Formal Languages, Automata and Codes

Oleg Gutik

Lecture 5

2.2 Nondeterministic Finite Accepters

If you examine the automata we have seen so far, you will notice a common feature: a unique transition is defined for each state and each input symbol. In the formal definition, this is expressed by saying that δ is a total function. This is the reason we call these automata deterministic. We now complicate matters by giving some automata choices in some situations where more than one transition is possible. We shall call such automata nondeterministic.

Nondeterminism is, at first sight, an unusual idea. Computers are deterministic machines, and the element of choice seems out of place. Nevertheless, nondeterminism is a useful concept, as we will see.

If you examine the automata we have seen so far, you will notice a common feature: a unique transition is defined for each state and each input symbol. In the formal definition, this is expressed by saying that δ is a total function. This is the reason we call these automata deterministic. We now complicate matters by giving some automata choices in some situations where more than one transition is possible. We shall call such automata nondeterministic.

Nondeterminism is, at first sight, an unusual idea. Computers are deterministic machines, and the element of choice seems out of place. Nevertheless, nondeterminism is a useful concept, as we will see.

If you examine the automata we have seen so far, you will notice a common feature: a unique transition is defined for each state and each input symbol. In the formal definition, this is expressed by saying that δ is a total function. This is the reason we call these automata deterministic. We now complicate matters by giving some automata choices in some situations where more than one transition is possible. We shall call such automata nondeterministic.

Nondeterminism is, at first sight, an unusual idea. Computers are deterministic machines, and the element of choice seems out of place. Nevertheless, nondeterminism is a useful concept, as we will see.

If you examine the automata we have seen so far, you will notice a common feature: a unique transition is defined for each state and each input symbol. In the formal definition, this is expressed by saying that δ is a total function. This is the reason we call these automata deterministic. We now complicate matters by giving some automata choices in some situations where more than one transition is possible. We shall call such automata nondeterministic.

Nondeterminism is, at first sight, an unusual idea. Computers are deterministic machines, and the element of choice seems out of place. Nevertheless, nondeterminism is a useful concept, as we will see.

If you examine the automata we have seen so far, you will notice a common feature: a unique transition is defined for each state and each input symbol. In the formal definition, this is expressed by saying that δ is a total function. This is the reason we call these automata deterministic. We now complicate matters by giving some automata choices in some situations where more than one transition is possible. We shall call such automata nondeterministic.

Nondeterminism is, at first sight, an unusual idea. Computers are deterministic machines, and the element of choice seems out of place. Nevertheless, nondeterminism is a useful concept, as we will see.

If you examine the automata we have seen so far, you will notice a common feature: a unique transition is defined for each state and each input symbol. In the formal definition, this is expressed by saying that δ is a total function. This is the reason we call these automata deterministic. We now complicate matters by giving some automata choices in some situations where more than one transition is possible. We shall call such automata nondeterministic.

Nondeterminism is, at first sight, an unusual idea. Computers are deterministic machines, and the element of choice seems out of place. Nevertheless, nondeterminism is a useful concept, as we will see.

If you examine the automata we have seen so far, you will notice a common feature: a unique transition is defined for each state and each input symbol. In the formal definition, this is expressed by saying that δ is a total function. This is the reason we call these automata deterministic. We now complicate matters by giving some automata choices in some situations where more than one transition is possible. We shall call such automata nondeterministic.

Nondeterminism is, at first sight, an unusual idea. Computers are deterministic machines, and the element of choice seems out of place. Nevertheless, nondeterminism is a useful concept, as we will see.

If you examine the automata we have seen so far, you will notice a common feature: a unique transition is defined for each state and each input symbol. In the formal definition, this is expressed by saying that δ is a total function. This is the reason we call these automata deterministic. We now complicate matters by giving some automata choices in some situations where more than one transition is possible. We shall call such automata nondeterministic.

Nondeterminism is, at first sight, an unusual idea. Computers are deterministic machines, and the element of choice seems out of place. Nevertheless, nondeterminism is a useful concept, as we will see.

If you examine the automata we have seen so far, you will notice a common feature: a unique transition is defined for each state and each input symbol. In the formal definition, this is expressed by saying that δ is a total function. This is the reason we call these automata deterministic. We now complicate matters by giving some automata choices in some situations where more than one transition is possible. We shall call such automata nondeterministic.

Nondeterminism is, at first sight, an unusual idea. Computers are deterministic machines, and the element of choice seems out of place. Nevertheless, nondeterminism is a useful concept, as we will see.

If you examine the automata we have seen so far, you will notice a common feature: a unique transition is defined for each state and each input symbol. In the formal definition, this is expressed by saying that δ is a total function. This is the reason we call these automata deterministic. We now complicate matters by giving some automata choices in some situations where more than one transition is possible. We shall call such automata nondeterministic.

Nondeterminism is, at first sight, an unusual idea. Computers are deterministic machines, and the element of choice seems out of place. Nevertheless, nondeterminism is a useful concept, as we will see.

If you examine the automata we have seen so far, you will notice a common feature: a unique transition is defined for each state and each input symbol. In the formal definition, this is expressed by saying that δ is a total function. This is the reason we call these automata deterministic. We now complicate matters by giving some automata choices in some situations where more than one transition is possible. We shall call such automata nondeterministic.

Nondeterminism is, at first sight, an unusual idea. Computers are deterministic machines, and the element of choice seems out of place. Nevertheless, nondeterminism is a useful concept, as we will see.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta \colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta \colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta \colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta\colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple
$$M = (Q, \Sigma, \delta, q_0, F),$$
where Q, Σ, q_0, F are defined as for deterministic finite accepters, but
$$\delta \colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and
$$\delta(q_1, a) = \{q_0, q_2\},$$
then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta \colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta \colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta \colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta \colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta \colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta\colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta \colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta \colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta \colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta \colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta\colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta \colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta\colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta\colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta \colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta \colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta\colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A *nondeterministic finite accepter* or *NFA* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q_0, F).$$

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta\colon Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a DFA. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the NFA. Also, we allow λ as the second argument of δ. This means that the NFA can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an NFA, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Like NFA's, nondeterministic accepters can be represented by transition graphs. The vertices are determined by Q, while an edge (q_i, q_j) with label a is in the graph if and only if $\delta(q_i, a)$ contains q_j. Note that since a may be the empty string, there can be some edges labeled λ.

A string is accepted by an NFA if there is some sequence of possible moves that will put the machine in a final state at the end of the string. A string is rejected (that is, not accepted) only if there is no possible sequence of moves by which a final state can be reached. Nondeterminism can therefore be viewed as involving "intuitive" insight by which the best move can be chosen at every state (assuming that the NFA wants to accept every string).

Like NFA's, nondeterministic accepters can be represented by transition graphs. The vertices are determined by Q, while an edge (q_i, q_j) with label a is in the graph if and only if $\delta(q_i, a)$ contains q_j. Note that since a may be the empty string, there can be some edges labeled λ.

A string is accepted by an NFA if there is some sequence of possible moves that will put the machine in a final state at the end of the string. A string is rejected (that is, not accepted) only if there is no possible sequence of moves by which a final state can be reached. Nondeterminism can therefore be viewed as involving "intuitive" insight by which the best move can be chosen at every state (assuming that the NFA wants to accept every string).

Like NFA's, nondeterministic accepters can be represented by transition graphs. The vertices are determined by Q, while an edge (q_i, q_j) with label a is in the graph if and only if $\delta(q_i, a)$ contains q_j. Note that since a may be the empty string, there can be some edges labeled λ.

A string is accepted by an NFA if there is some sequence of possible moves that will put the machine in a final state at the end of the string. A string is rejected (that is, not accepted) only if there is no possible sequence of moves by which a final state can be reached. Nondeterminism can therefore be viewed as involving "intuitive" insight by which the best move can be chosen at every state (assuming that the NFA wants to accept every string).

Like NFA's, nondeterministic accepters can be represented by transition graphs. The vertices are determined by Q, while an edge (q_i, q_j) with label a is in the graph if and only if $\delta(q_i, a)$ contains q_j. Note that since a may be the empty string, there can be some edges labeled λ.

A string is accepted by an NFA if there is some sequence of possible moves that will put the machine in a final state at the end of the string. A string is rejected (that is, not accepted) only if there is no possible sequence of moves by which a final state can be reached. Nondeterminism can therefore be viewed as involving "intuitive" insight by which the best move can be chosen at every state (assuming that the NFA wants to accept every string).

Like NFA's, nondeterministic accepters can be represented by transition graphs. The vertices are determined by Q, while an edge (q_i, q_j) with label a is in the graph if and only if $\delta(q_i, a)$ contains q_j. Note that since a may be the empty string, there can be some edges labeled λ.

A string is accepted by an NFA if there is some sequence of possible moves that will put the machine in a final state at the end of the string. A string is rejected (that is, not accepted) only if there is no possible sequence of moves by which a final state can be reached. Nondeterminism can therefore be viewed as involving "intuitive" insight by which the best move can be chosen at every state (assuming that the NFA wants to accept every string).

Like NFA's, nondeterministic accepters can be represented by transition graphs. The vertices are determined by Q, while an edge (q_i, q_j) with label a is in the graph if and only if $\delta(q_i, a)$ contains q_j. Note that since a may be the empty string, there can be some edges labeled λ.

A string is accepted by an NFA if there is some sequence of possible moves that will put the machine in a final state at the end of the string. A string is rejected (that is, not accepted) only if there is no possible sequence of moves by which a final state can be reached. Nondeterminism can therefore be viewed as involving "intuitive" insight by which the best move can be chosen at every state (assuming that the NFA wants to accept every string).

Like NFA's, nondeterministic accepters can be represented by transition graphs. The vertices are determined by Q, while an edge (q_i, q_j) with label a is in the graph if and only if $\delta(q_i, a)$ contains q_j. Note that since a may be the empty string, there can be some edges labeled λ.

A string is accepted by an NFA if there is some sequence of possible moves that will put the machine in a final state at the end of the string. A string is rejected (that is, not accepted) only if there is no possible sequence of moves by which a final state can be reached. Nondeterminism can therefore be viewed as involving "intuitive" insight by which the best move can be chosen at every state (assuming that the NFA wants to accept every string).

Like NFA's, nondeterministic accepters can be represented by transition graphs. The vertices are determined by Q, while an edge (q_i, q_j) with label a is in the graph if and only if $\delta(q_i, a)$ contains q_j. Note that since a may be the empty string, there can be some edges labeled λ.

A string is accepted by an NFA if there is some sequence of possible moves that will put the machine in a final state at the end of the string. A string is rejected (that is, not accepted) only if there is no possible sequence of moves by which a final state can be reached. Nondeterminism can therefore be viewed as involving "intuitive" insight by which the best move can be chosen at every state (assuming that the NFA wants to accept every string).

Like NFA's, nondeterministic accepters can be represented by transition graphs. The vertices are determined by Q, while an edge (q_i, q_j) with label a is in the graph if and only if $\delta(q_i, a)$ contains q_j. Note that since a may be the empty string, there can be some edges labeled λ.

A string is accepted by an NFA if there is some sequence of possible moves that will put the machine in a final state at the end of the string. A string is rejected (that is, not accepted) only if there is no possible sequence of moves by which a final state can be reached. Nondeterminism can therefore be viewed as involving "intuitive" insight by which the best move can be chosen at every state (assuming that the NFA wants to accept every string).

Like NFA's, nondeterministic accepters can be represented by transition graphs. The vertices are determined by Q, while an edge (q_i, q_j) with label a is in the graph if and only if $\delta(q_i, a)$ contains q_j. Note that since a may be the empty string, there can be some edges labeled λ.

A string is accepted by an NFA if there is some sequence of possible moves that will put the machine in a final state at the end of the string. A string is rejected (that is, not accepted) only if there is no possible sequence of moves by which a final state can be reached. Nondeterminism can therefore be viewed as involving "intuitive" insight by which the best move can be chosen at every state (assuming that the NFA wants to accept every string).

