Algorithmen und Programmierung

Algorithmen II

Dr. Felix Jonathan Boes
boes@cs.uni-bonn.de
Institut für Informatik

Ausblick: Kürzeste und möglichst kurze Wege

In dieser Vorlesung werden wir ein interessantes Problem diskutieren

Dadurch motivieren wir die kommenden Vorlesungsinhalte

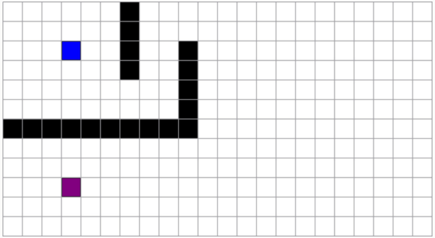
Außerdem erhalten Sie einen Einblick in den Vorlesungsinhalt der kommenden Semester

Ziel

Sie lernen ein Greedyverfahren kennen um möglichst kurze Wege möglichst schnell zu finden

Problemstellung und Ziel

Das **erklärte Ziel** ist es von einem Startpunkt aus möglichst schnell auf einem möglichst kurzen Weg zum Zielpunkt zu gelangen. Wir untersuchen die Problemstellung.



Wie findet man möglichst schnell einen möglichst kurzen Weg

- wenn man davon ausgeht, dass das Ziel in der Nähe ist oder
- wenn man eine geeignete Heuristik (oder Annäherung) hat um den Abstand zum Ziel abzuschätzen?

Außerdem betrachten wir naheliegende Verallgemeinerungen des Problems.

UNIVERSITÄT BONN

Greedyverfahren

Ein Algorithmus wird als **Greedyverfahren** bezeichnet, wenn die iterative Wahl des nächsten zu untersuchenden Elements von der aktuell, lokal besten Wahl abhängt.

Wir betrachten hier mehrere Greedyverfahren. Dabei untersuchen wir jeweils einen unerforschten Standort, der erreichbar ist und

- möglichst nahe am Startpunkt ist (wenn man das Ziel in der Nähe des Startpunkts erwartet) oder
- möglichst nahe am Zielpunkt ist (wenn man den Abstand zum Ziel abschätzen kann).

Ausblick: Kürzeste und möglichst kurze Wege

Wir wissen nur dass das Ziel in der Nähe ist

```
/*   PSEUDOCODE   */
void Suche(start, ziel):
  // Wir modellieren den aktuell bekannten Abstand zum Startpunkt
  abstand = ...       // Abbildung mit abstand(standort) = ∞ f.a. Standorte
  abstand(start) = 0; // Der Start hat natürlich den Abstand 0

  // Alle Standorte die wir noch untersuchen wollen nennen wir 'offen'
  // Wir beginnen unsere Suche mit dem Startpunkt.
  offene_standorte = { start };

  // Wir suchen solange das Ziel bis alle offenen Standorte untersucht wurden
  // oder das Ziel gefunden wurde
  while (offene_standorte != ∅):
    // Bestimme aktuell beste Standortwahl
    aktueller_standort = standort in offene_standorte mit min Abstand;
    offene_standorte.remove(aktueller_standort);

    // Ist der ausgewählte Standort das Ziel und wir sind fertig?
    if (aktueller_standort == ziel):
      // Ziel gefunden
      return;

    // Prüfe welche Nachbarstandorte offen sind
    for each nachbar von aktueller_standort:
      vermeindlicher_abstand  = abstand(aktueller_standort)
                              + abstand(aktueller_standort, nachbar);
      if vermeindlicher_abstand < abstand(nachbar):
        // Wir haben einen kürzeren Weg zu dem Nachbar gefunden
        abstand(nachbar) = vermeindlicher_abstand;
        if nachbar not in offene_standorte:
          offene_standorte.add(nachbar);

  // Das Ziel konnte nicht gefunden werden.
  return;
```

LIVEDEMO

Offene Frage:

Nachdem man das Ziel gefunden hat, wie findet man den Weg von Start zum Ziel?

Wegfindung

Um den Weg von Start zum Ziel zu erfassen reicht folgende Idee aus. Der Startpunkt weiß offenbar wie er sich selbst erreicht. Jeden anderen Standpunkt hat man im Schleifendurchlauf von einem anderen Standort aus erreicht der (zu dem Zeitpunkt) den geringsten Abstand zum Startknoten hatte.

