Behavior-Based Programming

Chapter 4

Objectives

- Understand what behavior -based programming is
- Look at types of behaviors and the issues involved with developing them
- Look at how to implement some simple behaviors
- Understand how behaviors can be combined
- Examine how behaviors interact to control the robot
- Understand what is involved with learning behaviors
- Investigate simple neural networks
- Learn how to hardwire instinctive behavior networks.

What's in Here ?

- Behaviors
  - Reflex Behaviors
  - Taxic Behaviors
  - Adaptive Behavior
- Behavior Interaction
  - Behavior Arbitration
  - Robustness
- Programming Behaviors
  - Collision Avoidance
  - Escape
  - Homing
  - GPS Homing
  - Wall-Following
- Emergent Behaviors
  - Overview
- Learning Behaviors
  - Overview
- Artificial Neural Networks
  - Neural Networks
  - Back Propagation
  - Neural Network – Leg Coordination
- Neuron Networks
  - Overview
  - Collision Avoidance
  - Escape
  - Light Seeking
  - Wandering
  - Edge Following
  - Arbitration

Behaviors

A Definition

- ***Behavior*** :

  The way a machine acts or functions

- A behavior can be:
  - Explicitly programmed
    - primitive behaviors are programmed
    - separate modules that are plugged in together
  - Emergent
    - combined primitive behaviors produce more complex behaviors
    - often unforeseen behavior emerges

Types

- Robot reacts according to its pre-programmed behaviors, which can be:
  - *Reflex*
    - A fast stereotyped response triggered by a particular type of sensor input. The intensity and duration of the response is entirely governed by the intensity and duration of the sensor readings.
  - *Taxes*
    - Involve the orientation of the robot toward or away from some environmental stimulus such as light, temperature, energy etc..

Reflex Behaviors

- Used for avoiding, escaping, or minimizing the effects of undesirable environmental stimuli.

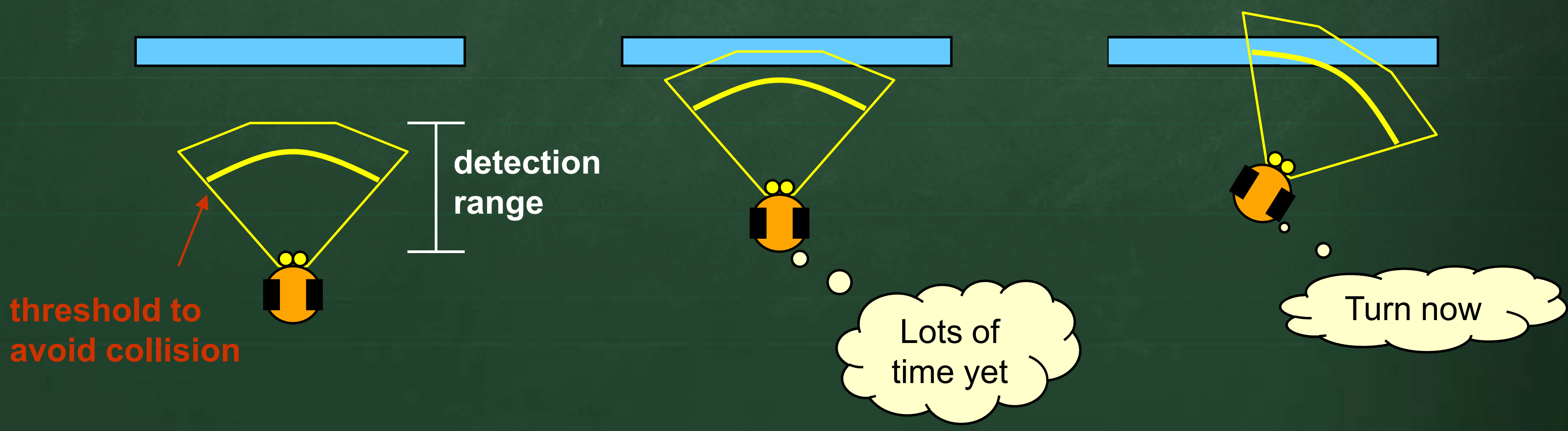
- Seven properties (as found in real life forms)
    1. Threshold
    2. Latency
    3. Refractory Period
    4. Temporal Summation
    5. Spatial Summation
    6. Momentum
    7. Habituation

Reflex Behaviors

1 – Threshold:

The minimum sensor reading required to cause the robot to react

e.g., Robot may sense obstacle ahead, but may not react until it is within certain range

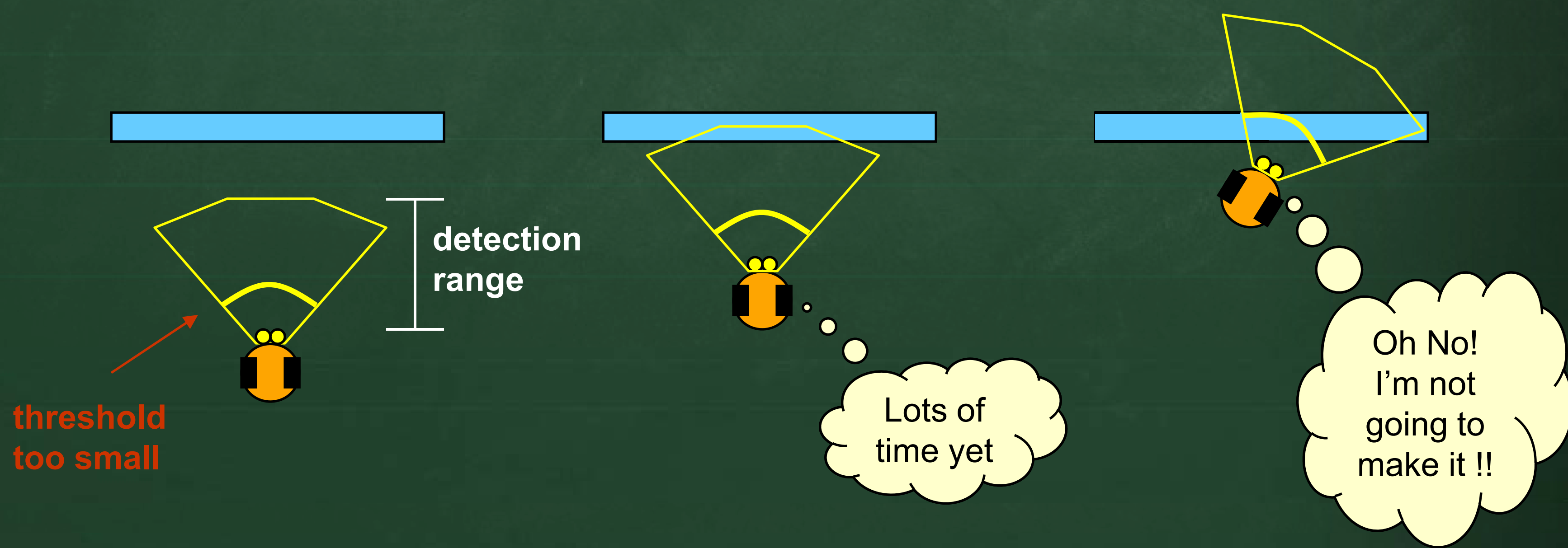


Reflex Behaviors

2 - Latency:

The time it takes for the robot to react once the sensor readings reach the threshold

e.g., If threshold too small, robot cannot react in time.

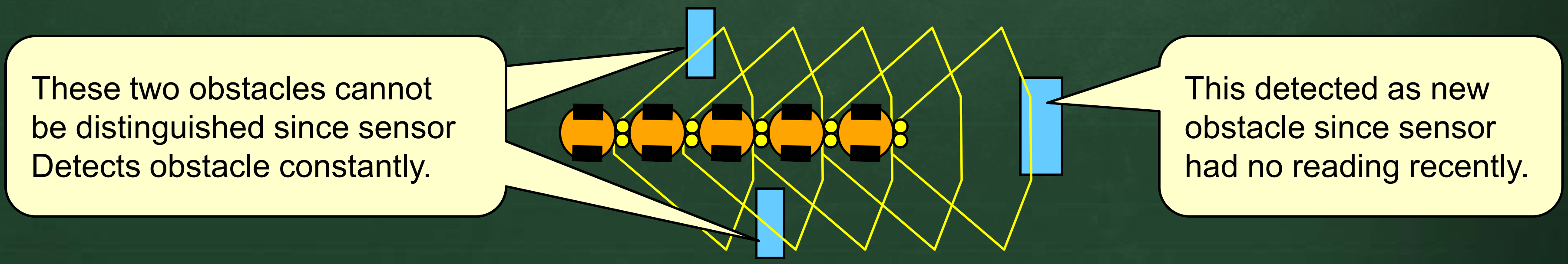


Reflex Behaviors

3 – Refractory Period:

The delay in response to the 2nd of two closely spaced stimuli.

e.g., Time between sensor readings may be too slow, so robot may re-adjust thresholds under certain environments (perhaps in slippery floors)

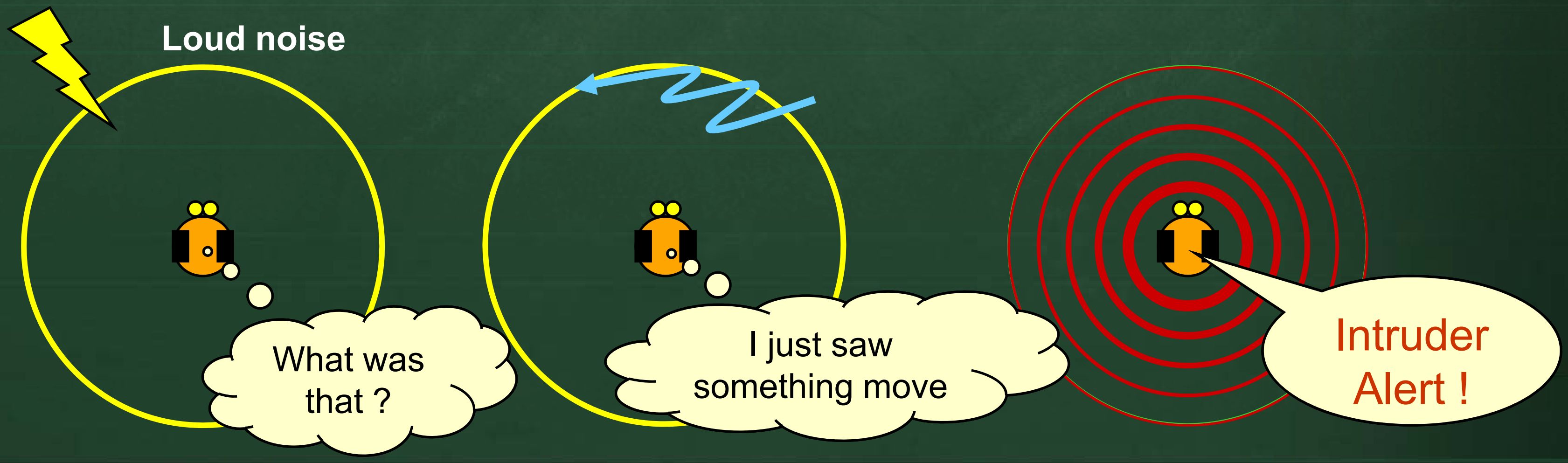


Reflex Behaviors

4 – Temporal Summation:

One sensor reading not enough to cause reaction, but when followed by additional sensory input, the reaction occurs

e.g., Security robot may sense a loud noise, but then wait for movement before sounding an alarm

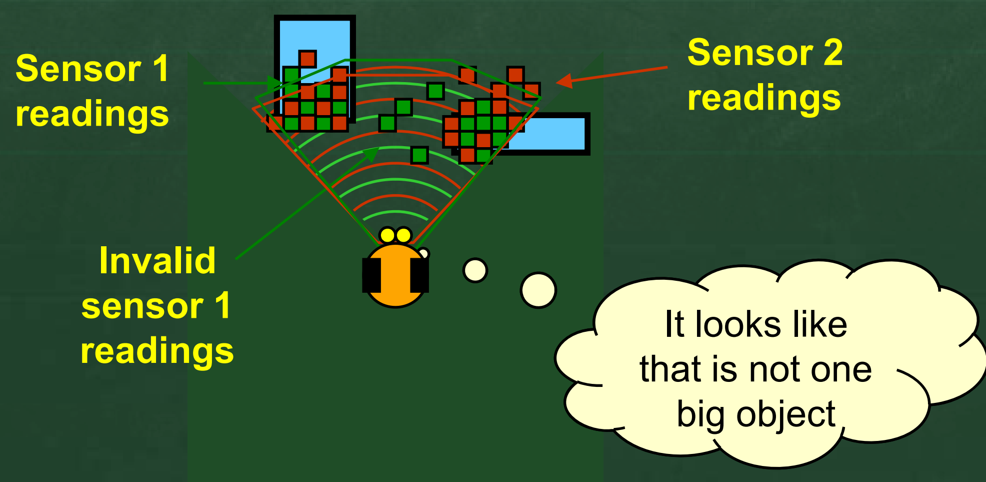


Reflex Behaviors

5 - Spatial Summation:

One sensor reading not enough to cause reaction, but when a second simultaneous sensor reading is observed, the reaction occurs

e.g., Due to sensor noise, invalid sensor readings occur and must be verified by additional nearby readings

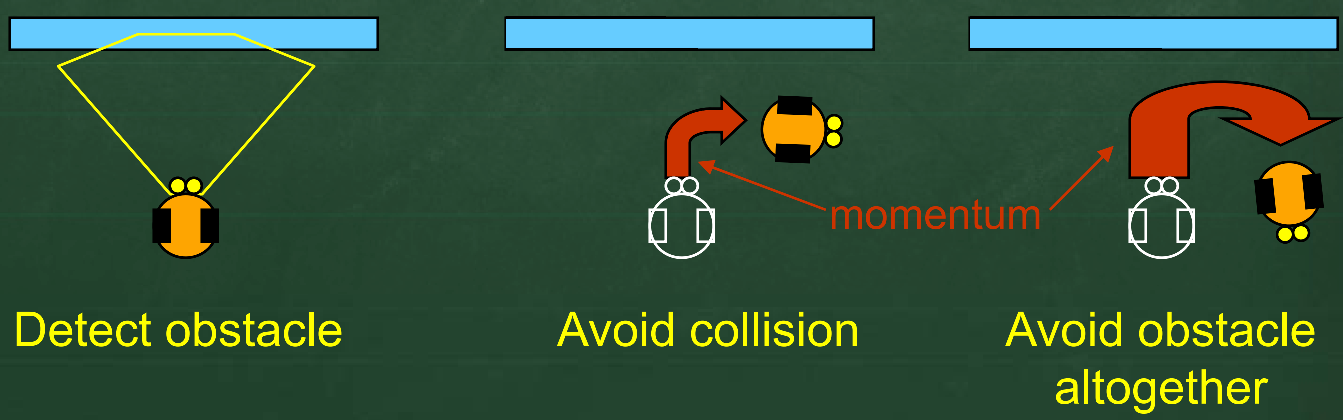


Reflex Behaviors

6 – Momentum:

The time that the robot's reflex takes to complete after the sensor stimulus has been removed

e.g., Upon encountering an obstacle, the robot may turn away for a specific amount of time or specific angle.

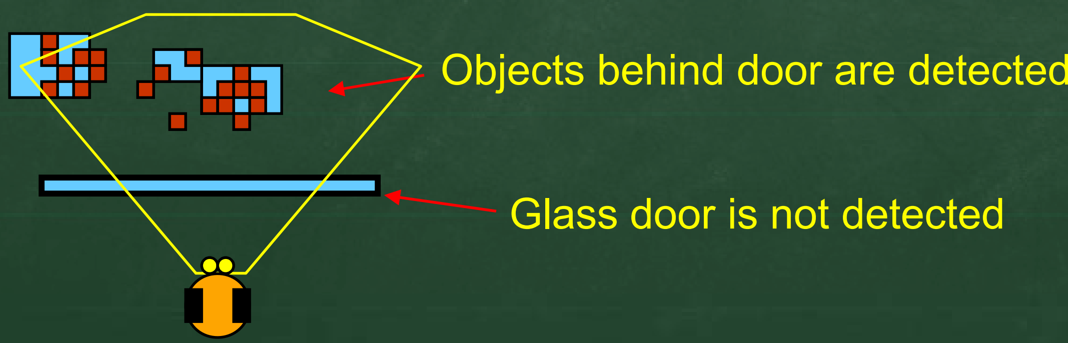


Reflex Behaviors

7 – Habituation:

The reduction in trustworthiness of a sensor over time, perhaps due to repeated false or inaccurate readings

e.g., Infrared light sensors are not trustworthy for collision avoidance in environments with many glass doors.



Taxic Behaviors

- There are 4 types of taxic reactions that represent the basic types of orientation strategies of a robot towards a stimulus:

  1. Klinotaxis
  2. Tropotaxis
  3. Telotaxis
  4. Light-compass reaction

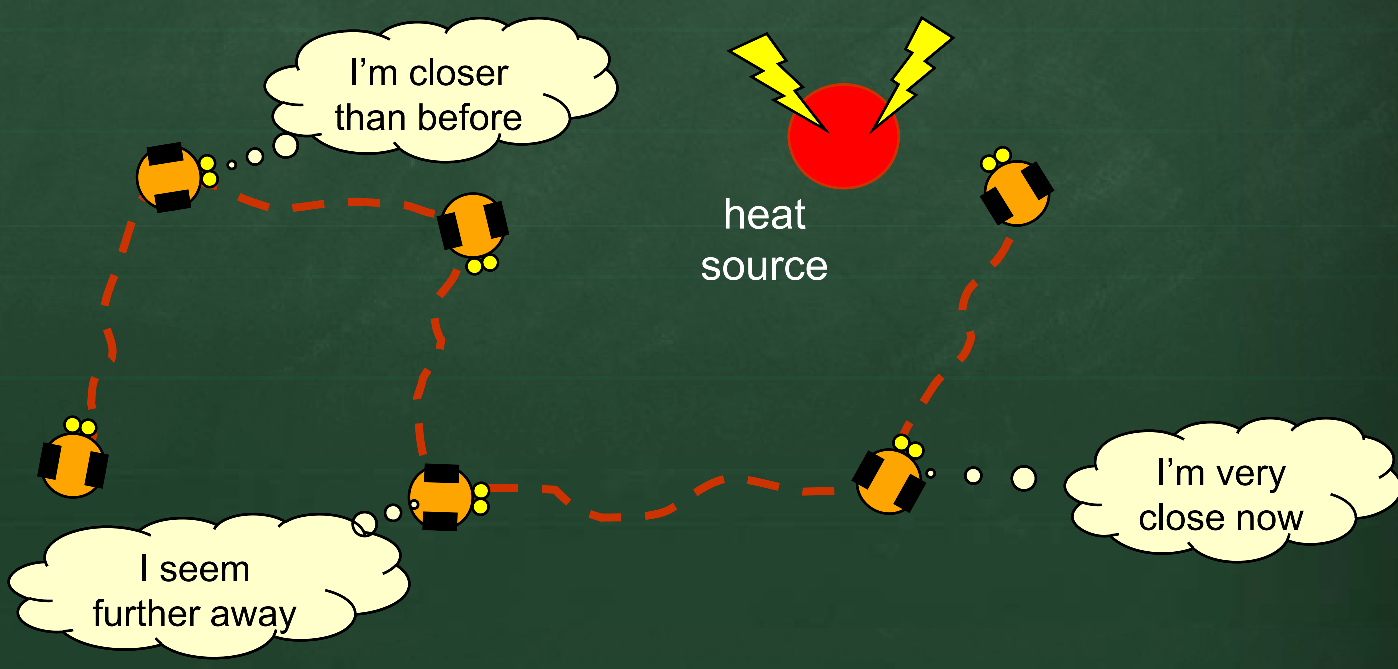
- Robot may use a variety of these at once depending on sensor types and the Behavior desired.

Taxic Behaviors

1 - Klinotaxis:

The use of successive comparisons to orient towards a sensor stimulus

e.g., Temperature sensing

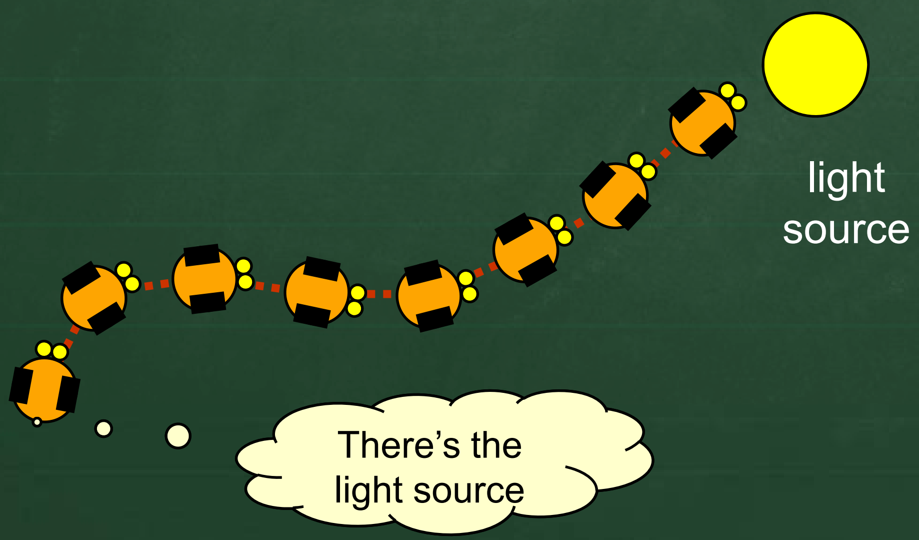


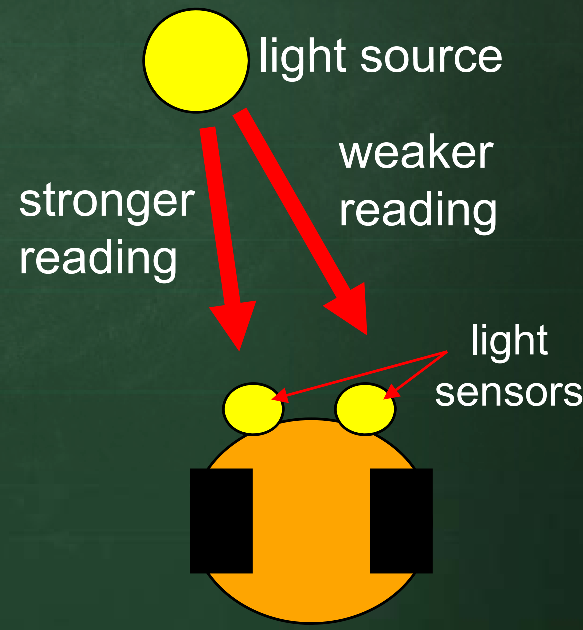
Taxic Behaviors

2 - Tropotaxis:

The balancing of two sensors, where the robot turns until sensors have equal readings, then proceeds forward

e.g., light seeking



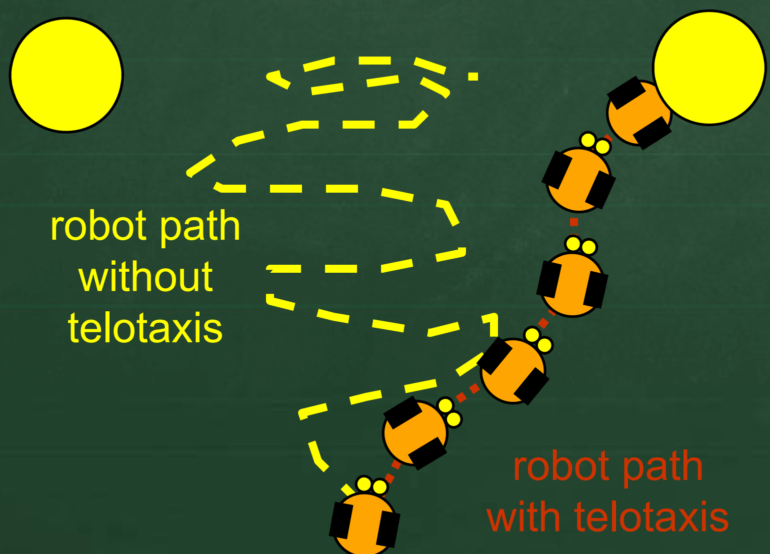


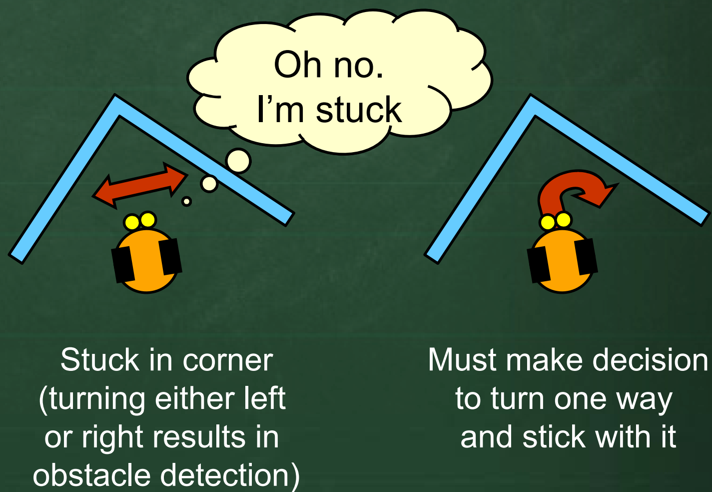
Taxic Behaviors

3 - Telotaxis:

When two sensor readings require opposing directions, robot must make decision by choosing and approaching one

e.g., Light seeking or corner escape



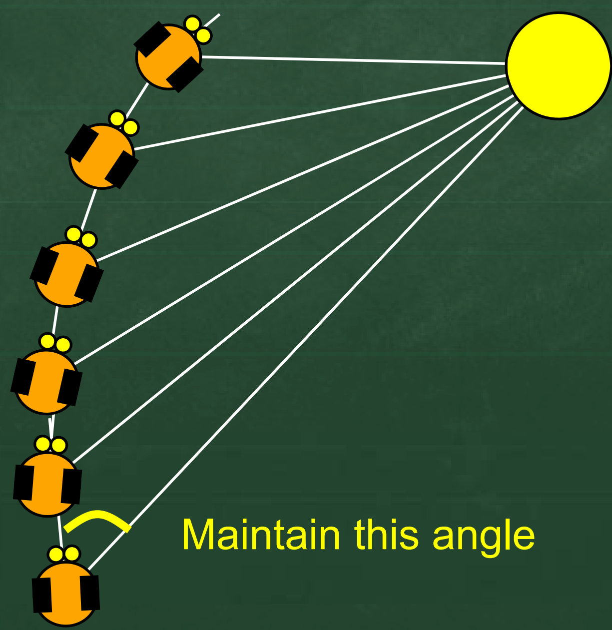


Taxic Behaviors

4 - Light-compass reaction:

Maintaining a fixed angle between the path of motion and the direction of the sensed stimulus

e.g., Light seeking



Adaptive Behaviors

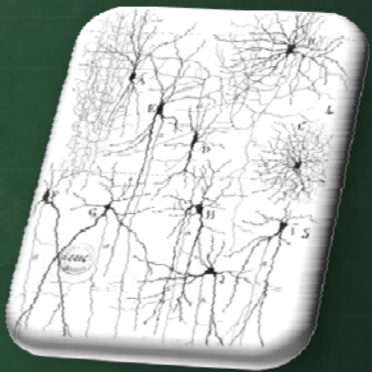
- ***Adaptive Behavior:***

  A behavior that adapts (or adjusts) to the environment in which it acts

- Robot can "adapt" by automatic variation in the strengths of the behavior properties and parameters that we just discussed.
  - Results in more efficient and safe behavior.
  - Degree of adaptability may be:
    - Built-in – *fine-tuned like instincts for certain environments*
    - Learnt – *from past experiences*

Adaptive Behaviors

- Reflexes and taxes depend only on recent events
  - limits overall usefulness towards adaptable behavior
  - most reflexes almost independent of feedback
- In this course:
  - we will not investigate adaptive behaviors
  - we will use instinctive hard-wired behaviors



- Areas of neural networks and evolutionary computing are two strategies for generating behaviors that adapt to their environments.

Behavior Interaction

Behavior Interaction

- Individual reflex and taxic "primitive" behaviors
  - provide basic functionality of robot
  - never stop running, but will not always affect robot
  - should be simple, not handling every situation that arises
- Overall robot behavior depends on interaction of the individual "primitive" behaviors
- Primitive behaviors all run at the same time and compete for control of the robot.

- Must define rules of interaction to decide which behavior has control of the robot at any one time.

