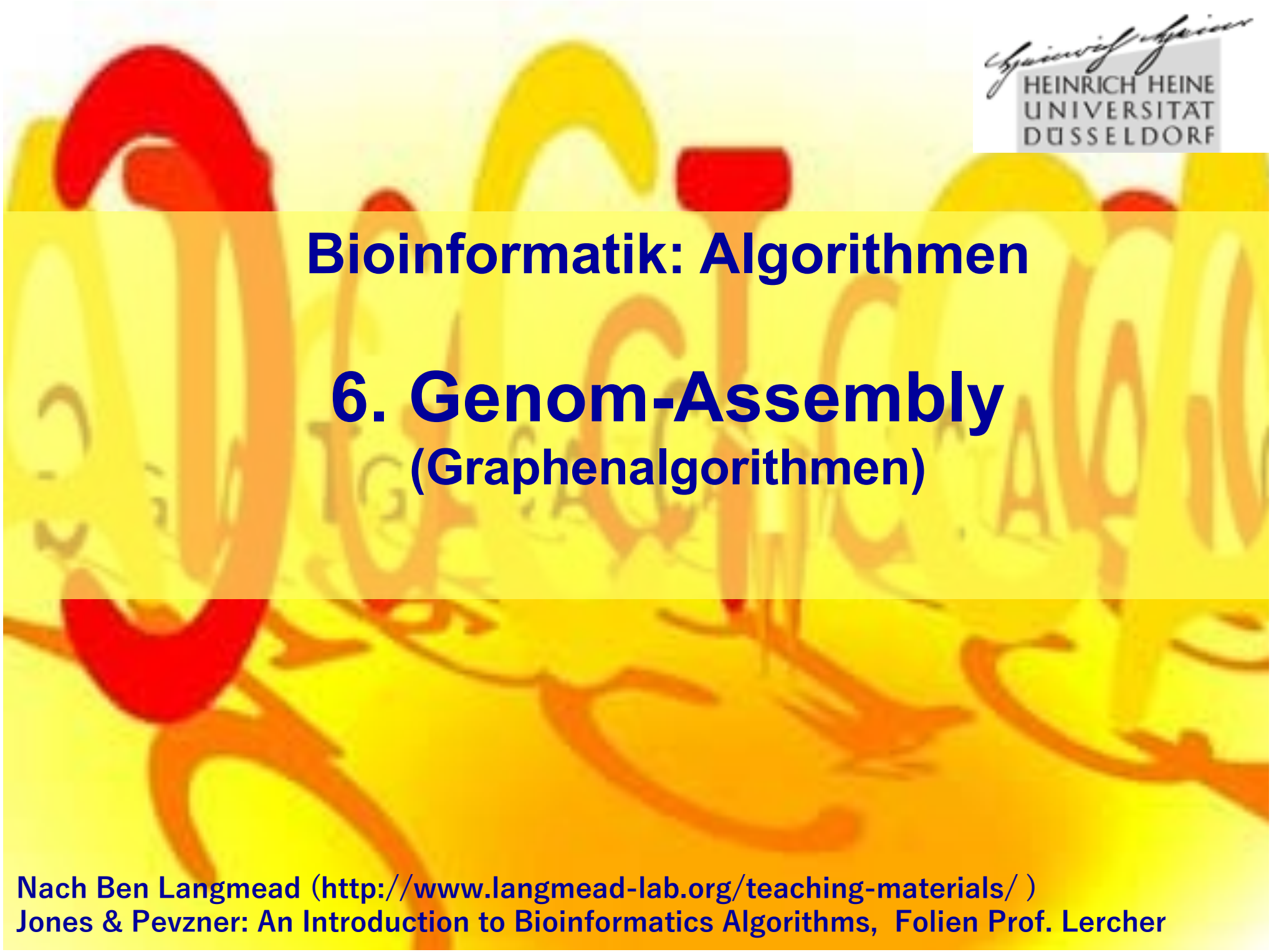


Bioinformatik: Algorithmen

6. Genom-Assembly
(Graphenalgorithmen)

Nach Ben Langmead (http://www.langmead-lab.org/teaching-materials/)
Jones & Pevzner: An Introduction to Bioinformatics Algorithms, Folien Prof. Lercher

Genomik: Grundlagenforschung

Genomics = "The branch of molecular biology concerned with the structure, function, evolution, and mapping of genomes."

Oxford dictionaries

Struktur/Organisation

Wie lautet die DNA-Sequenz eines Genoms?
Wo sind die Gene?
Wie ist die 3D-Struktur des Genoms in der Zelle?

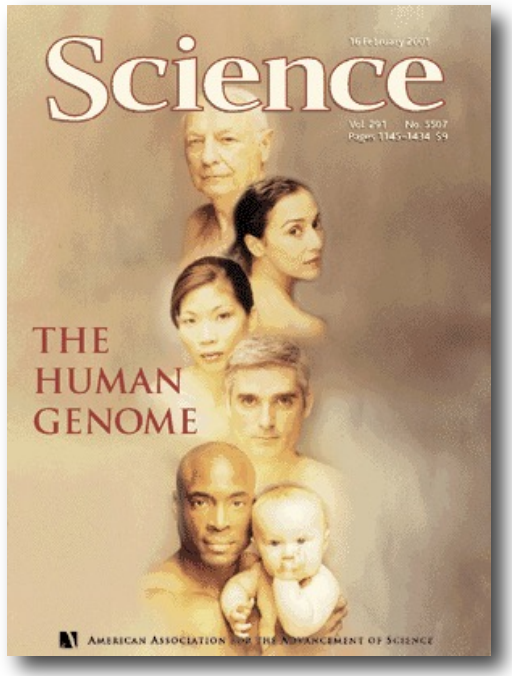
Funktion

Was macht die DNA in der Zelle?
Welche Gene interagieren mit welchen anderen Genen?
Wie weiß die Zelle, welche DNA "ein-" oder "ausgeschaltet" ist?

Evolution

Wie sind die heutigen ethnischen und Populationsstrukturen zustandegekommen?
Welche großen Ereignisse gab es in der Evolution unserer Genome?
Welche Teile des Genoms sind evolutionär konserviert?

Genomik: Grundlagenforschung



Das *Human Genome Project* hing entscheidend von Beiträgen von Informatikern ab, insbesondere von neuen Methoden zum DNA-Assembly.

Genomik: Werkzeug für die Medizin

"The branch of molecular genetics concerned with the study of genomes, specifically the identification and sequencing of their constituent genes and the application of this knowledge in medicine, pharmacy, agriculture, etc."

Collins English Dictionary

Wie sind Genotyp und Phänotyp verbunden?

Was ist der Unterschied zwischen Tumor-DNA und DNA aus gesundem Gewebe?

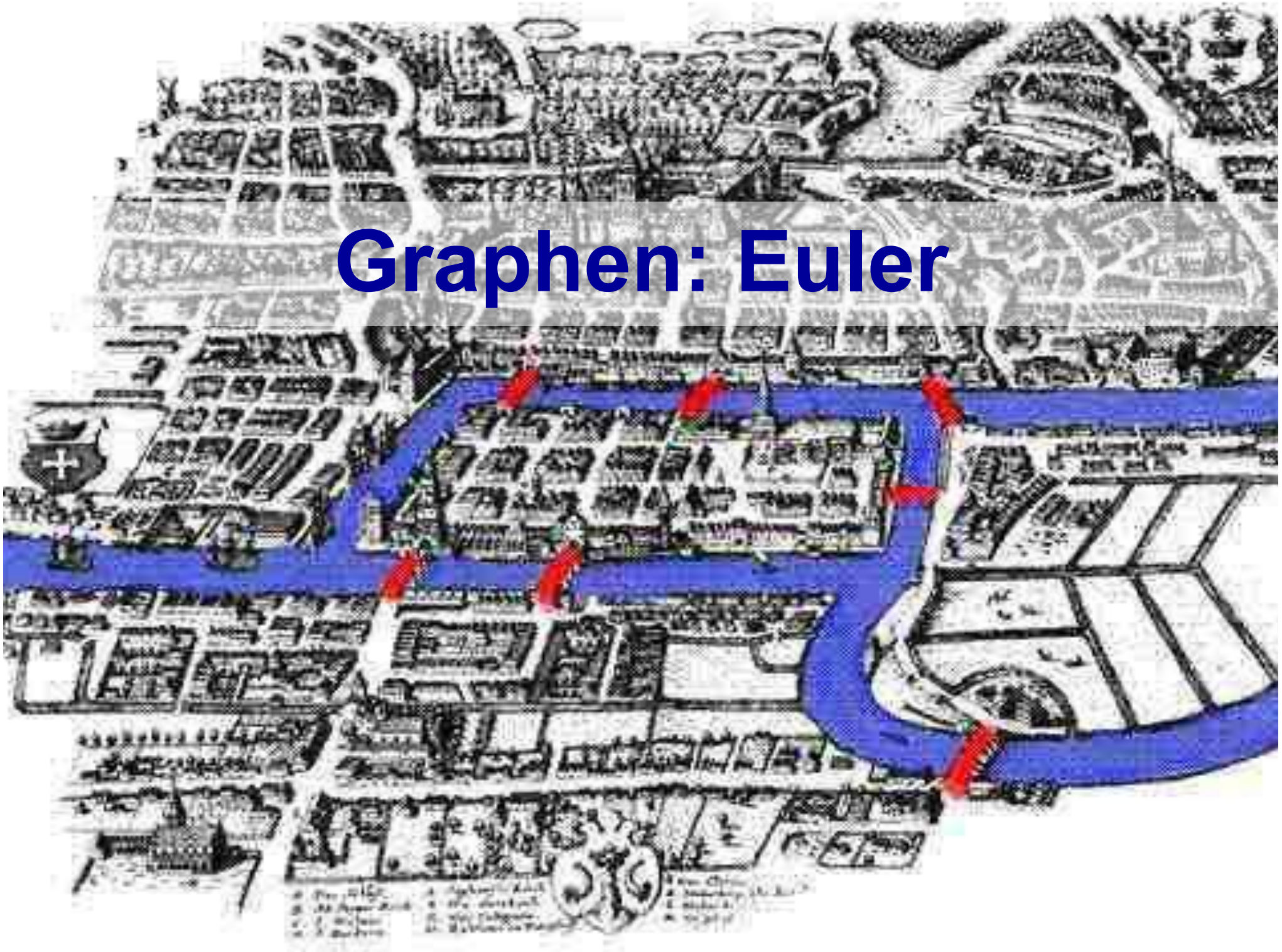
Können wir vorhersagen welche Medikamente geeignet sind für

- **einen bestimmten Krebspatienten?**
- **eine bestimmte seltene genetisch bedingte Krankheit?**

Können wir Schwachstellen in Verteidigungsmechanismen von Krankheitserregern entdecken und ausnutzen?

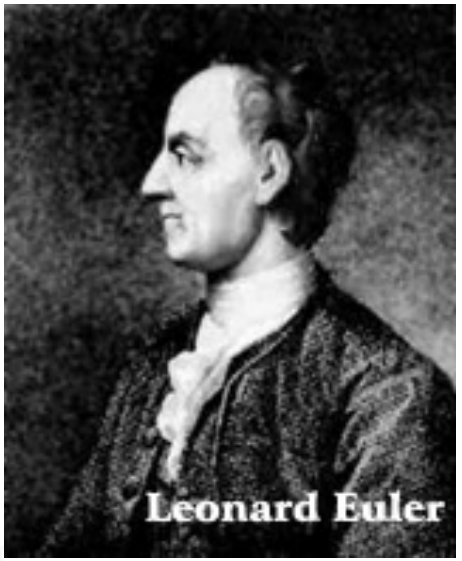
Können wir vorhersagen, mit welchem Grippe- oder Coronavirenstamm wir es in Zukunft zu tun haben?

Graphen: Euler



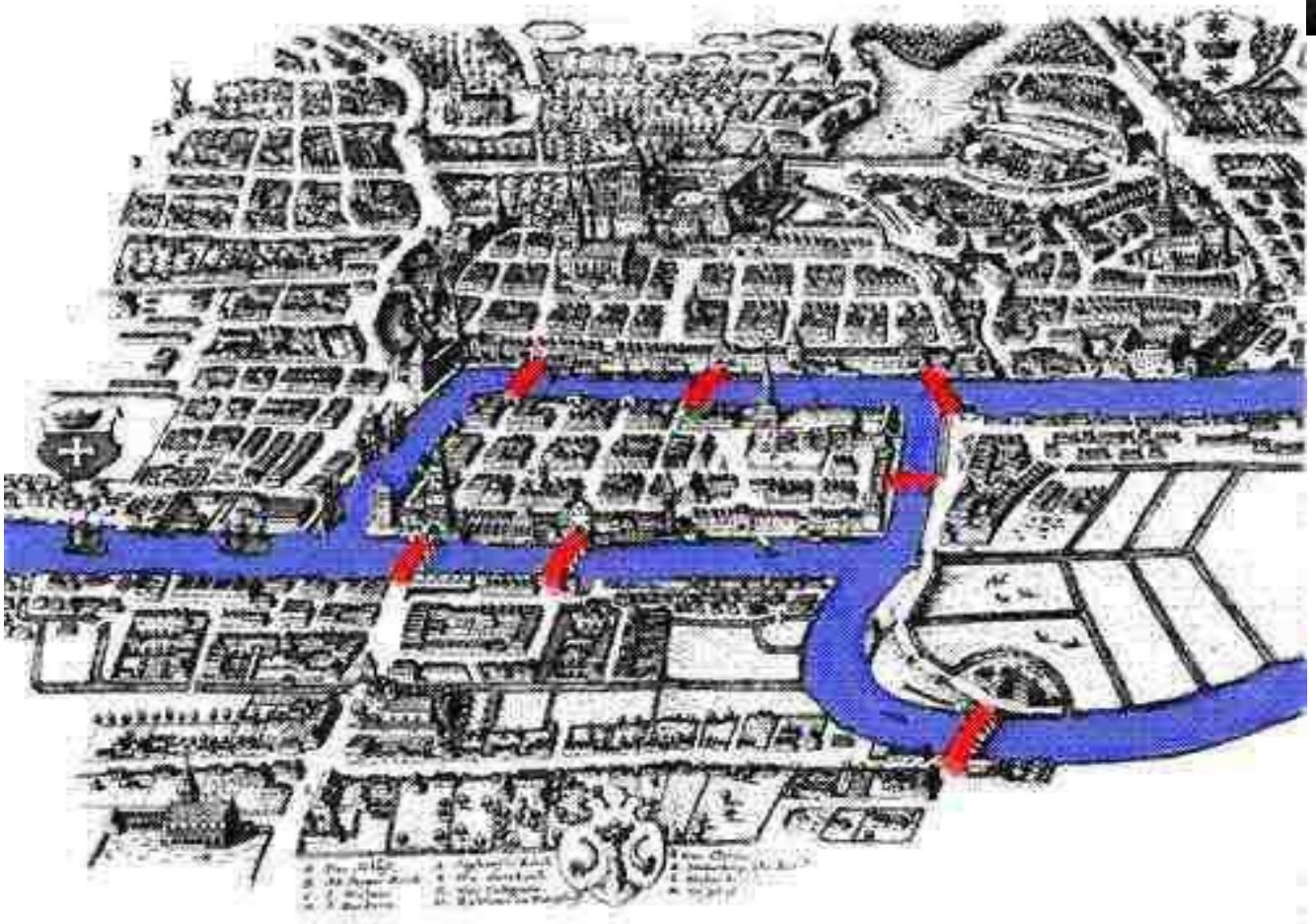
Die Brücken von Königsberg

(Kaliningrad)

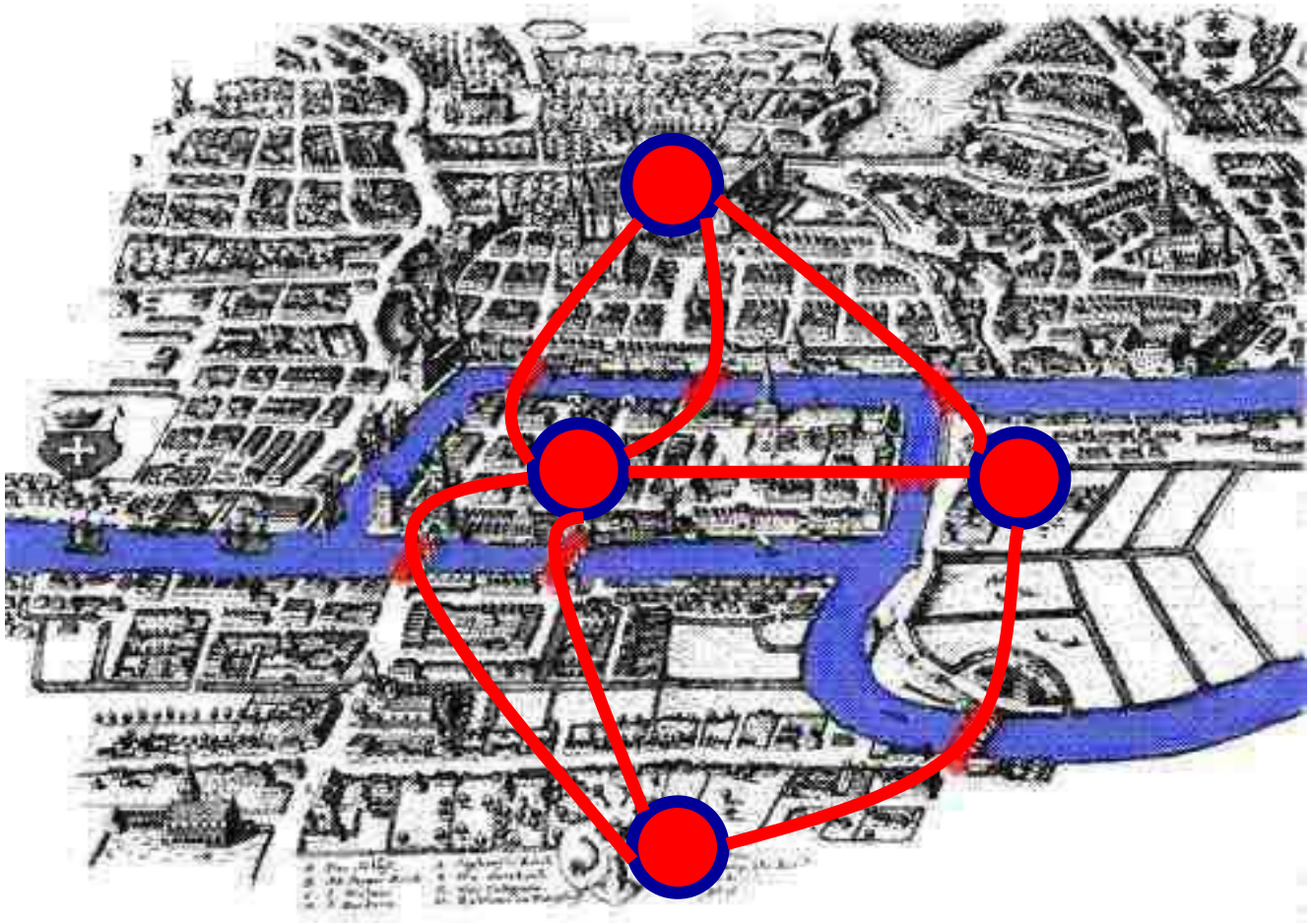


1735

Gibt es einen Stadtrundgang,
der jede Brücke genau einmal überquert?



