CS5100 Foundations of Artificial Intelligence

Module 04 Lesson 6

Introduction to Prolog

Some images and slides are used from CS188 UC Berkeley/AIMA with permission
All materials available at http://ai.berkely.edu / http://aima.cs.berkeley.edu

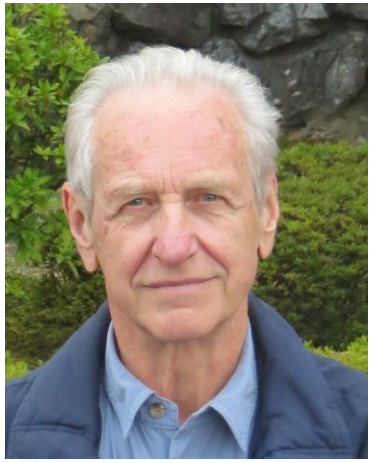
Overview

Quick overview
Of Prolog

Examples

History

- Prolog: Programming in Logic
- Declarative programming language
- Created around 1972
  by Alain Colmerauer and
  Robert (Bob) Kowalski



Logic programming: Prolog .. 1

- Algorithm = Logic + Control
- Basis: Definite clauses + bells & whistles

  Widely used in Europe, Japan (basis of 5th Generation project)

  Compilation techniques $\Rightarrow$ 60 million LIPS (logical inferences per second)

Logic programming: Prolog .. 2

Program = set of clauses

head :- $literal_1$, … $literal_n$.

criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).

[equivalent of $American(x) \wedge Weapon(y) \wedge Sells(x,y,z) \wedge Hostile(z) \Rightarrow Criminal(x)$

uppercase for vars, lowercase for constants!

read :- as 'if',

',' as AND,

and note period at the end]

Prolog Structure

Prolog programs specify **facts** and **rules**

- properties of objects and relationships between them
- rules, which helps us infer new information

A program

- Consists of one or more predicates
  - Which have one or more clauses
    - Base clauses (facts)
    - Other (Rules)

Logic programming: Prolog ..3

- Depth-first, left-to-right backward chaining;
  clause-order is important
- Built-in predicates for arithmetic etc.,
  e.g., X is Y*Z+3
- Built-in predicates that have side effects
  (e.g., input and output predicates, assert/retract predicates)
- **Closed-world assumption** ("negation as failure")
  - e.g., given
    alive(X) :- not dead(X).
    alive(joe) succeeds if dead(joe) fails

Prolog's "Database semantics"

- Unique names assumption: every constant and ground term refers to a distinct object
- Closed world assumption: only sentences that are true are those that are entailed by the KB
- No way to assert something is false in Prolog
- No occur check
- No check for recursion
- Less expressive, but more efficient

Prolog examples

- Base clauses, e.g.: loves(john, mary). [Predicate: loves/2 = loves with arity 2]
- Non-base clauses, e.g.: loves(A,B) :- loves(A, C), loves(C, B).
  Read as "loves(A,B) if loves(A,C) AND loves(C,B)".

Kind of flipped if-then. Uses commas for AND, and period as delimiter.

- Note:
  - 'atoms': alphanumeric, start with lower-case letters, or string in single quotes,
  - Variables: alphanumeric, starts with a capital letter;
    '_' treated as capital letter
    - _ is an anonymous variable; each instance a *different* anonymous variable
    - scope of variable is the clause in which it appears
    - no global variables
  - Comments: % single line,
    block comments /* … */

Executing a Prolog program

- Prolog clauses have declarative and procedural readings

```
father(A,B) :- parent(A,B),
male(A).
```

  - A is the father of B if A is a parent of B, and A is male
  - To prove A is the father of B, show A is B's parent, and then that A is male.

- Suppose we have that and

```
parent(quincy, rashida).
male(quincy).
parent(alice, ravi).
parent(john, ravi).
female(alice).
male(john).
```

- We can inquire:

```
?- father(P, ravi).   Or
?- father(P, Q).
```

- Clause of father/2 is located, unification is attempted, and we see:

```
?- father(M,ravi).
T Call: (8) father(_4388, ravi)
* Call: (8) father(_4388, ravi) ? creep
Call: (9) parent(_4388, ravi) ? creep
Exit: (9) parent(alice, ravi) ? creep
Call: (9) male(alice) ? creep
Fail: (9) male(alice) ? creep
Redo: (9) parent(_4388, ravi) ? creep
Exit: (9) parent(john, ravi) ? creep
Call: (9) male(john) ? creep
Exit: (9) male(john) ? creep
T Exit: (8) father(john, ravi)
* Exit: (8) father(john, ravi) ? creep
M = john.
```

Useful ‘commands’

reconsult(File) : read in a Prolog file

reconsult(user): type code in directly, till ^D
Type in a query to run it.

