Tutorium #4

Softwareentwicklung Praktikum

Ablauf des Tutoriums

- Templates
- Profiling
- Zeitplan
- Aufbaubeispiel
- Live-Demo
- Dijkstra

Templates

Templates

- Erlauben es, generische Funktionen zu schreiben
- Werden in der STL ausgiebig verwendet
- Es gibt Funktions- und Klassentemplates
- Deklarations-Syntax:

```C++
template <typename identifier> function_declaration;
```

- Aufrufs-Syntax:

```C++
function<argument_type>(arguments);
```

bzw.

```C++
classname<class_type>(constructor_values);
```

Templates - Beispiel (Funktionstemplate)

C++
```
template <typename Type>
Type max(Type a, Type b)
{
    if(a > b) return a;
    else return b;
}
```

C++
```
int main()
{
  // ruft max<int> auf (wird an Argumenten erkannt)
  std::cout << max(3, 7) << std::endl;
  // ruft max<double> (wird an Argumenten erkannt)
  std::cout << max(3.0, 7.0) << std::endl;
  // Argumente nicht eindeutig, Typ muss angegeben werden (<double>)
  std::cout << max<double>(3, 7.0) << std::endl;
  return 0;
}
```

Templates - Beispiel (Klassentemplate)

C++
```
template <typename T>
class Math {
public:
  T getMin(T x1, T x2) {
    if(x1 < x2) return x1;
    else return x2;
  }

  T getMax(T x1, T x2) {
    if(x1 > x2) return x1;
    else return x2;
  }
};
```

C++
```
int main() {
  Math<int> int_math;
  Math<std::string> str_math;

  std::cout << "max(2, 4): " << int_math.getMax(2, 4) << std::endl;
  std::cout << "min(\"abc\", \"bcd\"): "
    << str_math.getMin("abc", "bcd") << std::endl;
}
```

Templates - Beispiel (Klassentemplate)

C++

```
template <typename T>
struct LinkedList {
  LinkedListElement<T>* elements_;

  LinkedList();
  void insert(int index, T object);
};

template <typename T>
struct LinkedListElement {
  T object_;
  LinkedListElement<T>* next_;

  LinkedListElement(T object);
}
```

Templates - Beispiel (Klassentemplate)

C++

```
void LinkedList::insert(int index, T object)
{
  if(elements_ == null)
  {
    elements_ = new LinkedListElement(object);
    return;
  }
  LinkedListElement* previous = elements_;
  for(int i = 0; i < index && previous != null; i++)
  {
    previous = previous->next_;
  }

  LinkedListElement* next = previous->next_;
  previous->next_ = new LinkedListElement(object);

  previous->next_->next_ = next;
}
```

Profiling

// Premature optimization is the root of all evil (Donald Knuth)

Optimieren

- Optimierungen können enorm viel Geschwindigkeit herausholen
- Wichtig ist es, die richtigen Dinge zu optimieren
- Durch Profiling findet man heraus, was sich auszahlt zu optimieren
- Für Profiling benötigt man einen CPU Profiler:
  - Googles `gperf` (https://github.com/gperftools/gperftools)
  - oder Valgrinds `callgrind` zusammen mit KCachegrind (https://kcachegrind.github.io/)

gperf

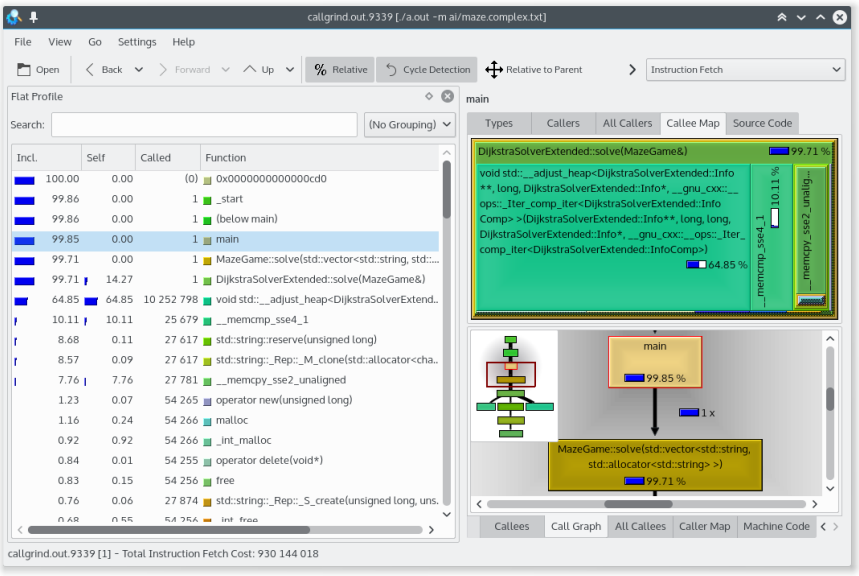
- Unter Ubuntu im Paketmanager als `google-perftools`
- C++ Code mit Compiler Flag `-pg` kompilieren und starten
- Mit `gprof [binary] gmon.out` analysieren

