Einführung in die Programmiersprache C++

... FÜR FORTGESCHRITTENE ...

Thomas Wiemann
Institut für Informatik
AG Wissensbasierte Systeme

OpenMP (1)

- Was ist OpenMP?
  - Offene Spezifikation für Multi-Processing
  - www.openmp.org – Vorträge, Beispiele, Forum, etc.
  - Besonders gut für SIMD Schemata
- High-level API
  - Präprocessor (compiler) Direktiven (~ 80%)
  - Bibliotheks-Aufrufe (~ 19%)
  - Umgebungsvariablen (~ 1%)
- OpenMP erlaubt:
  - Ein Programm einfach in serielle und parallele Teile zu zerlegen
  - Versteckt das Stack-Management
  - Stellt Synchronisierungs-Konstrukte zur Verfügung
- OpenMP erlaubt nicht:
  - Parallelisierungs Abhängigkeiten zu modellieren
  - Garantien zum speedup zu geben
  - Race Bedingungen zu unterdrücken

OpenMP (2)

- OpenMP benutzt größtenteils `#pragma`-Anweisungen
- Falls ein Compiler sie nicht unterstützt werden sie ignoriert
- Beispiel: Initialisierung einer Tabelle mit OpenMP:

```
#include <cmath>
int main()
{
  const int size = 256;
  double sinTable[size];

  #pragma omp parallel for
  for(int n=0; n<size; ++n)
    sinTable[n] = std::sin(2 * M_PI * n / size);

  // the table is now initialized
}
```

OpenMP (3)

- Sehr einfache Parallelisierung der for-Schleife
- `#pragma omp parallel` erzeugt einen Block, der parallel abgearbeitet wird
- Beispiel:

```
#pragma omp parallel
{
    // Code inside this region runs in parallel.
    printf("Hello!\n");
}
```

- Wie oft wird „Hello!“ ausgegeben?
- Das hängt vom verwendeten Prozessor ab
- Man kann die Anzahl der zu erzeugende Threads mit omp_num_threads() zur Laufzeit festlegen
- Programme, die OpenMP verwenden müssen mit `-fopenmp` übersetzt werden

OpenMP (4)

- Was ist die Ausgabe folgenden Code-Segmentes:

```
#pragma omp for
for(int n=0; n<10; ++n)
{
  printf(" %d", n);
}
printf(".\n");
```

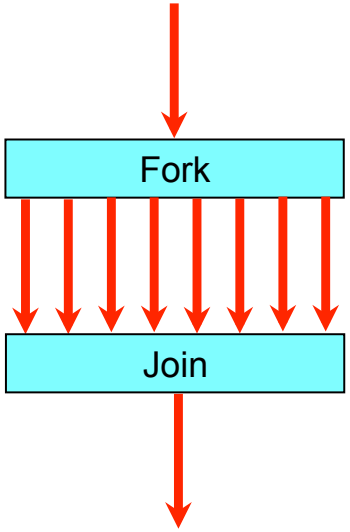
- Für jeden Schleifenindex wird ein eigener Thread angelegt
- Die Indices werden nicht notwendigerweise in geordneter Reihenfolge generiert
- Beispiel-Ausgabe:

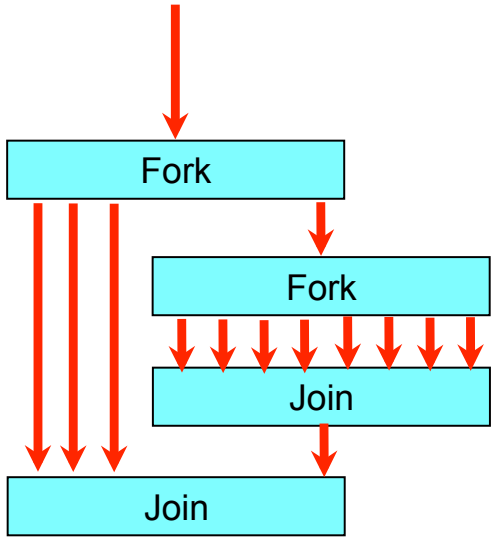
```
% 0 5 6 7 1 8 2 3 4 9.
```

- Daten, auf denen gearbeitet wird müssen unabhängig von einander sein
- SIMD

OpenMP (5)

- Was ist der Unterschied zwischen `parallel`, `parallel for` und `for`?
- Threads, die parallel laufen, nennt man ein Team
- Zu Programmbeginn besteht das Team aus genau einem Thread
- Eine `parallel`-Direktive splittet den Thread in ein neues Team auf
- Danach werden die Teams wieder zu einem Thread vereint





OpenMP (6)

- OpenMP definiert Locks
- `omp_init_lock` initialisiert ein Lock, sperrt aber nichts
- Analog: `omp_destroy_lock`
- `omp_set_lock` blockiert eine Region
- `omp_unset_lock` gibt ein vorheriges Lock wieder frei
- `omp_test_lock` testet, ob ein Datum blockiert ist
- Wie funktioniert Datenaustausch zwischen Threads?
- Man kann Variablen als private oder shared deklarieren:
- int a, b=0;

```
#pragma omp parallel for private(a) shared(b)
for(a=0; a<50; ++a)
{
  #pragma omp atomic
  b += a;
}
```

- Hier hat jeder Thread eine eigene Kopie von a, b wird geteilt

OpenMP (7)

- Synchronisierung von Threads mittels `barrier`
- Beispiel:

```
#pragma omp parallel
{
  /* All threads execute this. */
  SomeCode();

  #pragma omp barrier

  /* All threads execute this, but not before
   * all threads have finished executing
   * SomeCode().
   */
  SomeMoreCode();
}
```

OpenMP (8)

- Verschachtelte Schleifen sind problematisch
- Beispiel:

```
#pragma omp parallel for
 for(int y=0; y<25; ++y)
 {
   #pragma omp parallel for
   for(int x=0; x<80; ++x)
   {
     tick(x,y);
   }
 }
```

- Was passiert?
- Es sind N-Threads gleichzeitig am Laufen
- Das innere `#pragma`... wird ignoriert!
- In OpenMP 3.0 gibt es dafür das Schlüsselwort collapse...