; to look for more results, return for ‘done’

`:- listing(predicate).` To list all clauses

`:- debug.` To start debugging

`:- trace.` To switch on tracing.
Shows Call/Exit, Redo/Fail paths [more later]

`:-save(F)` and `:-restore(F)` : save/restore program state

Halt or ^D to exit

Data structures: Numbers

```
abs(X, Y) :- X < 0, Y is -X.
abs(X, X) :- X >= 0.
```

[infix operators for convenience]

Data structures: Lists ..1

Appending two lists to produce a third. Recursion rocks!

```
append([],Y,Y).
append([X|L],Y,[X|Z]) :- append(L,Y,Z).
```

query: `append(A,B,[1,2]) ?`

answers: `A=[]  B=[1,2]`

`A=[1]  B=[2]`

`A=[1,2]  B=[]`

What happens if we have

```
append([X|L],Y,[X|Z]) :- append(L,Y,Z).
append([],Y,Y).
?- append(A,B,C).
```

Base clauses come first!

Data structures: Lists ..2

Checking membership in a list

```
member(A, [A | _]).
member(A, [_ | B]) :- member(A, B).
```

Can have lists of lists etc.

Writing Prolog programs

1. Identify the task
2. Collect all information relevant to the problem
3. Identify constants, functions and predicates
4. Encode the general knowledge about domain
5. Encode the knowledge/constraints specific to the problem; think of all cases
6. Try out queries, get answers
7. Debug the program

Criminal Scenario in Prolog

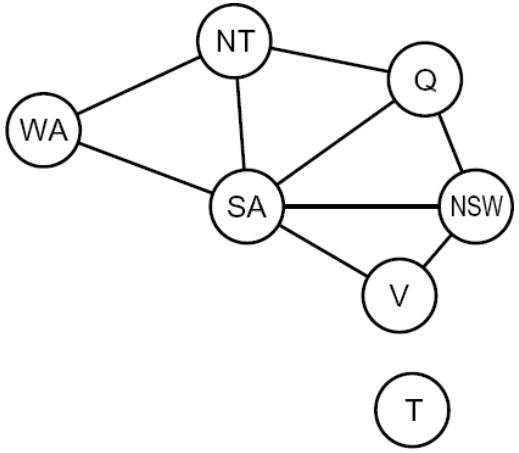
```
% it is a crime to sell weapons
%   to hostile nations
criminal(X):-
american(X),weapon(Y),sells(X,Y,Z),
   hostile(Z).

% Nono ... has some missiles,
owns(nono,m1).

missile(m1).

% all of its missiles were sold to it
%   by Colonel West
sells(west,X,nono) :- missile(X),
    owns(nono,X).
```

```
% Missiles are weapons
weapon(X) :- missile(X).


% An enemy of America counts as
``hostile''
hostile(X) :- enemy(X,america).

% The country Nono, an enemy of America
...
enemy(nono,america).

% West, who is American ...
american(west).
```

Sample Program
Coloring Australia

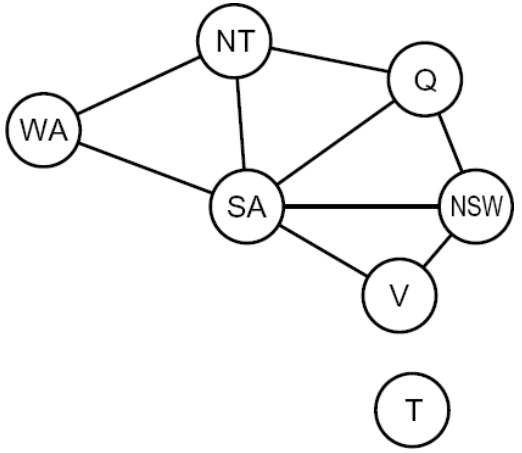
ColorAU.pl part 1

```
%% Map coloring Australia,
%%  with 3 colors
diff(red, green).
diff(red, blue).
diff(green, red).
diff(blue, red).
diff(blue, green).
diff(green, blue).

% A helper function to report results
report(State, Color) :-
   write('State '), write(State),
   write(' can be colored '),
   write(Color), nl.
```