C++

```
 time   seconds   seconds    calls   name
26.96      0.28     0.28 10252798   void std::__adjust_heap
23.53      0.52     0.24 61948292   std::remove_reference
11.77      0.64     0.12 10252798   void std::__push_heap
 6.86      0.71     0.07 31071480   DijkstraSolverExtended::InfoComp::operator()
 5.88      0.77     0.06 20338780   bool __gnu_cxx::__ops::_Iter_comp_iter<DijkstraSolverExtended::InfoComp>::operator()
 4.90 0.82 0.05 10591122 bool __gnu_cxx::__ops::_Iter_comp_val<DijkstraSolverExtended::InfoComp>::operator()
 4.90 0.87 0.05 12792 void std::__make_heap
 3.43 0.90 0.04 10265590 __gnu_cxx::__ops::_Iter_comp_val<DijkstraSolverExtended::InfoComp>
 2.45 0.93 0.03 25600 __gnu_cxx::__normal_iterator
 1.96 0.95 0.02 12796 void __gnu_cxx::new_allocator<DijkstraSolverExtended::Info*>::destroy
 1.96 0.97 0.02 108 void std::vector::_M_emplace_back_aux
 1.47 0.98 0.02 25588 __gnu_cxx::__ops::_Iter_comp_iter
 0.98 0.99 0.01 10278386 __gnu_cxx::__ops::_Iter_comp_val
 0.98 1.00 0.01 12796 std::enable_if
 0.98 1.01 0.01 1 MazeGame::solve(std::vector<std::string, std::allocator<std::string> >)
 0.98 1.02 0.01 MazeGame::checkWinning()
```

Valgrind

- Programm mit `valgrind --tool=callgrind --dump-instr=yes [binary]` starten
- Mit `kcachegrind callgrind.out.*` analysieren



Aufbaubeispiel

Zeitplan

- Ausgabe Aufbaubeispiel: **19.05.16**
- Abgabe Aufbaubeispiel: **09.06.16**
- Abgabegespräche: ab dem **20.06.16**

Bewertung

- Basisbeispiel
  - Grundanforderung Basisbeispiel: max. 5 Punkte
  - Abgabegespräch: max. 3 Punkte
  - Aufbaubeispiel: optional, max. 2 Punkte, mindestens 2 Punkte auf Basisbsp. erforderlich
- Vorlesungsteil Teil (= Klausur)

Aufbaubeispiel

- neue **Feldtypen**:
  - Counter
  - Hole
- neue **Befehle**:
  - whoami
  - solve
  - show more nopath

Solve

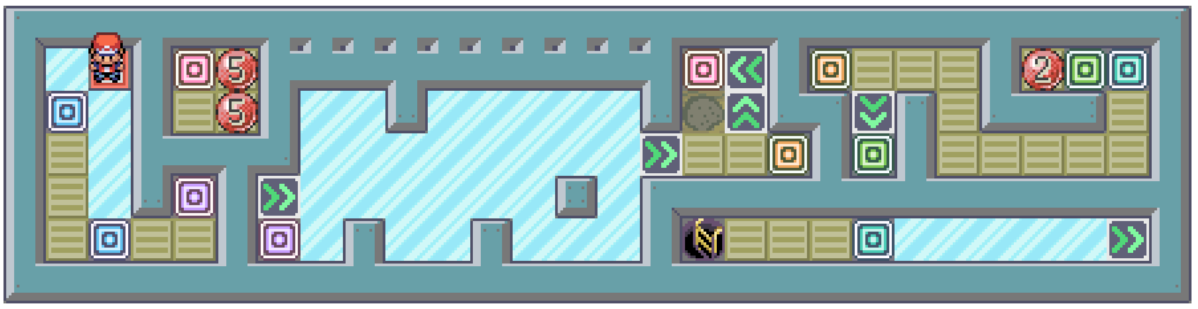
- KI:
  - Pfad vom aktuellen Standpunkt zum Ziel berechnen
  - Mit den wenigsten Schritten
  - Fastmove ausführen
  - Speichern
- Nur Felder aus Basisbeispiel (bei Highscore werden aber alle Felder getestet)
- Mindestanforderungen:
  - Lösen des Maze
  - Beachtung der max. Steps
- Restriktionen:
  - 15 Sekunden
  - 400 MB Arbeitsspeicher