Behavior Interaction

- Behaviors interact according to 3 principles:

1. *Inhibitory*
   - incompatible behaviors compete for control of robot

     e.g., obstacle avoidance vs. obstacle mapping/tracing

2. *Cooperative*
   - two behaviors may agree as to how to direct robot

     e.g., seek light source and seek energy source

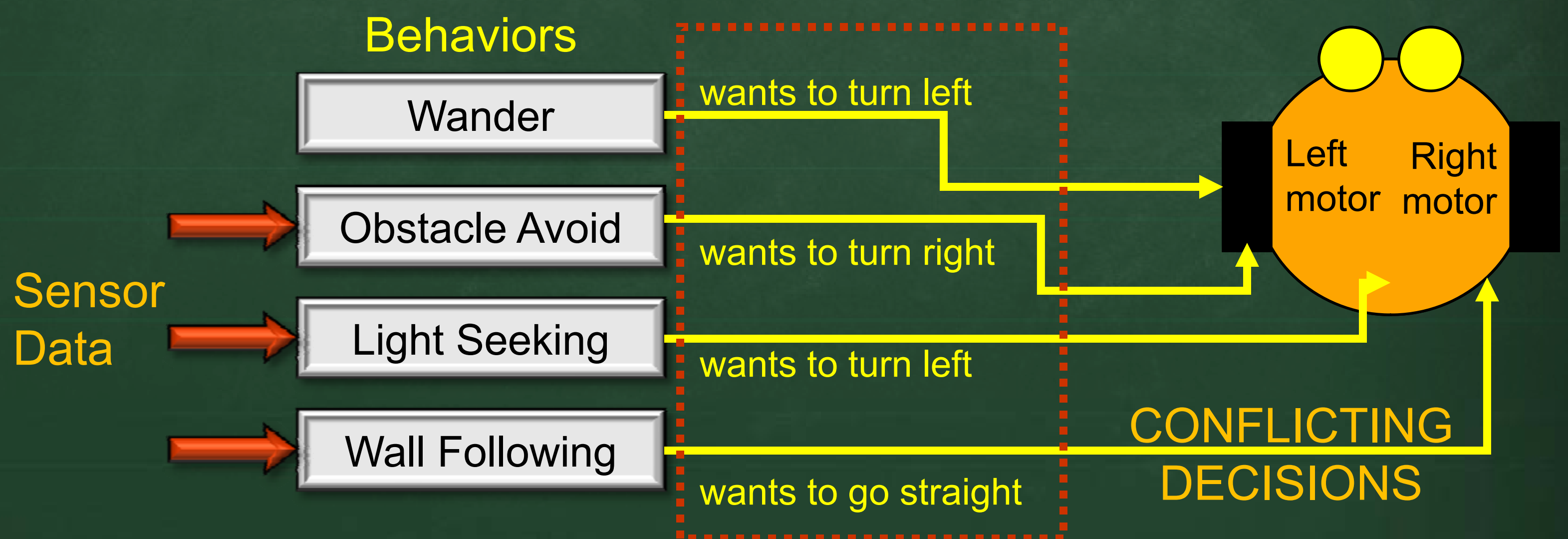
3. *Successive*
   - one behavior may cause another behavior to be exhibited

     e.g., seek base station and then recharge robot

Behavior Interaction

- Two Inhibiting behaviors cannot both control robot
  - must compete for control of robot's actuators
  - only one may be active at any given time

  e.g., Controlling robot direction:



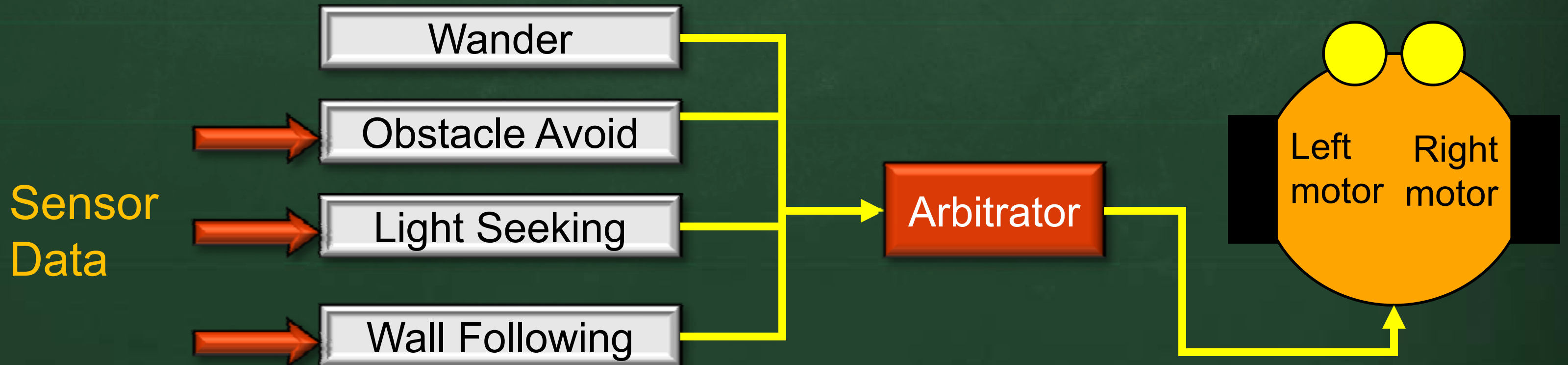
- A behavior arbitration scheme is required.

Behavior Arbitration

- *Arbitration scheme:*

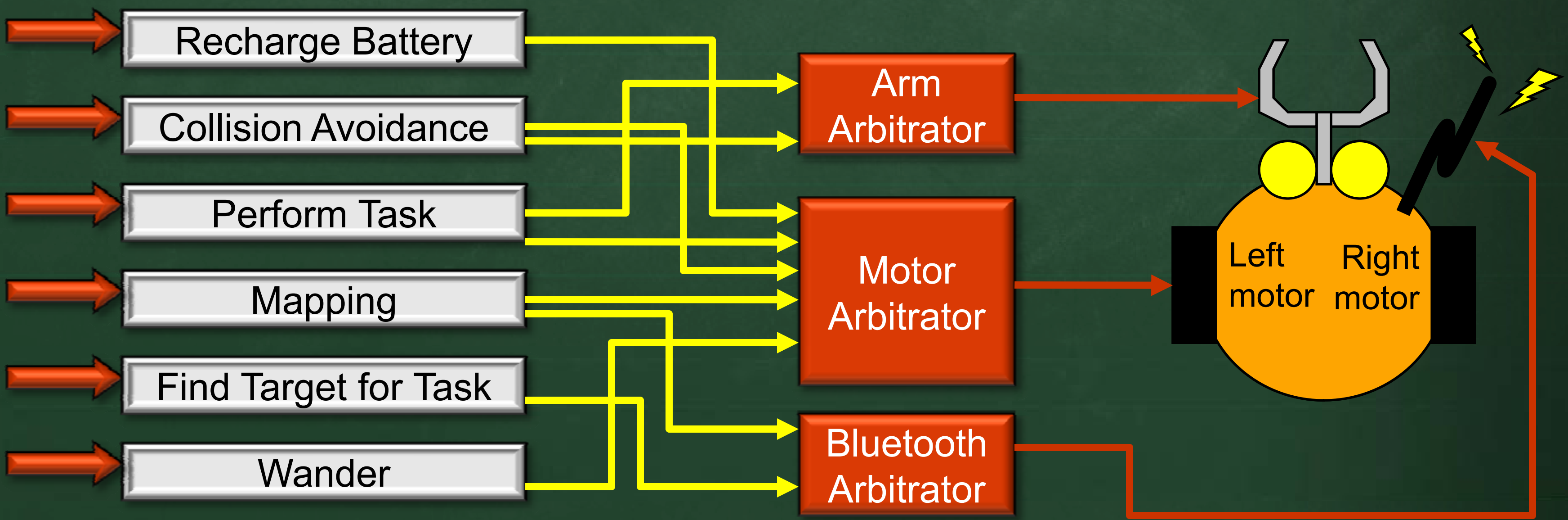
  The manner, priority and timing in which behaviors exhibit influence over a robot's actuators

- Behaviors plug-in to an **arbitrator** which decides which behavior(s) should be actively controlling the robot's actuators at any particular time.



Behavior Arbitration

- Can have multiple arbitrators
  - one per actuator
  - behavior can be connected to many



Behavior Arbitration

- Various ways to resolve conflicts
  - Round robin – each behavior gets equal turn
  - Average – average all behaviors to choose action
  - Vote – majority of desired actions wins
- These conflict resolution strategies do not account for urgent, time-critical or opportunistic behaviors
  - e.g., collision avoidance, low battery, etc..

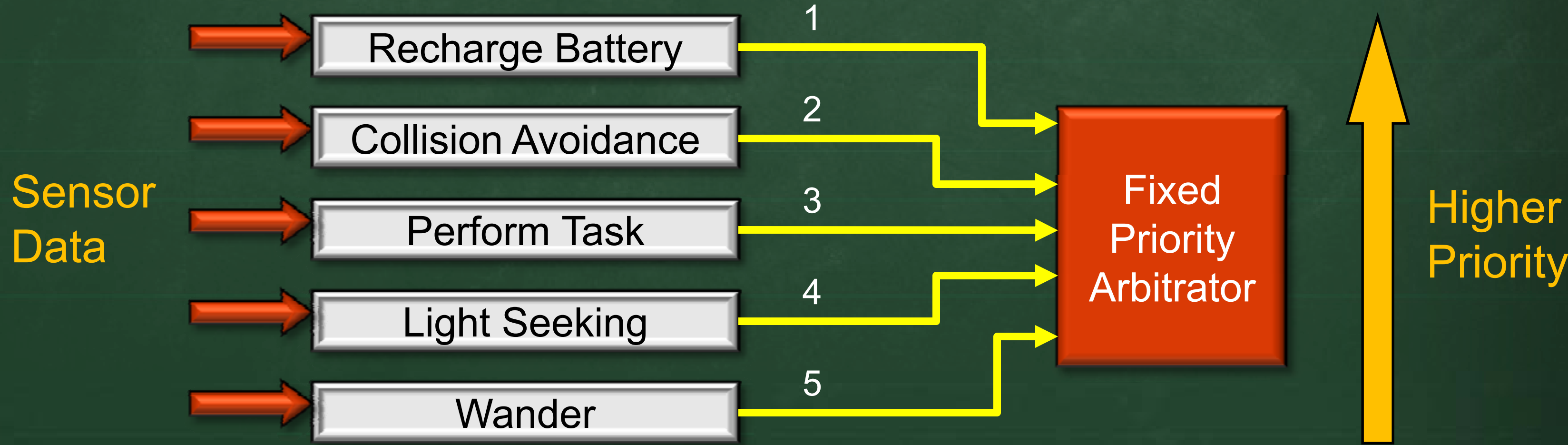
- Need a way to prioritize behaviors:
  - High priority behaviors greater influence on robot's action
  - Low priority behaviors ignored unless no high-priority available

Behavior Arbitration

- There are two main arbitration schemes:
  1. Fixed Priority
  2. Variable Priority
- Each differs in the way it prioritizes the behaviors
- Based on the ***subsumption architecture*** Rodney Brooks (1986).
  - Low-priority behaviors implemented first, higher ones added later
  - Higher level behaviors contain (i.e., *subsume*) or inhibit (i.e., *disable*) lower level ones

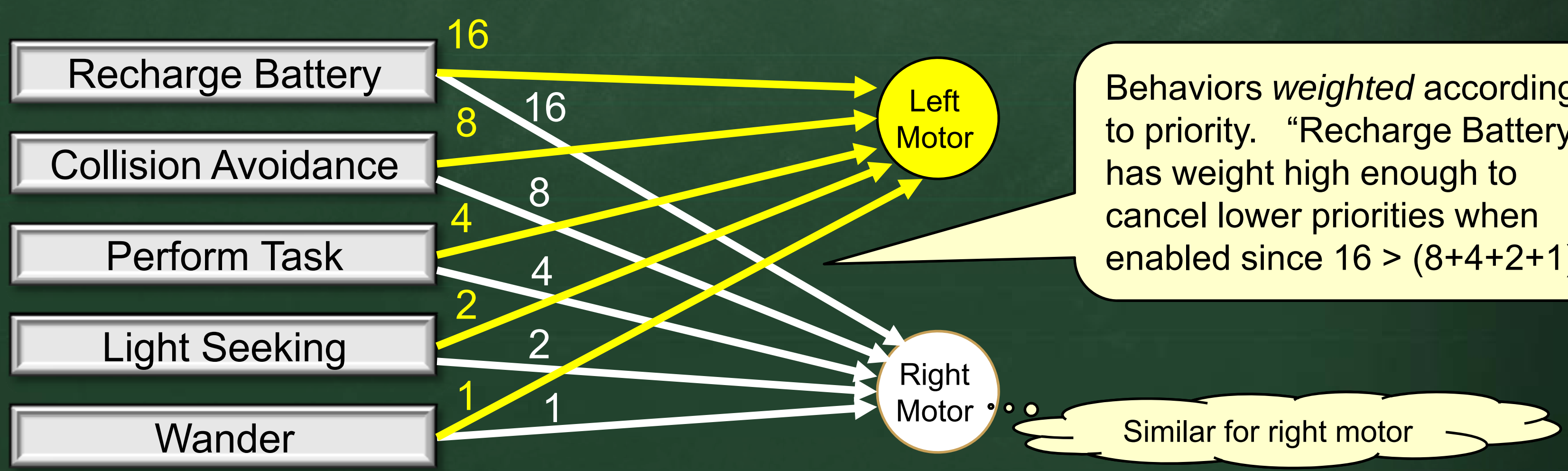
Behavior Arbitration

1. Fixed Priority Arbitration
   - Priorities decided in advance (hard-wired)
   - Priorities should be unique



Behavior Arbitration

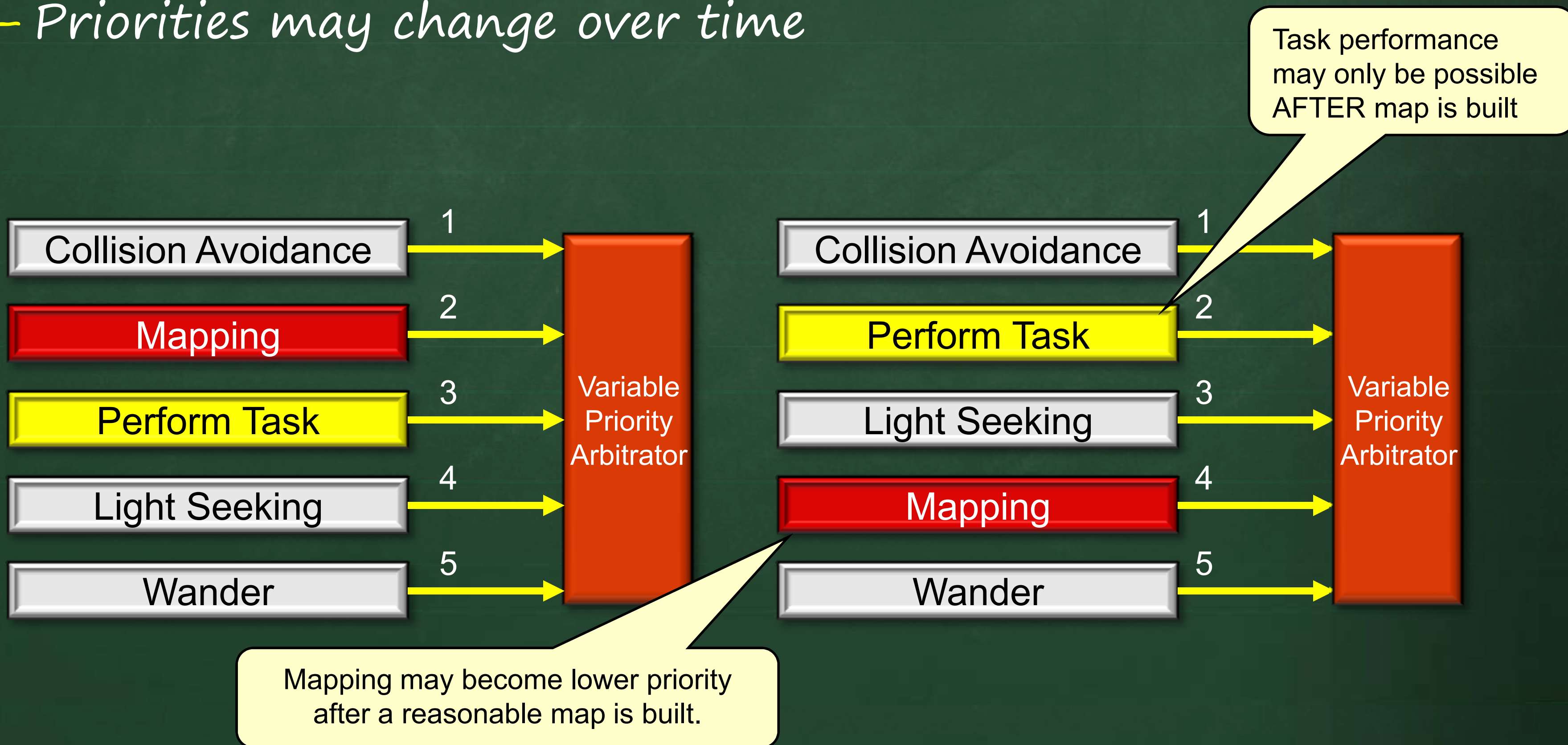
- How do you choose priorities ? Rules of thumb…
  - Behaviors critical to safe operation should be high
  - Task-oriented priorities should be medium
  - "Free-time" tasks (e.g., wander, map) may be low
- Here is one way of doing this using "weights":



Behavior Arbitration

2. Variable Priority Arbitration

- Priorities unique and decided in advance
- Priorities may change over time



Behavior Arbitration

- Various questions arise:
  - What determines new priorities order ?
    - depends on environment & learned information (e.g., maps)
  - How do we make sure 2 priorities are never the same ?
    - perhaps simply swapping with other behaviors
  - How often should we re-order ?
    - can depend on environmental structure changes or task-related changes
- Allows for more time-flexible behavior taking advantage of opportunism.
- Can be more complex to debug

Behavior Arbitration

- In generalized subsumption architecture, behaviors may also "inhibit" or "disable" others.

- For example, when exhibiting a mapping behavior, robot may wish to ignore other behaviors.

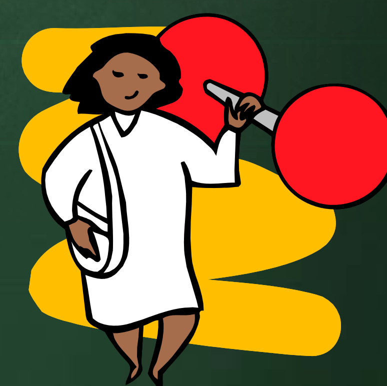
  

  – May need to disable collision avoidance or escape behaviors in order to get close enough to obstacles for mapping.

  – May not disable behavior but simply allow thresholds and tolerances to be adjusted.

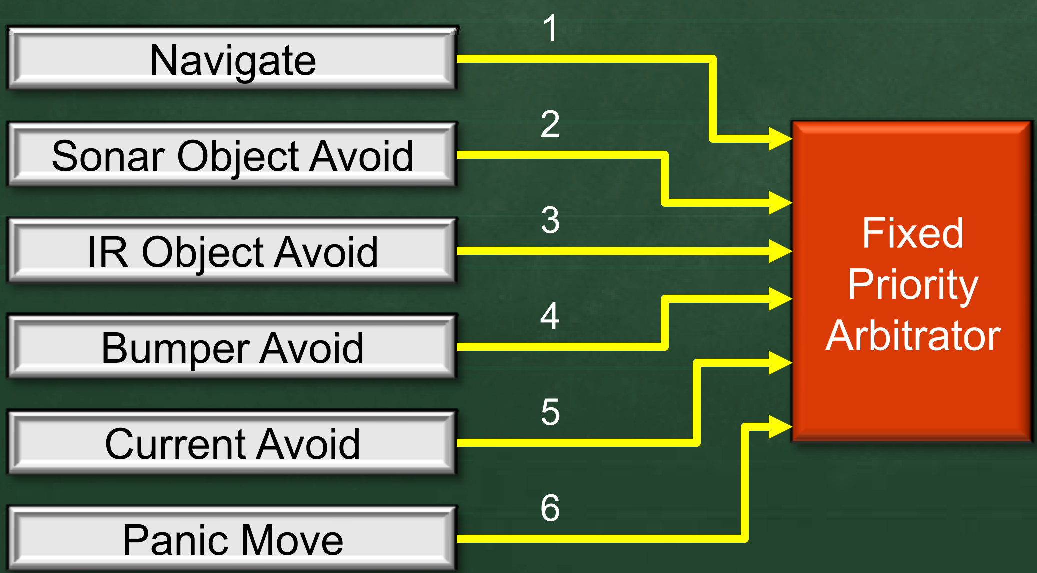
Robustness

- Behavior-based approach is more robust than top-down approach.
  - Behaviors typically rely on multiple low-level sensors rather than few high-level ones.
  - Some behaviors may fail, others "kick-in"
  - Robot performance degrades gracefully when behaviors fail, but still performs
  - Don't have to think of all scenarios



Robustness

- Consider the following hierarchy of obstacle avoidance where system performance degrades gracefully. For example,



Robot is navigating
- Sonar detects obstacle from far off, starts steering away

What if sonar fails to detect ?
- IR detects obstacle when closer (less time to turn, more awkward)

What if IR also fails ?
- Bumper switches hit, must backup

What if robot hits where no bumper ?
- Robot hits, motors draw more current
- Must backup

What if wheels slip, robot stuck ?
- Can detect lack of motion
- Try “panic” movement to get unstuck

Programming Behaviors

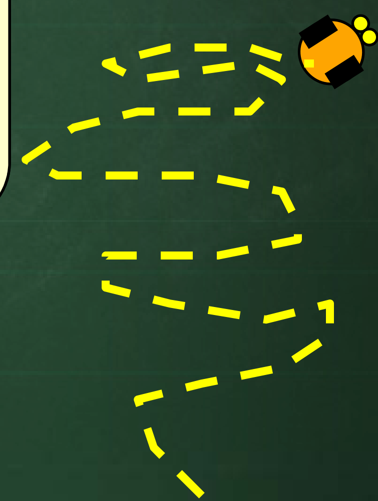
Wandering

- How do we program behaviors and arbitrators ?
- Simplest is a *Wandering* behavior.

```
int wanderLeft = 0;
int wanderRight = 0;
Random ranGen = new Random();

if (ranGen.next() % 5 == 0)
   if (ranGen.next() % 2 == 0)
      wanderLeft = 1;
   else
      wanderRight = 1;
```

Decide to make a random turn 1/5 = 20% of the time. Tweak this to adjust amount of wandering

Decide to turn left or right. Turn left 1/2 = 50% of the time.



- Can also incorporate degree (amount) of turning

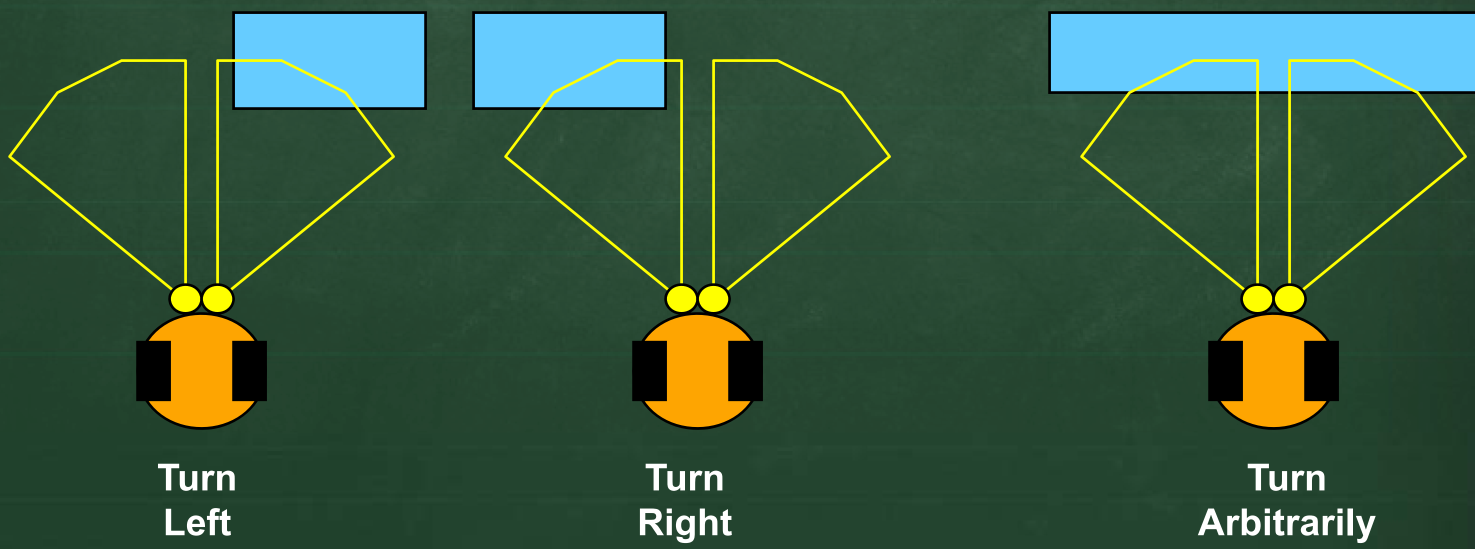
Wandering

- *Connect multiple behavior requests to the arbitrator, using their priorities:*

```
int leftTotal = (wanderLeft * wanderPriority) +
                (avoidLeft * avoidPriority) +
                (lightLeft * lightPriority) + ...;
int rightTotal = (wanderRight * wanderPriority) +
                (avoidRight * avoidPriority) +
                (lightRight * lightPriority) + ...;

if (leftTotal > rightTotal)
   // Turn on LEFT motor backwards & RIGHT motor forwards
else if (leftTotal < rightTotal)
   // Turn on LEFT motor forwards & RIGHT motor backwards
else
   // Turn on LEFT & RIGHT motors forward
```

Collision Avoidance

- What about Collision Avoidance behavior ?
- Simple using 2 "boolean" proximity sensors, although can use more and/or various types.



Collision Avoidance

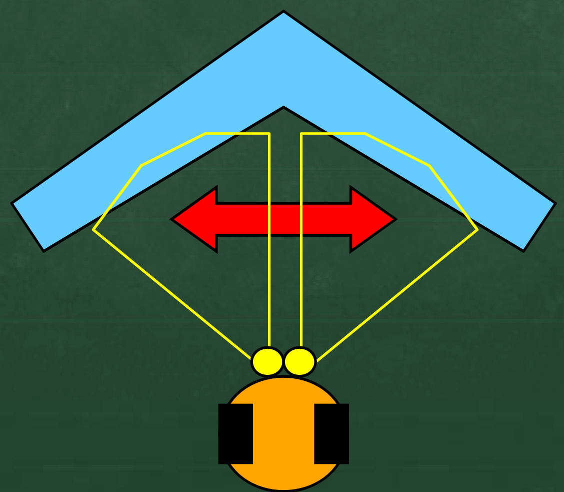
- Similar to wandering, decide to turn left or right:

```
int avoidLeft = 0;
int avoidRight = 0;
Random ranGen = new Random();

if (leftProxSensor.getValue() > 0)
   if (rightProxSensor.getValue() > 0)
      if (ranGen.next() % 2 == 0)
         avoidLeft = 1;
      else
         avoidRight = 1;
   else
      avoidRight = 1;
else
   if (rightProxSensor.getValue() > 0)
      avoidLeft = 1;
```

Decide to make a random turn if obstacle straight ahead. Other approaches can be taken here…

Collision Avoidance

- May end up in corner, oscillating back and forth.
- Random turns "may" get robot unstuck, but clumsy
- Perhaps ensure 120° to 180° turn or until one sensor does not read collision anymore.



