COMP3702 Tutorial 4

Matt Choy | matthew.choy@uq.edu.au

Constraint Satisfaction Problems (CSPs)

- Subset of search problems
- Have the same assumptions about the world / environment as search problems:
  - A single-agent problem
  - Deterministic actions
  - Fully observed state
  - Typically discrete state spaces

Constraint Satisfaction Problems (CSPs)

- Subset of search problems
- Have the same assumptions about the world / environment as search problems:
    - A single-agent problem
    - Deterministic actions
    - Fully observed state
    - Typically discrete state spaces
- CSPs are specialised to identification problems, in which we try to provide assignments to variables within the problem.
- We want to find variable assignments that don’t violate any of the constraints imposed on the problem.

CSP Definition

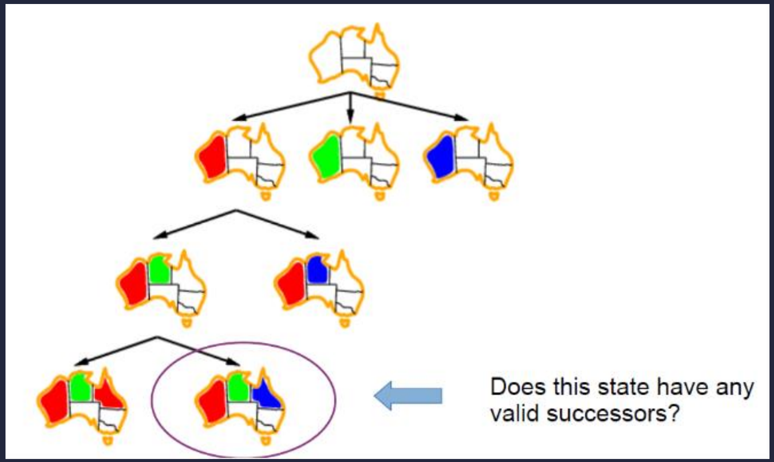
A CSP is given by:

- A set of variables $V_1, V_2, \cdots V_n$
- A domain – each variable V_i has an associated domain, dom_{V_i} of the possible values it can hold
- A set of constraints on various subsets of the variables, which are logical predicates specifying legal combinations of values for these variables.

A “model” of a CSP is an assignment of values variables that satisfies all of the constraints

A <u>model</u> is a solution to the CSP

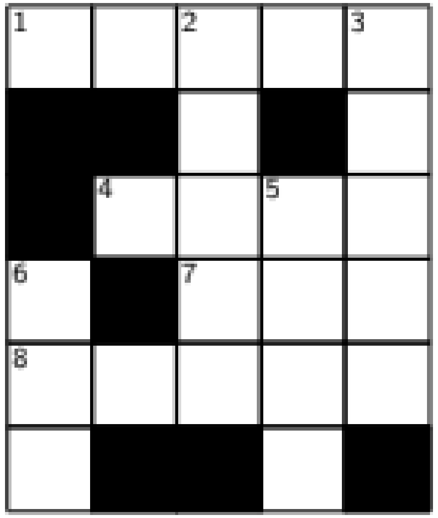
CSP Example – Map Colouring



1. Begin by colouring an initial state
2. Expand all of the possibilities for the next variable which meet the constraints of the search problem
3. Repeat until all the constraints have been met and the problem is solved.

Exercise 4.1

A **crossword puzzle** can be modeled as a CSP. Consider the puzzle below:



Words: AFT, ALE, EEL, HEEL, HIKE, HOSES, KEEL, KNOT, LASER, LEE, LINE, SAILS, SHEET, STEER, TIE.

Words must start at a position label number and run left-to-right or top-to-bottom only. Intersection boxes can contain only one letter. Each word in the list provided can be used only once. The words don't overlap, only intersect, so, e.g. 2 is 2-down only, and 7 is 7-across only (nb. this isn't a constraint, this is part of a typical crossword problem definition).