Wenn wir uns an jedem Standort merken **wer unser bester Vorgänger** war können wir so den Rückweg vom Ziel zum Start konstruieren. Also haben wir so auch den Weg vom Start zum Ziel.

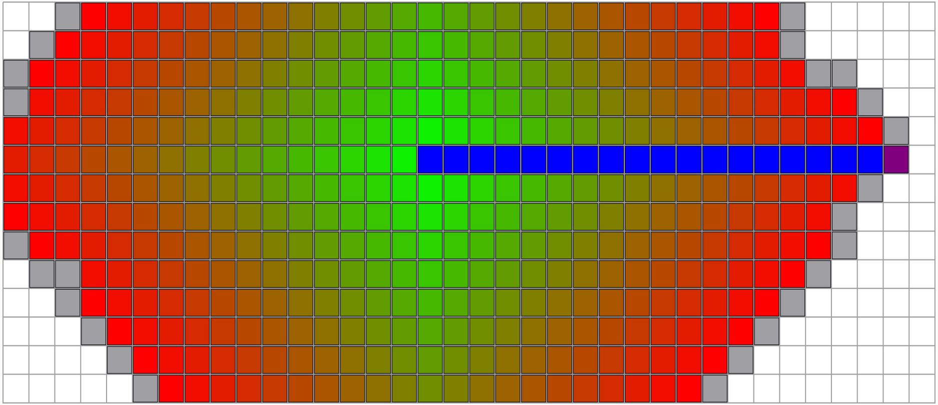
```
Pfad Suche(start, ziel):    /*   PSEUDOCODE   */
  // Wir modellieren den aktuell bekannten Abstand zum Startpunkt
  abstand = ...       // Abbildung mit abstand(standort) = ∞ f.a. Standorte
  abstand(start) = 0; // Der Start hat natürlich den Abstand 0
  vorgänger = ...     // Abbildung mit vorgänger(standort) = ? f.a. Standorte

  // Alle Standorte die wir noch untersuchen wollen nennen wir 'offen'
  // Wir beginnen unsere Suche mit dem Startpunkt.
  offene_standorte = { start };

  // Wir suchen solange das Ziel bis alle offenen Standorte untersucht wurden
  // oder das Ziel gefunden wurde
  while (offene_standorte != ∅):
    // Bestimme aktuell beste Standortwahl
    aktueller_standort = standort in offene_standorte mit min Abstand;
    offene_standorte.remove(aktueller_standort);

    // Ist der ausgewählte Standort das Ziel und wir sind fertig?
    if (aktueller_standort == ziel):
      // Ziel gefunden
      return RekonstruierePfad(vorgänger, start, ziel);

    // Prüfe welche Nachbarstandorte offen sind
    for each nachbar von aktueller_standort:
      vermeindlicher_abstand  = abstand(aktueller_standort)
                              + abstand(aktueller_standort, nachbar);
      if vermeindlicher_abstand < abstand(nachbar):
        // Wir haben einen kürzeren Weg zu dem Nachbar gefunden
        vorgänger(nachbar) = aktueller_standort;
        abstand(nachbar)   = vermeindlicher_abstand;
        if nachbar not in offene_standorte:
          offene_standorte.add(nachbar);

  // Das Ziel konnte nicht gefunden werden.
  return ∅;
```

Fortsetzung

```
/*   PSEUDOCODE   */
Pfad RekonstruierePfad(vorgänger, start, ziel):
  if (start != ziel):
    Teilpfad = RekonstruierePfad(vorgänger, start, vorgänger(ziel));
    return Teilpfad + " -> ziel";
  else:
    return "start";
```

LIVEDEMO

Einschätzung



In der Livedemo haben wir erkannt, dass wir mit dieser Herangehensweise leider viel Zeit benötigen, um einen kürzesten Weg zu erhalten (falls das Ziel weiter weg ist).

Haben Sie Fragen?

Zwischenfazit

Wenn wir nur ausnutzen, dass das Ziel in der Nähe ist, dann finden wir einen kürzesten Weg mit unerwünscht hohem Zeitaufwand.

Offene Fragen

Warum ist das gezeigte Verfahren ein Algorithmus?
(Welches Problem wollen wir wie gut lösen?)

Welche Laufzeit hat der Algorithmus?