Can get stuck turning left, then right, then left, then right etc…

Collision Avoidance

- Can include counter to ensure turn is complete:

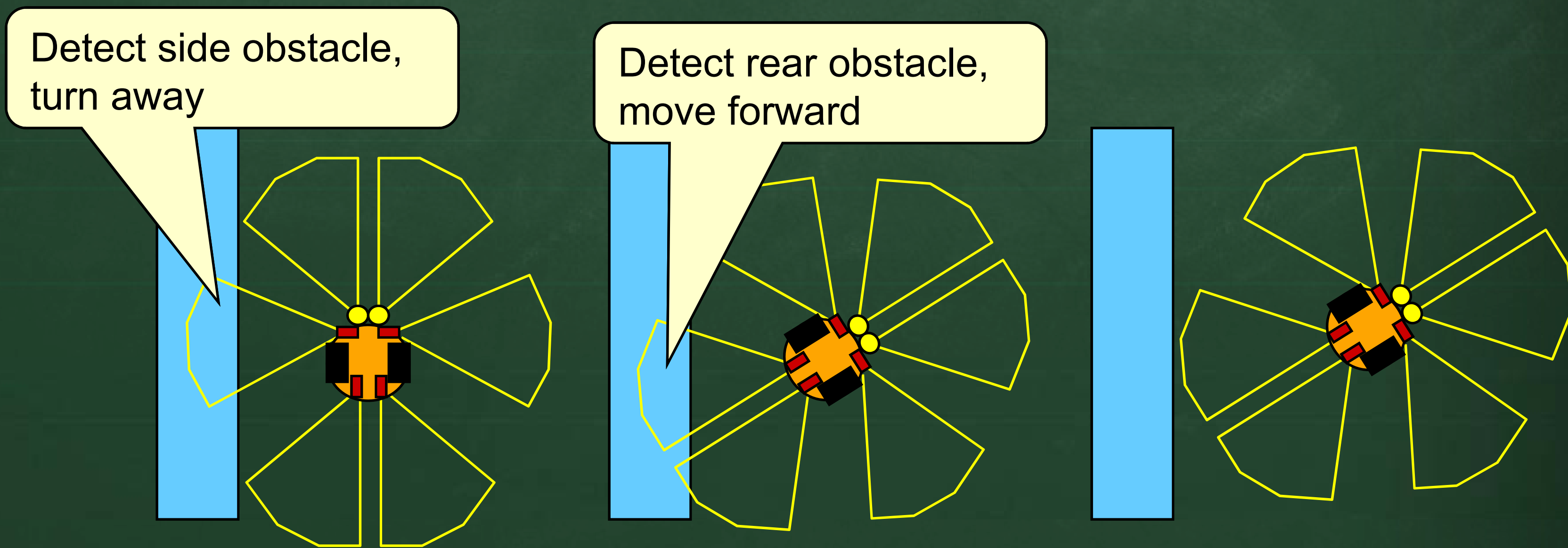
```
if (turnCount > 0) turnCount--;
else {
   avoidLeft = avoidRight = 0;
   if (leftProxSensor.getValue() > 0) {
      if (rightProxSensor.getValue() > 0) {
         if (ranGen.next() % 2 == 0)
            avoidLeft = 1;
         else avoidRight = 1;
         turnCount = 12;
      }
      else avoidRight = 1;
   }
   else if (rightProxSensor.getValue() > 0)
      avoidLeft = 1;
}
```

If already turning (since last time this code was called) don't make any new turn decisions, just keep doing the turn as decided upon before.

Turn for 12 "units", which is 120 degrees if each unit makes 10 degrees.

Escape

- An extension to obstacle avoidance is escaping (i.e., keeping away) from obstacles.
  - collision avoidance takes care of turning away from obstacles in the robot's path
  - if detecting obstacles on the side, can also turn away.



Escape

- Code easily written:

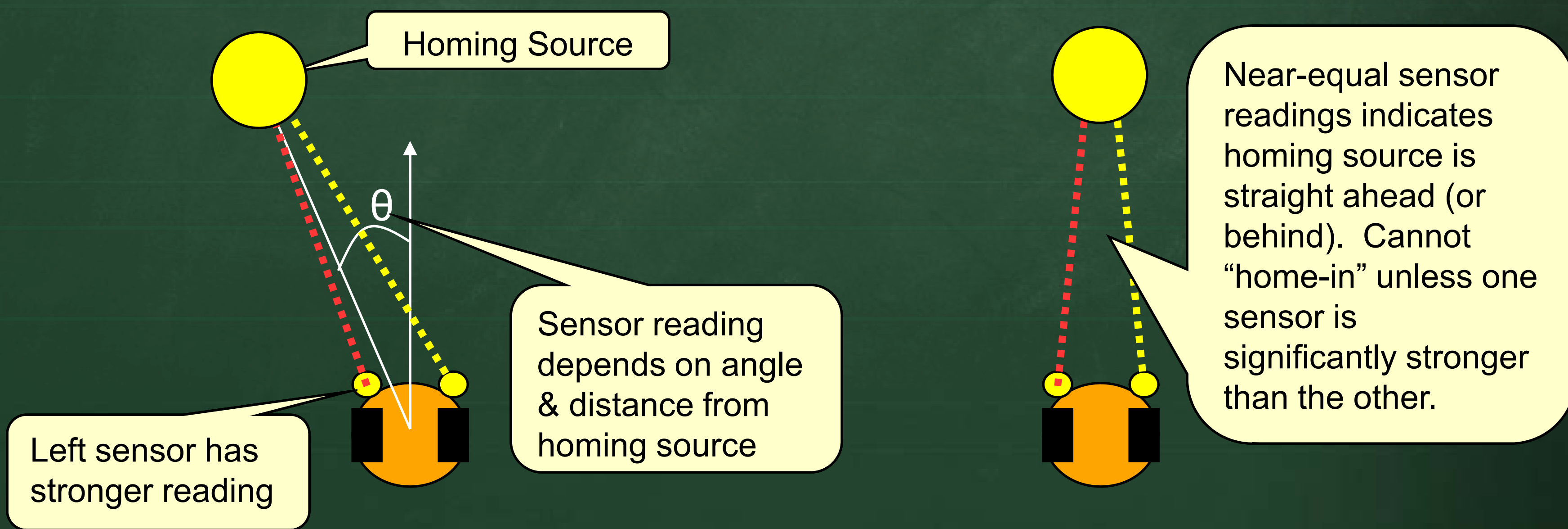
```
int escapeLeft = escapeRight = 0;

if (backProxSensor.getValue() == 0) {

    if ((leftSideProxSensor.getValue() > 0) &&
        (rightRideProxSensor.getValue() == 0))
        escapeRight = 1;

    if ((leftSideProxSensor.getValue() == 0) &&
        (rightRideProxSensor.getValue() > 0))
        escapeLeft = 1;
}
```

If detect rear obstacle, move forward

Turn only if one of the side sensors detects. If both detect, simply move forward.

Homing

- How do we program code for the robot to "home" towards an object (e.g., light seeking) ?
- Need 2 sensors whose readings increase when pointed towards homing source (recall tropotaxis)



Homing

- Simply compute difference between incoming sensor readings to determine direction.
- Choose threshold according to sensor sensitivity and range abilities.

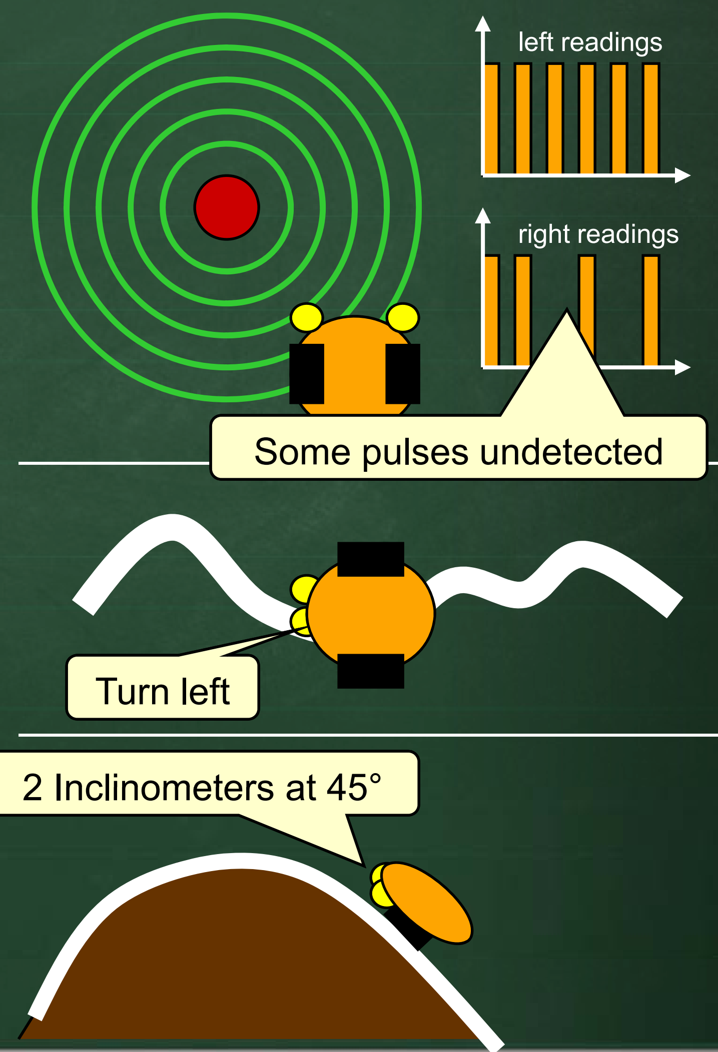
```
int lightLeft = 0;
int lightRight = 0;
int threshold = 2;
int diff = leftSensor.getValue() - rightSensor.getValue();

if (diff < (-1*threshold))
   lightRight = 1;
if (diff > threshold)
   lightLeft = 1;
```

Threshold depends on sensor reading range and sensitivity.

Only turn if difference is significant (i.e., above threshold)

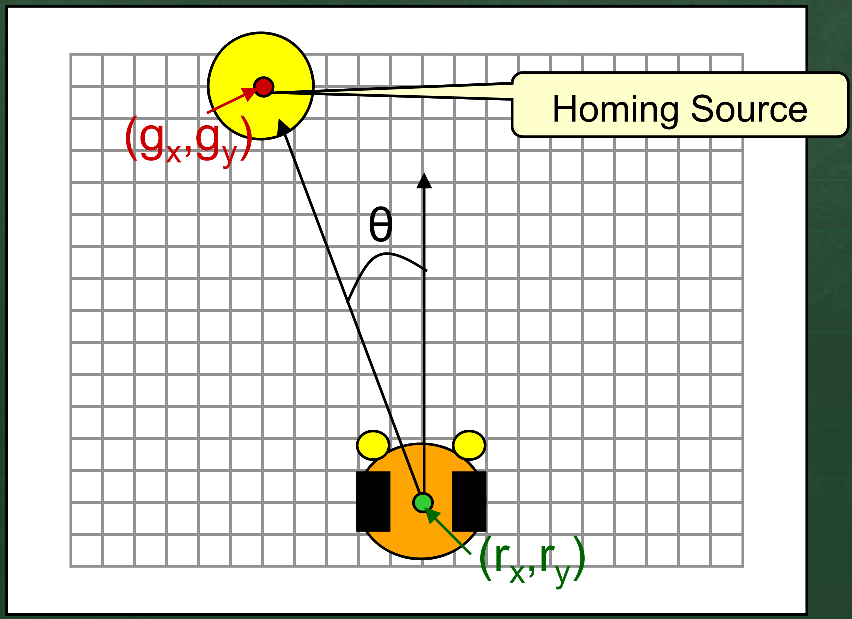
Homing

- Other forms of homing:
  - Beacon Following
    - Beacon emits pulsed signal which is more reliably detected by closer sensor
  - Line Following
    - Photodetectors read stronger on white, can detect when a sensor leaves black line
  - Hill Climbing
    - Climb hill by minimizing roll while keeping pitch positive using inclinometers



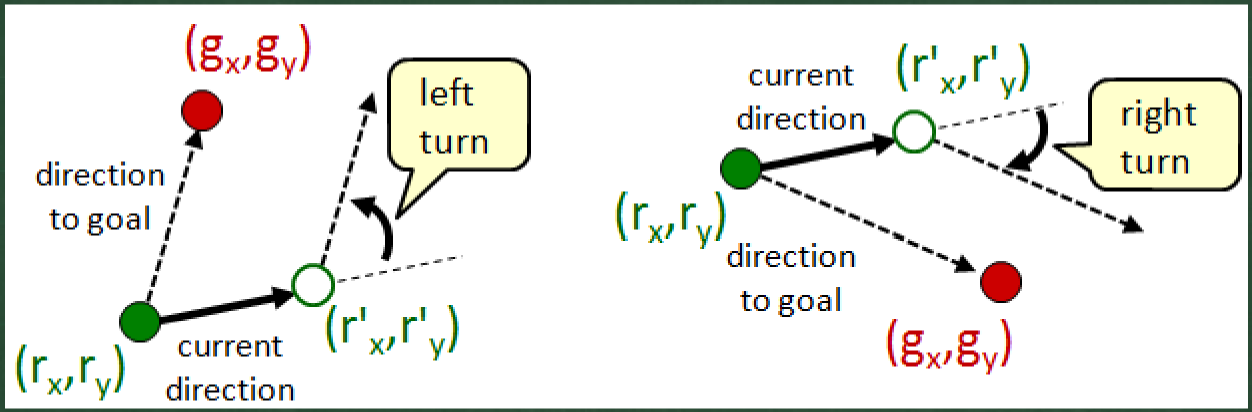
GPS Homing

- Can use a Global Positioning System (GPS) to determine desired homing direction
  - Gives (r_x,r_y) robot position, (g_x,g_y) goal position
  - Desired direction to travel can be computed using simple trigonometry



GPS Homing

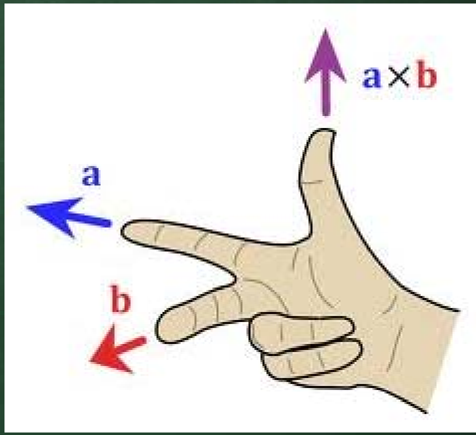
- Need to look at the where robot is heading and decide whether or not to turn right or left to goal:



- Examine type of turn from $(r_x, r_y) \rightarrow (r'_x, r'_y) \rightarrow (g_x, g_y)$.
  - (r'_x, r'_y) is any point forward in robot's current direction
  - If it is a straight line, the robot will need to either move straight ahead, or straight backwards (depending on where the goal location is).

GPS Homing

- Make use of the *cross product* of vector from $(r_x, r_y) \rightarrow (r'_x, r'_y)$ and vector from $(r_x, r_y) \rightarrow (g_x, g_y)$
- Cross product **a X b** is a vector perpendicular to the plane containing vectors **a** and **b**, computed as follows:



$$(r'_x - r_x)(g_y - r_y) - (r'_y - r_y)(g_x - r_x)$$

where:

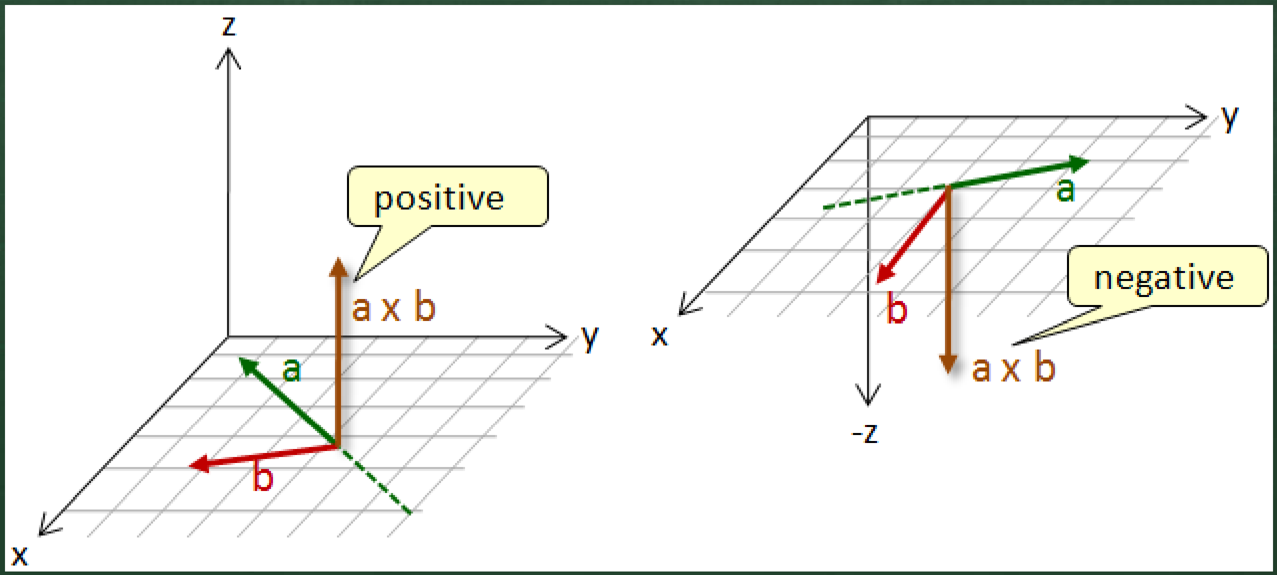
$r'_x = r_x + d*\cos(r_\theta)$

$r'_y = r_y + d*\sin(r_\theta)$

d = any non-zero positive value, r_θ is the robot's direction

GPS Homing

- The cross product will either be:
  - *positive* = a left turn
  - *negative* = a right turn
  - *zero* = no turn (i.e., vectors form 180° angle)



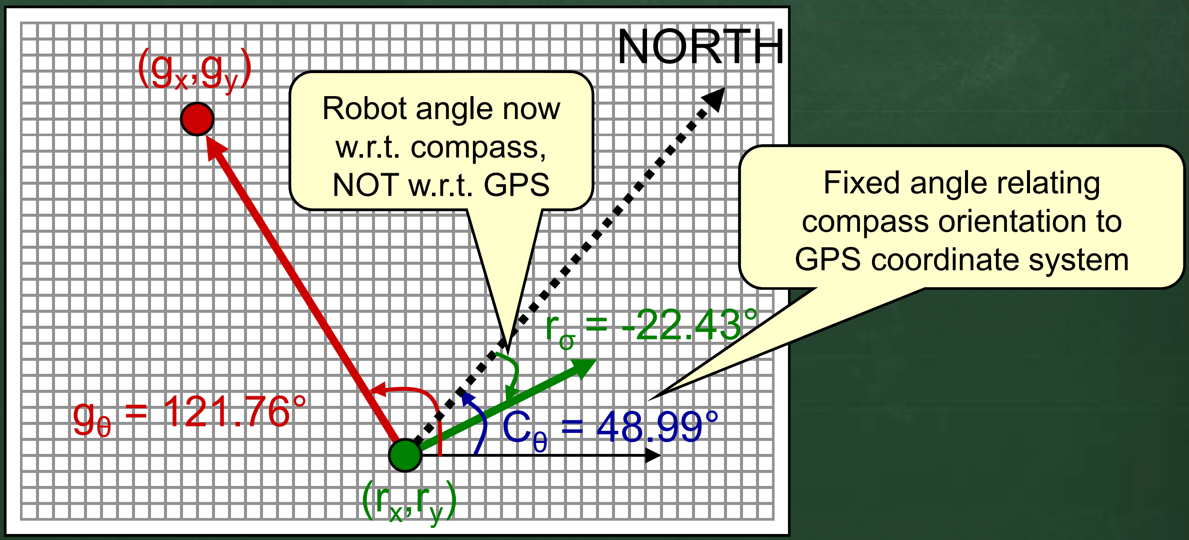
GPS Homing

- If robot's orientation r_θ is also available from GPS, then turn decision for homing is easy.
  - Robot can also remember its previous location and compute r_θ by comparing with current location.
- Easy to write the code now:

```
int goalLeft = 0;
int goalRight = 0;
int d = 10;         // any constant
Point r = GPS.getRobotLocation();
Point rTheta = GPS.getRobotDirection();
Point g = GPS.getGoalLocation();
Point r2 = new Point(r.x + d*cos(rTheta), r.y + d*sin(rTheta));

int turn = (r2.x - r.x)(g.y - r.y) - (r2.y - r.y)(g.x - r.x);
if (turn > 0) goalLeft = 1;
if (turn < 0) goalRight = 1;
```

GPS Homing

- What if robot's direction not provided by GPS ?
  - Must rely on other means (e.g., compass)
  - May use different coordinate system
- Must determine compass reference C_θ (e.g., North) in GPS coordinate system beforehand:



GPS Homing

- Similar code as before:

```
static float C_THETA = 49.0f;

int goalLeft = 0;
int goalRight = 0;
int d = 10;         // any constant
Point r = GPS.getRobotLocation();
Point rTheta = compassSensor().getValue();
Point g = GPS.getGoalLocation();
Point r2 = new Point(r.x + d*cos(rTheta+C_THETA), r.y + d*sin(rTheta+C_THETA));

int turn = (r2.x - r.x)(g.y - r.y) - (r2.y - r.y)(g.x - r.x);
if (turn > 0) goalLeft = 1;
if (turn < 0) goalRight = 1;
```

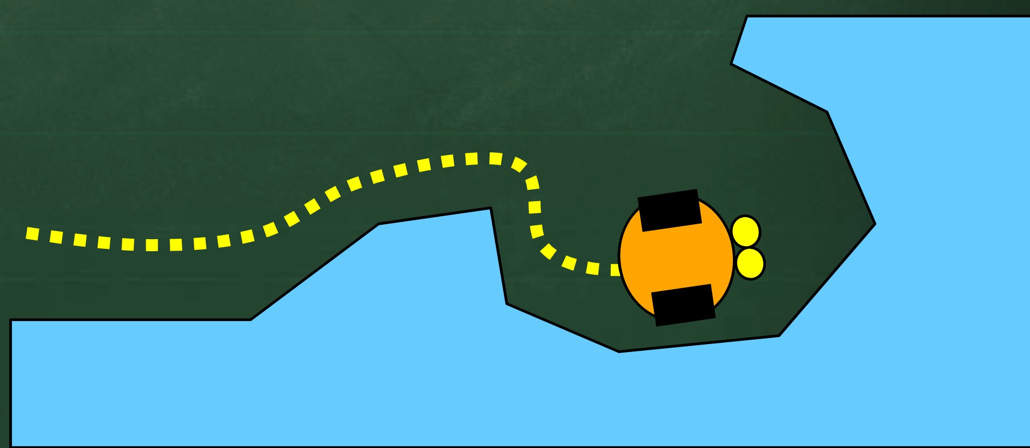
Hard-coded or perhaps calibrated upon startup.

Now use compass to get direction.

Transformation to new coordinate system.

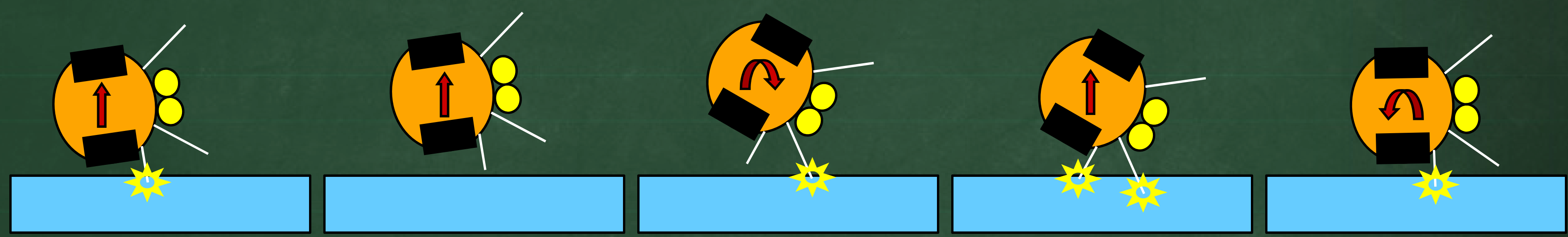
Wall Following

- Wall following behavior is useful for mapping, navigation, seeking wall outlets, performing cleaning tasks etc…
- Strategy varies depending on types of sensors.
- Robot usually follows wall by keeping itself aligned to the wall on its left or right side



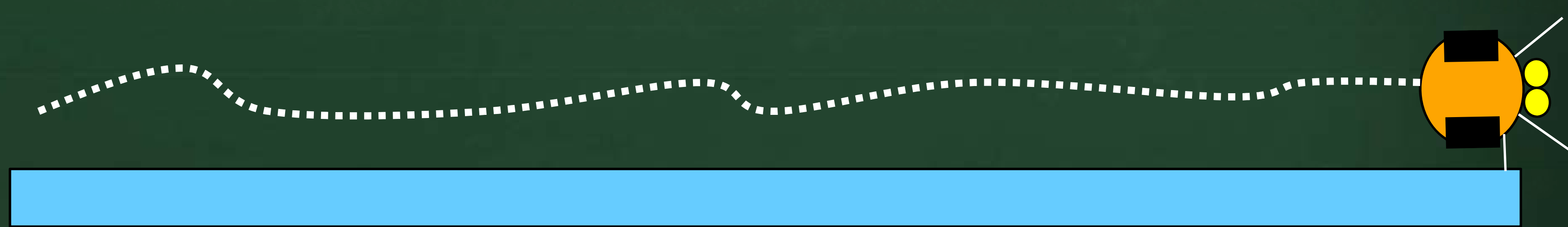
Wall Following

- Key is to maintain alignment along edge
- Can be done with 2 whisker sensors:



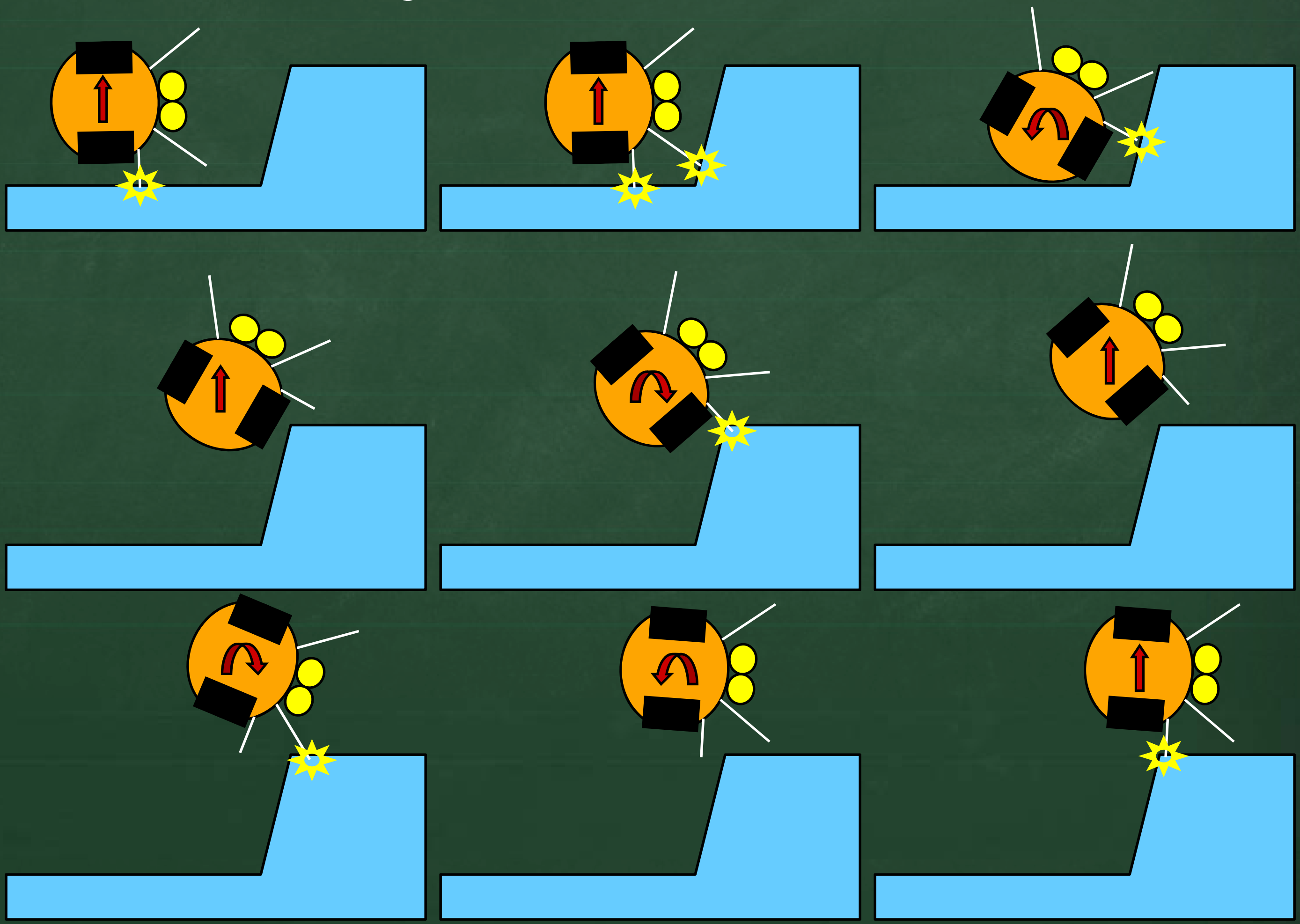
Move fwd while side contact | Lost contact must re-align | Turn right until front contact | Move fwd until side contact | Turn left until no front contact

- Robot moves in a bumpy motion along wall:



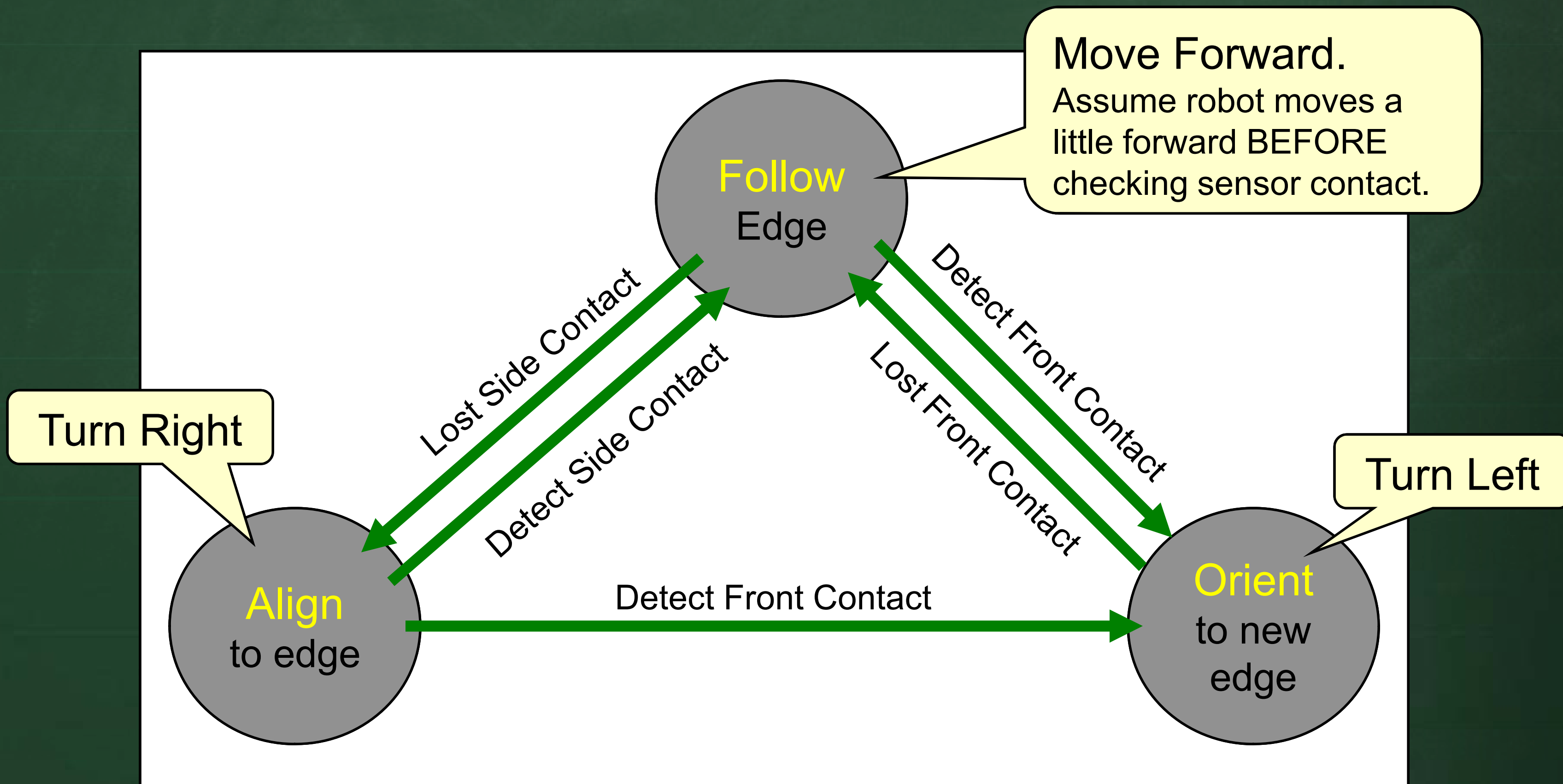
Wall Following

- Additional handling of concave/convex corners:



Wall Following

- Can follow state diagram as follows:
  - assumes right side edge following



Wall Following

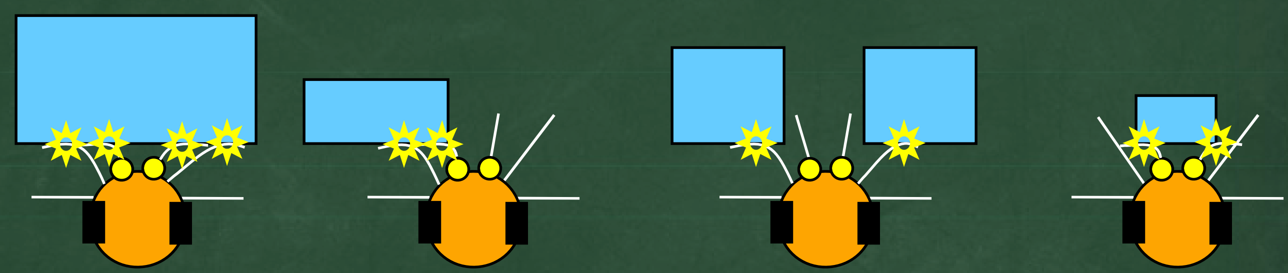
- Writing code is now easy:

```
int currentMode = FOLLOW;

int edgeFollowLeft = 0;
int edgeFollowRight = 0;
boolean detectFront = frontWhiskerSensor.getValue() > 0;
boolean detectSide = rightWhiskerSensor.getValue() > 0;
switch (currentMode) {
  case FOLLOW:
     if (!detectSide) currentMode = ALIGN;
     if (detectFront) currentMode = ORIENT; break;
  case ALIGN:
     edgeFollowRight – 1;
     if (detectSide) currentMode = FOLLOW;
     if (detectFront) currentMode = ORIENT; break;
  case ORIENT:
     edgeFollowLeft = 1;
     if (!detectFront) currentMode = FOLLOW;
}
```

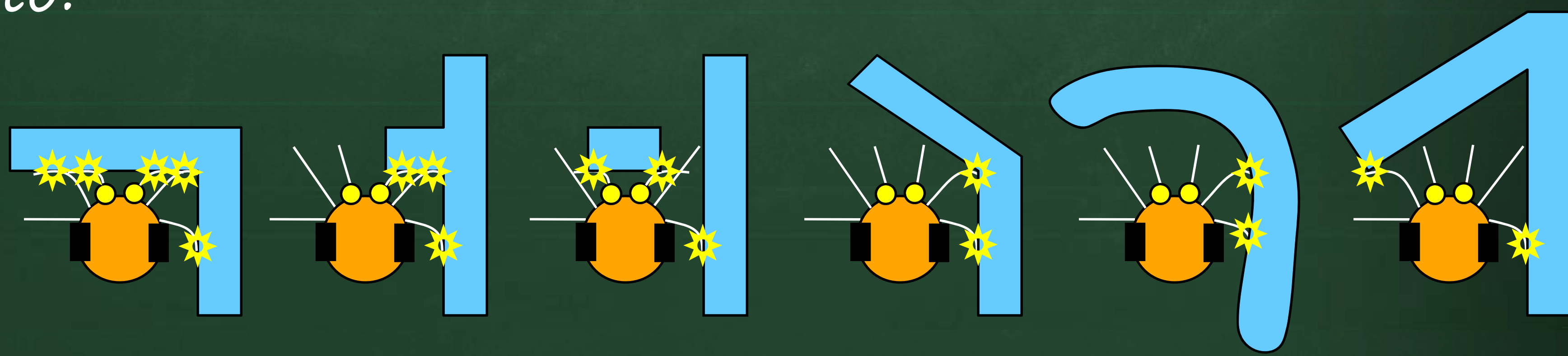
Set this once to initialize the starting of edge following. May be **FOLLOW**, **ALIGN** or **ORIENT**

Wall Following

- More whiskers allow for more sophisticated object shape detection:

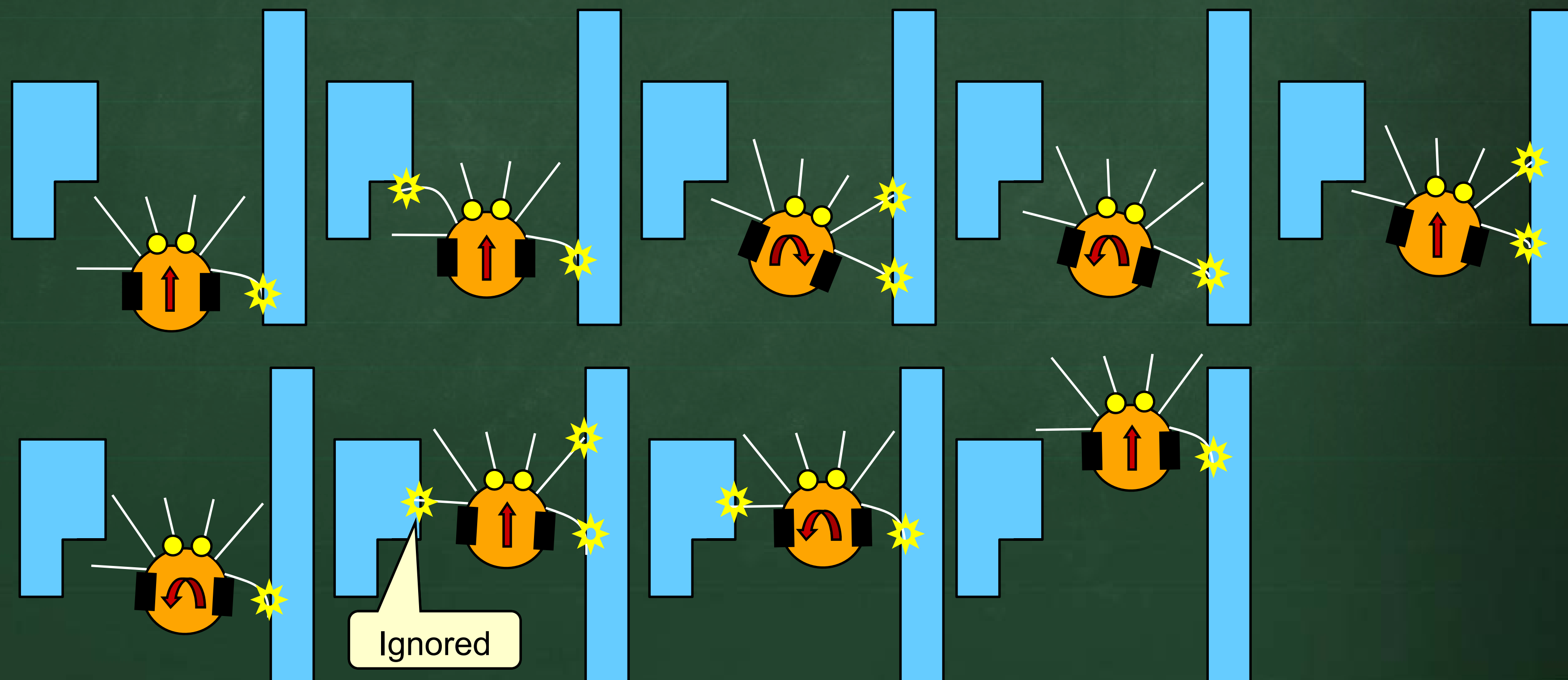


- Various wall shapes considered as new edges to orient to:



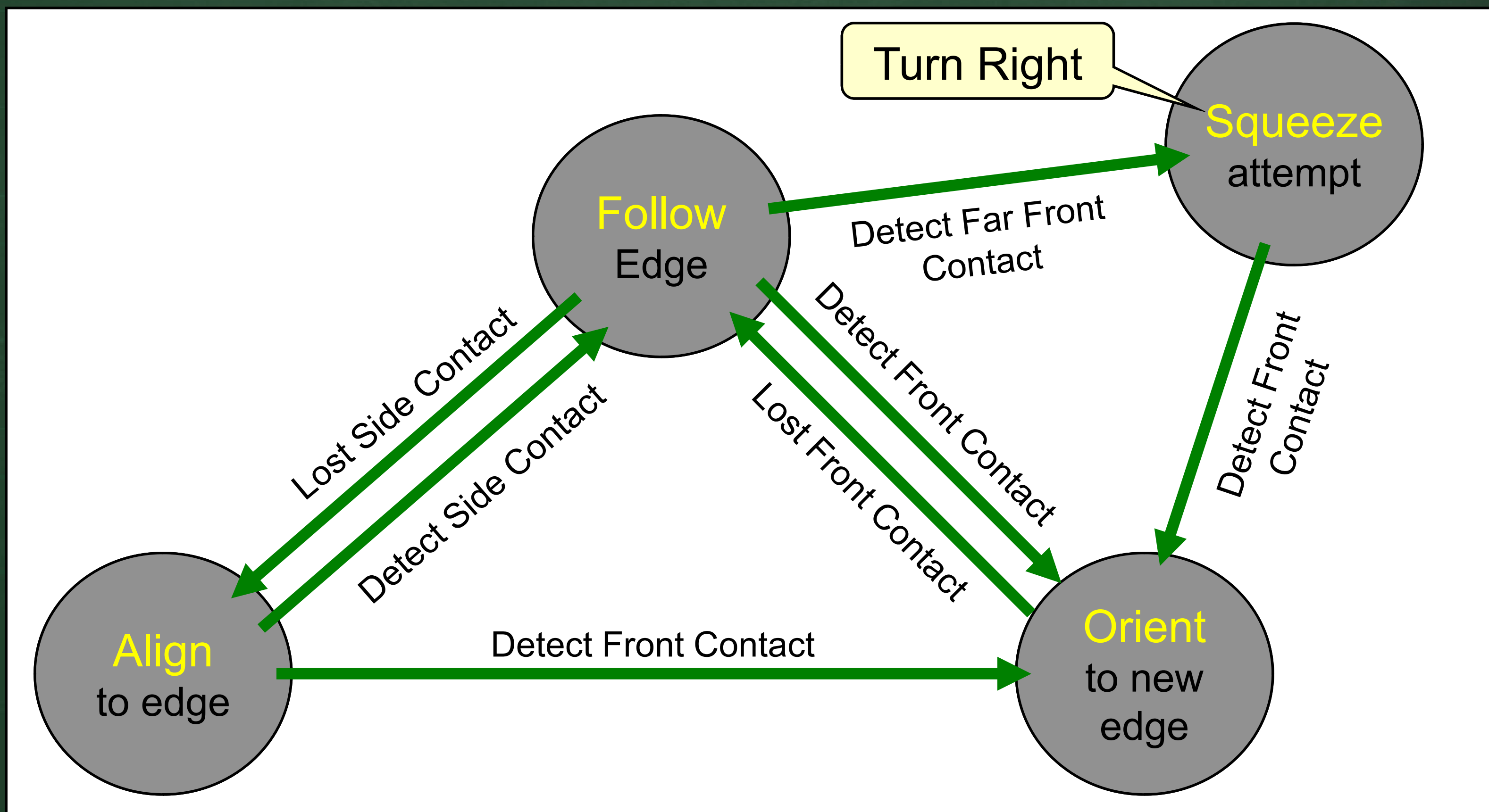
Wall Following

- Can allow squeezing into tight spaces
  - Need careful choice of whisker lengths on front & side



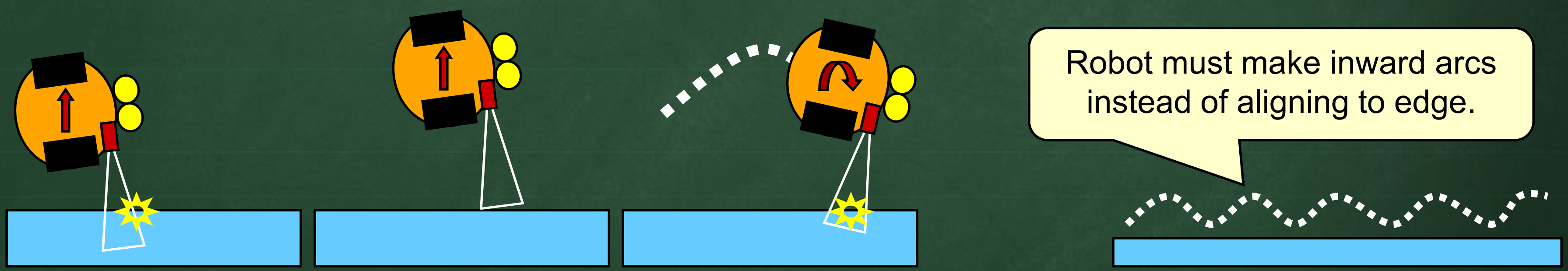
Wall Following

- Need to change state diagram to accomplish this:



Wall Following

- Can use proximity sensors instead of whiskers
  - Works the same but can set threshold easily (like setting whisker length)
- Can use just one sensor, requires arcing motion:



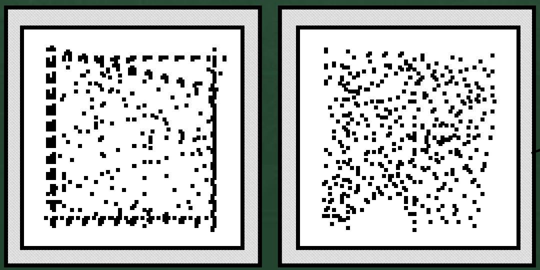
- Can use range sensors as well to allow variable distances from edges during following.

Emergent Behaviors

Emergent Behaviors

- When combined, behaviors produce additional *emergent* behavior.
  - Emergent behavior is higher level
  - Should have degree of randomness
- Consider some simulation results combining *wandering* and *escape* behaviors:

Robot's traced path shows much time spent rubbing up against boundaries when only wandering and collision avoidance behaviors are used.

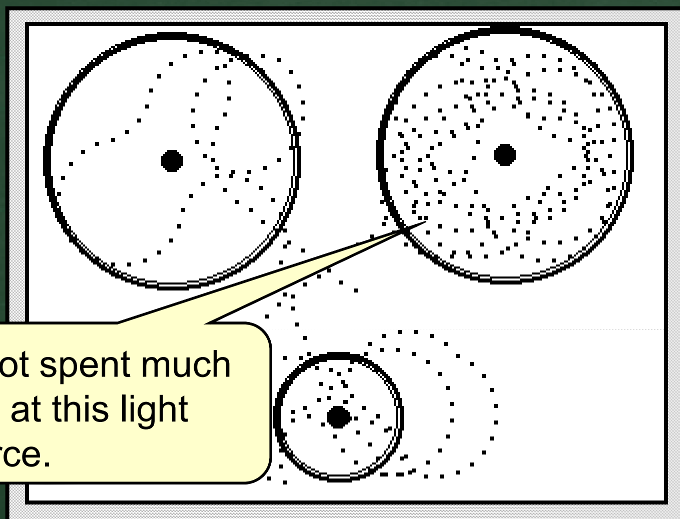


Wander   Wander + Escape

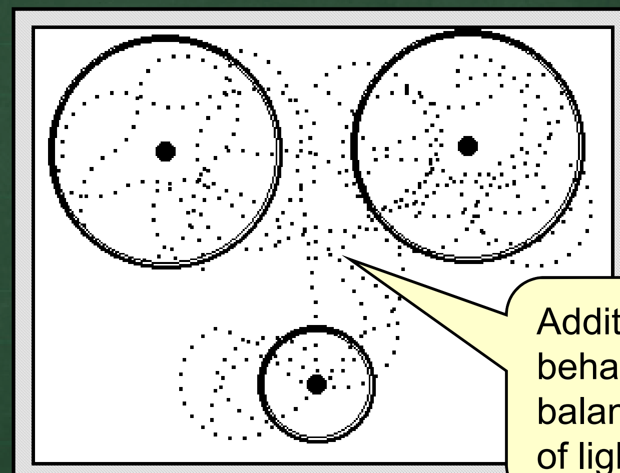
The additional escape behavior keeps robot in center of environment more often.

Emergent Behaviors

- Combining wandering with light seeking also helps alleviate the telotaxis problem:



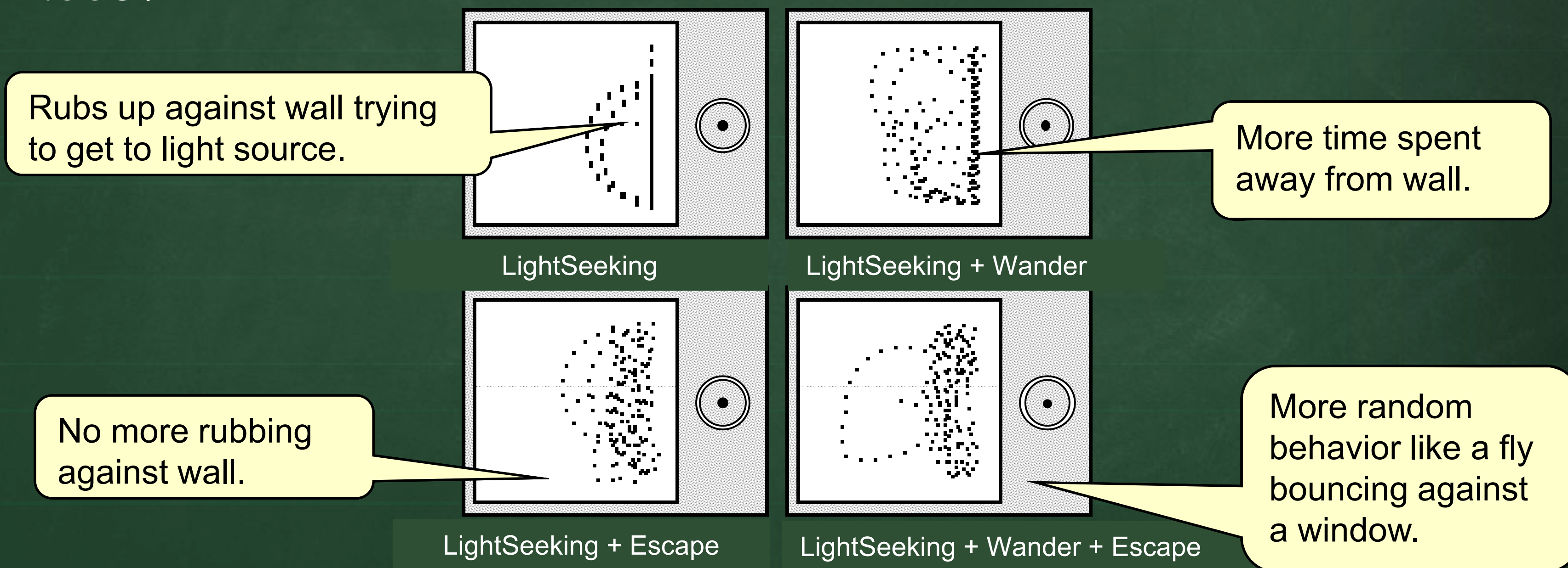
Light Seeking Only



Light Seeking + Wander

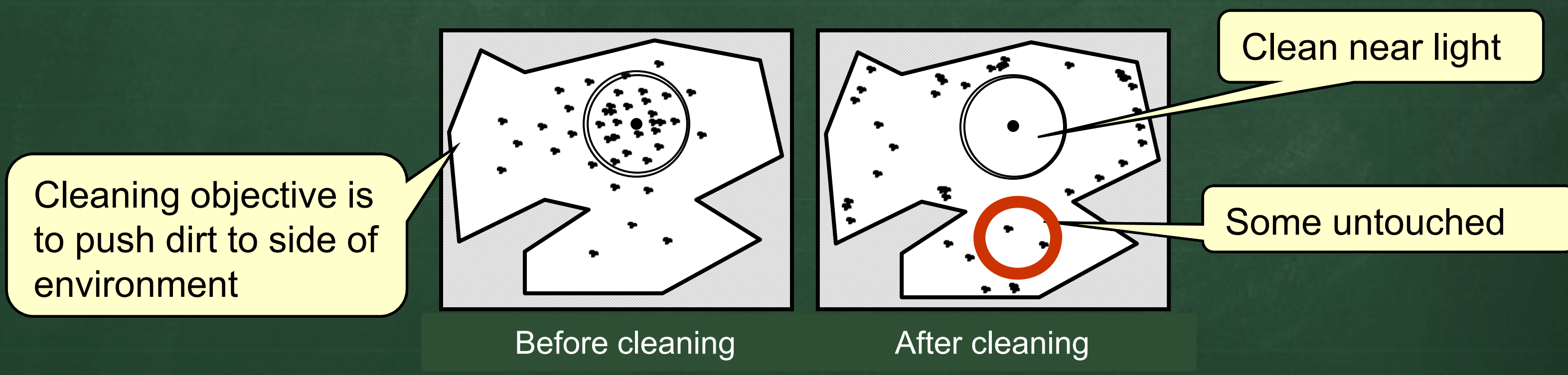
Emergent Behaviors

- Combining wandering, light seeking and escape behaviors results in a more "life-like" behavior pattern



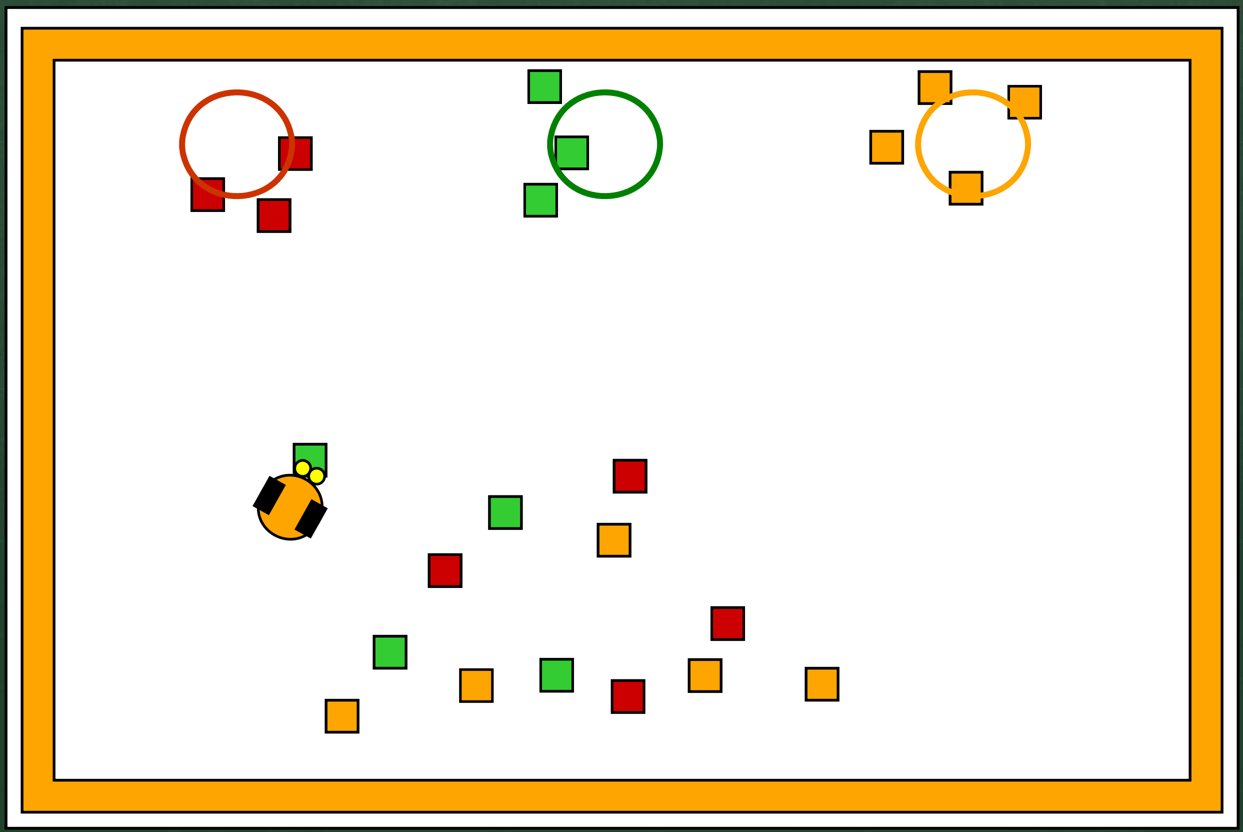
Emergent Behaviors

- Can use certain combinations to make robot more efficient:
  - *e.g.*, random cleaning task can be improved if combined with light seeking (*e.g.*, cleaning rooms with the lights on in an office building)



Emergent Behaviors

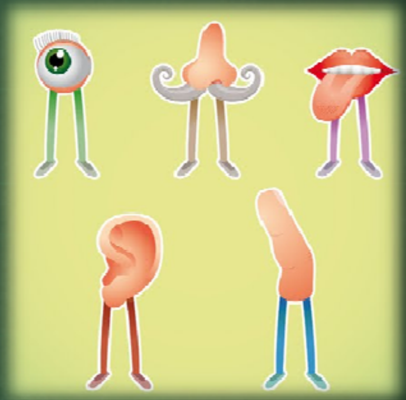
- With variety of sensors, may develop more sophisticated behavior (*e.g.*, block sorting)



Emergent Behaviors

- More interesting behavior requires rich sensory input as well as environmental knowledge

- We will discuss more later about:
  - various sensors
  - mapping the environment
  - navigation



Learning Behaviors

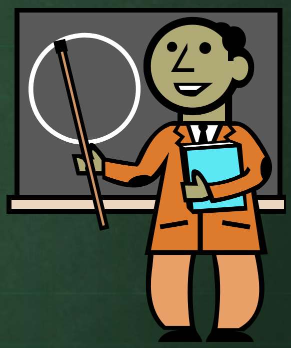
Learning Behaviors

- Until now, we have assumed that individual robot behaviors were well-defined and hardwired.
- Can also "***learn***" how to perform simple behaviors.
  - robot improves over time and is eventually able to exhibit the behavior (similar to the learning of a child).

- Most common approaches are based on:
  - Genetic Algorithms and Evolutionary Computing
  - Neural Networks
- <u>Idea:</u> Not always easy to code behaviors, let the robot figure it out by itself.