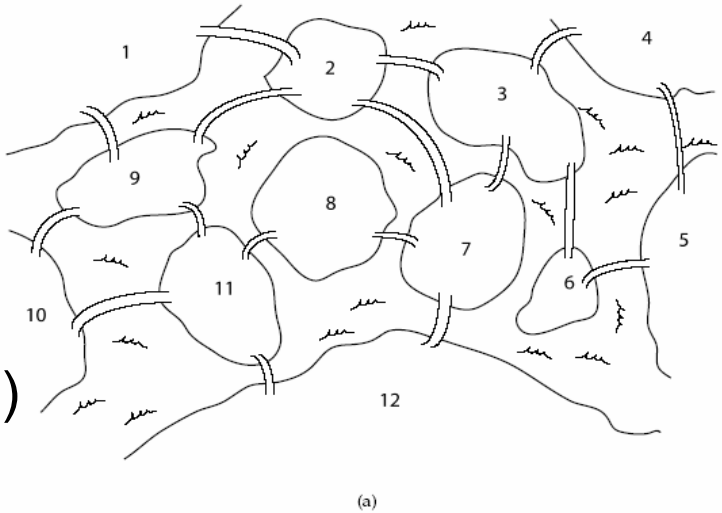
Leonard Eulers Lösung



- Jeder Knoten hat eine ungerade Zahl von Kanten
- Man kann keinen Knoten ‘korrekt’ durchlaufen

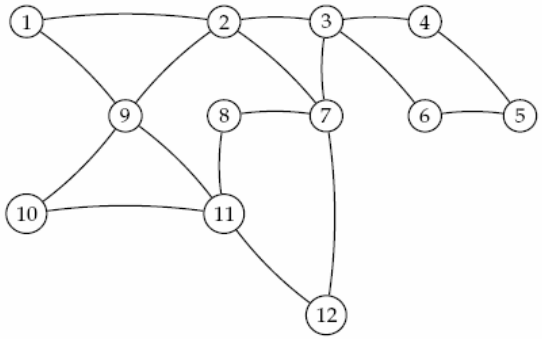
Satz von Euler

Definition: Ein **Graph** besteht aus einer Menge von **Knoten**, die durch **Kanten** verbunden sind.



(a)

Definition: Ein Knoten ist **ausbalanciert**, wenn sein **Knotengrad** (die Anzahl der Nachbarn) gerade ist



Definition: Ein **Euler-Zyklus** ist ein kontinuierlicher Weg, der jede Kante genau einmal durchläuft und am Ausgangsknoten wieder endet.

Satz: Ein zusammenhängender Graph $G = (V, E)$ besitzt **genau dann** einen Euler-Zyklus C, wenn alle Knoten ausbalanciert sind.

Satz von Euler: Beweis

Euler-Zyklen sind ausbalanciert:

Für jede eingehende Kante *k* muss es eine ausgehende Kante *k'* geben.

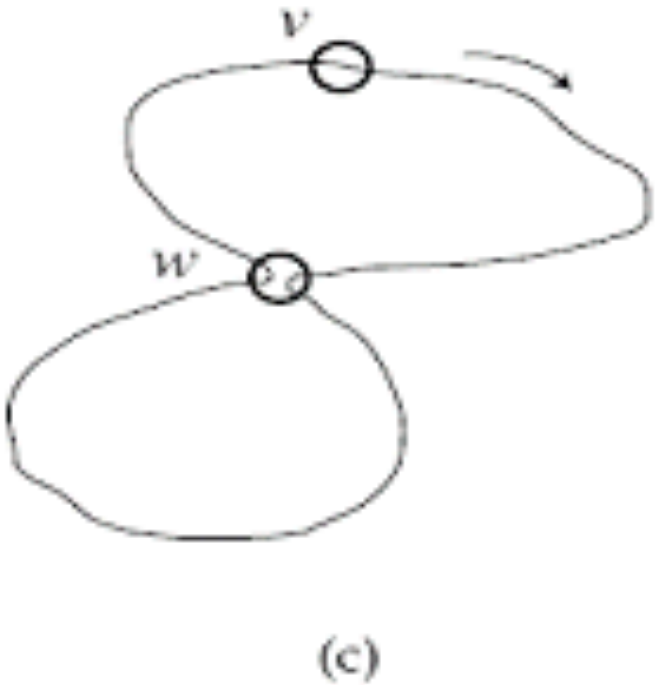
Ausbalancierte Graphen ergeben einen Euler-Zyklus:

Siehe Konstruktionsanweisung:

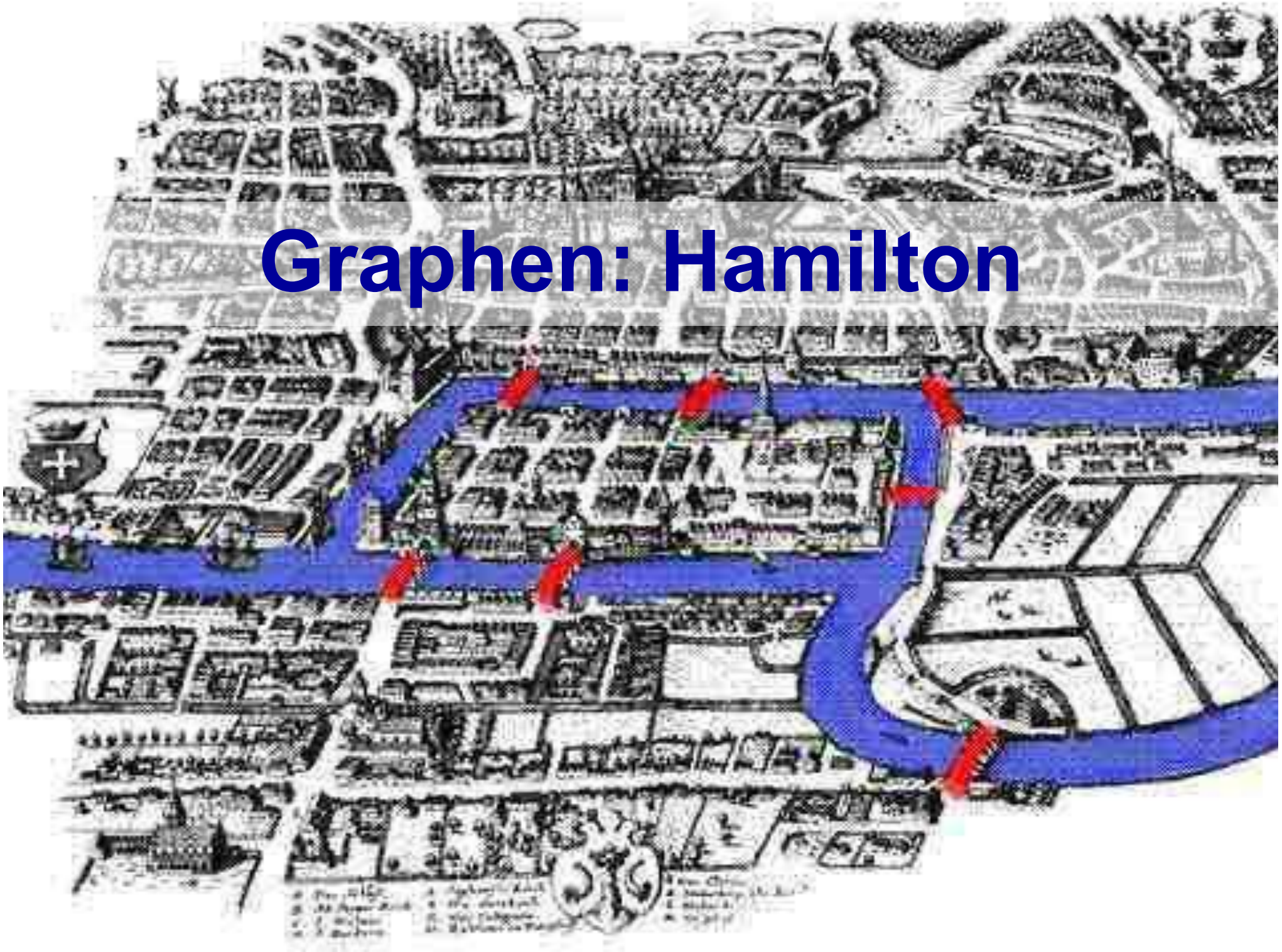
Konstruktion eines Euler-Zyklus

1. Beginne mit einem beliebigen Knoten *v* und forme einen beliebigen Zyklus bis zu einer Sackgasse. Weil der Graph ausbalanciert ist, muss das *v* sein.
2. Wenn das noch kein Euler-Zyklus ist, muss es einen Knoten *w* mit unbenutzten Kanten geben. Wiederhole 1 mit *w* als Startpunkt (und Endpunkt).
3. Verbinde die Beiden Zyklen zu einem Zyklus, und wiederhole (2.).

QED



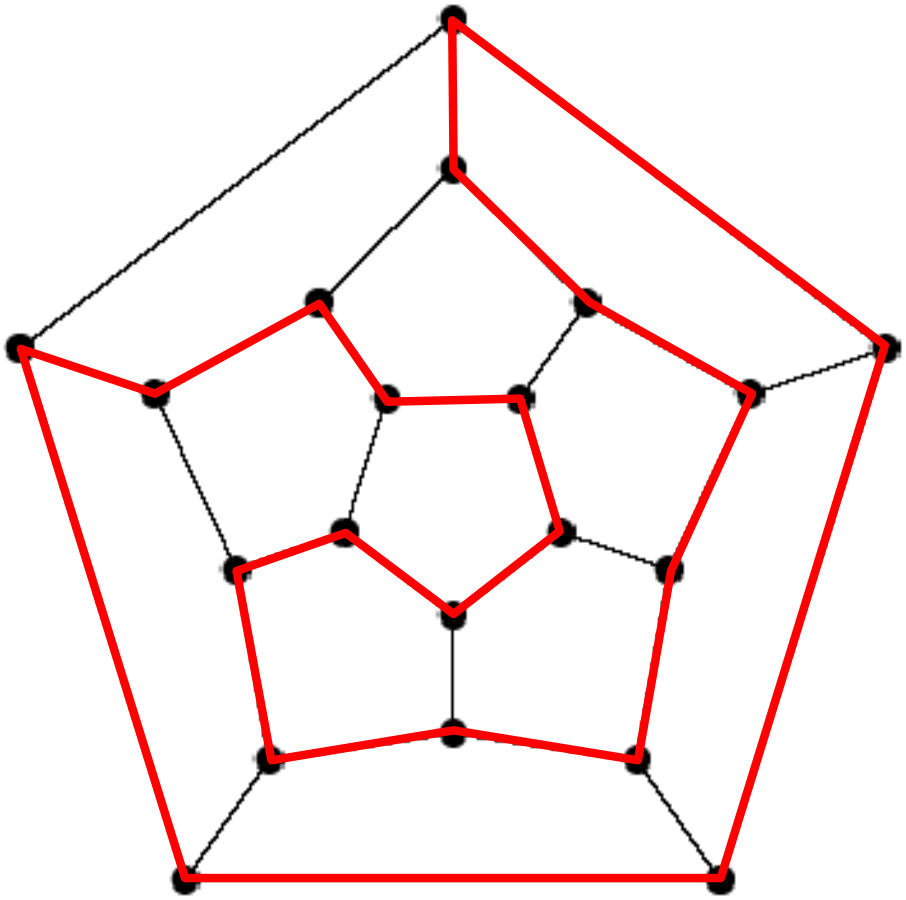
Graphen: Hamilton



Hamilton-Zyklus (HC)

Finde einen Zyklus, der jeden **Knoten** genau einmal durchläuft

- NP–schwer



Spiel
(erfunden von Sir William Hamilton, 1857)

5 Minuten über NP-Vollständigkeit

- **P** = {Probleme, die wir in polynomieller Zeit lösen können}
- Idealerweise zeigen wir, dass unsere Probleme in **P** sind.
- Falls wir das nicht hinkriegen, was sagen wir unserer Chefin?

1. "Ich bin zu blöd."
   gefährlich

2. "Es gibt keinen schnellen Algorithmus."
   Beweis einer unteren Schranke

3. "Ich kann es nicht, aber sonst kann es auch keiner"
   NP-Vollständigkeitsreduktion

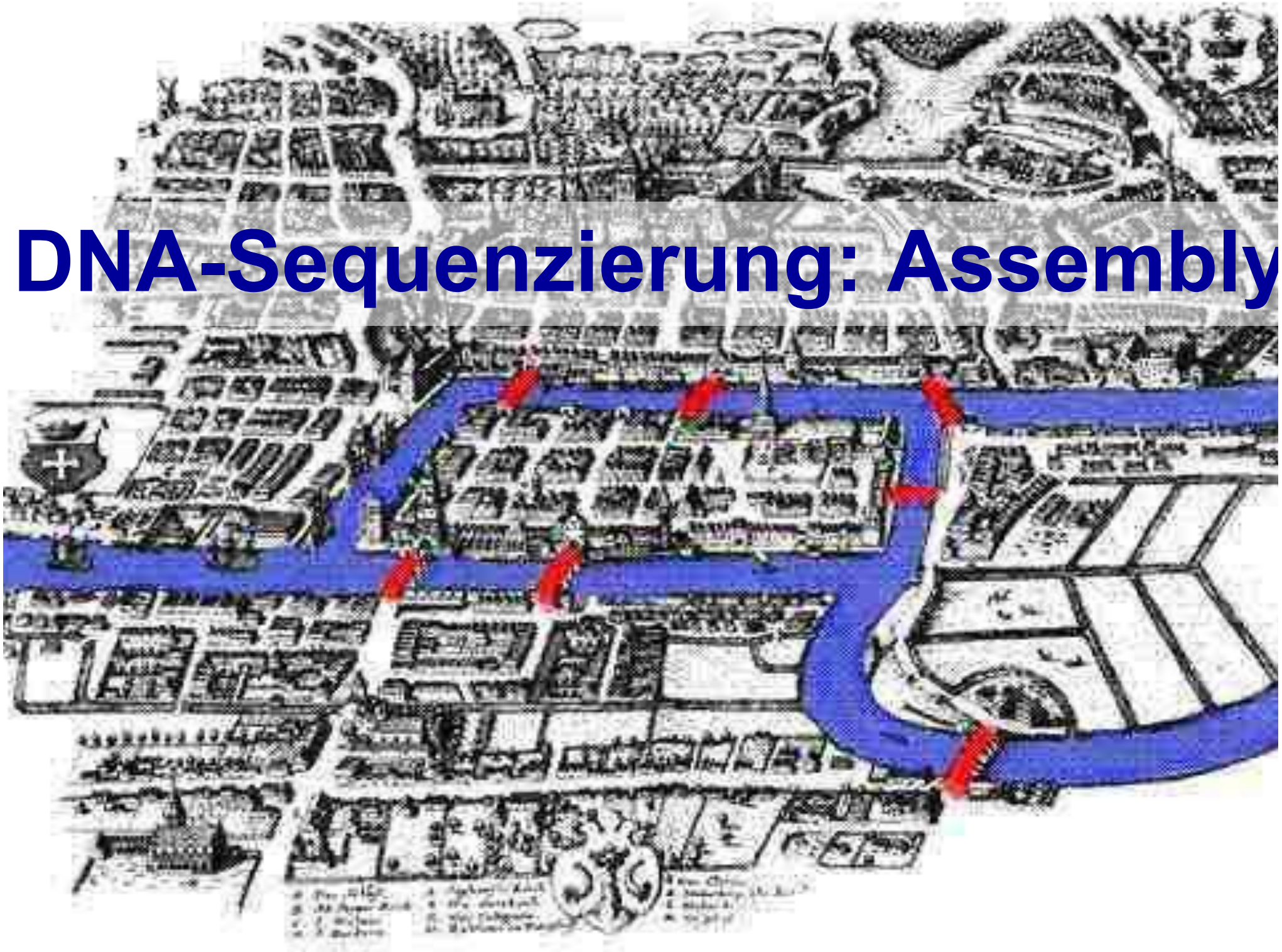
NP-Vollständigkeit

- Für viele Probleme finden wir weder einen Algorithmus in **P** noch eine exponentielle untere Laufzeitschranke.
- Zumindest können wir viele dieser Probleme kategorisieren ("alle gleich schwer", nämlich **NP-vollständig**) [frühe 1970er Jahre, Stephen Cook, Richard Karp]
- Dann: Problem **NP**-vollständig→ sehr wahrscheinlich, dass es nicht in **P** ist → mach was anderes
  - Heuristiken
  - Approximation
  - Exakte Algorithmen, die "niedrige Exponentialzeit" benötigen. (Algorithm Engineering)

Aber was heißt das genau?

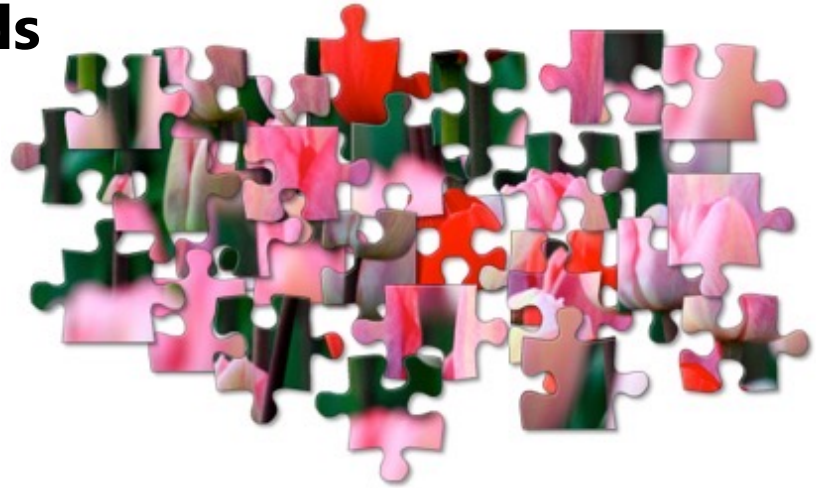
- **NP** = {
  }
- ”Zauberei”-Berechnungsmodell, “Glückliche” Algorithmen
- Beispiel: nichtdet. Algorithmus für Hamilton-Zyklus (HC)
- **NP** = {Probleme, deren Lösungen in **P** “überprüft” werden können}
- Klar: **P⊆NP.** Große Frage: **P = NP**?

