Lecture 1

What is AI?

AI is a field of study

"... to investigate the fundamental laws of those operations of the mind by which reasoning is performed; to give expression to them in the symbolical language of a Calculus, and upon this foundation to establish the science of Logic and construct its method; to make that method itself the basis of a general method for the application of the mathematical doctrine of Probabilities; and, finally, to collect from the various elements of truth brought to view in the course of these inquiries some probable intimations concerning the nature and constitution of the human mind.

George Boole (1854)
An Investigation of the Laws of Thought

George Boole defines AI here. We want to develop technology that is smarter than us, but in this process also understand how intelligence comes together. AI is not limited by computers, there is a 2400-year history behind it. AI goes back to discovering how humans think all-together and rational decision making.

- Logic (Aristotle c. 350BC, Boole 1848, Frege 1879, Tarski 1935)
- Formal algorithms (Euclid c. 300BC)
- Probability theory (Pascal 17th C, Bayes 18th C)
- Utility theory (Mill 1863)
- Dynamical systems (Poincare 1892)
- Structural linguistics (Saussure 1916, Bloomfield 1933)
- Formal systems (Gödel 1929, Turing 1936)
- Neural networks (McCullogh & Pitts 1943)
- Cybernetics/Control theory (Wiener 1948)
- Game theory (von Neumann & Morgernstern 1947)
- Decision theory (Bellman 1957)
- Formal linguistics (Chomsky 1957)

Foundations of AI

Philosophy (428 B.C. – present)

AI is not just about coding it is also trying to find logical ways to define intelligence. Your mind is like a machine, and we try to model and represent this machine as best we can. Our mind works with knowledge encoded in our own "internal language" and we use thought and reasoning to determine actions.

Mathematics (800 B.C. – present)

Mathematics provides tools to

- Manipulate logical statements
- Manipulate probability statements
- Allow us to perform algorithms and analyze them
- Study complexity issues
- Allow pattern recognition
- Create models using differential equations/statics

Mathematics also has applications in fields of

- Dynamical systems/RNN's
- Statistical physics/Hopfieled nets

All of this is used within AI

Psychology (1879 – present)

What can psychologists tell us about the way we think? Determine how does learning/memory/problem solving all work? We want to construct models of human intelligence and determine what is intelligence.

Linguistics (1957 – present)

Natural language understanding is a big part of AI (example siri/alexa using our natural language to perform tasks). Language is a major part of the information processing machine and computational linguistics forms a foundation for AI.

Computer Engineering (1940 – present)

We build computers and robots fast enough to make AI applications, and compute thought. We use implementations of algorithms on a computer to make theories operational.

Biocybernetics (1940 – present)

Control theory, feedback mechanisms. We try to create predictive models, for example a robotic dog and we model its behavior as a control system.

Neurology (1950 – present)

Our natural neural networks are the inspiration for AI neural networks. We further understand human neurology by modelling in AI. We try to learn how the brain works and create a model.

Thought as Calculation

The only way to rectify our reasonings is to make them as tangible as those of the Mathematicians, so that we can find our error at a glance, and when there are disputes among persons, we can simply say: “Let us calculate”, without further ado, to see who is right

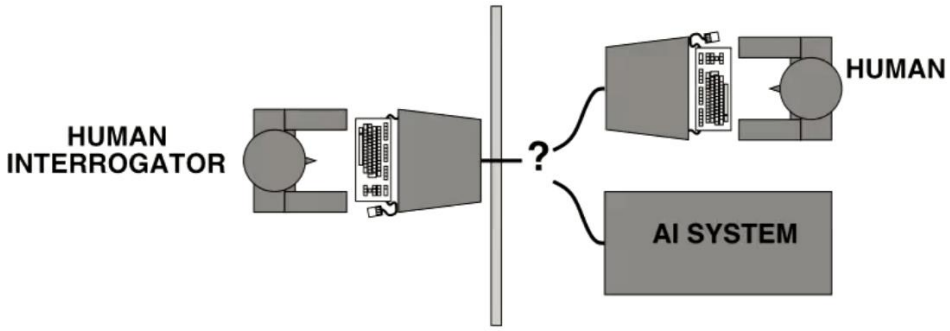
Liebnitz (1685)
The Art of Discovery

Here Leibnitz (co-inventor of calculus) is trying to mechanize reasoning and apply a logic to it, similar to natural deduction.

Alan Turing the beginning of AI

Turing’s arguments to can computers think, “if it walks like a duck, if it quacks like a duck, it is a duck”. Regardless of what happens underneath and how, if something is intelligent that’s that.

Turing’s imitation game was to sit a human interrogator behind a wall, and ask questions, and wouldn’t know who would respond, on the other side of the wall was an AI system and a human both answering at the same time and only one answer would go through. The interrogator would then have to guess who is answering the question from the response.



I believe that in about fifty years time [2000] it will be possible to programme computers with a storage capacity of about 10^9 to make them play the imitation game so well that an average interrogator will not have more than 70 per cent chance of making the right identification after five minutes of questioning.

Samples from Turing imitation game

Extract from Turing paper

Interrogator:	In the first line of your sonnet which reads 'Shall I compare thee to a summer's day', would not 'a spring day' do as well or better?
Witness:	It wouldn't scan.
Interrogator:	How about 'A winter's day' – That would scan all right.
Witness:	Yes, but nobody wants to be compared to a winter's day.
Interrogator:	Would you say Mr. Pickwick reminded you of Christmas?
Witness:	In a way.
Interrogator:	Yet Christmas is a winter's day, and I do not think Mr. Pickwick would mind the comparison.
Witness:	I don't think you're serious. By a winter's day one means a typical winter's day, rather than a special one like Christmas.

Doctor: What brought you here to see me today?
Visitor: Oh, nothing much. I'm feeling a bit tired, that's all.

Doctor: Why do you think you're feeling a bit tired?
Visitor: Well, I've been traveling a lot, and away from home.

Doctor: Tell me more about your family.

Visitor: The trouble is, my mother's ill.

Doctor: How long has she been ill?
...
Visitor: The trouble is, my mother's Irish.

Doctor: How long has she been Irish?

...

Visitor: The trouble is, my mother's poodle.

Doctor: How long has she been poodle?

Turing forgot one thing, it's that most conversations are deep and meaningful, filled with a lot of rubbish. Most of these imitators use pattern-based responses, the turing tests are intended to test reasoning behind the language, we want to fool a person that the machine is real.

The imitation game is testing linguistic capabilities, and only scratches surface level and doesn't understand context, for example people speak differently based on their mood, making the test very limited. The test cannot reason about human environments.

The Child Machine

Turing thought that making a hand-crafted machine would be extremely time consuming and almost unfeasible, he had thought that it would be better to create a machine which was teachable (like a child). Turing had already begun envisioning machine learning. The concept was, instead of making an adult brain from scratch (code it all from scratch), it would be better to make it like a child brain and teach it. Children are extremely teachable

> "Instead of trying to produce a programme to simulate the adult mind, why not rather try to produce one which simulates the child's? If this were then subjected to an appropriate course of education one would obtain the adult brain. Presumably the child-brain is something like a notebook as one buys from the stationers. Rather little mechanism, and lots of blank sheets... Our hope is that there is so little mechanism in the child-brain that something like it can be easily programmed. The amount of work in the education we can assume, as a first approximation, to be much the same as for the human child."
>
> **Alan Turing (1950)**
> **Computing Machinery and Intelligence**

Where has AI gone?

- Focused on the world brain more than the child machine
- Masses of data enable solving problems in way we couldn't predict
    - Crowd sourcing
    - Data centers with enormous computing powers
- We have a lot of more powerful tools to perform much more brute-forced approaches.

Agents, Robots and Autonomous Systems

- Complex behavior required in dynamic environments
- Have to integrate all aspects of AI
- Combines computing with many other disciplines

An autonomous System is any program that accepts input from an environment, then reasons/computes some output, for example

- A harvester can check crop levels and automatically harvest/do nothing based on the levels, not requiring human input
- A scanner at Coles/Woolworths, receives price of item and scans/calculates new total
- Remote cranes that move heavy stuff all by themselves
- Mining trucks, self-driving (no one in them, or if there is has no real use being there)

First AI Machine

Shakey, created by SRI (Stanford research institute) led by Nils nelson. First AI using slow computers, self-driving and can observe its environment.

Freddy, an AI attempting to build a toy car using a robotic machine.

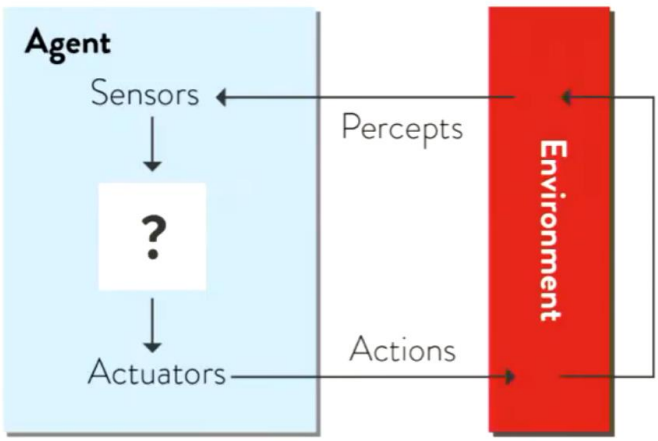
However, the fundamentals were discovered during these 1960s-1970s years. These were AI written to deal with non-predictable, non-deterministic environments, unlike a COMP2521 assignment with well-defined input. Things can happen uncertainly; we want to be able to deal with uncertainty. The techniques used in these machines are still used till this day. One of the first conferences for AI happened in Dartmouth College in 1956, and McCarthy thought of the name AI. (Lighthill tried to kill AI might be a random fact :P).

Agents

Types of Agents

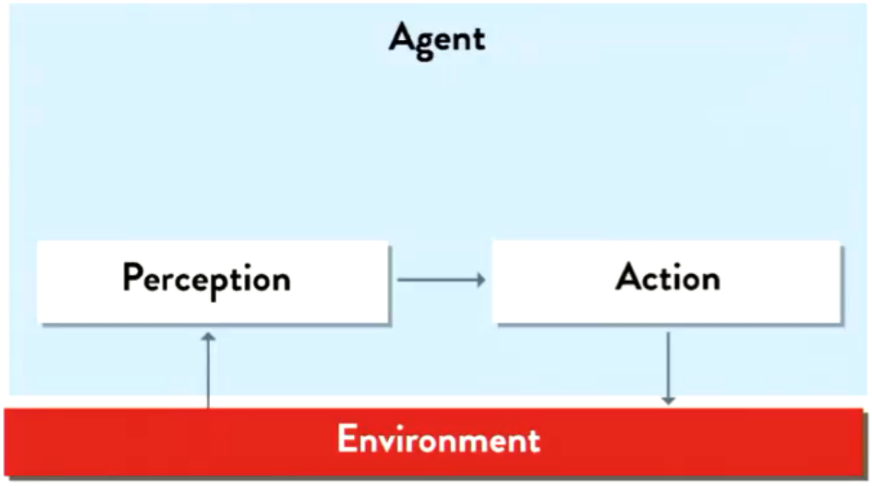
- Reactive Agents
- Model-Based Agents
- Planning Agents
- Utility-Based Agents
- Game Playing Agents
- Learning Agents

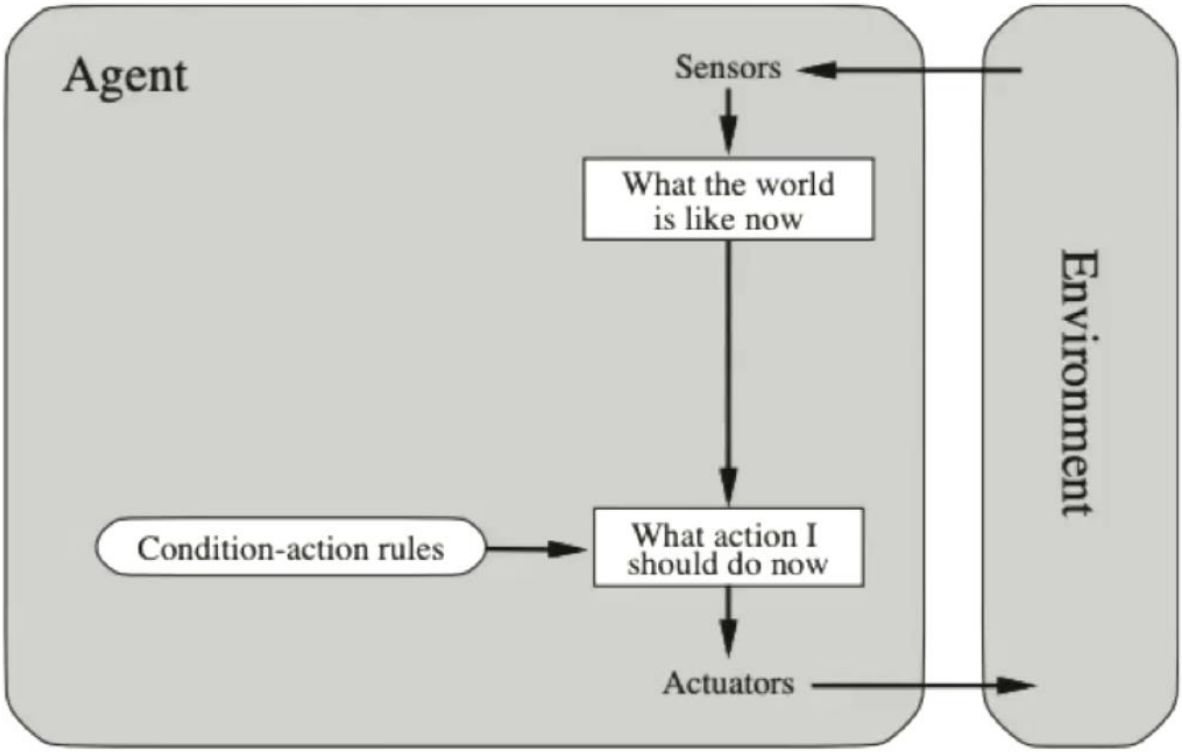
Model of an Agent



Percepts/inputs go into the sensors of our agents, logic is applied, and the actuators perform the output/actions into our environment.

