Physical Path Planning using a Network of Learning Sensors: Implementation and Testing

Amir Rasouli

The Active and Attentive Vision Lab
Department of Electrical Engineering and Computer Science, York University, Toronto

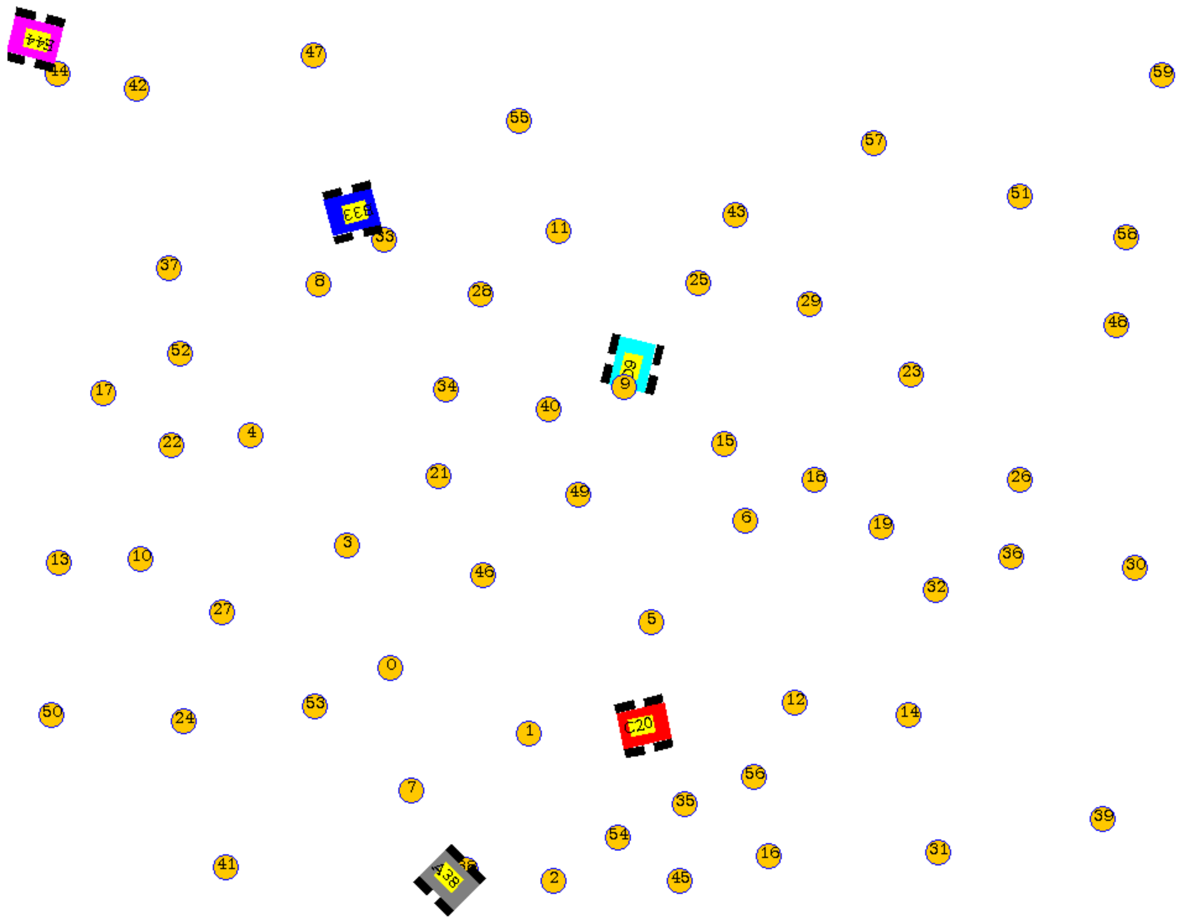
November 5, 2015

Swarm of Interacting Reinforcement Learners (SWIRLs)

- A physical path planning approach
- Network of sensors interacting to learn the best path
- Robots ask sensors where to go next