DNA-Sequenzierung: Assembly



Assembly

Reads



+

Referenzgenom

Wie lösen wir das Puzzle ohne auf den Deckel zu schauen?

Input-DNA

Assembly

Whole-genome “shotgun”-Sequenzierung: Kopieren und zufälliges Fragmentieren der DNA

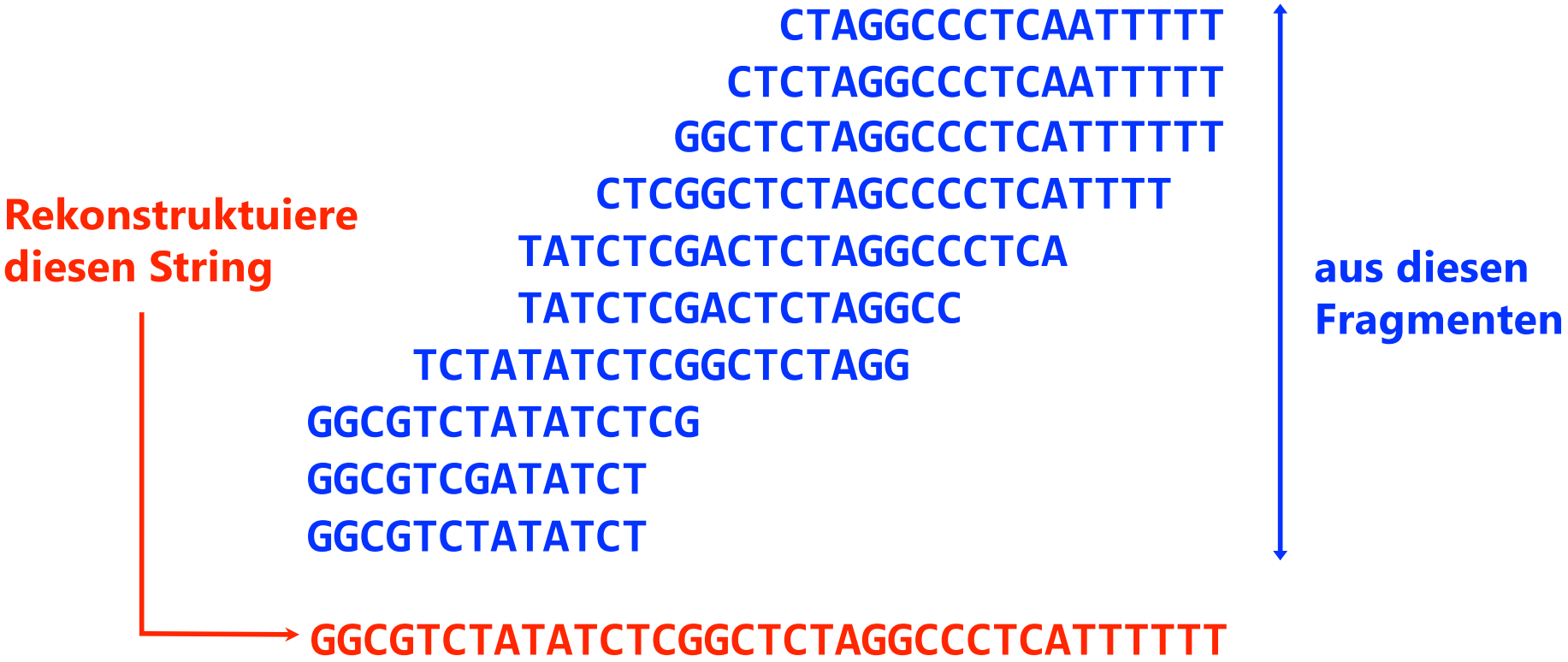
Input: GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

Kopien: GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

Fragmente: GGCGTCTA TATCTCGG CTCTAGGCCCTC ATTTTTT
GGC GTCTATAT CTCGGCTCTAGGCCCTCA TTTTTT
GGCGTC TATATCT CGGCTCTAGGCCCT CATTTTTT
GGCGTCTAT ATCTCGGCTCTAG GCCCTCA TTTTTT

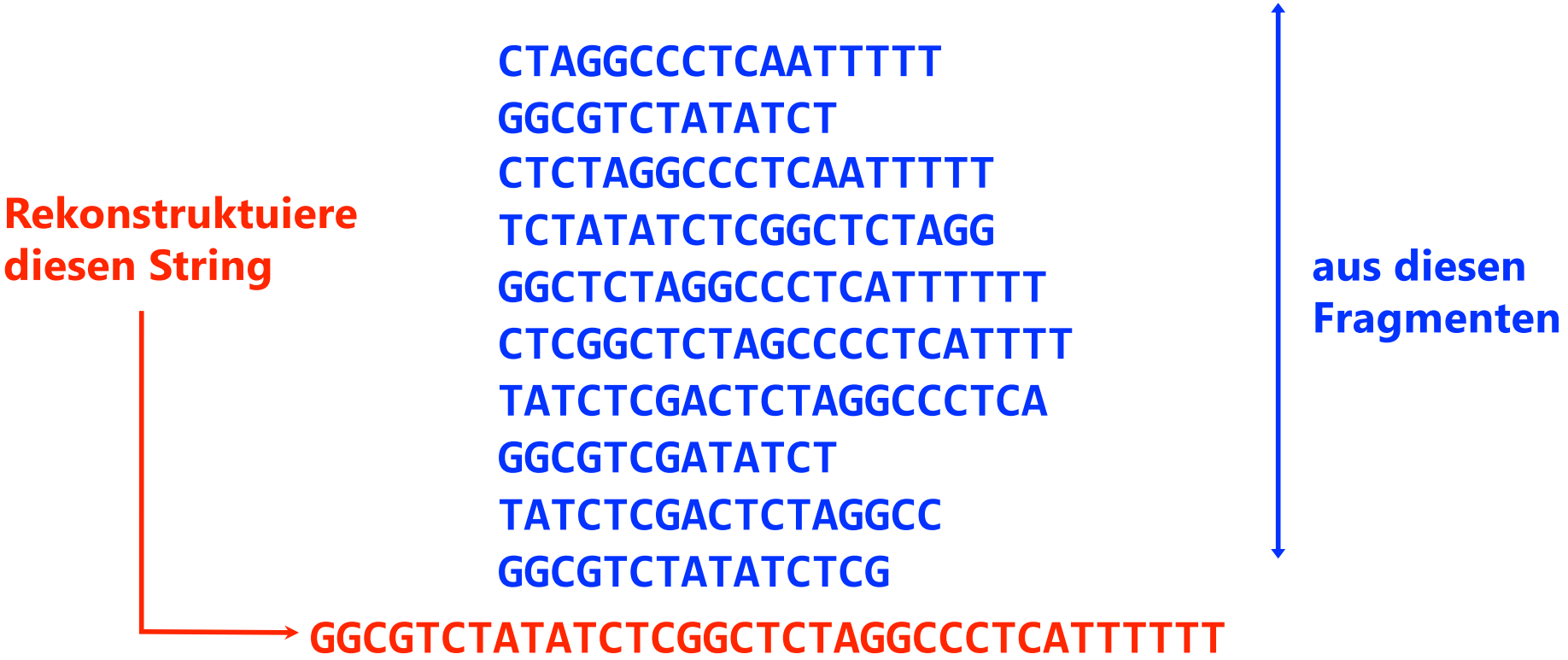
Assembly

Der Vorgang produziert eine große # von Fragmenten, so dass fast alle genomische Positionen von vielen Fragmenten überdeckt sind..



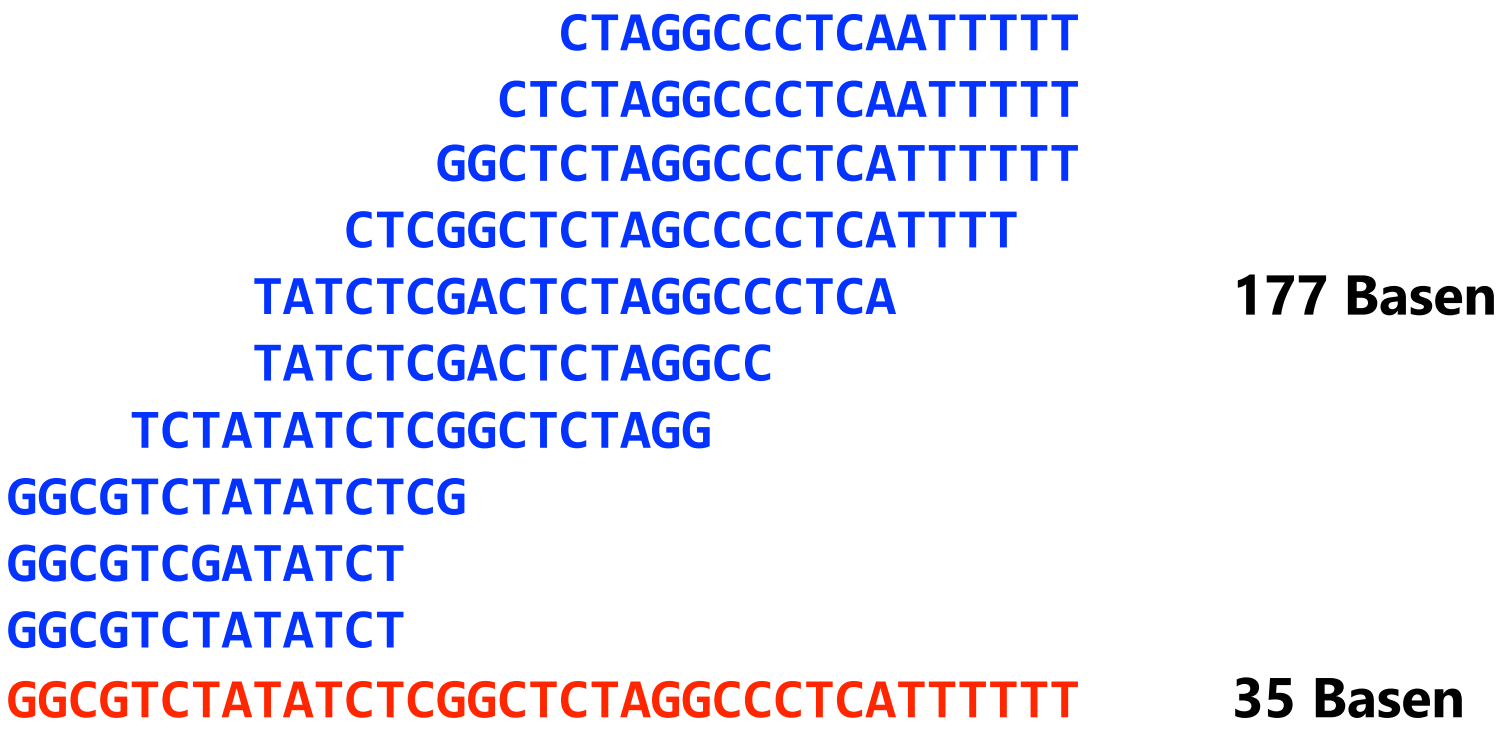
Assembly

...aber wir wissen nicht mehr, woher sie kommen



Assembly

Schlüsselbegriff: *coverage*: Wieviele *reads* überdecken gemittelt eine Position des Genoms.



Coverage= 177 / 35 ≈ 5x

Read Coverage

Länge des Genomsegments: ***L***
Anzahl Reads: ***n***
Read-Länge: ***l***

Coverage ***C = n l / L***

Wieviel Coverage ist genug?

Lander-Waterman Model:

Bei gleichförmiger Verteilung der Reads:
Coverage *C*=10 resultiert in 1 Lücke pro 1Mio Nukleotide (1Mb)

Assembly

Basisprinzip: je mehr Ähnlichkeit zwischen dem Ende eines Reads und dem Beginn eines anderen besteht...

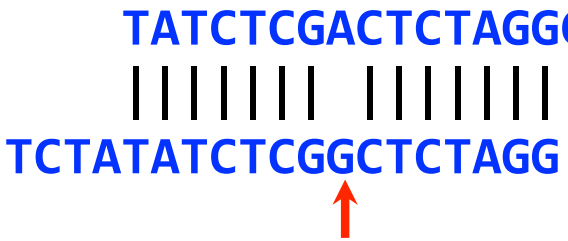
```
TATCTCGACTCTAGGCC
||||||| |||||||
TCTATATCTCGGCTCTAGG
```

...desto wahrscheinlicher kommen diese vom gleichen Bereich des Genoms:

```
TATCTCGACTCTAGGCC
 TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
```

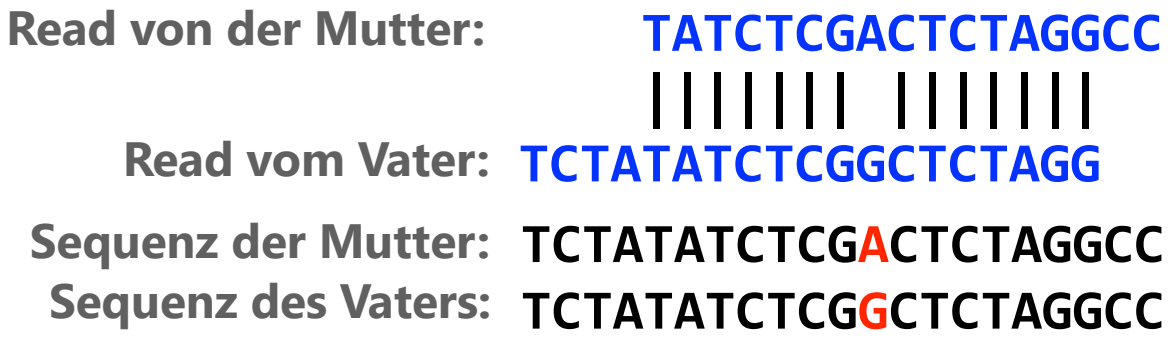
Assembly

Nehmen wir an, dass zwei *Reads* in der Tat vom gleichen genomischen Bereich stammen. Warum gibt es dann Unterschiede?



1. **Sequenzierungsfehler**
2. **Unterschied zwischen geerbten chromosomalen Kopien**

   Menschen, z. B., sind *diploid*; wir haben zwei Kopien jedes Chromosoms, eine vom Vater, eine von der Mutter. Die Kopien unterscheiden sich (leicht)



Strings: Definitionen

Ein String S ist eine endliche geordnete Folge von Zeichen.

Die Zeichen kommen aus einem Alphabet Σ. Wir nehmen oft an, dass sich O(1) Elemente in Σ befinden.

- **DNA-Alphabet: { A, C, G, T }**
- **Aminosäuren-Alphabet: { A, R, N, D, C, E, Q, G, H, I, L, K, M, F, P, S, T, W, Y, V }**

Die Länge von S, notiert als |S|, ist die Anzahl von Zeichen in in S

ϵ ist der leere String. |ϵ| = 0

Strings: Definitionen

Die Konkatenation zweier Strings S und T besteht aus den Zeichen von S gefolgt von den Zeichen in T und wird ST geschrieben.

S ist ein Teilstring von T wenn es (möglicherweise leere) Strings u und v gibt mit T = uSv

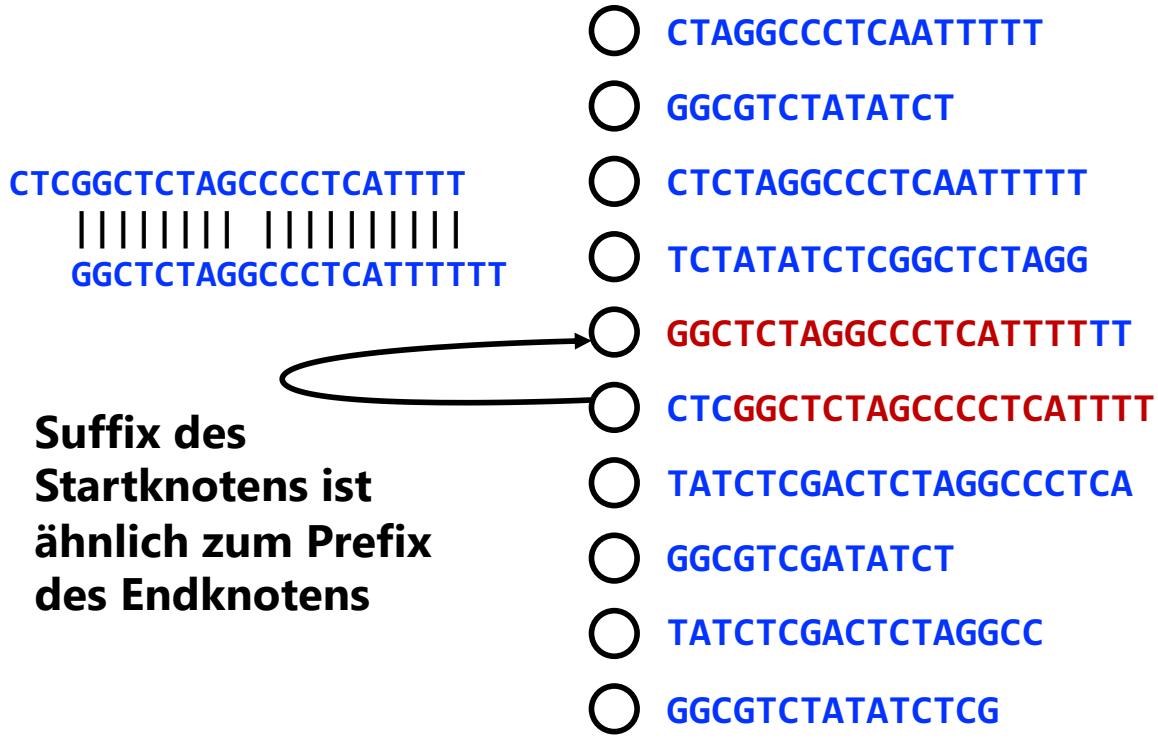
S ist ein Präfix von T wenn es einen String u gibt mit T = Su. Wenn weder S = ϵ noch u = ϵ, dann ist S echtes Präfix von T.

Die Definitionen von Suffix und echtem Suffix sind ähnlich.