Live-Demo

Map file

Choose File map1.txt Enable autoupdate

Steps: 0



Highscore

Rank	Group	Level	Steps	Time	Last Update	Errors
1	GSALT (5832)	4	8	9.013	2016-05-05 21:05:40	-
2	Workers (1234)	4	22	8.013	2016-05-05 21:06:48	-
3	MemoryBuster (1212)	3	7	7.015	2016-05-05 21:05:12	out of memory
4	JonSnow (9101)	2	7	7.014	2016-05-05 21:04:47	file not found
5	REC (9393)	1	12	8.014	2016-05-05 21:07:01	return code
6	Steppers (3232)	0	6	7.013	2016-05-05 21:05:58	maximum steps
7	NE (555)	0	19	4.015	2016-05-05 21:07:13	no win message
8	- (5678)	-1	-	0	2016-05-05 21:05:50	timeout
9	- (1277)	-1	-	0	2016-05-05 21:06:16	compile error
10	- (9999)	-1	-	0	2016-05-05 21:06:18	whoami error

https://server.nasahl.net/sep/

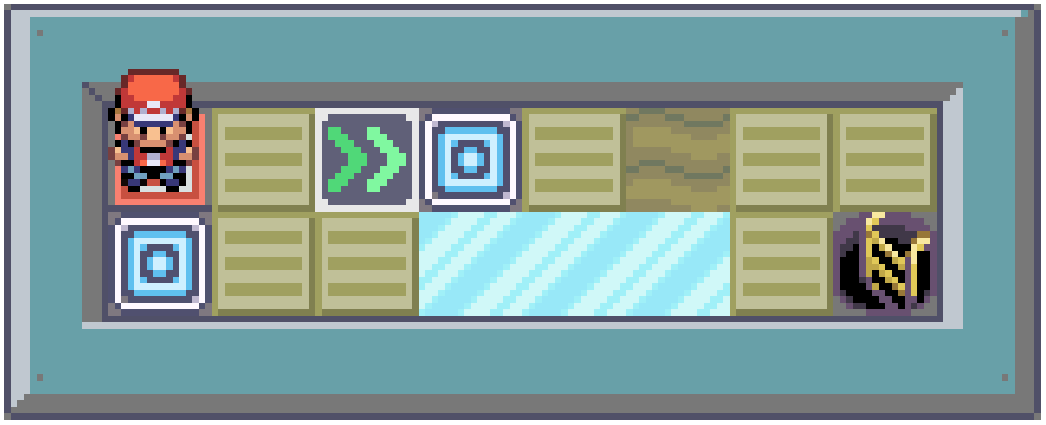
Wegsuche

// Damn it, how will I ever get out of this labyrinth? (Simón Bolívar)