www.flickr.com

SWIRLs Environments



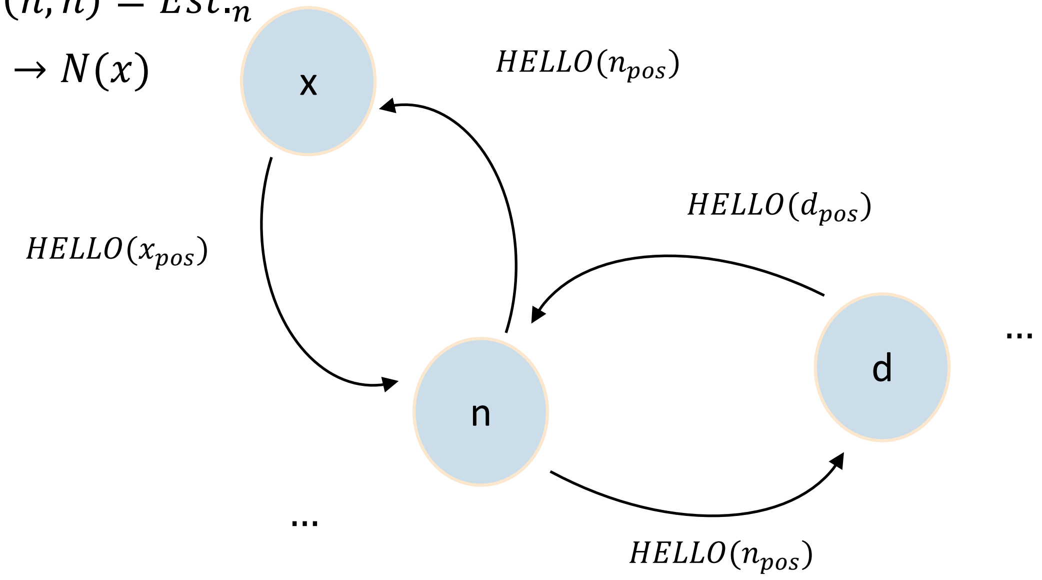
SWIRLs Stages of Operation

- **Initialization**

$Q_x(x, x) = 0$

$Q_x(n, n) = Est._n$

$n \rightarrow N(x)$

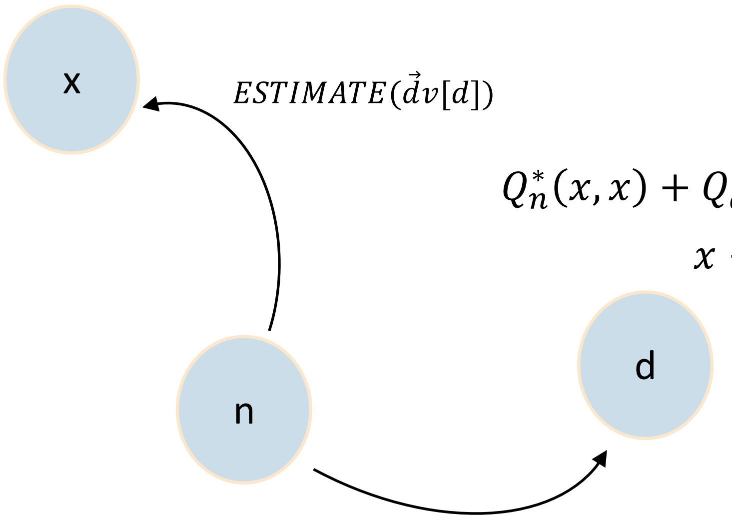


SWIRLs Stages of Operation

- **Initialization**
- **Estimation**

$$Q_n^*(d,d) + Q_x(n,n) \rightarrow Q_x(n,d)$$

$$d \rightarrow V(x)$$



$$ESTIMATE(\vec{d}v[d])$$

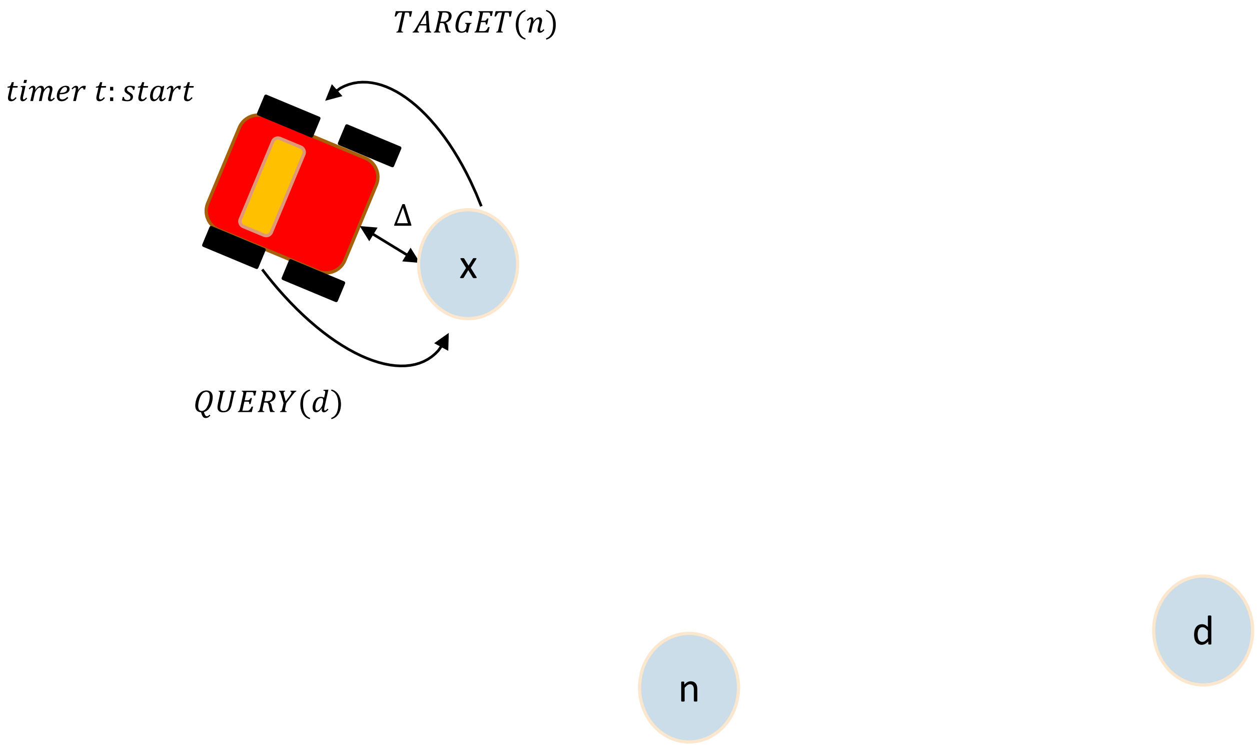
$$Q_n^*(x,x) + Q_d(n,n) \rightarrow Q_d(n,x)$$