Python-Demo: http://nbviewer.ipython.org/6512698

Overlap-Graph

Alle Überlappungen (*overlaps*) finden: Baue gerichteten Graphen, in dem gerichtete Kanten sich überlappende Knoten (*reads*) verbinden

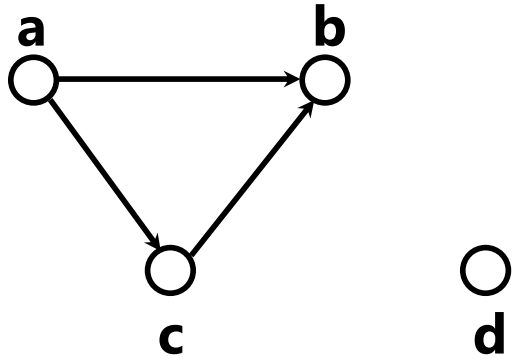


Gerichtete Graphen: Wiederholung

Ein gerichteter Graph G = (V, E) besteht aus der Knotenmenge V und der Menge der gerichteten Kanten, E

Eine gerichtete Kante ist ein geordnetes Knotenpaar (Startknoten, Zielknoten)

- **Knoten werden meist als Kreise dargestellt**
- **Kanten als Linien mit Pfeil, die Start und Zielknoten verbinden**



V = { a, b, c, d }

E = { (a, b), (a, c), (c, b) }

Startknoten Zielknoten

Gerichtete Kanten nennt man auch *Bögen*

Engl.: Directed graph oder *digraph*

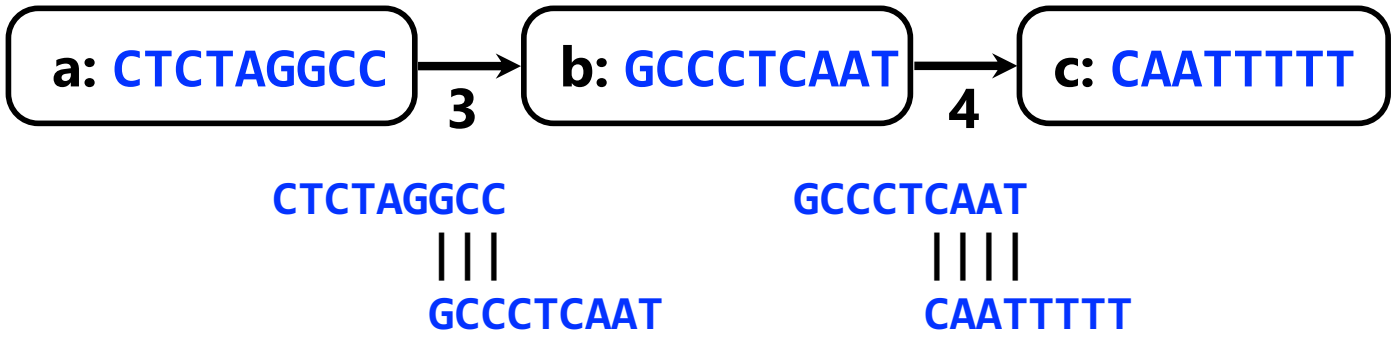
Overlap-Graph

Unten: Overlap-Graph, bei dem overlap = Suffix-Prefix-match von mindestens 3 Zeichen

Knoten = reads, Kanten = overlap zwischen Suffix des Startknotens und Prefix des Zielknotens

Knoten (reads): { a: CTCTAGGCC, b: GCCCTCAAT, c: CAATTTTT }

Kanten (overlaps): { (a, b), (b, c) }



Overlaps finden

Wie bauen wir den Overlap-Graphen?

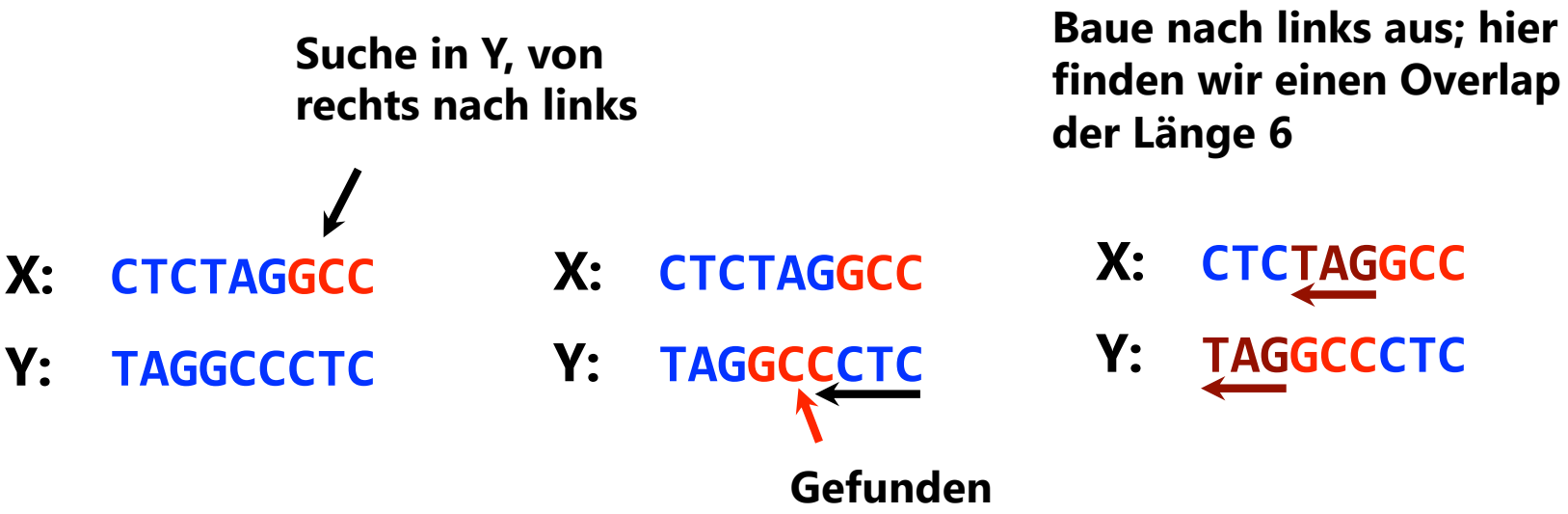
Beispieldefinition:

overlap = Längstes Suffix von X der Länge ≥ l matcht exakt einem Prefix von Y. Parameter l ist gegeben.

Overlaps finden

Idee: Suche in Y nach Vorkommen von Suffixes der Länge l von X. Baue diese nach links aus, um zu checken, ob das ganze Prefix von Y matcht.

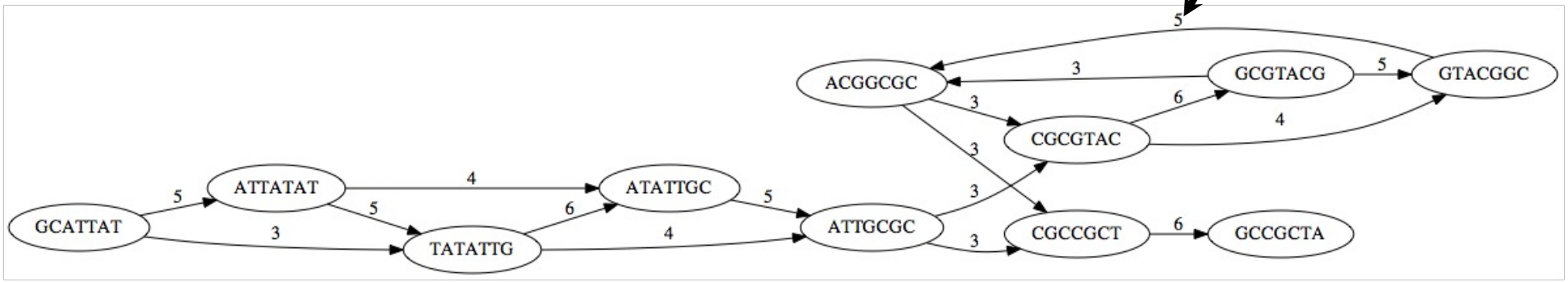
Z. B. $l = 3$



Overlaps finden

Beispiel-Overlap-Graph für l = 3

Kantenlabel = Länge des Overlaps



Originalstring: GCATTATATATTGCGCGTACGGCGCCGCTACA

Wie lösen wir jetzt das Assembly-Problem?

Shortest common superstring
Geg. Menge von Strings S, finde SCS(S) = kürzester String, der alle Strings in S als Substrings enthält

Ohne "kürzester"?

Beispiel: **S:** BAA AAB BBA ABA ABB BBB AAA BAB

Konkatenation: BAAAABBBAABAABBBBBAAABAB

SCS(S): AAABBBABAA

AAA
AAB
ABB
BBB
BBA
BAB
ABA
BAA

Shortest common superstring

Können wir es lösen?

Baue modifizierten Overlap-Graph mit
Kantenkosten = - (Overlaplänge)

SCS ist dann ein kürzester Pfad, der jeden Knoten genau einmal besucht

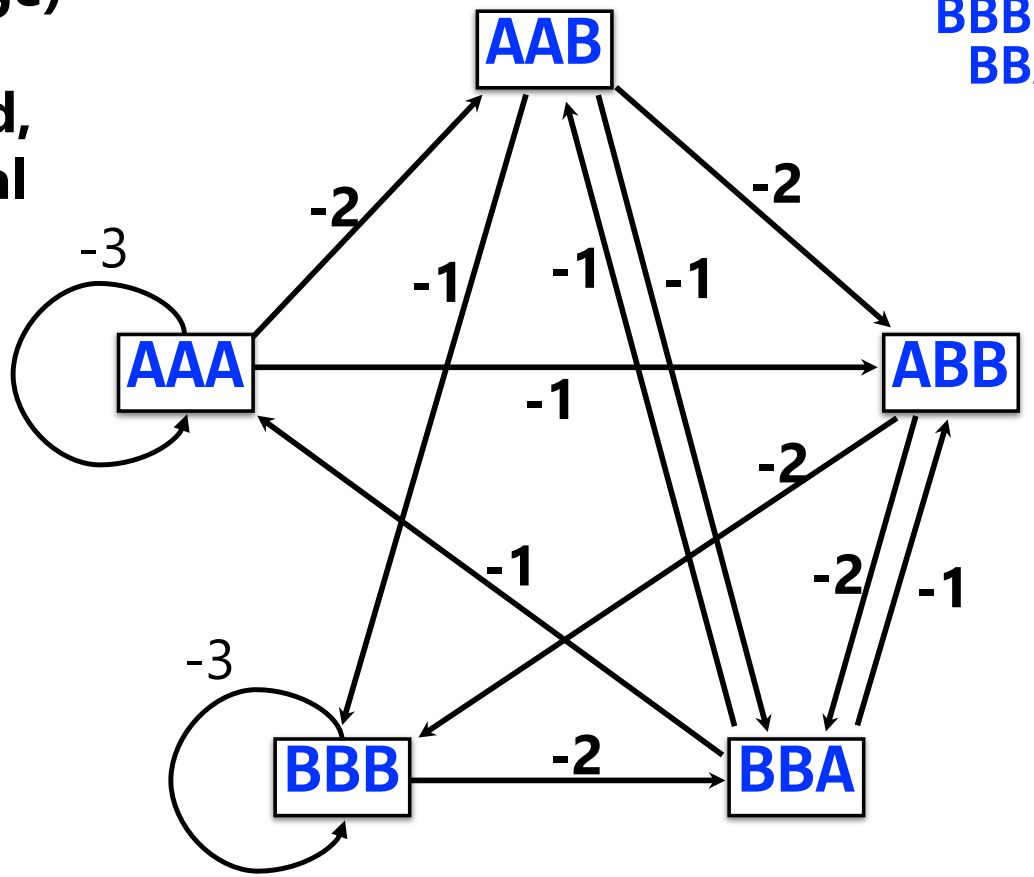
Das ist (fast) das Traveling-Salesman-Problem (TSP)... NP-schwer!

SCS ist auch NP-schwer (ohne Beweis)

S: AAA AAB ABB BBB BBA

SCS(S): AAABBBA

AAA
AAB
ABB
BBB
BBA



Shortest common superstring: Greedy

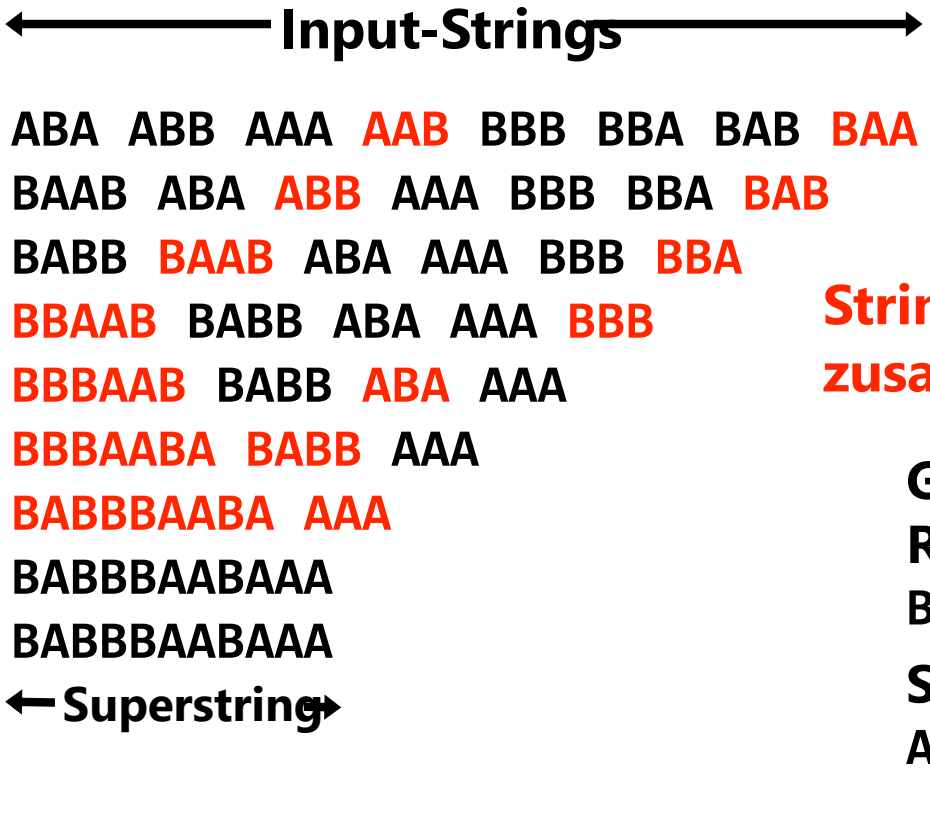
Dann geben wir es eben auf, den kürzestmöglichen Superstring zu finden

Neue Idee: finde kurze Superstrings mit Greedy-Algorithmus

1.) Wähle zwei Strings mit größtem Overlap und “merge” diese.
2.) Wiederhole Schritt 1.) bis nur noch ein String übrig ist

Shortest common superstring: Greedy

Greedy-SCS-Algorithmus in Aktion (l = 1):



Strings in rot werden zusammengefasst

Greedy-Resultat:
BABBBAABAAA

SCS:
AAABBBABAA

Zeilen = Iterationen, eine “merge”-Operation per Zeile.
Erste Spalte = Overlap-Länge in der vorigen Runde

Shortest common superstring: Greedy

Greedy-Algorithmus liefert natürlich nicht notwendigerweise einen SCS. Beweis?

Man kann aber zeigen (hier nicht), dass der Greedy-Algorithmus ein Approximationsalgorithmus mit Faktor 2.5 ist.

Shortest common superstring: Greedy

Input für Greedy-SCS: alle Substrings der Länge 6 des Strings `a_long_long_long_time`. $l = 3$.

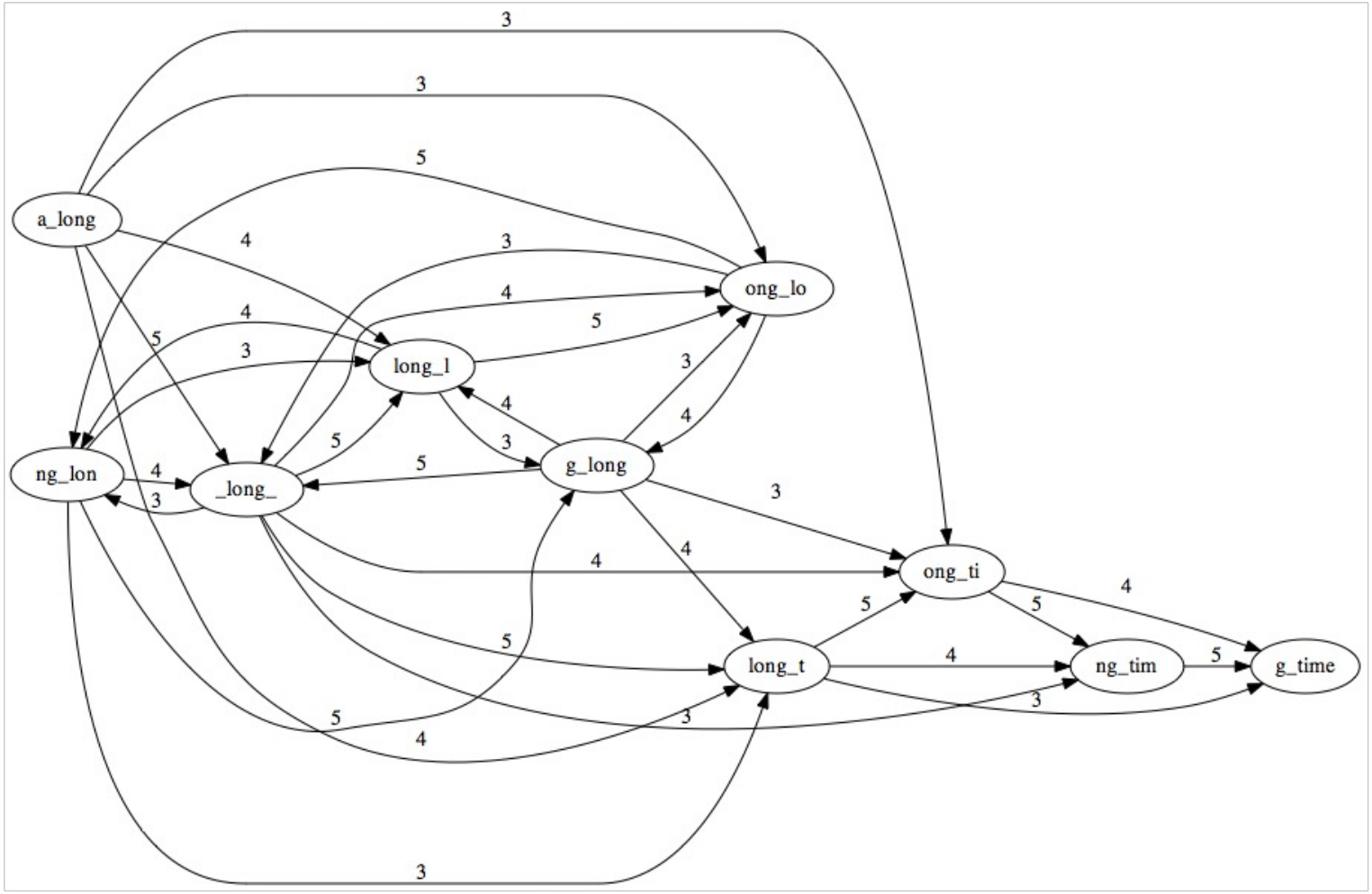
```
  ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long g_time ng_tim
5 ng_time ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long
5 ng_time g_long_ ng_lon a_long long_l ong_ti ong_lo long_t
5 ng_time long_ti g_long_ ng_lon a_long long_l ong_lo
5 ng_time ong_lon long_ti g_long_ a_long long_l
5 ong_lon long_time g_long_ a_long long_l
5 long_lon long_time g_long_ a_long
5 long_lon g_long_time a_long
5 long_long_time a_long
4 a_long_long_time
  a_long_long_time
```

Resultat: `a_long_long_time` **(ein** `_long` **fehlt)**

Was ist passiert?

