

Vorlesung: Graphalgorithmen

Zusammenfassung und Ausblick

Julian Schäfer

Julian.Schaefer@hs-karlsruhe.de

Hochschule Karlsruhe - Technik und Wirtschaft
Fakultät für Informatik und Wirtschaftsinformatik

25. Januar 2018



Agenda

- 1 Zusammenfassung der Vorlesung
- 2 Informationen zur Klausur
- 3 Ausblick in die Praxis
- 4 Werbung

Agenda

- 1 Zusammenfassung der Vorlesung
- 2 Informationen zur Klausur
- 3 Ausblick in die Praxis
- 4 Werbung

Agenda

- 1 Zusammenfassung der Vorlesung
- 2 Informationen zur Klausur
- 3 Ausblick in die Praxis
- 4 Werbung

Agenda

- 1 Zusammenfassung der Vorlesung
- 2 Informationen zur Klausur
- 3 Ausblick in die Praxis
- 4 Werbung

Konnektivität von Graphen

- Definitionen

- ▶ (un)gerichtete Graphen $G = (V, E)$
- ▶ Weg, Länge, Kreis (bzw. Zyklus)
- ▶ Knotengrad, Eingangs- und Ausgangsgrad
- ▶ Zusammenhangskomponenten und Zusammenhang
- ▶ Bäume und Eigenschaften

- Durchmusterungsmethoden

- ▶ Algorithmus GRAPHDURCHMUSTERUNG
- ▶ Algorithmus TIEFENSUCHE (DFS)
- ▶ Algorithmus BREITENSUCHE (BFS)

- Starke Zusammenhangskomponenten

- ▶ Definition von starken Zusammenhangskomponenten
- ▶ Algorithmus GEGENSEITIGE ERREICHBARKEIT
- ▶ Algorithmus STARKE ZUSAMMENHANGSKOMPONENTEN

- Topologische Sortierung

- ▶ Definition von topologischer Sortierung
- ▶ Algorithmus TOPOLOGISCHE SORTIERUNG

Konnektivität von Graphen

- Definitionen

- ▶ (un)gerichtete Graphen $G = (V, E)$
- ▶ Weg, Länge, Kreis (bzw. Zyklus)
- ▶ Knotengrad, Eingangs- und Ausgangsgrad
- ▶ Zusammenhangskomponenten und Zusammenhang
- ▶ Bäume und Eigenschaften

- Durchmusterungsmethoden

- ▶ Algorithmus GRAPHDURCHMUSTERUNG
- ▶ Algorithmus TIEFENSUCHE (DFS)
- ▶ Algorithmus BREITENSUCHE (BFS)

- Starke Zusammenhangskomponenten

- ▶ Definition von starken Zusammenhangskomponenten
- ▶ Algorithmus GEGENSEITIGE ERREICHBARKEIT
- ▶ Algorithmus STARKE ZUSAMMENHANGSKOMPONENTEN

- Topologische Sortierung

- ▶ Definition von topologischer Sortierung
- ▶ Algorithmus TOPOLOGISCHE SORTIERUNG

Konnektivität von Graphen

- Definitionen

- ▶ (un)gerichtete Graphen $G = (V, E)$
- ▶ Weg, Länge, Kreis (bzw. Zyklus)
- ▶ Knotengrad, Eingangs- und Ausgangsgrad
- ▶ Zusammenhangskomponenten und Zusammenhang
- ▶ Bäume und Eigenschaften

- Durchmusterungsmethoden

- ▶ Algorithmus GRAPHDURCHMUSTERUNG
- ▶ Algorithmus TIEFENSUCHE (DFS)
- ▶ Algorithmus BREITENSUCHE (BFS)

- Starke Zusammenhangskomponenten

- ▶ Definition von starken Zusammenhangskomponenten
- ▶ Algorithmus GEGENSEITIGE ERREICHBARKEIT
- ▶ Algorithmus STARKE ZUSAMMENHANGSKOMPONENTEN