OpenMP (9)

- Workarounds:

```
omp_set_nested(1);
#pragma omp parallel for
for(int y=0; y<25; ++y)
{
  #pragma omp parallel for
  for(int x=0; x<80; ++x)
  {
    tick(x,y);
  }
}
```

- Dieser Code erzeugt N*N Threads
- Wird in der Regel nicht gewünscht
- omp_set_nestet() ist daher per Default deaktiviert

OpenMP (10)

- Cleverste Variante: Versuche die innere Schleife aufzulösen
- Geht oft, wenn auf zweidimensionalen Arrays gerechnet wird:

```
#pragma omp parallel for
 for(int pos=0; pos<(25*80); ++pos)
 {
   int x = pos%80;
   int y = pos/80;
   tick(x,y);
 }
```

- So was geht offensichtlich nicht immer.
- Was dann?

OpenMP (11)

- Ansonsten ist es die beste Idee, nur die innere Schleife zu parallelisieren:

```
for(int y=0; y<25; ++y)
{
  #pragma omp parallel for
  for(int x=0; x<80; ++x)
  {
    tick(x,y);
  }
}
```

- Nur geringfügig langsamer als vorheriges Beispiel
- Meistens die einfachste Variante

Boost - Einführung (2)

- Die Implementierungen der Boost-Bibliotheken ermöglichen oft Sachen, von denen man nicht glaubt, dass es syntaktisch korrekt ist
- Diese „Syntaxdehnung“ macht es oft einfacher zu programmieren
- „A tribool implments 3-state boolean logic“:

```
boost::logic::tribool godExists = detectGod();
if (godExists)
   std::cout << "Good news - there is a God\n";
else if (!godExists)
   std::cout << "Good news - there is no God\n";
else
   std::cout << "There may or may not be a God\n";
```

- Weitere Beispiele folgen...

boost::thread (1)

- Verwendung ähnlich zu `pthreads`
- Beispiel:

```
#include <boost/thread/thread.hpp>
#include <iostream>

void hello()
{
  std::cout <<
    "Hello world, I'm a thread!"
    << std::endl;
}

int main(int argc, char* argv[])
{
  boost::thread thrd(&hello);
  thrd.join();
  return 0;
}
```

boost::thread (2)

- Mutexe:

```
#include <boost/thread/thread.hpp>
#include <boost/thread/mutex.hpp>
#include <iostream>
boost::mutex io_mutex;
struct count
{
  count(int id) : id(id) { }
  void operator()()
  {
    for (int i = 0; i < 10; ++i)
    {
      boost::mutex::scoped_lock
        lock(io_mutex);
      std::cout << id << ": "
        << i << std::endl;
    }
  }
  int id;
};
```

```
int main(int argc, char* argv[])
{
  boost::thread thrd1(count(1));
  boost::thread thrd2(count(2));
  thrd1.join();
  thrd2.join();
  return 0;
}
```

boost::thread (3)

- Boost-Threads erlauben bedingtes Warten
- Wie bekommt man in C++ Initialisierungen (z.B. Konstruktor) Thread-Safe?
- In Boost kann man „once routines“ definieren:

```
#include <boost/thread/thread.hpp>
#include <boost/thread/once.hpp>
#include <iostream>

int i = 0;
boost::once_flag flag =
  BOOST_ONCE_INIT;

void init()
{
  ++i;
}
```

```
void thread()
{
  boost::call_once(&init, flag);
}

int main(int argc, char* argv[])
{
  boost::thread thrd1(&thread);
  boost::thread thrd2(&thread);
  thrd1.join();
  thrd2.join();
  std::cout << i << std::endl;
  return 0;
}
```

boost::thread (4)

- Diverse Erweiterungen sind noch in der Entwicklung
- `boost::read_write_mutex`
  - Erlaubt mehreren Threads eine Variable zu lesen
  - Es darf aber nur einer gleichzeitig Schreibzugriff haben
- `boost::thread_barrier`
  - Erlaubt es einem Satz mehrerer Threads zu synchronisieren
  - Erst wenn alle Threads die Barriere erreicht haben wird der Programmfluss wieder freigegeben
- `boost::thread_pool`
  - Erlauben einen Thread ansynchron abzuarbeiten
  - Keine Synchronisierung notwendig
  - Wie `pthread_detach()`

boost::any (1)

- Normalerweise hat jede Variable einen eigenen Typ
- Hat man beispielsweise eine Variable des Typs „Person“ kann man sicher sein, dass sie eine Intanz einer Person speichert (oder Unterklassen)
- Hilft eine Menge an Laufzeitfehlern zu vermeiden
- Manchmal will man aber Typen unterschiedlichen Typs in einem Container speichern
- Dafür gibt‘s boost::any
- Wrapper-Klasse zum Speichern beliebiger Typen
- Sollte tunlichst vermieden werden
- Ist aber manchmal hilfreich ☺

boost::any (2)

- Beispiel:

```
boost::any a1 = std::string("Moose");
boost::any a2 = 6;
try
{
    // this works, a1 is a string
    std::string v1 = boost::any_cast< std::string >(a1);

    // nope, will throw an exception at runtime
    std::string v2 = boost::any_cast< std::string >(a2);
}
catch ( const boost::bad_any_cast& e )
{
    // tried to any_cast into something that wouldn't go
}
```

boost::any (3)

- Man kann den Typ einer `any`-Variablen mittels `typeid` bestimmen

```
std::string v;
if ( a1.type() == typeid(std::string) )
{
    v = boost::any_cast< std::string >( a1 );
    // this should never throw, since
    // we checked first
}
```

- Jeder Typ in einer `any`-Variablen muss einen Copy-Konstruktor haben
- Der Destruktor darf keine Exceptions werfen
- Manchmal einfacher, als eine eigene Klassenstruktur mit einer Allgemeinen Oberklasse zu entwerfen

boost::assign (1)

- STL-Container mit Daten zu initialisieren ist lästig:

```
vector<string> faceCards;
faceCards.push_back("jack");
faceCards.push_back("queen")
faceCards.push_back("king");
faceCards.push_back("ace");
```

- `boost::assign` hilft hier:

```
vector<string> faceCards;
faceCards += "jack","queen", "king", "ace";
```

