COMP250: Artificial Intelligence

4: Planning

FALMOUTH UNIVERSITY

Logic

Logical operations

Python	C family	Mathematics	Behaviour tree
`not a`	`!a`	$\neg A$ or $\overline{A}$	Inverter
`a and b`	`a && b`	$A \wedge B$	Sequence
`a or b`	`a || b`	$A \vee B$	Selector

The laws of thought

The laws of thought

- Let A be a **proposition** (a statement about the world)

The laws of thought

- Let A be a **proposition** (a statement about the world)
- `A` is a **boolean** value, either **true** or **false**

The laws of thought

- Let *A* be a **proposition** (a statement about the world)
- `A` is a **boolean** value, either **true** or **false**
- The law of **identity**: `A == A` is always true

The laws of thought

- Let *A* be a **proposition** (a statement about the world)
- `A` is a **boolean** value, either **true** or **false**
- The law of **identity**: `A == A` is always true
- The law of **non-contradiction**: `A && !A` is always false

The laws of thought

- Let *A* be a **proposition** (a statement about the world)
- `A` is a **boolean** value, either **true** or **false**
- The law of **identity**: `A == A` is always true
- The law of **non-contradiction**: `A && !A` is always false
  - I.e. `A` cannot be both true and false

The laws of thought

- Let *A* be a **proposition** (a statement about the world)
- `A` is a **boolean** value, either **true** or **false**
- The law of **identity**: `A == A` is always true
- The law of **non-contradiction**: `A && !A` is always false
  - I.e. `A` cannot be both true and false
- The law of the **excluded middle**: `A || !A` is always true;

The laws of thought

- Let *A* be a **proposition** (a statement about the world)
- `A` is a **boolean** value, either **true** or **false**
- The law of **identity**: `A == A` is always true
- The law of **non-contradiction**: `A && !A` is always false
  - I.e. `A` cannot be both true and false
- The law of the **excluded middle**: `A || !A` is always true;
  - I.e. `A` must be either true or false

Predicates

Predicates

- **Predicates** are propositions with **parameters**

Predicates

- **Predicates** are propositions with **parameters**
- In programming terms, a predicate is a function that returns a boolean

Predicates

- **Predicates** are propositions with **parameters**
- In programming terms, a predicate is a function that returns a boolean
- E.g. `LivesIn(Bob, Falmouth)` could be a predicate for "Bob lives in Falmouth"

Quantifiers

Quantifiers

- $P(x)$ is a predicate

Quantifiers

- $P(x)$ is a predicate
- $\forall x : P(x)$ means that $P(x)$ is true **for all** values of x

Quantifiers

- $P(x)$ is a predicate
- $\forall x : P(x)$ means that $P(x)$ is true **for all** values of x
- $\exists x : P(x)$ means that **there exists** at least one value of x such that $P(x)$ is true

Implication

Implication

- "A implies B" means "if A is true then B is true"

Implication

- "A implies B" means "if A is true then B is true"
- Written as $A \implies B$

Implication

- "A implies B" means "if A is true then B is true"
- Written as $A \implies B$
- E.g. if someone lives in Falmouth then they live in Cornwall

Implication

- "*A* implies *B*" means "if *A* is true then *B* is true"
- Written as $A \implies B$
- E.g. if someone lives in Falmouth then they live in Cornwall
- $\forall x : \mathrm{LivesIn}(x, \mathrm{Falmouth}) \implies \mathrm{LivesIn}(x, \mathrm{Cornwall})$

Contrapositive

Contrapositive

- $A \implies B$ is equivalent to $\neg B \implies \neg A$

Contrapositive

- $A \implies B$ is equivalent to $\neg B \implies \neg A$
- E.g. if someone does not live in Cornwall then we know they don't live in Falmouth

Contrapositive

- $A \implies B$ is equivalent to $\neg B \implies \neg A$
- E.g. if someone does not live in Cornwall then we know they don't live in Falmouth
- $\forall x : \neg \mathrm{LivesIn}(x, \mathrm{Cornwall}) \implies \neg \mathrm{LivesIn}(x, \mathrm{Falmouth})$

