Behavior Overview

Carnegie Mellon University

October 13, 2003

Scott Lenser
Manuela Veloso

Behaviors Approaches

There are 4 main approaches to behaviors:

- Reactive — Try to respond directly to the environment.
- Planning — Think about everything you need to do to achieve your goal then do it.
- Deliberative — General category where you think ahead some about what you need to do, includes planning as a special case.
- Hybrid — Combine a reactive and a deliberative method.

Reactive Behaviors

- Reactive behaviors map directly from sensations to actions to take.
- In other words, reactive behaviors have no memory.
- Pluses:
  - Very responsive to changes in environment.
  - Simple and easy to understand.
  - Produces good results for simple tasks.
  - Produces smooth control changes in response to smooth sensation changes.
- Minuses:
  - Can't perform different actions from the same state.
  - Gets stuck easily.
  - Doesn't scale to complex tasks well.

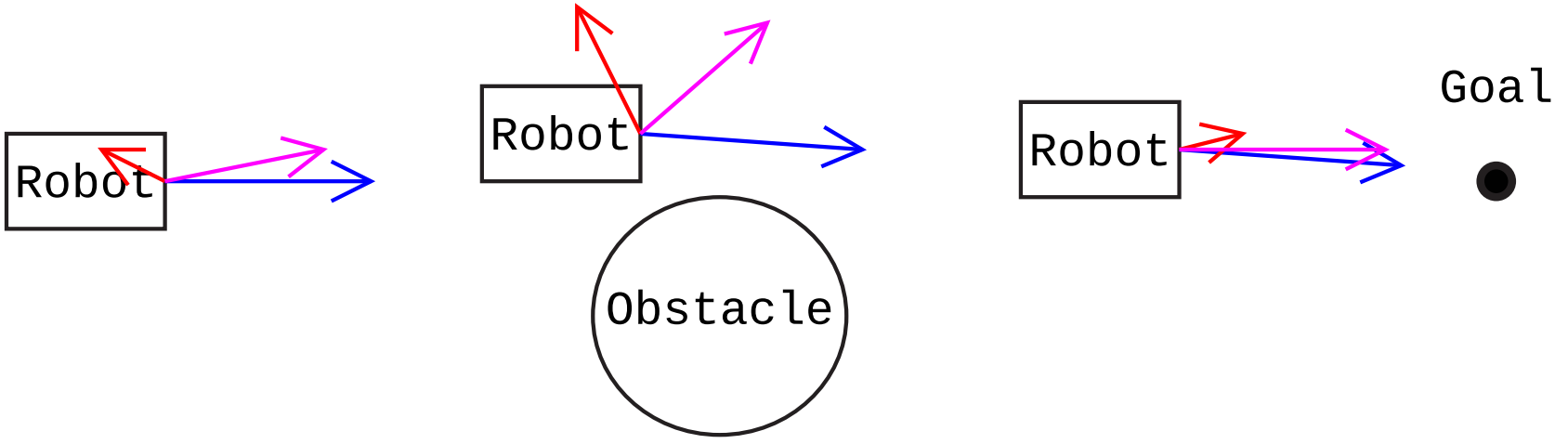
Reactive Behaviors

- Reactive behavior systems come under a wide variety of names.
- Despite the various names, reactive behavior systems largely behave in the same general way.
- Reactive behaviors form the basic building block of almost every successful behavior system.
- Two of the most influential reactive behavior systems are:
  - Motor schemas
  - Potential fields

Motor Schemas

- A motor schema is a mapping from sensations (usually robot position) to a force vector (usually for where the robot should go next).
- Each motor schema calculates a "force" on the robot due to some constraint of the problem.
- These forces are then summed to get the total force on the robot. The robot moves in the direction indicated by the force.
- Canonical example of robot trying to get to a goal while avoiding obstacles:
  - One motor schema produces a force in the direction of the goal.
  - Second motor schema produces a force away from obstacles.

