Cognitive Robotics

I.e., How to create “thinking” robots

Prof. Brian Williams and Steve Levine (TA)

Introduction

February 3rd, 2015.
Today’s Assignments

Problems Sets:
• None! Problem Set #1 out next Monday, due in 1½ weeks.

Readings:

Note:
• Problem sets, readings and lecture slides posted on 16.412J course site.
Outline

• Course in a Nutshell

• Course Structure and Logistics

• Programming Cognitive Robots

• Self-Adaptive and Self-Repairing Systems

• Programs that Monitor State
Coordinating Network Embedded Systems

- We are creating vast networks of embedded systems that perform critical functions over long periods of time.

- These long-lived systems achieve robustness by coordinating a complex network of devices.

- Programming these systems robustly is becoming an increasingly daunting task.
00:00 Go to \(x_1, y_1\)
00:20 Go to \(x_2, y_2\)
00:40 Go to \(x_3, y_3\)
...
04:10 Go to \(x_n, y_n\)

Command script

Plant
“Put out the fires at Locations 1 and 2, and return to the base in an hour. Avoid no fly zones. Here is a map, including reservoirs and gas stations.”

Qualitative State Plan

Model-based Executive

Observations

Commands

Plant

[Leaute & Williams, AAAI 05]
Robustness through Collaboration
Examples of Cognitive Robotic Systems

- Information-gathering: Droids from Star Wars
- Human-Robot Collaboration: HAL 9000
- Grids for Manufacturing: Skynet
- Cooperative Information Scouts

© sources unknown. All rights reserved. This content is excluded from our Creative Commons license. For more information, see [https://ocw.mit.edu/help/faq-fair-use/].
Programming Cognitive Systems

1. Embedded programming languages elevated to the goal-level through partial specification and operations on state (RMPL).

2. Language executives that achieve robustness by reasoning over constraint-based models and by bounding risk (Enterprise).

3. Interfaces to support human interaction fluidly and at the cognitive level (Uhura, Pike ...).
Outline

• Course in a Nutshell

• **Course Structure and Logistics**

• Programming Cognitive Robots

• Self-Adaptive and Self-Repairing Systems

• Programs that Monitor State
Crash Course in Autonomy
I.e., Programming Cognitive Robots

Prof. Brian Williams & Eric Timmons

Introduction
January 12, 2015.
How to program cognitive robots in terms of goals, to perform complex tasks.

Intuitions underlying how robots “reason.”

Exposure to basic computational concepts.

Driven by a Grand Challenge
Embedded in simulations and hardware

Working out-of-the-box with your ‘15 holiday gift
16:412J: Driven by a Grand Challenge

RobOrienteering

© British Orienteering. All rights reserved. This content is excluded from our Creative Commons license. For more information, see https://ocw.mit.edu/help/faq-fair-use/.
16:412J: Embedded in simulations and hardware

Works with a lot of elbow grease and a bit of luck
Traditional Programming

“Cognitive” Programming

Programs that Monitor State

Programs with Flexible Time

Programs with Goal States

Programs with Continuous State

Programs that Collaborate

Advanced lectures...

Risk-bounded Programming

Grand Challenge

Reactive Model-based Programming Language (RMPL)

+ Python

TODAY

TIME

END OF SEMESTER

2/3/2016

Cognitive Robotics-Introduction
You’ll walk out knowing how to

• program cognitive robots by specifying goals and models
• Research, describe, and implement decision-making algorithms that enable robots to monitor, plan and coordinate in the real world
• implement complex missions with real robot.
Be warned – bleeding edge!

A clipart of blooded knife removed due to copyright restrictions.
Policies, grading, and assignments

COURSE LOGISTICS
Websites & emails

• Stellar website

• Piazza (for course-related discussion)

• Staff email list
Lectures & Office Hours

- Students must attend lectures
  - *Vital* to learning course material
  - Plan on attending all lectures
  - Expected to do assigned readings before lecture

- Randomly-scheduled 5-minute mini quizzes
  - Not difficult / stressful
Assignments: Problem sets

- Modeling exercises
- Using existing autonomy tools
- Implementing algorithms (Python)
Assignments: Grand Challenge

• Course culminates in grand challenge!
  – Orienteering theme
  – Simulation & real hardware
Assignments: Advanced Lectures

• What it’s like doing research in autonomy

• Teams of 5-6 will:
  – Present full 80 minute lecture on researched topic
  – Implement topic (Python)
  – Release code, API, and tutorial / documentation for class
    • Will be used in grand challenge
Electronics use policy in lecture

Computers, tablets: *for note taking only!*

Please do not:

– Check email or facebook, surf web, watch adorable cat videos 😼, etc.