- Noch besser:

```
list<int> randomData;
// data will contain 1,2,(8 random numbers),7
randomData += 1, 2, repeat_fun(8, &rand),7;
```

- Es werden noch weitere Container unterstützt

boost::assign (2)

- Beispiel `map`:

```
map<string, string> maoriColors;
insert(maoriColors)("ma", "white")
                   ("whero", "red")
                   ("kakariki", "green")
                   ("kowhai", "yellow" );
```

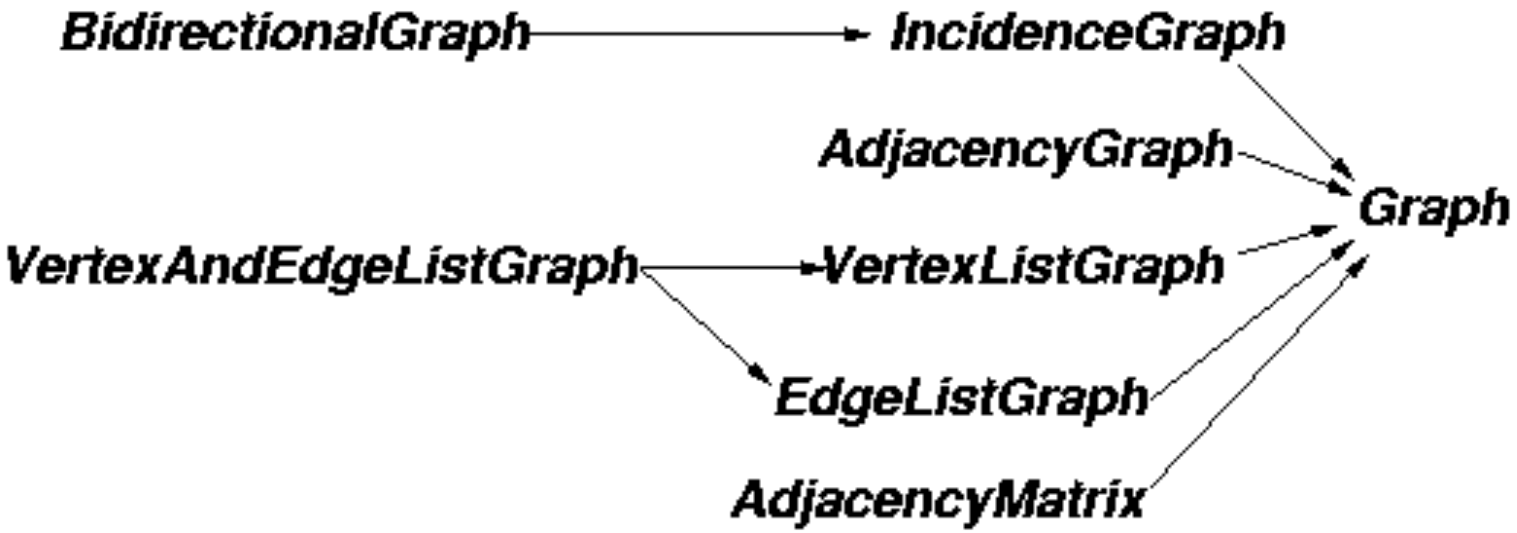
- Wieso geht das?
- Überladen der Operatoren `operator()` und `operator,`
- Funktoren werden zum Einfügen verwendet
- `boost::assign` ist sehr gut dokumentiert
- Man kann sich da ein paar Tricks abschauen
- Man kann `boost::assign` auch für eigene Klassen erweitern

Weitere nützliche Bibliotheken

- Boost Datetime
- Boost Filesystem
- Boost Program Options
- ...

Boost Graph Library (1)

- Ursprünglich Generic Graph Component Library
- Erste wirkliche generische Bibliothek für Graphen-Algorithmen
- Erste Version 1998 von Lie-Quan Lee als Master-Arbeit
- Weitere Entwicklungen bei SGI
- September 2000 Aufnahme in Boost und Umbenennung in BGL
- In der BGL gibt es verschiedene Konzepte für Graphen
- Einige Beispiele:



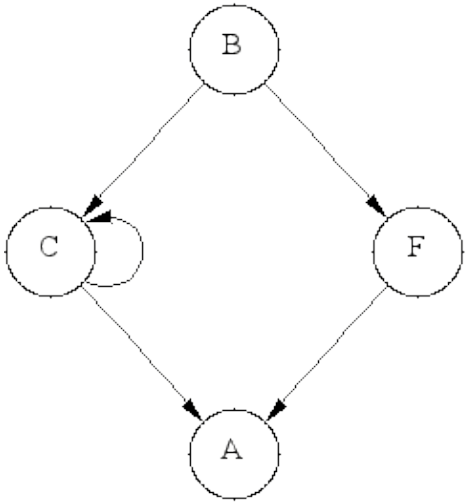
Boost Graph Library (2)

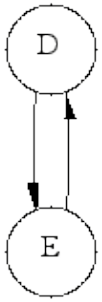
- Visitor-Konzepte werden analog zu Iteratoren und Funktoren eingesetzt, um das Verhalten von Algorithmen zu beeinflussen
- Visitors sind komplexer als Funktoren
- Jeder Algorithmus hat seinen dazugehörigen Visitor, mit dem die Parameter bestimmt werden können (Beispiele später)
- Konzepte:
  - BFS-Visitor (Breitensuche)
  - DFS-Visitor (Tiefensuche)
  - Dijkstra-Visitor (Shortest-Path)
  - Bellman-Ford-Visitor (Shortest Path)
  - AStar-Visitor (Shortest Path)
  - EventVisitor (Algortihmen können Events generieren)
  - PlanarFaceVisitor (Iteration über planare Graphen)
  - TSP-Tour-Visitor (Für TSP-Probleme)

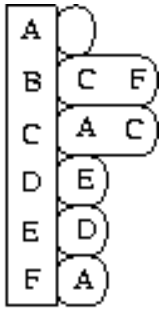
Boost Graph Library (3)

- Graphen Repräsentation

```
adjacency_list<OutEdgeList, VertexList, Directed,
               VertexProperties, EdgeProperties,
               GraphProperties, EdgeList>
```







Boost Graph Library (4)

