Introduction to Artificial Intelligence – script

Faculty of Electrical Engineering and Computing

Last updated: 18. April 2024.

Table of Contents

Introduction to Artificial Intelligence – script....1
2. State Space Search....3
Blind search....4
Beadth-first search (BFS)....4
Uniform cost search (UCS)....4
Depth-first search....5
Depth-limited search....5
Iterative deepening search....5
3. Heuristic search....6
Greedy best-first search....6
Hill climbing search....7
A* algorithm....8
4. Game playing....10
4.1 Minimax method:....10
4.2 Minimax with heuristic....11
4.3 Alpha-beta pruning....11
5. Knowledge Representation using Formal Logic....12
5.1 Propositional/sentenial logic PL....12
5.2 First-order/predicate logic FOL....14
5.2.1 Mapping natural language to FOL....16
6. Automated resoning....19
7. Prolog....20

2. State Space Search

- starting from an initial state, we try to reach a goal state

Problem:

- S – set of states (state space)

- **search problem – problem = (s0, succ, goal)**

- s0 - initial state

- successor function (succ) – defines the state transitions (tells us into which state we can go from the current state)

- goal – test predicate to check if a state is a goal state

Search:

- state based search is basically a directed graph (digraph) search

- **search tree** – contains information about the node and the depth of the node

- **open/front nodes** – nodes that can still be expanded

- **closed nodes** – nodes that have already been fully expanded

- a node stores a state, as well as some additional date like state and depth of the node in the search tree

- **the search strategy is defined by the order in which the nodes are expanded!**

n = (s,d) => n=node; s=state; d=depth

state(n) = s

depth(n) = d

initial(s) = (s,0)

General search algorithm

```
function search(s0, succ, goal)
    open ← [initial(s0)]
    while open ≠ [] do
        n ← removeHead(open)
        if goal(state(n)) then return n
        for m ∈ expand(n, succ) do
            insert(m, open)
    return fail
```

- problem properties:

- |S| - number of states
- b – search tree branching factor
- d – depth of the optimal solution in the search tree
- m – maximum depth of the search tree (can be infinite)

- an algorithm is **complete** if and only if it can find a solution whenever the solution exists

- an algorithm is **optimal** only if and only if the solution the solution it finds is optimal (has the smallest cost)

- **time complexity** – number of generated nodes

- **space complexity** – number of stored nodes

- two search strategies:
  - blind (uninformed) search
  - heuristic search

Blind search

Beadth-first search (BFS)

- we expand the root node, then the children of that node, then the children of those nodes etc. (we never go to a lower level before expanding all the children on the current level)

- we insert child nodes at the END of the list of open nodes

- list “open“ functions as a FIFO queue

- has completeness and optimality BUT it is time and space complex

- time complexity: 1 + b + b^2 + b^3 … = O(b^d+1)

- space complexity: O(b^d+1)

Uniform cost search (UCS)

- similar to BFS, but accounts for transition costs

g(n) = c => cost of the transition to node n

- list open becomes a priority queue

- has completeness and optimality BUT also time and space complex like BFS

d = floor(C*/epsilon)

- time and space complexity: O(b^1+floor(C*/epsilon))

Depth-first search

- always expands the deepest node in the search tree
- inserts generated nodes at the beginning of the open nodes list
- “open“ list function as a stack (LIFO)
- not complete (can get stuck in a loop) or optimal (can find the solution that is not the best)
- space complexity: O(b*m) => linear space complexity is really good
- time complexity: O(b^m) => unfavorable if m>>d

m => maximum tree depth

Depth-limited search

- similar to depth-first search but stops at a given depth “k”
- not complete (solution could be before cutoff) or optimal
- space complexity: O(b*k)
- time complexity: O(b^k)
- this algorithm is useful if we know the solution depth

Iterative deepening search

- avoids the problem of choosing the optimal depth limit by trying out all possible values, starting with depth 0
- complete and optimal!
- space complexity: O(b*d)
- time complexity: O(b^d)

- if we want to reconstruct the path from the root node to the goal we have to still store the closed nodes to reconstruct the path, otherwise the child can’t point to the parent node if we have deleted the parent