Equivalence

Equivalence

- If $A \implies B$ and $B \implies A$ then A and B are **logically equivalent**

Equivalence

- If $A \implies B$ and $B \implies A$ then A and B are **logically equivalent**
- A is true **if and only if** B is true

Equivalence

- If $A \implies B$ and $B \implies A$ then A and B are **logically equivalent**
- A is true **if and only if** B is true
- Written as $A \iff B$

Equivalence

- If $A \implies B$ and $B \implies A$ then A and B are **logically equivalent**
- A is true **if and only if** B is true
- Written as $A \iff B$
- E.g. "Alice lives in a city in Cornwall" if and only if "Alice lives in Truro"

Equivalence

- If $A \implies B$ and $B \implies A$ then A and B are **logically equivalent**
- A is true **if and only if** B is true
- Written as $A \iff B$
- E.g. "Alice lives in a city in Cornwall" if and only if "Alice lives in Truro"
- This relies on an extra piece of domain knowledge: Truro is the only city in Cornwall

Equivalence

- If $A \implies B$ and $B \implies A$ then A and B are **logically equivalent**
- A is true **if and only if** B is true
- Written as $A \iff B$
- E.g. "Alice lives in a city in Cornwall" if and only if "Alice lives in Truro"
- This relies on an extra piece of domain knowledge: Truro is the only city in Cornwall
  - $\forall x : \mathrm{InCornwall}(x) \wedge \mathrm{IsCity}(x) \implies x = \mathrm{Truro}$

Implication is transitive

Implication is transitive

- If $A \implies B$ and $B \implies C$ then $A \implies C$

Implication is transitive

- If $A \implies B$ and $B \implies C$ then $A \implies C$
- E.g. if someone lives in Falmouth then they live in Cornwall

Implication is transitive

- If $A \implies B$ and $B \implies C$ then $A \implies C$
- E.g. if someone lives in Falmouth then they live in Cornwall
- And if someone lives in Cornwall then they live in England

Implication is transitive

- If $A \implies B$ and $B \implies C$ then $A \implies C$
- E.g. if someone lives in Falmouth then they live in Cornwall
- And if someone lives in Cornwall then they live in England
- Therefore if someone lives in Falmouth then they live in England

Inverting quantifiers

Inverting quantifiers

- “Everyone who lives in Cornwall likes cider”

Inverting quantifiers

- "Everyone who lives in Cornwall likes cider"
- $\forall x : \mathrm{LivesIn}(x, \mathrm{Cornwall}) \implies \mathrm{Likes}(x, \mathrm{Cider})$

Inverting quantifiers

- "Everyone who lives in Cornwall likes cider"
- $\forall x : \text{LivesIn}(x, \text{Cornwall}) \implies \text{Likes}(x, \text{Cider})$
- What is the **opposite** of this statement?

Inverting quantifiers

- "Everyone who lives in Cornwall likes cider"
- $\forall x : \text{LivesIn}(x, \text{Cornwall}) \implies \text{Likes}(x, \text{Cider})$
- What is the **opposite** of this statement?
- $\neg(\forall x : \text{LivesIn}(x, \text{Cornwall}) \implies \text{Likes}(x, \text{Cider}))$

Inverting quantifiers

- "Everyone who lives in Cornwall likes cider"
- $\forall x : \text{LivesIn}(x, \text{Cornwall}) \implies \text{Likes}(x, \text{Cider})$
- What is the **opposite** of this statement?
- $\neg(\forall x : \text{LivesIn}(x, \text{Cornwall}) \implies \text{Likes}(x, \text{Cider}))$
- In logical terms, the opposite is **not** "nobody who lives in Cornwall likes cider"

Inverting quantifiers

- "Everyone who lives in Cornwall likes cider"
- $\forall x : \text{LivesIn}(x, \text{Cornwall}) \implies \text{Likes}(x, \text{Cider})$
- What is the **opposite** of this statement?
- $\neg(\forall x : \text{LivesIn}(x, \text{Cornwall}) \implies \text{Likes}(x, \text{Cider}))$
- In logical terms, the opposite is **not** "nobody who lives in Cornwall likes cider"
- It's "Not everyone who lives in Cornwall likes cider"