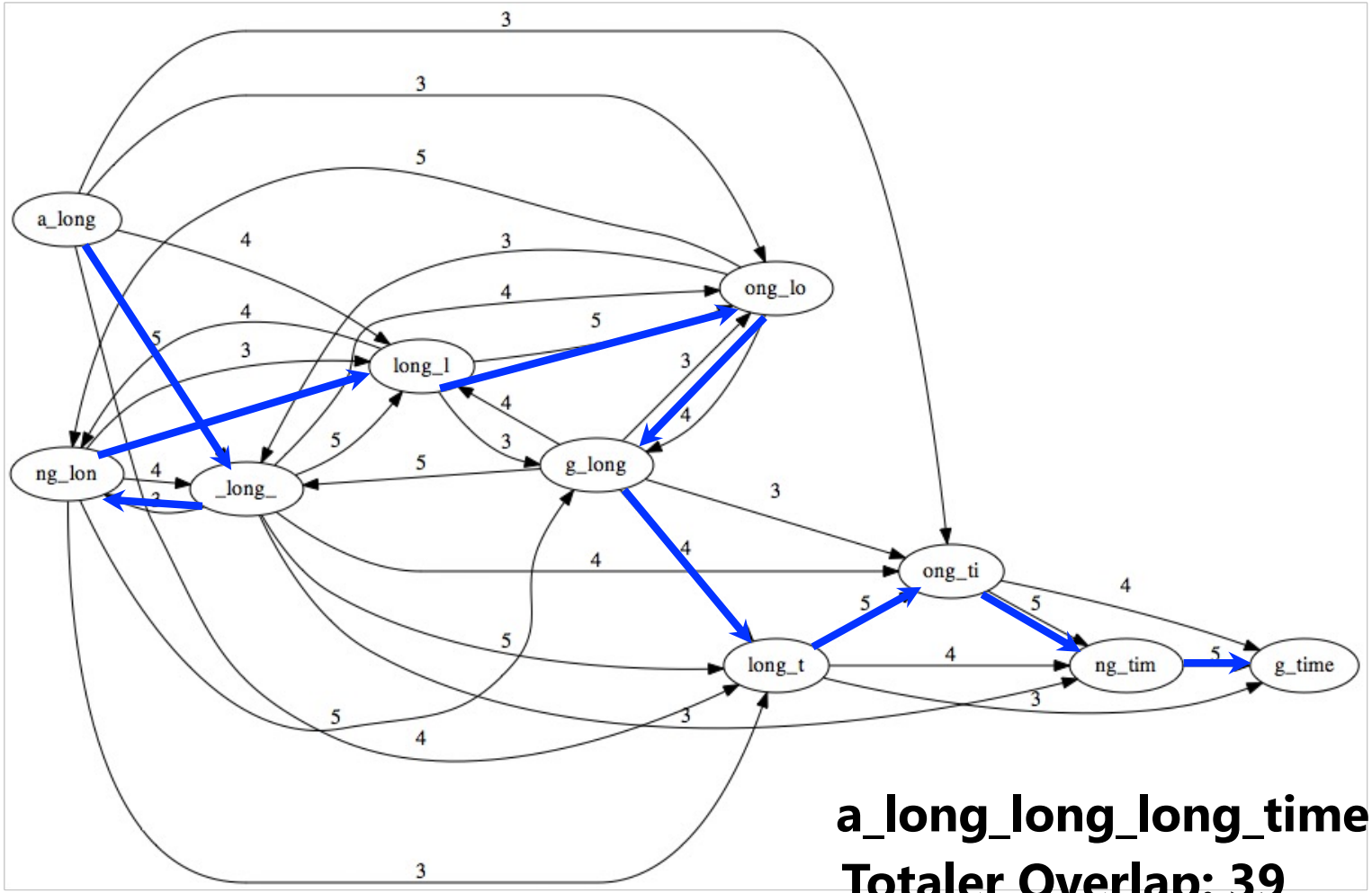
Shortest common superstring: Greedy

Hier ist der entsprechende Overlap-Graph (l = 3):



Shortest common superstring: greedy

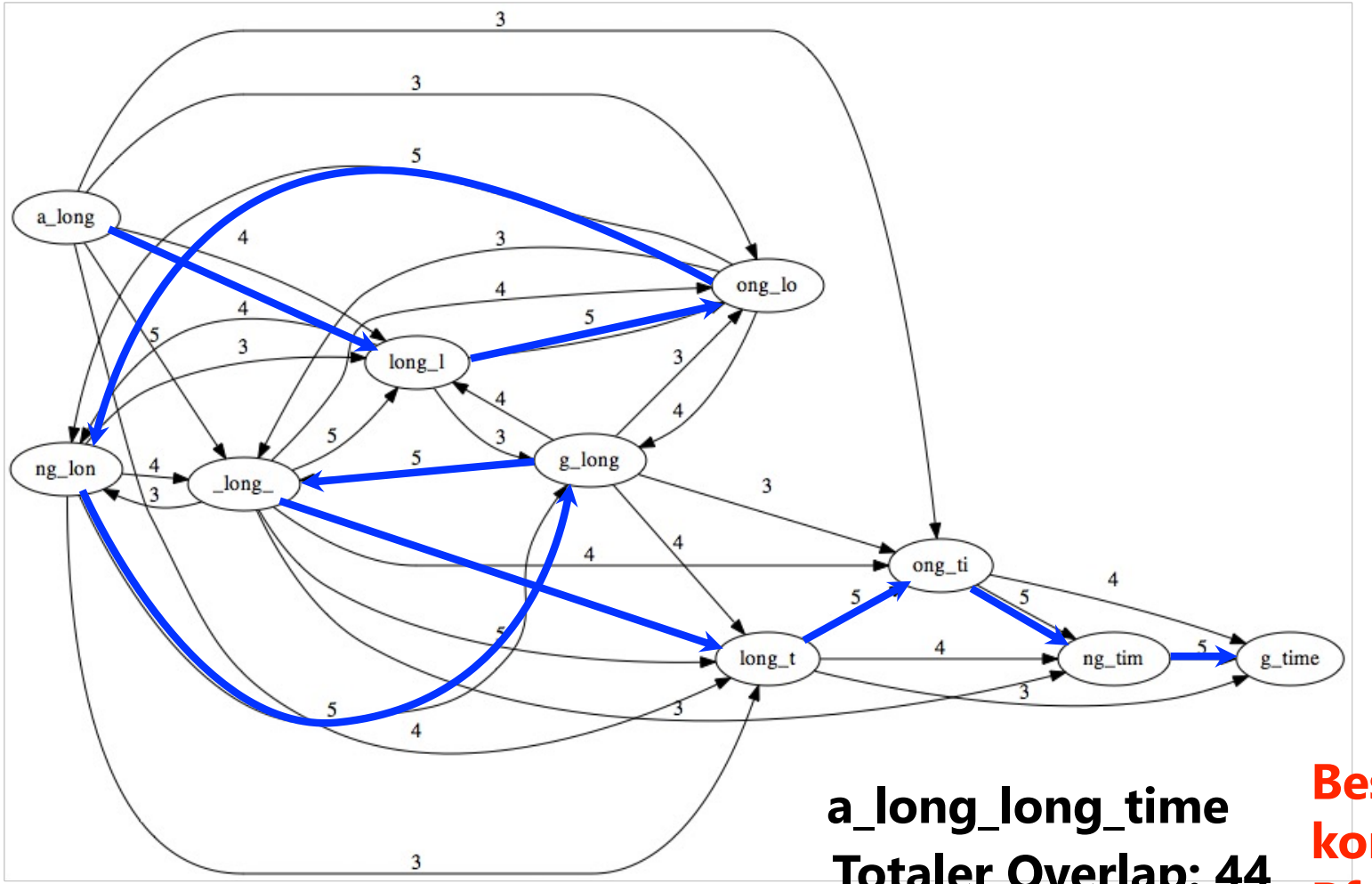
Hier ist der entsprechende Overlap-Graph (l = 3):



a_long_long_long_time
Totaler Overlap: 39

Shortest common superstring: greedy

Hier ist der entsprechende Overlap-Graph (l = 3):



a_long_long_time
Totaler Overlap: 44

Besser als korrekter Pfad!

Shortest common superstring: Greedy

Gleiches Beispiel, aber die Länge der Substrings ist jetzt 8, nicht 6

```
  long_lon ng_long_ _long_lo g_long_t ong_long g_long_l ong_time a_long_l _long_ti long_tim
7 long_time long_lon ng_long_ _long_lo g_long_t ong_long g_long_l a_long_l _long_ti
7 _long_time long_lon ng_long_ _long_lo g_long_t ong_long g_long_l a_long_l
7 _long_time a_long_lo long_lon ng_long_ g_long_t ong_long g_long_l
7 _long_time ong_long_ a_long_lo long_lon g_long_t g_long_l
7 g_long_time ong_long_ a_long_lo long_lon g_long_l
7 g_long_time ong_long_ a_long_lon g_long_l
7 g_long_time ong_long_l a_long_lon
7 g_long_time a_long_long_l
3 a_long_long_long_time
  a_long_long_long_time
```

Jetzt klappt es: a_long_long_long_time

Shortest common superstring: Greedy

Warum sind Substrings der Länge 8 gut genug, um die 3 Kopien von `long` aufzulösen?

a_long_long_long_time

g_long_l

Weil einer von ihnen alle drei `long`s umfasst.

Shortest common superstring: post mortem

SCS ist keine gute Formulierung für das Assembly-Problem

1.) Wir können das Problem nicht optimal in polynomieller Zeit lösen

→ Greedy-SCS. Resultat kann zu lang sein

2.) SCS kollabiert Repeat-Sequenzen fälschlicherweise

Resultat kann viel zu kurz sein (Genome sind hochrepetitiv)

3.) SCS ignoriert Sequenzierungsfehler und genomische Variabilität

Resultat kann falsch sein

Shortest common superstring: post mortem

SCS ist keine gute Formulierung für das Assembly-Problem

Wir brauchen Formulierungen, die (a) schnelle Algorithmen ermöglichen und (b) mit Repeat-Sequenzen und (c) Fehlern/kleinen Unterschieden vernünftig umgehen.

Bemerkung: Zu lange Repeat-Sequenzen vereiteln jeden Assembly-Versuch, wenn das Ziel ist, **einen** String zu produzieren. Das ist algorithmenunabhängig und liegt an Readlänge und der Struktur repetitiver Sequenzen

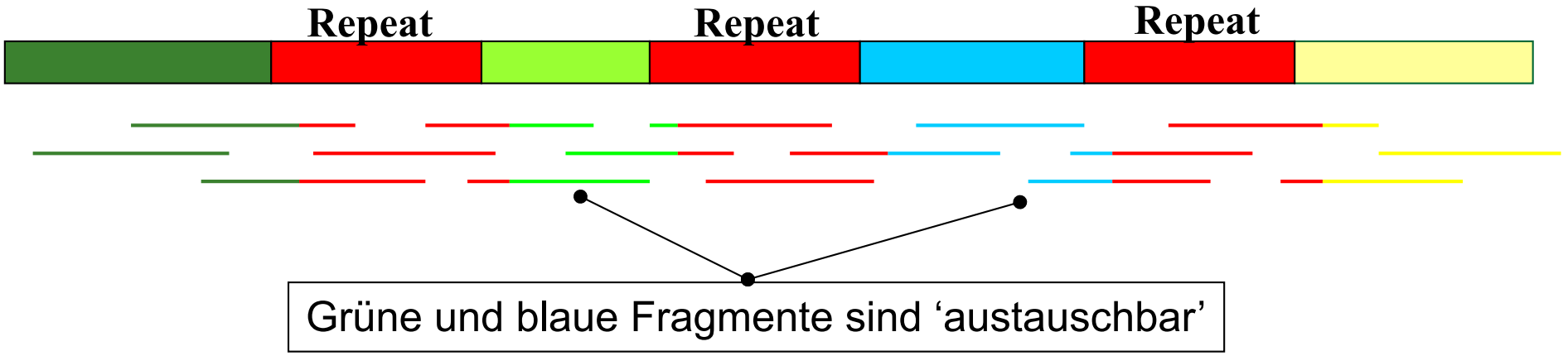
Repeats

Hauptproblem für Fragment-Assembly

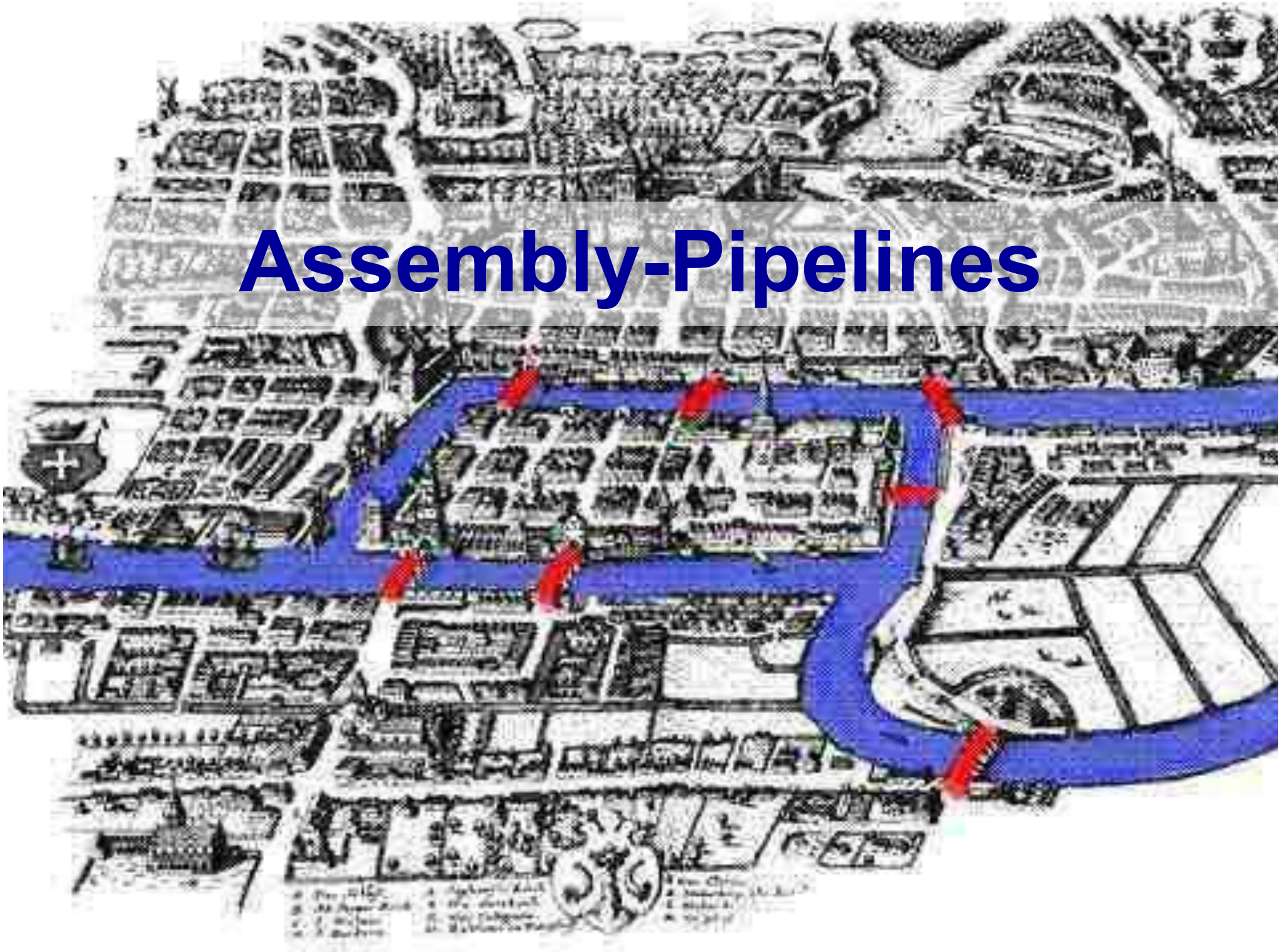
> 50% des menschlichen Genoms sind Repeats:

> 1 Million *Alu* Repeats (ca. 300 bp)

ca. 200.000 LINE Repeats (1000 bp und mehr)



Assembly-Pipelines



Wie sehen echte Assembly-Methoden aus?

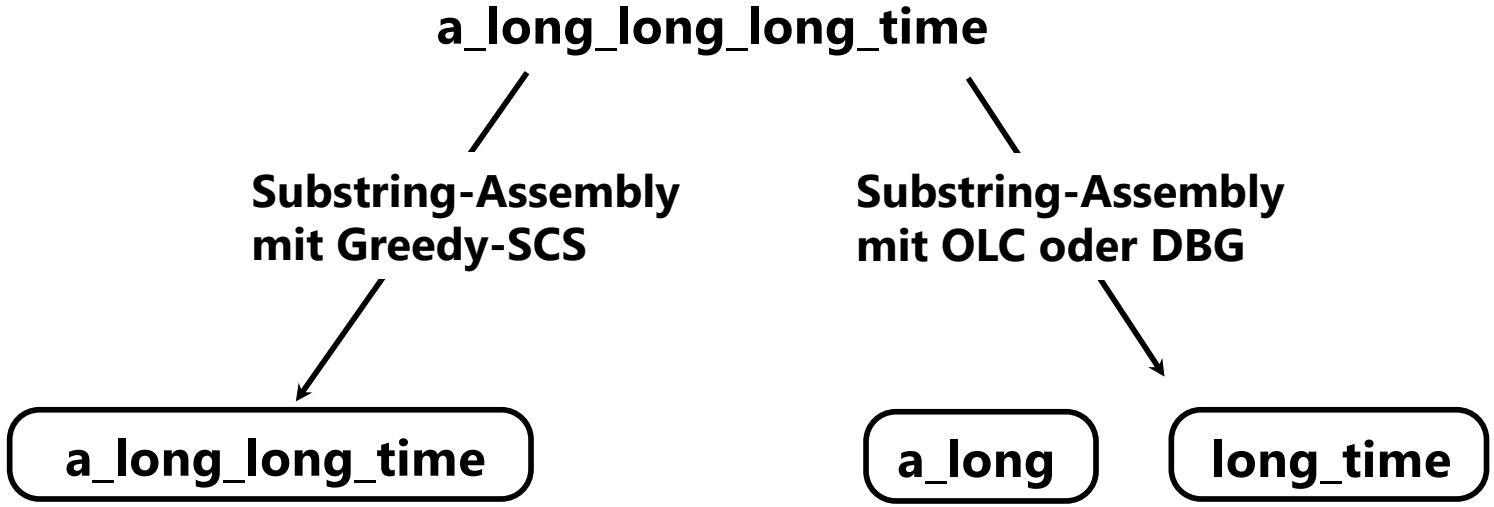
OLC: Overlap-Layout-Consensus-Assembly

DBG: De Bruijn graph-Assembly

Beide behandeln unauflösbare Repeats, indem sie sie weglassen

→ Assembly wird fragmentiert.