ColorAU.pl part 2

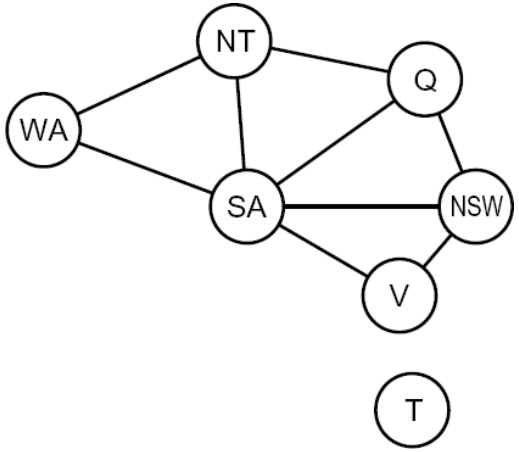
```
% Here comes the main predicate
colorsolution() :-
diff(WAcolor, NTcolor),diff(WAcolor, SAcolor),
diff(NTcolor, Qcolor),diff(NTcolor,SAcolor),
diff(Qcolor,NSWcolor),diff(Qcolor,SAcolor),
diff(NSWcolor,Vcolor),diff(NSWcolor,SAcolor),
diff(Vcolor,SAcolor),diff(Tcolor,_),

% When you have the solutions, print 'em out
report(wa, WAcolor), report(nt, NTcolor),
  report(sa, SAcolor),  report(q, Qcolor),
  report(nsw, NSWcolor), report(v, Vcolor),
  report(t, Tcolor).
```



ColorAU Solution(s)

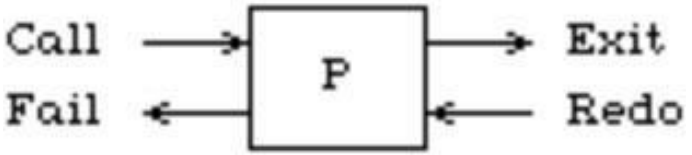
```
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

.swi-prolog.org
s(Word).
```

```
?- reconsult('~/pl/colorAU.pl').
true.

?- colorsolution().
State wa can be colored red
State nt can be colored green
State sa can be colored blue
State q can be colored red
State nsw can be colored green
State v can be colored red
State t can be colored red
true
```



Trace

- CALL: initial; entry to predicate
- EXIT: successful return
- REDO: when it backtracks for another possible answer
- FAIL: no more solutions



ColorAU.pl trace

```
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www
For built-in help, use ?- help(Topic). or ?- apropo

?- reconsult('~/pl/colorAU.pl').
true.

?- trace.
true.

[trace]  ?- colorsolution().
   Call: (8) colorsolution ? creep
   Call: (9) diff(_1786, _1788) ? creep
   Exit: (9) diff(red, green) ? creep
   Call: (9) diff(red, _1788) ? creep
   Exit: (9) diff(red, green) ? creep
   Call: (9) diff(green, _1788) ? creep
   Exit: (9) diff(green, red) ? creep
   Call: (9) diff(green, green) ? creep
   Fail: (9) diff(green, green) ? creep
   Redo: (9) diff(green, _1788) ? creep
   Exit: (9) diff(green, blue) ? creep
   Call: (9) diff(green, green) ? creep
   Fail: (9) diff(green, green) ? creep
   Redo: (9) diff(red, _1788) ? creep
   Exit: (9) diff(red, blue) ? creep
   Call: (9) diff(green, _1788) ? creep
   Exit: (9) diff(green, red) ? creep
   Call: (9) diff(green, blue) ? creep
   Exit: (9) diff(green, blue) ? creep
   Call: (9) diff(red, _1788) ? creep
   Exit: (9) diff(red, green) ? creep
   Call: (9) diff(red, blue) ? creep
   Exit: (9) diff(red, blue) ? creep
   Call: (9) diff(green, _1788) ? creep
   Exit: (9) diff(green, red) ? creep
   Call: (9) diff(green, blue) ? creep
   Exit: (9) diff(green, blue) ? creep
   Call: (9) diff(red, blue) ? creep
   Exit: (9) diff(red, blue) ? creep
   Call: (9) diff(_1786, _1788) ? creep
   Exit: (9) diff(red, green) ? creep
   Call: (9) report(wa, red) ? creep
   Call: (10) write('State ') ? creep
State
   Exit: (10) write('State ') ? creep
   Call: (10) write(wa) ? creep
wa
   Exit: (10) write(wa) ?
```

Useful built-in predicates

- Input: see(File), seen
- Output: write(X), tab(N), nl, tell(File), told

- Control: not X (also: \+ X), fail, ! (‘cut’)

- Arithmetic: X is E, X +Y, … X/Y, X mod Y,
  X =:= Y, X =\= Y , … (>,<, >=, =<)
  Compare with X == Y (identical?) and X \== Y

- Listing/debugging: listing(P), trace/notrace,
  debug/nodebug, spy(P)/nospy(P)