Inverting quantifiers

- "Everyone who lives in Cornwall likes cider"
- $\forall x : \text{LivesIn}(x, \text{Cornwall}) \implies \text{Likes}(x, \text{Cider})$
- What is the **opposite** of this statement?
- $\neg(\forall x : \text{LivesIn}(x, \text{Cornwall}) \implies \text{Likes}(x, \text{Cider}))$
- In logical terms, the opposite is **not** "nobody who lives in Cornwall likes cider"
- It's "Not everyone who lives in Cornwall likes cider"
- I.e. "There is at least one person living in Cornwall who does not like cider"

Inverting quantifiers

- "Everyone who lives in Cornwall likes cider"
- $\forall x : \mathrm{LivesIn}(x, \mathrm{Cornwall}) \implies \mathrm{Likes}(x, \mathrm{Cider})$
- What is the **opposite** of this statement?
- $\neg(\forall x : \mathrm{LivesIn}(x, \mathrm{Cornwall}) \implies \mathrm{Likes}(x, \mathrm{Cider}))$
- In logical terms, the opposite is **not** "nobody who lives in Cornwall likes cider"
- It's "Not everyone who lives in Cornwall likes cider"
- I.e. "There is at least one person living in Cornwall who does not like cider"
- $\exists x : \mathrm{LivesIn}(x, \mathrm{Cornwall}) \wedge \neg \mathrm{Likes}(x, \mathrm{Cider})$

Planning

Planning

Planning

- An **agent** in an **environment**

Planning

- An **agent** in an **environment**
- The environment has a **state**

Planning

- An **agent** in an **environment**
- The environment has a **state**
- The agent can perform **actions** to change the state

Planning

- An **agent** in an **environment**
- The environment has a **state**
- The agent can perform **actions** to change the state
- The agent wants to change the state so as to achieve a **goal**

Planning

- An **agent** in an **environment**
- The environment has a **state**
- The agent can perform **actions** to change the state
- The agent wants to change the state so as to achieve a **goal**
- Problem: find a sequence of actions that leads to the goal

STRIPS planning

STRIPS planning

- **St**anford **R**esearch **I**nstitute **P**roblem **S**olver

FALMOUTH UNIVERSITY

STRIPS planning

- **St**anford **R**esearch **I**nstitute **P**roblem **S**olver
- Describes the state of the environment by a set of **predicates** which are true

STRIPS planning

- **St**anford **R**esearch **I**nstitute **P**roblem **S**olver
- Describes the state of the environment by a set of **predicates** which are true
- Models a problem as:

STRIPS planning

- **St**anford **R**esearch **I**nstitute **P**roblem **S**olver
- Describes the state of the environment by a set of **predicates** which are true
- Models a problem as:
  - The **initial state** (a set of predicates which are true)

STRIPS planning

- **St**anford **R**esearch **I**nstitute **P**roblem **S**olver
- Describes the state of the environment by a set of **predicates** which are true
- Models a problem as:
  - The **initial state** (a set of predicates which are true)
  - The **goal state** (a set of predicates, specifying whether each should be true or false)

STRIPS planning

- **St**anford **R**esearch **I**nstitute **P**roblem **S**olver
- Describes the state of the environment by a set of **predicates** which are true
- Models a problem as:
  - The **initial state** (a set of predicates which are true)
  - The **goal state** (a set of predicates, specifying whether each should be true or false)
  - The set of **actions**, each specifying:

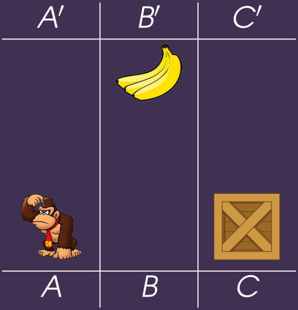
STRIPS planning

- **St**anford **R**esearch **I**nstitute **P**roblem **S**olver
- Describes the state of the environment by a set of **predicates** which are true
- Models a problem as:
    - The **initial state** (a set of predicates which are true)
    - The **goal state** (a set of predicates, specifying whether each should be true or false)
    - The set of **actions**, each specifying:
        - Preconditions (a set of predicates which must be satisfied for this action to be possible)

