Test Cases

From the document titled **Intra-community Real-time Marketplace** (see `MCP Calculation Procedure.pdf`).

Copy and paste https://play.golang.org/ to test.

Horizontal Supply Intersection

```
package main

import (
    "fmt"
    "sort"
)

func main() {

    type Bid struct {
        Price       int64 `json:"price"`
        Quantity    int64 `json:"quantity"`
        AgentID     string `json:"agent_id"`
        AgentType   string `json:"agent_type"`
        Unit        string `json:"unit"`
        TimeStamp   int64 `json:"time_stamp"`
    }

    type Bids struct {
        Supply []Bid
        Demand []Bid
        }

    bids := Bids{}
    bid := Bid{}

    bid.Price = 150
    bid.Quantity = 400
    bid.AgentID = "KK"
    bid.AgentType = "HVAC"
    bid.Unit = "Wh"

    bids.Demand = append(bids.Demand, bid)

    bid.Price = 100
    bid.Quantity = 600
    bid.AgentID = "KT"
    bid.AgentType = "EV"
    bid.Unit = "Wh"
```

```
    bids.Demand = append(bids.Demand, bid)

    bid.Price = 40
    bid.Quantity = 500
    bid.AgentID = "RT"
    bid.AgentType = "BESS"
    bid.Unit = "Wh"

    bids.Demand = append(bids.Demand, bid)


    bid.Price = 50
    bid.Quantity = -600
    bid.AgentID = "TY"
    bid.AgentType = "PV"
    bid.Unit = "Wh"

    bids.Supply = append(bids.Supply, bid)

    bid.Price = 70
    bid.Quantity = -600
    bid.AgentID = "YY"
    bid.AgentType = "PV"
    bid.Unit = "Wh"

    bids.Supply = append(bids.Supply, bid)

    bid.Price = 120
    bid.Quantity = -500
    bid.AgentID = "YK"
    bid.AgentType = "PV"
    bid.Unit = "Wh"

    bids.Supply = append(bids.Supply, bid)


    fmt.Println("Demand:", bids.Demand)

    fmt.Println("Supply:", bids.Supply)

    // sort supply bids increasing
    sort.Slice(bids.Supply, func(i, j int) bool { return bids.Supply[i].Price < bids
    // sort demand bids decreasing
    sort.Slice(bids.Demand, func(i, j int) bool { return bids.Demand[i].Price > bids

    NSupplyBids := len(bids.Supply)
    NDemandBids := len(bids.Demand)
    NBids := NSupplyBids + NDemandBids

    // We cannot use an array since it's sizes needs to be known at compile time
    // Instead we use a slice of slices, see:
    // https://mikevella.info/post/2d-slices-and-arrays-in-golang/
    bidTable := make([][]int64, NBids)
    for i := range bidTable {
        bidTable[i] = make([]int64, 6)
```

```
    }

    // No NaN for integers, so we assign an int to the variable NaN
    // Since the prices are always > 0, we can chose any int < 1
    var NaN int64
    NaN = -23
    // Fill in supply values
    j := 0
    var X int64
    X = 0
    for i := 0; i < NSupplyBids; i++ {
         // Accumulate quantity
        bidTable[j][0] = X - bids.Supply[i].Quantity
        bidTable[j][1] = NaN  // instead of a float NaN
        bidTable[j][2] = bids.Supply[i].Price
        bidTable[j][3] = NaN  // -1 instead of NaN
        bidTable[j][4] = bids.Supply[i].Price
        bidTable[j][5] = NaN // placeholder
        X = bidTable[j][0] // Accumulate quantity
        j++
    }
    // Fill in demand values
    X = 0
    for i := 0; i < NDemandBids; i++ {
         // Accumulate quantity
        bidTable[j][0] = X + bids.Demand[i].Quantity
        bidTable[j][1] = bids.Demand[i].Price
        bidTable[j][2] = NaN  // instead of NaN
        bidTable[j][3] = bids.Demand[i].Price
        bidTable[j][4] = NaN  // instead of NaN
        bidTable[j][5] = NaN // placeholder
        X = bidTable[j][0] // Accumulate quantity
        j++
    }

    // Sort bidTable in increasing X (column 0)
    sort.Slice(bidTable[:], func(i, j int) bool { return bidTable[i][0] < bidTable[
    var row int
    // Fill in the filled prices (cols 3 and 4), and the price diff (col 5)
    for i := NBids - 2; i >= 0; i-- {  // begin w 2nd to last row and move to 1st
        if bidTable[i][3] == NaN {
            bidTable[i][3] = bidTable[i+1][3]
        }
        if bidTable[i][4] == NaN {
            bidTable[i][4] = bidTable[i+1][4]
        }

        if bidTable[i][3] == NaN || bidTable[i][4] == NaN {
            bidTable[i][5] = NaN
        } else {
            bidTable[i][5] = bidTable[i][3] - bidTable[i][4]
        }
        /* Find the critical row for MCP and BPP
        * The sign (+/–) of this column is more important than the value,
        * where a value of 0 should be represented by a positive sign.