3. Heuristic search

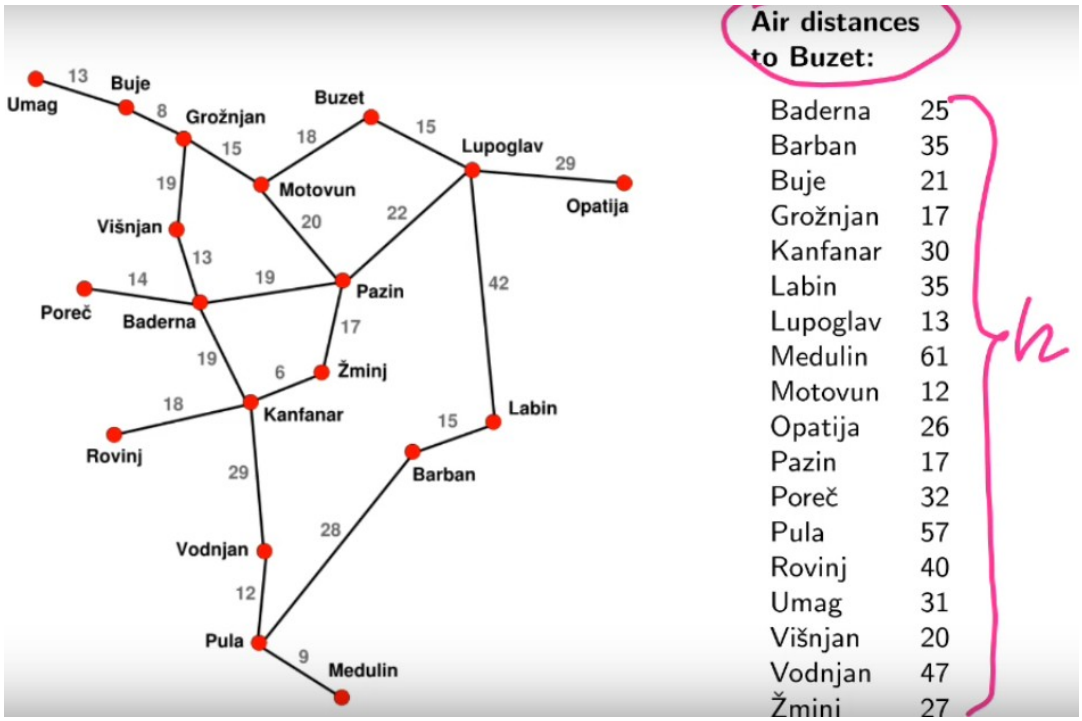
- the problem with blind search is that it only takes into account intial state, operators and the goal predicate, when a lot of the time we have other parameters that might help us get to our goal quicker

- heuristics – rules about the nature of the problem, whose purpose is to direct the search toward a goal so that it becomes more efficient

- heuristic function assigns to each state **an estimate** of the distance between that state and the goal state

- h(s) = [estimate number] (if estimate number is 0 then the state s is the goal state)

- for example, the heuristic function in the Istria example could be the air distance from a city to Buzet



Greedy best-first search

- always expands the node with the lowest heuristic value

Best-first search

```
function greedyBestFirstSearch(s0, succ, goal, h)
  open ← [initial(s0)]
  while open ≠ [] do
    n ← removeHead(open)
    if goal(state(n)) then return n
    for m ∈ expand(n, succ) do
      f = h(state(n))
      insertSortedBy(f, m, open)
  return fail
where f(n) = h(state(n))
```

- chooses the node that appears closest to the goal, disregarding the total path cost accumulated so far

- the chosen path is not optimal, but the algorithm does not backtrack to correct this

- the algorithm is not complete!! (it can get stuck in a loop, unless we use a visited states list)

- time and space complexity: O(b^m); b = branching factor; m = maximum depth of the search tree

Hill climbing search

- similar to greedy best-first search, but does not keep the generated nodes in history, it just expands the best child node (unless the values of child nodes are higher then the value of the parent node, then it literally gives up and gives us the parent node)

Hill-climbing search

```
function hillClimbingSearch(s0, succ, h)
  n ← initial(s0)
  loop do
    M ← expand(n, succ)
    if M = ∅ then return n
    m ← minimumBy(f, M)
    if f(n) < f(m) then return n
    n ← m
where f(n) = h(state(n))
```

- not complete (doesn’t guarantee that the solution will be found)

- easily trapped into a **local optima** (a value that is not a global but rather a local minimum)

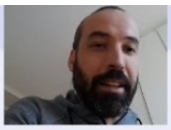
- time complexity: O(b^m)

- space complexity: O(1) → really good space complexity, this algorithm is more used when we aretrying to get a “good enough” result rather then the best one