Research shows it also *distracts others* nearby
## Grading

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participation &amp; attendance (mini quizzes)</td>
<td>10%</td>
</tr>
<tr>
<td>Problem sets</td>
<td>40%</td>
</tr>
<tr>
<td>Advanced lecture &amp; implementation</td>
<td>30%</td>
</tr>
<tr>
<td>Grand challenge</td>
<td>20%</td>
</tr>
</tbody>
</table>

* Staff reserves the right to consider other factors & adjust formula*
Collaboration Policy

• Collaboration allowed, such that you:
  – Acknowledge collaborators
  – Involved in all aspects of work (no dividing up)
  – Write your own solutions

• Advanced lectures & grand challenge
  – Working in teams, expect equal contributions

• Course bibles prohibited
TA: Steve Levine

4\textsuperscript{th} year Ph.D student in CS (MERS group)

Research:
Intent recognition & adaptation for robots

B.S. from MIT in ‘11, course 6
M.Eng from MIT in ‘12, course 6
9\textsuperscript{th} year (!) at MIT! So old!
Outline

• Course in a Nutshell

• Logistics

• Programming Cognitive Robots

• Self-Adaptive and Self-Repairing Systems

• Programs that Monitor State
Programs on State - Firefighting Scenario
Firefighting in RMPL: Traditioanl Imperative program

class Main{
    UAV uav1;
    Lake lake1;
    Lake lake2;
    Fire fire1;
    Fire fire2;
    ...

    method run() {
        sequence{
            uav1.takeoff();
            uav1.fly_base1_to_lake1();
            uav1.load_water(lake1);
            uav1.fly_lake1_to_fire1();
            uav1.drop_water_high_altitute(fire1);
            uav1.fly_fire1_to_lake1();
            uav1.load_water(lake1);
            uav1.fly_lake1_to_fire1();
            uav1.drop_water_low_altitute(fire1);
            uav1.fly_fire1_to_lake2();
            uav1.load_water(lake2);
            uav1.fly_lake2_to_fire2();
            uav1.drop_water_high_altitute(fire2);
            uav1.fly_fire2_to_lake2();
            uav1.load_water(lake2);
            uav1.fly_lake2_to_fire2();
            uav1.drop_water_low_altitute(fire2);
            uav1.fly_fire2_to_base1();
            uav1.land();
        }
    }
}
Firefighting in RMPL: A Program on State

class Main{
    UAV uav1;
    Lake lake1;
    Lake lake2;
    Fire fire1;
    Fire fire2;
    ...

    method run() {
        sequence{
            (fire1 == out);
            (fire2 == out);
            (uav1.flying == no &&
                uav1.location == base_1_location);
        }
    }
}
}
class Main {
    UAV uav1;
    Lake lake1;
    Lake lake2;
    Fire fire1;
    Fire fire2;
    ...

    method run() {
        sequence{
            (fire1 == out);
            (fire2 == out);
            (uav1.flying == no &&
                uav1.location == base_1_location);
        }
    }
}
Firefighting in RMPL: Model of Actions

class UAV {
    Roadmap location;
    Boolean flying;
    Boolean loaded;

    primitive method takeoff()
        flying == no => flying == yes;

    primitive method land()
        flying == yes => flying == no;

    primitive method load_water(Lake lakespot)
        ((flying == yes) && (loaded == no) && (lakespot.location == location)) => loaded == yes;

    primitive method drop_water_high_altitude(Fire firespot)
        ((flying == yes) && (loaded == yes) && (firespot.location == location) && (firespot == high))
        => ((loaded == no) && (firespot == medium));

    primitive method drop_water_low_altitude(Fire firespot)
        ((flying == yes) && (loaded == yes) && (firespot.location == location) && (firespot == medium))
        => ((loaded == no) && (firespot == out));