$$x \rightarrow V(d)$$

$$ESTIMATE(\vec{d}v[x])$$

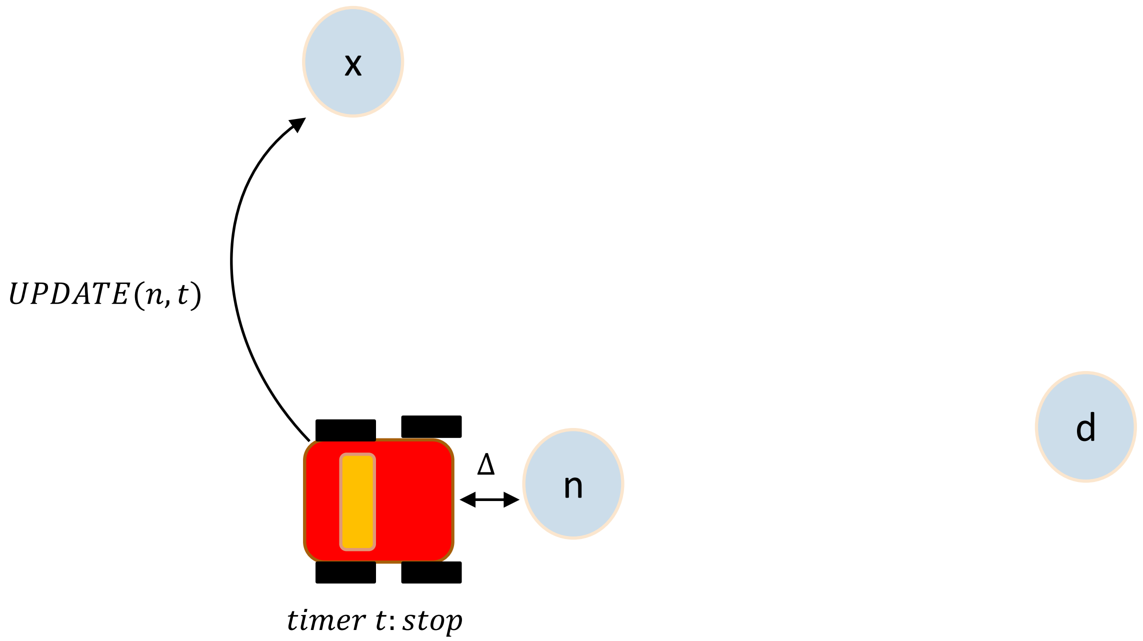
SWIRLs Stages of Operation

- **Initialization**
- **Estimation**
- **Query**



SWIRLs Stages of Operation

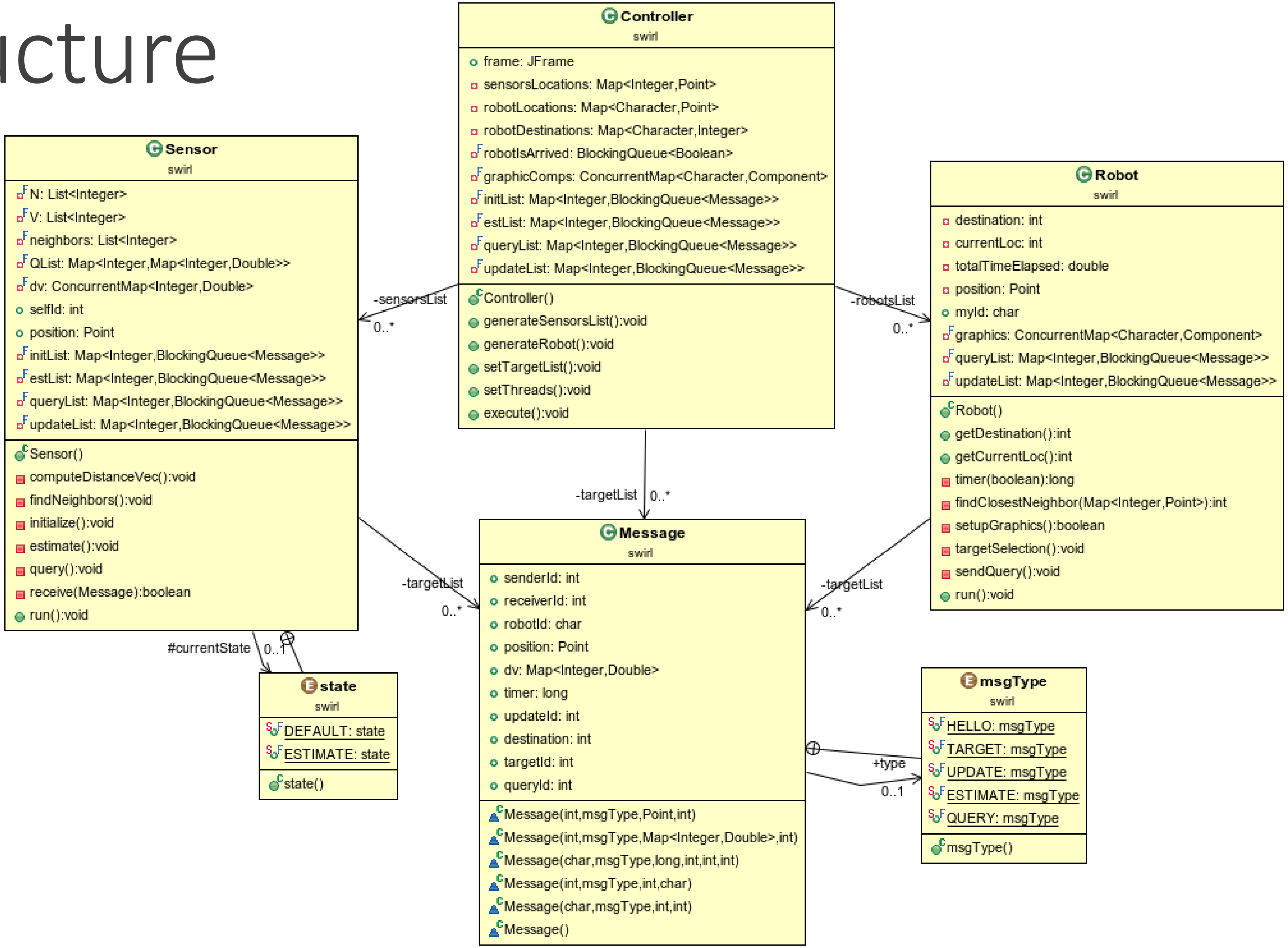
- **Initialization**
- **Estimation**
- **Query**
- **Update**



Implementations

- Sequential Implementation
  - All processing and message passing is on a **single thread**
- Semi-multi-threaded
  - Sensors and message passing on a **single thread**
  - Robots are run on **individual threads**
- Multi-threaded
  - Sensors and robots are run on **individual threads**
  - Communication is through **shared memory**

Code Structure



Sequential Controller

```
package swirl;
public class SwirlSequential {
private final List<SensorSeq> sensorsList = new ArrayList<SensorSeq>();
private final List<RobotSeq> robotsList = new ArrayList<RobotSeq>();
…
public void execute()
{
        initialize();
        List<Message> robotsRequestList = new ArrayList<Message>();
        Map<Character, Message> targetList = new HashMap<Character,Message>();
        …
        while(true)
        {
                if (timer(…)){ estimate();}
                for (int rIdx = 0; rIdx < this.robotsList.size(); rIdx++)
                {
                        robotsRequestList.add(this.robotsList.get(rIdx).operate(…));
                }
                …
                for (int m = 0 ; m < robotsRequestList.size(); m++)
                {
                        …
                        targetList.add(sensorsList.get(…).receive(robotsRequestList.get(m)));
                }
        }
}
```