Motor Schemas



Goal vector

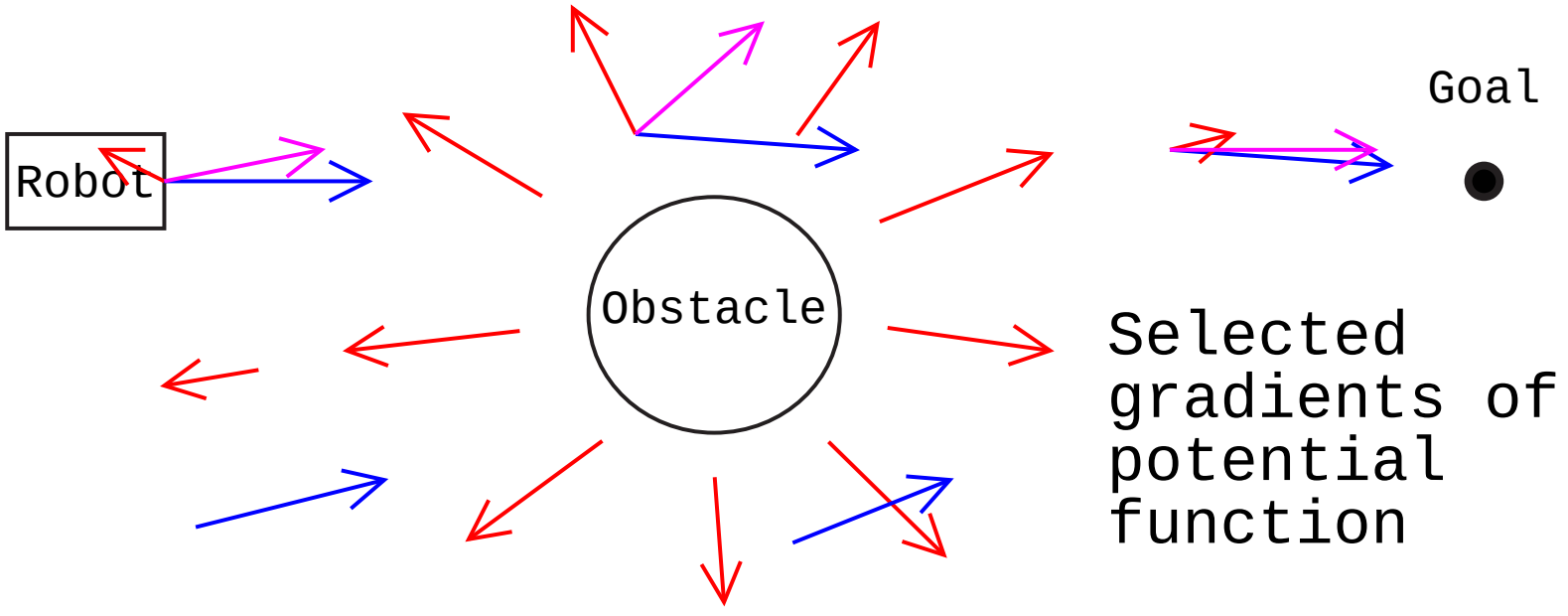
Avoidance vector

Resulting vector
(direction of robot movement)

Potential Fields

- Usually used for robot navigation.
- Idea is to associate an energy with every point in space.
- Robot moves from areas with high energy to areas with lower energy.

Potential Fields



Combining Reactive Behaviors

- Remember, reactive behaviors don't scale well.
- We need to find a way to combine reactive behaviors into a larger behavior system.
- Three ideas (other methods possible):
  - Competition - reactive behaviors compete for control of the robot.
  - Subsumption - reactive behaviors selectively take control of the robot.
  - Sequencing - reactive behaviors are sequenced by a higher level controller.

Competition

- Reactive behaviors compete for control of the robot.
- Very responsive to change in environment.
- Can adapt to different sets of available behaviors naturally.
- Tends to have massive problems committing to an action.
- Oscillation problems swamp nice properties.

Subsumption

- Idea is roughly to order the behaviors.
- All behaviors executed in parallel.
- Behaviors higher up can override output of lower behaviors.
- Similar to way `run.cfg` orders behaviors by priority.
- Improvement in scalability but we can do better.

Sequencing

- Basic idea is to run only one reactive behavior at a time and switch the active behavior from time to time.
- Convenient method for sequencing the reactive behaviors is to use a finite state machine (FSM).
- Each state of the FSM has an associated reactive behavior.
- Each transition of the FSM has a transition rule that must be sastified before the transition can happen.