    #MOTION_PRIMITIVES(location, fly, flying==yes)
}

class Lake {
    Roadmap location;
}

class Fire {
    initial value high;
    value medium;
    value out;
    Roadmap location;
}
Firefighting in RMPL: Model of Actions

class UAV {
    Roadmap location;
    Boolean flying;
    Boolean loaded;

    primitive method takeoff()
    flying == no => flying == yes;

    primitive method land()
    flying == yes => flying == no;

    primitive method load_water(Lake lakespot)
    ((flying == yes) && (loaded == no) && (lakespot.location == location)) => loaded == yes;

    primitive method drop_water_high_altitude(Fire firespot)
    ((flying == yes) && (loaded == yes) && (firespot.location == location) && (firespot == high))
    => ((loaded == no) && (firespot == medium));

    primitive method drop_water_low_altitude(Fire firespot)
    (flying == yes) && (loaded == yes) && (firespot.location == location) && (firespot == medium))
    => ((loaded == no) && (firespot == out));

    #MOTION_PRIMITIVES(location, fly, flying==yes)
}

class Lake {
    Roadmap location;
}

class Fire{
    initial value high;
    value medium;
    value out;
    Roadmap location;
}
Simulation Testing: Firefighting Scenario
Decision-making algorithm: Activity Planning

\[ f = g + h \]

Solve Relaxed Planning Problem
Outline

• Course in a Nutshell

• Logistics

• Programming Cognitive Robots

• Self-Adaptive and Self-Repairing Systems

• Programs that Monitor State
“Autonomous” vehicles explore far away places .. but often end in disaster!
What you should really learn from immune systems about adversarial design

Jacob Beal

NDIST
December, 2015

 Courtesy of Jacob Beal. Used with permission.
The Immune System
(as noticed by computer scientists)

Complements of Jake Beal

(Source: the Human Immune Response System www.uta.edu/chagas/images/immunSys.jpg)
But there’s more to the immune system...

- Physical barriers
- Inhospitable environments
- Tolerance
- Death

Complements of Jake Beal
Physical Barriers

- Skin, gastrointestinal wall, blood/brain barrier
- Saliva, tears, mucus (nose, lungs, gut, . . .)
- Defense by discardng
Inhospitable environments

- Fever: high temperature inhibits bacterial growth
- Enzymes in saliva, tears, nasal secretions, perspiration, milk, sperm
- ...
- Skin is acidic
Tolerance

- Follicular mites
- Parasitic worms
- Dead viruses in the genome
- Microbiome
  - Gut commensals
  - Flora for various cavities

>3000 cryptic drug-like molecule gene clusters!

*(Fischbach, Science, 2015)*

**Human follicular mite**

*Complements of Jake Beal*
Death

- Cell self-inhibition, suicide
- Reproductive barriers for parasites
Model-based Programs Offer Layers of Defense

Languages that achieve robustness through decision layers that are:

- **Suspicious**
  - Monitor states and goals.
- **Adapt to disturbance**
  - Adjust timing
  - Select contingencies
- **State Aware**
  - Plan to achieve goals states.
- **Precise**
  - Achieve continuous goal states.
- **Collaborate**
  - Executes programs, revise goals and plan with humans.
- **Manage Risk**
  - Executes programs in uncertain environments with bounded risk.
A single “cognitive system” language and executive.

- **User**
  - Kirk
  - Burton
  - Pike
  - Sulu
  - Bones

**Enterprise**
- Uhura: Collaboratively resolves goal failures
- Kirk: Sketches mission and assigns tasks
- Burton: Plans actions
- Pike: Coordinates and monitors tasks
- Sulu: Plans paths
- Bones: Diagnoses likely failures

© source unknown. All rights reserved. This content is excluded from our Creative Commons license. For more information, see [https://ocw.mit.edu/help/faq-fair-use/](https://ocw.mit.edu/help/faq-fair-use/).
Outline

- Course in a Nutshell
- Logistics
- Programming Cognitive Robots
- Self-Adaptive and Self-Repairing Systems
- Programs that Monitor State
When things go wrong...
Execution monitoring: detecting problems

• Something unexpected things happen
  – Not modeled in plan!
  – How to react?

• **Execution monitoring**: detecting when things go wrong

• **Replanning**: fixing the problem (later in this course)
Volcano eruption!

Program sequence of actions in RMPL

```java
method run() {
    sequence {
        uav.launch();
        uav.fly_to_base_station();
        uav.pick_up_med_kit();
        uav.fly_to_hikers();
        uav.drop_off_med_kit();
    }
}
```

Actions have preconditions & effects like before
Volcano eruption!