Finden wir das Ziel schneller, wenn wir den Abstand zum Ziel gut schätzen können?

Algorithmus Skizze

Input: Ein 2-dimensionales Array mit einem Startpunkt, einem Zielpunkt sowie unzugänglichen und freien Feldern.
Output: Ein kürzester Weg vom Startpunkt zum Ziel (oder die Information dass so ein Weg nicht existiert).
Das Verfahren löst das Problem:

- **Das Verfahren terminiert**: Pro Iteration werden Abstände verringert oder die Menge der offenen Standorte verringert.
- **Invariante**: Pro Durchlauf gibt die Abbildung `vorgänger` den kürzesten Weg an (implizit im Teilraum der besten Standorte).
- **Invariante**: Am Ende jedes Durchlaufs gibt die Menge `offen` die Standorte an,
    - die direkt an die bereits besuchten Standorte grenzen und
    - mindestens so weit entfernt vom Start sind wie die besuchten Standorte.

Laufzeitanalyse

Um eine Laufzeitanalyse durchzuführen, müssen wir verstehen, wie teuer die Operationen auf den Variablen `abstand`, `vermeindlicher_abstand`, `offene_standorte` und `vorgänger` sind.

Die Variablen `abstand`, `vermeindlicher_abstand` und `vorgänger` sind auf den ersten Blick Abbildungen und `offene_standorte` ist eine Menge. Wir verstehen in den kommenden Vorlesungen wie solche und ähnliche Datentypen modelliert werden und wie teuer die genannten Operationen sind.

Zwischenfazit

Wir haben skizziert, warum das gezeigte Verfahren ein Algorithmus ist

Um die Laufzeit analysieren zu können, müssen wir erst mehr über die Umsetzung von Abbildungen und Mengen lernen

Wir fragen uns weiterhin

Finden wir das Ziel schneller, wenn wir den Abstand zum Ziel gut schätzen können?

Ausblick: Kürzeste und möglichst kurze Wege

Wir können den Abstand zum Ziel abschätzen

Offene Frage

Finden wir das Ziel schneller wenn wir den Abstand zum Ziel gut schätzen können?

Motivation

Motivation: Uns ist eine „geeignete Heuristik“ gegeben um den Abstand zum Ziel abzuschätzen.

Beispiel: Die Koordinaten jedes Standorts und des Ziels ist bekannt (GPS). Dann ist die Heuristik durch die Länge der Luftlinie (vom Standort zum Ziel) gegeben.

Formal modellieren wir die Heuristik als eine Abbildung $h\colon \{Standorte\} \to \mathbb{R}$. Hier lassen wir zunächst auch negative Abstandsschätzungen zu. Eine ausgiebige Diskussion von gewünschten Eigenschaften von Heuristiken führt hier zu weit.

Herleitung I

Um noch schneller zu einem möglichst kurzen Weg zu gelangen ändern wir unser Suchverfahren ab.

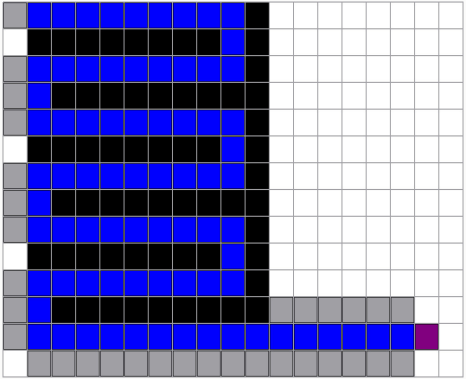
Erste Idee:

- Wir nutzen die Heuristik um den bestgeeignetsten Standort zu wählen.
- Da wir den kürzesten Weg finden wollen, nutzen wir weiterhin die Länge der bereits zurückgelegten Wege um offenen Standorte zu bestimmen.

```
Pfad Suche(start, ziel, heuristik):    /*   PSEUDOCODE   */
  // Wir modellieren den aktuell bekannten Abstand zum Startpunkt
  abstand = ...       // Abbildung mit abstand(start) = 0 und abstand(standort) = ∞ f.a. Standorte
  vorgänger = ...     // Abbildung mit vorgänger(standort) = ? f.a. Standorte

  // Alle Standorte die wir noch untersuchen wollen nennen wir 'offen'
  // Wir beginnen unsere Suche mit dem Startpunkt.
  offene_standorte = { start };

  // Wir suchen solange das Ziel bis alle offenen Standorte untersucht wurden
  // oder das Ziel gefunden wurde
  while (offene_standorte != ∅):
    // Bestimme aktuell beste Standortwahl
    aktueller_standort = standort in offene_standorte mit min Heuristik;
    offene_standorte.remove(aktueller_standort);

    // Ist der ausgewählte Standort das Ziel und wir sind fertig?
    if (aktueller_standort == ziel):
      // Ziel gefunden
      return RekonstruierePfad(vorgänger, start, ziel);

  // Prüfe welche Nachbarstandorte offen sind
  for each nachbar von aktueller_standort:
    vermeindlicher_abstand = abstand(aktueller_standort)
                           + abstand(aktueller_standort, nachbar);
  if vermeindlicher_abstand < abstand(nachbar):
    // Wir haben einen kürzeren Weg zu dem Nachbar gefunden
    vorgänger(nachbar) = aktueller_standort;
    abstand(nachbar)   = vermeindlicher_abstand;
    if nachbar not in offene_standorte:
      offene_standorte.add(nachbar);

  // Das Ziel konnte nicht gefunden werden.
  return ∅;
```

LIVEDEMO

Einschätzung



In der Livedemo haben wir erkannt, dass wir mit dieser Herangehensweise leider beliebig schlechte Annäherungen an einen kürzesten Weg erhalten. Genauer haben wir gesehen, dass wir für jede Konstante k ein Beispiel konstruieren können, bei dem der gefundene Weg mindestens k mal länger ist als der kürzeste Weg.

Nur die Heuristik zu verwenden ist also nicht geeignetet um möglichst schnell einen möglichst kurzen Weg zu finden.

Zwischenfazit

Wenn wir nur ausnutzen, dass wir den Abstand zum Ziel abschätzen können, dann finden wir schnell einen Weg, der unerwünscht lang sein kann.

Ausblick: Kürzeste und möglichst kurze Wege

Kombination der beiden Ansätze

Herleitung II

Um möglichst schnell zu einem möglichst kurzen Weg zu gelangen, ändern wir unser Suchverfahren ab.

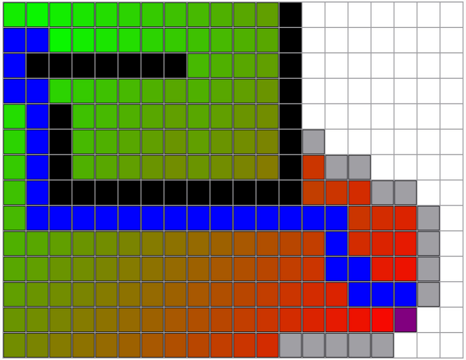
Nächte Idee:

- Wir erhalten einen Schätzung der Gesamtlänge des Wegs, wenn wir die Länge des bereits zurückgelegten Wegs mit der geschätzten Länge des noch zu laufenden Wegs verbinden.
- Da wir den kürzesten Weg finden wollen, nutzen wir weiterhin die Länge der bereits zurückgelegten Wege um offenen Standorte zu bestimmen.

```
Pfad Suche(start, ziel, heuristik):    /*   PSEUDOCODE   */
  // Wir modellieren den aktuell bekannten Abstand zum Startpunkt
  abstand = ...      // Abbildung mit abstand(start) = 0 und abstand(standort) = ∞ f.a. anderen Standorte
  gesch_länge = ...  // Abbildung mit gesch_länge = abstand + heuristik
  vorgänger = ...    // Abbildung mit vorgänger(standort) = ? f.a. Standorte

  // Alle Standorte die wir noch untersuchen wollen nennen wir 'offen'
  // Wir beginnen unsere Suche mit dem Startpunkt.
  offene_standorte = { start };

  // Wir suchen solange das Ziel bis alle offenen Standorte untersucht wurden
  // oder das Ziel gefunden wurde
  while (offene_standorte != ∅):
    // Bestimme aktuell beste Standortwahl
    aktueller_standort = standort in offene_standorte mit min gesch_länge;
    offene_standorte.remove(aktueller_standort);

    // Ist der ausgewählte Standort das Ziel und wir sind fertig?
    ...

    // Prüfe welche Nachbarstandorte offen sind
    for each nachbar von aktueller_standort:
      vermeindlicher_abstand = abstand(aktueller_standort)
                             + abstand(aktueller_standort, nachbar);
      if vermeindlicher_abstand < abstand(nachbar):
        // Wir haben einen kürzeren Weg zu dem Nachbar gefunden
        vorgänger(nachbar) = aktueller_standort;
        abstand(nachbar)   = vermeindlicher_abstand;
        gesch_länge(nachbar) = abstand(nachbar) + heuristik(nachbar);
        if nachbar not in offene_standorte:
          offene_standorte.add(nachbar);

    // Das Ziel konnte nicht gefunden werden.
    return ∅;
```