Sequencing Advantages

- Problems with oscillation are greatly reduced because transitions are gated by transition rules. Oscillation can only occur by switching rapidly between states.
- Reactive to environment.
- Can select different actions from same perceptual state using context.
- Easy to create macro actions.
- Easy to chain together actions into larger behavior.
- This is the approach we will concentrate on as we have found it to work the best in practice.

Hierarchy - Adding Scale

- In order to scale to larger problems, we would like to be able to reuse collections of reactive behaviors that together perform a task.
- We can simply allow each state of FSM to either be a reactive behavior or a collection of behaviors controlled by its own FSM.
- Adds a little complexity as subordinate FSM needs to know when it is switched from active to inactive so that it can appropriately switch states as needed.
- Can also allow reactive of FSM behaviors to take parameters that control details of the actions they perform. For instance, a speed for going to a point or an object id for finding an object.

Implementing Sequencing

- Sequencing is handled by a FSM.
- FSM is implemented by FiniteStateMachine class defined in agent/Behaviors/state_machine.h.
- This is a template class which needs a state type (usually an enumeration listing the possible states) and a timestamp type (usually an unsigned long).
- This class manages the transitions of the FSM, keeps track of a large variety of statistics about the FSM, and detects infinite loops in the FSM.

FiniteStateMachine methods

init() Initializes the state machine and sets the starting state. max_transitions is the maximum number of transitions out of any state. _event_cache_size is the maximum number of transitions that can happen in one decision cycle (one frame).

startLoop() Must be called before the start of each decision cycle.

endLoop() Must be called after the end of each decision cycle.

FiniteStateMachine methods

TRANS_CONT() Macro which transitions to the given state and continues the decision loop. The transition must be unique amongst all transitions out of this state.

timeInState() Returns the time spent in the current state since the last transition.

isNewState() Returns true if we just entered this state. Useful for any initialization required in a state.

setState() Function which sets the current state of the state machine from outside of the decision loop.

sleep() Indicates that the state machine is currently not in use. This affects the collection of statistics.

FiniteStateMachine methods

handleErr() Used when an infinite loop or other error is detected to indicate a problem. An infinite loop is detected by seeing the same transition from the same state in a single decision cycle. By default plays a short sound. By uncommenting the dance line in this function, problems become VERY obvious.

dumpLoopEvents() Prints out the states and transitions used during the last decision loop. Very useful for finding infinite loop bugs and hence printed out by default when infinite loops occur.

dumpStats() Prints out various statistics about the state machine such as time spent in each state, state/transition percentages, and transition counts. Can be very useful for debugging.

Behaviors - Two Kinds

- There are 2 basic kinds of behaviors.
- IndependentBehaviors can generate motion commands from the sensor readings with no extra input.
- ChaseBall is an example of an independent behavior.
- Subordinate behaviors can generate motion commands from the sensor readings and some control parameters.
- Goto point is an example of a subordinate behavior.
- Independent behaviors can also be used as subordinate behaviors (but not vice versa).

Behaviors Entry Points

- All behaviors should have an operator() defined. This is called when the behavior is used as a subordinate behavior.

- IndependentBehaviors must implement the EventProcessor interface which uses update and get to get the output of the behavior.

Behaviors Design

- Behavior design is more of an art form than a science.
- Good behaviors produce smoothly varying control signals.
- Control signals that oscillate or otherwise jump around lead to poor control performance. This occurs because the control target changes before the controller can achieve the previous target.
- Oscillation amongst behaviors needs to be avoided because it leads to oscillatory control signals.

