Approaches to testing

- Black Box Testing: Test without looking at the code/hardware
- White Box Testing (clear box testing): Test the internal structure of the software
It is all about coverage

- Black box testing: test by covering the specification
- White box testing: test by covering the source code
  - Execution paths
  - Statements
  - ...
Approach in the book

- Model your artefact. Typically as a graph.
- Unify different notions of coverage as some mathematical properties of graphs.
Does this program halt?

```c
main () {
    int i = 0;
    int z = 0;
    for (i = 0; i < 10; i++) {
        z = z + 1;
    }
}
```
Turing halting problem

Does this program halt?

```c
main() {
    int i = 0;
    int z = 0;
    while (1 != 0) {
        z = z + 1;
    }
}
```
Turing halting problem

Can I write a program that takes *any* program and decides if it halts?

Seems that is might be possible, but it is mathematically impossible.
Turing halting problem

Proof

- Enumerate all programs. There are infinitely many, but still a countable number.
- Give each program a number.
- The function

\[ h(i, x) = \begin{cases} 
1 & \text{if program } i \text{ halts on input } x \\
0 & \text{otherwise} 
\end{cases} \]

is not computable.
Given

\[ h(i, x) = \begin{cases} 
1 & \text{if program } i \text{ halts on input } x \\
0 & \text{otherwise} 
\end{cases} \]

Define

\[ g(i) = \begin{cases} 
0 & \text{if } h(i, i) = 0 \\
\text{loop forever} & \text{otherwise} 
\end{cases} \]

What of \( h(g, g) \)? Two possibilities:

- \( h(g, g) = 1 \) then \( g \) halts on input \( g \), so \( g(g) = 0 \) which implies \( h(g, g) = 0 \), hence Contradiction.
- \( h(g, g) = 0 \) then \( g \) loops for ever on input \( g \). Which implies \( h(g, g) \) is not equal to 0, hence contradiction.

Proof strategy is often referred to as a diagonal argument.
Common Caveats

- Halting function should work on *all* programs.
- Finite memory, finite number of registers a computer is just a finite state machine.
- So it is possible to write a function that decides if all programs up to a given size terminate, but not very efficiently.
Rice’s Theorem

- All interesting properties are non-computable.
- Ask yourself: Is what I’m trying to do equivalent to the halting problem?
- Are all execution paths covered?
  - If you could solve that problem, then you would solve the halting problem.
- This is the origin of

  “Program testing can be used to show the presence of bugs, but never to show their absence!” Edsger Dijkstra.
Pragmatics

- Admit we can not decide properties on all programs.
- Do our best on most programs.
White Box testing

Coverage criteria include:

▶ Function coverage - Has each function (or subroutine) in the program been called?
▶ Statement coverage - Has each statement in the program been executed?
▶ Branch coverage - Has each branch of each control structure (such as in if and case statements) been executed?
Graphs

Book unifies different notions of source coverage by properties of graphs.

- Graphs can come from many sources
  - Control flow graphs
  - Design structure
  - FSMs and statecharts
  - Use cases

- Tests usually are intended to cover the graph in some way

We will look at control flow graphs.
Book’s definition of a graph

This is a bit non-standard.

- A set of nodes $N$
- A set of initial nodes $N_o$.
- A set of final nodes $N_f$
- A set of directed edges $E$ of the form $(n, m)$ where $n$ and $m$ are taken from $N \cup N_o \cup N_f$.

Normally a graph is just a pair $(N, E)$. 

Example Graph

- $N_o = \{0\}$
- $N_f = \{3\}$
- $N = \{1, 2\}$
- $E = \{(0, 1), (0, 2), (1, 3), (2, 3)\}$
Example Graph

- $N_o = \{0, 1, 2\}$
- $N_f = \{8, 9, 10\}$
Other definitions possible.

- A path is a sequence of nodes.
- The length of a path is the number of edges. A path with only one node, hence no edge has length 0.
- Subpath is sub-sequence of a path.
- $\text{Reach}(n)$ the set of nodes that can be reach via a directed path from the node $n$. 
Paths

Paths include: [0, 3, 8], [0, 4, 8], [0, 4, 9], [1, 4, 8], ..., [4, 8], ....
Reach\((n)\)

- \(\text{Reach}(3) = \{8\}\), \(\text{Reach}(1) = \{4, 9, 8, 6, 10\}\).
Two notions in program analysis:
  ▶ Syntactic reach.
  ▶ Semantic Reach.

```c
main() {
    for (i = 0; true; i++) {
        if (f(i) == 0) { break; }
    }
    X();
}
```

`X()` is syntactically reachable, but semantically you have to infer something about `X()`.
Test Path

- A test path starts at an initial node and ends in a final node.
- Test paths represent execution of test cases.
  - Some paths can be executed by many test cases.
  - Some paths can not be executed by any test cases (halting problem again).
Visiting and Touring

- A path visits node $n$ is $n$ is in the path.
- A path visits an edge $e = (n, m)$ if the $[n, m]$ is a sub-path.
- A path tours a path if that path is a sub-path.
Tests and Test paths

- Given a test \( t \) then \( \text{path}(t) \) is the path executed by \( t \).
- Given a set of tests \( T \) then \( \text{path}(T) \) is the set of paths executed by tests in \( T \).