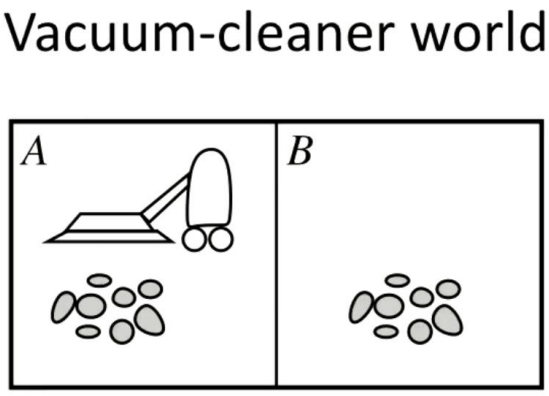
Reactive Agents





No thinking, reflex actions, input → output. The next action is chosen based only on the input, and the output is determined by a set of rules/policies on those inputs. These seem simple but can-do complicated tasks, such as a car hitting obstacles and determining how to correct itself.

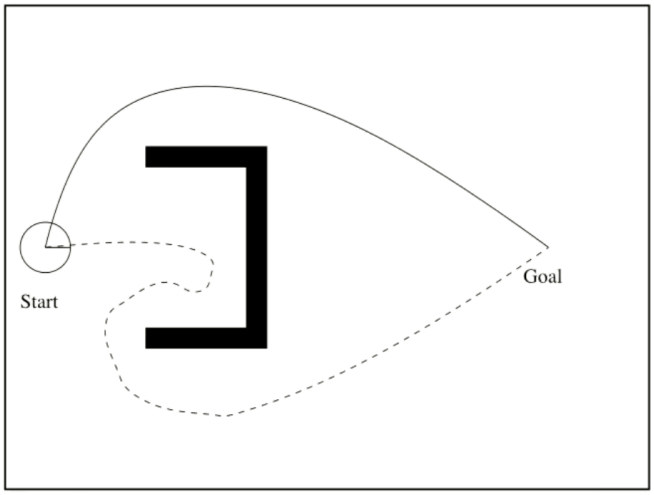
Example program




```
function REFLEX-VACUUM-AGENT([location,status]) returns an action
    if status = Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

Limitations of Reactive Agents

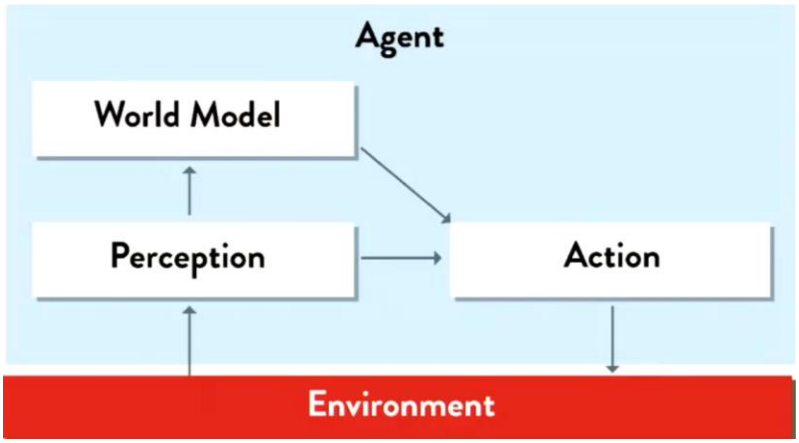
No Memory



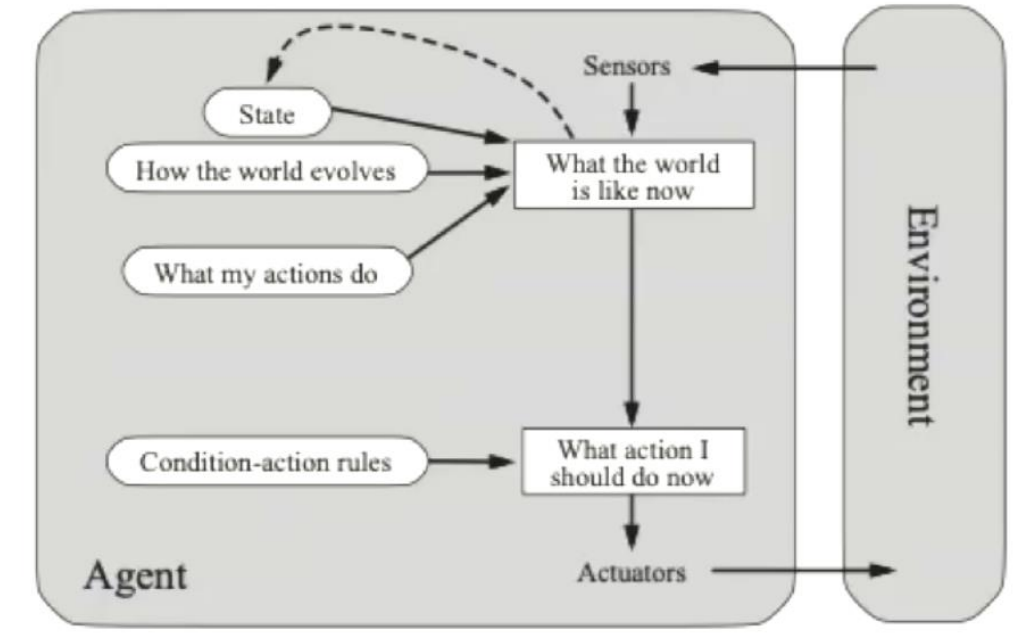
The dotted path shows the first approach when discovering the obstacle, going around it to find the new goal, however a reactive agent will never take the solid line the second time because it doesn't remember where the goal is and where the obstacle ends. Reactive agents have no sense of state and are unable to base decision on previous information gathered. It can repeat the same sequence of actions over and over and the only way it can escape from these infinite loops is if it can randomize its actions when stuck.

Model Based Reflex Agents

We can Augment our reactive agents by giving it a model of the environment. We now pass our inputs through our model and it becomes a part of the model. For example if we had a robot which can only see 180 degree FOV, if it looks in front it cannot see behind it, however when it looks what it sees in front of it is added to the model, so when it turns around it will remember what it saw before turning since it is a part of the model. It's a memory of all inputs. However, we cannot assume the world model is static, the entire room might disappear and if we assume, they are still there then this could cause problems.



Model based agents handle partial observability by keeping track of the part of the world it cannot see now. It maintains state that depends on the history of the inputs and remembers some of the unobserved aspects of the current state (it keeps updating its state based on inputs). Knowledge about how the world works is called the model, and agents that use models in their decisions are called model-based agents.

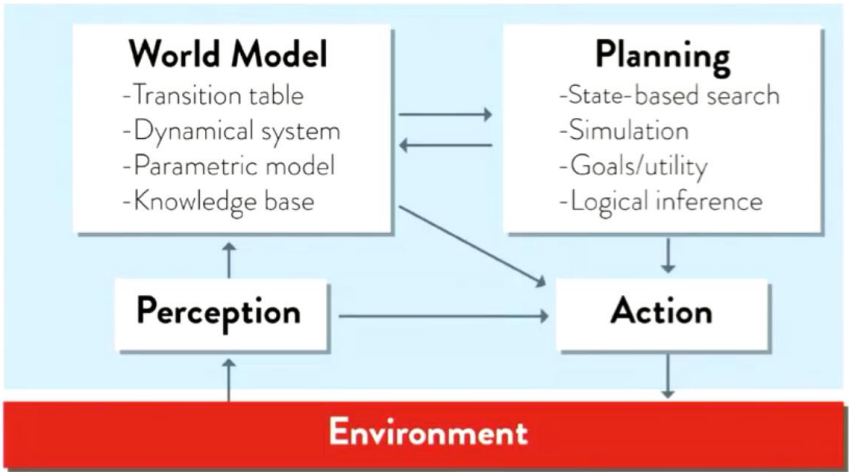


Limitations of Model-Based Reactive Agents

- An agent with a world model but no planning can look into the past but cannot predict the future
- It will perform poorly when the task requires any of the following
    - Searching Several moves ahead
        - Chess
        - Checkers
        - Rubik's cubes
        - Tick tack toe
    - Complex tasks requiring many individual steps
        - Cooking a meal
        - Assembling a watch
    - Logical reasoning to achieve goals
        - Travel to a country

Planning Agent

- Extension on Model-Based Reactive Agents
- We add another block that takes us from our starting state to goal state creating a plan and looks into the future (predicts)



Decision making is different from the reflex rules, it requires considerations into the future

- What happens if
- Will that help achieve my goal?

In the reflex agent designs, this information is not represented because the rules map directly the state to actions. In these planning agents rather than reacting to inputs, we take our inputs and try to create a plan to reach the goal. What sequence of steps takes you from your start state → goal state, it uses building blocks, example I want to sit my exam, a planning based agent would say step 1 how to get to exam, step 2 where to go at university, step 3 sit at exam, it creates a sequence of steps to reach the goal.

Reasoning About Future states

- What is the best action in this situation?
- Faking it
    - Sometimes agents appear to be planning ahead but it's just applying reactive rules
        - Soccer Robo cup examples
    - These rules can be hand coded or can be learned from experience
    - Agents may not be flexible enough to adapt to new situations.

Planning agents are more flexible because the knowledge that supports its decisions is represented explicitly and can be modified, the behavior can be easily changed and doesn't require a set-in stone rule book. However, it is slower since it has to think about what it's doing now and what it needs to do in the future. Planning agents are almost like A* search algorithms, it uses its current states and available actions to try reach the goal state, rather than reflexive agent which only knows its current rule set and at most a model, every new rule has to be written in and fixed.

If you require speed

- Reflexive agent/reflexive model agent dependent on speed required

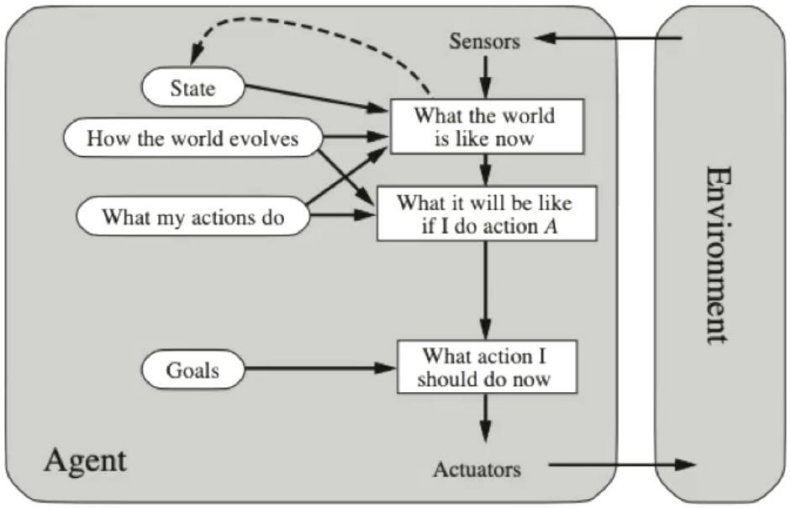
If you require flexibility

- Planning agents.

In some cases, you can also combine agents to perform certain tasks. For example if you were driving, you'd have a rule set, drive forward requires drive gear, reverse requires reverse gear, however things such as parallel parking between 2 cars and not hitting a car require more thoughts and a plan to make it in without hitting any cars.

Ideally, we want the planning to turn into reflexes, for example first time you parallel park your terrible and you might get too close to curb etc. take too long to plan it, but over time with practice it will become almost reflexive.

Goal-based (teleological) agents

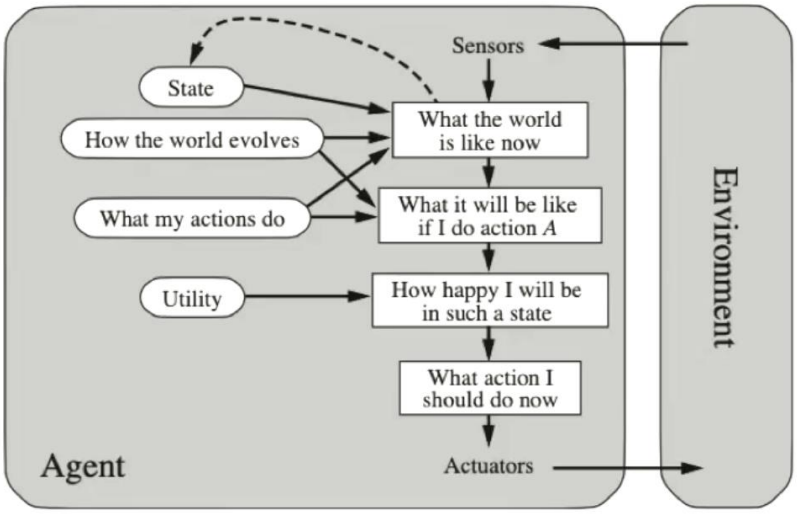


These agents determine the state of the environment after it performs its action and are an extension to planning agents

Utility-based agents

- A rational utility-based agent chooses the action that maximizes the expected utility (performance measure) of the action outcomes. These are heuristic based agents that try to perform actions that receives highest score (similar to A-B pruning)
- The utility-based agent is difficult to implement
    - It has to model and update its environment
    - Tasks involve research on perception/representation/reasoning and learning
    - Implemented as a decision-making agent that must handle the uncertainty inherent in stochastic or partially observable environments.

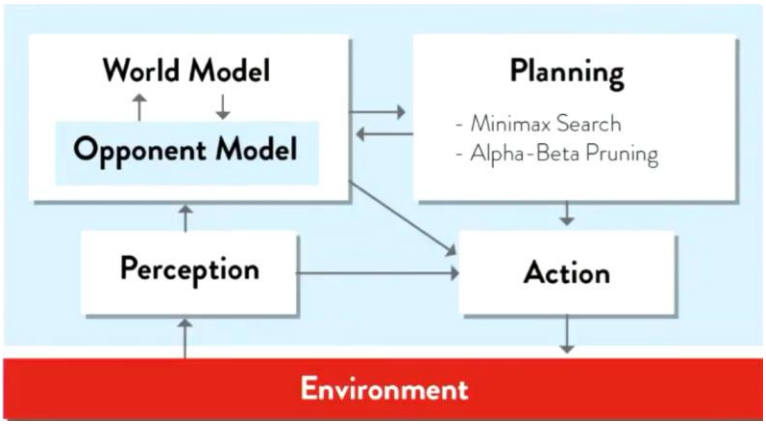
These agents add the idea of scoring certain states.



Game Playing Agents

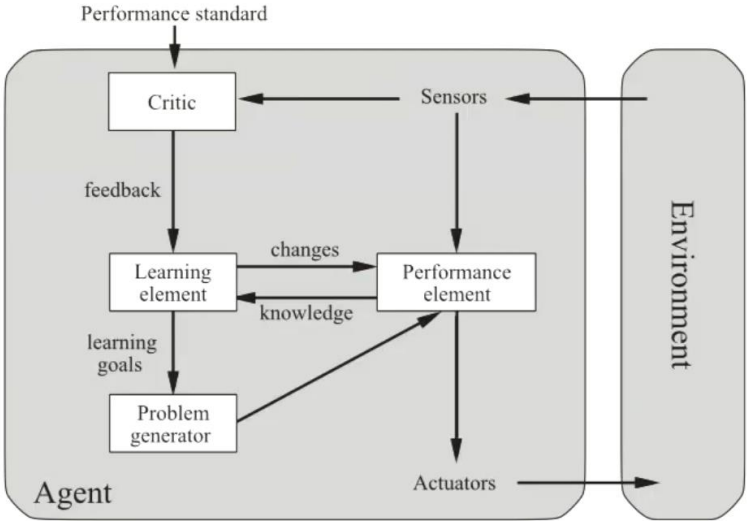
These are similar to Planning agents however they will use search methods to search for the best action to take, these include

- Alpha beta pruning
- Minimax search



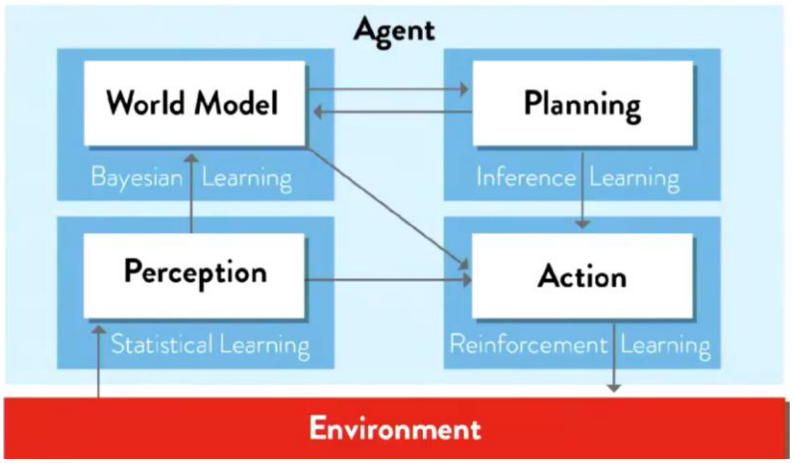
In a more dynamic game such as league of legends/soccer/football these models are difficult as they cannot search too far to determine the best move to do, however in games that are rule and reactive based such as chess, you can plan very far ahead. In these games you can predict the state of the world after every action (0 uncertainty) we know how to react to ANY situation.

Learning Agent



The performance element takes in inputs and decides on actions, the learning element uses feedback from the critic on how the agent is doing and determines how the performance element should be modified to perform better, and the problem generator creates new tasks that provide new and informative experiences. For example an agent playing league of legends, you would check its farm in

the first 10 minutes, and you would then give it a critique saying this was good/bad, and based on that we would give it a new task such as keep perfectly farming for the first 10 minutes of game.



Learning

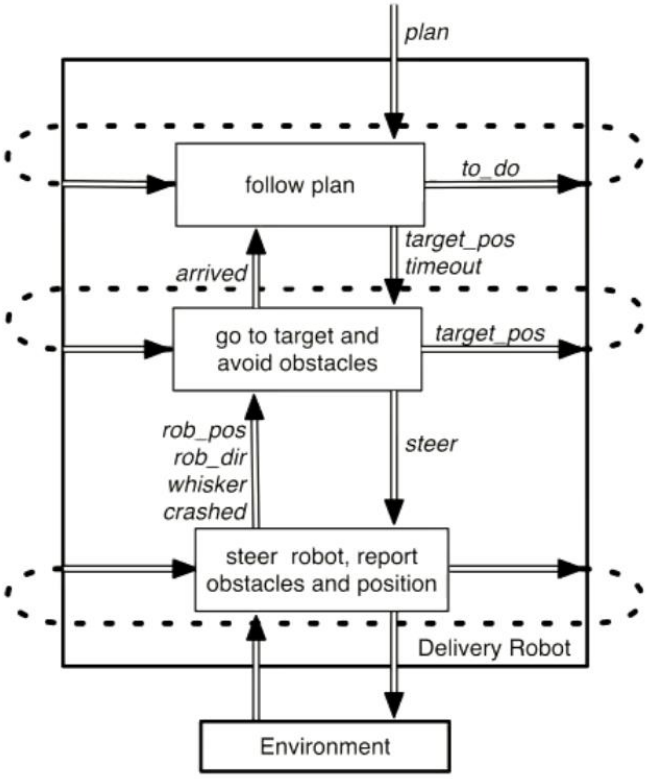
- Learning is not a separate module, but rather a set of techniques for improving the existing modules
- Learning is necessary because:
  - may be difficult or even impossible for a human to design all aspects of the system by hand