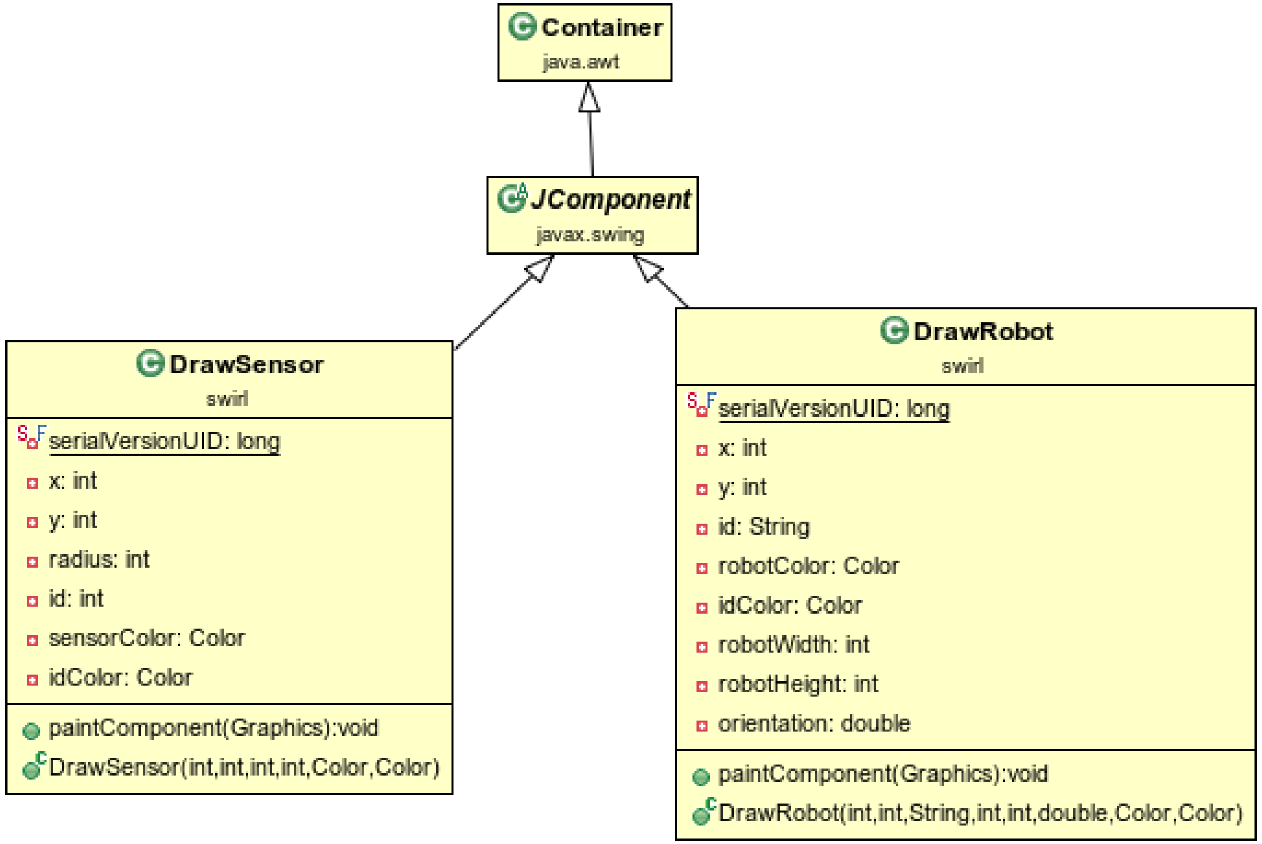
Semi-Multi-Threaded Controller

```
package swirl;
public class SwirlSemiMultiThreaded {
private final ConcurrentMap<Character, Message> targetList = new ConcurrentHashMap<Character,Message>();
private final BlockingQueue<Message> robotsRequestList = new ArrayBlockingQueue<Message>(SwirlController.NUMBER_OF_ROBOTS*2);
…
public void execute()
{
        …
        setThreads();
        BlockingQueue<Message> rejectList = new ArrayBlockingQueue<Message>(SwirlController.NUMBER_OF_ROBOTS);
        while(true)
        {
                if (timer(…)){...}
                while (!robotsRequestList.isEmpty())
                {
                        …
                        if (targetMsg.targetId == -1)
                        {
                                rejectList.offer(…);
                        }
                        …
                }
                        while (!rejectList.isEmpty())
                        {
                                this.robotsRequestList.offer(rejectList.poll());
                        }
                }
        }
}
```

Multi-Threaded Controller

```
package swirl;
public class SwirlMultiThread {
	private final Map<Integer,BlockingQueue<Message>> initList =  new HashMap<Integer,BlockingQueue<Message>>();
	private final Map<Integer,BlockingQueue<Message>>  estList = new HashMap<Integer,BlockingQueue<Message>>();
	private final Map<Integer,BlockingQueue<Message>>  queryList = new HashMap<Integer,BlockingQueue<Message>>();
	private final Map<Integer,BlockingQueue<Message>>  updateList = new HashMap<Integer,BlockingQueue<Message>>();
	private final ConcurrentMap<Character, Message> targetList = new ConcurrentHashMap<Character,Message>();
…
public void execute()
{
	…
	setThreads();
	…
}
```

Graphics


Container
java.awt
JComponent
javax.swing
DrawSensor
swirl
serialVersionUID: long
x: int
y: int
radius: int
id: int
sensorColor: Color
idColor: Color
paintComponent(Graphics):void
DrawSensor(int,int,int,int,Color,Color)
DrawRobot
swirl
serialVersionUID: long
x: int
y: int
id: String
robotColor: Color
idColor: Color
robotWidth: int
robotHeight: int
orientation: double
paintComponent(Graphics):void
DrawRobot(int,int,String,int,int,double,Color,Color)

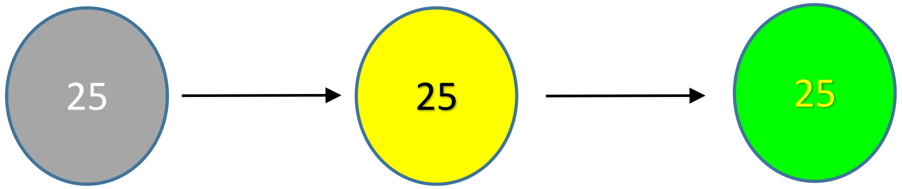
Multi-Threading in Graphics2D

```
SwingWorker<Boolean, DrawSensor> worker = new SwingWorker <Boolean, DrawSensor>()
{
        protected Boolean doInBackground() throws Exception
        {
                …
                DrawSensor s = new DrawSensor(…);
                publish(s);
                return true;
        }
        protected void process(java.util.List<DrawSensor> chunks) {
                frame.getContentPane().remove(…));
                graphics.put((char)selfId,frame.getContentPane().add(chunks.get(chunks.size()-1)));
                frame.revalidate();
                frame.repaint();
        }
};

worker.execute();
```