Exercise 4.1.

a) List the variables, and their domains, in the CSP representation of this crossword puzzle.

b) List the constraints in the CSP representation of the crossword puzzle.

Exercise 4.1 - Solution

Exercise 4.1a - Solution

Variables

- 1-Across, 2-Down, 3-Down, 4-Across, 5-Down, 6-Down, 7-Across, 8-Across

Domain

- {AFT, ALE, EEL, HEEL, HIKE, HOSES, KEEL, KNOT, LASER, LEE, LINE, SAILS, SHEET, STEER, TIE}

Exercise 4.1b – Solution

Constraints in Crossword CSP

- Domain Constraint Variables can be set to only the words of the correct length
- Binary Constraint Letters used at intersections must be equal
- Global Constraint Each word is only used once

Exercise 4.1b – Solution

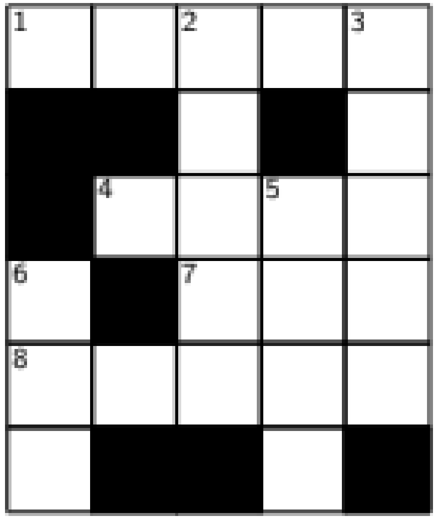
Constraints in Crossword CSP

- <u>Domain Constraint</u> Variables can be set to only the words of the correct length
- <u>Binary Constraint</u> Letters used at intersections must be equal
- <u>Global Constraint</u> Each word is only used once

The idea of constraints is that we want to use them to help is prune down the domains (possible values for each variable) and thus reduce the complexity of the problem.

Exercise 4.2

A **crossword puzzle** can be modeled as a CSP. Consider the puzzle below:



Words: AFT, ALE, EEL, HEEL, HIKE, HOSES, KEEL, KNOT, LASER, LEE, LINE, SAILS, SHEET, STEER, TIE.

Exercise 4.2.

Sketch out the bipartite graph, starting from 1A

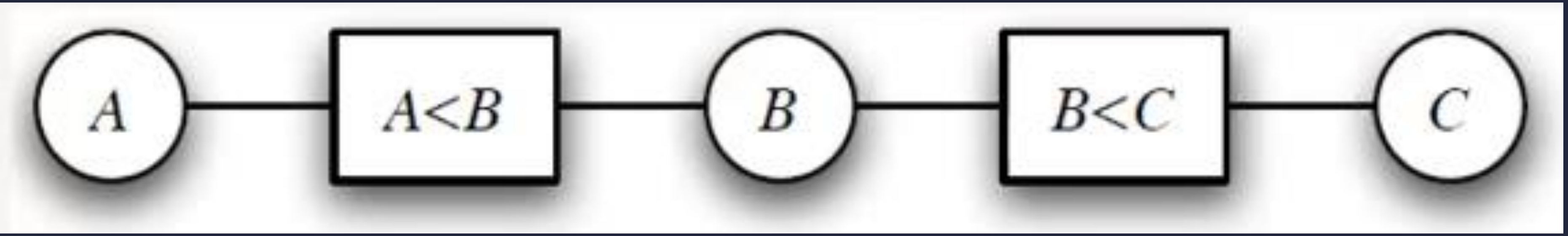
a) Start to sketch a constraint graph for the crossword CSP (you could use a tool like diagrams.net). Choose one variable, then sketch out the constraints that variable occurs in, and finally include any other variables that are also in scope of the constraints you have already drawn. The full constraint graph will be too large to draw easily.

b) Apply *domain consistency* to this CSP, and restate any variable domains that change. Does the structure of the constraint graph change?