Like NFA's, nondeterministic accepters can be represented by transition graphs. The vertices are determined by Q, while an edge (q_i, q_j) with label a is in the graph if and only if $\delta(q_i, a)$ contains q_j. Note that since a may be the empty string, there can be some edges labeled λ.

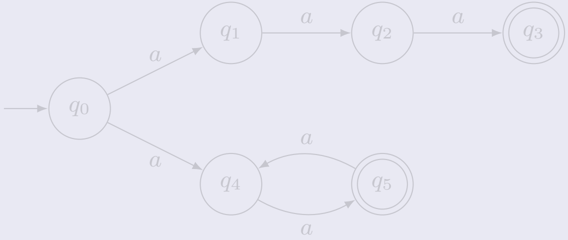
A string is accepted by an NFA if there is some sequence of possible moves that will put the machine in a final state at the end of the string. A string is rejected (that is, not accepted) only if there is no possible sequence of moves by which a final state can be reached. Nondeterminism can therefore be viewed as involving "intuitive" insight by which the best move can be chosen at every state (assuming that the NFA wants to accept every string).

Example 2.7

Consider the transition graph in the Figure. It describes a nondeterministic accepter since there are two transitions labeled a out of q_0.

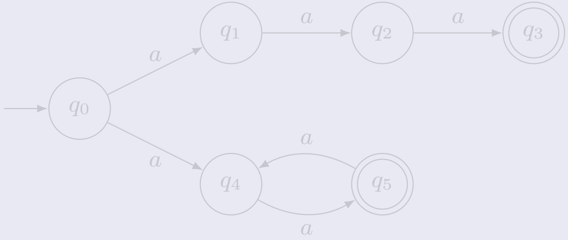
Example 2.7

Consider the transition graph in the Figure. It describes a nondeterministic accepter since there are two transitions labeled a out of q_0.



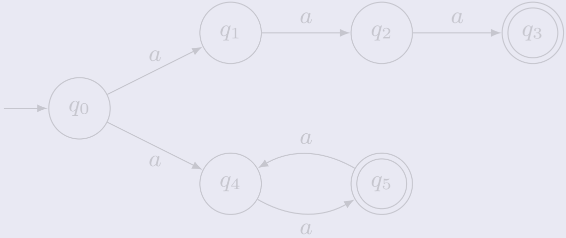
Example 2.7

Consider the transition graph in the Figure. It describes a nondeterministic accepter since there are two transitions labeled a out of q_0.



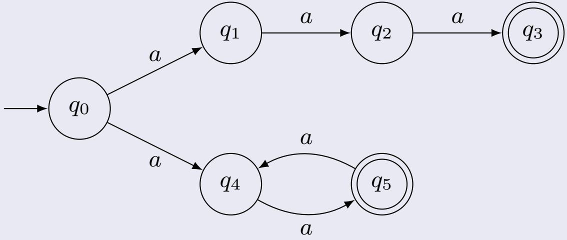
Example 2.7

Consider the transition graph in the Figure. It describes a nondeterministic accepter since there are two transitions labeled a out of q_0.



Example 2.7

Consider the transition graph in the Figure. It describes a nondeterministic accepter since there are two transitions labeled a out of q_0.

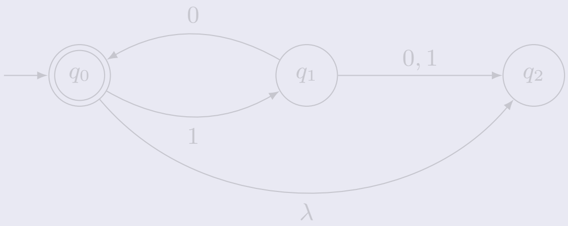


Example 2.8

A nondeterministic automaton is shown in the Figure. It is nondeterministic not only because several edges with the same label originate from one vertex, but also because it has a λ-transition. Some transitions, such as $\delta(q_2, 0)$, are unspecified in the graph. This is to be interpreted as a transition to the empty set, that is, $\delta(q_2, 0) = \varnothing$. The automaton accepts strings λ, 1010, and 101010, but not 110 and 10100. Note that for 10 there are two alternative walks, one leading to q_0, the other to q_2. Even though q_2 is not a final state, the string is accepted because one walk leads to a final state.

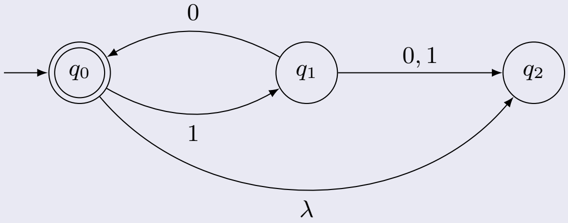
Example 2.8

A nondeterministic automaton is shown in the Figure. It is nondeterministic not only because several edges with the same label originate from one vertex, but also because it has a λ-transition. Some transitions, such as $\delta(q_2, 0)$, are unspecified in the graph. This is to be interpreted as a transition to the empty set, that is, $\delta(q_2, 0) = \varnothing$. The automaton accepts strings λ, 1010, and 101010, but not 110 and 10100. Note that for 10 there are two alternative walks, one leading to q_0, the other to q_2. Even though q_2 is not a final state, the string is accepted because one walk leads to a final state.



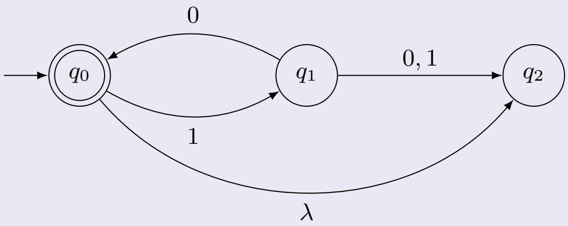
Example 2.8

A nondeterministic automaton is shown in the Figure. It is nondeterministic not only because several edges with the same label originate from one vertex, but also because it has a λ-transition. Some transitions, such as $\delta(q_2, 0)$, are unspecified in the graph. This is to be interpreted as a transition to the empty set, that is, $\delta(q_2, 0) = \varnothing$. The automaton accepts strings λ, 1010, and 101010, but not 110 and 10100. Note that for 10 there are two alternative walks, one leading to q_0, the other to q_2. Even though q_2 is not a final state, the string is accepted because one walk leads to a final state.



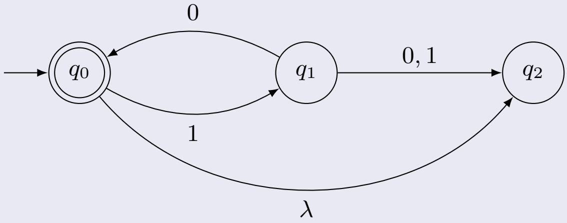
Example 2.8

A nondeterministic automaton is shown in the Figure. It is nondeterministic not only because several edges with the same label originate from one vertex, but also because it has a λ-transition. Some transitions, such as $\delta(q_2, 0)$, are unspecified in the graph. This is to be interpreted as a transition to the empty set, that is, $\delta(q_2, 0) = \varnothing$. The automaton accepts strings λ, 1010, and 101010, but not 110 and 10100. Note that for 10 there are two alternative walks, one leading to q_0, the other to q_2. Even though q_2 is not a final state, the string is accepted because one walk leads to a final state.



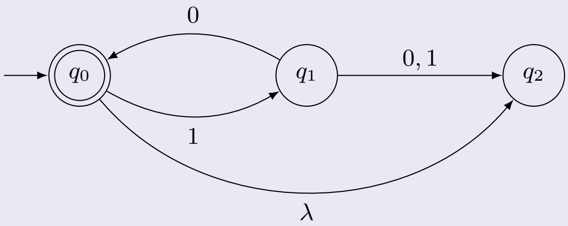
Example 2.8

A nondeterministic automaton is shown in the Figure. It is nondeterministic not only because several edges with the same label originate from one vertex, but also because it has a λ-transition. Some transitions, such as $\delta(q_2, 0)$, are unspecified in the graph. This is to be interpreted as a transition to the empty set, that is, $\delta(q_2, 0) = \varnothing$. The automaton accepts strings λ, 1010, and 101010, but not 110 and 10100. Note that for 10 there are two alternative walks, one leading to q_0, the other to q_2. Even though q_2 is not a final state, the string is accepted because one walk leads to a final state.



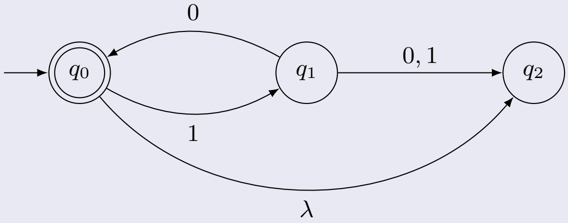
Example 2.8

A nondeterministic automaton is shown in the Figure. It is nondeterministic not only because several edges with the same label originate from one vertex, but also because it has a λ-transition. Some transitions, such as $\delta(q_2, 0)$, are unspecified in the graph. This is to be interpreted as a transition to the empty set, that is, $\delta(q_2, 0) = \varnothing$. The automaton accepts strings λ, 1010, and 101010, but not 110 and 10100. Note that for 10 there are two alternative walks, one leading to q_0, the other to q_2. Even though q_2 is not a final state, the string is accepted because one walk leads to a final state.



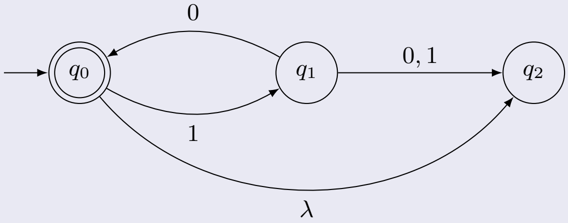
Example 2.8

A nondeterministic automaton is shown in the Figure. It is nondeterministic not only because several edges with the same label originate from one vertex, but also because it has a λ-transition. Some transitions, such as $\delta(q_2, 0)$, are unspecified in the graph. This is to be interpreted as a transition to the empty set, that is, $\delta(q_2, 0) = \varnothing$. The automaton accepts strings λ, 1010, and 101010, but not 110 and 10100. Note that for 10 there are two alternative walks, one leading to q_0, the other to q_2. Even though q_2 is not a final state, the string is accepted because one walk leads to a final state.



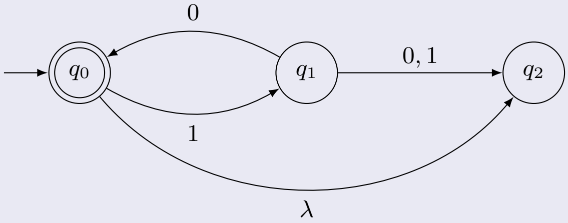
Example 2.8

A nondeterministic automaton is shown in the Figure. It is nondeterministic not only because several edges with the same label originate from one vertex, but also because it has a λ-transition. Some transitions, such as $\delta(q_2, 0)$, are unspecified in the graph. This is to be interpreted as a transition to the empty set, that is, $\delta(q_2, 0) = \varnothing$. The automaton accepts strings λ, 1010, and 101010, but not 110 and 10100. Note that for 10 there are two alternative walks, one leading to q_0, the other to q_2. Even though q_2 is not a final state, the string is accepted because one walk leads to a final state.



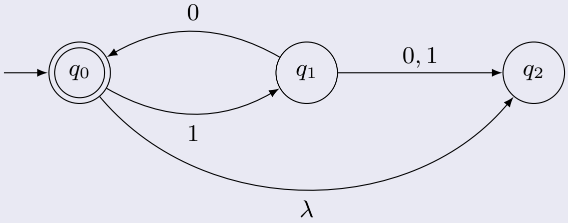
Example 2.8

A nondeterministic automaton is shown in the Figure. It is nondeterministic not only because several edges with the same label originate from one vertex, but also because it has a λ-transition. Some transitions, such as $\delta(q_2, 0)$, are unspecified in the graph. This is to be interpreted as a transition to the empty set, that is, $\delta(q_2, 0) = \varnothing$. The automaton accepts strings λ, 1010, and 101010, but not 110 and 10100. Note that for 10 there are two alternative walks, one leading to q_0, the other to q_2. Even though q_2 is not a final state, the string is accepted because one walk leads to a final state.