A* algorithm

- similar to greedy best-first algorithm but also takes into account both the heuristic and the **path cost**

A* search



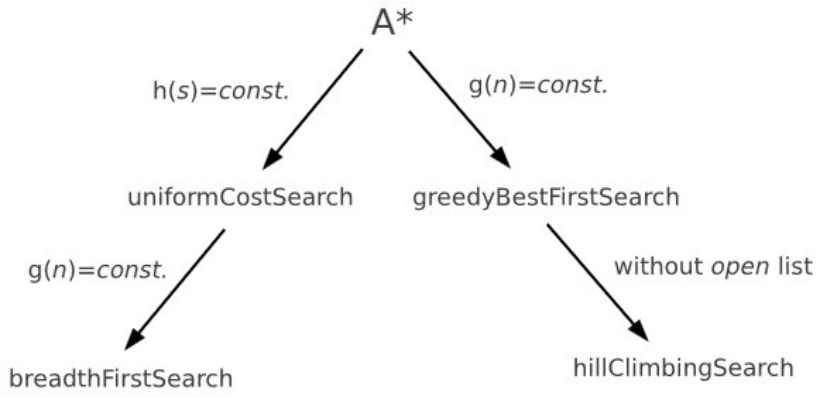
```
function aStarSearch(s0, succ, goal, h)
  open ← [initial(s0)]
  closed ← ∅
  while open ≠ [] do
    n ← removeHead(open)
    if goal(state(n)) then return n
    closed ← closed ∪ {n}
    for m ∈ expand(n) do
      if ∃m′ ∈ closed ∪ open such that state(m′) = state(m) then
        if g(m′) < g(m) then continue
        else remove(m′, closed ∪ open)
      insertSortedBy(f, m, open)
  return fail
where f(n) = g(n) + h(state(n))
```

- time and space complexity: O(min(b^(d+1), b|S|))

- it is both complete (because it accounts for path costs) and optimal (provided the heuristic h is optimistic)

- a heuristic is **optimistic** IFF it never overestimates, AKA the value is never greater then the true value needed to reach the goal

- the A* is NOT greedy because it considers the cost of the path from the initial node



- A* dominates uniform cost search and breadth-first search

- **consistent** heuristics = it is desireable that the function f(n) monotonically increases. In other words if we reach a certain state multiple times, the first time we encounter it should be the LOWEST cost, it shouldn't happen that later on we come across the same state using another path that is “cheaper” then the first one we encountered

- a consistent heuristic is not necessarily optimistic!

- A* variants:

  - without using a closed list
  - with closed list but without re-opening closed nodes
  - with closed list and without re-opening AND with pathmax correction:

    $$f(n) = \max(f(\text{parent}(n)), g(n) + h(\text{state}(n)))$$

Domination

Let A_1^* and A_2^* be optimal algorithms with *optimistic* heuristic functions h_1 and h_2. Algorithm A_1^* **dominates** algorithm A_2^* iff:

$$\forall s \in S.\ h_1(s) \geq h_2(s)$$

We also say that A_1^* is a **more informed** algorithm than A_2^*

- a good heuristic is:

  - optimistic – never overestimates the cost to reach the goal state
  - well informed
  - simple to compute

- if a heuristic is not optmistic we can go for a pessimistic heuristic – one which finds a “good enough” solution instead of the best one

4. Game playing

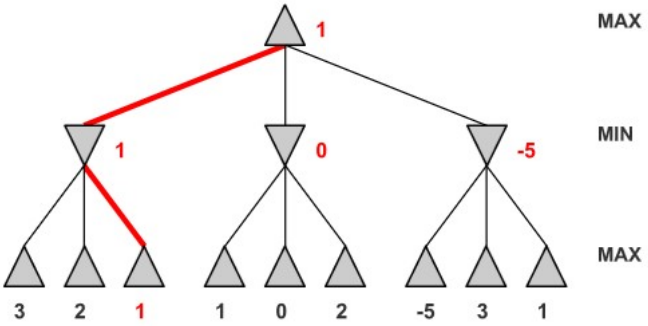
- in each move a player has to make an optimal move / has to find an optimal strategy

4.1 Minimax method:

- we have two players MIN (trying to minimize MAX's win) and MAX (trying to maxtimize his win)