Simulation

Sensor



Uninitialized Default Estimate

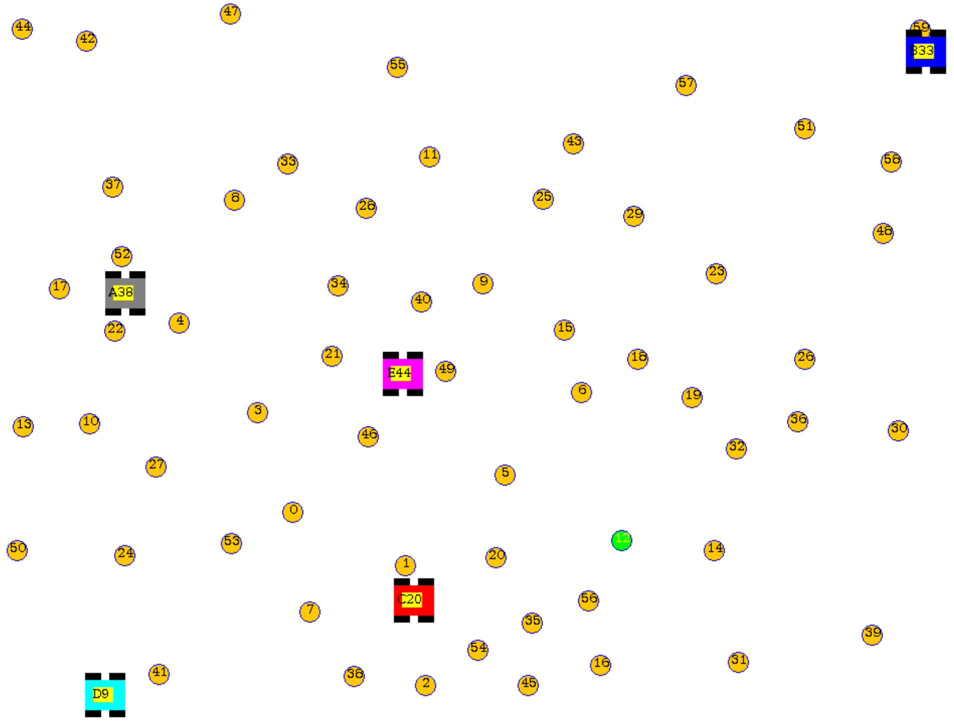
Robot

E = ID 44 = Destination

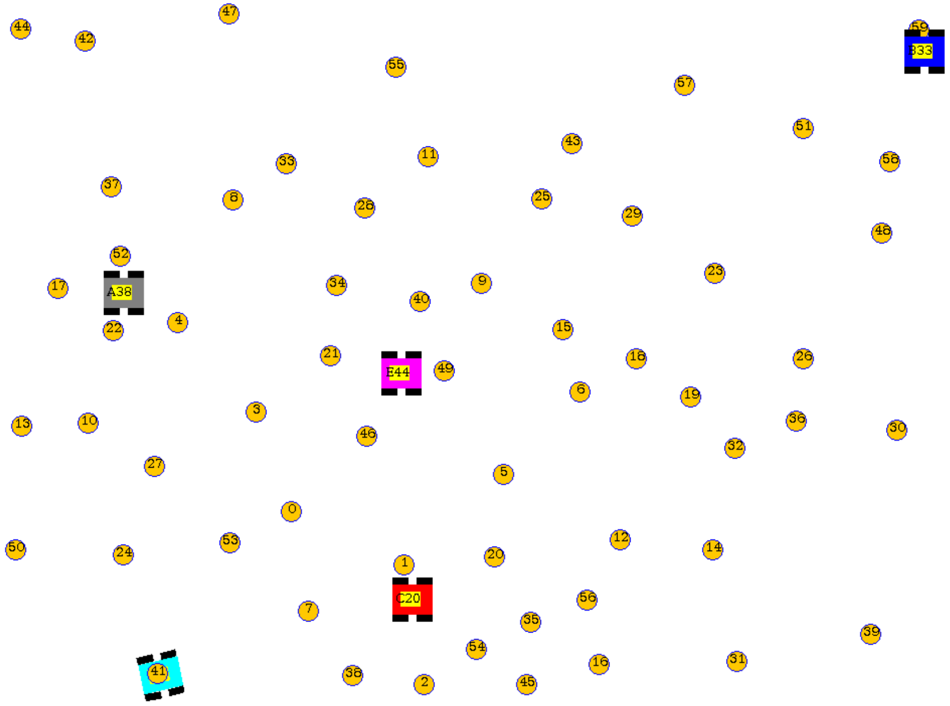
Sequential Run

Sequential Run

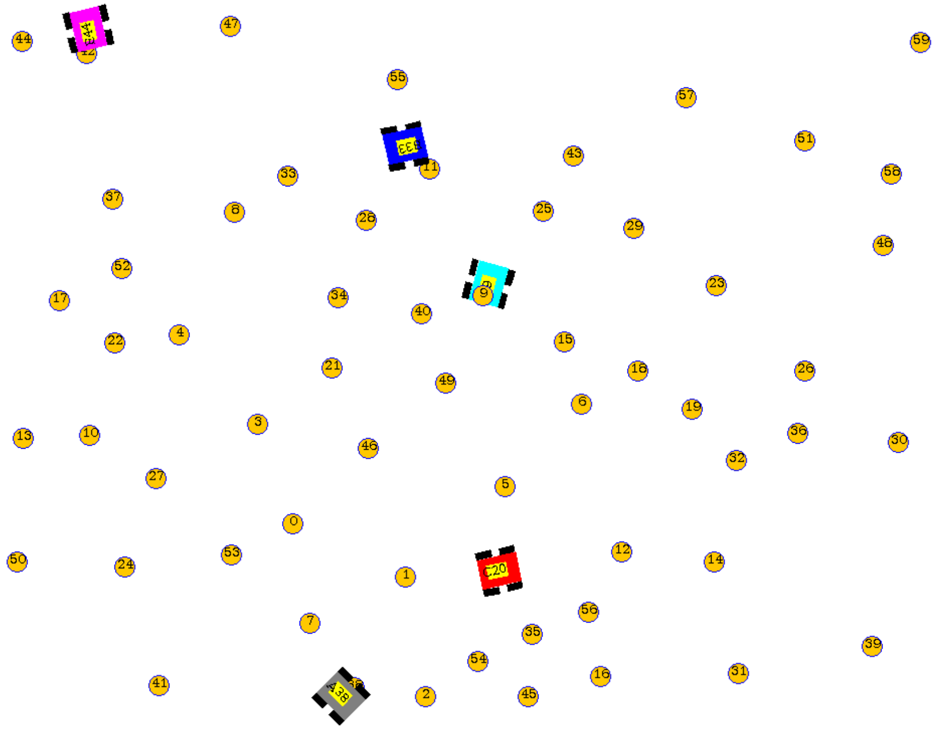
Sequential Run



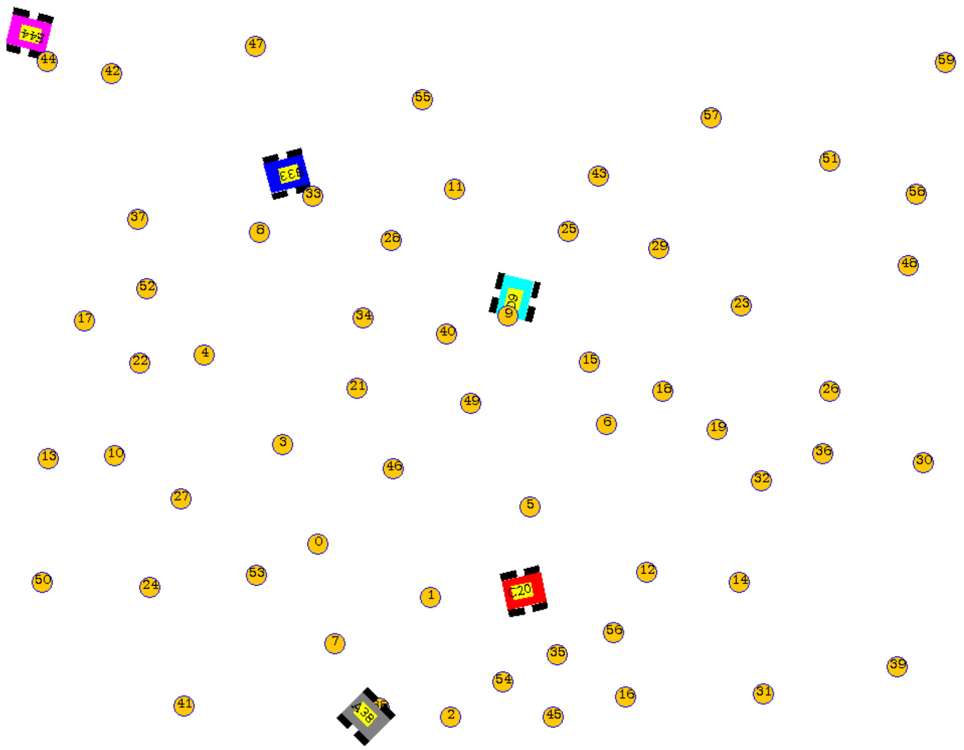
Sequential Run



Sequential Run

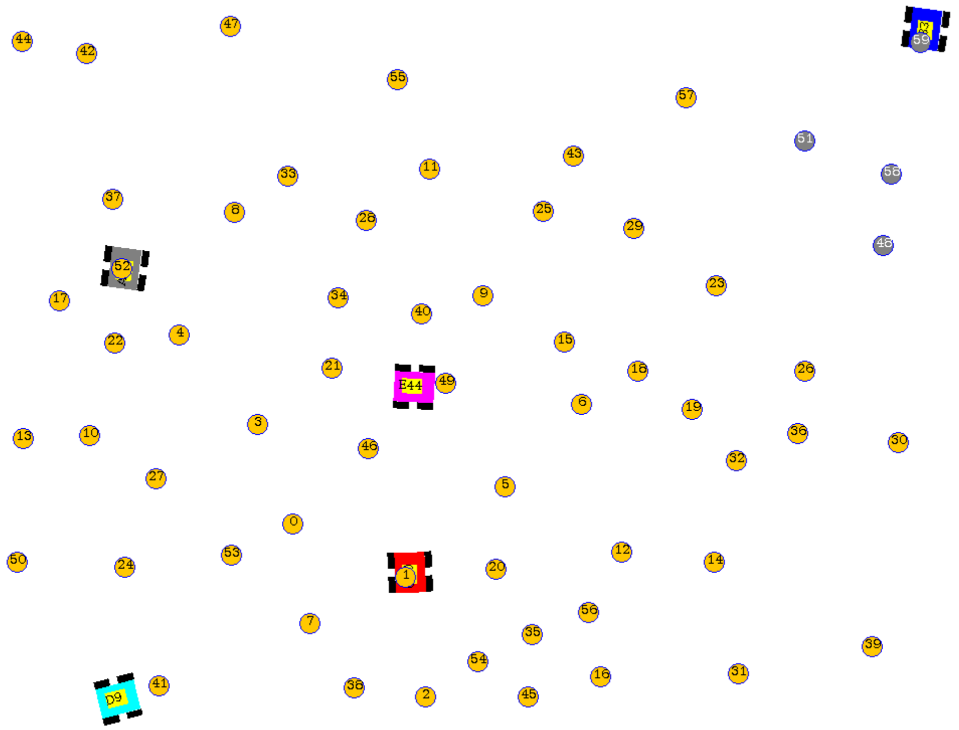


Sequential Run

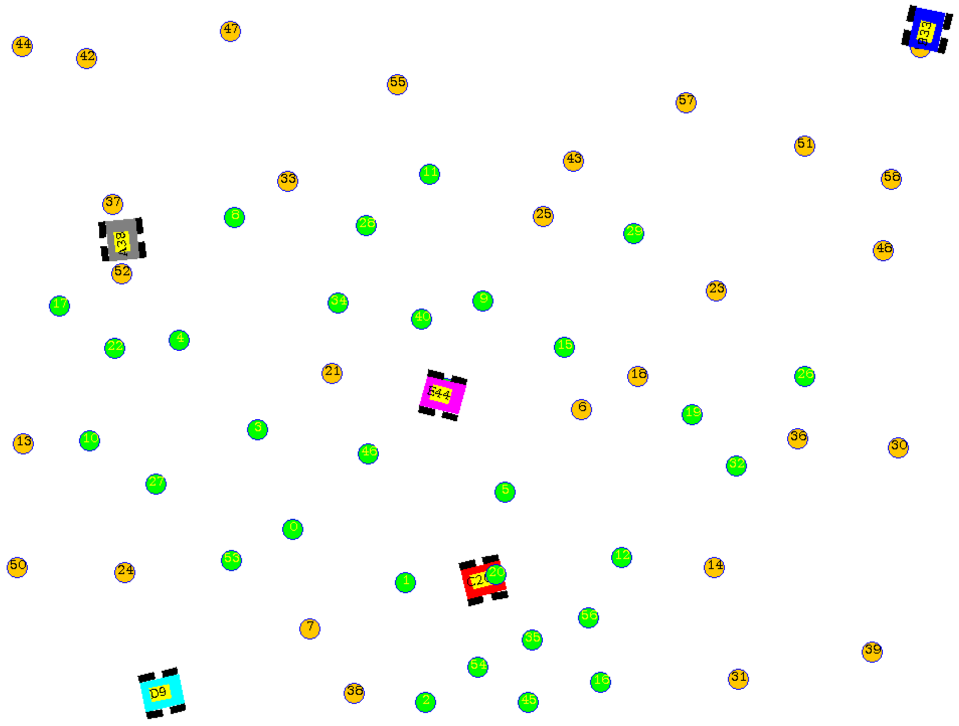


Multi-Threaded Run

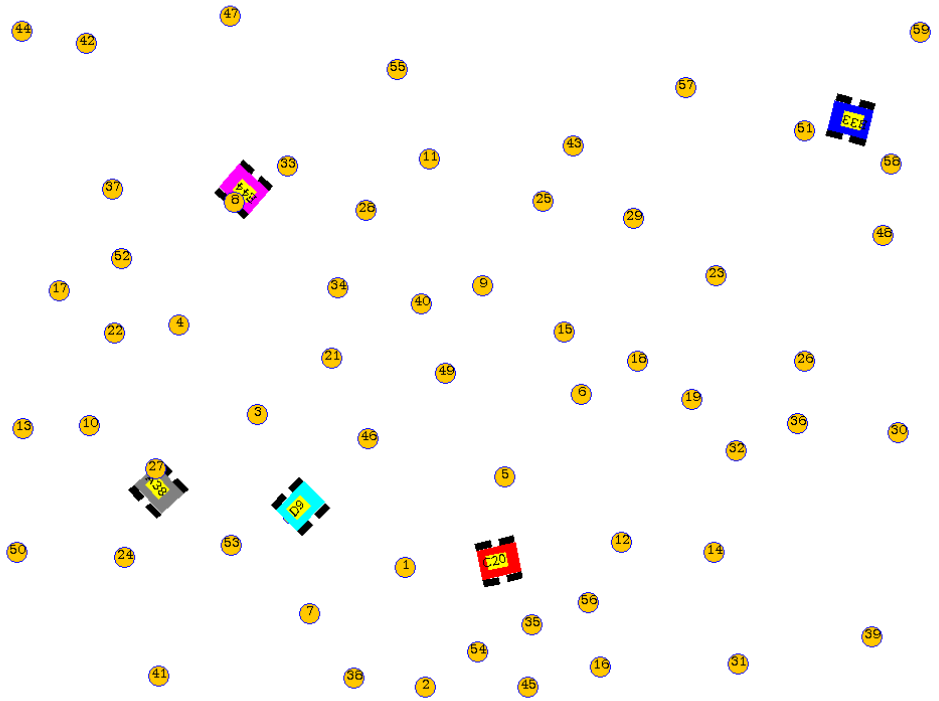
Multi-Threaded Run



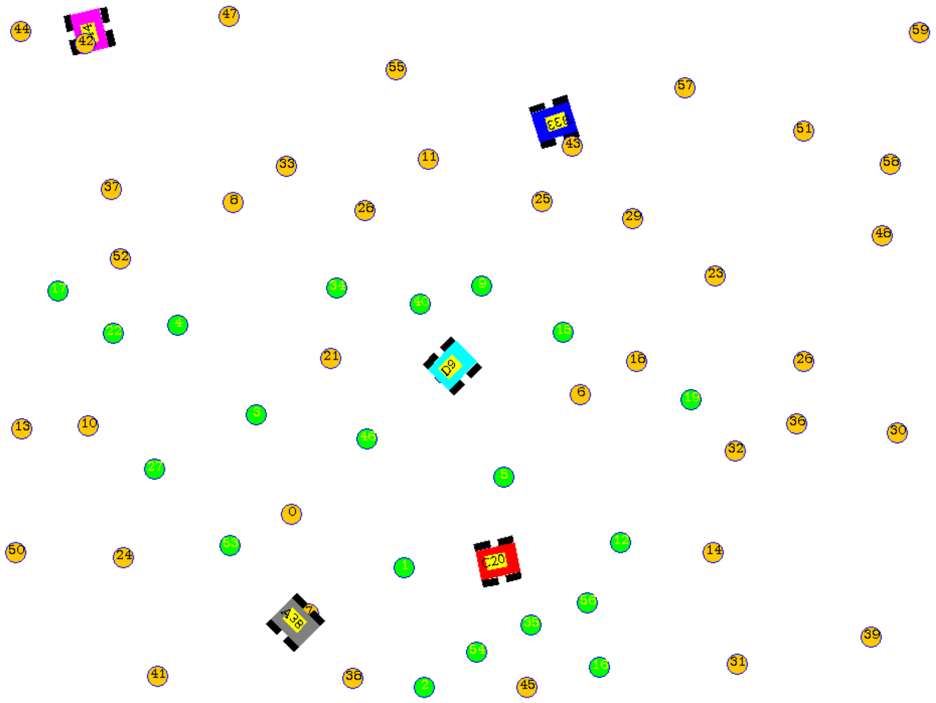
Multi-Threaded Run



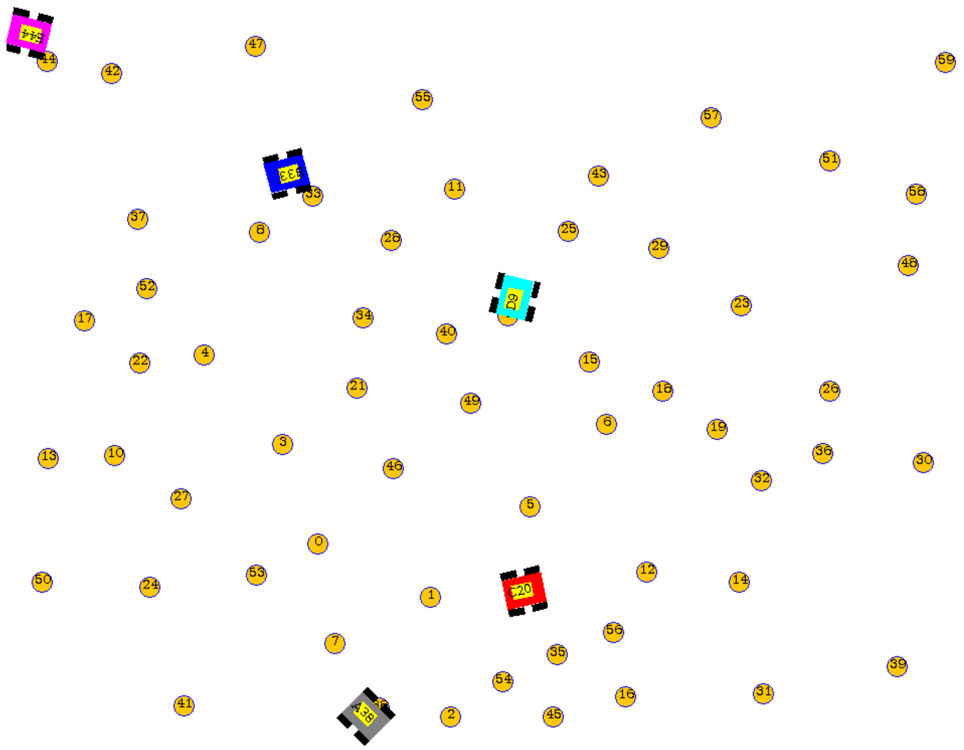
Multi-Threaded Run



Multi-Threaded Run



Multi-Threaded Run