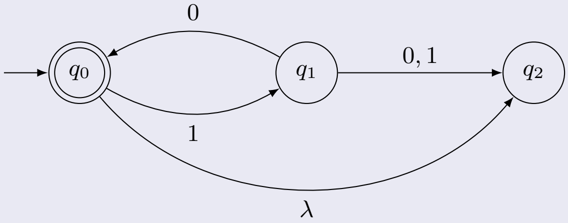
Example 2.8

A nondeterministic automaton is shown in the Figure. It is nondeterministic not only because several edges with the same label originate from one vertex, but also because it has a λ-transition. Some transitions, such as $\delta(q_2, 0)$, are unspecified in the graph. This is to be interpreted as a transition to the empty set, that is, $\delta(q_2, 0) = \varnothing$. The automaton accepts strings λ, 1010, and 101010, but not 110 and 10100. Note that for 10 there are two alternative walks, one leading to q_0, the other to q_2. Even though q_2 is not a final state, the string is accepted because one walk leads to a final state.



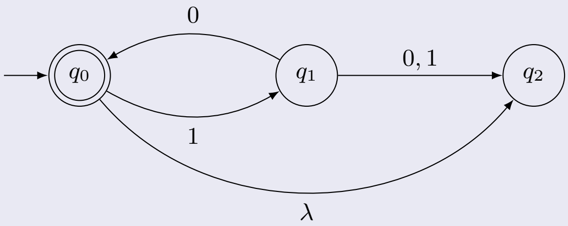
Example 2.8

A nondeterministic automaton is shown in the Figure. It is nondeterministic not only because several edges with the same label originate from one vertex, but also because it has a λ-transition. Some transitions, such as $\delta(q_2, 0)$, are unspecified in the graph. This is to be interpreted as a transition to the empty set, that is, $\delta(q_2, 0) = \varnothing$. The automaton accepts strings λ, 1010, and 101010, but not 110 and 10100. Note that for 10 there are two alternative walks, one leading to q_0, the other to q_2. Even though q_2 is not a final state, the string is accepted because one walk leads to a final state.



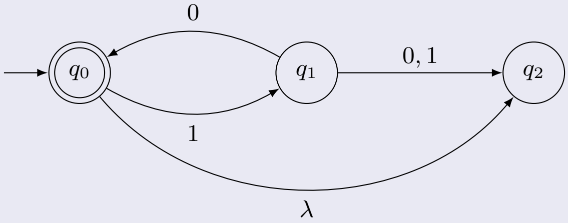
Example 2.8

A nondeterministic automaton is shown in the Figure. It is nondeterministic not only because several edges with the same label originate from one vertex, but also because it has a λ-transition. Some transitions, such as $\delta(q_2, 0)$, are unspecified in the graph. This is to be interpreted as a transition to the empty set, that is, $\delta(q_2, 0) = \varnothing$. The automaton accepts strings λ, 1010, and 101010, but not 110 and 10100. Note that for 10 there are two alternative walks, one leading to q_0, the other to q_2. Even though q_2 is not a final state, the string is accepted because one walk leads to a final state.



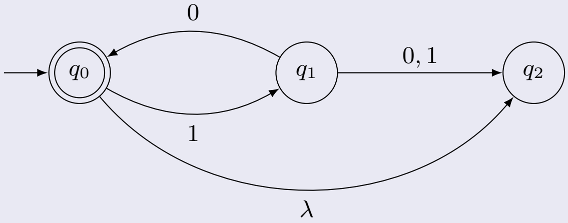
Example 2.8

A nondeterministic automaton is shown in the Figure. It is nondeterministic not only because several edges with the same label originate from one vertex, but also because it has a λ-transition. Some transitions, such as $\delta(q_2, 0)$, are unspecified in the graph. This is to be interpreted as a transition to the empty set, that is, $\delta(q_2, 0) = \varnothing$. The automaton accepts strings λ, 1010, and 101010, but not 110 and 10100. Note that for 10 there are two alternative walks, one leading to q_0, the other to q_2. Even though q_2 is not a final state, the string is accepted because one walk leads to a final state.



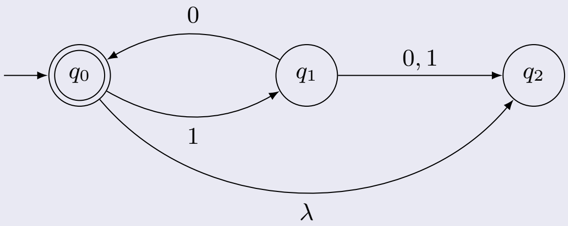
Example 2.8

A nondeterministic automaton is shown in the Figure. It is nondeterministic not only because several edges with the same label originate from one vertex, but also because it has a λ-transition. Some transitions, such as $\delta(q_2, 0)$, are unspecified in the graph. This is to be interpreted as a transition to the empty set, that is, $\delta(q_2, 0) = \varnothing$. The automaton accepts strings λ, 1010, and 101010, but not 110 and 10100. Note that for 10 there are two alternative walks, one leading to q_0, the other to q_2. Even though q_2 is not a final state, the string is accepted because one walk leads to a final state.



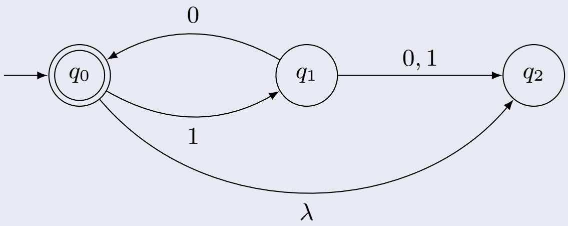
Example 2.8

A nondeterministic automaton is shown in the Figure. It is nondeterministic not only because several edges with the same label originate from one vertex, but also because it has a λ-transition. Some transitions, such as $\delta(q_2, 0)$, are unspecified in the graph. This is to be interpreted as a transition to the empty set, that is, $\delta(q_2, 0) = \varnothing$. The automaton accepts strings λ, 1010, and 101010, but not 110 and 10100. Note that for 10 there are two alternative walks, one leading to q_0, the other to q_2. Even though q_2 is not a final state, the string is accepted because one walk leads to a final state.



Example 2.8

A nondeterministic automaton is shown in the Figure. It is nondeterministic not only because several edges with the same label originate from one vertex, but also because it has a λ-transition. Some transitions, such as $\delta(q_2, 0)$, are unspecified in the graph. This is to be interpreted as a transition to the empty set, that is, $\delta(q_2, 0) = \varnothing$. The automaton accepts strings λ, 1010, and 101010, but not 110 and 10100. Note that for 10 there are two alternative walks, one leading to q_0, the other to q_2. Even though q_2 is not a final state, the string is accepted because one walk leads to a final state.



Again, the transition function can be extended so its second argument is a string. We require of the extended transition function δ^* that if

$$\delta^*(q_i, w) = Q_j,$$

then Q_j is the set of all possible states the automaton may be in, having started in state q_i and having read w. A recursive definition of δ^*, analogous to (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$
$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

is possible, but not particularly enlightening. A more easily appreciated definition can be made through transition graphs.

Definition 2.5

For an NDA, the extended transition function is defined so that $\delta^*(q_i, w)$ contains q_j if and only if there is a walk in the transition graph from q_i to q_j labeled w. This holds for all $q_i, q_j \in Q$, and $w \in \Sigma^*$.

Again, the transition function can be extended so its second argument is a string. We require of the extended transition function δ^* that if

$$\delta^*(q_i, w) = Q_j,$$

then Q_j is the set of all possible states the automaton may be in, having started in state q_i and having read w. A recursive definition of δ^*, analogous to (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

is possible, but not particularly enlightening. A more easily appreciated definition can be made through transition graphs.

Definition 2.5

For an NDA, the extended transition function is defined so that $\delta^*(q_i, w)$ contains q_j if and only if there is a walk in the transition graph from q_i to q_j labeled w. This holds for all $q_i, q_j \in Q$, and $w \in \Sigma^*$.

Again, the transition function can be extended so its second argument is a string. We require of the extended transition function δ^* that if

$$\delta^*(q_i, w) = Q_j,$$

then Q_j is the set of all possible states the automaton may be in, having started in state q_i and having read w. A recursive definition of δ^*, analogous to (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

is possible, but not particularly enlightening. A more easily appreciated definition can be made through transition graphs.

Definition 2.5

For an NDA, the extended transition function is defined so that $\delta^*(q_i, w)$ contains q_j if and only if there is a walk in the transition graph from q_i to q_j labeled w. This holds for all $q_i, q_j \in Q$, and $w \in \Sigma^*$.

Again, the transition function can be extended so its second argument is a string. We require of the extended transition function δ^* that if

$$\delta^*(q_i, w) = Q_j,$$

then Q_j is the set of all possible states the automaton may be in, having started in state q_i and having read w. A recursive definition of δ^*, analogous to (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

is possible, but not particularly enlightening. A more easily appreciated definition can be made through transition graphs.

Definition 2.5

For an NDA, the extended transition function is defined so that $\delta^*(q_i, w)$ contains q_j if and only if there is a walk in the transition graph from q_i to q_j labeled w. This holds for all $q_i, q_j \in Q$, and $w \in \Sigma^*$.

Again, the transition function can be extended so its second argument is a string. We require of the extended transition function δ^* that if

$$\delta^*(q_i, w) = Q_j,$$

then Q_j is the set of all possible states the automaton may be in, having started in state q_i and having read w. A recursive definition of δ^*, analogous to (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$
$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

is possible, but not particularly enlightening. A more easily appreciated definition can be made through transition graphs.

Definition 2.5

For an NDA, the extended transition function is defined so that $\delta^*(q_i, w)$ contains q_j if and only if there is a walk in the transition graph from q_i to q_j labeled w. This holds for all $q_i, q_j \in Q$, and $w \in \Sigma^*$.

Again, the transition function can be extended so its second argument is a string. We require of the extended transition function δ^* that if

$$\delta^*(q_i, w) = Q_j,$$

then Q_j is the set of all possible states the automaton may be in, having started in state q_i and having read w. A recursive definition of δ^*, analogous to (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

is possible, but not particularly enlightening. A more easily appreciated definition can be made through transition graphs.

Definition 2.5

For an NDA, the extended transition function is defined so that $\delta^*(q_i, w)$ contains q_j if and only if there is a walk in the transition graph from q_i to q_j labeled w. This holds for all $q_i, q_j \in Q$, and $w \in \Sigma^*$.

Again, the transition function can be extended so its second argument is a string. We require of the extended transition function δ^* that if

$$\delta^*(q_i, w) = Q_j,$$

then Q_j is the set of all possible states the automaton may be in, having started in state q_i and having read w. A recursive definition of δ^*, analogous to (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

is possible, but not particularly enlightening. A more easily appreciated definition can be made through transition graphs.

Definition 2.5

For an NDA, the extended transition function is defined so that $\delta^*(q_i, w)$ contains q_j if and only if there is a walk in the transition graph from q_i to q_j labeled w. This holds for all $q_i, q_j \in Q$, and $w \in \Sigma^*$.

Again, the transition function can be extended so its second argument is a string. We require of the extended transition function δ^* that if

$$\delta^*(q_i, w) = Q_j,$$

then Q_j is the set of all possible states the automaton may be in, having started in state q_i and having read w. A recursive definition of δ^*, analogous to (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

is possible, but not particularly enlightening. A more easily appreciated definition can be made through transition graphs.

Definition 2.5

For an NDA, the extended transition function is defined so that $\delta^*(q_i, w)$ contains q_j if and only if there is a walk in the transition graph from q_i to q_j labeled w. This holds for all $q_i, q_j \in Q$, and $w \in \Sigma^*$.