        */
```

```
        if bidTable[i][5] < 0 {
            row = i - 1
            }
    }

    // Hack to fix merge redundant rows for Vertical Intersection
    for i := 1; i < NBids; i++ {
        if bidTable[i][0] == bidTable[i-1][0] {
            if bidTable[i-1][1] == NaN {
                bidTable[i-1][1] = bidTable[i][1]
            } else if bidTable[i][1] == NaN {
                bidTable[i][1] = bidTable[i-1][1]
            }
            if bidTable[i-1][2] == NaN {
                bidTable[i-1][2] = bidTable[i][2]
            } else if bidTable[i][2] == NaN {
                bidTable[i][2] = bidTable[i-1][2]
            }
            if bidTable[i-1][3] == NaN {
                bidTable[i-1][3] = bidTable[i][3]
            } else if bidTable[i][3] == NaN {
                bidTable[i][3] = bidTable[i-1][3]
            }
            if bidTable[i-1][4] == NaN {
                bidTable[i-1][4] = bidTable[i][4]
            } else if bidTable[i][1] == NaN {
                bidTable[i][4] = bidTable[i-1][4]
            }
        }
    }

    // Find MCP and BPP
    var mcp int64
    var bpp int64
    // Horizontal Supply Intersection
    if bidTable[row][2] == NaN {
        mcp = bidTable[row][4]
        bpp = bidTable[row][3]
    // Horizontal Demand Intersection
    } else if bidTable[row][1] == NaN {
        for i := row; i >= 0; i-- {
            if bidTable[i][1] != NaN {
                mcp = bidTable[i][4]
                bpp = bidTable[i][1]
                break
            }
        }
    // Vertical intersection
    } else if bidTable[row][1] != NaN && bidTable[row][2] != NaN {
        mcp = bidTable[row][2]
        bpp = bidTable[row][1]
    }

    fmt.Println()
    fmt.Println("MCP row:", bidTable[row])
    fmt.Println()
```

```
    fmt.Println("MCP:", mcp, "BPP:", bpp)
}
```

Horizontal Demand Intersection

```
package main

import (
    "fmt"
    "sort"
)

func main() {

    type Bid struct {
        Price       int64 `json:"price"`
        Quantity    int64 `json:"quantity"`
        AgentID     string `json:"agent_id"`
        AgentType   string `json:"agent_type"`
        Unit        string `json:"unit"`
        TimeStamp   int64 `json:"time_stamp"`
    }

    type Bids struct {
        Supply []Bid
        Demand []Bid
        }

    bids := Bids{}
    bid := Bid{}

    // Demand
    bid.Price = 150
    bid.Quantity = 400
    bid.AgentID = "KK"
    bid.AgentType = "HVAC"
    bid.Unit = "Wh"

    bids.Demand = append(bids.Demand, bid)

    bid.Price = 100
    bid.Quantity = 800
    bid.AgentID = "KT"
    bid.AgentType = "EV"
    bid.Unit = "Wh"

    bids.Demand = append(bids.Demand, bid)

    bid.Price = 40
    bid.Quantity = 300
```

```
        bid.AgentID = "RT"
        bid.AgentType = "BESS"
        bid.Unit = "Wh"

        bids.Demand = append(bids.Demand, bid)

        // Supply
        bid.Price = 50
        bid.Quantity = -600
        bid.AgentID = "TY"
        bid.AgentType = "PV"
        bid.Unit = "Wh"

        bids.Supply = append(bids.Supply, bid)