  - the agent may need to adapt to new situations without being re-programmed by a human

Representation and Search

- The world model must be represented in a way that makes reasoning easy
- Reasoning (problem solving and planning) in AI always involves some kind of search amongst possible solutions/paths.

An important fact about agents is that the inputs and outputs are strongly coupled, otherwise there would be almost no way of knowing what is going on, for example a cleaning robot would get information regarding position/rubbish/obstacles directly in front of it, and it would have the goal of cleaning, it would clean as per usual, but if someone stands in front we don't want it to just freeze or crash into the person, the robot should continue its task and ADAPT to the new obstacle. Everything feeds and communicates into each other concurrently, the vision effects navigation effecting path etc.

Layered Architecture



Tutorial 1

Can AI solve it?

Choose one of the following tasks (or, a different task of your own choice):

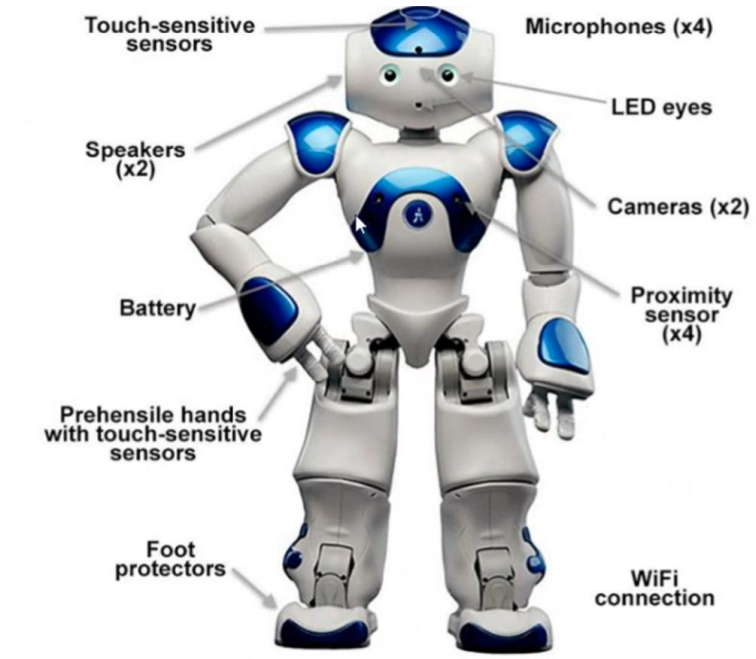
1. Play a decent game of table tennis (ping pong)
2. Drive in the centre of Cairo, Egypt
3. Drive along a winding mountain road
4. Play games such as Chess, Go, Bridge or Poker
5. Discover & prove new mathematical theorems
6. Compose a piece of music, or paint a painting
7. Write an intentionally funny story
8. Give competent legal advice in a specialised area of law
9. Translate spoken English into Chinese (or Swedish) in real time
10. Perform a complex surgical operationC

1. We can make an AI that plays table tennis and win every time very easily, a challenge with AI is that we want to model imperfect gameplay, we want to simulate different difficulties than perfect, as traditionally the AI would try to rip us a new one by playing perfectly to reach victory.
2. Extremely difficult since there is too much uncertainty and chaos in the environment, in the 2nd photo there doesn't seem to be regular rules for traffic its more of a free-for all and it would require dangerous driving to adjust to this environment.
3. Driving at a sensible speed yes, assuming it just needs to stay on the road and obey traffic condition the environment is not changing.
4. Very well done, despite being complex these games all have states and almost perfectly played at top levels by AI. Professionals use AI to train
5. AI has already discovered mathematical theorems, AI has already discovered a new representation of Pi, it can prove/disprove theorems using logic (Z3) however can AI prove a new theorem that has a purpose and is interesting? We can always create new useless theorem, whether or not the rule is meaningful is difficult to decide for AI.
6. Yes, in fact we can teach AI the techniques that musicians/painters use to mimic, however this is an issue on what do we call music/art, anyone can bang 2 pots together and say its music, but can AI compose pieces of certain styles or make a song of certain lengths, or paint pictures a certain way

7. Same issue as above, how do we define comedy, it's all subjective, what counts as comedy? We can teach AI techniques and styles comedians use, but can it fix the problems above. Also, we want to know if AI can do anything intentionally? This is a philosophical question
8. To an extent, AI can easily look up rules extremely quickly and understand how to answer basic questions/queries, it can also narrow down issues however it is limited right now to narrowing issues rather than telling you what to do in your situation like a Lawyer with experience would.
9. Likewise to an extent, sometimes there are context/specifics to a language that are not common in another language, this is a language issue however, since all languages aren't written the same way, however AI can translate to a degree almost perfectly between languages roughly (google translate). Translating in real time requires the entire statement since languages are different, however that isn't real time, since it won't wait for you to finish before translating it translates as it goes.
10. They did surgery on a grape bro. It can perform an operation step by step, however surgery requires extremely quick decision making and reflexes, for example with my corneal transplant midway through surgery the entire surgery had to be changed to a full transplant over partial transplant upon seeing the thinness of corneal layers. It would be extremely difficult to teach an AI mid surgery complex decision making, or what happens if a patient bleed uncontrollably. It is a huge trust issue to let a machine handle your life.

The commonality between all 10 of these issues is that AI struggles with certainty and reliability especially in unexpected situations.

Embodied Agents



How can we make the robot team play and win soccer in terms of code, what do we need to write?

- Vision (see the environment) determine what objects are what
- Position and team position
- Obstacles

- Strategy in soccer
    - Especially with other robots
- Predictions, what the other team will do, and the other robots will do (more complex)
- Recovery
- Movement
    - Can't run and kick at same time
- Kicking the ball
- Collision detection
- spatial awareness
- its own body positioning awareness, moving/human structure
- Rules of soccer (??)
- Defending ball at goal, how to block an incoming object using yourself as the shield
- Communication between other players on team informing
    - Position
    - Whatever the robots need to know
    - Language how they communicate, how to determine state of team
- Translating the information into a communication tool
- Motion/physics

Lecture 2

Prolog

A language designed for AI and the interpreter for prolog is written with AI.

Prolog is a declarative language; you tell it what you want (query) and it figures out how to do it. The interpreter for prolog is an AI program.

In prolog it is similar to a reflexive agent, you give it a set of information (rules) and then you can perform queries.

Relations in prolog

Prolog programs specify relationships among objects, when we say someone owns an object, we are creating a relationship of ownership between 2 objects. And then we can query the relationship asking who owns what

Prolog is similar to querying a database.

We can also add rules to relationship, for example our relationship for two people to be sisters is

- Both are female
- Both have same parents

Note how this isn't just a fact, it is derived from combining rules of other relationships, the female relationship and parent relationship.

In prolog we declare facts describing relationships between objects, we then describe rules by explaining relationship criteria between objects, and finally we ask questions about relationships between objects.

Example

Representing Regulations

The rules for entry into a professional computer science society are set out below:

An applicant to the society is acceptable if he or she has been nominated by two established members of the society and is eligible under the terms below:

- The applicant graduated with a university degree.
- The applicant has two years of professional experience.
- The applicant pays a joining fee of $200.

An established member is one who has been a member for at least two years.

Now we need to define facts, we need to define

- Experience
- Joining fee paid
- Graduated or not
- Which university
- Nominated by who
- Joined when
- Current year

Facts

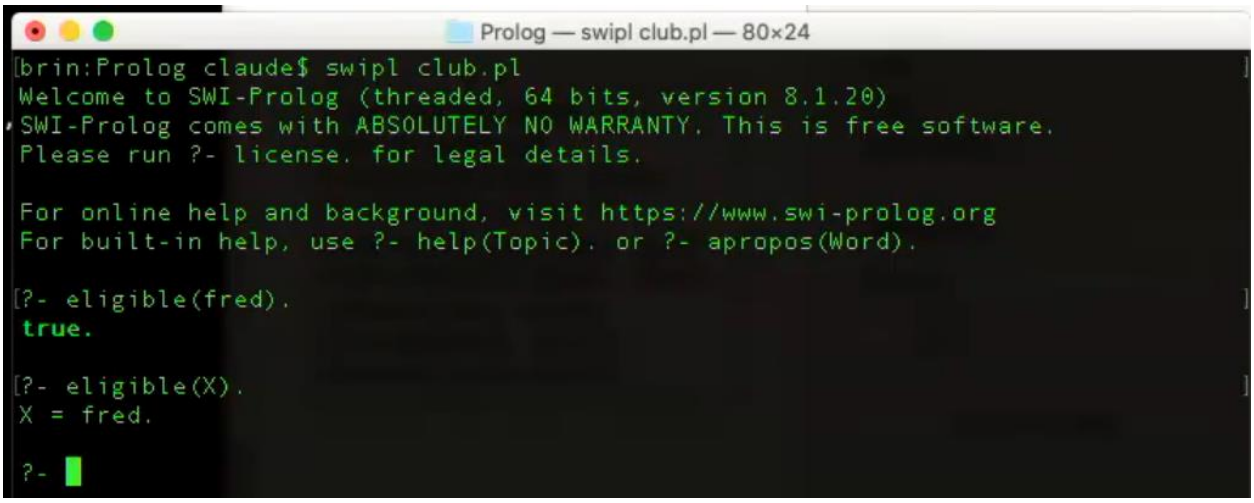
```
experience(fred, 3).
fee_paid(fred).
graduated(fred, unsw).
university(unsw).
nominated_by(fred, jim).
nominated_by(fred, mary).
joined(jim, 2015).
joined(mary, 2017).
current_year(2020).
```

Next, we describe rules if an applicant is

- Acceptable
- Nominated
- Eligible