Exercise 4.2a – Bipartite Graphs

Bipartite Graphs have:

- Circular or Oval-Shaped Nodes for each variable
- Rectangular Nodes for each constraint
- Domain of values associated with each variable
- Arc (Line) from variable X to each constraint that involves X



Exercise 4.2b – Domain Consistency

- General idea to prune the domains as much as possible before selecting values for them
- A variable is **domain consistent** if no value of the domain of the node is ruled impossible by any of the constraints

- Example: Is the scheduling example domain consistent?
  - `Variables` $X = \{A, B, C, D, E\}$ that represent the starting times of various activities
  - `Domains` Four start times for the activities

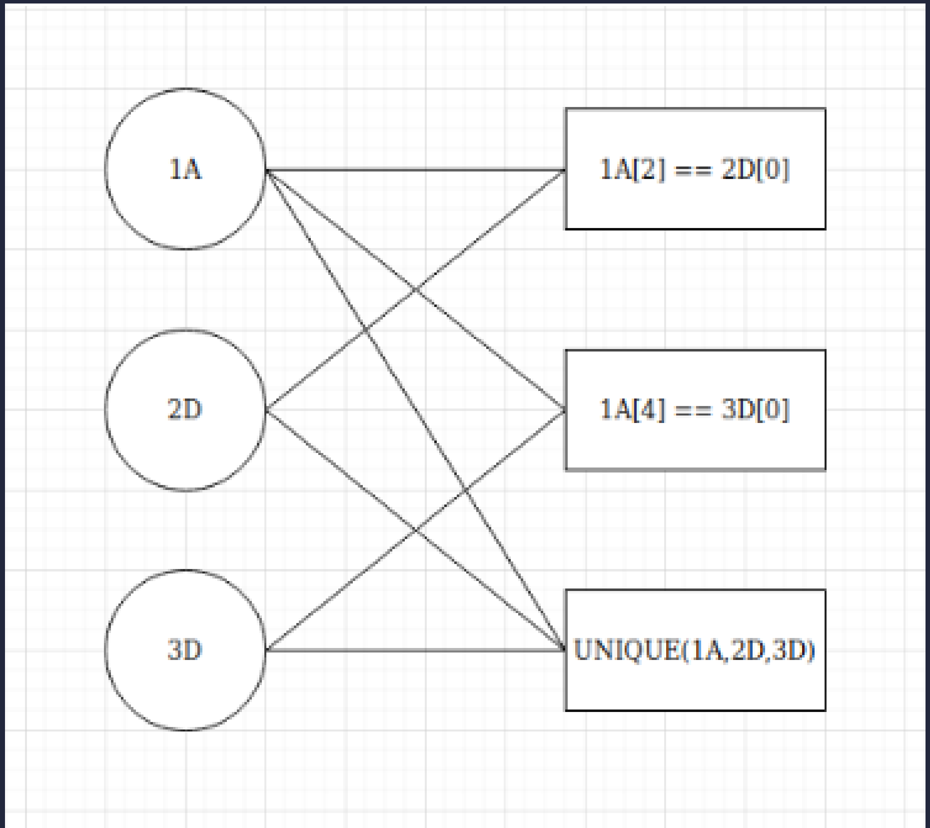
    $dom_A = \{1,2,3,4\}, dom_B = \{1,2,3,4\}, dom_C = \{1,2,3,4\}, dom_D = \{1,2,3,4\}$