Again, the transition function can be extended so its second argument is a string. We require of the extended transition function δ^* that if

$$\delta^*(q_i, w) = Q_j,$$

then Q_j is the set of all possible states the automaton may be in, having started in state q_i and having read w. A recursive definition of δ^*, analogous to (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$
$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

is possible, but not particularly enlightening. A more easily appreciated definition can be made through transition graphs.

Definition 2.5

For an NDA, the extended transition function is defined so that $\delta^*(q_i, w)$ contains q_j if and only if there is a walk in the transition graph from q_i to q_j labeled w. This holds for all $q_i, q_j \in Q$, and $w \in \Sigma^*$.

Again, the transition function can be extended so its second argument is a string. We require of the extended transition function δ^* that if

$$\delta^*(q_i, w) = Q_j,$$

then Q_j is the set of all possible states the automaton may be in, having started in state q_i and having read w. A recursive definition of δ^*, analogous to (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

is possible, but not particularly enlightening. A more easily appreciated definition can be made through transition graphs.

Definition 2.5

For an NDA, the extended transition function is defined so that $\delta^*(q_i, w)$ contains q_j if and only if there is a walk in the transition graph from q_i to q_j labeled w. This holds for all $q_i, q_j \in Q$, and $w \in \Sigma^*$.

Again, the transition function can be extended so its second argument is a string. We require of the extended transition function δ^* that if

$$\delta^*(q_i, w) = Q_j,$$

then Q_j is the set of all possible states the automaton may be in, having started in state q_i and having read w. A recursive definition of δ^*, analogous to (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

is possible, but not particularly enlightening. A more easily appreciated definition can be made through transition graphs.

Definition 2.5

For an NDA, the extended transition function is defined so that $\delta^*(q_i, w)$ contains q_j if and only if there is a walk in the transition graph from q_i to q_j labeled w. This holds for all $q_i, q_j \in Q$, and $w \in \Sigma^*$.

Again, the transition function can be extended so its second argument is a string. We require of the extended transition function δ^* that if

$$\delta^*(q_i, w) = Q_j,$$

then Q_j is the set of all possible states the automaton may be in, having started in state q_i and having read w. A recursive definition of δ^*, analogous to (1) and (2),

$$\delta^*(q, \lambda) = q, \tag{1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2}$$

is possible, but not particularly enlightening. A more easily appreciated definition can be made through transition graphs.

Definition 2.5

For an NDA, the extended transition function is defined so that $\delta^*(q_i, w)$ contains q_j if and only if there is a walk in the transition graph from q_i to q_j labeled w. This holds for all $q_i, q_j \in Q$, and $w \in \Sigma^*$.

Example 2.9

The following Figure represents an NFA.

It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

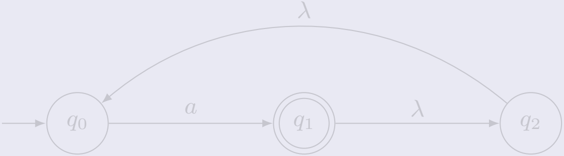
$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

Example 2.9

The following Figure represents an NFA.



It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

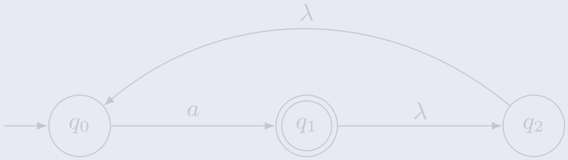
$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

Example 2.9

The following Figure represents an NFA.



It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

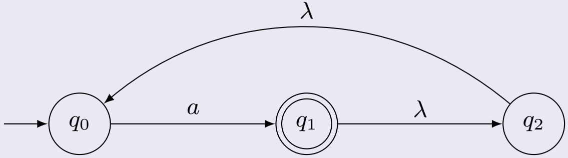
$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

Example 2.9

The following Figure represents an NFA.



It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

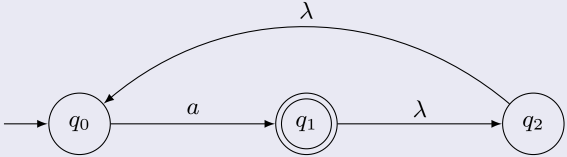
$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

Example 2.9

The following Figure represents an NFA.



It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

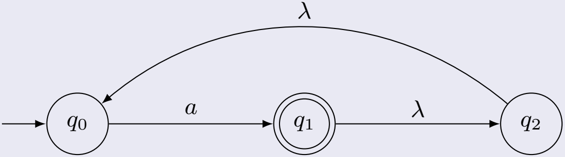
$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

Example 2.9

The following Figure represents an NFA.



It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

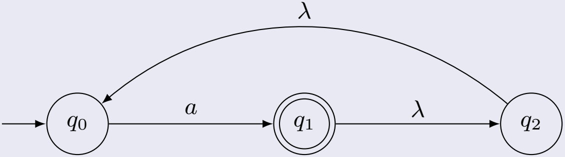
$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

Example 2.9

The following Figure represents an NFA.



It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

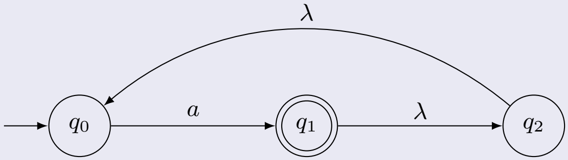
$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

Example 2.9

The following Figure represents an NFA.



It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

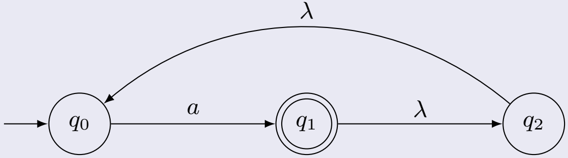
$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

Example 2.9

The following Figure represents an NFA.



It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

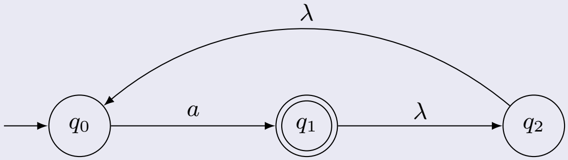
$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

Example 2.9

The following Figure represents an NFA.



It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

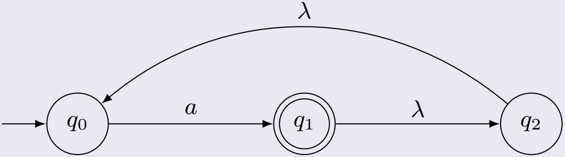
$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

Example 2.9

The following Figure represents an NFA.



It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

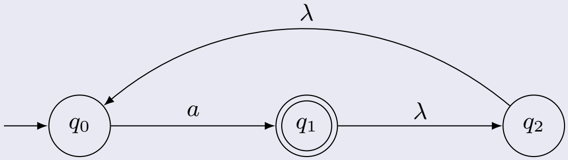
$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

Example 2.9

The following Figure represents an NFA.



It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

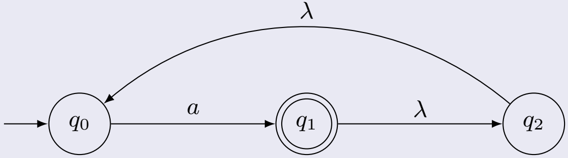
$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

Example 2.9

The following Figure represents an NFA.



It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

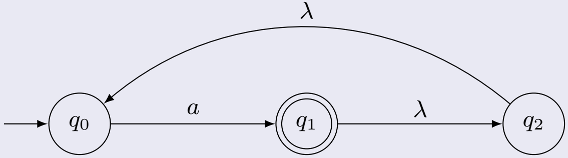
$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

Example 2.9

The following Figure represents an NFA.



It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

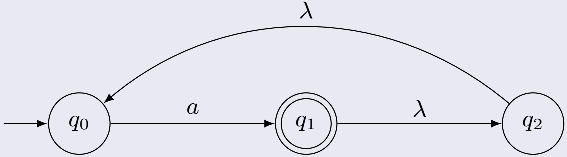
$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

Example 2.9

The following Figure represents an NFA.



It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

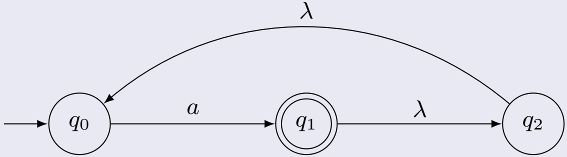
$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

Example 2.9

The following Figure represents an NFA.



It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\} .$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

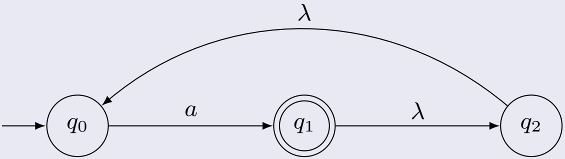
$$\delta^*(q_2, \lambda) = \{q_0, q_2\} .$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\} .$$

Example 2.9

The following Figure represents an NFA.



It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

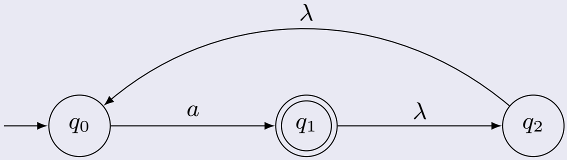
$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

Example 2.9

The following Figure represents an NFA.



It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

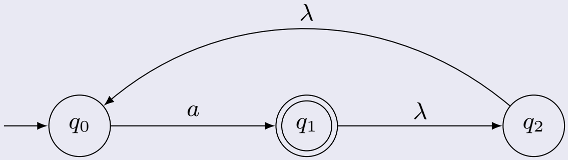
$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

Example 2.9

The following Figure represents an NFA.



It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2. Therefore,

$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, because between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In the first lecture, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly because, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

With this observation, we have a method for computing $\delta^*(q_i, w)$. We evaluate all walks of length at most $\Lambda + (1+\Lambda)|w|$ originating at q_i. We select from them those that are labeled w. The terminating vertices of the selected walks are the elements of the set $\delta^*(q_i, w)$.

As we have remarked, it is possible to define δ^* in a recursive fashion as was done for the deterministic case. The result is unfortunately not very transparent, and arguments with the extended transition function defined this way are hard to follow. We prefer to use the more intuitive and more manageable alternative in Definition 2.5.

As for DFA's, the language accepted by an NFA is defined formally by the extended transition function.

Definition 2.6

The language L accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as the set of all strings accepted in the above sense. Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing\}.$$

In words, the language consists of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.

With this observation, we have a method for computing $\delta^*(q_i, w)$. We evaluate all walks of length at most $\Lambda + (1 + \Lambda)|w|$ originating at q_i. We select from them those that are labeled w. The terminating vertices of the selected walks are the elements of the set $\delta^*(q_i, w)$.

As we have remarked, it is possible to define δ^* in a recursive fashion as was done for the deterministic case. The result is unfortunately not very transparent, and arguments with the extended transition function defined this way are hard to follow. We prefer to use the more intuitive and more manageable alternative in Definition 2.5.

As for DFA's, the language accepted by an NFA is defined formally by the extended transition function.

Definition 2.6

The language L accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as the set of all strings accepted in the above sense. Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing\}.$$

In words, the language consists of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.

With this observation, we have a method for computing $\delta^*(q_i, w)$. We evaluate all walks of length at most $\Lambda + (1+\Lambda)|w|$ originating at q_i. We select from them those that are labeled w. The terminating vertices of the selected walks are the elements of the set $\delta^*(q_i, w)$.

As we have remarked, it is possible to define δ^* in a recursive fashion as was done for the deterministic case. The result is unfortunately not very transparent, and arguments with the extended transition function defined this way are hard to follow. We prefer to use the more intuitive and more manageable alternative in Definition 2.5.

As for DFA's, the language accepted by an NFA is defined formally by the extended transition function.

Definition 2.6

The language L accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as the set of all strings accepted in the above sense. Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing\}.$$

In words, the language consists of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.

With this observation, we have a method for computing $\delta^*(q_i, w)$. We evaluate all walks of length at most $\Lambda + (1 + \Lambda)|w|$ originating at q_i. We select from them those that are labeled w. The terminating vertices of the selected walks are the elements of the set $\delta^*(q_i, w)$.