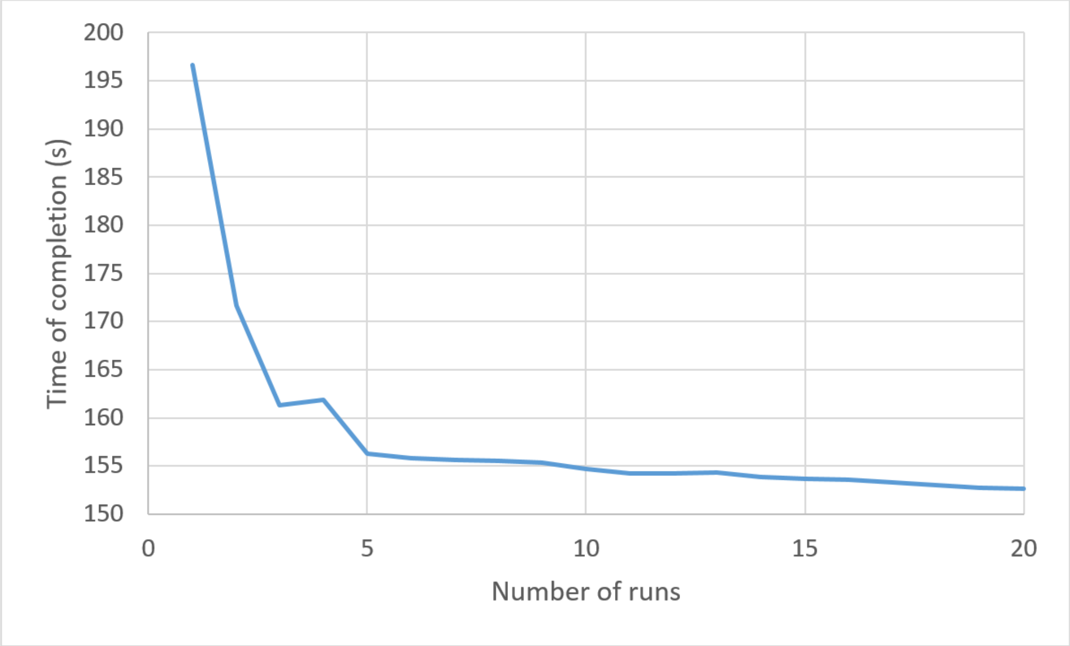


Testing

- **Functionality**
  - System behaves as expected
- **Fault-free**
  - E.g. DeadLock, ConcurrentModification and NullPointer exceptions
- **Performance**
  - What is the best concurrent structure
- **General Purpose Test**
  - Unexpected issues

Functionality

- **Case 1 :** Sensors get to know all other sensors
  - Random experiments with 4-10 robots and 10-300 sensors
- **Case 2:** Learning
  - Full learning hard to measure
  - Run the same experiments multiple times



20 runs with identical configuration using 4 robots and 1280 sensors

Fault-free

- Deadlock caused by BlockingQueue
- Null return from the queues