- Topologische Sortierung

- ▶ Definition von topologischer Sortierung
- ▶ Algorithmus TOPOLOGISCHE SORTIERUNG

Konnektivität von Graphen

- Definitionen

- ▶ (un)gerichtete Graphen $G = (V, E)$
- ▶ Weg, Länge, Kreis (bzw. Zyklus)
- ▶ Knotengrad, Eingangs- und Ausgangsgrad
- ▶ Zusammenhangskomponenten und Zusammenhang
- ▶ Bäume und Eigenschaften

- Durchmusterungsmethoden

- ▶ Algorithmus GRAPHDURCHMUSTERUNG
- ▶ Algorithmus TIEFENSUCHE (DFS)
- ▶ Algorithmus BREITENSUCHE (BFS)

- Starke Zusammenhangskomponenten

- ▶ Definition von starken Zusammenhangskomponenten
- ▶ Algorithmus GEGENSEITIGE ERREICHBARKEIT
- ▶ Algorithmus STARKE ZUSAMMENHANGSKOMPONENTEN

- Topologische Sortierung

- ▶ Definition von topologischer Sortierung
- ▶ Algorithmus TOPOLOGISCHE SORTIERUNG

Konnektivität von Graphen

- Definitionen
 - ▶ (un)gerichtete Graphen $G = (V, E)$
 - ▶ Weg, Länge, Kreis (bzw. Zyklus)
 - ▶ Knotengrad, Eingangs- und Ausgangsgrad
 - ▶ Zusammenhangskomponenten und Zusammenhang
 - ▶ Bäume und Eigenschaften
- Durchmusterungsmethoden
 - ▶ Algorithmus GRAPHDURCHMUSTERUNG
 - ▶ Algorithmus TIEFENSUCHE (DFS)
 - ▶ Algorithmus BREITENSUCHE (BFS)
- Starke Zusammenhangskomponenten
 - ▶ Definition von starken Zusammenhangskomponenten
 - ▶ Algorithmus GEGENSEITIGE ERREICHBARKEIT
 - ▶ Algorithmus STARKE ZUSAMMENHANGSKOMPONENTEN
- Topologische Sortierung
 - ▶ Definition von topologischer Sortierung
 - ▶ Algorithmus TOPOLOGISCHE SORTIERUNG

Kürzeste Wege

- Definitionen
 - ▶ (kanten-)gewichteter Digraph $G = (V, E, c)$
- Berechnung kürzester Wege von einem Startknoten
 - ▶ Algorithmus DIJKSTRA'S ALGORITHMUS
 - ▶ Algorithmus MOORE-BELLMAN-FORD
- Ermittlung von negativen Kreisen
 - ▶ Algorithmus TEST AUF NEGATIVE ZYKLEN
- Die A*-Strategie zur Verbesserung der Suchrichtung
 - ▶ Verwendung einer Schätzfunktion $h : V \mapsto \mathbb{R}_0^+$
 - ▶ Algorithmus A*-STRATEGIE

Kürzeste Wege

- Definitionen
 - ▶ (kanten-)gewichteter Digraph $G = (V, E, c)$
- Berechnung kürzester Wege von einem Startknoten
 - ▶ Algorithmus DIJKSTRA'S ALGORITHMUS
 - ▶ Algorithmus MOORE-BELLMAN-FORD
- Ermittlung von negativen Kreisen
 - ▶ Algorithmus TEST AUF NEGATIVE ZYKLEN
- Die A*-Strategie zur Verbesserung der Suchrichtung
 - ▶ Verwendung einer Schätzfunktion $h : V \mapsto \mathbb{R}_0^+$
 - ▶ Algorithmus A*-STRATEGIE