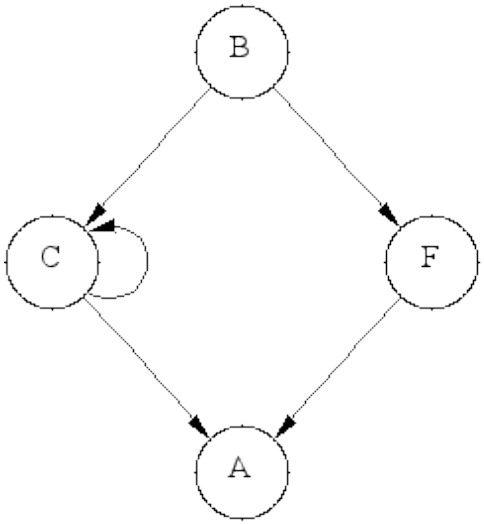
- Die Container, die für `OutEdgeList`, `VertexList` und `EdgeList` verwendet werden, bestimmen die Laufzeit der Algorithmen und den Speicherbedarf
- Unterstützt werden:

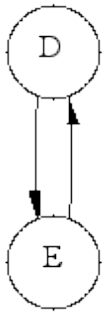
  `vecS` nimmt `std::vector`
  `listS` nimmt `std::list`
  `slistS` nimmt `std::slist`
  `setS` nimmt `std::set`
  `multisetS` nimmt `std::multiset`
  `hash_setS` nimmt `std::hash_set`

- `adjacency_list` modelliert die Konzepte `VertexAndEdgeListGraph`, `MutablePropertyGraph`, `CopyConstructible`, `Assignable` und `Serializable`

Boost Graph Library (5)

- Weitere Graphentypen sind `adjacency_matrix` und `compressed_sparse_row_graph`
- `adjacency_matrix`:





	A	B	C	D	E	F
A	0	0	0	0	0	0
B	0	0	1	0	0	1
C	1	0	1	0	0	0
D	0	0	0	0	1	0
E	0	0	0	1	0	0
F	1	0	0	0	0	0

Boost Graph Library (6)

- `PropertyGraphs` können Eigenschaften an Knoten und Kanten haben
- Die Eigenschaften werden in `PropertyMaps` gespeichert

```
property_map<Graph, edge_weight_t>::type w
   = get(edge_weight, g);
```

- Algorithmen benötigen die `PropertyMaps`
- Beispiel:

```
dijkstra_shortest_paths(g, src,
   distance_map(get(vertex_distance, g)).
   weight_map(get(edge_weight, g)).
   color_map(get(vertex_color, g)).
   vertex_index_map(get(vertex_index, g)));
```

- Oft müssen `PropertyMaps` vorgegeben werden, um z.B. Kantengewichte anzugeben

Six Degrees of Kevin Bacon (1)

- Beispiel aus der Boost-Dokumentation
- Aufgabe: Finde eine Verbindung zwischen einem beliebigen Schauspieler und Kevin Bacon durch eine Reihen von Schauspielern, die gemeinsam in einem Spielfilm aufgetreten sind
- Beispiel:
  - Theodore Hesburgh spielte in „Rudy“ zusammen mit Gerry Becker
  - Gerry Becker spielte zusammen mit Kevin Bacon in „Sleepers“
  - Warum Kevin Bacon?

> *For some reason the three students who invented the game, Mike Ginelli, Craig Fass, and Brian Turtle decided that Kevin Bacon is the center of the entertainment world.*

Six Degrees of Kevin Bacon (2)

- Das ist ein Graphenproblem
- Jeder Schauspieler stellt einen Knoten dar
- Falls zwei Schauspieler gemeinsam in einem Film auftreten, füge eine Kanten zwischen ihnen ein
- Sobald der Graph fertig ist, müssen wir einen Pfad von Kevin Bacon zu jedem anderen Schauspieler finden
- Der Pfad soll minimal sein
- Dijkstra ginge, ist hier aber ein Overkill, da die Kanten keine Gewichte haben
- Wie wärs mit Breitensuche -> OK
- Daten kommen im Format

```
Schauspieler1; Filmname; Schauspieler2
```

- Nun wollen wir das Problem in der BGL modellieren

Six Degrees of Kevin Bacon (3)

- Zunächst müssen wir einen Graphen konstruieren
- Die Relationen zwischen den Schauspielern sind symmetrisch
- Es reicht also ein ungerichteter Graph
- Graph wird nicht dicht besetzt sein, also wollen wir `adjacency_list` als Repräsentation nehmen
- Kanten und Knoten sollen mit Strings besetzt sein
- Konstruktion des Graphentyps

```
typedef adjacency_list<vecS, vecS, undirectedS,
    property<vertex_name_t, string>,
    property<edge_name_t, string > > Graph;
```

- Als nächstes brauchen wir die PropertyMaps des Graphen

Six Degrees of Kevin Bacon (4)

- Die bekommen wir so

```
property_map<Graph, vertex_name_t>::type actor_name =
    get(vertex_name, g);
property_map<Graph, edge_name_t>::type connecting_movie =
    get(edge_name, g);
```

- Nun müssen wir uns um die Verlinkung kümmern
- Sobald ein Schauspieler im Graph schon vorhanden ist, muss nur noch eine passende Kanten gezogen werden
- Für müssen als eine Relation zwischen Strings und Knoten herstellen können
- Also legen wir eine `std::map` an:

```
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef std::map<string, Vertex> NameVertexMap;
NameVertexMap actors;
```

Six Degrees of Kevin Bacon (5)

- Wenn ein Schauspielername nicht in der Map ist, fügen wir einen neuen Knoten in den Graphen ein
- Das neue Name/Knoten-Paar speichern wir dann
- Wenn der Name schon im Graphen ist, holen wir uns ihn aus der Map
- Code:

```
Vertex u;
NameVertexMap::iterator pos;
tie(pos, inserted) = actors.insert(
    std::make_pair(actors_name, Vertex()));
if (inserted) {
    u = add_vertex(g);
    actor_name[u] = actors_name;
    pos->second = u;
} else
    u = pos->second;
```

Six Degrees of Kevin Bacon (6)

