fig4. Control Architecture Diagram.

PseudoCode

AVL Data Structure with three nodes



0 = false
1 = true

Robot action:

```
Move.East()
Move.North()
Move.West()
Move.South()
Stop()
```

For sensors are located in four direction

This block running in another thread

```
SensorUpdate()   // update sensor data
{

    update(sn)
    update(se)
    update(ss)
    update(sw)

}
```

Data structure Implementation pseudocode:

```
Node
{
    //data
    label // which sensor sn/se/ss/sw. each in specific direction
    val   //  truth value 0/1


}


//----Methods

AVLTree.Add(sn,se,ss,sw)
{

    case1 (sn==0)

        Node.label = sn
        Node.val   = 0

        return Node

    case2 (se==0)

        Node.label = sn
        Node.val   = 0

        return Node

    case3 (ss==0)

        Node.label = sn
        Node.val   = 0

        return Node

    case4 (sw==0)

        Node.label = sn
        Node.val   = 0

        return Node

}


AVLTree.Rotate()
{
    root.rightchild  = root
    root = root.left
    root.left = AVLTree.Add(sn,se,ss,sw)
}
```

```
// Driver function

Main
{

    SensorUpdate();        // update sensors. running in a different thread

    while (true)            // start main process
    {
        Move.root.label()   // actuate based on the root label sn/se/ss/sw

        if(root.val==1)
        {
            stop();  // stop moving

            AVLTree.Rotate(); // restructure the DataStructure

        }



    }



}
```

Why AVL Tree

In general, an autonomous robot requires myriad number of sensors to collect different state variables. Think of an Unmanned Ariel Vehicle which has six degrees of freedom along with other state variables such as air pressure, temperature and etc. To perfectly navigate, it must make decisions based on these multiple state variables which are constantly updating in real time. Due to the self-balancing property of AVL tree, the search time is very fast O(log(n)) and it is quite useful for fast moving autonomous devices.

Reference

[1] Michael T. Goodrich, Roberto Tamassia, David M. Mount, “Data Structures and Algorithms in C++, 2nd Edition”

[2] M. A. Otaduy, O. Chassot, D. Steinemann and M. Gross, "Balanced Hierarchies for Collision Detection between Fracturing Objects," *2007 IEEE Virtual Reality Conference*, 2007, pp. 83-90, doi: 10.1109/VR.2007.352467.

[3] Francesco Bullo, Stephen L. Smith “Lectures on Robotic Planning and Kinematics”