As we have remarked, it is possible to define δ^* in a recursive fashion as was done for the deterministic case. The result is unfortunately not very transparent, and arguments with the extended transition function defined this way are hard to follow. We prefer to use the more intuitive and more manageable alternative in Definition 2.5.

As for DFA's, the language accepted by an NFA is defined formally by the extended transition function.

Definition 2.6

The language L accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as the set of all strings accepted in the above sense. Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing\}.$$

In words, the language consists of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.

With this observation, we have a method for computing $\delta^*(q_i, w)$. We evaluate all walks of length at most $\Lambda + (1+\Lambda)|w|$ originating at q_i. We select from them those that are labeled w. The terminating vertices of the selected walks are the elements of the set $\delta^*(q_i, w)$.

As we have remarked, it is possible to define δ^* in a recursive fashion as was done for the deterministic case. The result is unfortunately not very transparent, and arguments with the extended transition function defined this way are hard to follow. We prefer to use the more intuitive and more manageable alternative in Definition 2.5.

As for DFA's, the language accepted by an NFA is defined formally by the extended transition function.

Definition 2.6

The language L accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as the set of all strings accepted in the above sense. Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing\}.$$

In words, the language consists of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.

With this observation, we have a method for computing $\delta^*(q_i, w)$. We evaluate all walks of length at most $\Lambda + (1+\Lambda)|w|$ originating at q_i. We select from them those that are labeled w. The terminating vertices of the selected walks are the elements of the set $\delta^*(q_i, w)$.

As we have remarked, it is possible to define δ^* in a recursive fashion as was done for the deterministic case. The result is unfortunately not very transparent, and arguments with the extended transition function defined this way are hard to follow. We prefer to use the more intuitive and more manageable alternative in Definition 2.5.

As for DFA's, the language accepted by an NFA is defined formally by the extended transition function.

Definition 2.6

The language L accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as the set of all strings accepted in the above sense. Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing\}.$$

In words, the language consists of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.

With this observation, we have a method for computing $\delta^*(q_i, w)$. We evaluate all walks of length at most $\Lambda + (1 + \Lambda)|w|$ originating at q_i. We select from them those that are labeled w. The terminating vertices of the selected walks are the elements of the set $\delta^*(q_i, w)$.

As we have remarked, it is possible to define δ^* in a recursive fashion as was done for the deterministic case. The result is unfortunately not very transparent, and arguments with the extended transition function defined this way are hard to follow. We prefer to use the more intuitive and more manageable alternative in Definition 2.5.

As for DFA's, the language accepted by an NFA is defined formally by the extended transition function.

Definition 2.6

The language L accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as the set of all strings accepted in the above sense. Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing\}.$$

In words, the language consists of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.

With this observation, we have a method for computing $\delta^*(q_i, w)$. We evaluate all walks of length at most $\Lambda + (1+\Lambda)|w|$ originating at q_i. We select from them those that are labeled w. The terminating vertices of the selected walks are the elements of the set $\delta^*(q_i, w)$.

As we have remarked, it is possible to define δ^* in a recursive fashion as was done for the deterministic case. The result is unfortunately not very transparent, and arguments with the extended transition function defined this way are hard to follow. We prefer to use the more intuitive and more manageable alternative in Definition 2.5.

As for DFA's, the language accepted by an NFA is defined formally by the extended transition function.

Definition 2.6

The language L accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as the set of all strings accepted in the above sense. Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing\}.$$

In words, the language consists of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.

With this observation, we have a method for computing $\delta^*(q_i, w)$. We evaluate all walks of length at most $\Lambda + (1 + \Lambda)|w|$ originating at q_i. We select from them those that are labeled w. The terminating vertices of the selected walks are the elements of the set $\delta^*(q_i, w)$.

As we have remarked, it is possible to define δ^* in a recursive fashion as was done for the deterministic case. The result is unfortunately not very transparent, and arguments with the extended transition function defined this way are hard to follow. We prefer to use the more intuitive and more manageable alternative in Definition 2.5.

As for DFA's, the language accepted by an NFA is defined formally by the extended transition function.

Definition 2.6

The language L accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as the set of all strings accepted in the above sense. Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing\}.$$

In words, the language consists of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.

With this observation, we have a method for computing $\delta^*(q_i, w)$. We evaluate all walks of length at most $\Lambda + (1+\Lambda)|w|$ originating at q_i. We select from them those that are labeled w. The terminating vertices of the selected walks are the elements of the set $\delta^*(q_i, w)$.

As we have remarked, it is possible to define δ^* in a recursive fashion as was done for the deterministic case. The result is unfortunately not very transparent, and arguments with the extended transition function defined this way are hard to follow. We prefer to use the more intuitive and more manageable alternative in Definition 2.5.

As for DFA's, the language accepted by an NFA is defined formally by the extended transition function.

Definition 2.6

The language L accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as the set of all strings accepted in the above sense. Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing\}.$$

In words, the language consists of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.

With this observation, we have a method for computing $\delta^*(q_i, w)$. We evaluate all walks of length at most $\Lambda + (1+\Lambda)|w|$ originating at q_i. We select from them those that are labeled w. The terminating vertices of the selected walks are the elements of the set $\delta^*(q_i, w)$.

As we have remarked, it is possible to define δ^* in a recursive fashion as was done for the deterministic case. The result is unfortunately not very transparent, and arguments with the extended transition function defined this way are hard to follow. We prefer to use the more intuitive and more manageable alternative in Definition 2.5.

As for DFA's, the language accepted by an NFA is defined formally by the extended transition function.

Definition 2.6

The language L accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as the set of all strings accepted in the above sense. Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing\}.$$

In words, the language consists of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.

With this observation, we have a method for computing $\delta^*(q_i, w)$. We evaluate all walks of length at most $\Lambda + (1 + \Lambda)|w|$ originating at q_i. We select from them those that are labeled w. The terminating vertices of the selected walks are the elements of the set $\delta^*(q_i, w)$.

As we have remarked, it is possible to define δ^* in a recursive fashion as was done for the deterministic case. The result is unfortunately not very transparent, and arguments with the extended transition function defined this way are hard to follow. We prefer to use the more intuitive and more manageable alternative in Definition 2.5.

As for DFA's, the language accepted by an NFA is defined formally by the extended transition function.

Definition 2.6

The language L accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as the set of all strings accepted in the above sense. Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing\}.$$

In words, the language consists of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.

With this observation, we have a method for computing $\delta^*(q_i, w)$. We evaluate all walks of length at most $\Lambda + (1 + \Lambda)|w|$ originating at q_i. We select from them those that are labeled w. The terminating vertices of the selected walks are the elements of the set $\delta^*(q_i, w)$.

As we have remarked, it is possible to define δ^* in a recursive fashion as was done for the deterministic case. The result is unfortunately not very transparent, and arguments with the extended transition function defined this way are hard to follow. We prefer to use the more intuitive and more manageable alternative in Definition 2.5.

As for DFA's, the language accepted by an NFA is defined formally by the extended transition function.

Definition 2.6

The language L accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as the set of all strings accepted in the above sense. Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing\}.$$

In words, the language consists of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.

With this observation, we have a method for computing $\delta^*(q_i, w)$. We evaluate all walks of length at most $\Lambda + (1 + \Lambda)|w|$ originating at q_i. We select from them those that are labeled w. The terminating vertices of the selected walks are the elements of the set $\delta^*(q_i, w)$.

As we have remarked, it is possible to define δ^* in a recursive fashion as was done for the deterministic case. The result is unfortunately not very transparent, and arguments with the extended transition function defined this way are hard to follow. We prefer to use the more intuitive and more manageable alternative in Definition 2.5.

As for DFA's, the language accepted by an NFA is defined formally by the extended transition function.

Definition 2.6

The language L accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is defined as the set of all strings accepted in the above sense. Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing\}.$$

In words, the language consists of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.

Example 2.10

What is the language accepted by the automaton in the Figure?

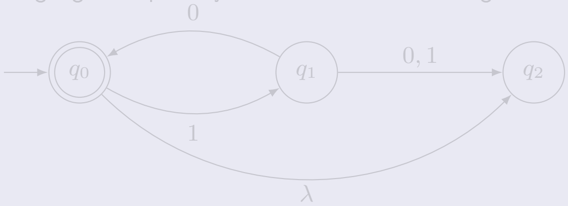
It is easy to see from the graph that the only way the NFA can stop in a final state is if the input is either a repetition of the string 10 or the empty string. Therefore, the automaton accepts the language $L = \{(10)^n : n \geqslant 0\}$.
What happens when this automaton is presented with the string $w = 110$? After reading the prefix 11, the automaton finds itself in state q_2, with the transition $\delta(q_2, 0)$ undefined. We call such a situation a *dead configuration*, and we can visualize it as the automaton simply stopping without further action. But we must always keep in mind that such visualizations are imprecise and carry with them some danger of misinterpretation. What we can say precisely is that

$$\delta^*(q_0, 110) = \varnothing.$$

Thus, no final state can be reached by processing $w = 110$, and hence the string is not accepted.

Example 2.10

What is the language accepted by the automaton in the Figure?



It is easy to see from the graph that the only way the NFA can stop in a final state is if the input is either a repetition of the string 10 or the empty string. Therefore, the automaton accepts the language $L = \{(10)^n : n \geqslant 0\}$.

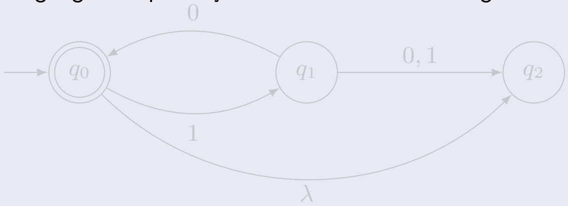
What happens when this automaton is presented with the string $w = 110$? After reading the prefix 11, the automaton finds itself in state q_2, with the transition $\delta(q_2, 0)$ undefined. We call such a situation a *dead configuration*, and we can visualize it as the automaton simply stopping without further action. But we must always keep in mind that such visualizations are imprecise and carry with them some danger of misinterpretation. What we can say precisely is that

$$\delta^*(q_0, 110) = \varnothing.$$

Thus, no final state can be reached by processing $w = 110$, and hence the string is not accepted.

Example 2.10

What is the language accepted by the automaton in the Figure?



It is easy to see from the graph that the only way the NFA can stop in a final state is if the input is either a repetition of the string 10 or the empty string. Therefore, the automaton accepts the language $L = \{(10)^n : n \geqslant 0\}$.

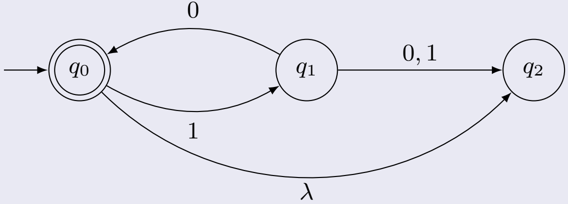
What happens when this automaton is presented with the string $w = 110$? After reading the prefix 11, the automaton finds itself in state q_2, with the transition $\delta(q_2, 0)$ undefined. We call such a situation a *dead configuration*, and we can visualize it as the automaton simply stopping without further action. But we must always keep in mind that such visualizations are imprecise and carry with them some danger of misinterpretation. What we can say precisely is that

$$\delta^*(q_0, 110) = \varnothing.$$

Thus, no final state can be reached by processing $w = 110$, and hence the string is not accepted.

Example 2.10

What is the language accepted by the automaton in the Figure?



It is easy to see from the graph that the only way the NFA can stop in a final state is if the input is either a repetition of the string 10 or the empty string. Therefore, the automaton accepts the language $L = \{(10)^n : n \geqslant 0\}$.