- Das selbe machen wir für den zweiten Schauspieler (Vertex v)
- Als letztes müssen wir noch die Kanten erzeugen und in den Graphen einfügen:

```
graph_traits<Graph>::edge_descriptor e;
tie(e, inserted) = add_edge(u, v, g);
if (inserted)
  connecting_movie[e] = movie_name;
```

- Wir wollen aber auch die „Kevin Bacon Numbers“ speichern
- Dazu müssen wir die Schritte aufsummieren
- Dies können wir tun, indem wir unseren eigenen BFS-Visitor bauen
- Jedes mal, wenn ein Knoten besucht wird, wird dessen tree_edge-Methode aufgerufen

Six Degrees of Kevin Bacon (7)

```
template <typename DistanceMap>
class bacon_number_recorder : public default_bfs_visitor
{
public:
  bacon_number_recorder(DistanceMap dist) : d(dist) { }

  template <typename Edge, typename Graph>
  void tree_edge(Edge e, const Graph& g) const
  {
    typename graph_traits<Graph>::vertex_descriptor
      u = source(e, g), v = target(e, g);
      d[v] = d[u] + 1;
  }
private:
    DistanceMap d;
};
```

Six Degrees of Kevin Bacon (8)

- Dann bauen wir uns eine Factory-Funktion für unseren BFS-Visitor (macht das schreiben einfacher)

```
template <typename DistanceMap>
  bacon_number_recorder<DistanceMap>
  record_bacon_number(DistanceMap d)
  {
    return bacon_number_recorder<DistanceMap>(d);
  }
```

- Die Bacon-Numbers speichern wir in einem Vector:

```
std::vector<int> bacon_number(num_vertices(g));
```

Six Degrees of Kevon Bacon (9)

- Jetzt starten wir die Breitensuche mit Kevin Bacon als Quellknoten:

```
Vertex src = actors["Kevin Bacon"];
bacon_number[src] = 0;
breadth_first_search(g, src,
    visitor(record_bacon_number(&bacon_number[0])));
```

- Für die Ausgabe iterieren wir über alle Knoten im Graphen:

```
graph_traits<Graph>::vertex_iterator i, end;
for (boost::tie(i, end) = vertices(g); i != end; ++i)
    std::cout << actor_name[*i]
              << "'s bacon number is "
              << bacon_number[*i] << std::endl;
```

Six Degrees of Kevin Bacon (10)

- Objekte vom Typ `graph_traits<Graph>::vertex_iterator` können nur verwendet werden, wenn mit `adjacency_list` mit `VertexList=vecS` anlegen
- Haben wir hier aber getan :-)

Bestensuche (Greedy Best-First)

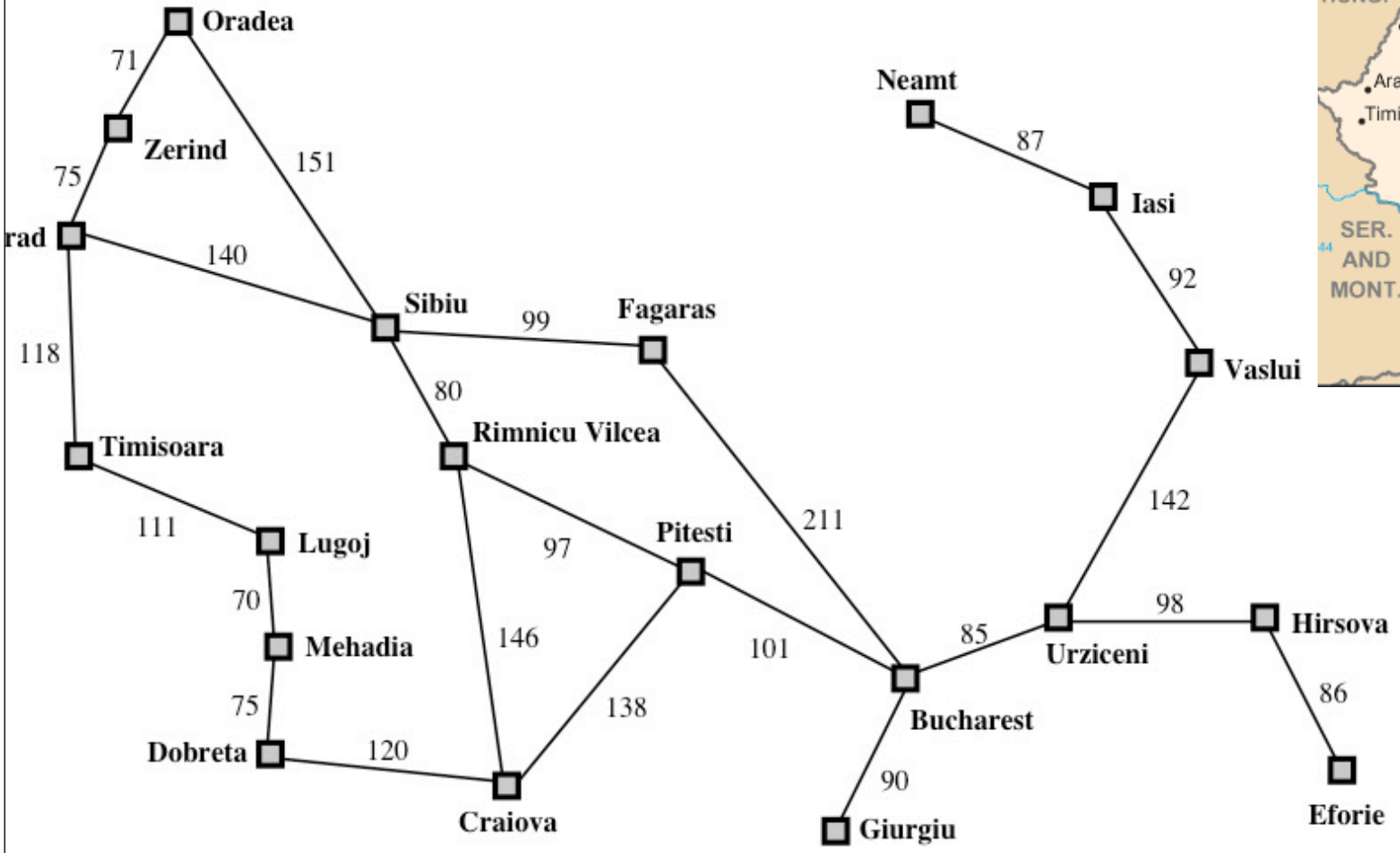
- Beispiel für einen greedy („gierigen“) Algorithmus
- Gehe aus von `TreeSearch`
- Bewerte Knoten durch $f(n) = h(n)$ (zulässig oder nicht)
- Sortiere bei `InsertAll` nach Knotenwerten (billigste vor)
- Ende bei zuerst gefundenen Zielknoten