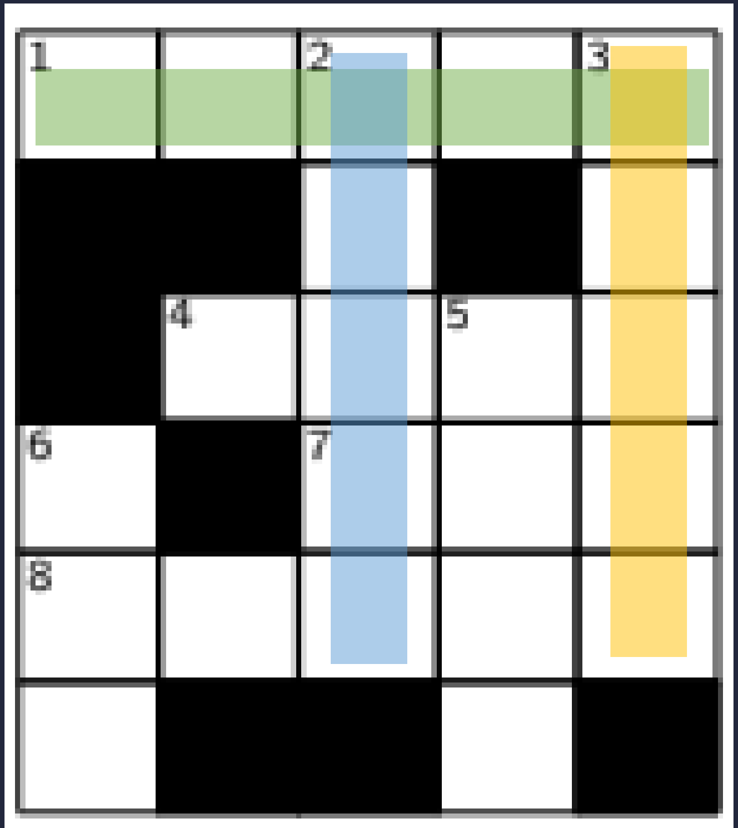
    Each of the four activities can choose from 1 of 4 start times, $\{1,2,3,4\}$
  - `Constraints` Represent illegal conflicts between variables

    $(B \neq 3), (C \neq 2), (A \neq B), (B \neq C), (C < D), (A = D), (E < A), (E < B), (B < C), (E < D), (B \neq D)$

Exercise 4.2 - Solution

Exercise 4.2a - Solution





Exercise 4.2b - Solution

In applying domain consistency, domains are reduced to those with answers with word lengths that are consistent with the number of blank spaces, so all variable domains are changed.

The updated domains are given as:

dom[1A] = {HOSES, LASER, SAILS, SHEET, STEER}
dom[2D] = {HOSES, LASER, SAILS, SHEET, STEER}
dom[3D] = {HOSES, LASER, SAILS, SHEET, STEER}
dom[4A] = {HEEL, HIKE, KEEL, KNOT, LINE}
dom[5D] = {HEEL, HIKE, KEEL, KNOT, LINE}
dom[6D] = {AFT, ALE, EEL, LEE, TIE}
dom[7A] = {AFT, ALE, EEL, LEE, TIE}
dom[8A] = {HOSES, LASER, SAILS, SHEET, STEER}

Exercise 4.2b - Solution

In applying domain consistency, domains are reduced to those with answers with word lengths that are consistent with the number of blank spaces, so all variable domains are changed.

The updated domains are given as:

dom[1A] = {HOSES, LASER, SAILS, SHEET, STEER}
dom[2D] = {HOSES, LASER, SAILS, SHEET, STEER}
dom[3D] = {HOSES, LASER, SAILS, SHEET, STEER}
dom[4A] = {HEEL, HIKE, KEEL, KNOT, LINE}
dom[5D] = {HEEL, HIKE, KEEL, KNOT, LINE}
dom[6D] = {AFT, ALE, EEL, LEE, TIE}
dom[7A] = {AFT, ALE, EEL, LEE, TIE}
dom[8A] = {HOSES, LASER, SAILS, SHEET, STEER}

At this point, we're not concerned about the constraints on letter intersections or word duplicates – that will come later.