Learning Behaviors

- Typical "things" that are learnt by robots:
  - "How" to perform various behaviors:
    - light seeking, block pushing, obstacle avoidance, wall following
  - "When" to exhibit certain behaviors
  - How to walk … robot develops a walking "gait"
  - Trajectory planning
  - Navigation and Localization



- The "learning" approach is useful when the problem is not well understood and when it is difficult to hard-code solutions.

Learning Behaviors

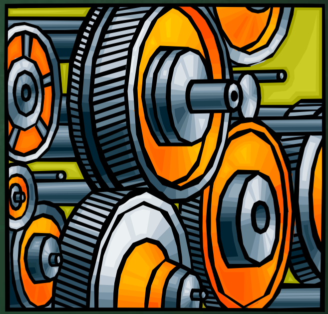
- There are advantages:
  + can produce solutions that may have otherwise been difficult to program.

  + solutions can be more robust and handle unpredicted scenarios
- There are disadvantages:
  - takes time to "train" the robot (may be impractical on real-robots with real sensors and battery limitations)
  - difficult to determine useful and efficient "fitness functions" (for GAs) for complex problems

  - optimal solution is not always found.

Learning Behaviors

- Perhaps a hybrid compromise is best:
  - Hardwire simple (obvious) behaviors (e.g., obstacle avoidance, wall following, light seeking)
  - Learn "when" to exhibit the behaviors.
- Only seems interesting when robot is sufficiently complex, containing many:
  - sensors (e.g., proximity, light, sound, vision)
  - actuators (e.g., wheels, arms, etc…)
  - internal monitors (e.g., clock, battery life)



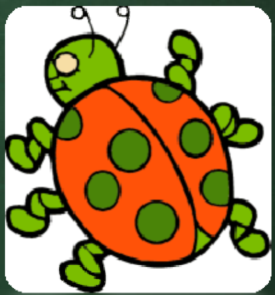
- Unfortunately, most work so far is based on learning simple behaviors on very simple robots.

Learning Behaviors

- Although genetic algorithms and neural networks can result is "good" behaviors, they are usually impractical for real robots due to the need for extensive training iterations.

- Also, for simple behaviors, there is no benefit to learning them as opposed to hardwiring them as instincts.

  e.g., research on training a 6-legged robot to walk results in a standard walking pattern that can be very easily hard-coded.



- We will examine a neural network to do this.

Artificial Neural Networks

Neural Networks

- There are opinions as to how to precisely define what a neural network is.
- It is commonly agreed upon that a ***neural network*** is a network of simple processing elements (called ***neurons***) which can exhibit complex global behavior, determined by the connections between the processing elements and element parameters.
- They are also called ***Artificial Neural Networks*** (ANN) or ***Simulated Neural Networks*** (SNN).

Neural Networks

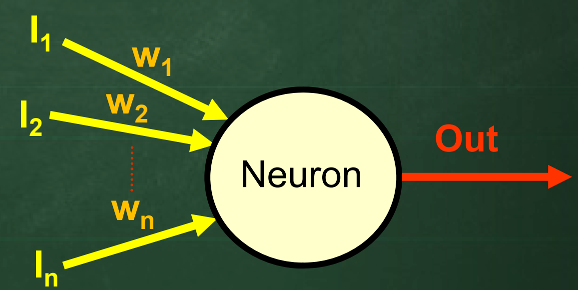
- Neural Networks are commonly used for:
  - *system identification and control*
    - (e.g., vehicle control, process control)
  - *game-playing* and *decision making*
    - (e.g., backgammon, chess, racing)
  - *pattern recognition*
    - (e.g., radar systems, face identification, object recognition)
  - *sequence recognition*
    - (e.g., gesture, speech, handwritten text recognition)
  - *medical diagnosis*

Neural Networks

- They are typically used to learn over time
  - They are typically trained to produce desired output
  - They are quite robust at handling unexpected data and dealing with noisy input
  - Their performance depends on various parameters
- There are many, kinds of networks that differ in:
  - The number of neurons
  - The organization and interconnectivity of neurons
  - The number of processing layers
  - The order of processing
  - The type of processing at each neuron

Neural Networks

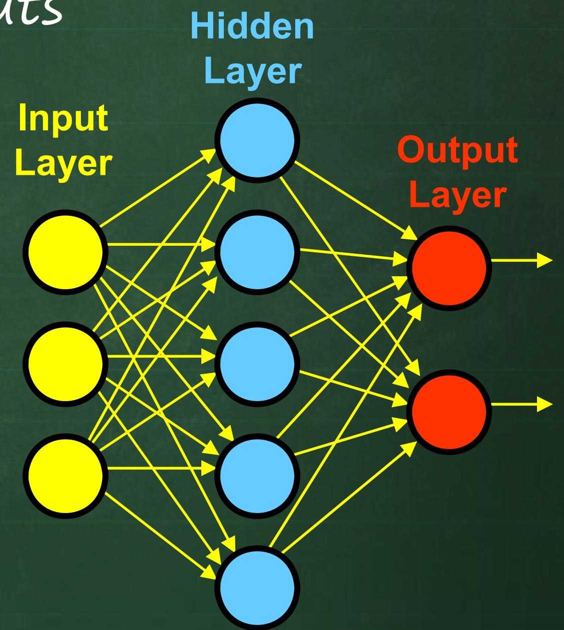
- Each neuron has multiple inputs I_1, I_2, ..., I_n
  - these are typically outputs from other neurons
  - they are represented by real number from 0 to 1
  - each input I_i has a corresponding weight w_i indicating the "significance" of the input for the neurons computation



- A neuron has one output, Out
  - it is a function of the inputs, usually a weighted sum
  - also represented by real number from 0 to 1

Neural Networks

- A neuron continually repeats this process:
  1. compute activation (usually a weighted sum)
  2. perform a simple operation based on inputs
  3. emit an output
- A network usually has an input layer, an output layer and one or more hidden layers of neurons.



- Activation begins at the input layer and *spreads* throughout the network to the outputs.

Neural Networks

- Over time, the values of each neuron as well as the weights are adjusted so as to produce the correct output.
- Initially the values are adjusted dramatically (usually using some sort of punishment/reward learning strategy) and then over time, only small changes are made to the network values.

- The most commonly used type of neural network is a *feed-forward* neural network in which there are no feedback connects (i.e., no loops).

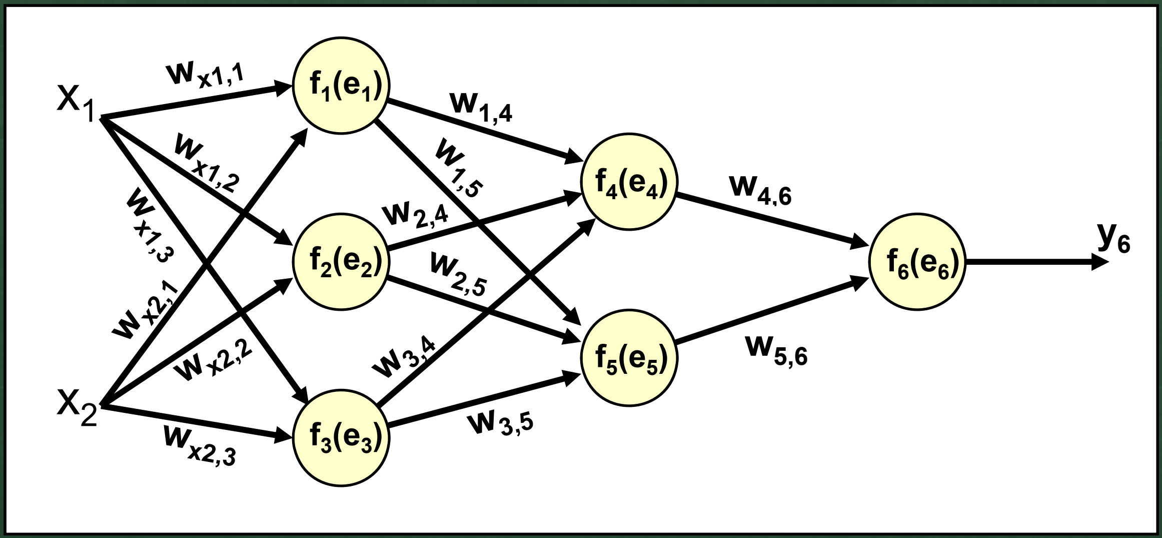
Back Propagation

- The training of a feed-forward neural network is commonly based on a supervised learning strategy called ***back propagation***:

1. Present a training sample to the neural network.
2. Compare the network's output to the desired output from that sample. Calculate the error in each output neuron.
3. For each neuron, calculate what the output should have been, and a ***scaling factor***, how much lower or higher the output must be adjusted to match the desired output. This is the local error.
4. Adjust the weights of each neuron to lower the local error.
5. Assign "blame" for the local error to neurons at the previous level, giving greater responsibility to neurons connected by stronger weights.
6. Repeat the steps above on the neurons at the previous level, using each one's "blame" as its error.

Back Propagation

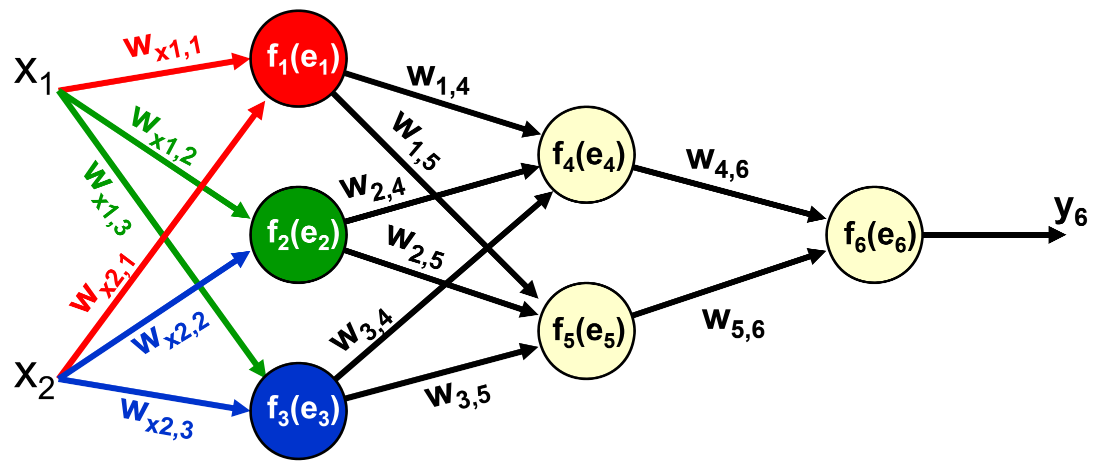
- Here is an example of how back-propagation works (example from **http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html**):



- E.g., $y_6 = f_6(e_6)$ is some (possibly non-linear) function of e_6 where $e_6 = f_4(e_4) \cdot w_{4,6} + f_5(e_5) \cdot w_{5,6}$

Back Propagation

- Assume that inputs are presented to the network.
- Compute activations for first level of network:



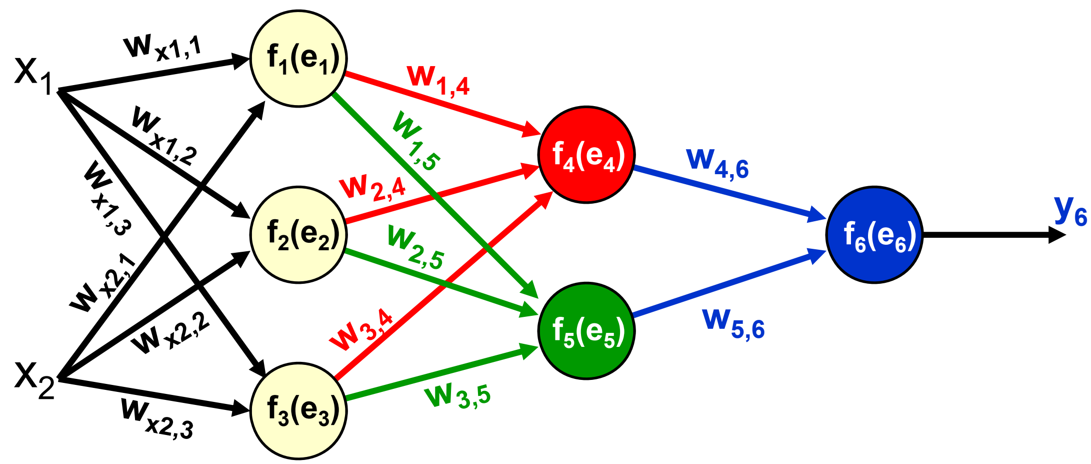
$$y_1 = f_1(e_1) = f(w_{x1,1} \cdot x_1 + w_{x2,1} \cdot x_2)$$

$$y_2 = f_2(e_2) = f(w_{x1,2} \cdot x_1 + w_{x2,2} \cdot x_2)$$

$$y_3 = f_3(e_3) = f(w_{x1,3} \cdot x_1 + w_{x2,3} \cdot x_2)$$

Back Propagation

- Compute activations for second and third levels of network:



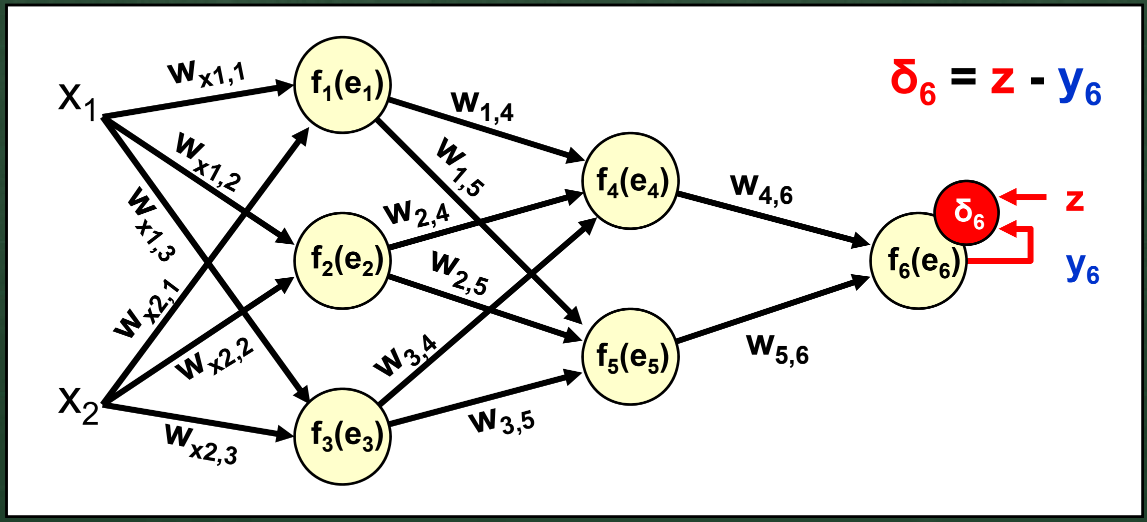
$$y_4 = f_4(e_4) = f(w_{1.4} \cdot y_1 + w_{2.4} \cdot y_2 + w_{3.4} \cdot y_3)$$

$$y_5 = f_5(e_5) = f(w_{1,5} \cdot y_1 + w_{2,5} \cdot y_2 + w_{3,5} \cdot y_3)$$

$$y_6 = f_6(e_6) = f(w_{4,6} \cdot y_4 + w_{5,6} \cdot y_5)$$

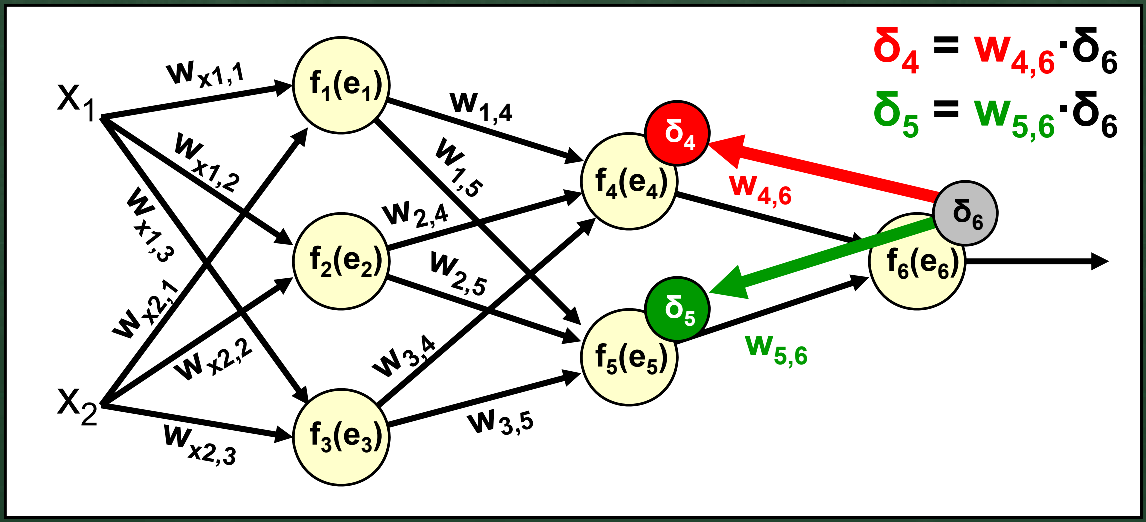
Back Propagation

- Now the output signal y_6 is compared with the desired output z which is usually found in the set of data used to train the network:



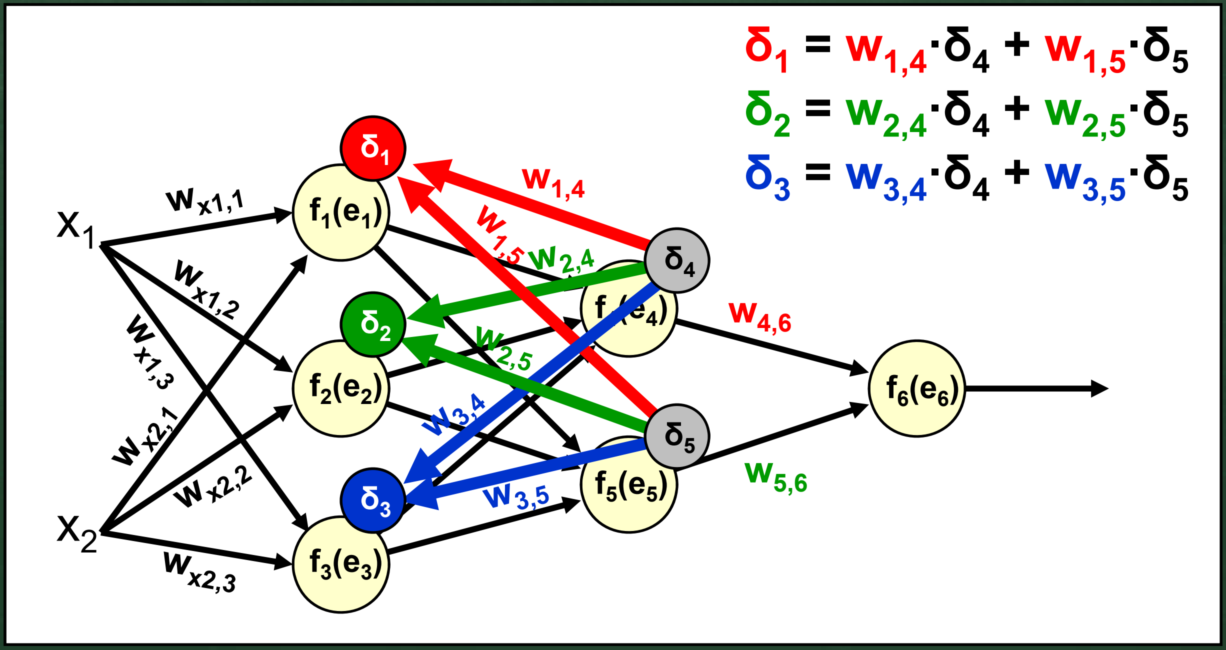
Back Propagation

- Now the output error δ_6 is propagated backwards to the previous layer, using the same weights as when computing the output:



Back Propagation

- Similarly, these errors are also propagated backwards:



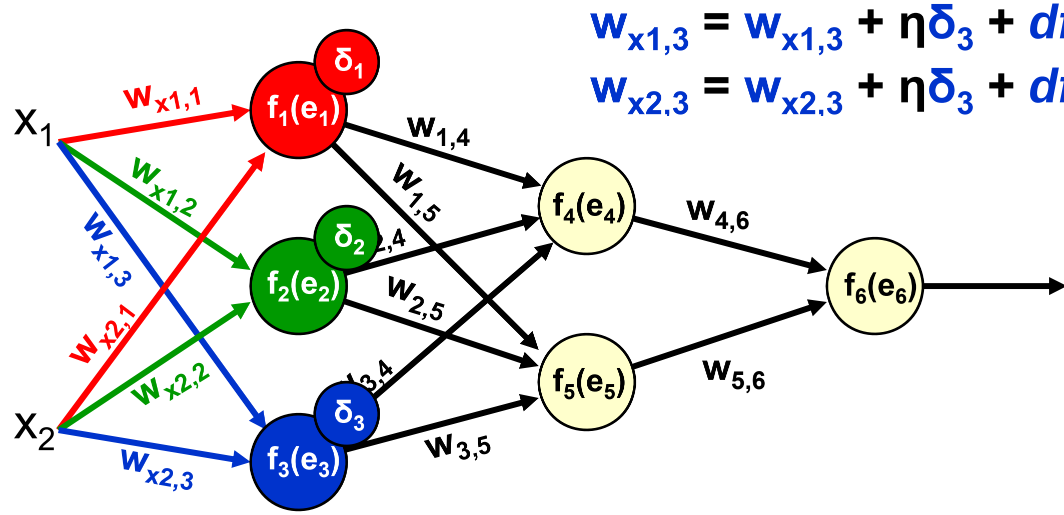
Back Propagation

- Now we update the weights throughout the network, starting at the first layer:

$df_i(e_i)/de_i$ is the derivative of the neuron activation function.

η is a "learning constant" which can vary over time.

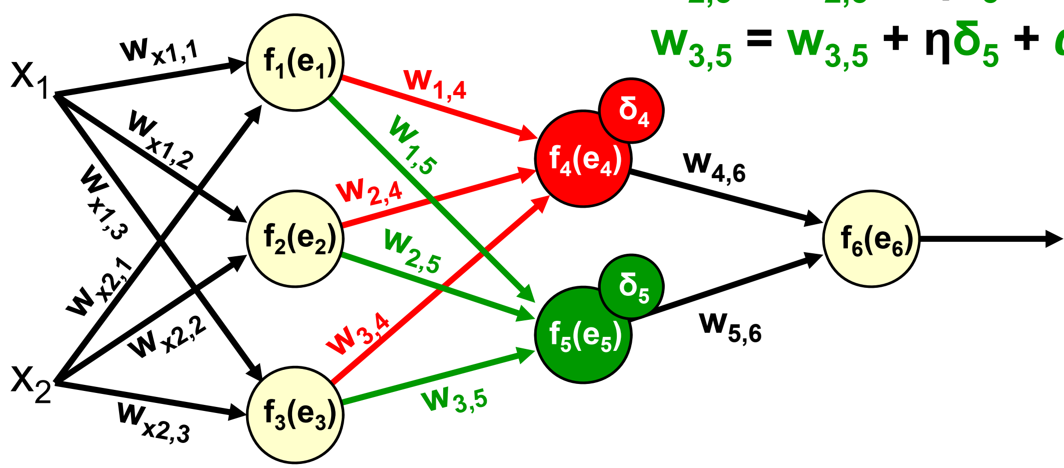
$$w_{x1,1} = w_{x1,1} + \eta\delta_1 + df_1(e_1)/de_1 \cdot x_1$$
$$w_{x2,1} = w_{x2,1} + \eta\delta_1 + df_1(e_1)/de_1 \cdot x_2$$
$$w_{x1,2} = w_{x1,2} + \eta\delta_2 + df_2(e_2)/de_2 \cdot x_1$$
$$w_{x2,2} = w_{x2,2} + \eta\delta_2 + df_2(e_2)/de_2 \cdot x_2$$
$$w_{x1,3} = w_{x1,3} + \eta\delta_3 + df_3(e_3)/de_3 \cdot x_1$$
$$w_{x2,3} = w_{x2,3} + \eta\delta_3 + df_3(e_3)/de_3 \cdot x_2$$



Back Propagation

- Update the weights again for the next layer, using the outputs from the previous layer:

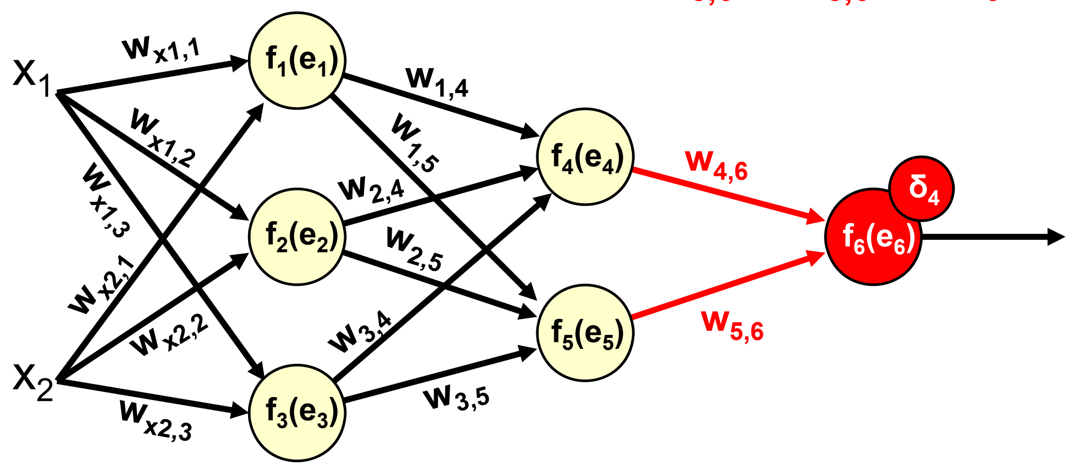
$$w_{1,4} = w_{1,4} + \eta\delta_4 + df_4(e_4)/de_4 \cdot y_1$$
$$w_{2,4} = w_{2,4} + \eta\delta_4 + df_4(e_4)/de_4 \cdot y_2$$
$$w_{3,4} = w_{3,4} + \eta\delta_4 + df_4(e_4)/de_4 \cdot y_3$$
$$w_{1,5} = w_{1,5} + \eta\delta_5 + df_5(e_5)/de_5 \cdot y_1$$
$$w_{2,5} = w_{2,5} + \eta\delta_5 + df_5(e_5)/de_5 \cdot y_2$$
$$w_{3,5} = w_{3,5} + \eta\delta_5 + df_5(e_5)/de_5 \cdot y_3$$



Back Propagation

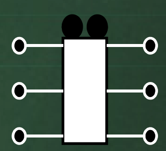
- Finally update the weights in last layer.
  - The coefficient η affects teaching speed. There are a few techniques to select this parameter. Often, however, this value is initially large and decreases over time. This allows "big" weight changes initially and eventually the network does not change much.

$$w_{4,6} = w_{4,6} + \eta\delta_6 + df_6(e_6)/de_6 \cdot y_4$$
$$w_{5,6} = w_{5,6} + \eta\delta_6 + df_6(e_6)/de_6 \cdot y_5$$

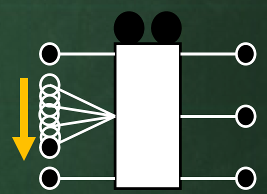


Neural Network Example

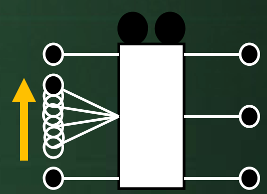
- Consider a back propagation network used to teach a robot to walk.