Labyrinth zu Graphen

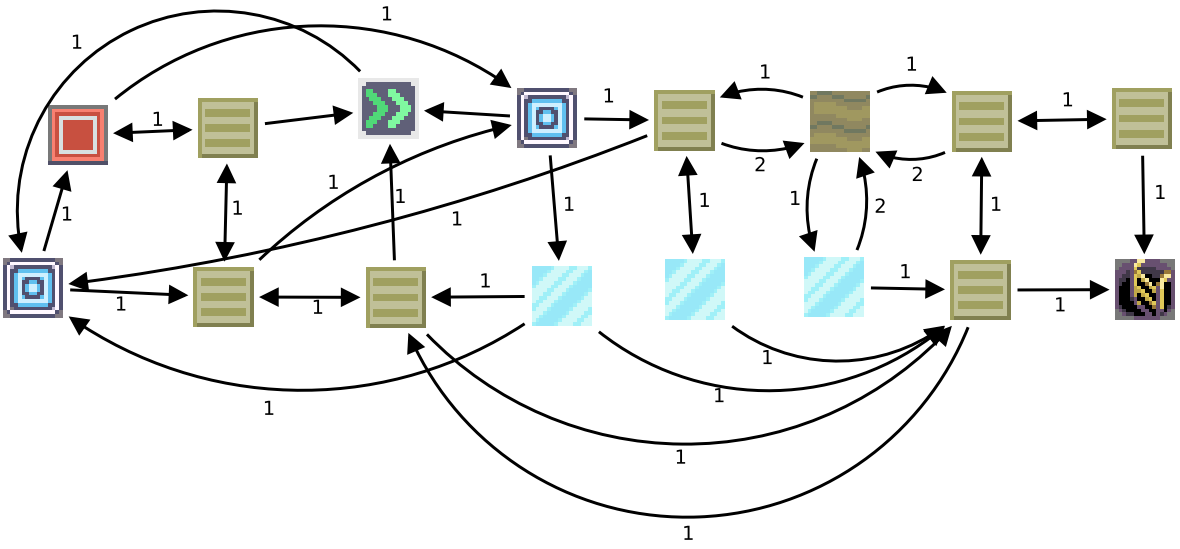
- Labyrinthe gehören zu den klassischen Problemen der Wegsuche in Graphen
- Prominente Beispiele sind Navigationssoftware oder KI in Computerspielen
- Es gibt effiziente Algorithmen zur Lösung
- Wir können jedes Labyrinth in einen Graphen umwandeln
  - Jedes Feld ist ein **Knoten**
  - Wenn man von einem Feld auf ein anderes gelangt, werden sie durch eine **Kante** verbunden
  - Wichtig: Kanten sind **gerichtet**, da es z.B. Einbahnen gibt
  - Das **Gewicht** der Kante ist die Anzahl der Schritte, die man für die Verbindung braucht (durch Bonus- oder Treibsand-Felder)

Beispiel: Labyrinth zu Graphen



- Mauern sind keine Knoten im Graph (da man sie nicht "besuchen" kann)
- Wir bilden alle möglichen Bewegungen ab (und die Schritte dazu)

Beispiel: Labyrinth zu Graphen



Erster Versuch: Breitensuche

- Breitensuche (Breadth-First-Search, BFS) ist eine Möglichkeiten, den gesamten Graphen zu durchsuchen
- Wir beginnen beim Start, und suchen das Ausgangsfeld
- **Idee:**
  - Wir bauen eine Warteschlange, in der wir die nächsten zu untersuchenden Knoten speichern
  - Von jedem Knoten den wir besuchen, legen wir die Nachbar-Knoten in die Warteschlange
  - Wir betrachten (und entfernen) immer den ersten Knoten der Warteschlange, bis wir das Ziel gefunden haben

Erster Versuch: Breitensuche

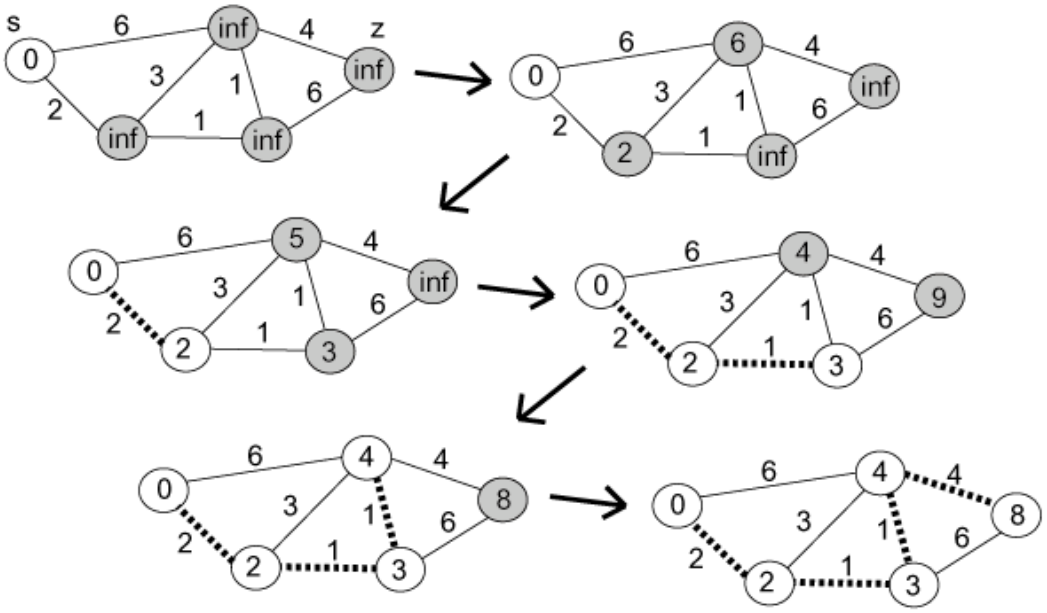
- **Pro**
  - **Einfach** zu implementieren
  - Findet **immer** einen Weg
- **Contra**
  - **Langsam**, da Felder immer wieder untersucht werden
  - Umso mehr Wege es gibt, umso länger braucht die Suche
  - Findet **irgendeinen** Pfad, nicht unbedingt den kürzesten Pfad
- Geht das noch besser?
  - Besuchte Knoten markieren, und nicht erneut besuchen
  - Vor dem Hinzufügen in die Warteschlange schauen ob man schon das Ziel erreicht hat