        bid.Price = 100
        bid.Quantity = -400
        bid.AgentID = "YY"
        bid.AgentType = "PV"
        bid.Unit = "Wh"

        bids.Supply = append(bids.Supply, bid)

        bid.Price = 120
        bid.Quantity = -700
        bid.AgentID = "YK"
        bid.AgentType = "PV"
        bid.Unit = "Wh"

        bids.Supply = append(bids.Supply, bid)


        fmt.Println("Demand:", bids.Demand)

        fmt.Println("Supply:", bids.Supply)

        // sort supply bids increasing
        sort.Slice(bids.Supply, func(i, j int) bool { return bids.Supply[i].Price < bids
        // sort demand bids decreasing
        sort.Slice(bids.Demand, func(i, j int) bool { return bids.Demand[i].Price > bids

        NSupplyBids := len(bids.Supply)
        NDemandBids := len(bids.Demand)
        NBids := NSupplyBids + NDemandBids


        // We cannot use an array since it's sizes needs to be known at compile time
        // Instead we use a slice of slices, see:
        // https://mikevella.info/post/2d-slices-and-arrays-in-golang/
        bidTable := make([][]int64, NBids)
        for i := range bidTable {
            bidTable[i] = make([]int64, 6)
        }

        // No NaN for integers, so we assign an int to the variable NaN
        // Since the prices are always > 0, we can chose any int < 1
```

```
	var NaN int64
	NaN = -23
	// Fill in supply values
	j := 0
	var X int64
	X = 0
	for i := 0; i < NSupplyBids; i++ {
	     // Accumulate quantity
	    bidTable[j][0] = X - bids.Supply[i].Quantity
	    bidTable[j][1] = NaN  // instead of a float NaN
	    bidTable[j][2] = bids.Supply[i].Price
	    bidTable[j][3] = NaN  // -1 instead of NaN
	    bidTable[j][4] = bids.Supply[i].Price
	    bidTable[j][5] = NaN // placeholder
	    X = bidTable[j][0] // Accumulate quantity
	    j++
	}
	// Fill in demand values
	X = 0
	for i := 0; i < NDemandBids; i++ {
	     // Accumulate quantity
	    bidTable[j][0] = X + bids.Demand[i].Quantity
	    bidTable[j][1] = bids.Demand[i].Price
	    bidTable[j][2] = NaN  // instead of NaN
	    bidTable[j][3] = bids.Demand[i].Price
	    bidTable[j][4] = NaN  // instead of NaN
	    bidTable[j][5] = NaN // placeholder
	    X = bidTable[j][0] // Accumulate quantity
	    j++
	}

	// Sort bidTable in increasing X (column 0)
	sort.Slice(bidTable[:], func(i, j int) bool { return bidTable[i][0] < bidTable[
	var row int
	// Fill in the filled prices (cols 3 and 4), and the price diff (col 5)
	for i := NBids - 2; i >= 0; i-- {  // begin w 2nd to last row and move to 1st
	    if bidTable[i][3] == NaN {
	        bidTable[i][3] = bidTable[i+1][3]
	    }
	    if bidTable[i][4] == NaN {
	        bidTable[i][4] = bidTable[i+1][4]
	    }

	    if bidTable[i][3] == NaN || bidTable[i][4] == NaN {
	        bidTable[i][5] = NaN
	    } else {
	        bidTable[i][5] = bidTable[i][3] - bidTable[i][4]
	    }
	    /* Find the critical row for MCP and BPP
	    * The sign (+/–) of this column is more important than the value,
	    * where a value of 0 should be represented by a positive sign.