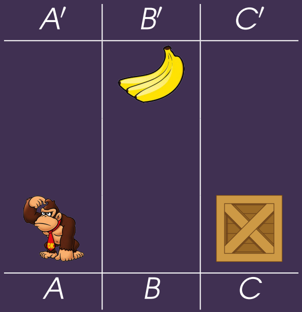
STRIPS planning

- **St**anford **R**esearch **I**nstitute **P**roblem **S**olver
- Describes the state of the environment by a set of **predicates** which are true
- Models a problem as:
  - The **initial state** (a set of predicates which are true)
  - The **goal state** (a set of predicates, specifying whether each should be true or false)
  - The set of **actions**, each specifying:
    - Preconditions (a set of predicates which must be satisfied for this action to be possible)
    - Postconditions (specifying what predicates are made true or false by this action)

STRIPS example



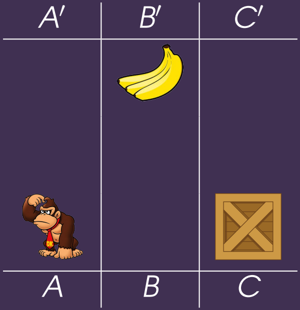
STRIPS example



Initial state:

```
At(A),
BoxAt(C),
BananasAt(B')
```

STRIPS example



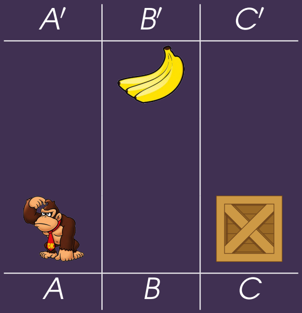
Initial state:

```
At(A),
BoxAt(C),
BananasAt(B')
```

Goal:

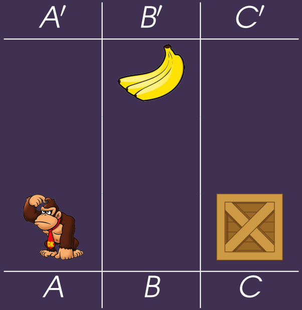
```
HasBananas
```

STRIPS example — Actions

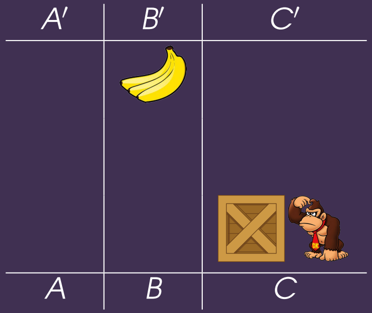



```
Move(x, y)
  Pre:   At(x)
  Post: !At(x), At(y)

ClimbUp(x)
  Pre:   At(x), BoxAt(x)
  Post: !At(x), At(x′)

ClimbDown(x′)
  Pre:   At(x′), BoxAt(x)
  Post: !At(x′), At(x)

PushBox(x, y)
  Pre:   At(x), BoxAt(x)
  Post: !At(x), At(y),
        !BoxAt(x), BoxAt(y)

TakeBananas(x)
  Pre:   At(x), BananasAt(x)
  Post: !BananasAt(x), HasBananas
```