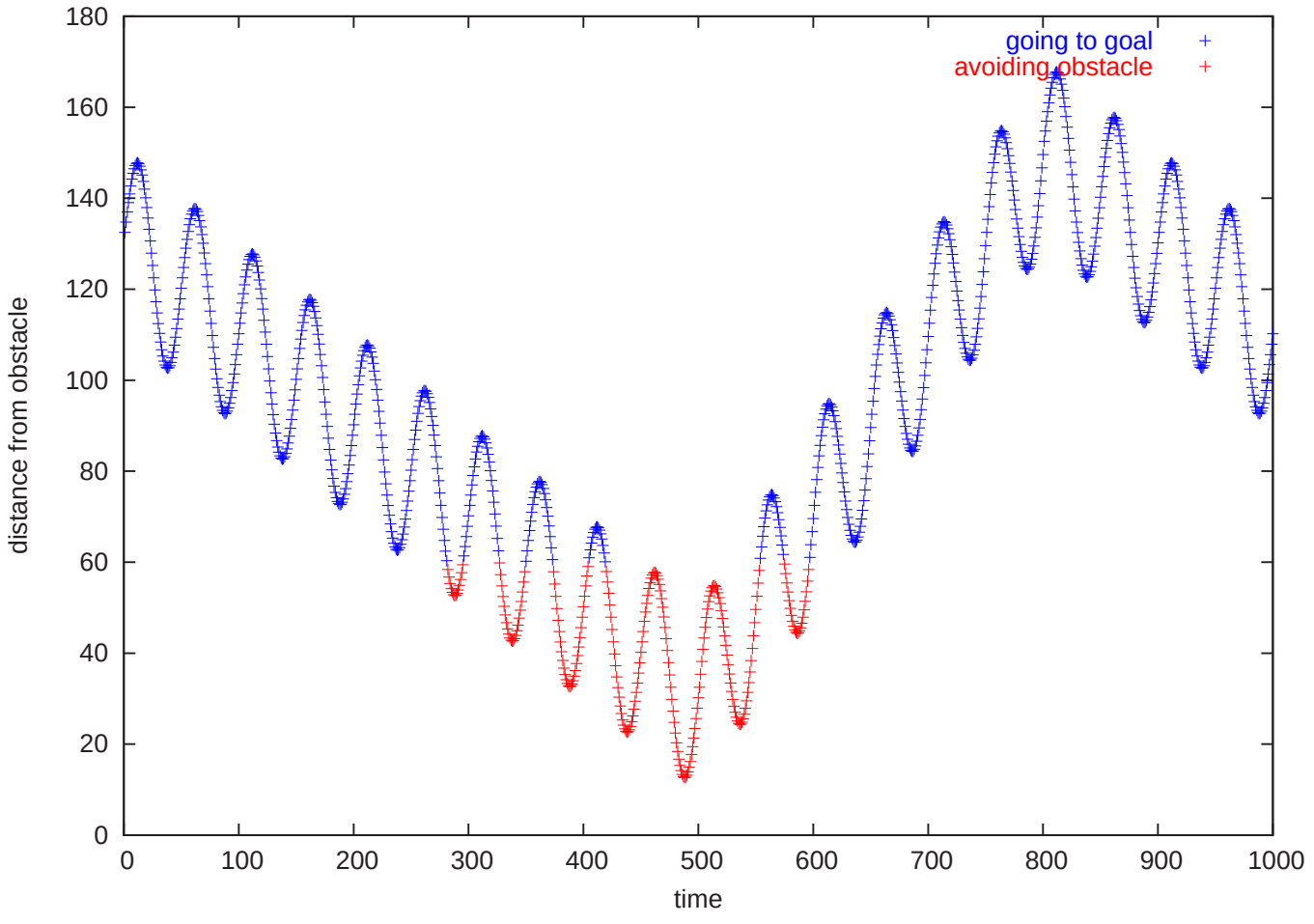
Avoiding Oscillation

- Oscillation can occur any time there is a transition path among a set of states.
- It usually happens between small number of states, usually 2, sometimes 3.
- There are two basic ways to reduce oscillation:
  - Merge 2 similar states together that there might otherwise be oscillation between.
  - Add hysteresis to the transition rules.
- Oscillation can also occur if there are states where the robot fails to make progress towards the goal.