- Concurrent modification of Maps
- Ongoing detection by setting up all possible scenarios

Performance

- **Case 1:** Code Structure,
  - e.g. order of message handling
  - Timing of estimates
- **Case 2:** Handling Concurrency
  - *ConcurrentMap* is more efficient than *Collection.Synchronized()*
  - Single-Channel vs Four-Channel Communication no significant difference

		Number of Robots						
		5	6	7	8	9	10	11
Number of sensors	60	0.114	-11.307	0.031	0.29	-12.925	0.291	0.052
	100	0.36	-4.494	1.035	0.473	1.105	0.894	0.346
	140	1.277	2.064	1.866	1.383	-2.63	1.489	10.5
	180	1.219	2.172	7.268	2.193	2.145	-9.714	6.359
	220	2.088	4.8	-8.025	-6.699	2.815	1.325	9.11
	260	7.564	1.671	12.176	2.793	9.035	-7.868	2.568
	300	2.11	5.361	5.425	4.296	24.26	3.231	6.889

The results are obtained by subtracting single channel running times from the corresponding four channel trials

Unexpected problems

- Over 300 of experiments
- Various systems
  - Intel Xeon 2.67 GHz with **24 single-threaded cores**
  - Intel I7 2.00 GHz with **2 double-threaded cores**
  - Intel Xeon 2.66 GHz with **8 single-threaded cores**
- Systematic change of parameters
  - Number of sensors
  - Number of robots
  - Maximum Speed of robots
  - Timing of estimate steps

Future Work

- Measure the performance of each implementation
- Observe how each method scales to the size of the system