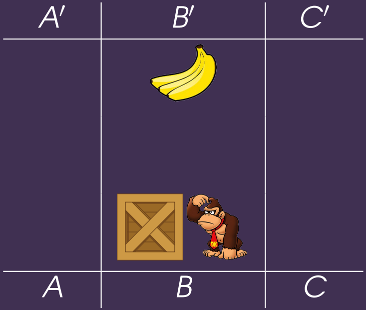
STRIPS example — Solution



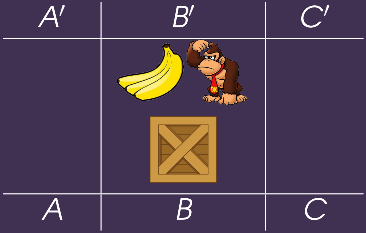
STRIPS example — Solution



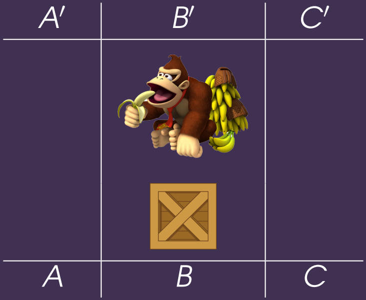
STRIPS example — Solution



STRIPS example — Solution



STRIPS example — Solution



Finding the solution

Finding the solution

- For a given state, we can construct a list of all **valid actions** based on their **preconditions**

Finding the solution

- For a given state, we can construct a list of all **valid actions** based on their **preconditions**
- We can also find the **next state** resulting from each action based on their **postconditions**

Finding the solution

- For a given state, we can construct a list of all **valid actions** based on their **preconditions**
- We can also find the **next state** resulting from each action based on their **postconditions**
- This should sound familiar (from 2 weeks ago)...

Finding the solution

- For a given state, we can construct a list of all **valid actions** based on their **preconditions**
- We can also find the **next state** resulting from each action based on their **postconditions**
- This should sound familiar (from 2 weeks ago)...
- We can construct a **tree** of states and actions

Finding the solution

- For a given state, we can construct a list of all **valid actions** based on their **preconditions**
- We can also find the **next state** resulting from each action based on their **postconditions**
- This should sound familiar (from 2 weeks ago)...
- We can construct a **tree** of states and actions
- We can then **search** this tree to find a goal state

Tree traversal

Tree traversal

procedure DEPTHFIRSTSEARCH

Tree traversal

procedure DepthFirstSearch
    let S be a stack

Tree traversal

procedure DepthFirstSearch

  let S be a stack

  push root node onto S

Tree traversal

procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty **do**

Tree traversal

procedure DEPTHFIRSTSEARCH
    let S be a stack
    push root node onto S
    while S is not empty **do**
        pop n from S

Tree traversal

procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty **do**
        pop n from S
        push children of n onto S

Tree traversal

procedure DEPTHFIRSTSEARCH
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        push children of *n* onto *S*
    end while
end procedure

procedure BREADTHFIRSTSEARCH

Tree traversal

procedure DEPTHFIRSTSEARCH
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        push children of *n* onto *S*
    end while
end procedure

procedure BREADTHFIRSTSEARCH
    let *Q* be a queue

Tree traversal

procedure DepthFirstSearch
  let S be a stack
  push root node onto S
  while S is not empty **do**
    pop n from S
    push children of n onto S
  end while
end procedure

procedure BreadthFirstSearch
  let Q be a queue
  enqueue root node into Q

Tree traversal

procedure DEPTHFIRSTSEARCH
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        push children of *n* onto *S*
    end while
end procedure

procedure BREADTHFIRSTSEARCH
    let *Q* be a queue
    enqueue root node into *Q*
    while *Q* is not empty **do**

Tree traversal

procedure DepthFirstSearch
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        push children of *n* onto *S*
    end while
end procedure

procedure BreadthFirstSearch
    let *Q* be a queue
    enqueue root node into *Q*
    while *Q* is not empty **do**
        dequeue *n* from *Q*

Tree traversal

procedure DepthFirstSearch
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        push children of *n* onto *S*
    end while
end procedure

procedure BreadthFirstSearch
    let *Q* be a queue
    enqueue root node into *Q*
    while *Q* is not empty **do**
        dequeue *n* from *Q*
        enqueue children of *n* into *Q*

Tree traversal

procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty **do**
        pop n from S
        push children of n onto S
    end while
end procedure

procedure BreadthFirstSearch
    let Q be a queue
    enqueue root node into Q
    while Q is not empty **do**
        dequeue n from Q
        enqueue children of n into Q
    end while
end procedure

Tree traversal example