```
acceptable(Applicant) :-
    nominated(Applicant),
    eligible(Applicant).

nominated(Applicant) :-
    nominated_by(Applicant, Member1),
    nominated_by(Applicant, Member2),
    Member1 \= Member2,
    current_year(ThisYear),
    joined(Member1, Year1), ThisYear >= Year1 + 2,
    joined(Member2, Year2), ThisYear >= Year2 + 2.

eligible(Applicant) :-
    graduated(Applicant, University), university(University),
    experience(Applicant, Experience), Experience >= 2,
    fee_paid(Applicant).
```

Prolog operates different to other languages, it's not clear on inputs and outputs, it will answer the question



In the first one we ask is Fred eligible, it answers yes, in the second one we say eligible(X) where capital X is our variable, here it tells us which X is eligible in the data available. The second one is asking, does there exist an x which satisfies eligible relationship.

Prolog Style

- A fact such as, "Claude lectures in course COMP3411", is written as:

  lectures(claude, 3411).

- The names of relationships are in lower case letters.
- The name of the relationship appears as the first term and the objects appears as arguments to a function.
- A period "." must end a fact.
- *lectures(claude, 3411)* is also called a predicate.

Capital letters are for variables.

First step in prolog, load the database file, then query

- First load database file:

  ?- *[courses.]* *there is a file* courses.pl *in the current directory*

  true. *output from Prolog*

- We can ask:

  ?- *lectures(mike, 9417).*

  true. *output from Prolog*

- To answer this question, Prolog consults its database to see if this is a known fact.
- Suppose we ask:

  ?- *lectures(fred, 9417).*

  false. *output from Prolog*

- Prolog can't find a fact matching the question, so answer "false" is printed
- This query is said to have *failed*.

Prolog similar to Dafny will only use the information it has to answer queries, and if it cannot come to the conclusion using the theorem prover using the information it has, then it will simply return false.

We can also use Prolog to find out certain things, for example which courses does someone lecture?

- A variable must begin with a capital letter or "_".
- To ask Prolog to find the subject that Ashesh teaches, type:

  ?- *lectures(ashesh, Subject).*

  Subject = 2521
- To ask which subjects that Claude teaches, ask:

  ?- *lectures(claude, Subject).*

  Subject = 3411 ;

  Subject = 3431.

  - ***Prolog presents first answer and waits for user input.***
  - ***Type ';' to get next answer.***
  - ***Prolog can find all answers that satisfy a query***

Note we can have multiple values bound to a variable, see in the last example, it first returns 3411 and asks do we want the next answer. Typing in ";" gives us the next answer.

Prolog essentially does a DFS to find the first value that satisfies the predicate, when we type ";" it will essentially repeat this DFS ignoring all previous values we returned to in the end give us all values that satisfy.

Conjunctions

We can combine multiple predicates to make a statement, for example if we want to find out if Fred is lectured by Ashesh, we can combine these two predicates

- Lectures(ashesh, Subject), studies(fred, Subject)

Note the order doesn't matter in conjunction since both have to match, however for efficiency sake always put the predicate with a smaller number of possible variables first. Here what we ask prolog, does ashesh lecture a subject that fred also studies, and using subject as its variable prolog attempts to find a Subject that satisfies the first and second predicate.

Another example to see which students' study which courses taught by claude.

Who does Claude teach:

```
?- lectures(claude, Subject), studies(Student, Subject).
Subject = 3411
Student = jack ;
Subject = 3431
Student = Jill ;
Subject = 3431
Student = henry.
```

Back tracking in Prolog

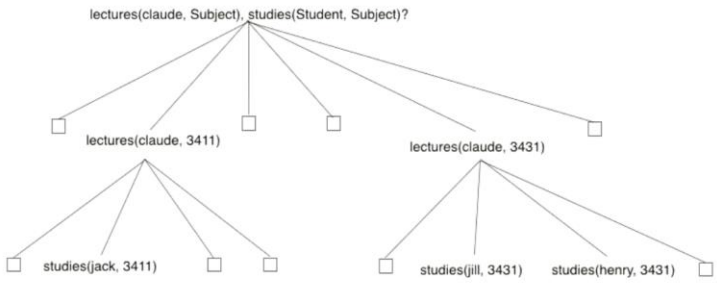
- Prolog solves problem by proceeding left to right and then *backtracking*.
- Given the initial query, Prolog tries to solve

```
lectures(claude, Subject)
```

- There are four lectures clauses, but only two have *claude* as first argument.
- Prolog chooses the first clause containing a reference to *claude*, i.e. *lectures(claude, 3411)*.

Proof Tree

- With *Subject = 3431*, tries to satisfy the next goal, *studies(Student, 3431)*.
- After solution found, Prolog backtracks and looks for alternative solutions.
- May go down branch containing *lectures(claude, 3431)* and then try *studies(Student, 3431)*.



Creating rules

We can create rules in prolog to convert complicated combination of predicates into one predicate, for example we can create a predicate called teaches(Teacher, Student) which tells us if a teacher teaches a student.

```
teaches(Teacher, Student) :-
  lectures(Teacher, Subject),
  studies(Student, Subject).
?- teaches(ashesh, Student).
```

This is a clause

Note the difference between facts and rules are that

1. Facts are unit clauses, meaning they have no ":-" (body) and are just always true
2. Rules contain a body which is the content after the ":-"

The facts are like our axioms in natural deduction, always true.

Structure in Prolog

- Functional terms can be used to construct complex data structures.
- E.g. to say that John owns the book *"I, Robot"*, this may be expressed as:

```
owns(john, "I, Robot ").
```

- Often objects have a number of attributes.
- A book may have a title and an author.

```
owns(john, book("I, Robot ", asimov)).
```

- To be more accurate we should give the author's family ad given names.

```
owns(john, book("I, Robot ", author(asimov, isaac))).
```

Also note to make something not a variable put it in string.

What we can see here is that we can add more and more rules to define what a book really is, because a book is more than just the title, this is done to add more meaning to the program and perform more checks. We compound predicates to create structure for example up there

- A book relation needs
    - Title
    - Author which also needs
        - First name
        - Last name

And this now adds structure to our book relation. Now asking which book john owns with the different structures looks like this.