Exercise 4.3 – Backtracking Search

- We want to systematically explore the domain by instantiating variables one at a time
- Evaluate each constraint predicate as soon as all of its variables are bound
- Any partial assignment that doesn’t satisfy the constraint can be pruned from the domain
- Essentially “brute forcing” our way to a solution.

Exercise 4.3 – Backtracking Search

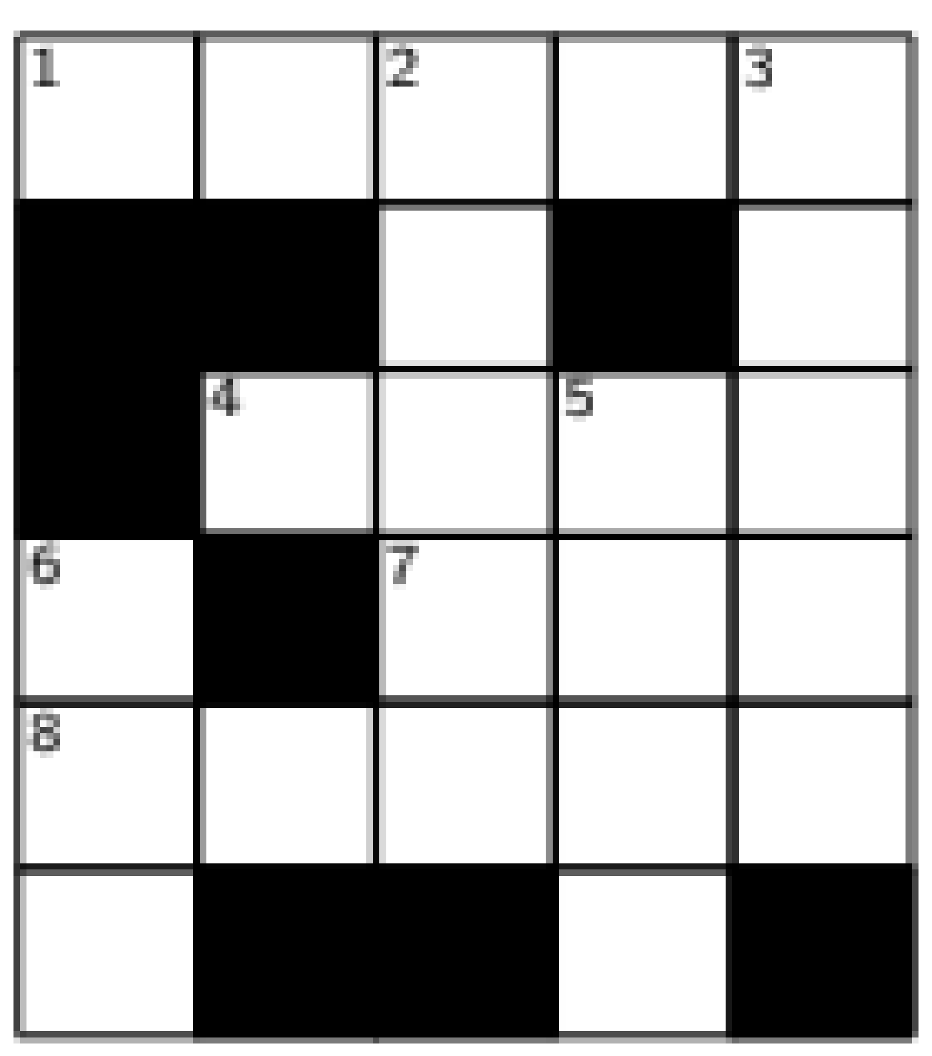
Exercise 4.3. Apply *backtracking search* to the domain-consistent constraint graph (pseudocode given below). Record the number of nodes expanded in the search procedure. You can trace the algorithm manually, e.g. by sketching a search tree, or develop code to answer this question (reusing some of your tree search code from previous weeks).