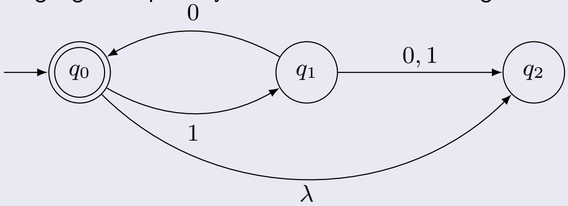
What happens when this automaton is presented with the string $w = 110$? After reading the prefix 11, the automaton finds itself in state q_2, with the transition $\delta(q_2, 0)$ undefined. We call such a situation a *dead configuration*, and we can visualize it as the automaton simply stopping without further action. But we must always keep in mind that such visualizations are imprecise and carry with them some danger of misinterpretation. What we can say precisely is that

$$\delta^*(q_0, 110) = \varnothing.$$

Thus, no final state can be reached by processing $w = 110$, and hence the string is not accepted.

Example 2.10

What is the language accepted by the automaton in the Figure?



It is easy to see from the graph that the only way the NFA can stop in a final state is if the input is either a repetition of the string 10 or the empty string. Therefore, the automaton accepts the language $L = \{(10)^n : n \geqslant 0\}$.

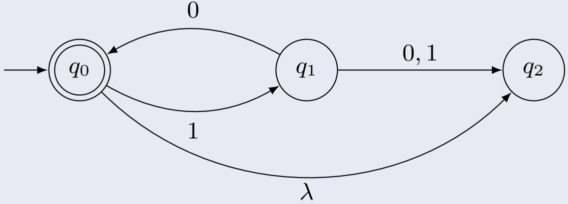
What happens when this automaton is presented with the string $w = 110$? After reading the prefix 11, the automaton finds itself in state q_2, with the transition $\delta(q_2, 0)$ undefined. We call such a situation a *dead configuration*, and we can visualize it as the automaton simply stopping without further action. But we must always keep in mind that such visualizations are imprecise and carry with them some danger of misinterpretation. What we can say precisely is that

$$\delta^*(q_0, 110) = \varnothing.$$

Thus, no final state can be reached by processing $w = 110$, and hence the string is not accepted.

Example 2.10

What is the language accepted by the automaton in the Figure?



It is easy to see from the graph that the only way the NFA can stop in a final state is if the input is either a repetition of the string 10 or the empty string. Therefore, the automaton accepts the language $L = \{(10)^n : n \geqslant 0\}$.

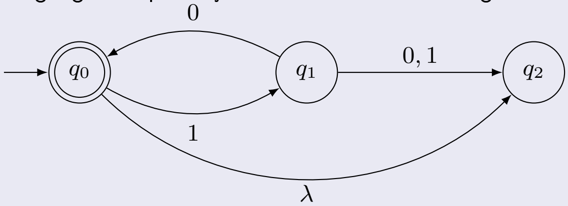
What happens when this automaton is presented with the string $w = 110$? After reading the prefix 11, the automaton finds itself in state q_2, with the transition $\delta(q_2, 0)$ undefined. We call such a situation a *dead configuration*, and we can visualize it as the automaton simply stopping without further action. But we must always keep in mind that such visualizations are imprecise and carry with them some danger of misinterpretation. What we can say precisely is that

$$\delta^*(q_0, 110) = \varnothing.$$

Thus, no final state can be reached by processing $w = 110$, and hence the string is not accepted.

Example 2.10

What is the language accepted by the automaton in the Figure?



It is easy to see from the graph that the only way the NFA can stop in a final state is if the input is either a repetition of the string 10 or the empty string. Therefore, the automaton accepts the language $L = \{(10)^n : n \geqslant 0\}$.

What happens when this automaton is presented with the string $w = 110$?

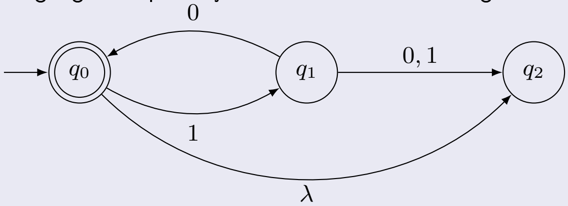
After reading the prefix 11, the automaton finds itself in state q_2, with the transition $\delta(q_2, 0)$ undefined. We call such a situation a *dead configuration*, and we can visualize it as the automaton simply stopping without further action. But we must always keep in mind that such visualizations are imprecise and carry with them some danger of misinterpretation. What we can say precisely is that

$$\delta^*(q_0, 110) = \varnothing.$$

Thus, no final state can be reached by processing $w = 110$, and hence the string is not accepted.

Example 2.10

What is the language accepted by the automaton in the Figure?



It is easy to see from the graph that the only way the NFA can stop in a final state is if the input is either a repetition of the string 10 or the empty string. Therefore, the automaton accepts the language $L = \{(10)^n : n \geqslant 0\}$.

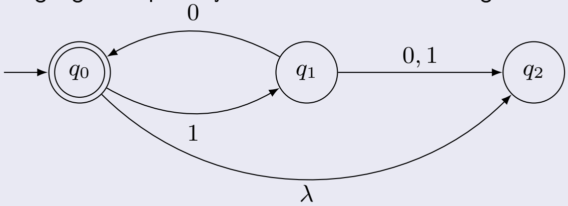
What happens when this automaton is presented with the string $w = 110$? After reading the prefix 11, the automaton finds itself in state q_2, with the transition $\delta(q_2, 0)$ undefined. We call such a situation a *dead configuration*, and we can visualize it as the automaton simply stopping without further action. But we must always keep in mind that such visualizations are imprecise and carry with them some danger of misinterpretation. What we can say precisely is that

$$\delta^*(q_0, 110) = \varnothing.$$

Thus, no final state can be reached by processing $w = 110$, and hence the string is not accepted.

Example 2.10

What is the language accepted by the automaton in the Figure?



It is easy to see from the graph that the only way the NFA can stop in a final state is if the input is either a repetition of the string 10 or the empty string. Therefore, the automaton accepts the language $L = \{(10)^n : n \geqslant 0\}$.

What happens when this automaton is presented with the string $w = 110$?

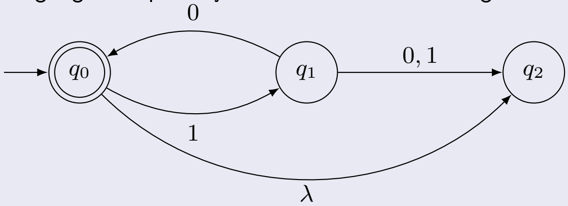
After reading the prefix 11, the automaton finds itself in state q_2, with the transition $\delta(q_2, 0)$ undefined. We call such a situation a *dead configuration*, and we can visualize it as the automaton simply stopping without further action. But we must always keep in mind that such visualizations are imprecise and carry with them some danger of misinterpretation. What we can say precisely is that

$$\delta^*(q_0, 110) = \varnothing.$$

Thus, no final state can be reached by processing $w = 110$, and hence the string is not accepted.

Example 2.10

What is the language accepted by the automaton in the Figure?



It is easy to see from the graph that the only way the NFA can stop in a final state is if the input is either a repetition of the string 10 or the empty string. Therefore, the automaton accepts the language $L = \{(10)^n : n \geqslant 0\}$.

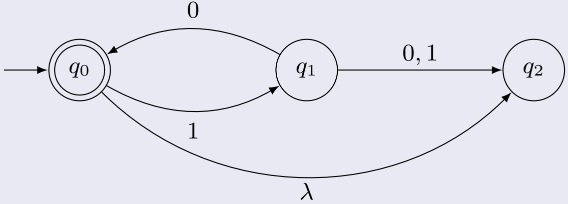
What happens when this automaton is presented with the string $w = 110$? After reading the prefix 11, the automaton finds itself in state q_2, with the transition $\delta(q_2, 0)$ undefined. We call such a situation a *dead configuration*, and we can visualize it as the automaton simply stopping without further action. But we must always keep in mind that such visualizations are imprecise and carry with them some danger of misinterpretation. What we can say precisely is that

$$\delta^*(q_0, 110) = \varnothing.$$

Thus, no final state can be reached by processing $w = 110$, and hence the string is not accepted.

Example 2.10

What is the language accepted by the automaton in the Figure?



It is easy to see from the graph that the only way the NFA can stop in a final state is if the input is either a repetition of the string 10 or the empty string. Therefore, the automaton accepts the language $L = \{(10)^n : n \geqslant 0\}$.

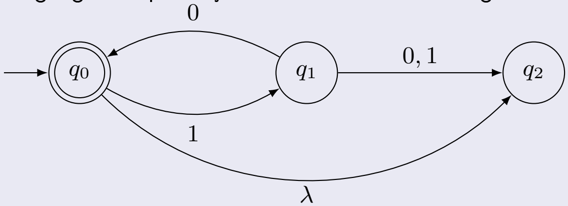
What happens when this automaton is presented with the string $w = 110$? After reading the prefix 11, the automaton finds itself in state q_2, with the transition $\delta(q_2, 0)$ undefined. We call such a situation a *dead configuration*, and we can visualize it as the automaton simply stopping without further action. But we must always keep in mind that such visualizations are imprecise and carry with them some danger of misinterpretation. What we can say precisely is that

$$\delta^*(q_0, 110) = \varnothing.$$

Thus, no final state can be reached by processing $w = 110$, and hence the string is not accepted.

Example 2.10

What is the language accepted by the automaton in the Figure?



It is easy to see from the graph that the only way the NFA can stop in a final state is if the input is either a repetition of the string 10 or the empty string. Therefore, the automaton accepts the language $L = \{(10)^n : n \geqslant 0\}$.

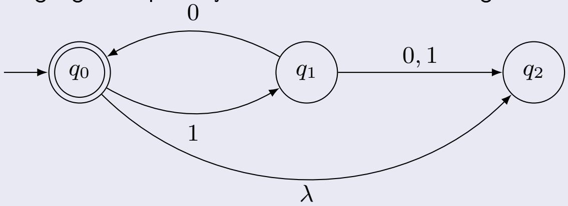
What happens when this automaton is presented with the string $w = 110$? After reading the prefix 11, the automaton finds itself in state q_2, with the transition $\delta(q_2, 0)$ undefined. We call such a situation a *dead configuration*, and we can visualize it as the automaton simply stopping without further action. But we must always keep in mind that such visualizations are imprecise and carry with them some danger of misinterpretation. What we can say precisely is that

$$\delta^*(q_0, 110) = \varnothing.$$

Thus, no final state can be reached by processing $w = 110$, and hence the string is not accepted.

Example 2.10

What is the language accepted by the automaton in the Figure?



It is easy to see from the graph that the only way the NFA can stop in a final state is if the input is either a repetition of the string 10 or the empty string. Therefore, the automaton accepts the language $L = \{(10)^n : n \geqslant 0\}$.

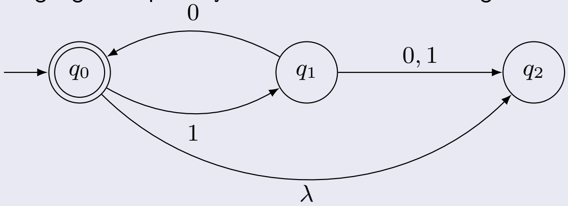
What happens when this automaton is presented with the string $w = 110$? After reading the prefix 11, the automaton finds itself in state q_2, with the transition $\delta(q_2, 0)$ undefined. We call such a situation a *dead configuration*, and we can visualize it as the automaton simply stopping without further action. But we must always keep in mind that such visualizations are imprecise and carry with them some danger of misinterpretation. What we can say precisely is that

$$\delta^*(q_0, 110) = \varnothing.$$

Thus, no final state can be reached by processing $w = 110$, and hence the string is not accepted.

Example 2.10

What is the language accepted by the automaton in the Figure?



It is easy to see from the graph that the only way the NFA can stop in a final state is if the input is either a repetition of the string 10 or the empty string. Therefore, the automaton accepts the language $L = \{(10)^n : n \geqslant 0\}$.