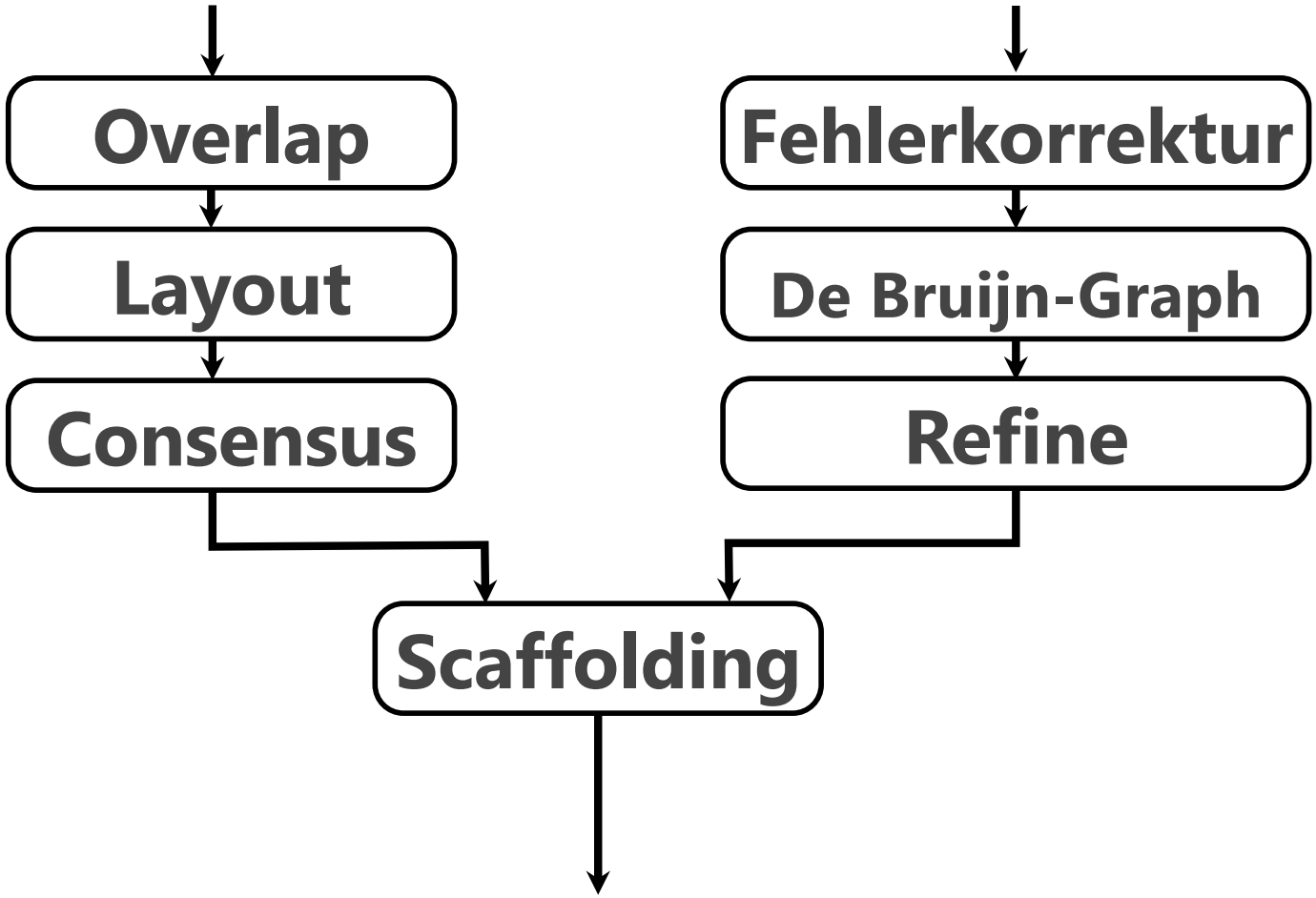
Fragmente = Contigs (für “contiguous")



Assembly-Alternativen

Alternative 1: Overlap-Layout-Consensus (OLC)-Assembly

Alternative 2: De Bruijn-Graph (DBG)-Assembly

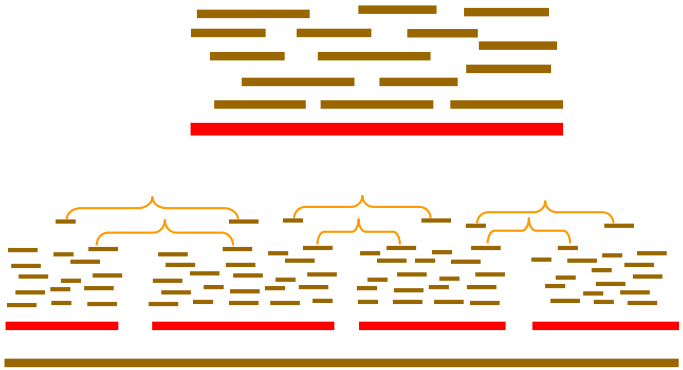


Pipeline: Overlap-Layout-Consensus

Programme: ARACHNE, PHRAP, CAP, TIGR, CELERA, …

Overlap: Finde überlappende Reads

Layout: Verbinde Reads zu Contigs, und Contigs zu Supercontigs



Consensus: Ermittle die DNA-Sequenz und korrigiere Fehler

..ACGATTACAATAGGTT..

Overlap-Phase

Wir wollen auch Mismatches und Gaps erlauben

→ Problem: wie finden wir das beste Alignment eines Suffixes von X zu einem Präfix von Y?

```
X: CTCGGCCCTAGG
      |||  |||||
Y:    GGCTCTAGGCCC
```

Dynamische Programmierung

Aber: Backtracking darf nur von Suffix von X zu einem Präfix von Y gehen. Wie machen wir das?

Overlaps mit Alignment

Finde das beste Suffix-Präfix-Alignment von X und Y

```
X: CTCGGCCCTAGG
       ||| |||||
Y:     GGCTCTAGGCCC
```

Ähnlich zu globalem Alignment:
(hier Minimierung, da Distanz)

$$D[i,j] = \min \begin{cases} D[i-1,j] + s(x[i-1], -) \\ D[i,j-1] + s(-, y[j-1]) \\ D[i-1,j-1] + s(x[i-1], y[j-1]) \end{cases}$$

		A	C	G	T	-
		0	4	2	4	8
A	0					
C	4					
G	2					
T	4					
-	8					

Aber wie erzwingen wir Präfix-Suffix-Matches?

Overlaps mit Alignment

Finde das beste Suffix-Präfix-Alignment von X und Y

$$D[i,j] = \min \begin{cases} D[i-1,j] + s(x[i-1], -) \\ D[i,j-1] + s(-, y[j-1]) \\ D[i-1,j-1] + s(x[i-1], y[j-1]) \end{cases}$$

	A	C	G	T	-
A	0	4	2	4	8
C	4	0	4	2	8
G	2	4	0	4	8
T	4	2	4	0	8
-	8	8	8	8	

Wie initialisieren wir die erste Zeile und Spalte?

Jedes Suffix von X ist möglich: erste Spalte = 0

Muss Präfix von Y sein:

Erste Zeile = ∞s

Backtrack von letzter Zeile zu li. Spalte

Y

X \ Y	-	G	G	C	T	C	T	A	G	G	C	C	C
-	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
C	0	4	12	20	28	36	44	52	60	68	76	84	92
T	0	4	8	14	2								
C	0	4	8	8	1								
G	0		4	12	1								
G	0	0		8	16	16	24	26	30	36	44	52	60
C	0	4	4		8	16	18	26	30	34	36	44	52
C	0	4	8	4	2	8	16	22	30	34	34	36	44
C	0	4	8	8	6	2	10	18	26	34	34	34	36
T	0	4	8	10	8	8	2	10	18	26	34	36	36
A	0	2	6	12	14	12	10	2	10	18	26	34	40
G	0	0	2	10	16	18	16	10	2	10	18	26	34
G	0	0	0	6	14	20	22	18	10		10	18	26

X: CTCGGCCCTAGG

Y: GGCTCTAGGCCC

Overlaps mit Alignment

Finde das beste Suffix-Präfix-Alignment von X und Y

$$D[i,j] = \min \begin{cases} D[i-1,j] + s(x[i-1], -) \\ D[i,j-1] + s(-, y[j-1]) \\ D[i-1,j-1] + s(x[i-1], y[j-1]) \end{cases}$$

	A	C	G	T	-
A	0	4	2	4	8
C	4	0	4	2	8
G	2	4	0	4	8
T	4	2	4	0	8
-	8	8	8	8	

Problem: sehr kurze Matches kriegen (zufällig) guten Score ...

...und verdecken relevantere Matches

Lösung: fordere minimale Overlaplänge l, z. B. l = 5

		-	G	G	C	T	C	T	A	G	G	C	C	C
	-	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
	C	0	4	12	20	28	36	44	52	60	68	76	84	92
	T	0	4	8	14	2								
	C	0	4	8	8	1								
	G	0	0	4	12	1								
	G	0	0	0	8	16	16	24	26	30	36	44	52	60
	C	0	4	4	0	8	16	18	26	30	34	36	44	52
X	C	0	4	8	4	2	8	16	22	30	34	34	36	44
	C	0	4	8	8	6	2	10	18	26	34	34	34	36
	T	0	4	8	10	8	8	2	10	18	26	34	36	36
	A	0	2	6	12	14	12	10	2	10	18	26	34	40
	G	0	0	2	10	16	18	16	10	2	10	18	26	34
	G	0	0	0	6	14	20	22	18	10	2	10	18	26

Y

X: CTCGGCCCTAGG
||| |||||
Y: GGCTCTAGGCCC

Overlaps mit Alignment

Finde das beste Suffix-Präfix-Alignment von X und Y

$$D[i,j] = \min \begin{cases} D[i-1,j] + s(x[i-1], -) \\ D[i,j-1] + s(-, y[j-1]) \\ D[i-1,j-1] + s(x[i-1], y[j-1]) \end{cases}$$

	A	C	G	T	-
A	0	4	2	4	8
C	4	0	4	2	8
G	2	4	0	4	8
T	4	2	4	0	8
-	8	8	8	8	

Setze zusätzliche Zellen auf ∞

Veränderte Werte in rot

Jetzt ist der relevanteste Match das beste Alignment

Y

X	-	G	G	C	T	C	T	A	G	G	C	C	C
-	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
C	0	4	12	20	28	36	44	52	60	68	76	84	92
T	0	4	8	14	2								4
C	0	4	8	8	1								5
G	0	0	4	12	1								3
G	0	0	0	8	16	16	24	26	30	36	44	52	60
C	0	4	4	0	8	16	18	26	30	34	36	44	52
C	0	4	8	4	2	8	16	22	30	34	34	36	44
C	∞	4	8	8	6	2	10	18	26	34	34	34	36
T	∞	4	8	10	8	8	2	10	18	26	34	36	36
A	∞	12	6	12	14	12	10	2	10	18	26	34	40
G	∞	20	12	10	16	18	16	10	2	10	18	26	34
G	∞	∞	∞	∞	∞	20	22	18	10	2	10	18	26

X: CTCGGCCCTAGG

Y: GGCTCTAGGCCC

Überlappungs-Alignment: Laufzeit

- *d* reads der Länge *n* → Gesamtlänge *N* = *dn*

Anzahl Vergleiche:	$O(d^2)$
Größe jeder DP-Matrix:	$O(n^2)$
Insgesamt:	$O(d^2n^2) = O(N^2)$

- Nicht mehr machbar für große *d* → Heuristik

Überlappungs-Alignment: Heuristik

- **Reduzierung der Vergleiche: Filtere Paare aus, die wahrscheinlich keinen signifikanten Overlap haben**
- Ein “*k*-mer” ist ein Substring der Länge k
- Sortiere alle *k*-mere in Reads ($k \sim 24$)
- Finde Read-Paare mit gleichem *k*-mer
- Erweitere nach beiden Seiten zu vollem Alignment – ungültig wenn Überlappung < 95% identisch

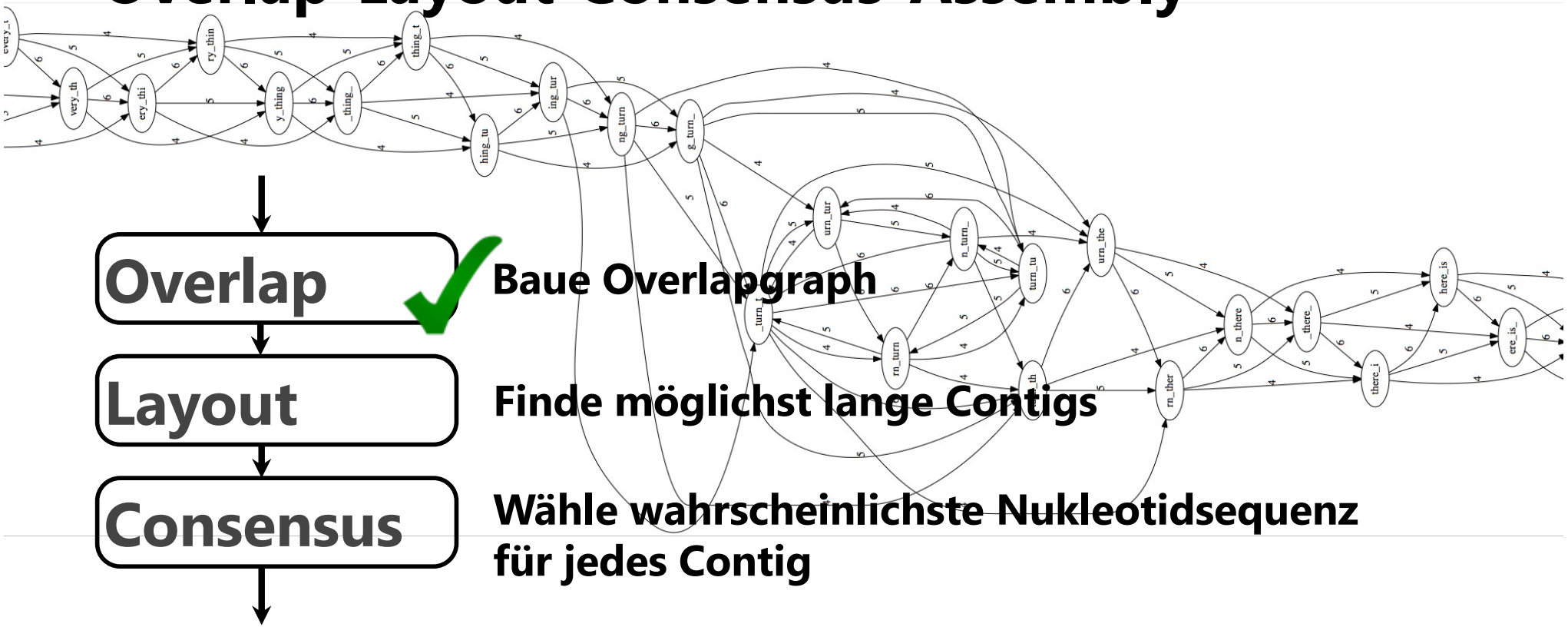
hier: $k = 17$

Überlappende Reads und Repeats

- Ein *k*-mer mit *r* Kopien verursacht r^2 Überlappungen mit sich selbst
- Für *Alu* mit 10^6 Kopien → 10^{12} Überlappung mit sich selbst - zu viele
- **Lösung:**

  Ignoriere alle *k*-mere mit mehr als *t* × Coverage Kopien (*t* ~ 10)

Overlap-Layout-Consensus-Assembly



Overlap ✓ **Baue Overlapgraph**

Layout **Finde möglichst lange Contigs**

Consensus **Wähle wahrscheinlichste Nukleotidsequenz für jedes Contig**

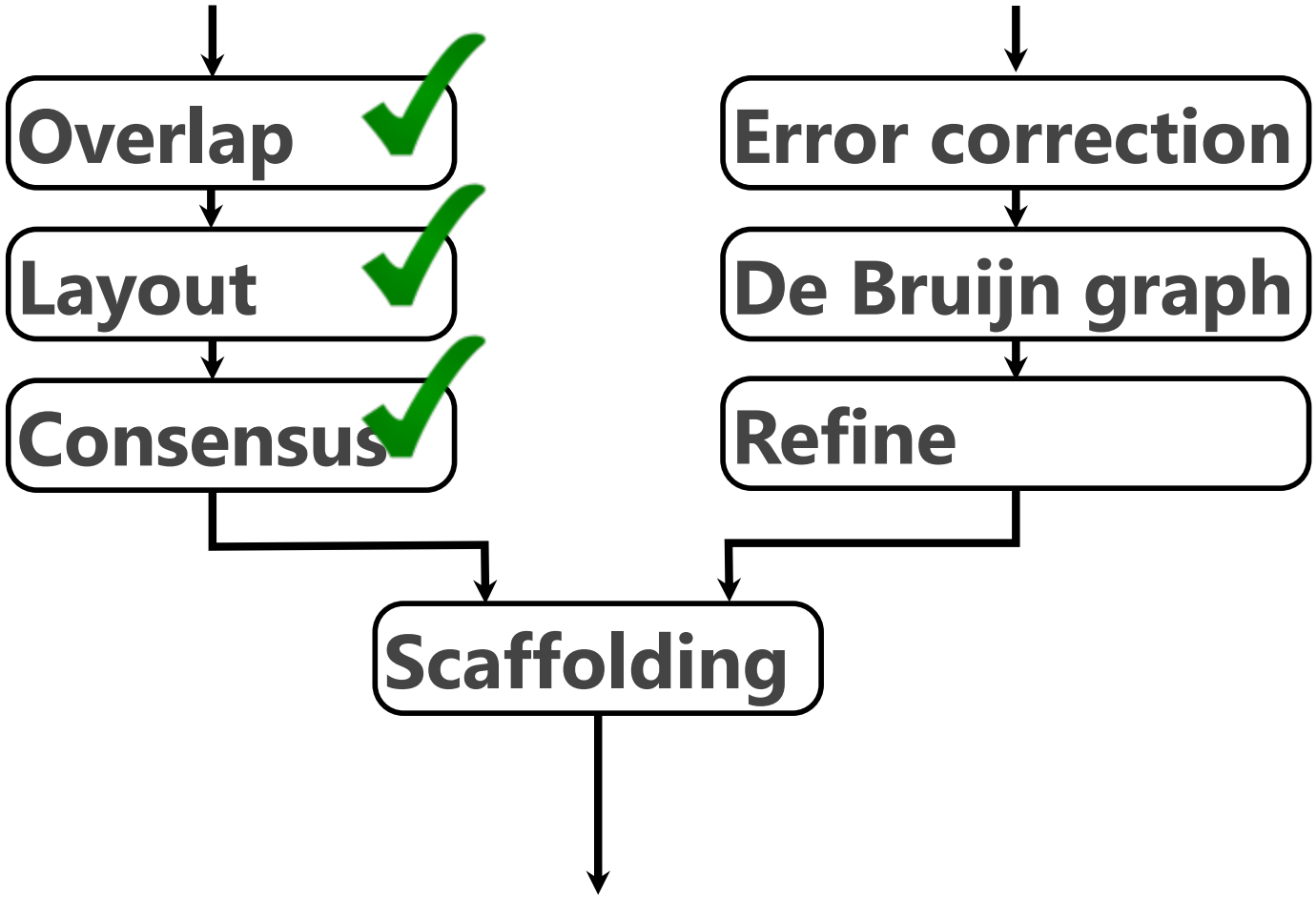
Nachteile:

- **Konstruktion des Overlap-Graphen langsam**
- **Overlap-Graph ist groß (ein Knoten pro Read, Anzahl der Kanten wächst superlinear mit Anzahl Reads)**
- **Milliarden von Reads → Problem**

Assembly-Alternativen

Alternative 1: Overlap-Layout-Consensus (OLC)-Assembly

Alternative 2: De Bruijn graph (DBG)-Assembly



k-mer

Ein “k-mer” ist ein Substring der Länge k

S: GGCGATTCATCG

Ein 4-mer von S: ATTC

Alle 3-mere von S:

GGC
GCG
CGA
GAT
ATT
TTC
TCA
CAT
ATC
TCG

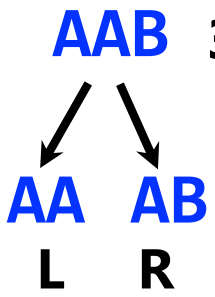
mer: *griech.*: “Teil”

De Bruijn-Graph

Input: Reads (~ Substrings einer genomischen Sequenz)

AAA, AAB, ABB, BBB, BBA

AAB ist ein k-mer (k = 3). AA ist sein linkes (k-1)-mer, AB ist sein rechtes (k



AABs linkes 2-mer AABs rechtes 2-mer

De Bruijn-Graph

Nimm alle k-mere des Inputs und splitte sie in zwei sich überlappende Substrings der Länge 2 (linke und rechte (k-1)-mere)

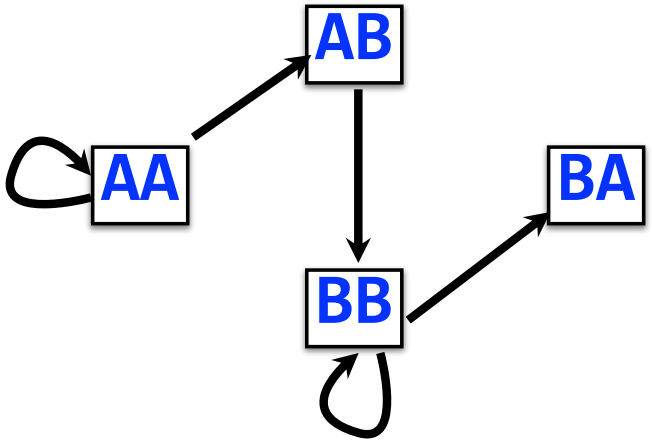
AAABBBA

alle 3-mere: AAA, AAB, ABB, BBB, BBA

baue L/R 2-mere: AA, AA, AA, AB, AB, BB, BB, BB, BB, BA

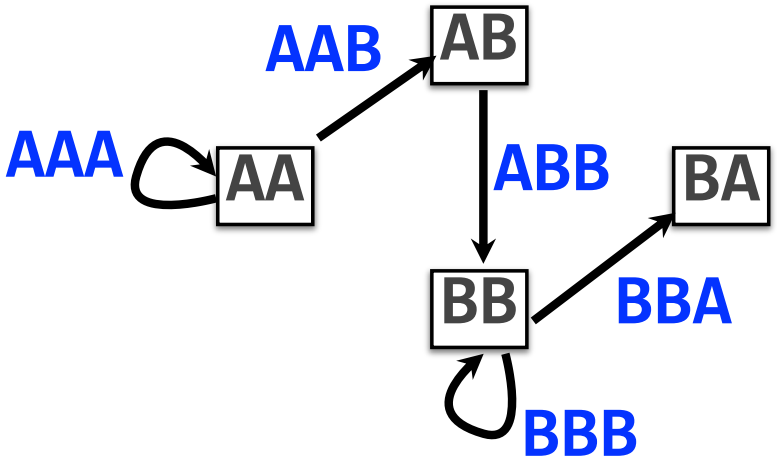
L R L R L R L R L R

Die (k-1)-mere werden Knoten im De Bruijn-Graph. Gerichtete Kanten laufen von den linken zu den rechten (k-1)-meren:



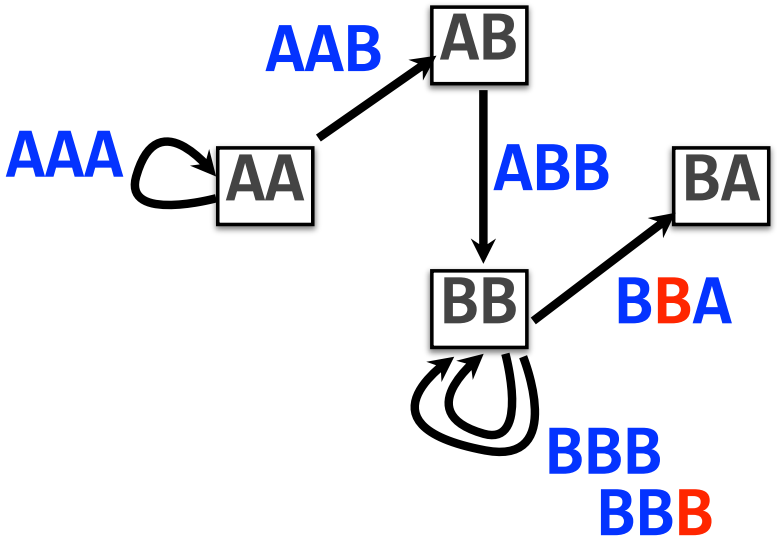
Jede Kante in diesem Graph entspricht einem k-mer des Inputs.

De Bruijn-Graph



Kante = overlap (der Länge k-2) zweier (k-1)-mere = k-mer des Inputs.

De Bruijn-Graph



Wenn wir ein B zu unserem Input hinzufügen (AAABBBBA), bekommen wir eine Multikante.

De-Bruijn-Graph ist also Multigraph.

Erinnerung: Eulerzyklen und -wege

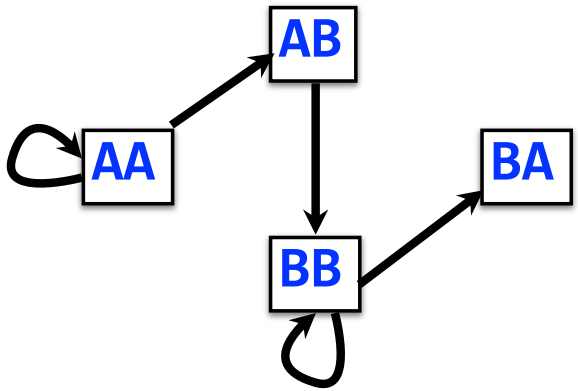
Knoten balanciert ⇔ Eingangsgrad = Ausgangsgrad

Knoten halb-balanciert ⇔ |Eingangsgrad – Ausgangsgrad| = 1

Ein Eulerweg besucht jede Kante genau einmal.

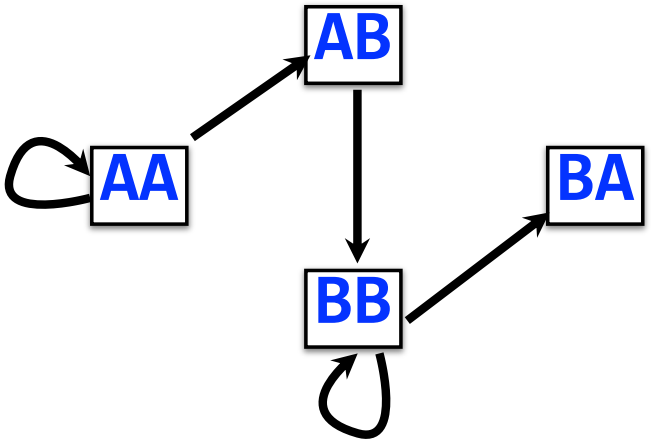
Nicht alle Graphen erlauben Eulerwege.

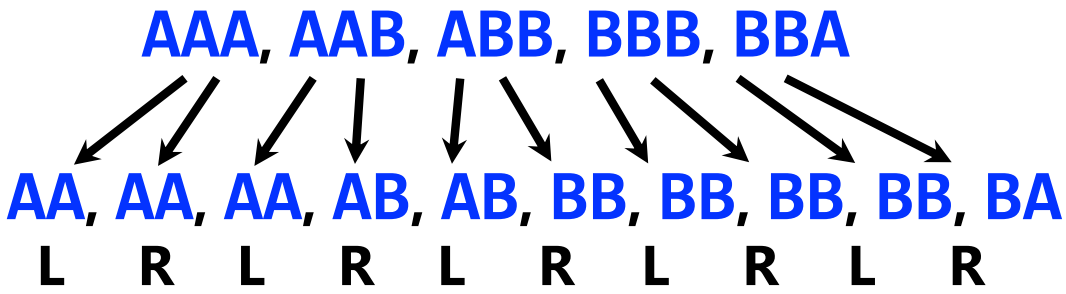
Ein gerichteter zusammenhängender Graph erlaubt einen Eulerweg ⇔ er hat höchstens zwei halb-balancierte Knoten und ansonsten nur balancierte Knoten.



De Bruijn-Graph

Zurück zum De Bruijn-Graphen



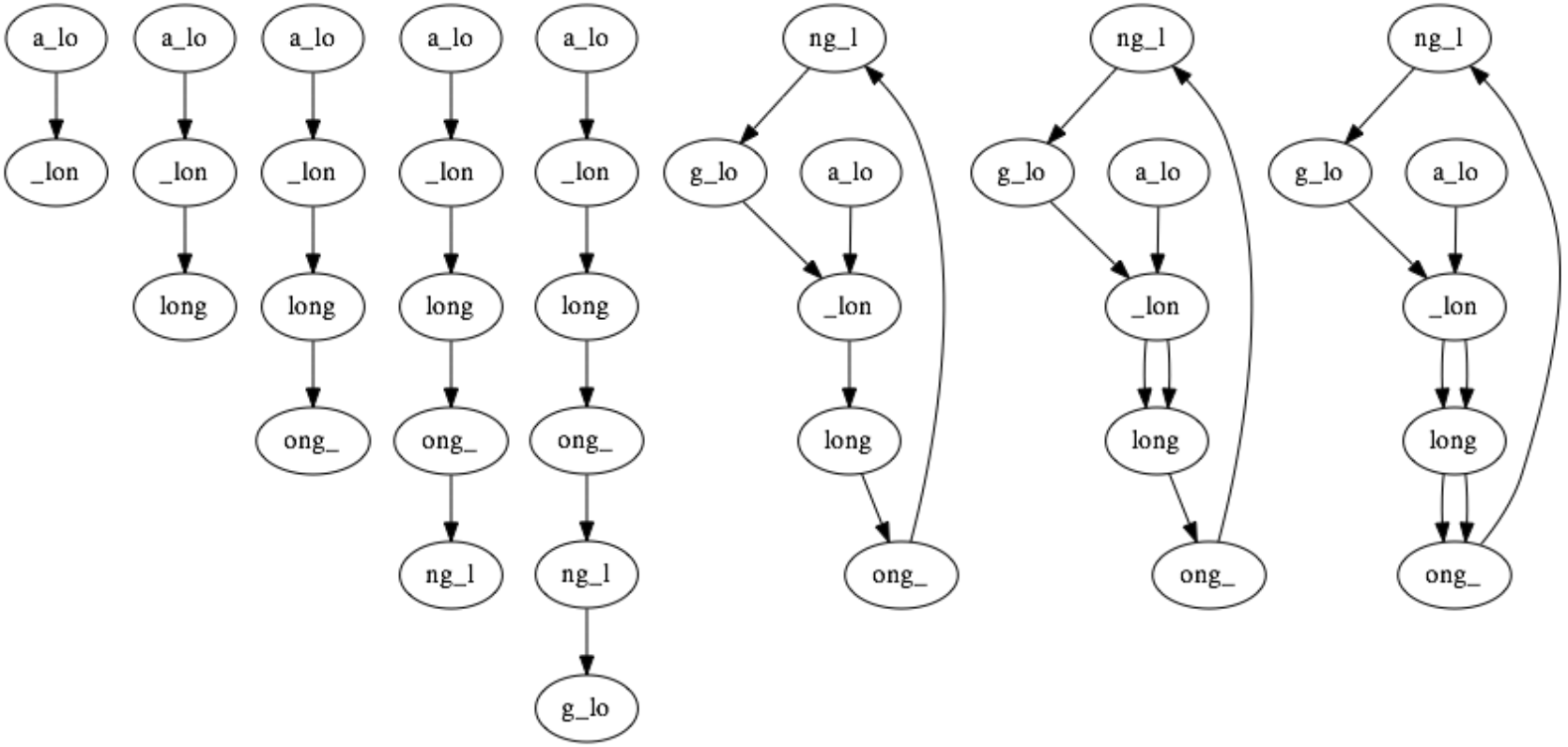


Ist er Eulersch? **Ja**

Argument 1: AA → AA → AB → BB → BB → BA

Argument 2: AA **und** BA **sind halb-balanciert,** AB **und** BB **sind balanciert**

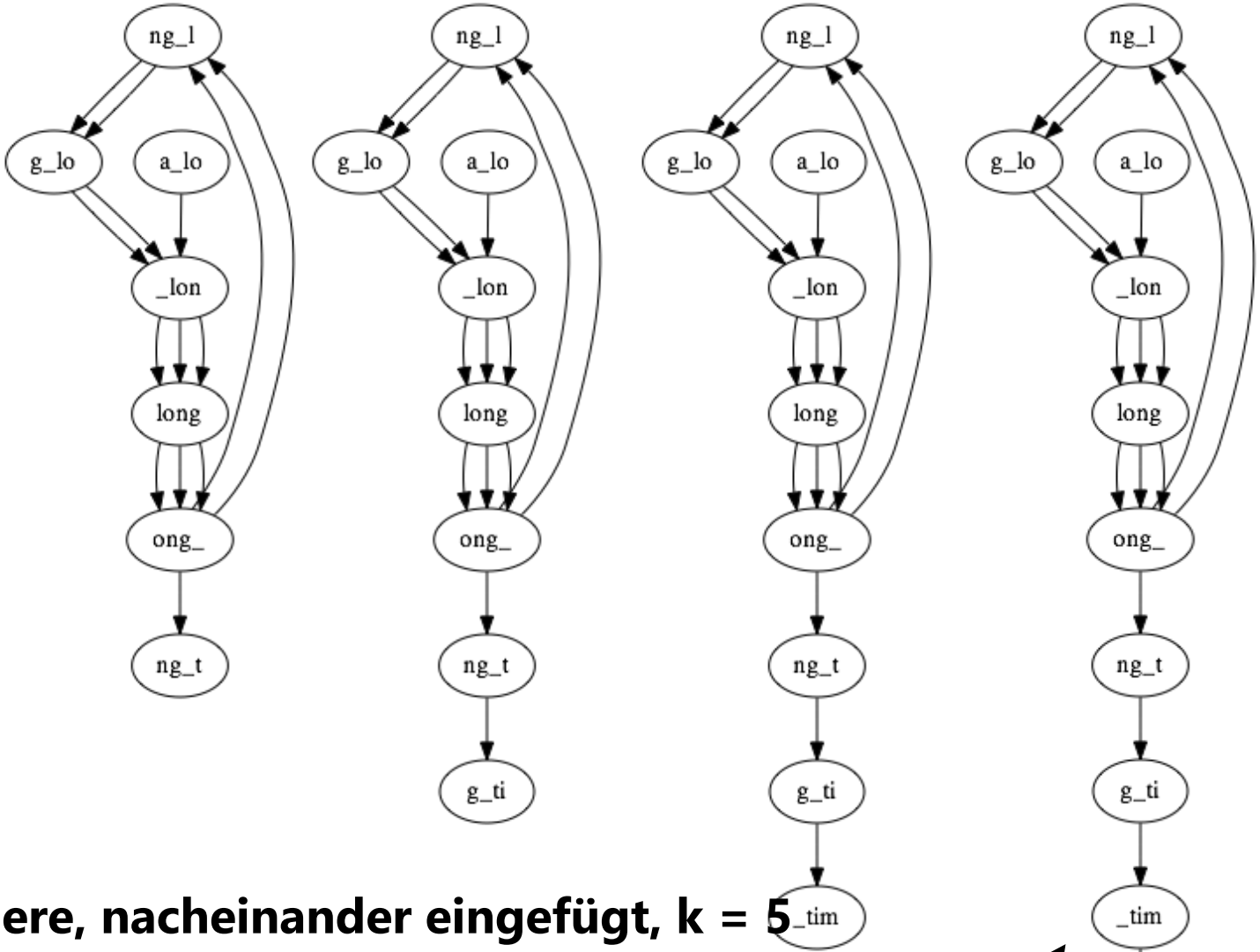
De Bruijn-Graph



Erste 8 k-mere, nacheinander eingefügt, k = 5

`a_long_long_long_time`

De Bruijn-Graph



Letzte 5 k-mere, nacheinander eingefügt, k = 5

`a_long_long_long_time`

De Bruijn-Graph

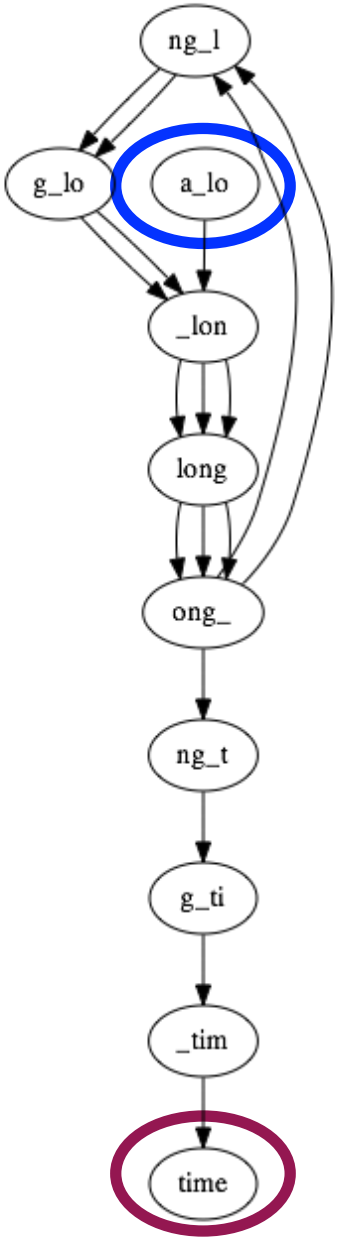
De Bruijn-Graph

Fehlerlose Sequenzierung → De Bruijn-Graph Eulersch. Warum?