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
       if value is consistent with assignment given CONSTRAINTS[csp] then
           add {var = value} to assignment
           result ← RECURSIVE-BACKTRACKING(assignment, csp)
           if result ≠ failure then return result
           remove {var = value} from assignment
   return failure
```

https://gist.github.com/MattPChoy/a0d13ba4ca3515d3357f4055574efd70

Exercise 4.3 – Manual Solution



Exercise 4.3 – Code Solution

- Algorithm is implemented recursively
- The maximum recursive depth is determined by the number of variables.

```
class CrossWord:
  def __init__(self):
    self.constraint_checks = 0
    self.words = { # Set of possible variable allocations
        "AFT", "ALE", "EEL", "HEEL", "HIKE", "HOSES", "KEEL", "KNOT", "LASER", "LEE",
        "LINE", "SAILS", "SHEET", "STEER", "TIE"
    }

    self.vars = { # Variables and their current assignments;
                  # a map of variable names -> variable assignments
                  # (assignment from self.words)
        "1A": "", "2D": "", "3D": "", "4A": "",
        "5D": "", "6D": "", "7A": "", "8A": ""
    }
```

Exercise 4.3 – Code Solution

```
self.length_constraints = {
    "1A": 5, "2D": 5, "3D": 5, "4A": 4,
    "5D": 4, "6D": 3, "7A": 3, "8A": 5
}
self.intersect_constraints = {
    "1A": [("1A", 2, "2D", 0), ("1A", 4, "3D", 0)],
    "2D": [("2D", 0, "1A", 2), ("2D", 2, "4A", 1),
           ("2D", 3, "7A", 0), ("2D", 4, "8A", 2)],
    "3D": [("3D", 0, "1A", 4), ("3D", 2, "4A", 3),
           ("3D", 3, "7A", 2), ("3D", 4, "8A", 4)],
    "4A": [("4A", 1, "2D", 2), ("4A", 2, "5D", 0), ("4A", 3, "3D", 2)],
    "5D": [("5D", 0, "4A", 2), ("5D", 1, "7A", 1), ("5D", 2, "8A", 3)],
    "6D": [("6D", 1, "8A", 0)],
    "7A": [("7A", 0, "2D", 3), ("7A", 1, "5D", 1), ("7A", 2, "3D", 3)],
    "8A": [("8A", 0, "6D", 1), ("8A", 2, "2D", 4),
           ("8A", 3, "5D", 2), ("8A", 4, "3D", 4)]
}

self.domains = {
    k: [word for word in self.words if len(word) == self.length_constraints[k]]
    for k, v in self.vars.items()
}
```

Exercise 4.3 – Code Solution