LIVEDEMO

Einschätzung



In der Livedemo haben wir erkannt, dass wir mit dieser Herangehensweise einen kürzesten Weg erhalten. Allerdings liefert die Heuristik auf dem „freien Feld“ die exakte Länge des kürzesten Wegs. Da es sehr viele kürzeste Wege zum Ziel gibt und wir keinen von diesen bevorzugen, erhalten wir eine unerwünscht hohe Laufzeit.

Zwischenfazit

Wenn wir nur zu gleichen Teilen ausnutzen, dass das Ziel in der Nähe ist und dass wir den Abstand zum Ziel abschätzen können, dann finden wir recht schnell den kürzsten Weg.

Ausblick: Kürzeste und möglichst kurze Wege

Der A*-Algorithmus

Ziel

Was sind naheliegende Verallgemeinerungen?

Diese Verallgemeinerung führen zum A*-ALGORITHMUS

Verallgemeinerung auf Graphen

Für die oben beschriebenen Verfahren ist es unerheblich, dass wir eine zweidimensionale Karte übergeben haben. Es ist nur wichtig, dass wir „Standorte“ und „Nachbarn“ haben. Eine offensichtliche Verallgemeinerung ist es Graphen zu verwenden. Hierbei entsprechen die Standorte den Knoten und die Kanten den direkten Verbindungen zwischen zwei Standorten. Außerdem können wir auf Graphen Kantengewichte einführen, um den Abstand zwischen zwei benachbarten Knoten zu definieren.

UNIVERSITÄT BONN

Verallgemeinerung der Heuristik

Desweiteren ist es mögliche die Heuristik als dynamische Funktion definieren, die den aktuellen Zustand des Algorithmus mit einbezieht. Das heißt, $h(v)$ hängt nicht nur vom Knoten v ab, sondern auch vom aktuellen Ausführungszustand des Verfahrens.

Verallgemeinerung durch Gewichtung

In der Berechnung des erwarteten Abstands gewichten wir den zurückgelegtem Weg und Heuristik. Wir erhalten durch eine Gewichtung $0 \leq \omega \leq 1$ eine kontinuierliche Auswahl zwischen den oben besprochenen Ansätzen:

$$\text{Auswahl} = \underset{v \in \mathit{offen}}{\text{argmin}} \quad \omega \cdot \text{ZurückgelegterWeg}(v) + (1 - \omega) \cdot \text{Heuristik}(v)$$

Die Gewichtung $\omega = 1$ liefert den erste Ansatz ohne Heuristik und $\omega = 0$ liefert den Ansatz, welcher nur die Heuristik nutzt, um den aktuell besten Standort auszuwählen.

Gewichtung $\omega = \frac{1}{2}$ liefert den Ansatz, bei dem zurückgelegter Weg und Heuristik gleichberechtigt Einfluss nehmen. Die Gewichtung $0 < \omega < \frac{1}{2}$ liefert einen Ansatz, bei dem die Heuristik stärkeren Einfluss nimmt. Zum Beispiel wird mit $\omega = \frac{1}{3}$ die Heuristik doppelt so stark gewichtet.

Haben Sie Fragen?

A*-Algorithmus

Durch die oben genannten Verallgemeinerungen erhalten wir den **A*-Algorithmus** zum Finden von kürzeren Wegen.

Als **Input** erhält der Algorithmus einen Graphen mit gewichteten Kanten, zusammen mit gewähltem Start- und Zielknoten, sowie eine Heuristik und eine Auswahlgewichtung $0 \leq \omega \leq 1$.