```
?- owns(john, book(Title, author(asimov, GivenName))).
Title = "I, Robot"
GivenName = isaac

?- owns(john, Book).
Book = book("I, Robot", author(asimov, isaac))

?- owns(john, book(Title, Author)).
Title = "I, Robot"
Author = author(asimov, isaac)
```

In the first one we bind the variables to Title and Given Name part of the book structure, in the second one we bind the variable to the book itself, and in the final one we put the variables to the title and author.

An example library database

- A database of books in a library contains facts of the form:

```
book(CatNo, Title, author(Family, Given)).
member(MemNo, name(Family, Given), Address).
loan(CatNo, MemNo, Borrowed, Due).
```

```
has_borrowed(MemFamily, Title, CatNo) :-
  member(MemNo, name(MemFamily, _), _),
  loan(CatNo, MemNo, _, _),
  book(CatNo, Title, _).
```

Here has borrowed predicate does the following

- Is there a member who matches the family name?
- Is there a loan out for a member with that member number?
- Is there a book with the catalog number out on loan?

This is similar to a database query and checking the keys match since we need information from each predicate linked together to have an answer.

Note _ means we don't care what the value is for that variable, if we don't want to bind a variable just put _.

We can also add complexity to see if a member has an overdue book

1. Write a predicate to check if a date is later than a different date
2. Write a predicate overdue which check
    a. Book out on loan to a member
    b. Check if today is later than the due date for loan
    c. Check the book and member match to the loan

```
later(date(Y, M, D1), date(Y, M, D2)) :- D1 > D2.
later(date(Y, M1, _), date(Y, M2, _)) :- M1 > M2.
later(date(Y1, _, _), date(Y2, _, _)) :- Y1 > Y2.

overdue(Today, Title, CatNo, MemFamily) :-
  loan(CatNo, MemNo, _, DueDate),
  later(Today, DueDate),
  book(CatNo, Title, _),
  member(MemNo, name(MemFamily, _), _).
```

Note we can also do arithmetic in prolog to evaluate certain things

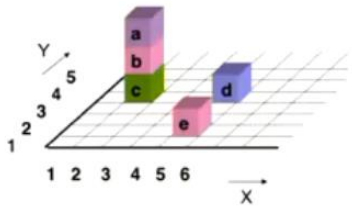
```
due_date(date(Y, M1, D), date(Y, M2, D)) :-
   M1 < 12,
   M2 is M1 + 1.
due_date(date(Y1, 12, D), date(Y2, 1, D)) :-
   Y2 is Y1 + 1.
```

The is statement will check if the left-hand side is arithmetically equivalent to the right-hand side.

Recursion

We can use recursion to define some complex relationship

Take for instance checking if an object a is ABOVE object B




```
Given facts and rules:

on(a, b).
on(b,c).
on(c, table).
…
above(B1, B2) :-
 on(B1, B2).
above(B1, B2) :-
 on(B1, B),
 above(B, B2).
```

```
What can we derive?

on(a, b).
on(b,c).
on(a, table).
…
above(a, b).
above(b, c).
above(a, c).
above(a, table)
…

All this constitutes the
declarative meaning or model
```

In this example first we check if its directly on top, if not the we check if it's on top another block which is above the current block. Similar to Haskell we only have recursion no such thing as looping.

More Recursion in Prolog

A simple Tree in Prolog

- A compound term can contain the same kind of term, i.e. it can be *recursive*.

```
tree(tree(empty, jack, empty ), fred, tree( empty, jill, empty ))
```

Now if we want to traverse the tree and check if something is in the tree, we can add the following predicates

Check if value is in our tree

```
in_tree(X, tree(_, X, _)).
in_tree(X, tree(Left, Y, Right) :-
    X \= Y,
    in_tree(X, Left).
in_tree(X, tree(Left, Y, Right) :-
    X \= Y,
    in_tree(X, Right).
```

Compute size of tree

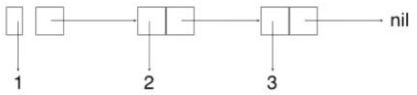
```
tree_size(empty, 0).
tree_size(tree(Left, _, Right), N) :-
    tree_size(Left, LeftSize),
    tree_size(Right, RightSize),
    N is LeftSize + RightSize + 1.
```

Lists in Prolog

A list is either an empty list or a value with another list as its tail, recursive definition of a list

- A list of numbers, [1, 2, 3] can be represented as:

```
list(1, list(2, list(3, nil)))
```



- Since lists are used so often, Prolog has a special notation:

```
[1, 2, 3] = list(1, list(2, list(3, nil)))
```

Some example of list functions

```
?- [X, Y, Z] = [1, 2, 3].
X = 1
Y = 2
Z = 3
```

Unify the two terms on either side of the equals sign.

Variables match terms in corresponding positions.

```
?- [X | Y] = [1, 2, 3].
X = 1
Y = [2, 3]
```

The head and tail of a list are separated by using '|' to indicate that the term following the bar should unify with the tail of the list

```
?- [X | Y] = [1].
X = 1
Y = []
```

The empty list is written as '[]'.

The end of a list is *usually* '[]'.

```
?- [X, Y | Z] = [fred, jim, jill, mary].

X = fred
Y = jim
Z = [jill, mary]
```

There must be at least two elements in the list on the right

```
?- [X | Y] = [[a, f(e)], [n, b, [2]]].

X = [a, f(e)]
Y = [[n, b, [2]]]
```

The right hand list has two elements:
`[a, f(e)]` `[n, b, [2]]`
Y is the tail of the list, [n, b, [2]] is just one element

Now if we want to check is something is a member of a list, first we check if it's the head of the list, otherwise we recall our member relation on the remaining tail of the list

```
member(X, [X | _]).
member(X, [_ | Y]) :-
        member(X, Y).
```

List Concatenation in Prolog

Case 1

Concatenating to an empty list, simply return 2nd list

Start planning by considering simplest case:

```
conc([], [1, 2, 3], [1, 2, 3])
```

Clause for this case:

```
conc([], X, X).
```

Same for the reverse case.

Next, we check for the case of concatenating an actual list

```
conc([A | B], C, [A | D]) :- conc(B, C, D).
```

Our resulting list would be a new list with the head as A, and the new tail being C concatenated to B (see 3rd argument)

We recursively call this till we get to the case where we are concatenating an empty tail!

```
conc([], X, X).
conc([A | B], C, [A | D]) :-
   conc(B, C, D).
```

NOTE THE 3rd ARGUMENT IS THE RESULT!!!! AVOIDS MAJOR CONFUSION

Reversing a list in prolog

```
rev([1, 2, 3], [3, 2, 1])
```

Start planning by considering simplest case:

```
rev([], [])
```

Note:

```
rev([2, 3], [3, 2])
```

and

```
conc([3, 2], [1], [3, 2, 1])
```

```
rev([], []).
rev([A | B], C) :-
  rev(B, D),
  conc(D, [A], C).
```

Summing a list

An Application of Lists

Find the total cost of a list of items:

```
cost(flange, 3).
cost(nut, 1).
cost(widget, 2).
cost(splice, 2).
```

We want to know the total cost of [flange, nut, widget, splice]

```
total_cost([], 0).
total_cost([A | B], C) :-
        total_cost(B, B_cost),
        cost(A, A_cost),
        C is A_cost +  B_cost.
```

Lecture 3

Search is critical to AI in solving all problems and reaching a goal state. It's particularly needed for problems that require

- Motion planning
- Navigation
- Speech and natural language
- Task planning
- Machine learning
- Game playing

There are two types of searches

Uninformed Search

- No problem-specific information is used
- Blind search strategies use only the information available in the problem definition, all it knows is the goal state and whether or not it is in the goal state

Informed Search

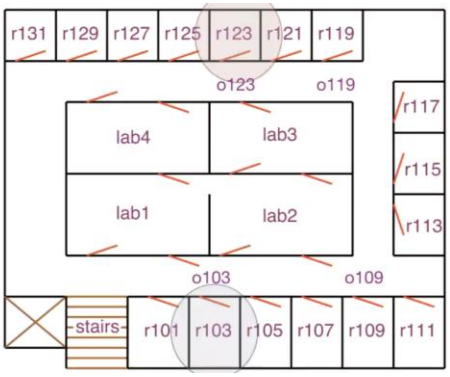
- Uses heuristics to improve efficiency (A*)
- Informed search strategies use task-specific knowledge

State Space Search Problems

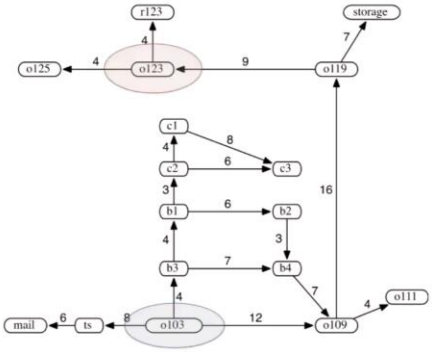
- **State space** — set of all states reachable from initial state(s) by any action sequence
- **Initial state(s)** — element(s) of the state space
- Transitions
  - **Operators** — set of possible actions at agent's disposal; describe state reached after performing action in current state, or
  - **Successor function** — s(x)= set of states reachable from state x by performing a single action
- **Goal state(s)** — element(s) of the state space
- **Path cost** — cost of a sequence of transitions used to evaluate solutions (applies to optimisation problems)

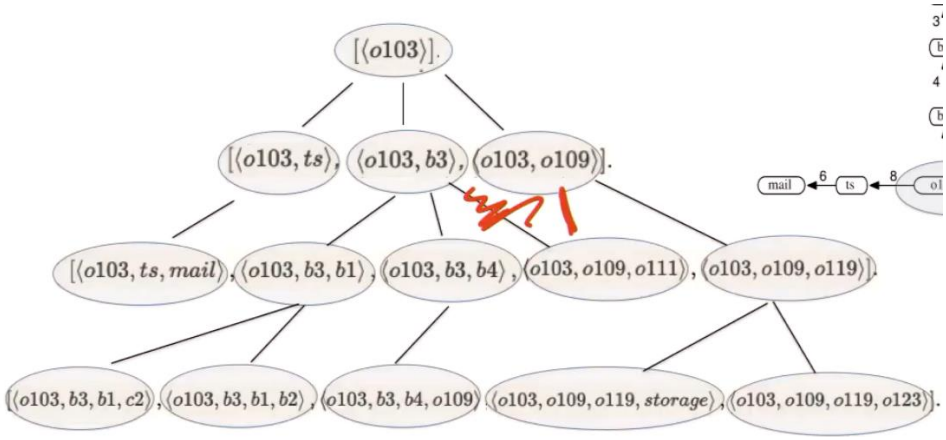
Breadth First Search

- The robot wants to get from outside room 103 to the inside of room 123.
  - The only way a robot can get through a doorway is to push the door open in the direction shown.
- The task is to find a path from o103 to to r123



To do this we need to model our entire search space as a graph, where the nodes are the positions the robot can be in and the cost is how long it takes to get from node to node





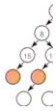
Simple BFS using path tree. What can be seen is the only path to the goal is from the move to 109, all other initial moves lead to a dead end.

Complexity of BFS

- BFS guarantees to find the shortest path to the goal
- To avoid cycles we often use a visited queue to see where we have been and avoid re-visiting old nodes
- The time complexity is good for BFS since it finds the shortest path first
- The space complexity is awful since it keeps EVERY path expanded, the branching factor grows exponentially

Complete? Yes (if b is finite the shallowest goal is at a fixed depth d and will be found before any deeper nodes are generated)

Time? $1 + b^2 + b^3 + \ldots + b^d = \frac{b^{d+1} - 1}{b - 1} = O(b^d)$



Space? $O(b^d)$ (keeps every node in memory; generate all nodes up to level d)

Optimal? Yes, but only if all actions have the same cost

Space is the big problem for BFS. It grows exponentially with depth

Depth First Search

- Expands 1 path at a time
- Good space complexity since it doesn't store all paths
- Isn't complete, doesn't guarantee shortest paths
- Very slow in comparison to BFS since

Use only if memory is an issue (high branching factors and depth)

Complete? No! fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path → complete in finite spaces

Time? $O(b^m)$, m = maximum depth of search tree terrible if m is much larger than d, but if solutions are dense, may be much faster than breadth-first

Space? $O(b^m)$, i.e., linear space

Optimal? No, can find suboptimal solutions first.

DFS in prolog

First, we state our facts to list our graph

```
edge(o103, ts).
edge(o103, b3).
edge(o103, o109).
edge(ts, mail).
edge(b3, b1).
edge(b3, b4).
edge(b1, c2).
edge(b1, b2).
edge(c2, c1).
edge(c2, c3).
```

Next, we state our case that if we are at our goal state

```
% Check if the goal node has beeen reached
depthfirst(N, [N]) :-
    goal(N).
```

Finally, we put in the DFS recursive neighbor checking

```
% Put the current node at the front of the path list,
% then explore each neighbouring node, exploiting Prolog's
% backtracking to maintain a stack implictly
depthfirst(N, [N|Path]) :-
    edge(N, Neighbour),
    depthfirst(Neighbour, Path).
```

Note however this doesn't work with cycles since it has no visited check for its path. A way to fix this is to keep track of a list of visited node. In order to fix this, we can modify the DFS slightly

```
solve(Node, Solution)  :-
    depthfirst([], Node, Solution).