- optimal strategy - one that ensures the highest win

- each player is trying to minimize the maximum loss




```
function maxValue(s)
  if terminal(s) then return utility(s)
  m ← −∞
  for t ∈ succ(s) do
    m ← max(m, minValue(t))
  return m
function minValue(s)
  if terminal(s) then return utility(s)
  m ← +∞
  for t ∈ succ(s) do
    m ← min(m, maxValue(t))
  return m
```

==========

For those who want to know more – minimax extensions/variations:

1) Negamax min(x,y) = -max(-x, -y)

2) Negascout – improves on alfa-beta pruning by considering how to sort the nodes

3) Expectimax

4) Monte Carlo tree search (MCTS) – used when we can’t really come up with a heuristic function

==========

4.2 Minimax with heuristic

- in reality we dont have the time to search the entire game tree

- we have to cut off the search at a certain depth d and use a heuristic to estimate the pay off (for example, in chess a heuristic can be the sum of values of player's chess pieces)

$$h(s) = w_1 x_1(s) + w_2 x_2(s) + \cdots + w_n x_n(s)$$

Minimax with a cut-off

```
function maxValue(s, d)
  if terminal(s) then return utility(s)
  if d = 0 then return h(s)
  m ← −∞
  for t ∈ succ(s) do
    m ← max(m, minValue(t, d − 1))
  return m

function minValue(s, d)
  if terminal(s) then return utility(s)
  if d = 0 then return h(s)
  m ← +∞
  for t ∈ succ(s) do
    m ← min(m, maxValue(t, d − 1))
  return m
```

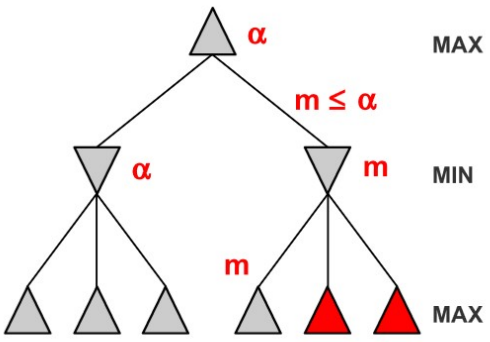
4.3 Alpha-beta pruning

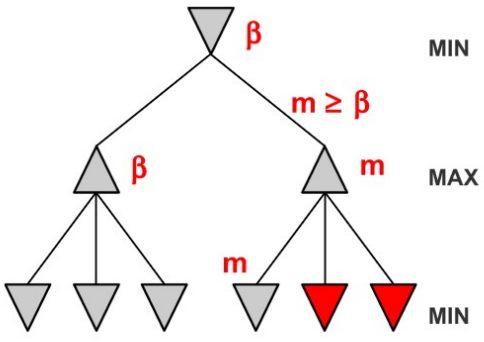
- the problem is that number of states increases exponentially with the number of turns, we can cut this number in half using alpha-beta pruning

- we prune every time that the unexplored nodes **can't be better** then the best move so far

- if pruning is below the MIN node: alpha pruning

- if pruning below the MAX node: beta pruning





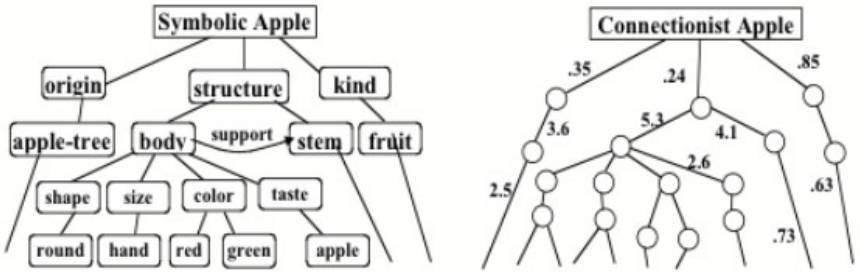
α – the largest MAX value found

β – the smallest MIN value found

5. Knowledge Representation using Formal Logic

- symbolism - external world represented with symbols

- connectionism - interaction of a large number of interconnected and simple processing units (for example neural network)



- formal logic system consists of:

- syntax - describes the structure
- semantics - describes the meaning of language structures
- proof theory - describes the mechanisms of inference (aka, how new conclusions can be derived from the premise)

- logic can be more (more details but really complex and hard to prove) or less expressive (less detail but simpler)

- **decidable** - logic is decidable if we can wheck the validity of ALL its formulae

5.1 Propositional/sentenial logic PL

- each fact is either true or false

Symbols of propositional logic

(1) Set of **propositional variables** or **atomic formulae**, $V = \{A, B, C, \dots\}$