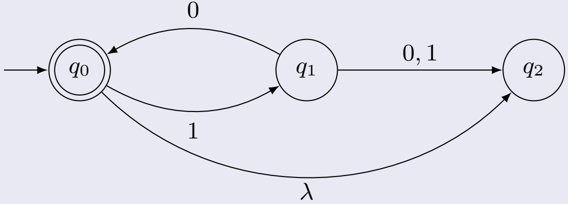
What happens when this automaton is presented with the string $w = 110$? After reading the prefix 11, the automaton finds itself in state q_2, with the transition $\delta(q_2, 0)$ undefined. We call such a situation a *dead configuration*, and we can visualize it as the automaton simply stopping without further action. But we must always keep in mind that such visualizations are imprecise and carry with them some danger of misinterpretation. What we can say precisely is that

$$\delta^*(q_0, 110) = \varnothing.$$

Thus, no final state can be reached by processing $w = 110$, and hence the string is not accepted.

Example 2.10

What is the language accepted by the automaton in the Figure?



It is easy to see from the graph that the only way the NFA can stop in a final state is if the input is either a repetition of the string 10 or the empty string. Therefore, the automaton accepts the language $L = \{(10)^n : n \geqslant 0\}$.

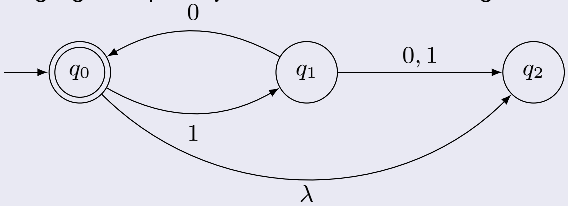
What happens when this automaton is presented with the string $w = 110$? After reading the prefix 11, the automaton finds itself in state q_2, with the transition $\delta(q_2, 0)$ undefined. We call such a situation a *dead configuration*, and we can visualize it as the automaton simply stopping without further action. But we must always keep in mind that such visualizations are imprecise and carry with them some danger of misinterpretation. What we can say precisely is that

$$\delta^*(q_0, 110) = \varnothing.$$

Thus, no final state can be reached by processing $w = 110$, and hence the string is not accepted.

Example 2.10

What is the language accepted by the automaton in the Figure?



It is easy to see from the graph that the only way the NFA can stop in a final state is if the input is either a repetition of the string 10 or the empty string. Therefore, the automaton accepts the language $L = \{(10)^n : n \geqslant 0\}$.

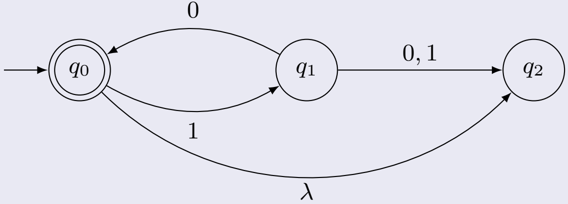
What happens when this automaton is presented with the string $w = 110$? After reading the prefix 11, the automaton finds itself in state q_2, with the transition $\delta(q_2, 0)$ undefined. We call such a situation a *dead configuration*, and we can visualize it as the automaton simply stopping without further action. But we must always keep in mind that such visualizations are imprecise and carry with them some danger of misinterpretation. What we can say precisely is that

$$\delta^*(q_0, 110) = \varnothing.$$

Thus, no final state can be reached by processing $w = 110$, and hence the string is not accepted.

Example 2.10

What is the language accepted by the automaton in the Figure?



It is easy to see from the graph that the only way the NFA can stop in a final state is if the input is either a repetition of the string 10 or the empty string. Therefore, the automaton accepts the language $L = \{(10)^n : n \geqslant 0\}$.

What happens when this automaton is presented with the string $w = 110$? After reading the prefix 11, the automaton finds itself in state q_2, with the transition $\delta(q_2, 0)$ undefined. We call such a situation a *dead configuration*, and we can visualize it as the automaton simply stopping without further action. But we must always keep in mind that such visualizations are imprecise and carry with them some danger of misinterpretation. What we can say precisely is that

$$\delta^*(q_0, 110) = \varnothing.$$

Thus, no final state can be reached by processing $w = 110$, and hence the string is not accepted.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

Nondeterminism is sometimes helpful in solving problems easily. Look at the NFA in the Figure.

It is clear that there is a choice to be made. The first alternative leads to the acceptance of the string a^3, while the second accepts all strings with an even number of a's. The language accepted by the NFA is $\{a^3\} \cup \{a^{2n} : n \geqslant 1\}$. While it is possible to find a DFA for this language, the nondeterminism is quite natural. The language is the union of two quite different sets, and the nondeterminism lets us decide at the outset which case we want. The deterministic solution is not as obviously related to the definition, and so is a little harder to find. As we go on, we will see other and more convincing examples of the usefulness of nondeterminism.

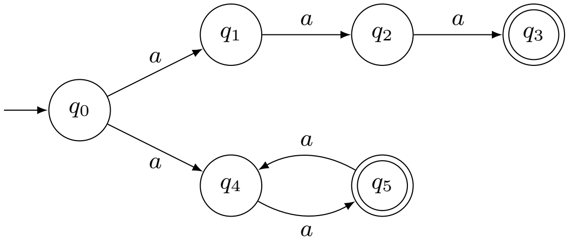
Nondeterminism is sometimes helpful in solving problems easily. Look at the NFA in the Figure.

It is clear that there is a choice to be made. The first alternative leads to the acceptance of the string a^3, while the second accepts all strings with an even number of a's. The language accepted by the NFA is $\{a^3\} \cup \{a^{2n} : n \geqslant 1\}$. While it is possible to find a DFA for this language, the nondeterminism is quite natural. The language is the union of two quite different sets, and the nondeterminism lets us decide at the outset which case we want. The deterministic solution is not as obviously related to the definition, and so is a little harder to find. As we go on, we will see other and more convincing examples of the usefulness of nondeterminism.

Nondeterminism is sometimes helpful in solving problems easily. Look at the NFA in the Figure.

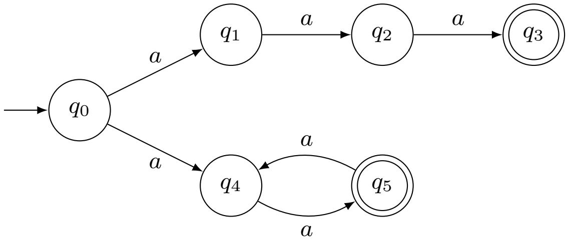
It is clear that there is a choice to be made. The first alternative leads to the acceptance of the string a^3, while the second accepts all strings with an even number of a's. The language accepted by the NFA is $\{a^3\} \cup \{a^{2n} : n \geqslant 1\}$. While it is possible to find a DFA for this language, the nondeterminism is quite natural. The language is the union of two quite different sets, and the nondeterminism lets us decide at the outset which case we want. The deterministic solution is not as obviously related to the definition, and so is a little harder to find. As we go on, we will see other and more convincing examples of the usefulness of nondeterminism.

Nondeterminism is sometimes helpful in solving problems easily. Look at the NFA in the Figure.



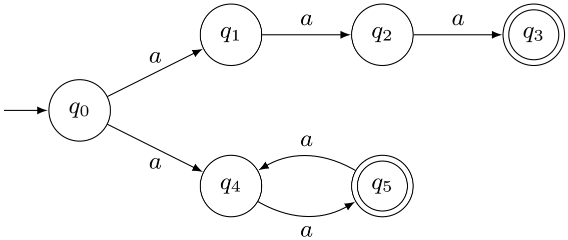
It is clear that there is a choice to be made. The first alternative leads to the acceptance of the string a^3, while the second accepts all strings with an even number of a's. The language accepted by the NFA is $\{a^3\} \cup \{a^{2n} : n \geqslant 1\}$. While it is possible to find a DFA for this language, the nondeterminism is quite natural. The language is the union of two quite different sets, and the nondeterminism lets us decide at the outset which case we want. The deterministic solution is not as obviously related to the definition, and so is a little harder to find. As we go on, we will see other and more convincing examples of the usefulness of nondeterminism.

Nondeterminism is sometimes helpful in solving problems easily. Look at the NFA in the Figure.



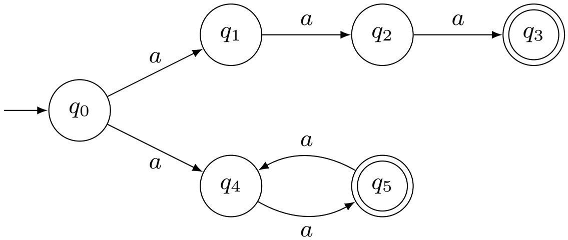
It is clear that there is a choice to be made. The first alternative leads to the acceptance of the string a^3, while the second accepts all strings with an even number of a's. The language accepted by the NFA is $\{a^3\} \cup \{a^{2n} : n \geqslant 1\}$. While it is possible to find a DFA for this language, the nondeterminism is quite natural. The language is the union of two quite different sets, and the nondeterminism lets us decide at the outset which case we want. The deterministic solution is not as obviously related to the definition, and so is a little harder to find. As we go on, we will see other and more convincing examples of the usefulness of nondeterminism.

Nondeterminism is sometimes helpful in solving problems easily. Look at the NFA in the Figure.



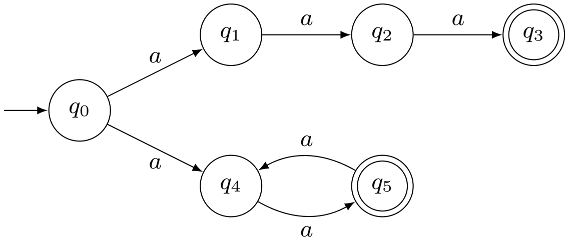
It is clear that there is a choice to be made. The first alternative leads to the acceptance of the string a^3, while the second accepts all strings with an even number of a's. The language accepted by the NFA is $\{a^3\} \cup \{a^{2n} : n \geqslant 1\}$. While it is possible to find a DFA for this language, the nondeterminism is quite natural. The language is the union of two quite different sets, and the nondeterminism lets us decide at the outset which case we want. The deterministic solution is not as obviously related to the definition, and so is a little harder to find. As we go on, we will see other and more convincing examples of the usefulness of nondeterminism.

Nondeterminism is sometimes helpful in solving problems easily. Look at the NFA in the Figure.



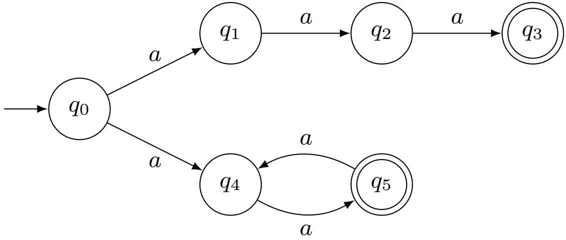
It is clear that there is a choice to be made. The first alternative leads to the acceptance of the string a^3, while the second accepts all strings with an even number of a's. The language accepted by the NFA is $\{a^3\} \cup \{a^{2n} : n \geqslant 1\}$. While it is possible to find a DFA for this language, the nondeterminism is quite natural. The language is the union of two quite different sets, and the nondeterminism lets us decide at the outset which case we want. The deterministic solution is not as obviously related to the definition, and so is a little harder to find. As we go on, we will see other and more convincing examples of the usefulness of nondeterminism.

Nondeterminism is sometimes helpful in solving problems easily. Look at the NFA in the Figure.



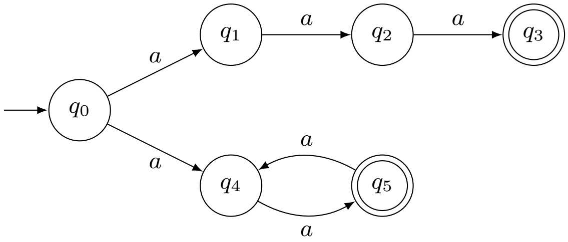
It is clear that there is a choice to be made. The first alternative leads to the acceptance of the string a^3, while the second accepts all strings with an even number of a's. The language accepted by the NFA is $\{a^3\} \cup \{a^{2n} : n \geqslant 1\}$.
While it is possible to find a DFA for this language, the nondeterminism is quite natural. The language is the union of two quite different sets, and the nondeterminism lets us decide at the outset which case we want. The deterministic solution is not as obviously related to the definition, and so is a little harder to find. As we go on, we will see other and more convincing examples of the usefulness of nondeterminism.