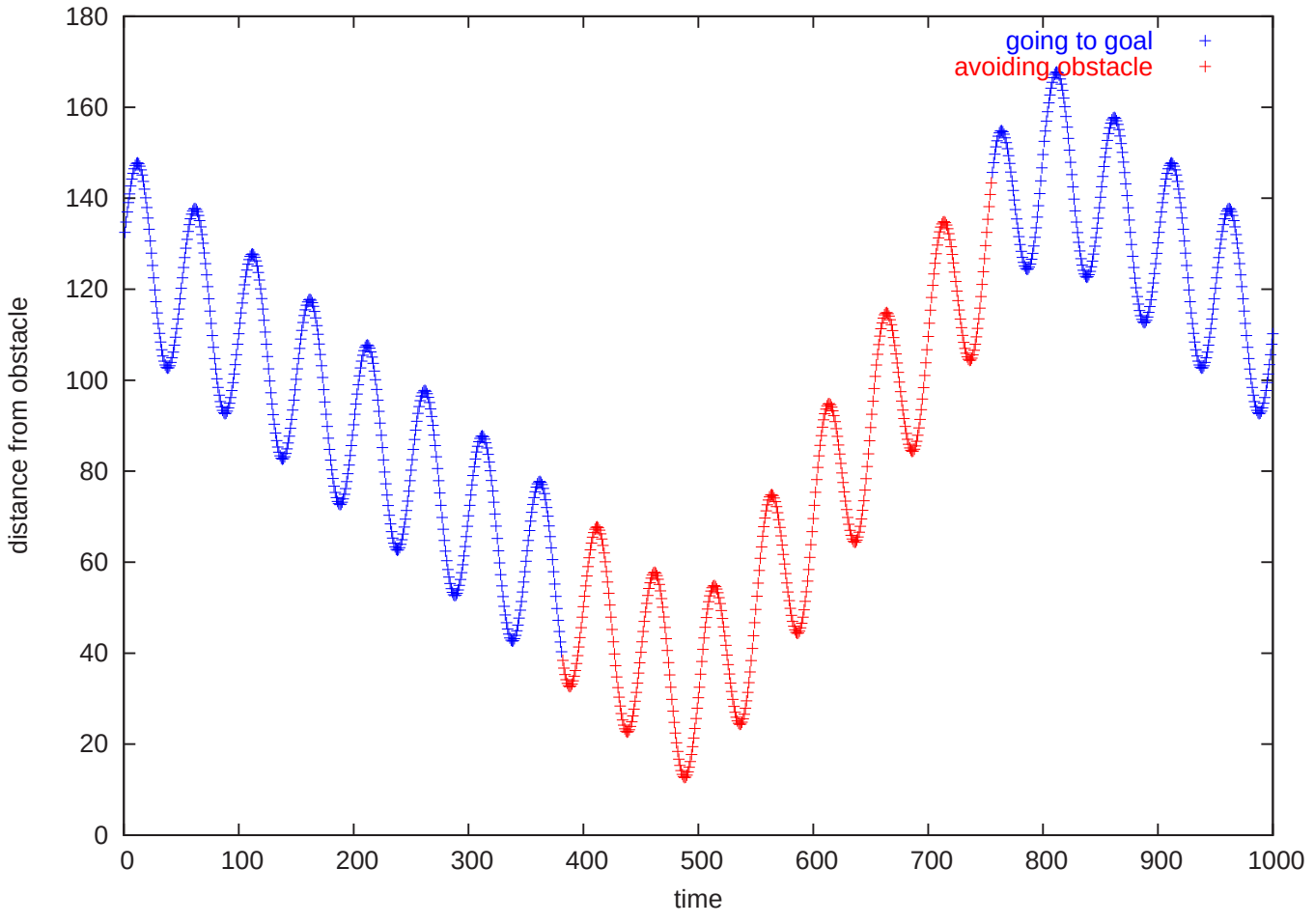
Hysteresis

- A system is said to exhibit hysteresis when the behavior of the system depends not only on its current state, but also on its history.
- In the context of behaviors, hysteresis refers to the creation of a buffer zone between two states.
- Within the buffer zone, the robot simply uses whatever state it was using when it entered the buffer zone.
- This is sometimes called a dual threshold because there are 2 thresholds involved instead of one.

Hysteresis


going to goal
avoiding obstacle
distance from obstacle
time
0
20
40
60
80
100
120
140
160
180
0
100
200
300
400
500
600
700
800
900
1000

Hysteresis


going to goal
avoiding obstacle
distance from obstacle
time
0
20
40
60
80
100
120
140
160
180
0
100
200
300
400
500
600
700
800
900
1000

Behavior Design Principles

- Design the behavior in stages. Only work on one stage of the behavior at a time.
- Start with the state the robot should execute first from the position you expect to start from. Working on later stages first tends to fail because the later states depend on the way in which the earlier states transition to them.
- As each state is completed, add the next state to be executed in the chain.
- Make one change at a time. The extra time spent compiling/booting will be saved by the not wasting time trying to figure out which change you just made messed things up.