	    */
	    if bidTable[i][5] < 0 {
	        row = i - 1
	        }
	}
```

```
    // Hack to fix merge redundant rows for Vertical Intersection
    for i := 1; i < NBids; i++ {
        if bidTable[i][0] == bidTable[i-1][0] {
            if bidTable[i-1][1] == NaN {
                bidTable[i-1][1] = bidTable[i][1]
            } else if bidTable[i][1] == NaN {
                bidTable[i][1] = bidTable[i-1][1]
            }
            if bidTable[i-1][2] == NaN {
                bidTable[i-1][2] = bidTable[i][2]
            } else if bidTable[i][2] == NaN {
                bidTable[i][2] = bidTable[i-1][2]
            }
            if bidTable[i-1][3] == NaN {
                bidTable[i-1][3] = bidTable[i][3]
            } else if bidTable[i][3] == NaN {
                bidTable[i][3] = bidTable[i-1][3]
            }
            if bidTable[i-1][4] == NaN {
                bidTable[i-1][4] = bidTable[i][4]
            } else if bidTable[i][1] == NaN {
                bidTable[i][4] = bidTable[i-1][4]
            }
        }
    }

    // Find MCP and BPP
    var mcp int64
    var bpp int64
    // Horizontal Supply Intersection
    if bidTable[row][2] == NaN {
        mcp = bidTable[row][4]
        bpp = bidTable[row][3]
    // Horizontal Demand Intersection
    } else if bidTable[row][1] == NaN {
        for i := row; i >= 0; i-- {
            if bidTable[i][1] != NaN {
                mcp = bidTable[i][4]
                bpp = bidTable[i][1]
                break
            }
        }
    // Vertical intersection
    } else if bidTable[row][1] != NaN && bidTable[row][2] != NaN {
        mcp = bidTable[row][2]
        bpp = bidTable[row][1]
    }

    fmt.Println()
    fmt.Println("MCP row:", bidTable[row])
    fmt.Println()
    fmt.Println("MCP:", mcp, "BPP:", bpp)
}
```

Vertical Intersection

```
package main

import (
    "fmt"
    "sort"
)

func main() {

    type Bid struct {
        Price       int64 `json:"price"`
        Quantity    int64 `json:"quantity"`
        AgentID     string `json:"agent_id"`
        AgentType   string `json:"agent_type"`
        Unit        string `json:"unit"`
        TimeStamp   int64 `json:"time_stamp"`
    }

    type Bids struct {
        Supply []Bid
        Demand []Bid
        }

    bids := Bids{}
    bid := Bid{}

    // Demand
    bid.Price = 150
    bid.Quantity = 400
    bid.AgentID = "KK"
    bid.AgentType = "HVAC"
    bid.Unit = "Wh"

    bids.Demand = append(bids.Demand, bid)

    bid.Price = 100
    bid.Quantity = 800
    bid.AgentID = "KT"
    bid.AgentType = "EV"
    bid.Unit = "Wh"

    bids.Demand = append(bids.Demand, bid)

    bid.Price = 40
    bid.Quantity = 300
    bid.AgentID = "RT"
    bid.AgentType = "BESS"
    bid.Unit = "Wh"
```

```
	bids.Demand = append(bids.Demand, bid)

	// Supply
	bid.Price = 50
	bid.Quantity = -600
	bid.AgentID = "TY"
	bid.AgentType = "PV"
	bid.Unit = "Wh"

	bids.Supply = append(bids.Supply, bid)

	bid.Price = 70
	bid.Quantity = -600
	bid.AgentID = "YY"
	bid.AgentType = "PV"
	bid.Unit = "Wh"

	bids.Supply = append(bids.Supply, bid)

	bid.Price = 120
	bid.Quantity = -500
	bid.AgentID = "YK"
	bid.AgentType = "PV"
	bid.Unit = "Wh"

	bids.Supply = append(bids.Supply, bid)


	fmt.Println("Demand:", bids.Demand)

	fmt.Println("Supply:", bids.Supply)

	// sort supply bids increasing
	sort.Slice(bids.Supply, func(i, j int) bool { return bids.Supply[i].Price < bids
	// sort demand bids decreasing
	sort.Slice(bids.Demand, func(i, j int) bool { return bids.Demand[i].Price > bids

	NSupplyBids := len(bids.Supply)
	NDemandBids := len(bids.Demand)
	NBids := NSupplyBids + NDemandBids


	// We cannot use an array since it's sizes needs to be known at compile time
	// Instead we use a slice of slices, see:
	// https://mikevella.info/post/2d-slices-and-arrays-in-golang/
	bidTable := make([][]int64, NBids)
	for i := range bidTable {
	    bidTable[i] = make([]int64, 6)
