Topics that will be covered in this lecture series:

Automated analysis

- Graphs
- Markov chains
- Markov decision processes
- Games
- Timed Automata

**Example:**

\[ S = \{1, 2, \ldots, n\} \]

\[ E = \{(i,j) \mid 1 \leq i \leq n, 1 \leq j \leq n\} \]

\[
\begin{array}{ccc}
1 & 2 & 3 \\
E &=& \{(1,1), (1,2), (2,1), (2,3), (3,2)\}
\end{array}
\]

\[
E = \{(1,1) (1,2) 2 \\
(2,1) (2,3) 2 \}
\]

\[
E = \{(3,2)\} 3 \\
0 \}
\]

**n x n Matrix**

\[
E [i,j] = \{1 \text{ (if there is any edge from } i \text{ to } j) \}
\]

0\}
Example:

(S, E)  
\[ S = \{1, 2, 3, 4, 5\} \]

T = target states  
\[ T = \{2, 4\} \]

Example:

Suppose we wish to obtain the DNA sequence S of an organism. The Sequence by Hybridization technique gives us the set \( L = \{s_1, s_2, \ldots, s_m\} \) of all substrings of S of length k, for some k large enough such that every substring \( s_i \) appears exactly once in S. Our task is to infer S given L.

Given a graph G, an Eulerian path in G is a path that traverses each edge exactly once. Given L, we construct a graph G such that every node of G is a substring of length k-1 that is a substring of an element in L. There is an edge from u to v labeled with some letter t if and only if v is a suffix of the string obtained by appending t to u. It is easy to see that the original string S can be reconstructed by finding an Eulerian path in G. Then S is exactly the string obtained by considering the first node of the path, together with the labels of the edges it traverses, in the order they are traversed.

Objectives:

Our analysis on graphs will be done with respect to some objectives. Formally, an objective is a (possibly infinite) set of infinite paths satisfying some property, and given an objective, our goal is to identify the initial states of paths from it (i.e., states from which the objective is satisfied). The set of all infinite valid paths in a graph \( G = (V, E) \) is some set \( I \subseteq V^\omega \) (that is, an infinite juxtaposition of elements from V that respects the edge relation). Our objectives are defined based on some set of target nodes \( T \).

- Reachability \( \Diamond T = \{ p \in I: \exists i \text{ such that } p[i] \in T \} \), that is, the set of paths that at some point visit \( T \).
- Liveness \( \Box \Diamond T = \{ p \in I: \forall i \exists j > i \text{ such that } p[j] \in T \} \), that is, the set of paths that hit \( T \) infinitely often. The Liveness objective is often called Buchi objective.
- Safety \( \Box \Diamond T = I \setminus \Diamond (V \setminus T) \), that is, the set of paths that always visit nodes only from \( T \).
- Persistence \( \Diamond \Box T = I \setminus \Box \Diamond (V \setminus T) \), that is, the set of paths eventually visit nodes only from \( T \).

Example:

How can we compute the set of targets state in the diagram below? We want to find a trajectory to the target states (reachability)
- reach in $\leq k$ steps $\beta T_{old}$
- reach in $\leq k + 1$ steps $\beta T_{new}$

Basic algorithm:

```plaintext
while (flag == true)
{[flag := false]
for i = 1 to n
    {
        for j=1 to n
            {if (T_{old}[i] = 1)
                then  T_{new}[i] = 1)
            else
                if (E[i,j] = 1 ^ T_{old}[j] = 1)
                    then
                        T_{new}[i] = 1
                        flag == true
```
Notes: the original algorithm did not stop since it contained no stopping criteria. Therefore, a flag was added that is set to “true” when $T_{old}$ can not be reached.

The two for-loops iterate through all the nodes of the graph, hence each one is repeated $n$ times. Because they are nested, the outermost for-loop has quadratic complexity $O(n^2)$. The while loop will be executed once for each nodes that is discovered. Since in the works case all nodes can be discovered, the while loop can be repeated $n$ times, and by our analysis, each time the outermost for-loop has $O(n^2)$ iterations, hence the overall complexity of the algorithm is $O(n^3)$.

However, there are still “stupid” things in the algorithm that make it slower than it has to be:

The argument:

if ($T_{old}[i] = 1$)
then $T_{new}[i] = 1$)

should be moved up, directly after the first “for” statement;

Example:

(S,E)

$i \in S$

$B(i)$ à compute set of states with path to $i$

$T = \{i\}$

$F(i)$ à set of states $j$ that $i$ can reach , path from $i$ to $j$

ReachAlgo ($T$)

for $j = 1$ to $n$

{ $B_j = \text{ReachAlgo} (G, \{j\})$

if ($B_j[i] = 1$)
then $F[j] = 1;$

} à the algorithm has complexity $O(n^4)$. Intuitively, this means that if a graph has $n$ nodes, the algorithm will finish in around $cn^4$ steps for some constant $c$.

Alternative:

à reverse the edges
for i = 1 to n
    for j = 1 to n
        if (E[i,j] = 1)
            then
                E^R[j,i] = 1
            else
                E^R[j,i] = 0
    F:= ReachAlgo ((S, E^R), {i})

Instead of O(n^4), this algorithm has complexity O(n^3). So, for a graph of n nodes, it will finish in around cn^3 steps for some constant c. Note that this is faster than the previous algorithm.

**Strongly connected components:**

A Strongly connected component is a set C of states that for all i and j in C there is a path from i to j only going through nodes in C.
A maximal strongly connected component is a strongly connected component C with the property that every strict superset C’ (i.e., C⊂C’) is not a strongly connected component.

e.g.

C1 and C2

- C1 n C2 is not empty
- C1 u C2 is also a strongly connected component

à how do you show this?
Pick $i$ and $j \in C_1 \cup C_2$ à this will create another strongly connected component

$C(i) = B(i) \cap F(i)$

à $C(i)$ is strongly connected

states $j \& k \in C(i)$ j à k

since $j \in B(i)$ j à i

$k \in F(i)$ k à i

à $C / C(i)$ is not empty

pick $j \in C / C(i)$

path from i to j , then $j \in F(i)$

path from j to i, then $j \in B (i)$

$j \in B(i)$ and $F(i) = C(i)$

LivenessAlgo (T)
- compute SCC’s
- good SCC’s à contain a non-trivial target state (meaning that it does not only has a path to itself and the loop can go on forever