% depthfirst(Path, Node, Solution)
% Use depth first search to find a solution recursively.

% If the next node to be expanded is a goal node, add it to
% the current path and return this path.
depthfirst(Path, Node, [Node|Path])  :-
    goal(Node).

% Otherwise, use Prolog backtracking to explore all successors
% of the current node, in the order returned by s.
depthfirst(Path, Node, Solution)  :-
    edge(Node, Node1),
    \+ member(Node1, Path),     % Prevent a cycle
    depthfirst([Node|Path], Node1, Solution).
```

We keep track of our path (also our visited list) using a new argument path, and check that new nodes expanding aren't a member of the path.

Lowest cost first search (Uniform cost Search)

In a majority of cases all transitions have a cost, paths have a length not necessarily of number of transitions, but it can be for example, distance, time, speed etc. An optimal search which is desirable has the minimum cost path to reach the goal state.

The cost of a path is the cost of each transition along the path summed up, for example a→b→ C path cost would be the cost to go from a → b, then from b → c

- **Delivery robot example:**
  - cost of arc may be resources (e.g., time, energy) required to execute action represented by the arc
  - aim is to reach goal using least resources

BFS can be slightly modified to give us the lowest cost search. We can use a priority queue that uses a cost comparator and always places smallest cost transitions at the front, instead of the first node being what it sees first.

- Complete? Yes, if b is finite and if transition $cost \geq \epsilon$ with $\epsilon > 0$
- Time? Worst case, $O(b^{\lceil C^*/\epsilon \rceil})$ where C^* = cost of the optimal solution and assume every transition costs at least ϵ
- Space? $O(b^{\lceil C^*/\epsilon \rceil})$, $b^{\lceil C^*/\epsilon \rceil} = b^d$ if all step costs are equal
- Optimal? Yes – nodes expanded in increasing order of $g(n)$

Note this can have much worse complexity than BFS, however we are finding the lowest cost path not the shortest hop path.

Summary of Search Strategies

Strategy	Frontier Selection	Complete	Halts	Space
Depth-first	Last node added	No	No	Linear
Breadth-first	First node added	Yes	No	Exp
Lowest-cost-first	Minimal $cost(p)$	Yes	No	Exp

Complete: guaranteed to find a solution if there is one (for graphs with finite number of neighbours, even on infinite graphs)
Halts: on finite graph (perhaps with cycles).
Space: as a function of the length of current path

Notice how none of these searches guarantee halting. There is a way to manually guarantee a halt, and that is by bounding the search to a maximum depth.

Bounded DFS

Expands nodes like Depth First Search but imposes a cutoff on the maximum depth of path.

- Complete? Yes (no infinite loops anymore)
- Time? $O(b^k)$ where k is the depth limit
- Space? $O(bk)$, i.e., linear space similar to DFS
- Optimal? No, can find suboptimal solutions first.

Problem: How to pick a good limit ?

(typo on the slide, in fact it is more incomplete, but it guarantees a halt).

A way to combine the best of all worlds is through a search known as

Iterative Deepening

The problem with bounded DFS is we don't know what bound we should set on the path. With iterative deepening we try all depth bounds iteratively, for example, look at all paths depth 1, then all paths depth 2 etc.

This combines the benefits of BFS and DFS, we only look at 1 path at a time, and we look at all neighbors in each depth.

The cost is a trade-off, it is a lot slower than DFS, but the search is complete.

- Complete? Yes.
- Time: nodes at the bottom level are expanded once, nodes at the next level up twice, and so on:
  - depth-bounded: $1 + b^2 + b^3 + \ldots + b^d = \frac{b^{d+1} - 1}{b - 1} = O(b^d)$
  - Iterative deepening:

    $(d+1)b^0 + db^1 + (d-1)b^2 + \cdots + 2 \cdot b^{d-1} + 1 \cdot b^d = O(b^d)$
  - Example b=10, d=5:
    - depth-bounded: 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111
    - iterative-deepening: 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456
    - only about 11% more nodes (for b = 10).

Iterative deepening starts of small and only goes deeper depths when it cannot find the goal state, it is in the middle of DFS where it goes all the way to the end of a path, and BFS where it only expands 1 depth at a time. This search is very practical in real world situations.

Bi-Directional Search

(REALLY FAST USED THIS IN DATABASE ASSIGNMENTS)

We try to perform the search from start → goal, and from goal → start, and see where they meet in the middle. This is extremely quick in comparison to the other searches as we don't expand at larger depths we essentially half the depth we expand at.

COST ANALYSIS OF SEARCHES

Complexity Results for Uninformed Search

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Time	$O(b^d)$	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(b^m)$	$O(b^k)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(bm)$	$O(bk)$	$O(bd)$
Complete?	Yes[1]	Yes[2]	No	No	Yes[1]
Optimal ?	Yes[3]	Yes	No	No	Yes[3]

b = branching factor, d = depth of the shallowest solution,
m = maximum depth of the search tree, k = depth limit.
1 = complete if b is finite.
2 = complete if b is finite and step costs $\geq \varepsilon$ with $\varepsilon > 0$.
3 = optimal if actions all have the same cost.

Bounded search in prolog

```
bounded_search(N, [N], _) :-
    goal(N).
bounded_search(N, [N|Path], D) :-
    D > 0,
    Depth is D - 1,
    edge(N, Neighbour),
    bounded_search(Neighbour, Path, Depth).
```

Note the variable depth just counts how many iterations have been done starts with maximum, decreases to 0

Iterative Deepening in prolog

```
iter(Start, Solution, L, MaxDepth) :-
    L < MaxDepth,
    bounded_search(Start, Solution, L).
iter(Start, Solution, L, MaxDepth) :-
    L < MaxDepth,
    Limit is L + 1,
    write("Increasing limit to "), writeln(Limit),
    iter(Start, Solution, Limit, MaxDepth).
```

All iterative deepening does is use our existing bounded search, and iteratively increases the limit from initially 1 to the maximum depth allowed, each time performing our bounded search on new depth limit.

Returning all Neighbors in prolog (find all)

```
neighbours(Node, Neighbours) :-
    findall(N, edge(Node, N), Neighbours).
```

Find all is built into prolog, what it will do is find all assignments that satisfy the joint statement. This is useful in a BFS when we want to find all neighbors and add them to the priority queue.

Reachability

Similar to DFS however it only checks if the goal is reachable it won't care for the path (uses neighbors above). Note also union is a built in append that will add rest to neighbors into newQ and ignore duplicates. The third clause in reachable is just a measure incase the node has no neighbors, we want to check the rest of the queue.