Der **Output** des Algorithmus ist ein kürzester Weg falls $\omega \geq \frac{1}{2}$ gilt (und so ein Weg existiert und falls die Heuristik „geeignet ist“). Andernfalls liefert der Algorithmus eine approximative Lösung (falls mind. ein Weg existiert).

```
Pfad AStar(graph, start, ziel, heuristik, ausw_gew):
  // Wir modellieren den aktuell bekannten Abstand zum Startpunkt
  abstand = ...      // Abbildung mit abstand(start) = 0 und abstand(standort) = ∞ f.a. anderen Standorte
  gesch_länge = ...  // Abbildung mit gesch_länge = ausw_gew*abstand + (1-ausw_gew)*heuristik
  vorgänger = ...    // Abbildung mit vorgänger(standort) = ? f.a. Standorte

  // Alle Standorte die wir noch untersuchen wollen nennen wir 'offen'
  // Wir beginnen unsere Suche mit dem Startpunkt.
  offene_standorte = { start };

  // Wir suchen solange das Ziel bis alle offenen Standorte untersucht wurden
  // oder das Ziel gefunden wurde
  while (offene_standorte != ∅):
    // Bestimme aktuell beste Standortwahl
    aktueller_standort = standort in offene_standorte mit min gesch_länge;
    offene_standorte.remove(aktueller_standort);

    // Ist der ausgewählte Standort das Ziel und wir sind fertig?
    if (aktueller_standort == ziel) { return RekonstruierePfad(vorgänger, start, ziel); }

    // Prüfe welche Nachbarstandorte offen sind
    for each nachbar von aktueller_standort:
      vermeindlicher_abstand = abstand(aktueller_standort)
                             + abstand(aktueller_standort, nachbar);
      if vermeindlicher_abstand < abstand(nachbar):
        // Wir haben einen kürzeren Weg zu dem Nachbar gefunden
        vorgänger(nachbar) = aktueller_standort;
        abstand(nachbar)   = vermeindlicher_abstand;
        gesch_länge(nachbar) = ausw_gew*abstand(nachbar) + (1-ausw_gew)*heuristik(nachbar);
        if nachbar not in offene_standorte:
          offene_standorte.add(nachbar);

  // Das Ziel konnte nicht gefunden werden.
  return ∅;
```

Spezialfälle

Der A*-Algorithmus besitzt folgende, wohlbekannte Spezialfälle:

- Für $\omega \neq 0$, identische Kantengewichte und $h = 0 \rightsquigarrow$(gezielte) Breitensuche.
- Für $\omega < 1$, identische Kantengewichte und die dynamisch Heuristik $h =$ Anz. abgeschlossener Iterationen $\rightsquigarrow$(gezielte) Tiefensuche.
- Für $\omega = 1$ und nicht-negative Kantengewichte erhalten wir **Dijkstras Algorithmus**.

Exakte und approximative Algorithmen

Oben haben wir die Gewichtung wie folgt angepasst.

$$\text{Auswahl} = \underset{v \in \mathit{offen}}{\text{argmin}} \quad \omega \cdot \text{ZurückgelegterWeg}(v) + (1 - \omega) \cdot \text{Heuristik}(v)$$

Die Laufzeit des Verfahrens und die **Approximationsqualität** des kürzesten Wegs hängt damit von ω und der Heuristik ab.

Für $\omega = 1$ erhalten wir einen exakten Algorithmus der immer einen kürzesten Weg liefert. Für $\omega = \frac{1}{2}$ und eine „geeignete Heuristik" erhalten wir einen exakten Algorithmus der mindestens genauso schnell einen kürzesten Weg liefert.

Für $\omega < \frac{1}{2}$ erhalten wir einen approximativen Algorithmus. Laufzeit und Approximationsqualität hängen sehr stark von der Heuristik und der Gestalt des Graphen ab.

Zusammenfassung

Sie haben den A*-ALGORITHMUS kennen gelernt

Je nach Konfiguration liefert der A*-ALGORITHMUS schnell einen kürzsten Weg oder möglichst schnell einen möglichst kurzen Weg

Haben Sie Fragen?

Empfehlung

Wiederholen Sie aus der Vorlesung IMPERATIVE PROGRAMMIERUNG TEIL 3 die Inhalte Stackframes und lokale Variablen im Speicher & Der Heapspeicher

Wiederholen Sie die Inhalte der Vorlesung IMPERATIVE PROGRAMMIERUNG TEIL 5 inklusive der Verwaltung von Membervariablen in Stackframes