```
def recursive_backtracking(env, assignment, expanded=0, verbose=False):
    unassigned = None # Locate an empty assignment
    for key, val in assignment.items():
        if val == "": # The variable hasn't been assigned;
            unassigned = key
            break
    if unassigned is None: # All variables have been assigned
        return assignment, expanded
    for word in env.domains[unassigned]:
        # Select a potential assignment
        assignment[unassigned] = word
        # Check assignments validity
        if not env.check_constraints(assignment):
                        # Find another assignment;
            continue
        # lock in current value and expand other nodes
        result, expanded = recursive_backtracking(env, {k: v for k, v in assignment.items()}, expanded + 1)
        if result is not None:
            if verbose:
                print("Number of backtracks", expanded)
            return result, expanded
    return None, expanded
```

Exercise 4.4 – Arc Consistency

Exercise 4.4.

a) Apply *arc-consistency* to this CSP (manually or in code; pseudocode given below). Record the number of arc-consistency check operations that are performed. What is the outcome of applying arc consistency?

b) If needed, apply backtracking search to the arc-consistent CSP.

c) Compare the number of search expansion and/or consistency check operations of backtracking search and (arc-consistency + backtracking search).

One *efficient* algorithm for arc-consistency is commonly called AC-3 (Source: R&N textbook):

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true
```

```
function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

Figure 6.3 The arc-consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be solved. The name "AC-3" was used by the algorithm's inventor (Mackworth, 1977) because it's the third version developed in the paper.

Exercise 4.4 – Arc Consistency

- The simplest form of constraint propagation, which repeatedly enforces local constraints.
- An arc $\langle X, r(X, \bar{Y}) \rangle$ is arc consistent of for every value $x \in dom(X)$ there is some value $\bar{y} \in dom(\bar{Y})$ such that the constraint $r(x, \bar{y})$ is satisfied.
- An arc $\langle X, r(X, \bar{Y}) \rangle$ is arc consistent if, for every value x of X, there is some allowed y.
- We want to prune (reduce) the domains using arc consistency.

Exercise 4.4 - Solution

Exercise 4.4 Solution

1. Begin with $dom_{1A} = \{HOSES, LASER, SAILS, SHEET, STEER\}$
   - Check for a value in dom_{2D} consistent with each element of dom_{1A}
   - We have the constraint that $1A[2] == 2D[0]$
   - Remove $\{SAILS, SHEET, STEER\}$ from dom_{1A} as they violate arc-consistency (are not consistent)
   - That is, set $dom_{1A} = \{HOSES, LASER\}$

   - Continue with checking – check for a value in dom_{3D} that is consistent with each element in dom_{1A}
   - Remove $\{LASER\}$ as it is not arc consistent.
   - Therefore, $dom_{1A} = \{HOSES\}$ as the other values are not arc consistent.

- Continue repeating this process – if you apply arc consistency correctly, it should solve the crossword puzzle.
- Even allowing for different variations of counting the number of operations, the AC-3 algorithm still solves the problems in many fewer operations than backtracking search

Exercise 4.4 Solution - Code

```python
def arc_consistency(env, verbose=False):
    constraint_checks = 0
    domains = {
        k: [word for word in env.words if len(word) == env.length_constraints[k]]
        for k, v in env.vars.items()
    }
    container = [x for k, v in env.intersect_constraints.items() for x in v]
    while container != []:
        constraint_checks += 1
        n1, i1, n2, i2  = container.pop(0)         # Revise the CSP
        prior_len = len(domains[n1])
        avail_letters1 = set(word[i1] for word in domains[n1])
        avail_letters2 = set(word[i2] for word in domains[n2])
        result = avail_letters1.intersection(avail_letters2)
        # Update the domain based on the revision
        domains[n1] = [word for word in domains[n1] if word[i1] in result]
        if domains[n1] == []:  # No solution exists
            return None
        elif len(domains[n1]) != prior_len:
            # If domain changes, reconsider constraints for neighbouring nodes
            for n, i, m, j in env.intersect_constraints[n1]:
                if m != n2:
                    # We shedule neighbour m to be updated with
                    # the new domain of n (our current node)
                    container.append((m, j, n, i))

    if verbose:
        print(f"Number of arc constraint checks: {constraint_checks}")

    return domains
```

Number of backtracks: 37
Number of arc-constraint checks: 54
Number of backtracking constraint checks: 167

Backtracking: 0.0005723941326141358 s
Arc: 0.00013731718063354493 s

Exercise 4.4 Solution - Code

```
env = CrossWord()

recursive_backtracking(env, {k: v for k, v in
env.vars.items()}, verbose=True)
arc_consistency(env, verbose=True)

print("Number of backtracking constraint checks",
env.constraint_checks)
```

Number of backtracks: 37
Number of arc-constraint checks: 54
Number of backtracking constraint checks: 167

Backtracking: 0.0005723941326141358 s
Arc: 0.00013731718063354493 s

```
import time
samples = 1000
env = CrossWord()
start = time.time()
for i in range(samples):
    recursive_backtracking(env, {k: v for k, v in env.vars.items()})
print("backtracking", (time.time() - start) / samples)
start = time.time()
for i in range(samples):
    arc_consistency(env)

print("arc", (time.time() - start) / samples)
```