	}

	// No NaN for integers, so we assign an int to the variable NaN
	// Since the prices are always > 0, we can chose any int < 1
	var NaN int64
	NaN = -23
	// Fill in supply values
	j := 0
```

```
    var X int64
    X = 0
    for i := 0; i < NSupplyBids; i++ {
         // Accumulate quantity
        bidTable[j][0] = X - bids.Supply[i].Quantity
        bidTable[j][1] = NaN  // instead of a float NaN
        bidTable[j][2] = bids.Supply[i].Price
        bidTable[j][3] = NaN  // -1 instead of NaN
        bidTable[j][4] = bids.Supply[i].Price
        bidTable[j][5] = NaN // placeholder
        X = bidTable[j][0] // Accumulate quantity
        j++
    }
    // Fill in demand values
    X = 0
    for i := 0; i < NDemandBids; i++ {
         // Accumulate quantity
        bidTable[j][0] = X + bids.Demand[i].Quantity
        bidTable[j][1] = bids.Demand[i].Price
        bidTable[j][2] = NaN  // instead of NaN
        bidTable[j][3] = bids.Demand[i].Price
        bidTable[j][4] = NaN  // instead of NaN
        bidTable[j][5] = NaN // placeholder
        X = bidTable[j][0] // Accumulate quantity
        j++
    }

    // Sort bidTable in increasing X (column 0)
    sort.Slice(bidTable[:], func(i, j int) bool { return bidTable[i][0] < bidTable[
    var row int
    // Fill in the filled prices (cols 3 and 4), and the price diff (col 5)
    for i := NBids - 2; i >= 0; i-- {  // begin w 2nd to last row and move to 1st
        if bidTable[i][3] == NaN {
            bidTable[i][3] = bidTable[i+1][3]
        }
        if bidTable[i][4] == NaN {
            bidTable[i][4] = bidTable[i+1][4]
        }

        if bidTable[i][3] == NaN || bidTable[i][4] == NaN {
            bidTable[i][5] = NaN
        } else {
            bidTable[i][5] = bidTable[i][3] - bidTable[i][4]
        }
        /* Find the critical row for MCP and BPP
        * The sign (+/-) of this column is more important than the value,
        * where a value of 0 should be represented by a positive sign.
        */
        if bidTable[i][5] < 0 {
            row = i - 1
            }
    }

    // Hack to fix merge redundant rows for Vertical Intersection
    for i := 1; i < NBids; i++ {
        if bidTable[i][0] == bidTable[i-1][0] {
```

```
            if bidTable[i-1][1] == NaN {
                bidTable[i-1][1] = bidTable[i][1]
            } else if bidTable[i][1] == NaN {
                bidTable[i][1] = bidTable[i-1][1]
            }
            if bidTable[i-1][2] == NaN {
                bidTable[i-1][2] = bidTable[i][2]
            } else if bidTable[i][2] == NaN {
                bidTable[i][2] = bidTable[i-1][2]
            }
            if bidTable[i-1][3] == NaN {
                bidTable[i-1][3] = bidTable[i][3]
            } else if bidTable[i][3] == NaN {
                bidTable[i][3] = bidTable[i-1][3]
            }
            if bidTable[i-1][4] == NaN {
                bidTable[i-1][4] = bidTable[i][4]
            } else if bidTable[i][1] == NaN {
                bidTable[i][4] = bidTable[i-1][4]
            }
        }
    }

    // Find MCP and BPP
    var mcp int64
    var bpp int64
    // Horizontal Supply Intersection
    if bidTable[row][2] == NaN {
        mcp = bidTable[row][4]
        bpp = bidTable[row][3]
    // Horizontal Demand Intersection
    } else if bidTable[row][1] == NaN {
        for i := row; i >= 0; i-- {
            if bidTable[i][1] != NaN {
                mcp = bidTable[i][4]
                bpp = bidTable[i][1]
                break
            }
        }
    // Vertical intersection
    } else if bidTable[row][1] != NaN && bidTable[row][2] != NaN {
        mcp = bidTable[row][2]
        bpp = bidTable[row][1]
    }

  fmt.Println("\nbidTable")
    for i := 0; i < NBids; i++ {
        fmt.Println(bidTable[i])
    }

    fmt.Println()
    fmt.Println("MCP row:", bidTable[row])
    fmt.Println()
    fmt.Println("MCP:", mcp, "BPP:", bpp)
}
```