Kürzeste Wege

- Definitionen
 - ▶ (kanten-)gewichteter Digraph $G = (V, E, c)$
- Berechnung kürzester Wege von einem Startknoten
 - ▶ Algorithmus DIJKSTRA'S ALGORITHMUS
 - ▶ Algorithmus MOORE-BELLMAN-FORD
- Ermittlung von negativen Kreisen
 - ▶ Algorithmus TEST AUF NEGATIVE ZYKLEN
- Die A*-Strategie zur Verbesserung der Suchrichtung
 - ▶ Verwendung einer Schätzfunktion $h : V \mapsto \mathbb{R}_0^+$
 - ▶ Algorithmus A*-STRATEGIE

Kürzeste Wege

- Definitionen
 - ▶ (kanten-)gewichteter Digraph $G = (V, E, c)$
- Berechnung kürzester Wege von einem Startknoten
 - ▶ Algorithmus DIJKSTRA'S ALGORITHMUS
 - ▶ Algorithmus MOORE-BELLMAN-FORD
- Ermittlung von negativen Kreisen
 - ▶ Algorithmus TEST AUF NEGATIVE ZYKLEN
- Die A*-Strategie zur Verbesserung der Suchrichtung
 - ▶ Verwendung einer Schätzfunktion $h : V \mapsto \mathbb{R}_0^+$
 - ▶ Algorithmus A*-STRATEGIE

Kürzeste Wege

- Definitionen
 - ▶ (kanten-)gewichteter Digraph $G = (V, E, c)$
- Berechnung kürzester Wege von einem Startknoten
 - ▶ Algorithmus DIJKSTRA'S ALGORITHMUS
 - ▶ Algorithmus MOORE-BELLMAN-FORD
- Ermittlung von negativen Kreisen
 - ▶ Algorithmus TEST AUF NEGATIVE ZYKLEN
- Die A*-Strategie zur Verbesserung der Suchrichtung
 - ▶ Verwendung einer Schätzfunktion $h : V \mapsto \mathbb{R}_0^+$
 - ▶ Algorithmus A*-STRATEGIE

Erlaubte Hilfsmittel

- **Eine** DIN A4-Seite (beidseitig) mit Notizen
- Stifte (nicht entfernbar)

Tipps zur Vorbereitung

- Bearbeiten sie alle Übungsblätter
- Erstellen sie sich eigene Beispiele
- Versuchen sie Beispiel zu finden, die die Schwächen der uns bekannten Graphalgorithmen aufzeigen

Verifikation

- statisch
- "Are we building the product right?"
- Theorembeweiser (z.B. Isabelle/HOL, Coq, PVS)
 - ▶ Basieren auf funktionalen Sprachen und Typen (z.B. ML)
 - ▶ Spezifikation mithilfe von Logiken höherer Ordnung und Funktionen
 - ▶ Implementierung von Algorithmen mit Funktionen und abstrakten Datentypen
 - ▶ Beweisführung durch die Anwendung von Regeln
- Modellprüfung (z.B. Promella/Spin, JavaPathFinder)
 - ▶ Im Kern der Prüfung steht die Frage: $M \stackrel{?}{\models} \phi$
 - ▶ ϕ repräsentiert die Eigenschaft (z.B. Freiheit von Deadlocks), diese wird in speziellen Logiken definiert (z.B. Linare-Zeit Logik LTL)
 - ▶ M repräsentiert das zu prüfende Modell (z.B. endlichen Automaten)
 - ▶ Mithilfe einer Durchsuchung des Zustandsraumes kann sichergestellt werden, dass das Modell die geforderte Eigenschaft erfüllt (oder nicht)
 - ▶ Es können Gegenbeispiele gefunden werden

JavaPathFinder: <http://javapathfinder.sourceforge.net>

Isabelle: <https://isabelle.in.tum.de>

Spin: <http://spinroot.com/spin/whatispin.html>

Weitere Tools zur Modellprüfung: https://en.wikipedia.org/wiki/List_of_model_checking_tools