- Each leg can be in one of 3 modes:
  - Stance = leg down and pushing backward

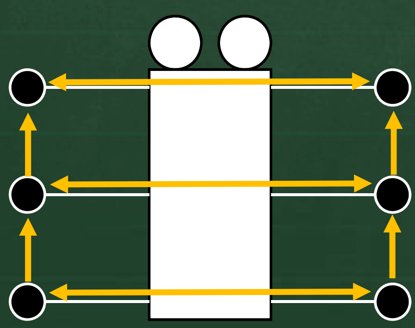


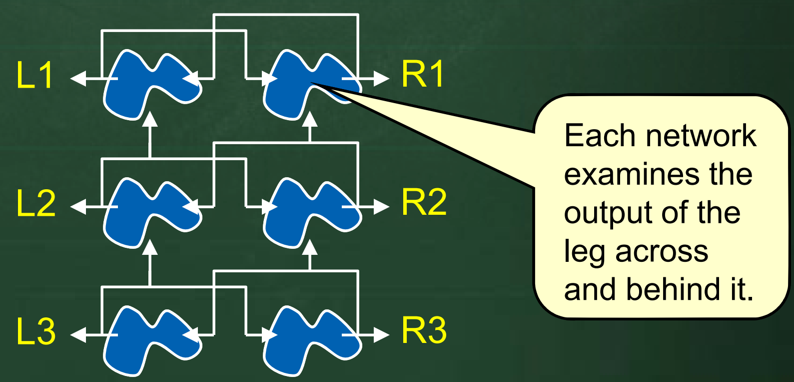
  - Still = leg not moving
  - Swing = leg swinging forward



Neural Network Example

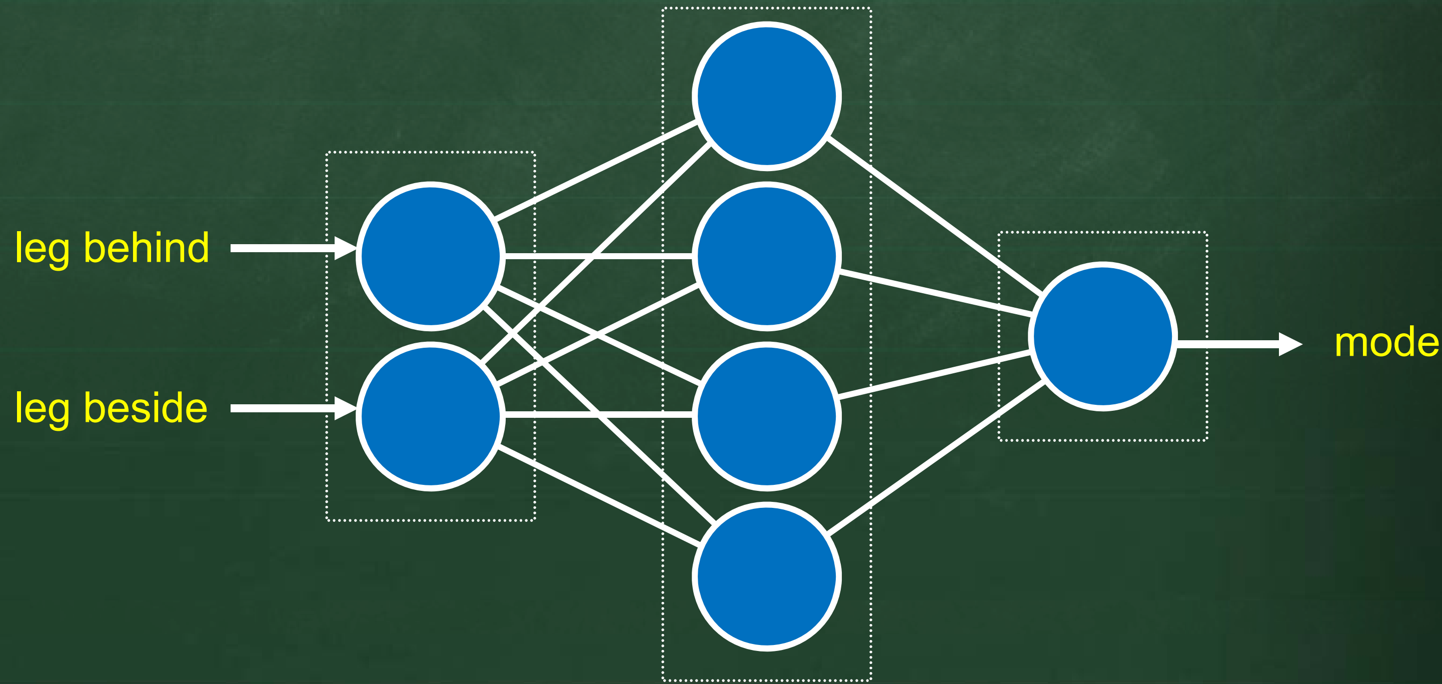
- Each leg must coordinate with the other legs in order to achieve walking.
- Can build a neural network for each leg and then interconnect them.
- One way of interconnecting is to only connect to the legs beside and behind it.





Neural Network Example

- The output of each network is the mode of the leg
  - 1 = stance, 0 = still, -1 = swing
- Each network may look like this:

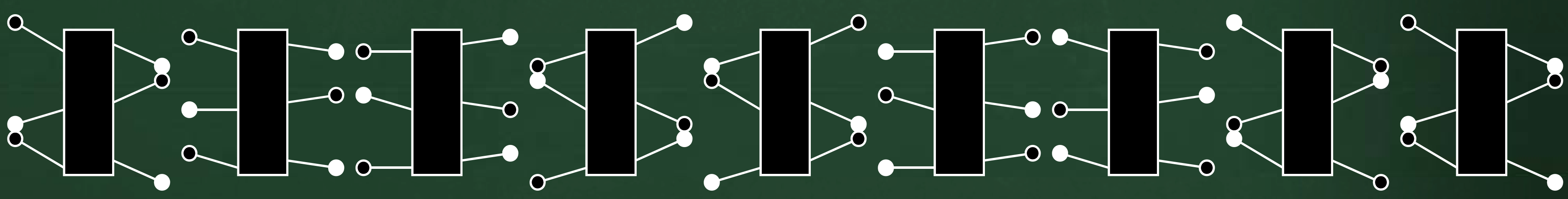


Neural Network Example

- Each leg must *learn* the correct mode per leg
- Networks begin with random weights on each link
- Weights are updated based on feedback from robot's success at walking.
  - When robot falls down, networks are punished
    - Based on backwards propagation which lessens weight on links that led to the chosen mode
    - more likely next time to use a different phase
    - punish ALL networks (even if one was correct)
  - When robot moves forward without falling they are all rewarded … by increasing weights along proper path.

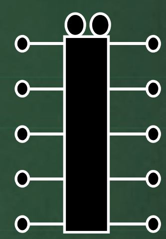
Neural Network Example

- As a result, robot learns to coordinate the legs over time.
- Typically, the "amount" of punishment and reward is large at the beginning and decreases over time.
  - Larger weight updates cause quicker changes in modes
  - Smaller weights can take a long time to converge to a proper behavior.
- Typical tripod walking "gait" is obtained:

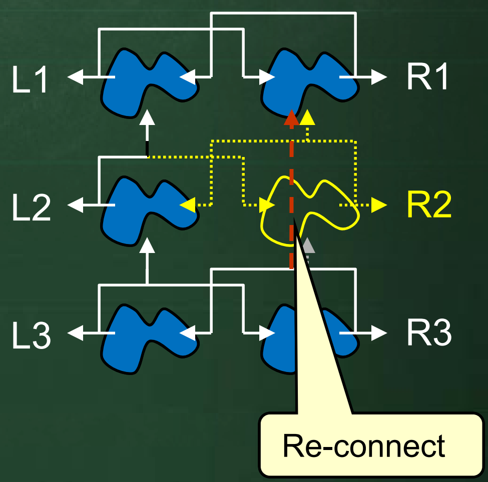


Neural Network Example

- Technique applies to robots with any number of legs.



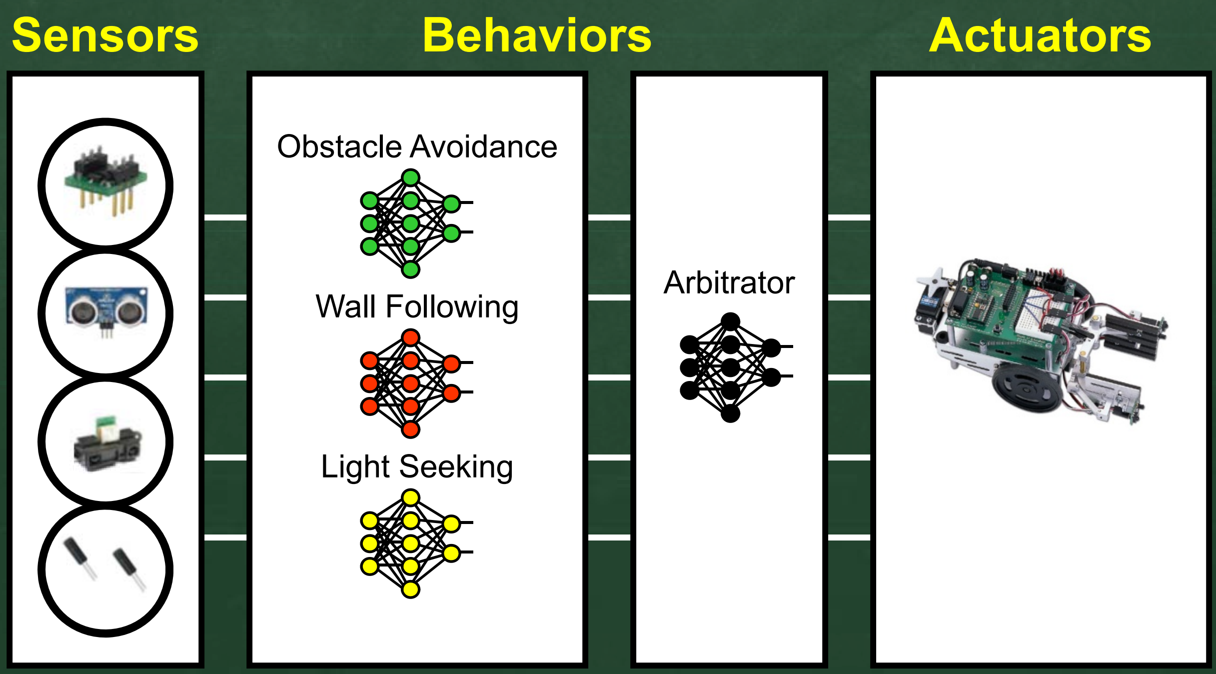
- Can also handle leg failure with minor adjustment:
  - Will need to re-teach all networks again
  - Better performance if reconnection of networks is allowed:
    - can do this manually
    - can re-design all networks to have inputs from ALL other legs.



Instinctive Behaviors Using Neuron Networks

Neuron Networks

- Consider a network containing a mix and match of various neurons to control a robot's behavior.
  - We can create a network for each type of behavior and then plug them all in together to steer the robot

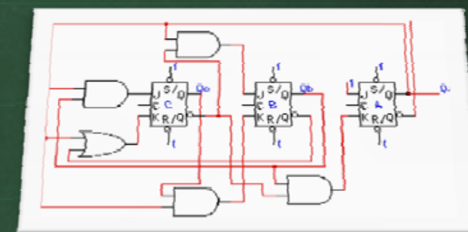


Neuron Networks

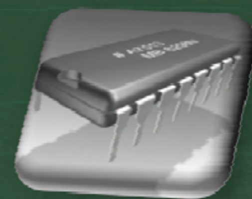
- Many neural networks are trained to learn how to perform simple behaviors.
- Hardwired *Neuron Networks* bypass the learning process of traditional neural networks
  - Same idea as building *instincts* into the robot
- Idea is to avoid training stage for behaviors that are already well-defined … for example:
  - wandering, obstacle avoidance, light seeking, edge following, map building etc…

Neuron Networks

- Neuron networks can be better than programming:
  - + neurons implemented with electronics



  - + entire networks can be made on single electronic "chip"



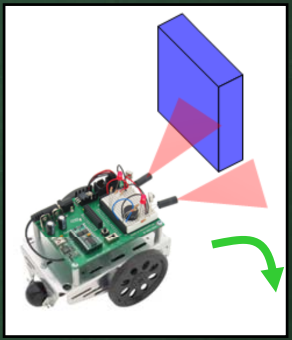
  - + can be made very small using very little power
  - + cheap to produce

Neuron Networks

- Of course, traditional neural networks can be trained to accomplish the same thing.
- Unlike neural networks, these hard-wired networks allow easy enabling and disabling of behaviors over time.
- They can even be mixed and matched with traditional neural networks

Neuron Networks

- Consider modeling a behavior that always moves a robot forward until it detects an obstacle ahead using its left or right IR proximity sensor.
  - If it detects an obstacle on its left it should then turn right (and vice-versa).



- We can create three types of neurons:
  - ***sensor neuron*** that acts as a binary *input* neuron and outputs a value of 0 if no obstacle is detected and 1 otherwise.
  - ***motor neuron*** that acts as a binary *output* neuron that turns on a motor when its output is 1 and turns off otherwise.
  - ***control neuron*** that enables a behavior

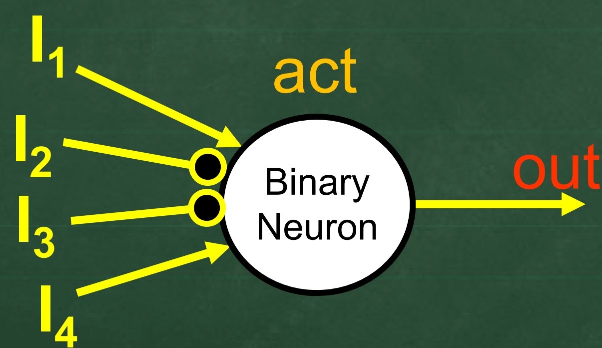
Neuron Networks

- We can also create two types of connections:
  - **excitatory** – weight of 1.0
  - **inhibitory** – weight of -1.0
- In general, a neuron computes its **activation** as
  - sum of its inputs (i.e., any real number, possibly negative) times the weight of incoming connection: $act = \sum_{i=1}^{n} I_i w_i$
- The output of a neuron is the activation itself or some function of the activation
  - (e.g., *binary* neurons may output 1 if activation > 0 and 0 otherwise).

Neuron Networks

- For example for the following neuron, the activation is computed as:

$$act = I_1(1) + I_2(-1) + I_3(-1) + I_4(1)$$

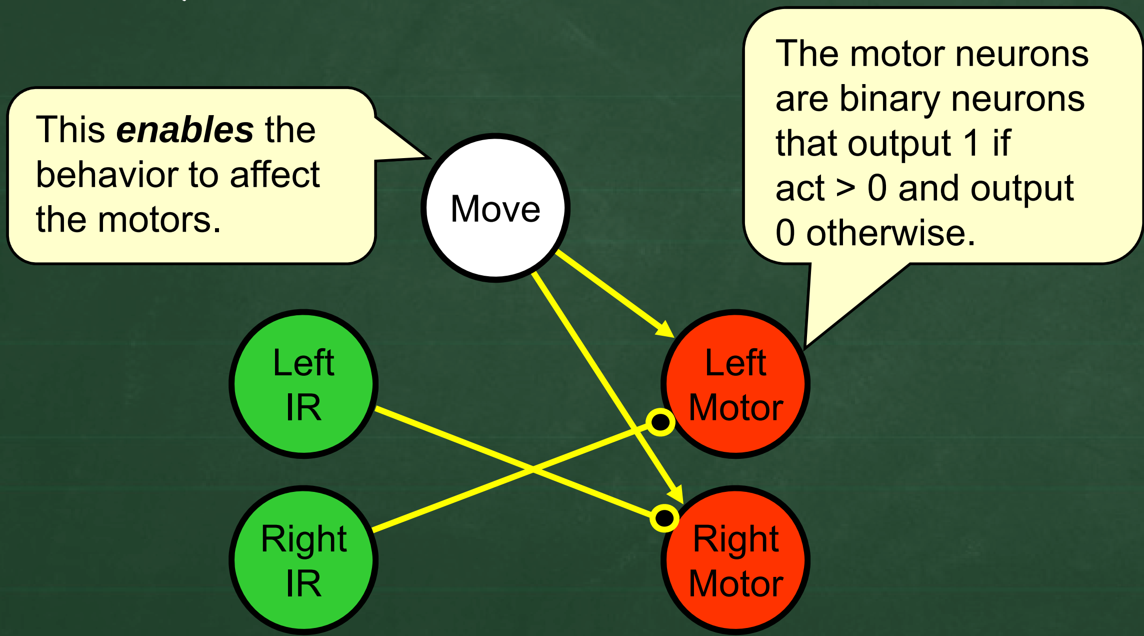


I_1	I_2	I_3	I_4	act	out
0	0	0	0	0	0
1	0	0	0	1	1
0	1	0	0	-1	0
1	0	0	1	2	1
0	1	1	0	-2	0
1	1	1	1	0	0
-1	0	0	0	-1	0
0	-1	0	0	1	1
-1	0	0	-1	-2	0
0	-1	-1	0	2	1
-1	-1	-1	-1	0	0
1	-1	-1	1	4	1
-1	1	1	-1	-4	0

- The table to the right gives the activation and output values of a binary neuron for some possible input values.

Neuron Nets – Collision Avoidance

- *Here is the network and a table showing the possible outputs:*

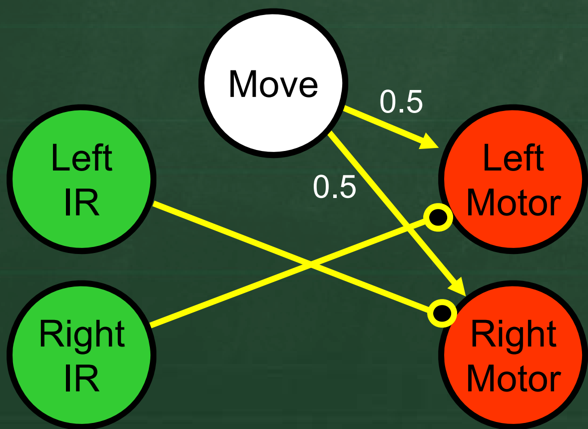


L IR	R IR	Move	L Mtr	R Mtr
0	0	0	0	0
0	1	0	0	0
1	0	0	0	0
1	1	0	0	0
0	0	1	1	1
0	1	1	0	1
1	0	1	1	0
1	1	1	0	0

- *Notice that the motors turn in the appropriate direction so as to avoid the obstacle.*

Neuron Nets – Collision Avoidance

- Can allow robot to spin away from obstacle by reversing opposite motor:
  - allow motor neuron to output -1, 0 or 1 according to sign of activation value
  - supplying smaller weight from Move neuron



L IR	R IR	Move	L Mtr	R Mtr
0	0	0	0	0
0	1	0	0	0
1	0	0	0	0
1	1	0	0	0
0	0	1	1	1
0	1	1	-1	1
1	0	1	1	-1
1	1	1	-1	-1

Forward

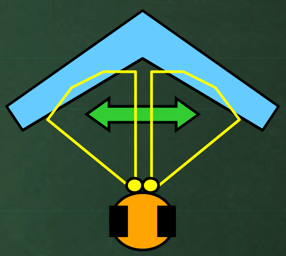
Spin Left

Spin Right

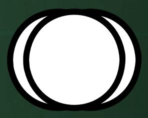
Backward

Neuron Nets – Collision Avoidance

- Recall the problem where the robot may become "stuck" in a corner, turning back and forth ?



- We can solve this by introducing another type of neuron called a binary ***sustain neuron*** which computes its output based on the current activation as well as the previous output:
  - act > 0 → out = 1
  - act = 0 → out = previous out value
  - act < 0 → out = 0



Neuron Nets – Collision Avoidance

- Here is an updated network … but it does not work properly … what's wrong ?

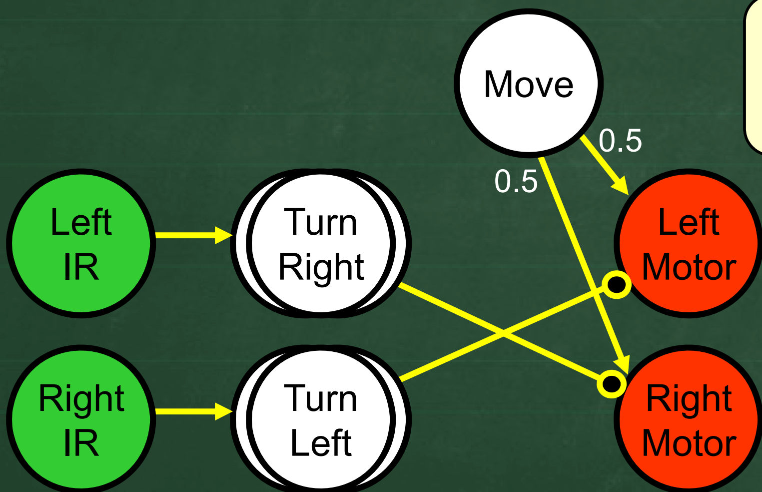


Table assumes Move = 1

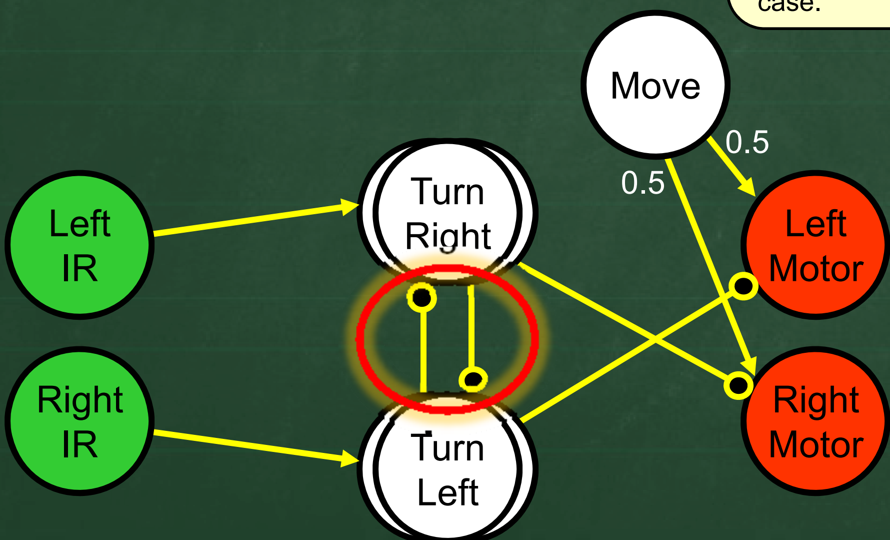
L IR	R IR	Prev Trn L	Prev Trn R	New Trn L	New Trn R	L Mtr	R Mtr
0	0	0	0	0	0	1	1
0	1	0	0	1	0	-1	1
1	0	0	0	0	1	1	-1
1	**1**	**0**	**0**	**1**	**1**	**0**	**0**
0	0	0	1	0	1	1	-1
0	**1**	**0**	**1**	**1**	**1**	**0**	**0**
1	0	0	1	0	1	1	-1
1	**1**	**0**	**1**	**1**	**1**	**0**	**0**
0	0	1	0	1	0	-1	1
0	1	1	0	1	0	-1	1
1	**0**	**1**	**0**	**1**	**1**	**0**	**0**
1	**1**	**1**	**0**	**1**	**1**	**0**	**0**
0	**0**	**1**	**1**	**1**	**1**	**0**	**0**
0	**1**	**1**	**1**	**1**	**1**	**0**	**0**
1	**0**	**1**	**1**	**1**	**1**	**0**	**0**
1	**1**	**1**	**1**	**1**	**1**	**0**	**0**

All yellow rows indicate no movement. Basically, the **Turn Left** and **Turn Right** neurons should NEVER be on together.

Neuron Nets – Collision Avoidance

- Add inhibitory links to prevent one sustain neuron from turning on if other one is on:

A ***race condition*** occurs here, whichever neuron is processed first. X is either 0 or 1 in this case.



L IR	R IR	Prev Trn L	Prev Trn R	New Trn L	New Trn R	L Mtr	R Mtr
0	0	0	0	0	0	1	1
0	1	0	0	1	0	-1	1
1	0	0	0	0	1	1	-1
1	**1**	**0**	**0**	**x**	**1-x**	**1|-1**	**-1|1**
0	0	0	1	0	1	1	-1
0	**1**	**0**	**1**	**0**	**1**	**1**	**-1**
1	0	0	1	0	1	1	-1
1	**1**	**0**	**1**	**0**	**1**	**1**	**-1**
0	0	1	0	1	0	-1	1
0	1	1	0	1	0	-1	1
1	**0**	**1**	**0**	**1**	**0**	**-1**	**1**
1	**1**	**1**	**0**	**1**	**0**	**-1**	**1**
0	**0**	**1**	**1**	**N/A**	**N/A**	**N/A**	**N/A**
0	**1**	**1**	**1**	**N/A**	**N/A**	**N/A**	**N/A**
1	**0**	**1**	**1**	**N/A**	**N/A**	**N/A**	**N/A**
1	**1**	**1**	**1**	**N/A**	**N/A**	**N/A**	**N/A**

All yellow rows indicate an impossible state since we have prevented the sustain neurons from being on together.

SOMETHING IS STILL WRONG !!!!

Neuron Nets – Collision Avoidance

- The network does not allow the robot to stop turning once it has started turning away from the obstacle.
  - should disable sustain neurons when collision is no longer detected

- Introduce a new neuron called a ***pulse neuron***:
  - *Falling Edge Pulse* Neuron

    out = 1 when its activation changes from > 0 to ≤ 0

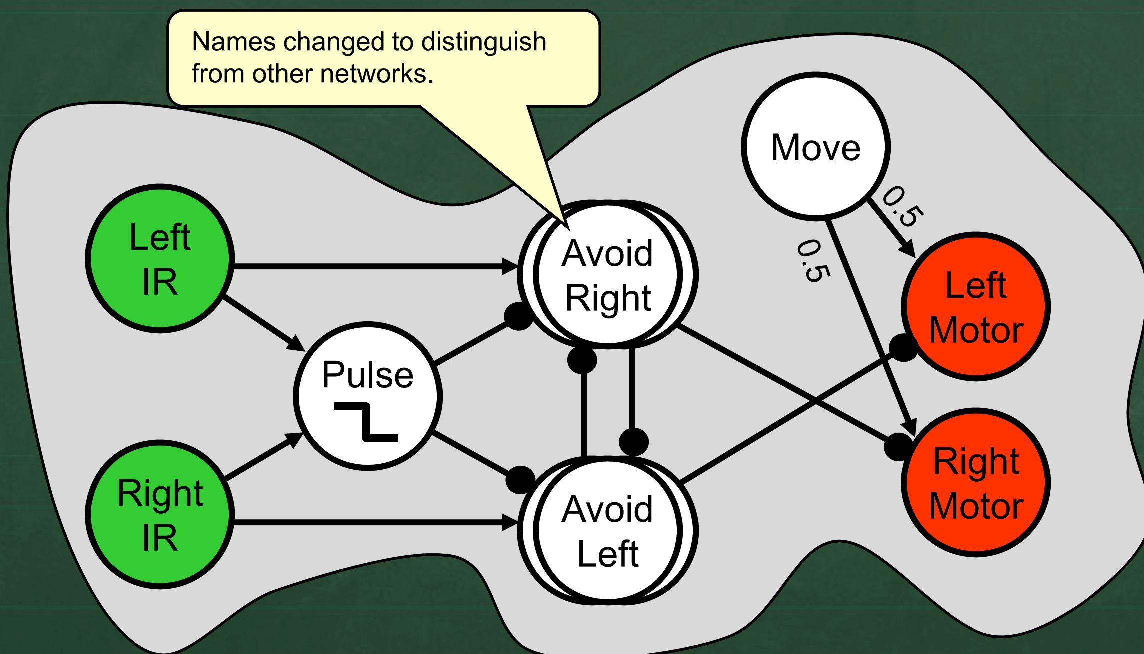
  - *Rising Edge Pulse* Neuron

    out = 1 when its activation changes from ≤ 0 to > 0

  - Output is 0 otherwise in both cases

Neuron Nets – Collision Avoidance

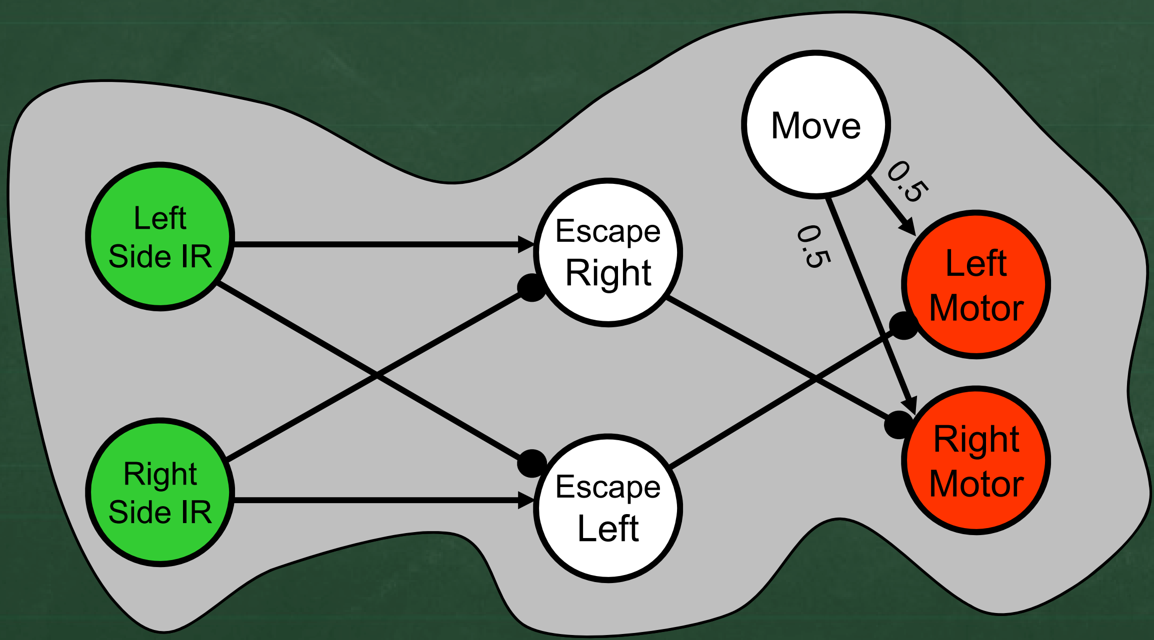
- Here is the completed collision avoidance network:

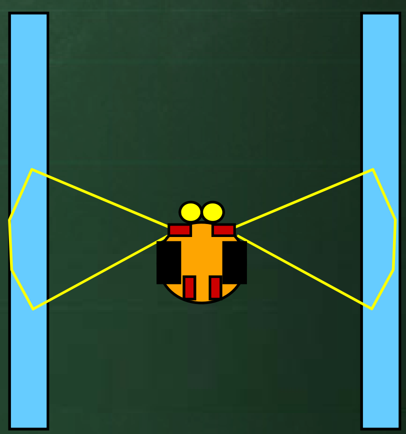


- Do you remember which way the robot turns when both its front IR sensors detect an obstacle ?