```
reachable(Goal, [Goal|_]).
reachable(Goal, [First|Rest]) :-
    neighbours(First, Neighbours),
    union(Rest, Neighbours, NewQ),
    writeln(NewQ),
    reachable(Goal, NewQ).
reachable(Goal, [_|Rest]) :-
    reachable(Goal, Rest).
```

Lecture 4

Heuristics

Heuristic searches are informed searches that use problem-specific knowledge to improve the search and find a solution more efficiently. Uninformed search algorithms don't use any problem specific knowledge other than the definition and are not as efficient.

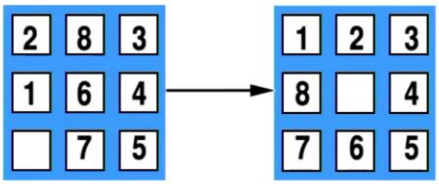
Informed searches can do well when given guidance on where to look for solutions. These are implemented using priority queue to store frontier nodes with the lowest cost being at the head.

Heuristics are a "rule of thumb" for deciding which course of action to take, they offer a cost to reach the end HOWEVER this cost must be an underestimate of the actual cost. This is known as an admissible heuristic. If your heuristic is too weak, then the search is slower, if your heuristic is too strong and over-estimating then it might not give correct solutions.

Our heuristic h(n) will evaluate to 0 when n is the goal node, since cost to reach goal is 0.

Heuristics — Example

8-Puzzle — number of tiles out of place



$h(n) = 5$

A simple heuristic for the example above could just be, how many tiles are out of place, or we can use Manhattan distance to evaluate the heuristic cost, where we calculate total distance needed to fix the state of the board.

For heuristics we have general rules to follow

- Don't ignore the goal when selecting paths (States)
- Extra knowledge can be used to guide the search as long as it is admissible
- H(n) estimates the cost of shortest path from node n to a goal node
- Should be efficient to compute, no point in a heuristic if it slows our search down
- We tend to compute the heuristic cost of the path itself rather than just one node on the path
- The heuristic must be admissible, it must be an non-negative function that is an underestimate of the actual cost of a path to a goal, when we over-estimate we can skip over solutions (BAD)

Example Heuristic Functions

- If the nodes are points on a Euclidean plane and the cost is the distance, $h(n)$ can be the straight-line distance from n to the closest goal.
- If the nodes are locations and cost is time, we can use the distance to a goal divided by the maximum speed.
- If goal is to collect a bunch of coins and not run out of fuel, cost is an estimate of how many steps to collect rest of the coins, refuel when necessary, and return to goal.
- A heuristic function can be found by simplifying calculation of true cost

Search Strategies

For all search algorithms they tend to follow these steps

1. Add initial state to queue
2. Take node from head of queue
3. Test if goal state, if so terminate
4. Expand the node, get all successor paths from the neighboring nodes and add them to the queue
5. Repeat step 2-4 till no more left in queue or we reached goal state

Search strategies only differ by the order in which new nodes are added to the queue of nodes awaiting expansion.

In a BFS we use a queue and add all new nodes to the back of the queue, whereas in DFS we use a queue and add them to the front (more stack-like behavior tbh)

In a Best First search, we use an evaluation function to order the nodes in our priority queue, we use cost of paths to put the lowest cost node at the front

- Similar to uniform cost search

There are two common variations

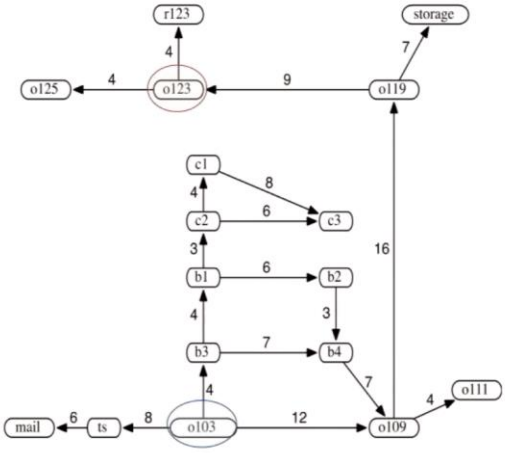
- Greedy search
  - Cost = h(n), only uses the estimate cost from current path to the end, greedy finds the quickest solution to end
- A* Search (THE BOI)
  - Cost = g(n) + h(n), it uses the current cost of the path and the estimate cost to goal, note h(n) must be admissible!!

Delivery Robot Heuristic Function

Use straight-line distance as heuristic, and assume these values:

$$
\begin{array}{lll}
h(mail) = 26 & h(ts) = 23 & h(o103) = 21 \\
h(o109) = 24 & h(o111) = 27 & h(o119) = 11 \\
h(o123) = 4 & h(o125) = 6 & h(r123) = 0 \\
h(b1) = 13 & h(b2) = 15 & h(b3) = 17 \\
h(b4) = 18 & h(c1) = 6 & h(c2) = 10 \\
h(c3) = 12 & h(storage) = 12 &
\end{array}
$$

Heuristic function can be extended to paths by making heuristic value of path equal to heuristic value of node at the end of the path: $h(\langle n_0, \cdots, n_k \rangle) = h(n_k)$.



Greedy Best-First Search

- Select nodes closest to goal according to heuristic function
- Evaluation function is just the heuristic h(n)
- H(n) = 0 if n is goal state
- Greedy search minimizes the estimated cost to the goal, it expands nodes that is estimated to be closest to the goal
- Frontier queue is priority queue ordered by our evaluation function
- Greedy algorithm will take the best node first.

Greedy Best-First searches are as the name implies will always take the lowest cost path to expand till it finds a goal, similar to DFS. A problem with Greedy Best-First search is that it can go down un-optimal paths and will continue down the bad path since it only looks at whats the next best node to expand and doesn't look at its current path.

Properties of Greedy Best-First Search

Complete: No. Can get stuck in loops.
(Complete in finite space with repeated-state checking)

Time: $O(b^m)$, where m is the maximum depth in search space.

Space: $O(bm)$ (retains all nodes in memory)

Optimal: No.

- Greedy Search has the same deficits as Depth-First Search.
- However, a good heuristic can reduce time and memory costs substantially.

This is almost identical to DFS, however the main advantage is a good heuristic can make the DFS a lot faster.

A* Search

The cost of each path in A* is the current cost of the path, added to the estimate to the goal. A* will always find the optimal path AS LONG AS THE HEURISTIC IS ADMISSBLE. A* is a BFS using cost of paths and a heuristic making it much more efficient.

- Lowest-cost path is eventually found.
- Forced to try many different paths, because some temporarily seem to have the lowest cost.
- Still does better than lowest-cost-first search and greedy best-first search.

Optimality of A*

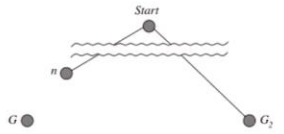
- Heuristic h is said to be admissible if

  $\forall n \; h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost from n to goal
- If h is admissible then $f(n)$ never overestimates the actual cost of the best solution through n.
  - $f(n) = g(n) + h(n)$, where $g(n)$ is the actual cost to n and $h(n)$ is an underestimate
- Example: h = *straight line distance* is admissible because the shortest path between any two points is a line.
- A^* is optimal if h is admissible.
- Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is.

It is impossible with an admissible heuristic to receive a sub-optimal path

Optimality of A* Search

Suppose a sub-optimal goal node G_2 has been generated and is in the queue. Let n be the last unexpanded node on a shortest path to an optimal goal node G.



$$
\begin{aligned}
f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\
&= g(G) && \text{since } G_2 \text{ is sub-optimal} \\
&\geq f(n) && \text{since } h \text{ is admissible}
\end{aligned}
$$

Hence $f(G_2) > f(n)$, and A* will never select G_2 for expansion because queue is always ordered, e.g. $[\cdots, n, \cdots, G_2]$.

In this example, suppose G2 is one path to the goal that is longer than the optimal path G, the way A* works is that since f(n) is a smaller cost than f(g2), f(n) will be expanded before and the path to G is found, it is impossible to take the path G2.

Optimality of A* Search

Complete:	Yes, unless infinitely many nodes with $f \leq$ cost of solution
Time:	Exponential in *relative error in* $h \times$ *length of solution*
Space:	Keeps all nodes is memory
Optimal:	Yes (assuming h is admissible).

Similar to BFS however informed.

Now as can be seen we can take the best of Greedy Search and best of A* search similar to how iterative deepening uses best of both worlds and we get

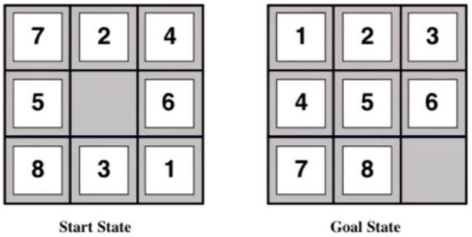
Iterative Deepening A* Search (IDAStar)

Iterative deepening A* is a low memory variant of A* that performs a series of DFS but cuts off at threshold values of our evaluation function, and the threshold is increased with each successive search.

Examples of Admissible Heuristics

$h_1(n)$ = total number of misplaced tiles

$h_2(n)$ = total Manhattan distance = $\sum$ distance from goal position



Start State Goal State

$h_1(Start) = 6$

$h_2(Start) = 4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14$

Both these heuristics are admissible so when choosing the heuristic what one is better?

- The second one is better since it gives us a closer estimate of how to reach our goal
- The first one only tells us tiles out of place which is a worse estimate

The one that gives us the closer estimate will more likely cut off bad branches from the search.

If h2(n) is >= h1(n) for all n and both are admissible, then h2 dominates h1 and is better for the search. The aim is to make our heuristic as close to the estimate as possible whilst being admissible.

- typical search costs:

14-puzzle	IDS	= 3,473,941 nodes
	$A^*(h_1)$	= 539 nodes
	$A^*(h_2)$	= 113 nodes
24-puzzle	IDS	$\approx 54 \times 10^9$ nodes
	$A^*(h_1)$	= 39,135 nodes
	$A^*(h_2)$	= 1,641 nodes

This is a clear example of how much a heuristic can make a difference. The higher the quality of an heuristic the better as long as it's admissible.

Lecture 5

Constraint Satisfaction Problems

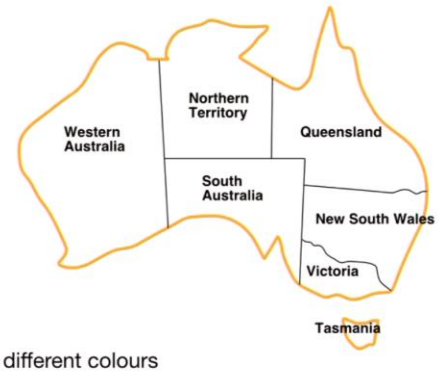
Often resources are scheduled and require optimization for example

- Assignment problems (e.g. who teaches what class)
- Timetabling problems (e.g. which class is offered when and where?)
- Hardware configuration (e.g. minimise space for circuit layout)
- Transport scheduling (e.g. courier delivery, vehicle routing)
- Factory scheduling (optimise assignment of jobs to machines)
- Gate assignment (assign gates to aircraft to minimise transit)

These have real world impact and economic impacts. For example, when we did our Shipment Planner A* search for COMP2511, we had to find the optimal schedule that would hit every shipment using the least fuel.

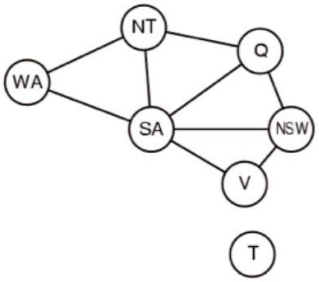
Problems are defined by a set of variables each with a domain of possible values, and a set of constraints that specify what combination of values are allowed. The aim is to find an assignment of the variables from the domains in a way that none of the constraints are violated.

Example: Map-Colouring

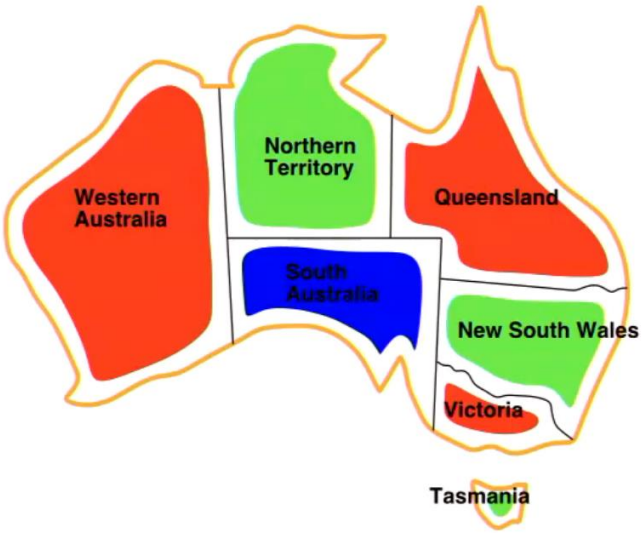


Variables: WA, NT, Q, NSW, V, SA, T
Domains: D_i = {red, green, blue}
Constraints: adjacent regions must have different colours
e.g. WA≠ NT, etc.

This is a small problem because there are only 7 states, however if we used America with its 50 something states, this becomes a more complex problem.



Some variables such as WA or V are very simple since they only have 2 variables to work around, 2 constraints, whereas SA has 5 constraints.



Varieties of CSPs

In this course we will be using discrete variables for our problems

- Finite domains
    - Boolean CSPs, satisfiability
- Infinite domains
    - Strings, integers
        - Job scheduling where variables are time taken for job
        - Constraint language needed
        - Linear constraints are solvable, but nonlinear constraints are undecidable (problems may not hold, requires heuristics).
- Continuous variables
    - Start and end time for observations
    - Linear constraints solvable in polynomial time by linear programming methods.

Types of Constraints

- Unary constraints involve a single variable
  - $M \neq 0$
- Binary constraints involve pairs of variables
  - $SA \neq WA$
- Higher-order constraints involve 3 or more variables
  - $Y = D + E$ or $Y = D + E - 10$
- Inequality constraints on continuous variables
  - $EndJob1 + 5 \leq StartJob3$
- Soft constraints (preferences)
  - 11am lecture is better than 8am lecture!

Path Search vs Constraint Satisfaction

Path search problems – sliding puzzle

- Knowing final state is simple and apart of the problem
- Difficult part is how do we get to the final state

Constraint search problems – n Queens

- Difficult part is knowing the final state
- How to get there is easy

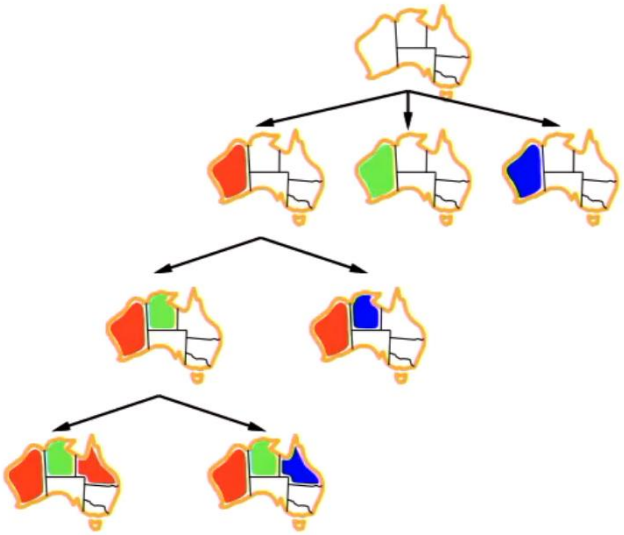
In path searching problems, we know our goal, and we want to know how to get to that goal optimally, or at all. In the case of CSPs, we don't know what our goal state is, all we know is what is not allowed and what is allowed.

Backtracking Search

CSPs can be solved by assigning values to variables one by one in different combinations, and whenever a constraint is violated, go back to most recently assigned variable and assign it a new value.

We can use a DFS where states are the values assigned to the variables so far,

- Initial state no variables assigned
- Successor function, assign a value to an unassigned value that does not conflict with previously assigned values, if no legal values remain, then fail
- Goal test: all variables are assigned a value and no constraints are violated.



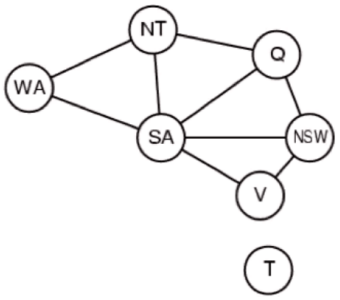
Properties of backtracking search

If there are n variables, each solution occurs at depth n, since that is where all n variables are assigned.

- Variable assignments are commutative
    - E.g. 3 + 4 == 4 + 3, same solution
- Backtracking search can solve n-queens up until around n == 25.

Prolog code for map

```
variables([wa=_, nt=_, q=_, nsw=_, v=_, sa=_, t=_]).

domain(red).
domain(green).
domain(blue).

connected(wa, nt).
connected(wa, sa).
connected(nt, q).
connected(nt, sa).
connected(sa, q).
connected(sa, nsw).
connected(sa, v).
connected(q, nsw).
connected(v, nsw).

adjacent(A, B) :- connected(A, B).
adjacent(A, B) :- connected(B, A).

solve(V) :-
    variables(V),
    assign_all(V).

assign_all([]).
assign_all([State|OtherVariables]):-
    assign_all(OtherVariables),
    assign_variable(State, OtherVariables).

assign_variable(Var = Colour, OtherVariables) :-
    domain(Colour),
    constraint(Var = Colour, OtherVariables).