Verifikation

- statisch
- "Are we building the product right?"
- Theorembeweiser (z.B. Isabelle/HOL, Coq, PVS)
 - ▶ Basieren auf funktionalen Sprachen und Typen (z.B. ML)
 - ▶ Spezifikation mithilfe von Logiken höherer Ordnung und Funktionen
 - ▶ Implementierung von Algorithmen mit Funktionen und abstrakten Datentypen
 - ▶ Beweisführung durch die Anwendung von Regeln
- Modellprüfung (z.B. Promella/Spin, JavaPathFinder)
 - ▶ Im Kern der Prüfung steht die Frage: $M \stackrel{?}{\models} \phi$
 - ▶ ϕ repräsentiert die Eigenschaft (z.B. Freiheit von Deadlocks), diese wird in speziellen Logiken definiert (z.B. Linare-Zeit Logik LTL)
 - ▶ M repräsentiert das zu prüfende Modell (z.B. endlichen Automaten)
 - ▶ Mithilfe einer Durchsuchung des Zustandsraumes kann sichergestellt werden, dass das Modell die geforderte Eigenschaft erfüllt (oder nicht)
 - ▶ Es können Gegenbeispiele gefunden werden

JavaPathFinder: <http://javapathfinder.sourceforge.net>

Isabelle: <https://isabelle.in.tum.de>

Spin: <http://spinroot.com/spin/whatispin.html>

Weitere Tools zur Modellprüfung: https://en.wikipedia.org/wiki/List_of_model_checking_tools

Verifikation

- statisch
- "Are we building the product right?"
- Theorembeweiser (z.B. Isabelle/HOL, Coq, PVS)
 - ▶ Basieren auf funktionalen Sprachen und Typen (z.B. ML)
 - ▶ Spezifikation mithilfe von Logiken höherer Ordnung und Funktionen
 - ▶ Implementierung von Algorithmen mit Funktionen und abstrakten Datentypen
 - ▶ Beweisführung durch die Anwendung von Regeln
- Modellprüfung (z.B. Promella/Spin, JavaPathFinder)
 - ▶ Im Kern der Prüfung steht die Frage: $M \stackrel{?}{\models} \phi$
 - ▶ ϕ repräsentiert die Eigenschaft (z.B. Freiheit von Deadlocks), diese wird in speziellen Logiken definiert (z.B. Linare-Zeit Logik LTL)
 - ▶ M repräsentiert das zu prüfende Modell (z.B. endlichen Automaten)
 - ▶ Mithilfe einer Durchsuchung des Zustandsraumes kann sichergestellt werden, dass das Modell die geforderte Eigenschaft erfüllt (oder nicht)
 - ▶ Es können Gegenbeispiele gefunden werden

JavaPathFinder: <http://javapathfinder.sourceforge.net>

Isabelle: <https://isabelle.in.tum.de>

Spin: <http://spinroot.com/spin/whatispin.html>

Weitere Tools zur Modellprüfung: https://en.wikipedia.org/wiki/List_of_model_checking_tools

Verifikation

- statisch
- "Are we building the product right?"
- Theorembeweiser (z.B. Isabelle/HOL, Coq, PVS)
 - ▶ Basieren auf funktionalen Sprachen und Typen (z.B. ML)
 - ▶ Spezifikation mithilfe von Logiken höherer Ordnung und Funktionen
 - ▶ Implementierung von Algorithmen mit Funktionen und abstrakten Datentypen
 - ▶ Beweisführung durch die Anwendung von Regeln
- Modellprüfung (z.B. Promella/Spin, JavaPathFinder)
 - ▶ Im Kern der Prüfung steht die Frage: $M \stackrel{?}{\models} \phi$
 - ▶ ϕ repräsentiert die Eigenschaft (z.B. Freiheit von Deadlocks), diese wird in speziellen Logiken definiert (z.B. Lineare-Zeit Logik LTL)
 - ▶ M repräsentiert das zu prüfende Modell (z.B. endlichen Automaten)
 - ▶ Mithilfe einer Durchsuchung des Zustandsraumes kann sichergestellt werden, dass das Modell die geforderte Eigenschaft erfüllt (oder nicht)
 - ▶ Es können Gegenbeispiele gefunden werden