Neuron Nets – *Escape*

- The escape network is similar, but much easier:

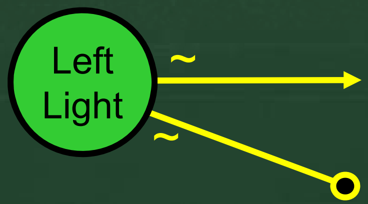




- What happens when both side IR sensors detect an obstacle ?

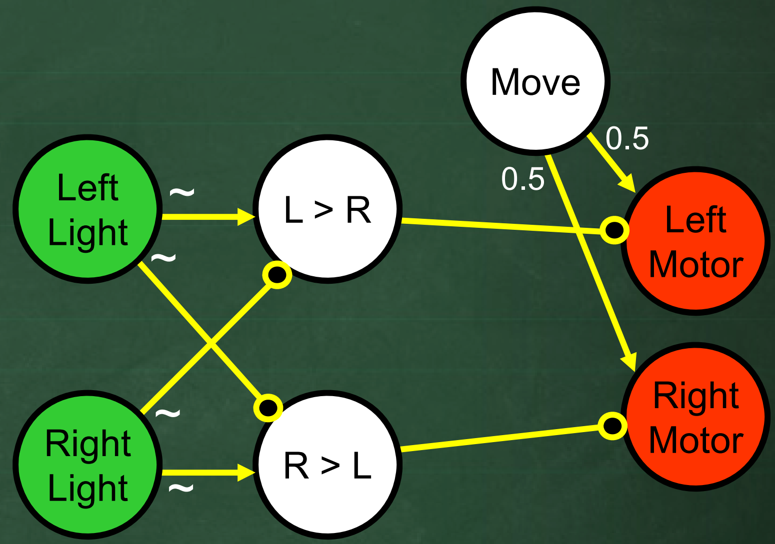
Neuron Nets – Light Seeking

- Light-seeking behavior involves determining the difference between two light sensor values.
- We will allow sensor neurons to have an output corresponding to the intensity of the light
  - (e.g., voltage value normalized so that it outputs a value from 0.0 to 1.0 depending on the light intensity)
  - designate a non-binary output with a ~ symbol on the links leaving the neuron:



Neuron Nets – Light Seeking

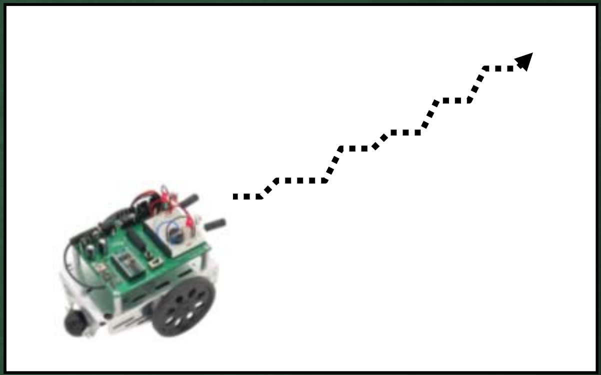
- Here is a similar network:
- Notice the outputs in the table below.
- This works, but can you foresee any problems with a practical implementation on a real robot ?



L Lgt	R Lgt	L > R	R > L	L Mtr	R Mtr
0	0	0	0	1	1
0	> 0	0	1	1	-1
> 0	0	1	0	-1	1
x	x	0	0	1	1
x	< x	1	0	-1	1
x	> x	0	1	1	-1

Neuron Nets – Light Seeking

- Yes, when the light sensors are both pointed towards or away from the light source equally, real sensors will fluctuate in t heir readings
  - causes robot to "flutter" or zig-zag
  - can be hard on motors



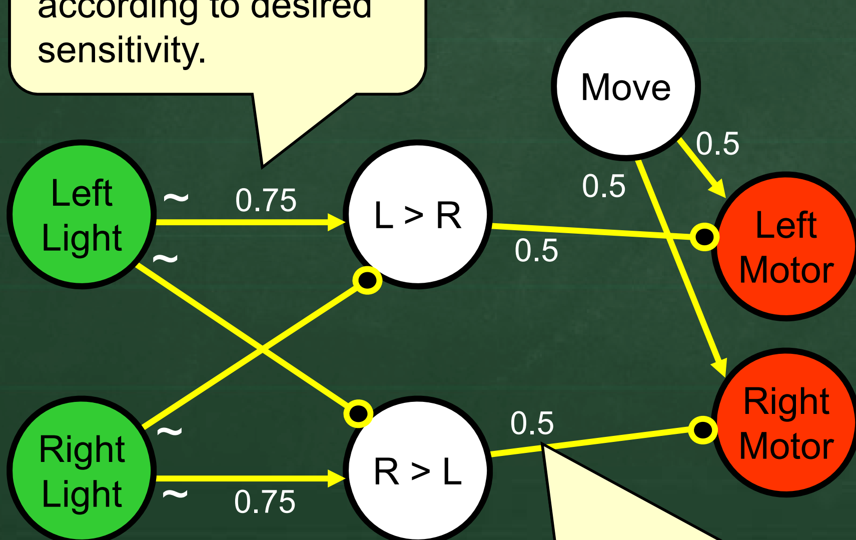
- Turns are also "spins" so robot actually stops at each zig and zag.
- Can reduce this effect a little by only turning when one sensor has a value "significantly" larger than the other.

Neuron Nets – *Light Seeking*

- *Just modify the weights from the sensors and to motors:*

Choose threshold according to desired sensitivity.

Insignificant differences do not cause turning.



L Lgt	R Lgt	L > R	R > L	L Mtr	R Mtr
0	0	0	0	1	1
0	> 0	0	1	1	0
> 0	0	1	0	0	1
x	≥0.75x & ≤1.33x	0	0	1	1
x	<0.75x	1	0	0	1
x	>1.33x	0	1	1	0

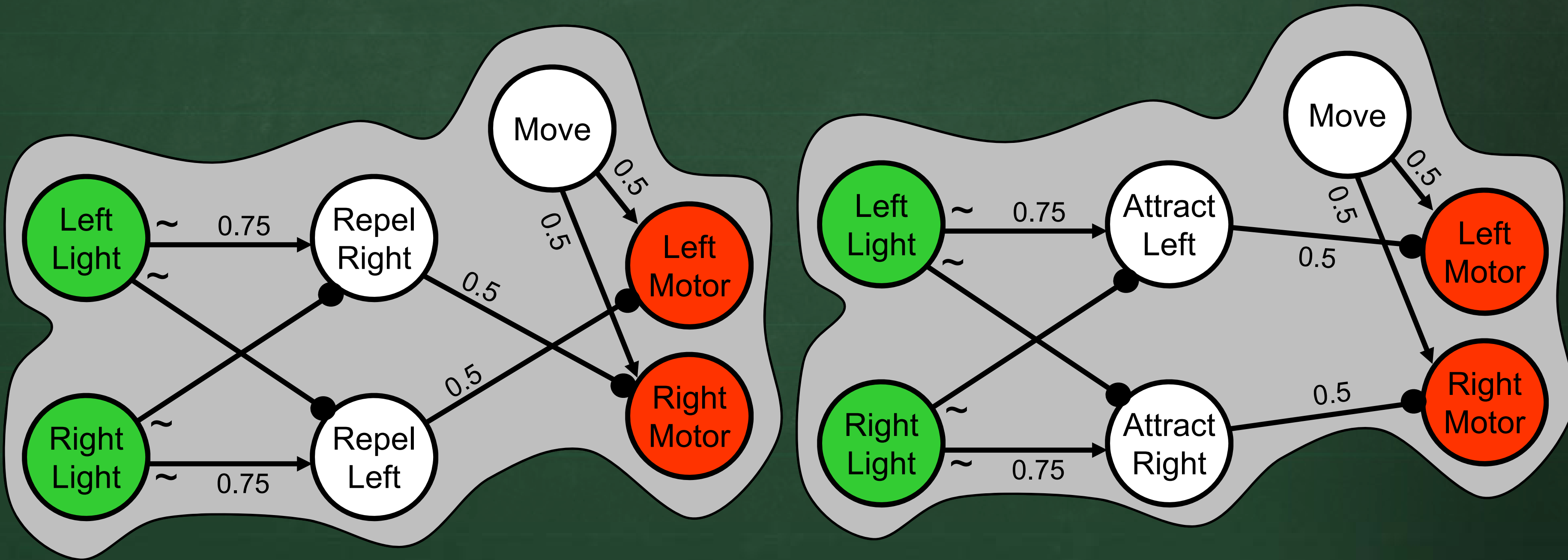
1/0.75 = 1.33333

No more negative, means smoother turning.

New weights prevent spinning, but simply turning off one motor instead of reversing it.

Neuron Nets – Light Seeking

- Finally, can add neurons to decide whether to be attracted or repelled from light source:

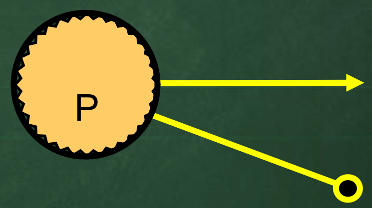


Repel from Light

Attract to Light

Neuron Nets – Wandering

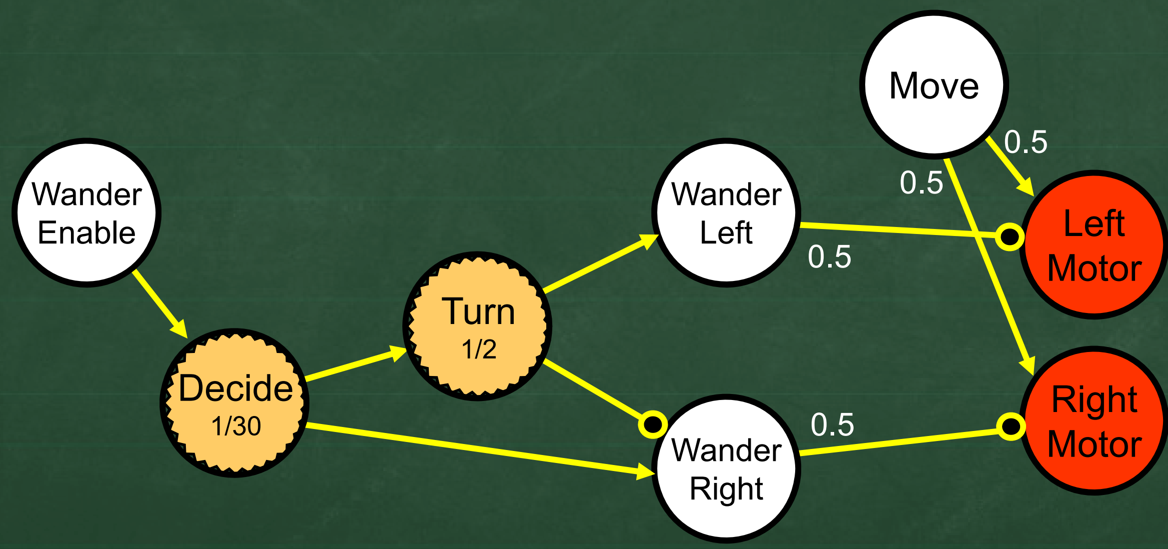
- For wandering, we must introduce the notion of a ***random neuron*** that can produce a random value:
  - value neuron hard-codes a fixed probability value P representing the likelihood of producing a binary output
  - act = input sum * random value from 0.0 to 1.0
  - if act > P → out = 0
  - if act ≤ P → out = 1



- For smooth wandering, we need to decide:
  - when to make a turn
  - which way to turn
  - how long to turn.

Neuron Nets – Wandering

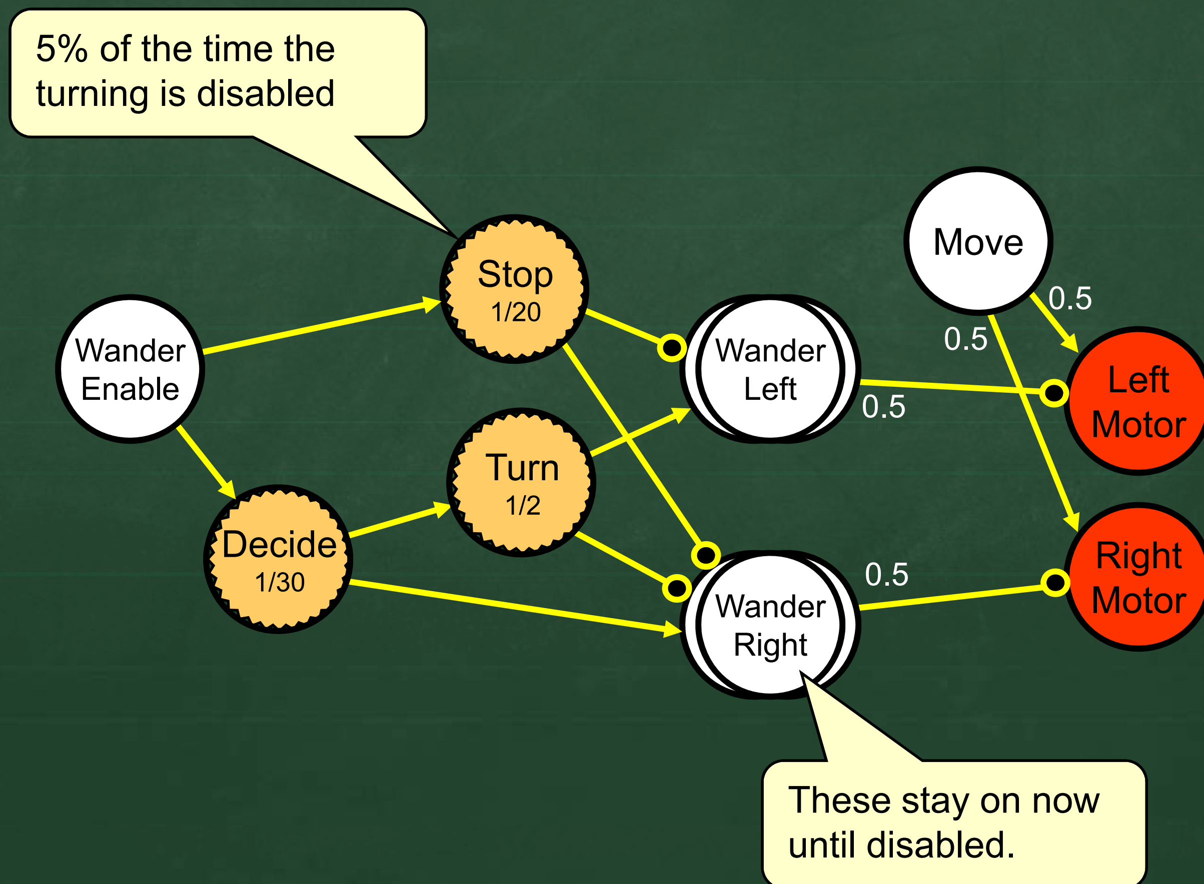
- This network allows the robot to make a random turn roughly 1/30 = 3% of the time:



- Problem:
  - robot makes a single turn … will appear as a "twitch"
- Must keep turning by some random amount

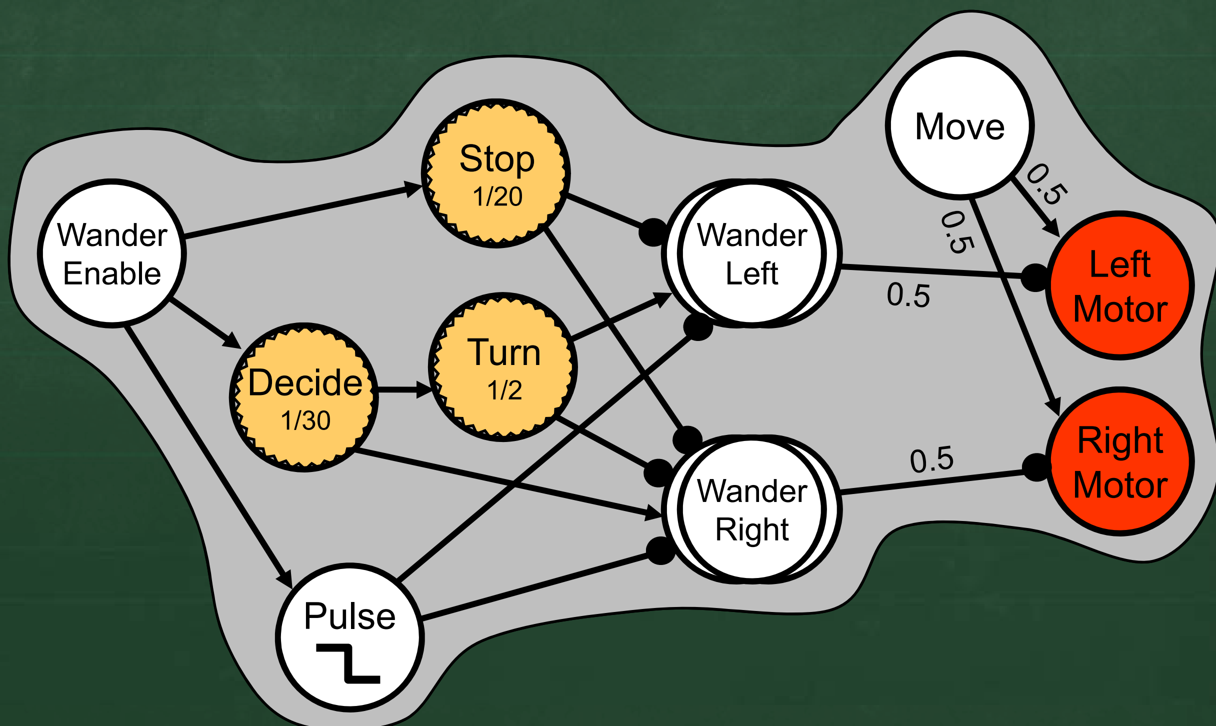
Neuron Nets – Wandering

- Use sustain neurons and disable them randomly:



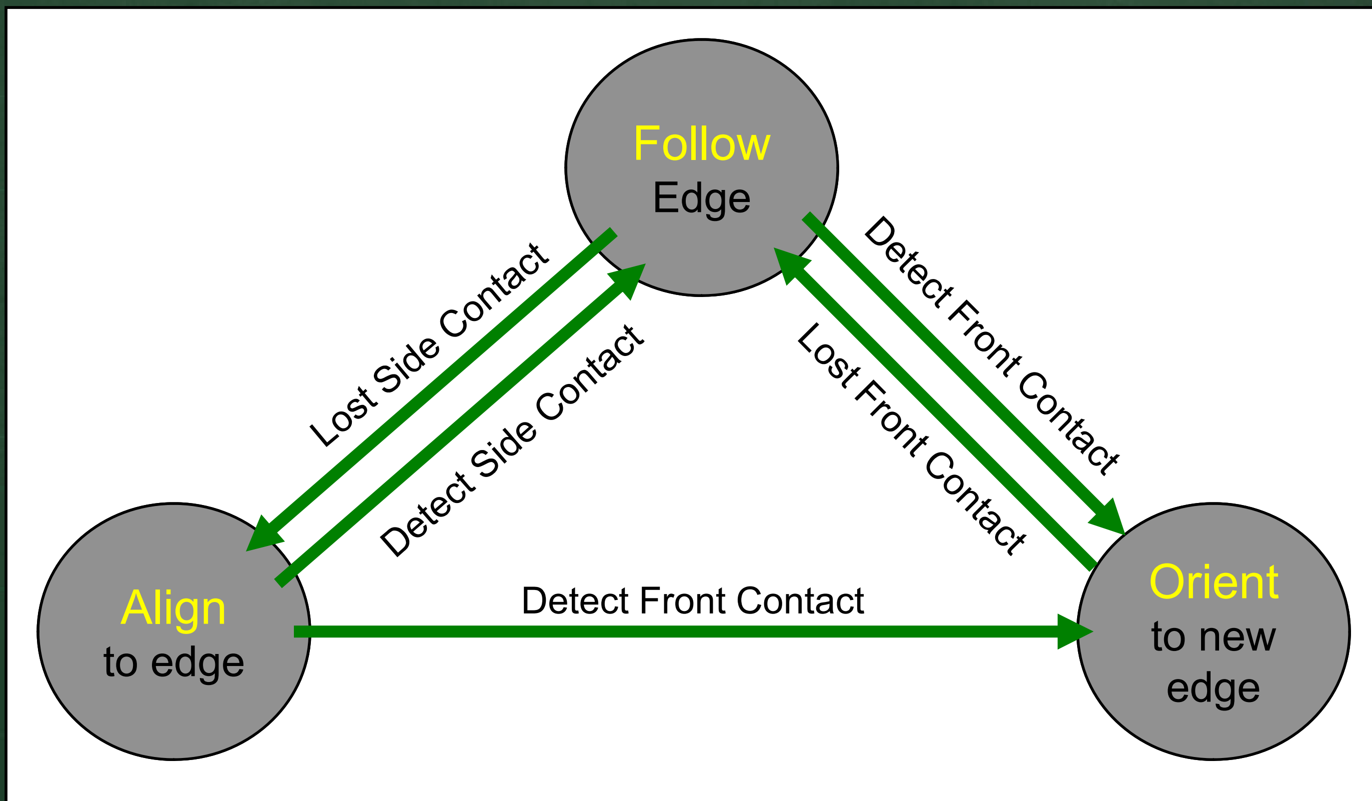
Neuron Nets – Wandering

- Can also disable whenever enabling neuron is disabled by using a pulse neuron. Here is the final network:



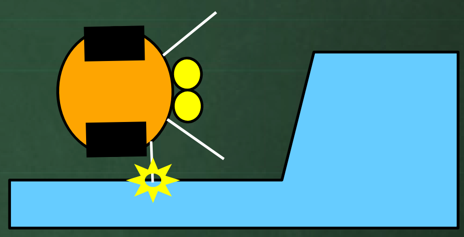
Neuron Nets – *Edge Following*

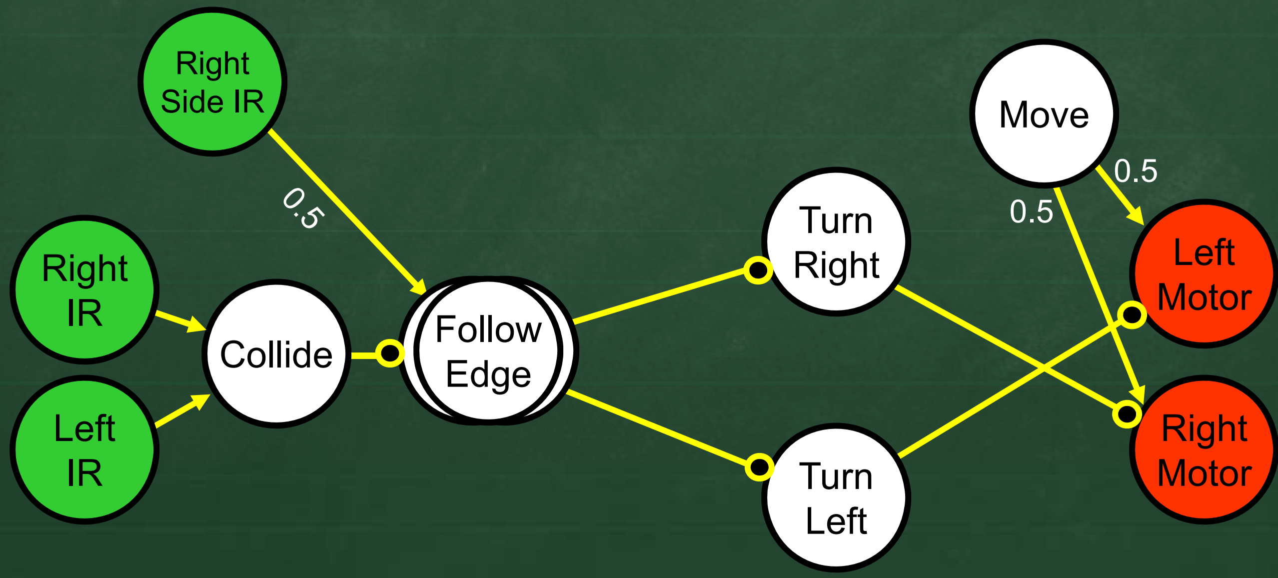
- Recall the stages of edge following:



Neuron Nets – *Edge Following*

- Consider following an edge on the right and moving forward as long as the right sensor detects the edge:

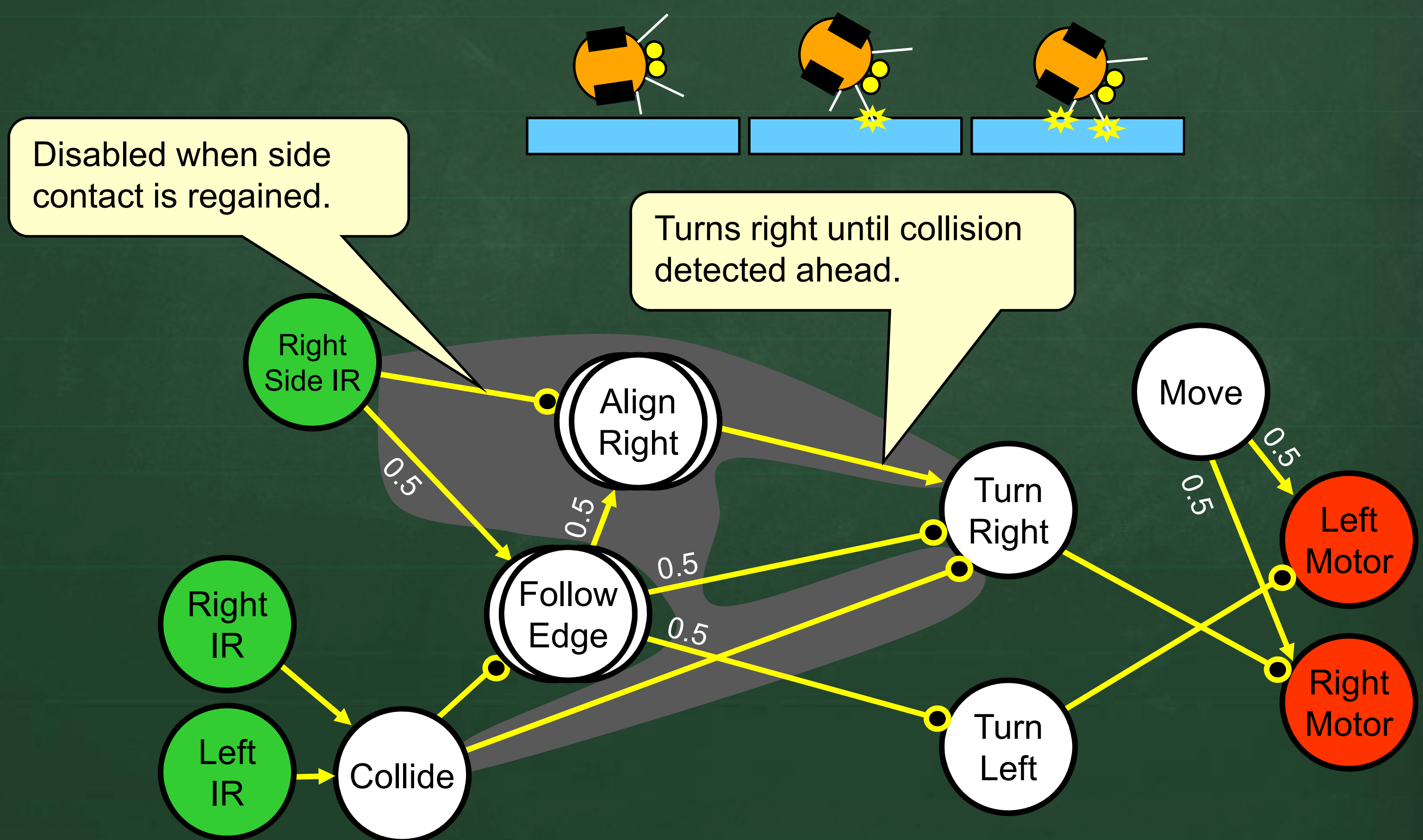




- What happens if the robot loses contact ?