(2) **Logical operators** or logical connectives:
- $\neg$ (negation)
- $\vee$ (operator *or*)
- $\wedge$ (operator *and*)
- $\rightarrow$ (implication)
- $\leftrightarrow$ (equivalence, biconditional)

(3) Logical constants $True$ and $False$, which denote a a proposition that is always true or false, respectively

(4) Parentheses (and)

Well-formed formula (wff)

A **well-formed formula** (wff) or simply **formula** of propositional logic is defined recursively as follows:

(1) An atom is a formula

(2) If F is a formula, then $(\neg F)$ is also a formula

(3) If F and G are formulae, then the following are also formulae:

- $(F \wedge G)$
- $(F \vee G)$
- $(F \rightarrow G)$
- $(F \leftrightarrow G)$

(4) Nothing else is a formula

- **interpretation** of of formula F is an assignment of truth values to variables from set V

Truth tables

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \rightarrow G$	$F \leftrightarrow G$
$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\bot$
$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$

- interpretation I is a **model** of formula F IFF F is true in I (in that case I also **satisfies** formula F)

- a model describes a single **world situation**

- **valid formula** - IFF it is true in all interpretations

- **inconsistent formula** - IFF it is false in every interpretation

- **satisfiable formula** - IFF it is true in AT LEAST one interpretation

- F is **equivalent** to G IFF truth values of both formulas are same in every interpretation of F and G

Logical consequence

Formula G is a **logical (semantic) consequence** of formulae $F_1, \ldots, F_n$ if and only if every interpretation that satisfies $F_1 \wedge \cdots \wedge F_n$ also satisfies G.

Put differently:
Formula G is a logical consequence of formulae $F_1, \ldots, F_n$ iff every model of $F_1 \wedge \cdots \wedge F_n$ also is a model of G.

We write $F_1, F_2, \ldots, F_n \vDash G$
and read "$F_1, \ldots, F_n$ **logically (semantically) entails** G".

Refutation method

Formula G is a logical consequence of $F_1, F_2, \ldots, F_n$ if and only if

$$F_1 \wedge \cdots \wedge F_n \wedge \neg G$$

if **inconsistent** (contradiction). In other words:

$$F_1 \wedge \cdots \wedge F_n \ \vDash \ G$$

if and only if

$$\vDash \neg\big(F_1 \wedge \cdots \wedge F_n \wedge \neg G\big)$$

5.2 First-order/predicate logic FOL

- PL: there are only facts which are either true or false

- FOL: there are objects, and there are relationships between objects

- relations can be n-ary:

  - 0-arity: proposutions, PL
  - unary: (STUDENT(John))
  - binary: LOVES(John, Marry)
  - ternary: GIVES(John, Marry, flowers)
  - ...

- higher expressivity comes at the price of **limited provability**

Term

(i) A constant is a term

(ii) A variable is a term

(iii) If f is an n-arity functional symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term

(iv) Nothing else is a term

- Examples of terms: 2, 3, $add(3, 4)$, $add(x, add(1, 4))$

Atom

If P is an n-arity predicate symbol and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is an **atom**. Nothing else is an atom.

- Examples of atoms: $LOVES(John, Marry)$, $GT(add(1, 2), 4)$

- **bound variable** - any variable that occurs within a scope of a quantifier that refers to that very same variable

- **free variable** - any variable that occurs not bound somewhere in the formula

Well-formed formula (wff)

A **well-formed formula** (wff) of FOL is defined recursively as follows:

(1) An atom is a formula

(2) If F is a formula, then $(\neg F)$ is also a formula

(3) If F and G are formulae, then so are $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$, and $(F \leftrightarrow G)$

(4) If F is a formula that contains a variable x that is not bound, then $(\forall x)F$ and $(\exists x)F$ are also formulae

(5) Nothing else is a formula

Wff or not?