constraint(S1 = C, OtherVariables) :-
    findall(S, (adjacent(S1, S), member(S = C, OtherVariables)), []).
```



Improving Backtracking search

We can improve the method of searching by asking

- Which variable should we assign next?
- In what order should its values be tried?
- Can we detect failure early?

For example, in the map problem, we would want to start with the variable that has the most constraints (most neighbors) since it will eliminate failure cases early. The order which we choose variables is important. Note that the method chosen is dependent on the domain of the problem, no one problem trumps another, you need to pick based on your problem domain.

Minimum Remaining Values

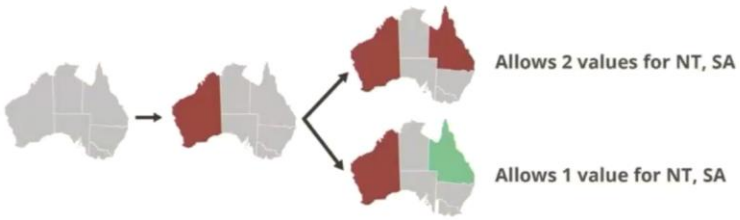
Minimum Remaining Values is where we choose the variable with the fewest legal values, the most constrained variable.

Degree Heuristic

- Tie breaker among MVR variables
- Choose the variable with the most constraints on remaining variables.

Least Constraining Value

- Given a variable choose the least constraining value, the one that rules out the fewest values in the remaining variables.

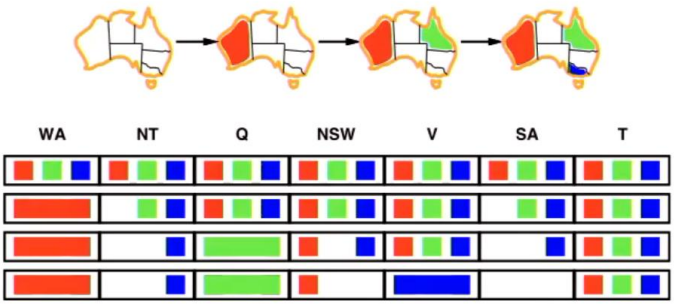


- More generally, 3 allowed values would be better than 2, etc.
  Combining these heuristics makes 1000 queens feasible.

This will take the paths with the most values allowed, the least constraining path.

Forward Checking

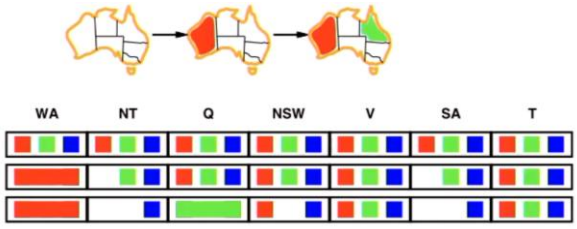
- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values
    - Prune off bad paths and early backtrack



Here instead of continuing down the path we can just terminate as we know it leads to a dead end, pruning off the path.

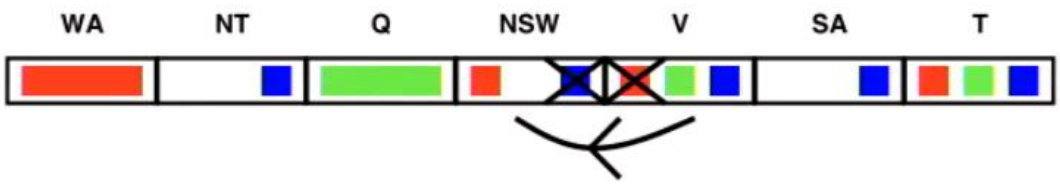
Constraint Propagation

Forward checking propagates information from assigned to unassigned variable but doesn't provide early detection for all failures

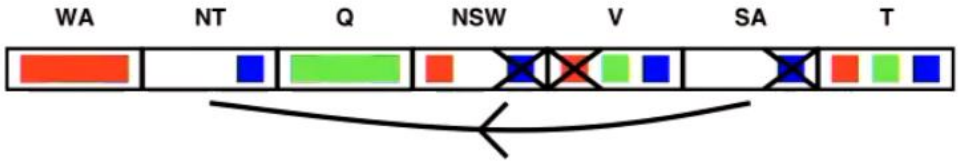


NT and SA cannot both be blue!
Constraint propagation repeatedly enforces constraints locally

We do this by checking Arc Consistency which is the simplest form of propagation. An arc X → Y is consistent if, for every value in X there is some allowed value in Y, if we change a variable it will have impact on another variable.

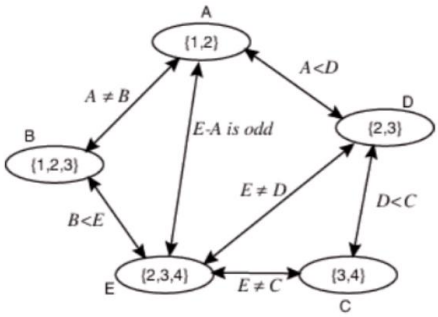


- If X loses a value, neighbours of X need to be rechecked.
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor after each assignment



- Arc consistency detects failure earlier than forward checking.
- For some problems, it can speed up search enormously.
- For others, it may slow the search due to computational overheads

Variable Elimination

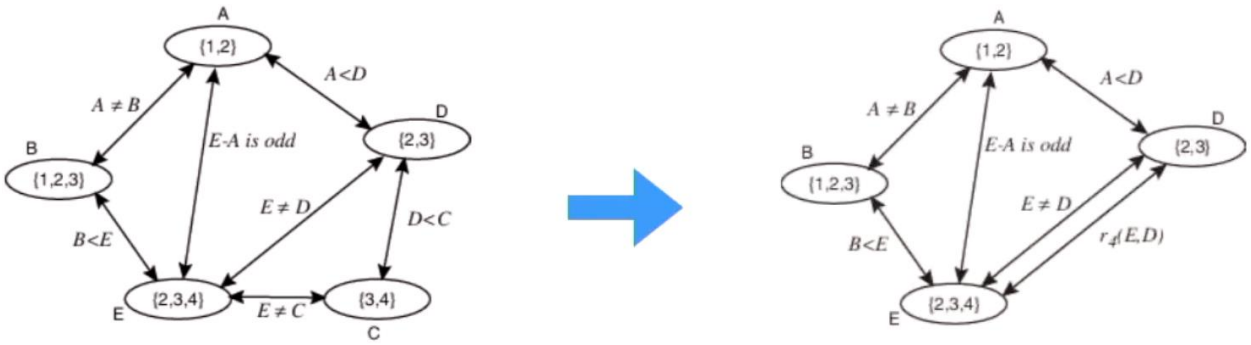


Variables: A, B, C, D, E

Domains: A = {1,2}, B = {1,2,3}, C = {3,4}, D = {2,3}, E = {2,3,4}

Constraints: A ≠ B, E ≠ C, E ≠ D, A < D, B < E, D < C, E-A is odd

We eliminate variables one by one and replace them with constraints on adjacent variables, workout how C effects D and E, then create a new constraint on D and E and remove C



1. Select a variable X
2. Join the constraints in which X appears
3. Project join onto its variables other than X (forming r_4)

$r_1 : C \neq E$	C	E
	3	2
	3	4
	4	2
	4	3

$r_2 : C > D$	C	D
	3	2
	4	2
	4	3

$r_3 : r_1 \bowtie r2$	C	D	E
	3	2	2
	3	2	4
	4	2	2
	4	2	3
	4	3	2
	4	3	3

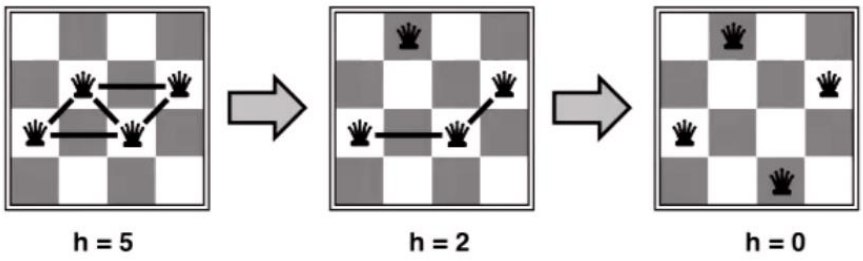
$r_4 : \pi_{\{D,E\}} r_3$	D	E
	2	2
	2	3
	2	4
	3	2
	3	3

↪ new constraint

1. Select a variable X
2. Join the constraints in which X appears, forming constraint R1
3. Project R1 onto its variables other than X, forming R2
4. Replace all of the constraints in which X appears by R2
5. Recursively solve the simplified problem, forming R3
6. Return R1 joined with R3

If there is only one variable left, pick the values which satisfy its unary constraints, then join on the tables going back up the solution.

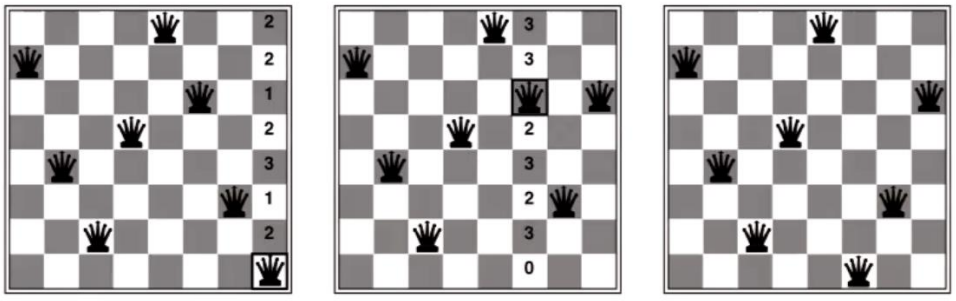
Iterative Improvement (Local Search)



- Iterative Improvement
  - assign all variables randomly in the beginning (thus violating several constraints),
  - change one variable at a time, trying to reduce the number of violations at each step.
  - Greedy Search with h = number of constraints violated

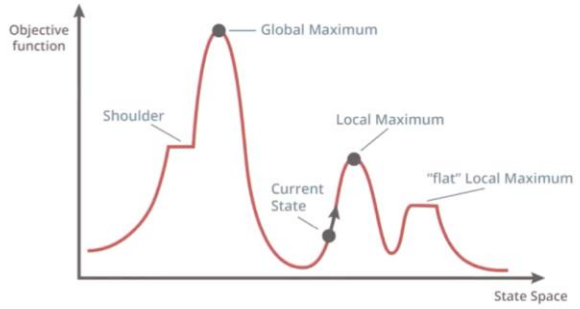
Too chose the new value to change the variable too, we can use hill climbing techniques to pick the position which will result in the least conflicts.

Hill-climbing by min-conflicts



- Variable selection: randomly select any conflicted variable
- Value selection by min-conflicts heuristic
  - choose value that violates the fewest constraints

Flat regions and local optima



Sometimes, have to go sideways or even backwards in order to make progress towards the actual solution.

We don't want to get stuck on local optimum choices if there is a greater choice previously. To solve this issue of flat regions and local maxima/minima we use

Simulated Annealing

- Stochastic hill climbing based on difference between evaluation of the previous state and new state
    - If the new state has a larger value than the old state, make the change
    - Otherwise make the change with probability

$$e^{-(h_1-h_0)/T}$$

where T is a "temperature" parameter.

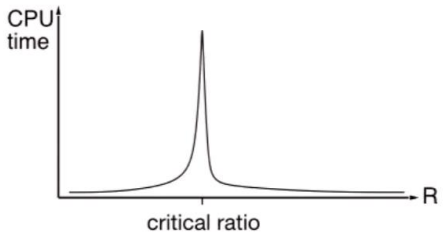
    - As T approaches 0, we get ordinary hill climbing
    - As T gets larger, we get totally random search
    - We start off with a value of T and slowly decrease it during the search

This means that when we do get stuck, and the change is small, we pick a completely different state

Phase Transition in CSP's

- Given random initial state, hill climbing by min-conflicts with random restarts can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000).
- In general, randomly-generated CSP's tend to be easy if there are very few or very many constraints. They become extra hard in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$



What this is saying is that, if we have very few or very high constraints, solving CSP's tend to be easy and quick, however in the middle is where CSP's become very hard to compute.

Summary

- CSPs are a special kind of search problem:
  - states defined by values of a fixed set of variables
  - goal test defined by constraints on variable values
- Backtracking = depth-first search with one variable assigned per node
- Variable ordering and value selection heuristics help significantly
- Forward checking prevents assignments that guarantee later failure
- Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies
- Iterative min-conflicts is usually effective in practice
- Simulated Annealing can help to escape from local optima