Hints for better Prolog'ing

- Use an editor to type in your program;
  reconsult from Prolog any time you change the program.
- up-arrow in SWI-Prolog is your friend!
- Put clauses of the same predicate together
- Order of clauses is IMPORTANT (why?)
- Watch out for misspellings
  - Singleton error messages are useful
- Take care with **is** and **=**
- Try out trace

alldiff

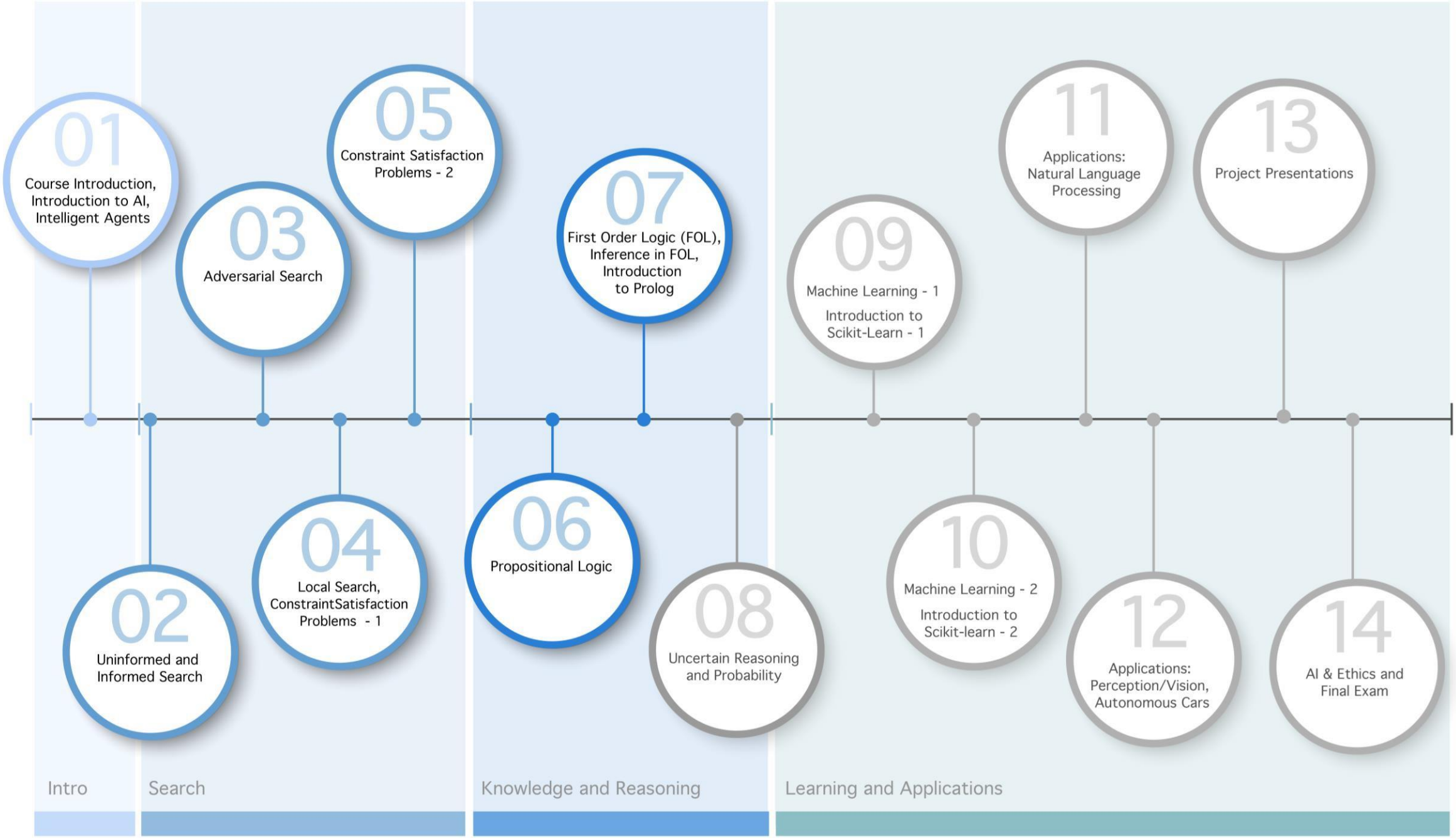
```
% Succeeds if all elements in the argument list are bound and all different.
% Fails if any of the elements are unbound, or equal to some other element.
alldiff([H | T]) :- member(H, T), !, fail.
alldiff([_ | T]) :- alldiff(T).
alldiff([_]).
```

Negation

Negation used to represent constraints.

They block unification

- Cannot use them to identify answers
- Instead, use them to block certain answers


01
Course Introduction, Introduction to AI, Intelligent Agents
02
Uninformed and Informed Search
03
Adversarial Search
04
Local Search, ConstraintSatisfaction Problems - 1
05
Constraint Satisfaction Problems - 2
06
Propositional Logic
07
First Order Logic (FOL), Inference in FOL, Introduction to Prolog
08
Uncertain Reasoning and Probability
09
Machine Learning - 1
Introduction to Scikit-Learn - 1
10
Machine Learning - 2
Introduction to Scikit-learn - 2
11
Applications: Natural Language Processing
12
Applications: Perception/Vision, Autonomous Cars
13
Project Presentations
14
AI & Ethics and Final Exam
Intro
Search
Knowledge and Reasoning
Learning and Applications