Schnelle Wegsuche: Dijkstra

- Dijkstra-Algorithmus findet den kürzesten Weg in einem Graph
- **Idee:**
  - Alle Knoten haben am Anfang "Distanz" = ∞ (Startknoten hat "Distanz" = 0) und "Vorgänger" = `NULL`
  - Solange es **unbesuchte** Knoten gibt, wähle den mit **niedrigster Distanz**
    - Markiere Knoten als **besucht**
    - Für alle **unbesuchten** Nachbarn, berechne die Summe des jeweiligen Kantengewichtes und der aktuellen Distanz
    - Ist der Wert **kleiner** als die Distanz des Knotens, **aktualisiere** sie und setze aktuellen Knoten als Vorgänger.

Schnelle Wegsuche: Dijkstra

- Dijkstra-Algorithmus findet den kürzesten Weg in einem Graph



Schnelle Wegsuche: Dijkstra

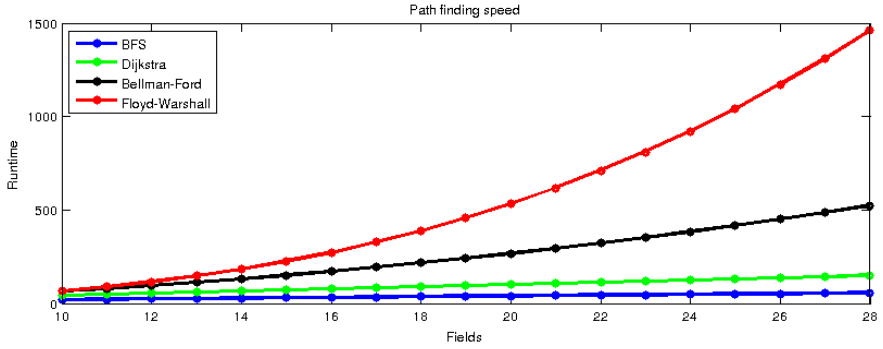
- **Pro**
  - Relativ **einfach** zu implementieren
  - Findet den **kürzesten** Weg
  - Sehr **schnell**
- **Contra**
  - Kann **nicht** mit **Bonus-Feldern** umgehen
- Warum kann Dijkstra Level mit Bonus nicht lösen?
  - Bonus Felder sind **Endlosschleifen**, wenn der Graph nicht aktualisiert wird
  - Den Graph zu aktualisieren während der Algorithmus läuft ist **kompliziert**
  - Dijkstra kann allgemein nicht mit **negativen Kantengewichten** umgehen

Alternativen zu Dijkstra

- Algorithmen, die negative Kanten unterstützen
  - **Floyd-Warshall**
    - berechnet die kürzesten Pfade zwischen **allen** Knoten
    - dadurch sehr langsam
  - **Bellman-Ford**
    - wie Dijkstra, kann allerdings negative Kanten
    - viel langsamer als Dijkstra

Benchmark (25x25 Labyrinth)

Algorithmus	Zeit	Laufzeit
BFS	0,6ms	$O(E)$
Dijkstra	5ms	$O(E + V * \log(V))$
Bellman-Ford	30ms	$O(V * E)$
Floyd-Warshall	23 278ms	$O(V3)$



Zurück zu Dijkstra

- Können wir Dijkstra ändern, um mit Bonus-Feldern umzugehen?
- **Grundidee**:
  - Problem in Teilprobleme zerlegen
  - Start, Ziel und jedes Bonusfeld ist ein **Wegpunkt**
  - Kürzeste Wege zwischen allen Wegpunkten berechnen
  - Beste Kombination an Wegpunkten wählen

Zurück zu Dijkstra

- **Pro**
  - Funktioniert für Bonusfelder
  - **Schnell** für wenige Bonusfelder
- **Contra**
  - **Langsam**, für viele Bonusfelder
  - **Komplizierter** zu implementieren
  - Braucht **Sonderbehandlung**, wenn Bonusfelder auf einem Weg liegen

Bis zum Abgabegespräch!