  - ☹ Zeitbedarf: $O(b^m)$, wenn m Maximaltiefe des Baums
  - ☹ Speicherbedarf: $O(b^m)$ (da alle Knoten im Speicher)
  - ☹ Unvollständig (da anfällig für „Sackgassen“)
  - ☹ Nicht optimal (siehe folgendes Beispiel)

Zeit und Speicher praktisch oft besser bei guter *h*-Funktion

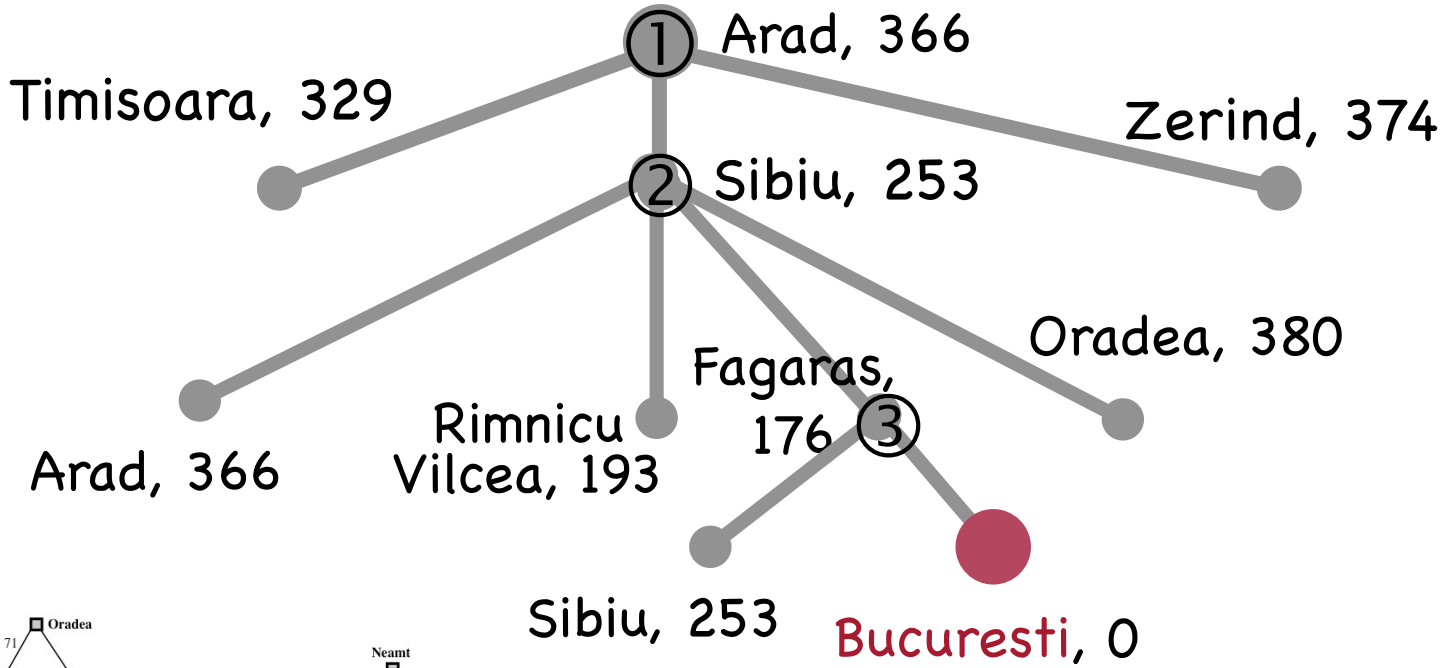
Worst-Case-Komplexitätsbetrachtungen sind inadäquat für (gute) Heuristikfunktionen!

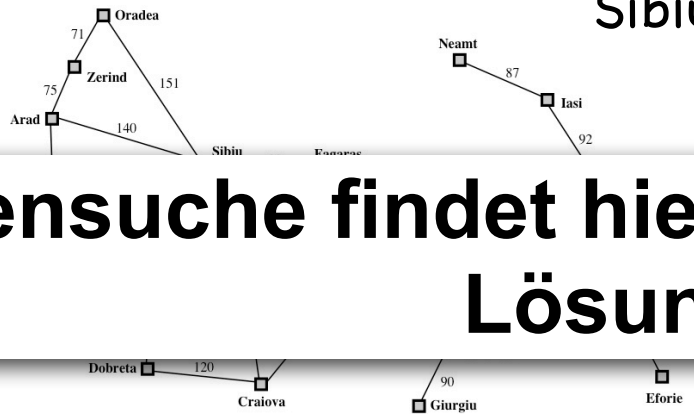
Das Reiseproblem



Bestensuche beim Reiseproblem (1)