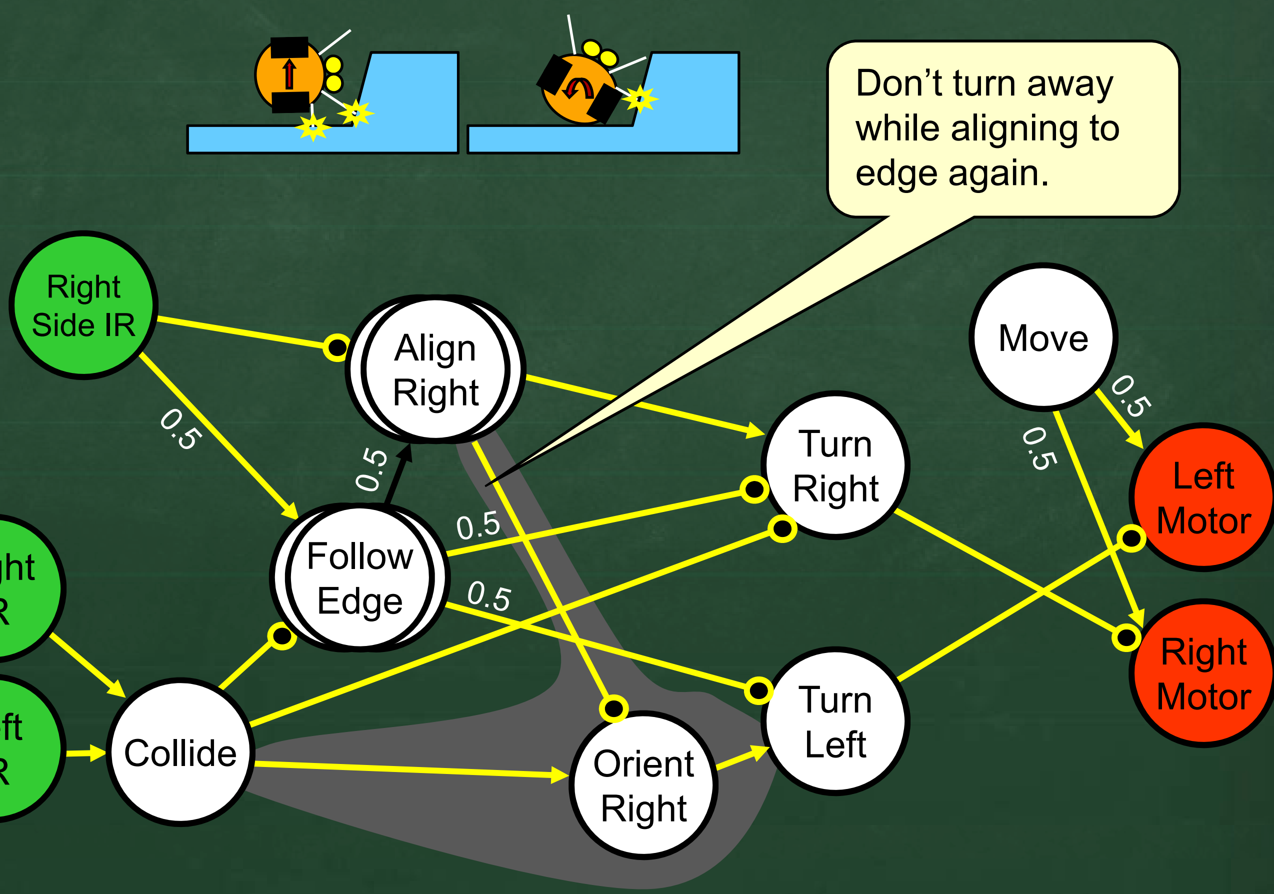
Neuron Nets – *Edge Following*

- When contact lost, must turn right to regain:



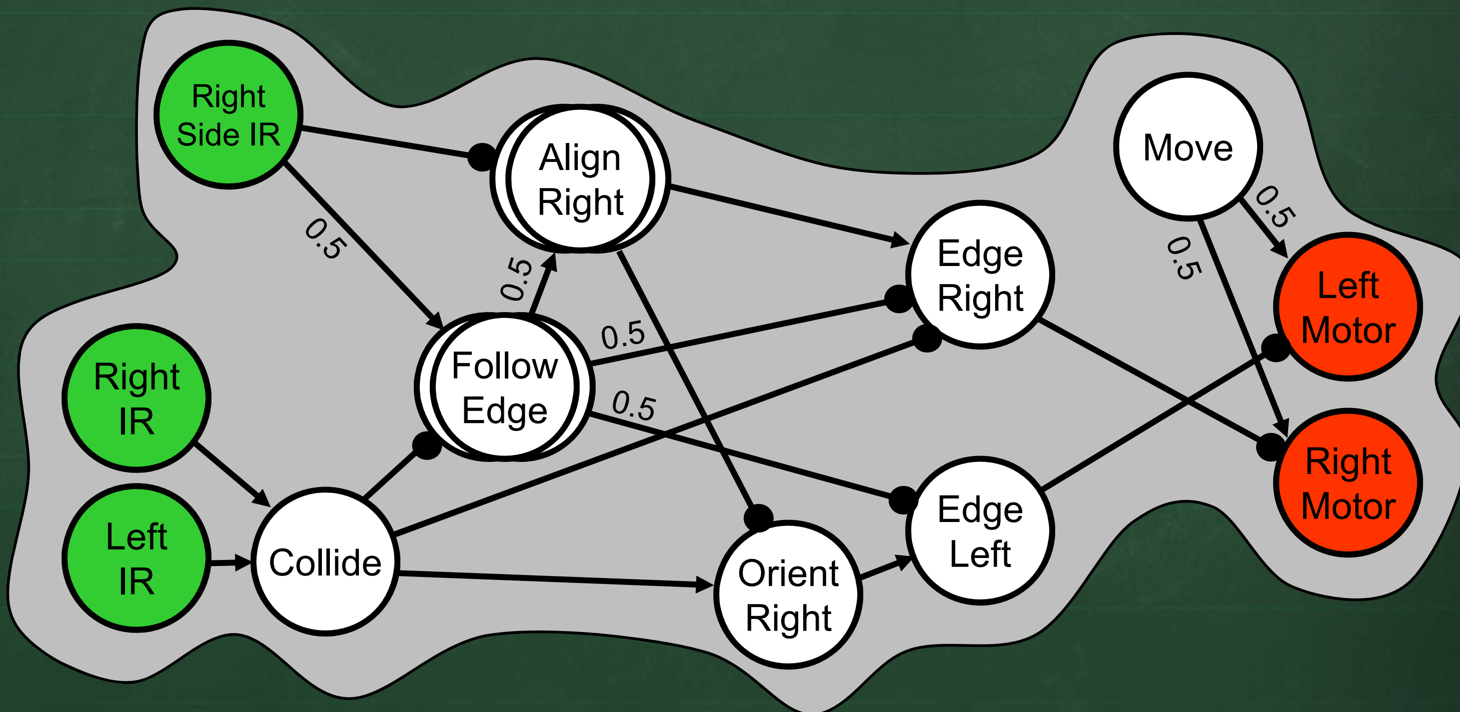
Neuron Nets – Edge Following

- Whenever a collision is detected, turn left to avoid it … unless we are aligning to the edge again:



Neuron Nets – Edge Following

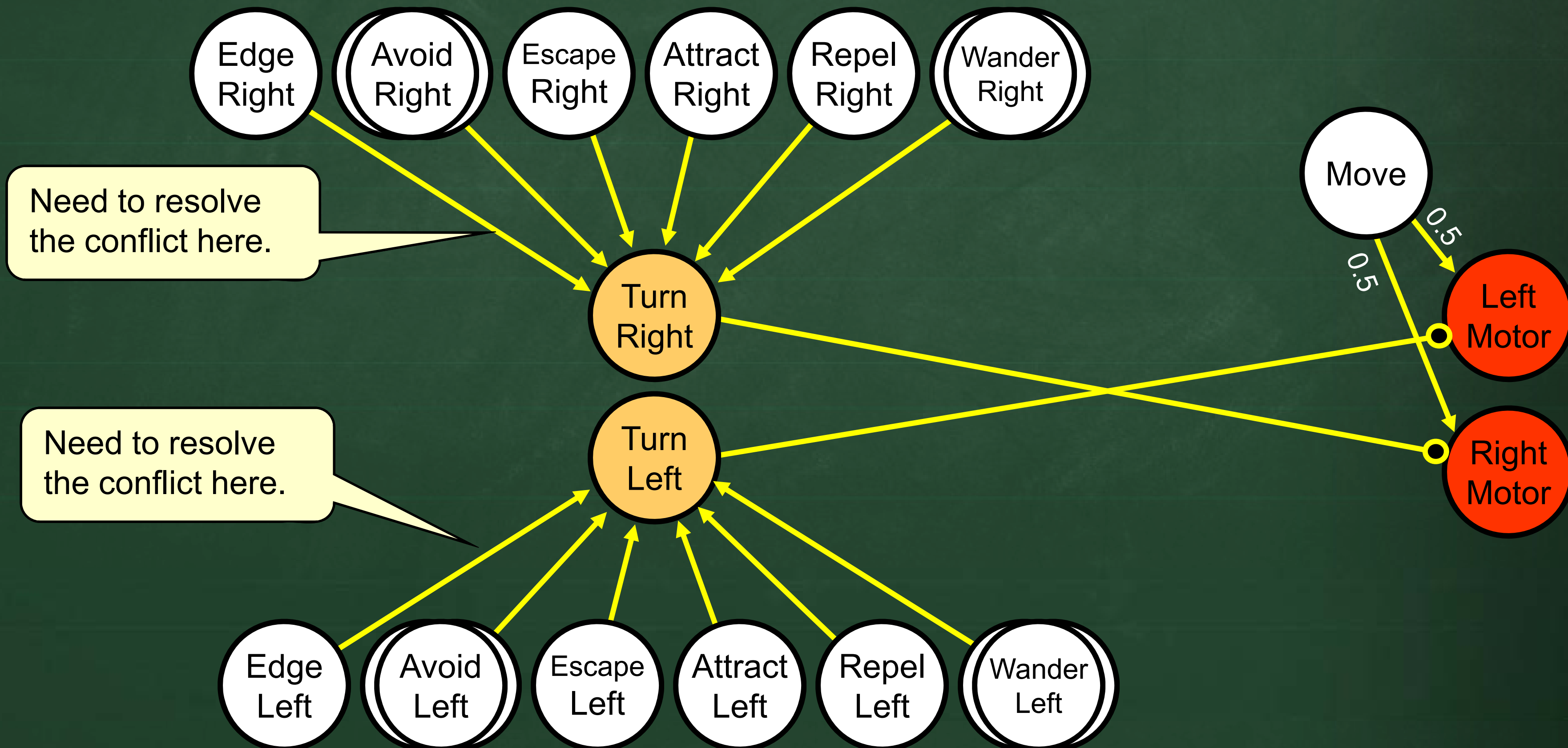
- Here is the completed edge-following network:



- A similar network can be constructed to follow edges on the left side of the robot.

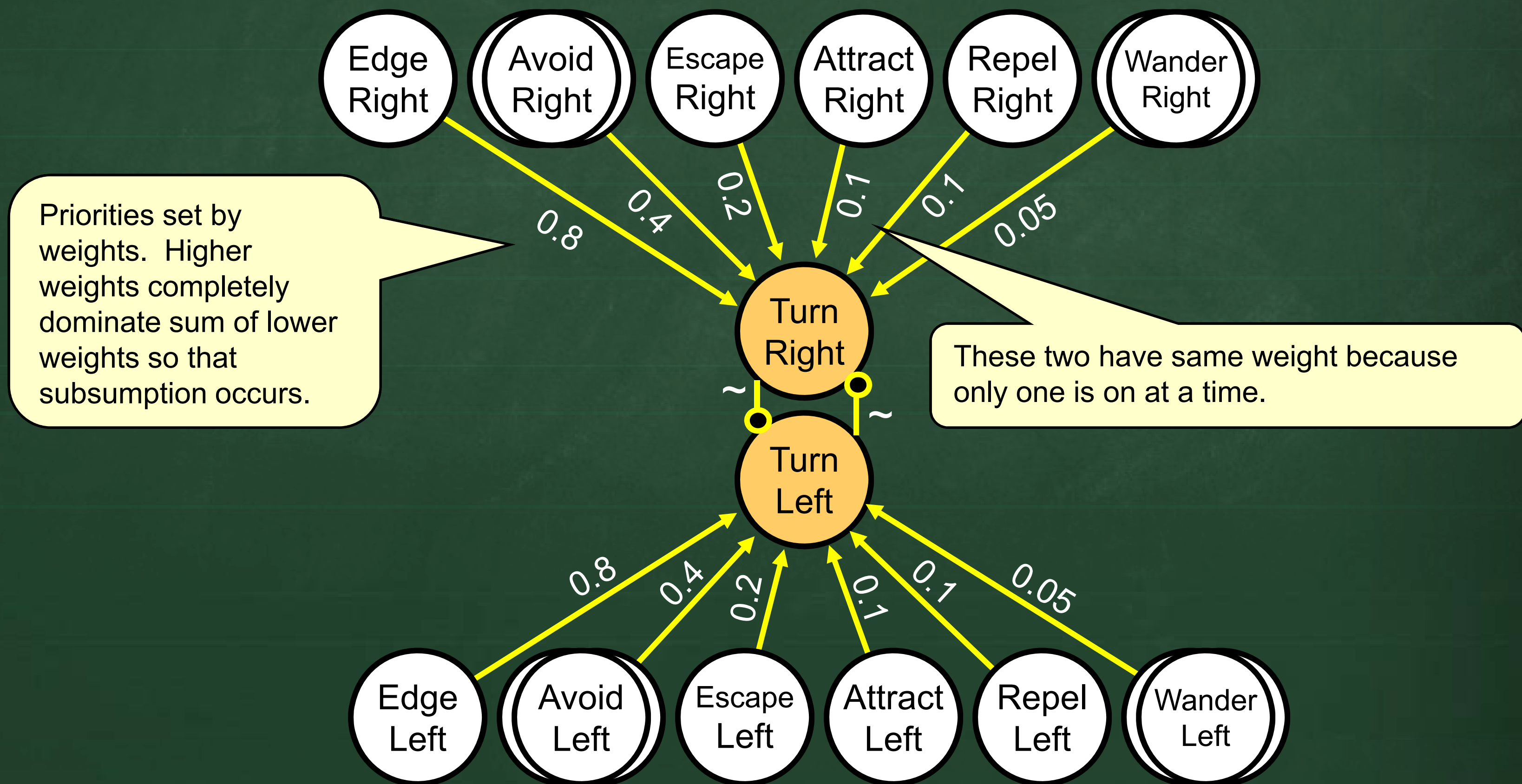
Neuron Nets – Arbitration

- Notice that all networks access the motors.
  - Must arbitrate to decide which one has motor control



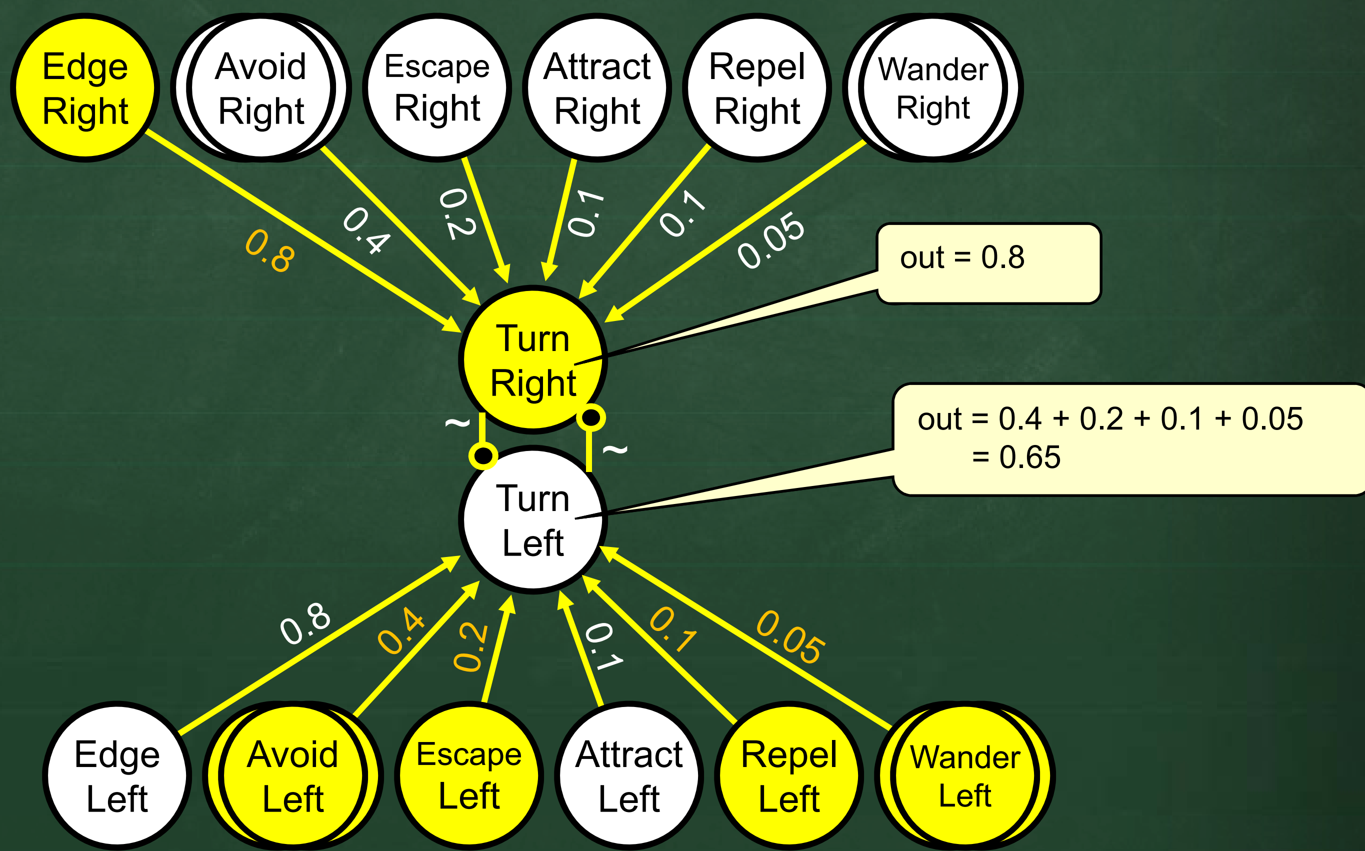
Neuron Nets – Arbitration

- Must assign weights so that more important behaviors override the less important ones:



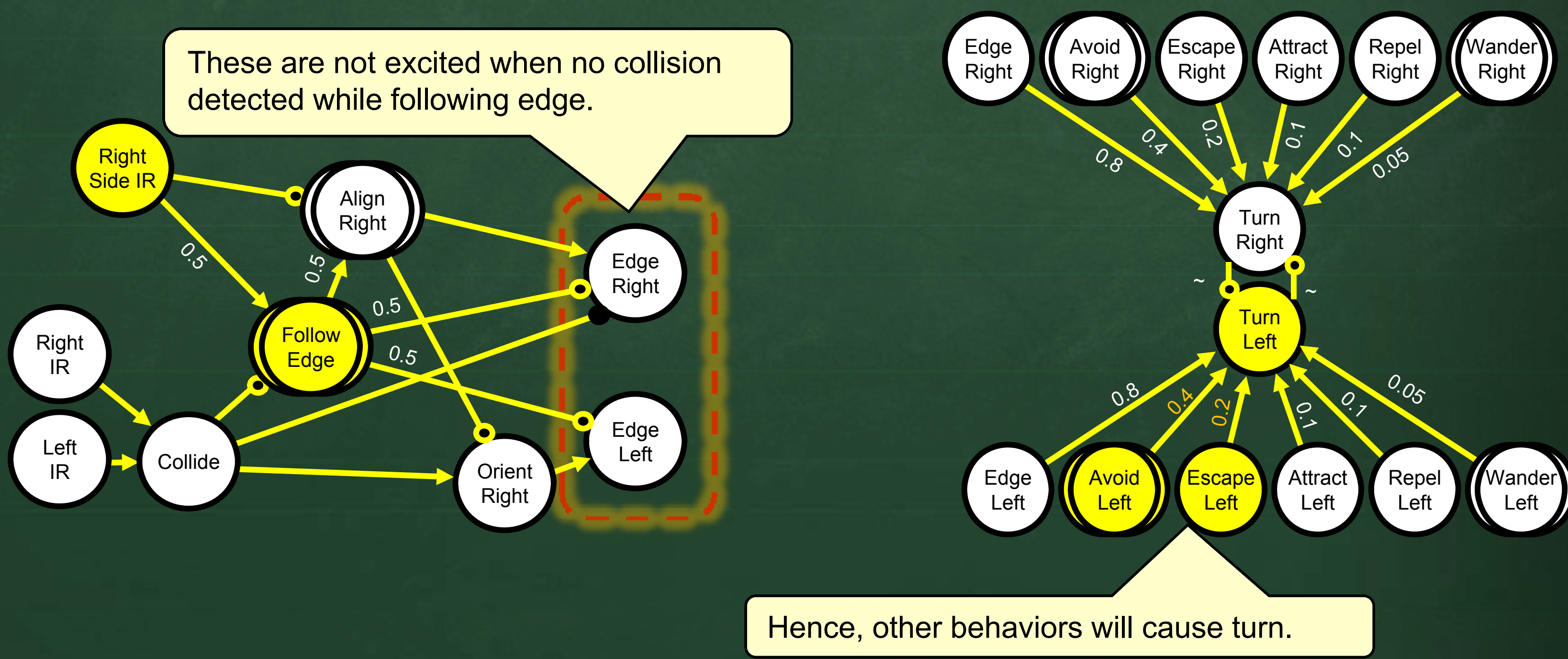
Neuron Nets – Arbitration

- Here is an example of how edge following can override the other behaviors:



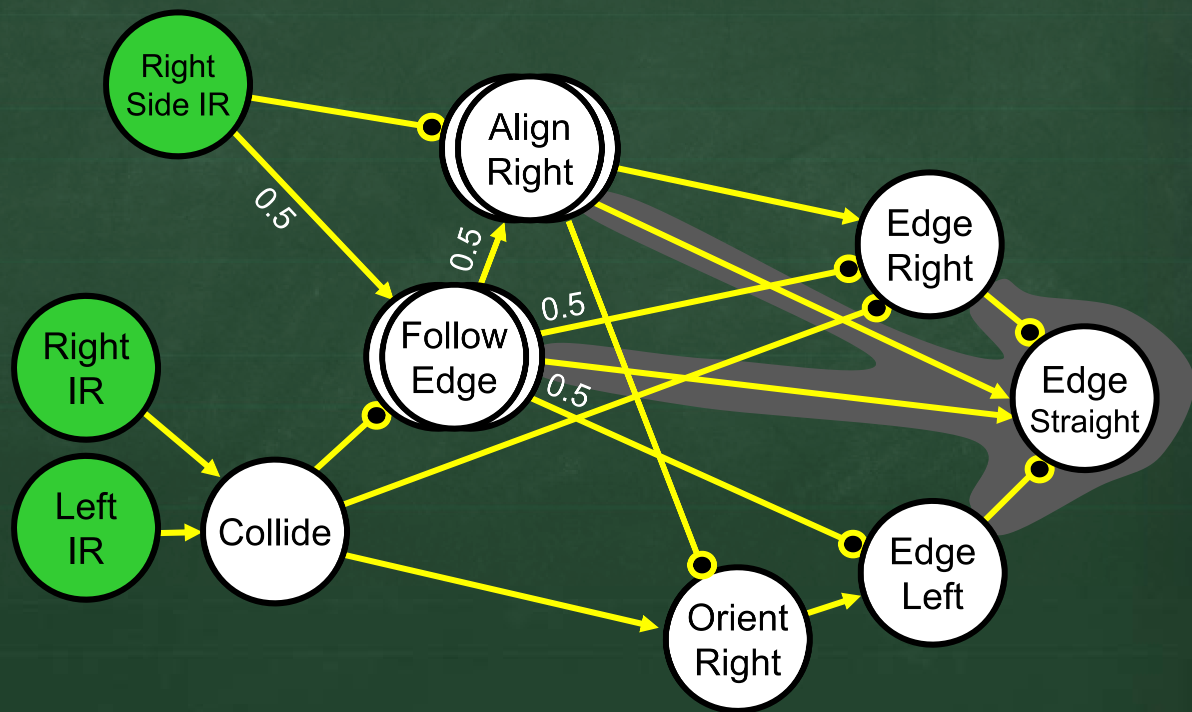
Neuron Nets – Arbitration

- Arbitration problem with edge following:
  - If the robot is following edge and wants to move forward, neither Turn neurons are excited, which allows other behaviors to take control of the steering.



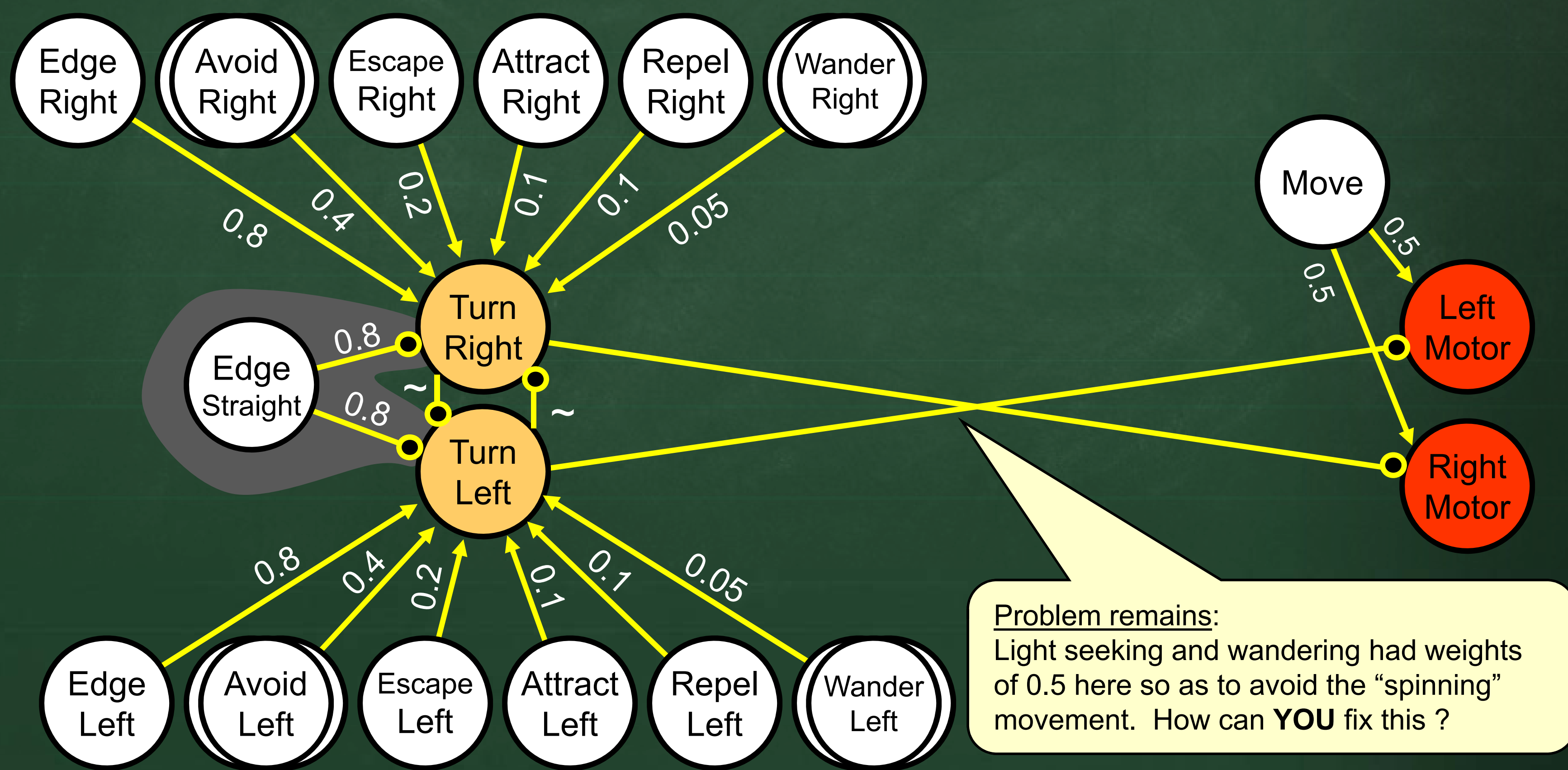
Neuron Nets – Arbitration

- Must fix this by allowing a "go straight ahead" neuron in the edge following network that disables turning:



Neuron Nets – Arbitration

- The new *Edge Straight* neuron must have the same weight as the *Edge Left* and *Edge Right*:



Summary

- You should now understand:
  - What behaviors are and how they interact together
  - How to program simple behaviors
  - The ideas behind learning behaviors
  - How to program behaviors using neuron networks