Base Station

Launch Q

Q fly to Base Station

Q pick up med kit

Q fly to hikers

Q give med kit to climbers

What could possibly go wrong??
Preconditions & effects of actions

Launch Q

Q fly to Base Station

Q pick up med kit

Q fly to hikers

Q give med kit to climbers

time
Preconditions & effects of actions

Q pick up med kit

time
Preconditions & effects of actions

- Q near med kit
- Q has empty cargo bay
- Q has med kit!
- Q’s cargo bay full

Time

Q pick up med kit
Preconditions & effects of actions

- Q has a med kit
- Q near climbers

Q give med kit to climbers
Preconditions & effects of actions

- **Launch Q**
- **Q fly to Base Station**
- **Q pick up med kit**
- **Q fly to hikers**
- **Q give supplies to climbers**

Q has med kit
Where do preconditions come from?

Launch Q

Q fly to Base Station

Q pick up med kit

Q fly to hikers

Q give supplies to climbers

Q has med kit

Q near climbers

Time
Where do preconditions come from?

Launch Q → Q fly to Base Station → Q pick up med kit → Q fly to hikers → Q give supplies to climbers

Q in the air → Q has med kit → Q near climbers

Where do preconditions come from?
Where do preconditions come from?

Launch Q

Q fly to Base Station

Q in the air

Q has med kit

Q pick up med kit

Q near climbers

Q fly to hikers

Q give supplies to climbers

Q in the air

time
Where do preconditions come from?

- Launch Q
- Q fly to Base Station
- Q pick up med kit
- Q fly to hikers
- Q give supplies to climbers

Where do preconditions come from?
Where do preconditions come from?

- Launch Q
- Q fly to Base Station
- Q pick up med kit
- Q fly to hikers
- Q give supplies to climbers

Preconditions:
- Q has med kit
- Q in the air
- Q near med kit
- Q in the air
- Q near climbers
- Q has empty cargo bay
These are called “causal links”

Causal link: one action produces something needed by a later action
What happens if something goes wrong?

- Launch Q
- Q fly to Base Station
- Q pick up med kit
- Q fly to hikers
- Q give supplies to climbers

Q has med kit

Q drops med kit!
What happens if something goes wrong?

Launch Q
Q fly to Base Station
Q pick up med kit
Q fly to hikers
Q give supplies to climbers

Q has med kit

Q drops med kit!
Causal links allow execution monitoring

If condition is violated during causal link, signal failure.
Causal links for execution monitoring

• Causal links tell you what needs to hold, and when

• Tells you what’s *relevant* to the plan

• Can be used offline, for error checking
How do we do execution monitoring?

We need:
1. Sensing.
2. Action model.
3. Plan.
1. Sensing

“Q has medical kit”
2. Action model

- Q near med kit
- Q has empty cargo bay
- Q has med kit!
- Q’s cargo bay full

Q pick up med kit

Time
3. Plan

Launch Q
Q fly to Base Station
Q pick up med kit
Q fly to hikers
Q give med kit to climbers

time
How do we do execution monitoring?

Offline:
• Use **plan & action** model to extract causal links

Online:
• Continuously **sense** monitor those causal links
Key takeaways

• Need to monitor the plan when executing

• Causal links:
  – What needs to be true, and when
  – Offline (checking) and online (monitoring)
A single “cognitive system” language and executive.

User

Goals & models in RMPL

Control Commands

Enterprise

Uhura
Collaboratively resolves goal failures

Kirk
Sketches mission and assigns tasks

Burton
Plans actions

Pike
Coordinates and monitors tasks

Sulu
Plans paths

Bones
Diagnoses likely failures

© source unknown. All rights reserved. This content is excluded from our Creative Commons license. For more information, see https://ocw.mit.edu/help/faq-fair-use/.
Causal link extraction algorithm

Extraction (non-temporal, totally-ordered plan - offline):
for each action in plan:
    for each precondition p of action:
        a_p = latest action in plan with effect p
        Add causal link: a_p to action over p

Monitoring (online):
while True:
    cls = currently active causal links (based on actions)
    state = measure state with sensors
    for each causal link cl in cls:
        p = predicate associated with cl
        if not(p in state):
            Trigger execution monitoring exception!