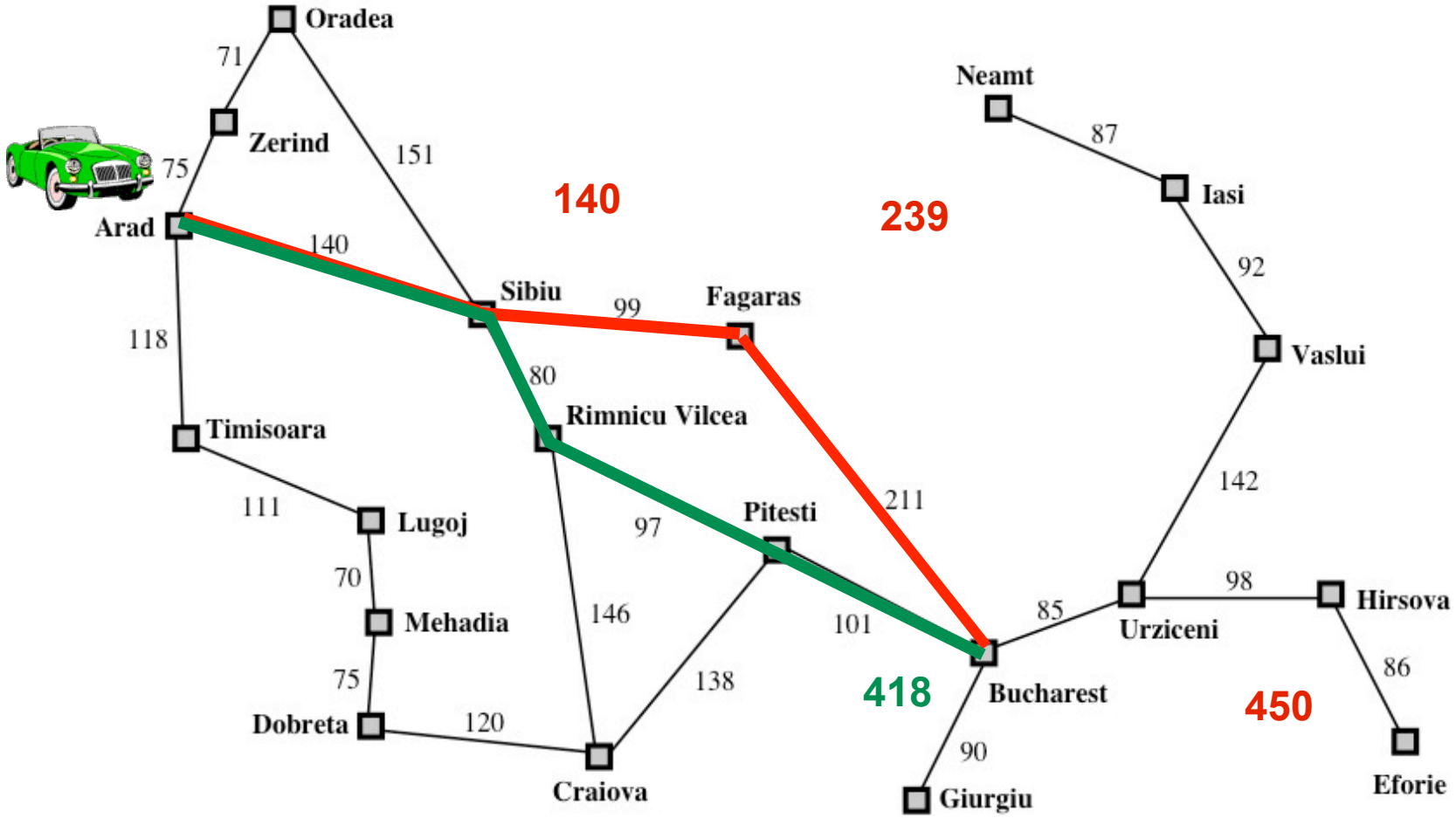
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374





Bestensuche findet hier nicht die optimale Lösung!

Bestensuche beim Reiseproblem (2)



Der Algorithmus A*

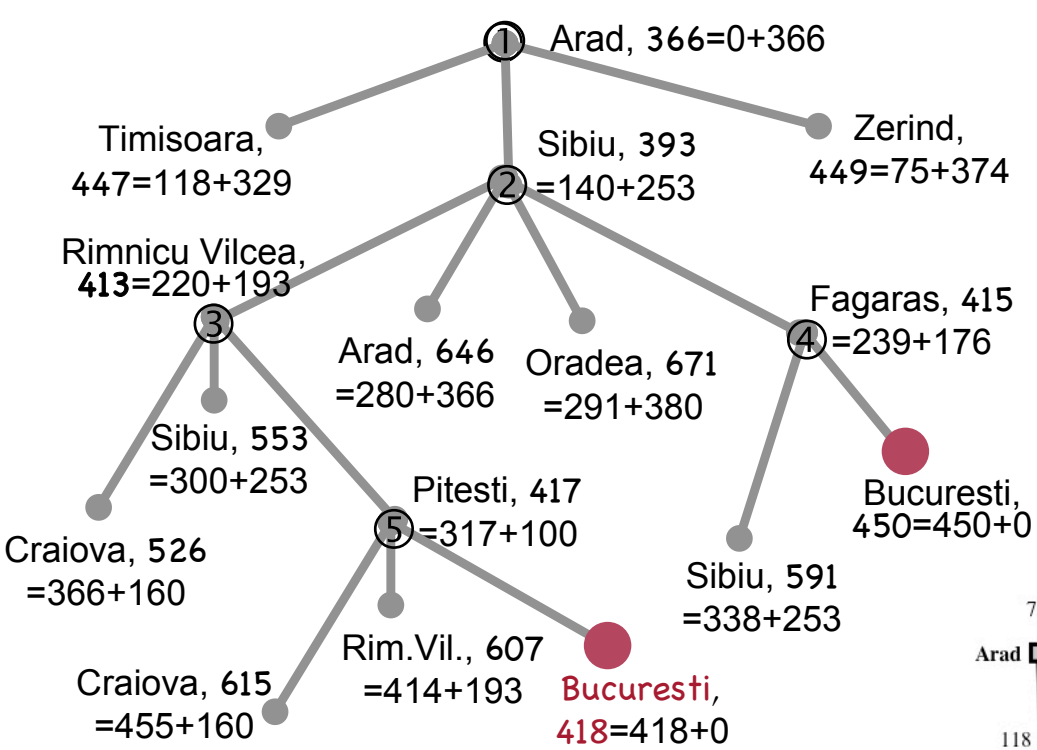
- Geh vor wie bei *uniform-cost-Suche* (Standard-Kostensuche)
- bewerte Knoten durch $f(n)=g(n)+h(n)$, wobei $h(n)$ zulässig
- sortiere bei `INSERTALL` nach Knotenwerten (billigste vor)
- ende erst, wenn ein Zielknoten expandiert werden müsste

  - ☹ Zeitbedarf: $O(b^{(1+\lfloor C^*/\varepsilon \rfloor)})$ (A* ≈ *uniform cost* für $h(n)=0$)
  - ☹ Speicherbedarf: $O(b^{(1+\lfloor C^*/\varepsilon \rfloor)})$ (alle Knoten im Speicher)
  - ☺ Vollständig (Details folgen)
  - ☺ Optimal (Details folgen)