Nondeterminism is sometimes helpful in solving problems easily. Look at the NFA in the Figure.



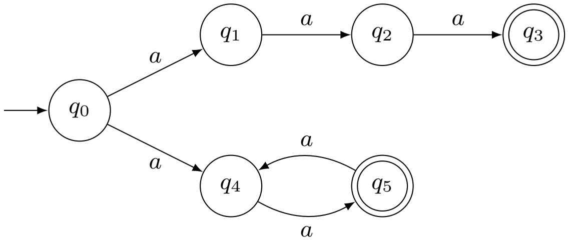
It is clear that there is a choice to be made. The first alternative leads to the acceptance of the string a^3, while the second accepts all strings with an even number of a's. The language accepted by the NFA is $\{a^3\} \cup \{a^{2n} : n \geqslant 1\}$. While it is possible to find a DFA for this language, the nondeterminism is quite natural. The language is the union of two quite different sets, and the nondeterminism lets us decide at the outset which case we want. The deterministic solution is not as obviously related to the definition, and so is a little harder to find. As we go on, we will see other and more convincing examples of the usefulness of nondeterminism.

Nondeterminism is sometimes helpful in solving problems easily. Look at the NFA in the Figure.



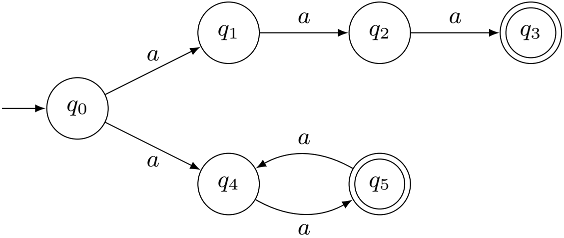
It is clear that there is a choice to be made. The first alternative leads to the acceptance of the string a^3, while the second accepts all strings with an even number of a's. The language accepted by the NFA is $\{a^3\} \cup \{a^{2n} : n \geqslant 1\}$. While it is possible to find a DFA for this language, the nondeterminism is quite natural. The language is the union of two quite different sets, and the nondeterminism lets us decide at the outset which case we want. The deterministic solution is not as obviously related to the definition, and so is a little harder to find. As we go on, we will see other and more convincing examples of the usefulness of nondeterminism.

Nondeterminism is sometimes helpful in solving problems easily. Look at the NFA in the Figure.



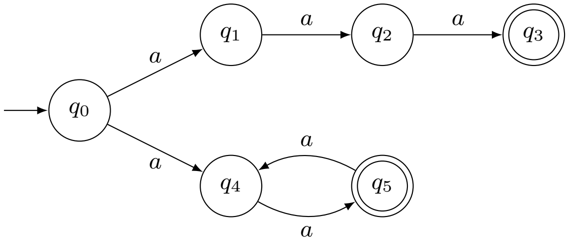
It is clear that there is a choice to be made. The first alternative leads to the acceptance of the string a^3, while the second accepts all strings with an even number of a's. The language accepted by the NFA is $\{a^3\} \cup \{a^{2n} : n \geqslant 1\}$. While it is possible to find a DFA for this language, the nondeterminism is quite natural. The language is the union of two quite different sets, and the nondeterminism lets us decide at the outset which case we want. The deterministic solution is not as obviously related to the definition, and so is a little harder to find. As we go on, we will see other and more convincing examples of the usefulness of nondeterminism.

Nondeterminism is sometimes helpful in solving problems easily. Look at the NFA in the Figure.



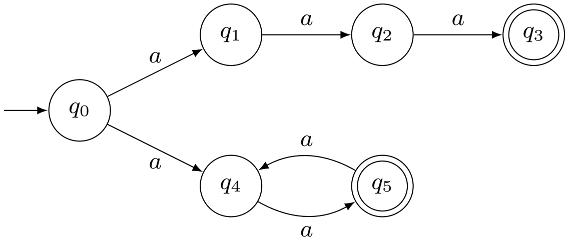
It is clear that there is a choice to be made. The first alternative leads to the acceptance of the string a^3, while the second accepts all strings with an even number of a's. The language accepted by the NFA is $\{a^3\} \cup \{a^{2n} : n \geqslant 1\}$. While it is possible to find a DFA for this language, the nondeterminism is quite natural. The language is the union of two quite different sets, and the nondeterminism lets us decide at the outset which case we want. The deterministic solution is not as obviously related to the definition, and so is a little harder to find. As we go on, we will see other and more convincing examples of the usefulness of nondeterminism.

Nondeterminism is sometimes helpful in solving problems easily. Look at the NFA in the Figure.



It is clear that there is a choice to be made. The first alternative leads to the acceptance of the string a^3, while the second accepts all strings with an even number of a's. The language accepted by the NFA is $\{a^3\} \cup \{a^{2n} : n \geqslant 1\}$. While it is possible to find a DFA for this language, the nondeterminism is quite natural. The language is the union of two quite different sets, and the nondeterminism lets us decide at the outset which case we want. The deterministic solution is not as obviously related to the definition, and so is a little harder to find. As we go on, we will see other and more convincing examples of the usefulness of nondeterminism.

Nondeterminism is sometimes helpful in solving problems easily. Look at the NFA in the Figure.



It is clear that there is a choice to be made. The first alternative leads to the acceptance of the string a^3, while the second accepts all strings with an even number of a's. The language accepted by the NFA is $\{a^3\} \cup \{a^{2n} : n \geqslant 1\}$. While it is possible to find a DFA for this language, the nondeterminism is quite natural. The language is the union of two quite different sets, and the nondeterminism lets us decide at the outset which case we want. The deterministic solution is not as obviously related to the definition, and so is a little harder to find. As we go on, we will see other and more convincing examples of the usefulness of nondeterminism.

In the same vein, nondeterminism is an effective mechanism for describing some complicated languages concisely. Notice that the definition of a grammar involves a nondeterministic element. In

$$S \to aSb|\lambda$$

we can at any point choose either the first or the second production. This lets us specify many different strings using only two rules.

Finally, there is a technical reason for introducing nondeterminism. As we will see, certain theoretical results are more easily established for NFA's than for DFA's. Our next major result indicates that there is no essential difference between these two types of automata. Consequently, allowing nondeterminism often simplifies formal arguments without affecting the generality of the conclusion.

In the same vein, nondeterminism is an effective mechanism for describing some complicated languages concisely. Notice that the definition of a grammar involves a nondeterministic element. In

$$S \to aSb|\lambda$$

we can at any point choose either the first or the second production. This lets us specify many different strings using only two rules.

Finally, there is a technical reason for introducing nondeterminism. As we will see, certain theoretical results are more easily established for NFA's than for DFA's. Our next major result indicates that there is no essential difference between these two types of automata. Consequently, allowing nondeterminism often simplifies formal arguments without affecting the generality of the conclusion.

In the same vein, nondeterminism is an effective mechanism for describing some complicated languages concisely. Notice that the definition of a grammar involves a nondeterministic element. In

$$S \to aSb|\lambda$$

we can at any point choose either the first or the second production. This lets us specify many different strings using only two rules.

Finally, there is a technical reason for introducing nondeterminism. As we will see, certain theoretical results are more easily established for NFA's than for DFA's. Our next major result indicates that there is no essential difference between these two types of automata. Consequently, allowing nondeterminism often simplifies formal arguments without affecting the generality of the conclusion.

In the same vein, nondeterminism is an effective mechanism for describing some complicated languages concisely. Notice that the definition of a grammar involves a nondeterministic element. In

$$S \to aSb|\lambda$$

we can at any point choose either the first or the second production. This lets us specify many different strings using only two rules.

Finally, there is a technical reason for introducing nondeterminism. As we will see, certain theoretical results are more easily established for NFA's than for DFA's. Our next major result indicates that there is no essential difference between these two types of automata. Consequently, allowing nondeterminism often simplifies formal arguments without affecting the generality of the conclusion.

In the same vein, nondeterminism is an effective mechanism for describing some complicated languages concisely. Notice that the definition of a grammar involves a nondeterministic element. In

$$S \to aSb|\lambda$$

we can at any point choose either the first or the second production. This lets us specify many different strings using only two rules.

Finally, there is a technical reason for introducing nondeterminism. As we will see, certain theoretical results are more easily established for NFA's than for DFA's. Our next major result indicates that there is no essential difference between these two types of automata. Consequently, allowing nondeterminism often simplifies formal arguments without affecting the generality of the conclusion.

In the same vein, nondeterminism is an effective mechanism for describing some complicated languages concisely. Notice that the definition of a grammar involves a nondeterministic element. In

$$S \to aSb|\lambda$$

we can at any point choose either the first or the second production. This lets us specify many different strings using only two rules.

Finally, there is a technical reason for introducing nondeterminism. As we will see, certain theoretical results are more easily established for NFA's than for DFA's. Our next major result indicates that there is no essential difference between these two types of automata. Consequently, allowing nondeterminism often simplifies formal arguments without affecting the generality of the conclusion.

In the same vein, nondeterminism is an effective mechanism for describing some complicated languages concisely. Notice that the definition of a grammar involves a nondeterministic element. In

$$S \to aSb|\lambda$$

we can at any point choose either the first or the second production. This lets us specify many different strings using only two rules.

Finally, there is a technical reason for introducing nondeterminism. As we will see, certain theoretical results are more easily established for NFA's than for DFA's. Our next major result indicates that there is no essential difference between these two types of automata. Consequently, allowing nondeterminism often simplifies formal arguments without affecting the generality of the conclusion.

In the same vein, nondeterminism is an effective mechanism for describing some complicated languages concisely. Notice that the definition of a grammar involves a nondeterministic element. In

$$S \to aSb|\lambda$$

we can at any point choose either the first or the second production. This lets us specify many different strings using only two rules.

Finally, there is a technical reason for introducing nondeterminism. As we will see, certain theoretical results are more easily established for NFA's than for DFA's. Our next major result indicates that there is no essential difference between these two types of automata. Consequently, allowing nondeterminism often simplifies formal arguments without affecting the generality of the conclusion.

In the same vein, nondeterminism is an effective mechanism for describing some complicated languages concisely. Notice that the definition of a grammar involves a nondeterministic element. In

$$S \to aSb|\lambda$$

we can at any point choose either the first or the second production. This lets us specify many different strings using only two rules.

Finally, there is a technical reason for introducing nondeterminism. As we will see, certain theoretical results are more easily established for NFA's than for DFA's. Our next major result indicates that there is no essential difference between these two types of automata. Consequently, allowing nondeterminism often simplifies formal arguments without affecting the generality of the conclusion.

In the same vein, nondeterminism is an effective mechanism for describing some complicated languages concisely. Notice that the definition of a grammar involves a nondeterministic element. In

$$S \to aSb|\lambda$$

we can at any point choose either the first or the second production. This lets us specify many different strings using only two rules.

Finally, there is a technical reason for introducing nondeterminism. As we will see, certain theoretical results are more easily established for NFA's than for DFA's. Our next major result indicates that there is no essential difference between these two types of automata. Consequently, allowing nondeterminism often simplifies formal arguments without affecting the generality of the conclusion.

In the same vein, nondeterminism is an effective mechanism for describing some complicated languages concisely. Notice that the definition of a grammar involves a nondeterministic element. In

$$S \to aSb|\lambda$$

we can at any point choose either the first or the second production. This lets us specify many different strings using only two rules.

Finally, there is a technical reason for introducing nondeterminism. As we will see, certain theoretical results are more easily established for NFA's than for DFA's. Our next major result indicates that there is no essential difference between these two types of automata. Consequently, allowing nondeterminism often simplifies formal arguments without affecting the generality of the conclusion.

Thank You for attention!