1. $\forall x(L(x,z))$
2. $(\forall y)(\exists x)P(x,y,z)$
3. $\exists x P(x) \rightarrow Q(a)$
4. $\exists x\Big(P(x) \rightarrow \forall x \forall y Q(x,y)\Big)$
5. $\forall x \forall y\Big(\forall z P(z,x) \rightarrow Q(y,z)\Big)$

1) no - because of brackets around L)

2) yes - even though z is not present, it is fine for vairables to be free)

3) yes - a is free but thats okay)

4) no! - x is double bound to Q

5) yes

Interpretation of a FOL wff

An **interpretation** of a wff consists of:

- a non-empty **domain** D (the things being described)
- a mapping from each constant to an element from D
- a mapping from each n-ary function symbol to a function $D^n \rightarrow D$
- a mapping from each n-ary predicate symbol to a function $D^n \rightarrow \{\top, \bot\}$

- **extension** - subset of D^n for which the predicate is true

Evaluating the truth value of a formula

Given an interpretation, we can evaluate the truth value of a formula, as follows:

1. For an atom $F(x_1, \ldots, x_n)$, the truth value is given by applying the mappings for constants, functions, and predicate symbols
2. Truth values of $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$ are evaluated using truth tables (as in PL)
3. $(\forall x)F$ is evaluated as true iff F is true for every element $d \in D$
4. $(\exists x)F$ is evaluated as true iff F is true for at least one element $d \in D$

Question 1

Evaluate the truth value of the formule $\forall x \exists y P(x,y)$ in an interpretation with the domain $D = \{1,2\}$ and the following mapping for the predicate P:

$P(1,1)$	$P(1,2)$	$P(2,1)$	$P(2,2)$
$\top$	$\bot$	$\bot$	$\top$

Question 1 - for EVERY x, there is AT LEAST one y that satisfies the formula?

for x=1 and y=1 P is true

for x=2 and y=2 P is true

the formula is true!

Question 2

We are given an intepretation with the domain $D = \{a, b\}$ and the extension for P defined as $\{(a, a), (b, b)\}$. Evaluate the truth value of the following wffs:

1. $\exists x \forall y P(x, y)$
2. $\exists y \neg P(a, y)$
3. $\forall x \forall y P(x, y)$
4. $\exists x \neg P(x, y)$
5. $\forall x \forall y \Big(P(x, y) \rightarrow P(y, x) \Big)$

1 - false (for example for x=a, y=b is false)

2 - true (for example we can take y=b, P(a,b) gives us false and the inverse of it gives us true, therefore there exists AT LEAST one y variable that satisfies the formula)

3 - false

4 - false - y is free (it is a syntactically valid formula since free variables are allowe BUT we can't evaluate whether it is true or false)

5 - true

5.2.1 Mapping natural language to FOL

1. *John is a hardworking (H) student (S).*
   - $H(John) \wedge S(John)$ ✓
   - $HS(John)$
   - HSJ
   - $\forall x \big(John(x) \rightarrow (H(x) \wedge S(x)) \big)$

technically all of these are correct but they are overcomplicated except for the first one

2. *All students are clever (C).*
   - $\forall x\big(S(x) \wedge C(x)\big)$ ✗
   - $\forall student\ C(student)$ ✗
   - $\forall x \in student\ C(x)$ ✗
   - $\forall x\big(S(x) \rightarrow C(x)\big)$ ✓
   - $\forall x S(x) \rightarrow C(x)$ ✗

1 - it assumes that every element of the domain is a student and is clever... BUT what if an element of the domain isn't a student? then the formula is false

2 - we test if all elements of the domain are clever but we don't test if all elements are students

3 - not wff (the E looking sign is not valid in a formula)

4 - true

5 - x is free for C(x) which makes us unable to verify the formula

3. *No student is clever.*
   - $\forall x\big(S(x) \rightarrow \neg C(x)\big)$ ✓
   - $\forall x \neg\big(S(x) \wedge C(x)\big)$ ✓
   - $\neg\exists x\big(S(x) \wedge C(x)\big)$ ✓
   - $\neg\forall x\big(S(x) \rightarrow C(x)\big)$ ✗
   - $\neg\forall x\big(S(x) \wedge C(x)\big)$ ✗
   - $\forall x\big(\neg S(x) \wedge \neg C(x)\big)$ ✗

4 - false - will be true if at least one student is not clever but we are trying to test is ALL of them are not clever

5 - false - it assumes that all elements of x are students which does not have to be true, for example if there is a professor in x, the formula will always be false, and the negation of it will ALWAYS be true

6 - false - same as 5

4. *Some students are clever.*
   - $\exists x\big(S(x) \wedge C(x)\big)$ ✓
   - $\exists x\big(S(x) \rightarrow C(x)\big)$ ✗

2 - false - it will be true if any element in x is not a student, since equivalency returns true when the left side is false

6. Automated resoning

7. Prolog