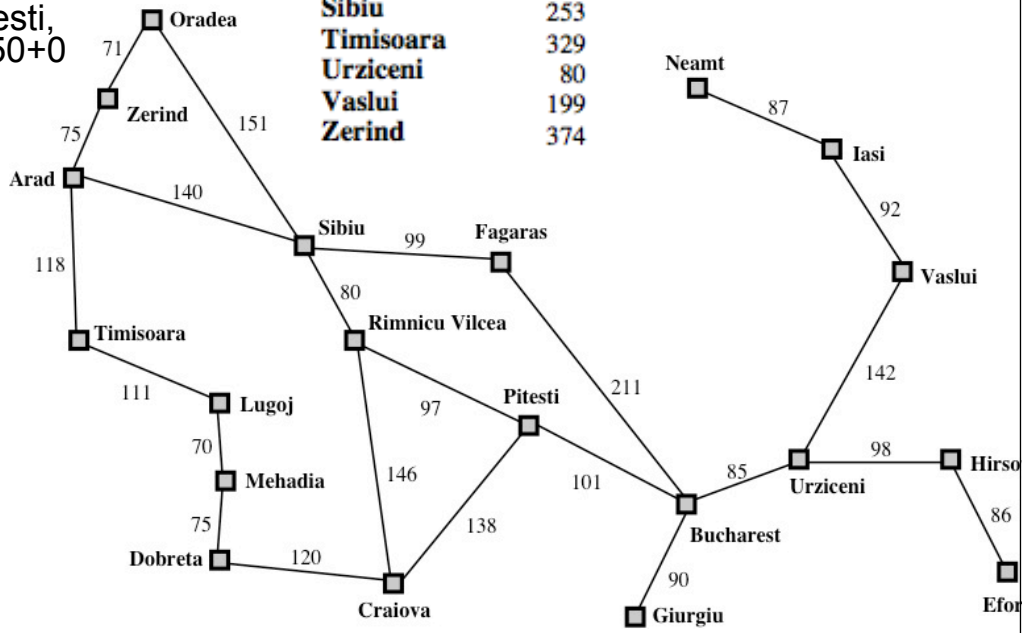
- Vollständigkeit & Optimalität schon früh bewiesen (1968)
- Nochmal: worst-case-Betrachtungen inadäquat für (gute) Heuristiken

Nils Nilsson, *1933

Beispiel: A* beim Reiseproblem

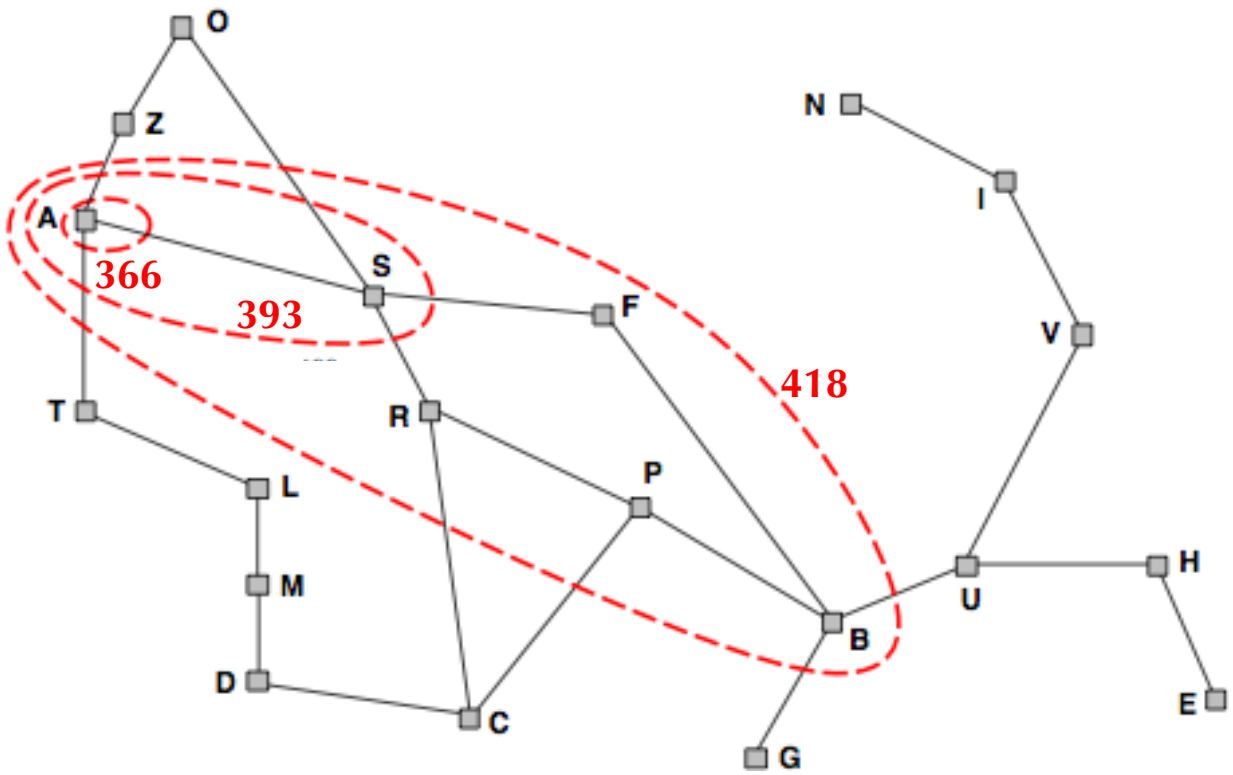


Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



Qualitatives Verhalten von A*

- A* propagiert wachsende „f-Konturen“ ab Start, wobei $f_i < f_{i+1}$



- expand. alle n: $f(n)<C^*$
- exp. einige n: $f(n)=C^*$
- exp. kein n: $f(n)>C^*$
- A* ist optimal effizient unter den optimalen Algorithmen, gegeben h

Andererseits
(schlecht bei vielen Zielknoten im Suchraum):
A* macht „Breitensuche durch alle plausiblen Pfade“!