Two notions of reach:
- A location in a graph (node or edge) can be reached from another location if there is a sequence of edges from the first location to the second
- Syntactic reach : A subpath exists in the graph
- Semantic reach : A test exists that can execute that subpath
Test and Test Paths

Many to one. Deterministic software, each test path has identical execution.

Many to many, non-deterministic software (you’ll meet it all the time) a test can execute many test paths.
Testing and Covering Graphs

- **Test Requirements (TR)**: Describe properties of test paths
- **Test Criterion**: Rules that define test requirements
- **Satisfaction**: Given a set TR of test requirements for a criterion \( C \), a set of tests \( T \) satisfies \( C \) on a graph if and only if for every test requirement in TR, there is a test path in \( \text{path}(T) \) that meets the test requirement.

General idea in testing. Define your test requirements separately from the tests cases. Reformulate your requirements into test criteria and then try to find test paths that satisfy your test criteria.
Control Flow Graphs

Control flow graphs models the control structures of the program.

- Nodes: Statements of sequence of statements
- Edges: Transfer of control
- Basic Block: Sequence of statements with no transfer of control.
If statements

```c
if (x < y)
{
    y = 0;
} else
{
    x = y;
}
```
if (x < y) {
    y = 0;
};
If return statements

```c
if (x < y)
{
    return;
}
print(x);
return;
```

Note that we do not collapse the two return statements.
Loops

```plaintext
for (i = 0; i < x; i++) {
    loop_body();
}
```
- See the book for the rest of the constructs.
- Each node only to be labelled with one basic block.
- Beware of hidden control structures. (C’s case statement).
Node Coverage

- Node Coverage (NC) : Test set $T$ satisfies node coverage on graph $G$ iff for every syntactically reachable node $n$ in $N$, there is some path $p$ in $\text{path}(T)$ such that $p$ visits $n$. 
Edge Coverage

- Edge Coverage (EC) : TR contains each reachable path of length up to 1, inclusive, in $G$.

Is there any difference between node and edge coverage?
Difference between node and edge coverage

- **Node coverage**
  - Test requirement (TR) = \{0, 1, 2\}.
  - Test path = [0, 1, 2].

- **Edge Coverage**
  - Test requirement (TR) = \{(0, 1), (0, 2), (1, 2)\}.
  - Test paths = [0, 1, 2], [0, 2].
Complete Path Coverage

- Require that all paths are covered.

Often there are too many paths. So various approximations.
- Require that all paths up to length $k$ are covered.
  - $k = 0$, node coverage.
  - $k = 1$, edge coverage.
  - $k = 2$, edge pair coverage.
Structural Coverage Example

- **Node Coverage:** $TR = \{0, 1, 2, 3, 4, 5, 6\}$, test paths $= \{[0, 1, 2, 3, 6], [0, 1, 2, 4, 5, 4, 6]\}$.

- **Edge Coverage:** $TR = \{(0, 1), (0, 2), (1, 2), (2, 3), (2, 4), (3, 6), (4, 5), (4, 6), (5, 4)\}$
  Test paths $[0, 1, 2, 3, 6], [0, 2, 4, 5, 4, 6]$.

- **Complete Path Coverage.** Test paths $[0, 1, 2, 3, 6], [0, 1, 2, 4, 6], [0, 1, 2, 4, 5, 4, 6], [0, 1, 2, 4, 5, 4, 6], \text{ etc.}$
Structural Coverage Example

Edge-Pair Coverage: \( TR = \{ [0, 1, 2], [0, 2, 3], [0, 2, 4], [1, 2, 3], [1, 2, 4], [2, 3, 6], [2, 4, 5], [2, 4, 6], [4, 5, 4], [5, 4, 5], [5, 4, 6] \} \)

Test Paths
- \([0, 1, 2, 3, 6], [0, 1, 2, 4, 6], [0, 2, 3, 6]\)
- \([0, 2, 4, 5, 4, 5, 4, 6]\).
Model, define, and approximate

- Model what you want to test.
- Define coverage criteria.
- If coverage criteria is undecidable or requires too many test cases then approximate.
Separate test requirements and test cases

- I told you to have a reason for a test.
- Test requirements are the reasons for tests.
- You need to find test cases satisfying test cases.