JavaPathFinder: <http://javapathfinder.sourceforge.net>

Isabelle: <https://isabelle.in.tum.de>

Spin: <http://spinroot.com/spin/whatispin.html>

Weitere Tools zur Modellprüfung: https://en.wikipedia.org/wiki/List_of_model_checking_tools

Verifikation

- statisch
- "Are we building the product right?"
- Theorembeweiser (z.B. Isabelle/HOL, Coq, PVS)
 - ▶ Basieren auf funktionalen Sprachen und Typen (z.B. ML)
 - ▶ Spezifikation mithilfe von Logiken höherer Ordnung und Funktionen
 - ▶ Implementierung von Algorithmen mit Funktionen und abstrakten Datentypen
 - ▶ Beweisführung durch die Anwendung von Regeln
- Modellprüfung (z.B. Promella/Spin, JavaPathFinder)
 - ▶ Im Kern der Prüfung steht die Frage: $M \stackrel{?}{\models} \phi$
 - ▶ ϕ repräsentiert die Eigenschaft (z.B. Freiheit von Deadlocks), diese wird in speziellen Logiken definiert (z.B. Linare-Zeit Logik LTL)
 - ▶ M repräsentiert das zu prüfende Modell (z.B. endlichen Automaten)
 - ▶ Mithilfe einer Durchsuchung des Zustandsraumes kann sichergestellt werden, dass das Modell die geforderte Eigenschaft erfüllt (oder nicht)
 - ▶ Es können Gegenbeispiele gefunden werden

JavaPathFinder: <http://javapathfinder.sourceforge.net>

Isabelle: <https://isabelle.in.tum.de>

Spin: <http://spinroot.com/spin/whatispin.html>

Weitere Tools zur Modellprüfung: https://en.wikipedia.org/wiki/List_of_model_checking_tools



Validation

- dynamisch
- "Are we building the right product?"
- Tests
 - ▶ Arten: Unit, Integration, Akzeptanz, ...
 - ▶ Testframeworks stehen für jede seriöse Programmierumgebung bereit (z.B. JUnit, Fitness, JBehave)
 - ▶ Tests prüfen Stichprobenhaft verschiedenen Aspekte der Anwendung
 - ▶ Regelmäßige Ausführung schützt vor Regressionen (Stichwort Sicherheitsnetz)
 - ▶ Test-Abdeckung kann erhöht werden um Risiko von Änderungen zu minimieren
- Design by Contract
 - ▶ Defensiver Programmierstil
 - ▶ Idee: Vorbedingung, Invarianten und Nachbedingung der Komponenten sind explizit im Code formuliert und werden geprüft
 - ▶ Ursprünglich für Programmiersprache Eiffel, mittlerweile gibt es für die meisten Mainstream-Sprachen Support (z.B. Java)

Design by Contract: https://en.wikipedia.org/wiki/Design_by_contract

Contracts for Java: <https://github.com/nhatminhle/cofoja>

C4J: <https://c4j-team.github.io/C4J/index.html>

Valid4J: <http://www.valid4j.org/>

Validation

- dynamisch
- "Are we building the right product?"
- Tests
 - ▶ Arten: Unit, Integration, Akzeptanz, ...
 - ▶ Testframeworks stehen für jede seriöse Programmierumgebung bereit (z.B. JUnit, Fitness, JBehave)
 - ▶ Tests prüfen Stichprobenhaft verschiedenen Aspekte der Anwendung
 - ▶ Regelmäßige Ausführung schützt vor Regressionen (Stichwort Sicherheitsnetz)
 - ▶ Test-Abdeckung kann erhöht werden um Risiko von Änderungen zu minimieren
- Design by Contract
 - ▶ Defensiver Programmierstil
 - ▶ Idee: Vorbedingung, Invarianten und Nachbedingung der Komponenten sind explizit im Code formuliert und werden geprüft
 - ▶ Ursprünglich für Programmiersprache Eiffel, mittlerweile gibt es für die meisten Mainstream-Sprachen Support (z.B. Java)

Design by Contract: https://en.wikipedia.org/wiki/Design_by_contract

Contracts for Java: <https://github.com/nhatminhle/cofoja>

C4J: <https://c4j-team.github.io/C4J/index.html>

Valid4J: <http://www.valid4j.org/>

Validation

- dynamisch
- "Are we building the right product?"
- Tests
 - ▶ Arten: Unit, Integration, Akzeptanz, ...
 - ▶ Testframeworks stehen für jede seriöse Programmierumgebung bereit (z.B. JUnit, Fitness, JBehave)
 - ▶ Tests prüfen Stichprobenhaft verschiedenen Aspekte der Anwendung
 - ▶ Regelmäßige Ausführung schützt vor Regressionen (Stichwort Sicherheitsnetz)
 - ▶ Test-Abdeckung kann erhöht werden um Risiko von Änderungen zu minimieren
- Design by Contract
 - ▶ Defensiver Programmierstil
 - ▶ Idee: Vorbedingung, Invarianten und Nachbedingung der Komponenten sind explizit im Code formuliert und werden geprüft
 - ▶ Ursprünglich für Programmiersprache Eiffel, mittlerweile gibt es für die meisten Mainstream-Sprachen Support (z.B. Java)

Design by Contract: https://en.wikipedia.org/wiki/Design_by_contract

Contracts for Java: <https://github.com/nhatminhle/cofoja>

C4J: <https://c4j-team.github.io/C4J/index.html>

Valid4J: <http://www.valid4j.org/>

Validation

- dynamisch
- "Are we building the right product?"
- Tests
 - ▶ Arten: Unit, Integration, Akzeptanz, ...
 - ▶ Testframeworks stehen für jede seriöse Programmierumgebung bereit (z.B. JUnit, Fitness, JBehave)
 - ▶ Tests prüfen Stichprobenhaft verschiedenen Aspekte der Anwendung
 - ▶ Regelmäßige Ausführung schützt vor Regressionen (Stichwort Sicherheitsnetz)
 - ▶ Test-Abdeckung kann erhöht werden um Risiko von Änderungen zu minimieren
- Design by Contract
 - ▶ Defensiver Programmierstil
 - ▶ Idee: Vorbedingung, Invarianten und Nachbedingung der Komponenten sind explizit im Code formuliert und werden geprüft
 - ▶ Ursprünglich für Programmiersprache Eiffel, mittlerweile gibt es für die meisten Mainstream-Sprachen Support (z.B. Java)

Design by Contract: https://en.wikipedia.org/wiki/Design_by_contract

Contracts for Java: <https://github.com/nhatminhle/cofoja>

C4J: <https://c4j-team.github.io/C4J/index.html>

Valid4J: <http://www.valid4j.org/>

Validation

- dynamisch
- "Are we building the right product?"
- Tests
 - ▶ Arten: Unit, Integration, Akzeptanz, ...
 - ▶ Testframeworks stehen für jede seriöse Programmierungsumgebung bereit (z.B. JUnit, Fitness, JBehave)
 - ▶ Tests prüfen Stichprobenhaft verschiedenen Aspekte der Anwendung
 - ▶ Regelmäßige Ausführung schützt vor Regressionen (Stichwort Sicherheitsnetz)
 - ▶ Test-Abdeckung kann erhöht werden um Risiko von Änderungen zu minimieren
- Design by Contract
 - ▶ Defensiver Programmierstil
 - ▶ Idee: Vorbedingung, Invarianten und Nachbedingung der Komponenten sind explizit im Code formuliert und werden geprüft
 - ▶ Ursprünglich für Programmiersprache Eiffel, mittlerweile gibt es für die meisten Mainstream-Sprachen Support (z.B. Java)

Design by Contract: https://en.wikipedia.org/wiki/Design_by_contract

Contracts for Java: <https://github.com/nhatminhle/cofoja>

C4J: <https://c4j-team.github.io/C4J/index.html>

Valid4J: <http://www.valid4j.org/>

CAS Software AG



<http://www.cas-selbst-erleben.de>