Knoten für k-1-mer vom linken Ende ist halb-balanciert

Knoten für k-1-mer vom rechten Ende ist halb-balanciert

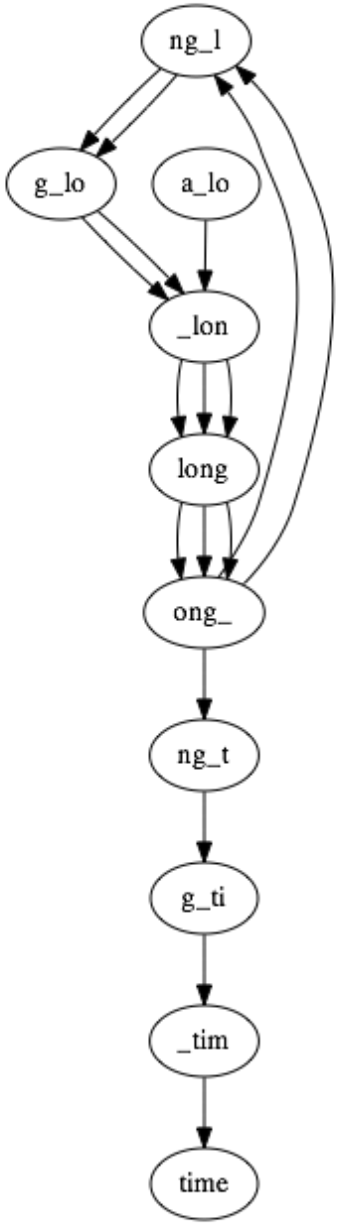
Die anderen Knoten sind balanciert, denn jedes (k-1)-mer taucht gleichoft als linkes und rechtes (k-1)-mer auf



De Bruijn-Graph

Perfekte Sequenzierung → De Bruijn-Graph Eulersch → Finde Eulerweg in linearer Zeit.

Entspricht ein Eulerweg immer dem genomischen Input?



De Bruijn-Graph

Nein: Der Graph kann mehrere Eulerwege haben, von denen nur einer dem Input entspricht

Rechts: Graph für ZABCDABEFABY, k = 3

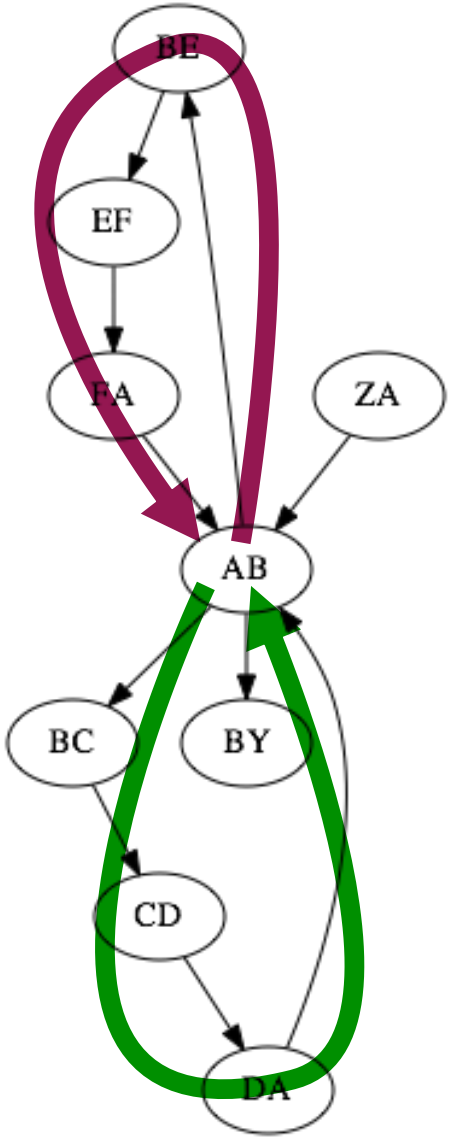
Alternative Eulerwege:

ZA → AB → BE → EF → FA → AB → BC → CD → DA → AB → BY

ZA → AB → BC → CD → DA → AB → BE → EF → FA → AB → BY

Diese entsprechen zwei kantendisjunkten gerichteten Kreisen, die am Knoten AB verbunden sind

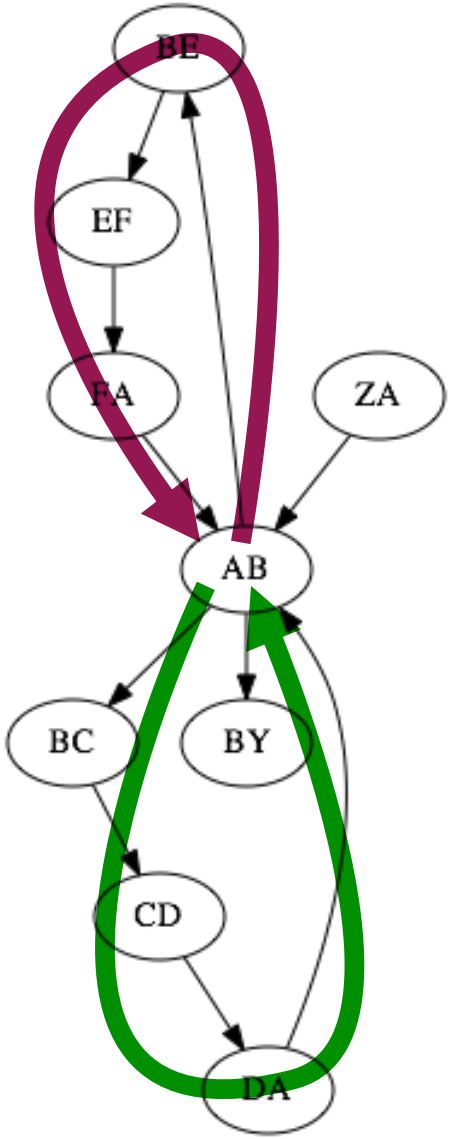
AB ist ein Repeat: ZABCDABEFABY



De Bruijn-Graph

Also lösen de Bruijn-Graphen nicht alle unsere Probleme

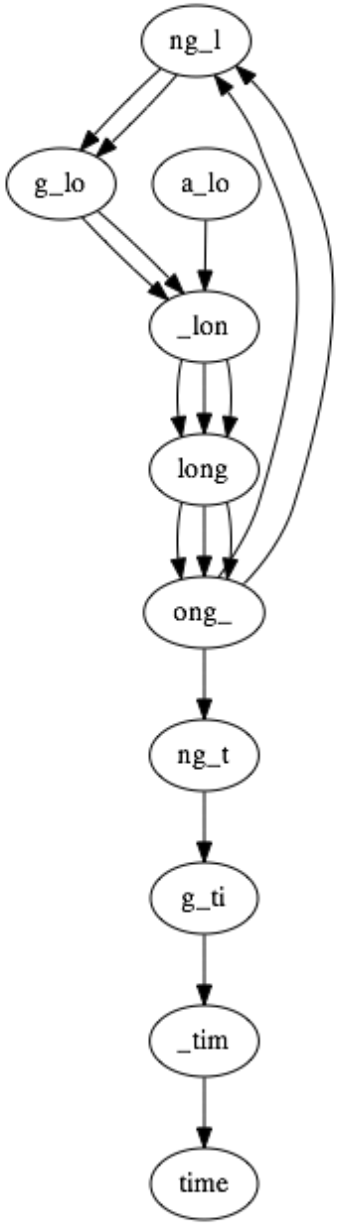
Es kommt noch schlimmer, wenn wir statt perfekter realistische Sequenzierung betrachten…



De Bruijn-Graph

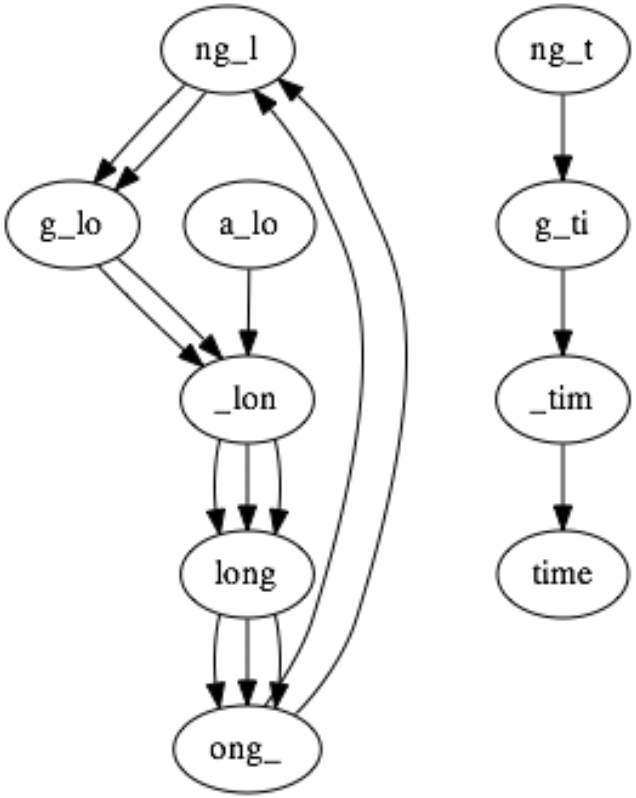
Lücken in der coverage → unzusammenhängender Graph

Graph für `a_long_long_long_time`, k = 5:



De Bruijn-Graph

Graph für `a_long_long_long_time`, k = 5 aber ohne `ong_t` :



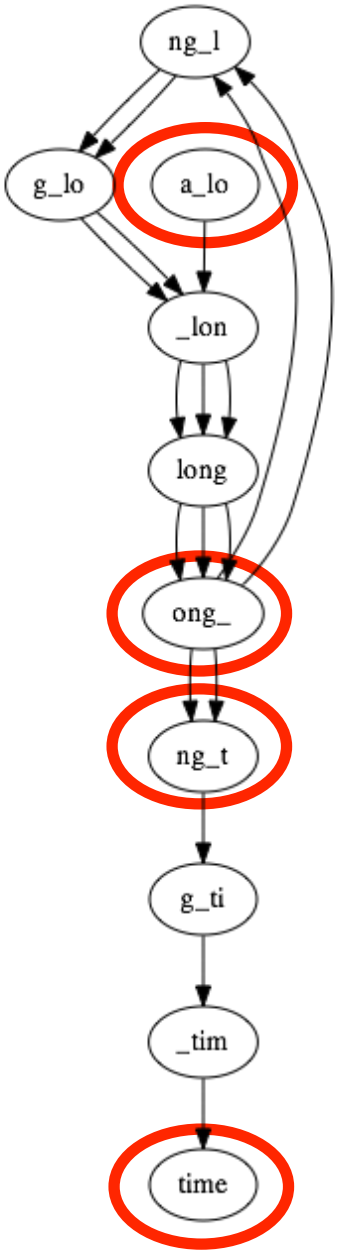
Zusammenhangskomponenten sind Eulersch, ganzer Graph nicht mehr

De Bruijn-Graph

Unterschiede in der coverage → auch nicht mehr Eulersch

Graph für `a_long_long_long_time`, k = 5 mit Extrakopie von `ong_t` :

→ 4 halbbalancierte Knoten
→ nicht Eulersch

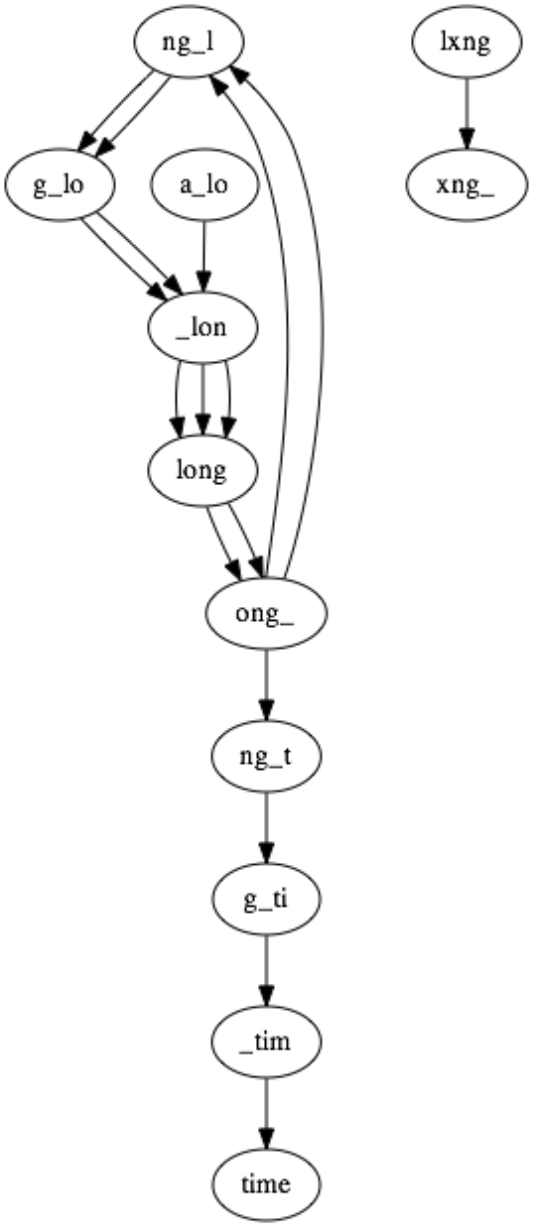


De Bruijn-Graph

Sequenzierfehler und Ploidie → auch nicht mehr Eulersch

Graph für `a_long_long_long_time`, k = 5 aber mit Fehler (`long_` → `lxng_`)

Graph ist nicht zusammenhängend, größte Komponente nicht mehr Eulersch



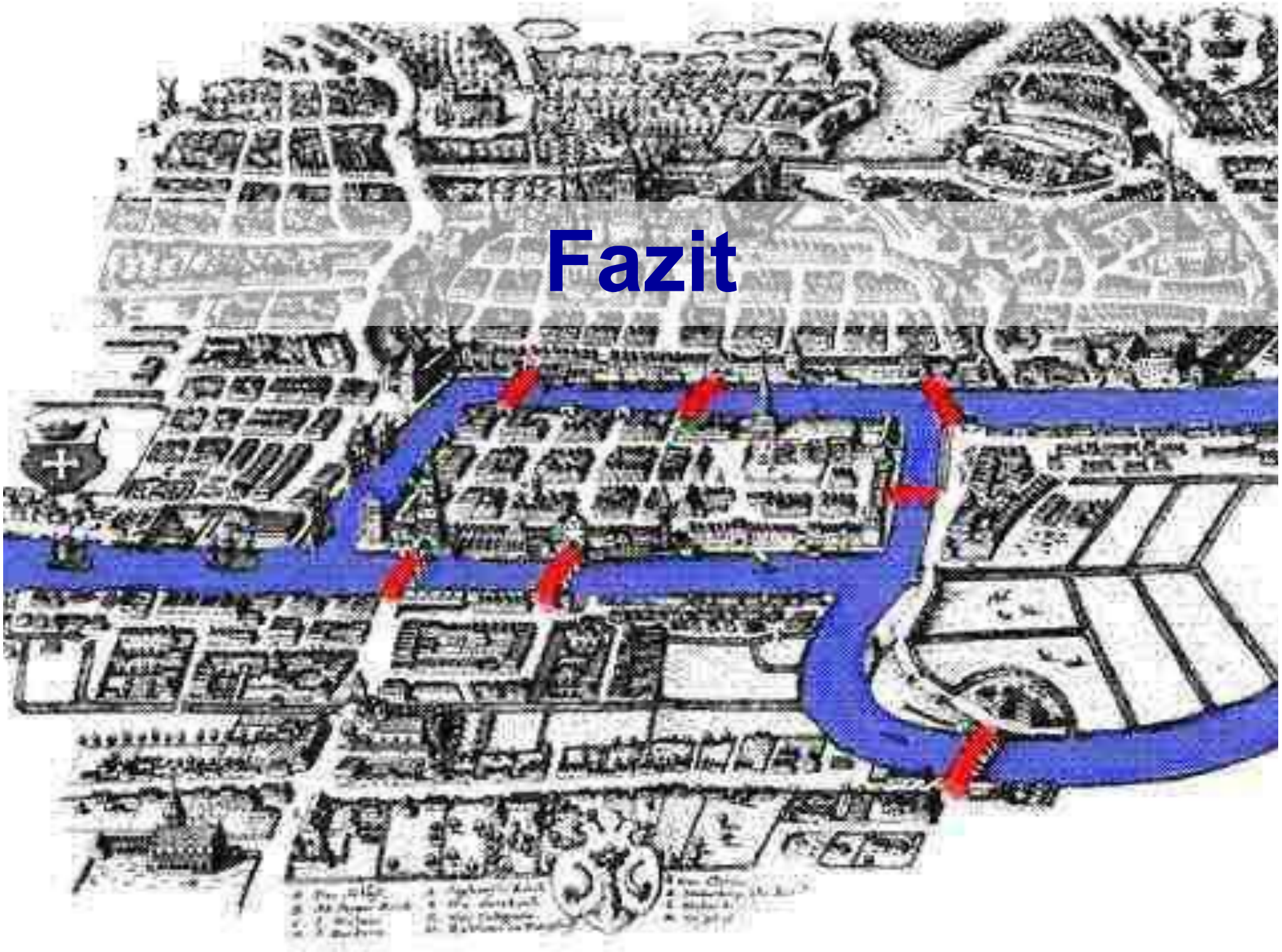
De Bruijn-Graph

Assembly als Eulerweg zu modellieren scheint elegant, ist aber leider nicht praktikabel

- **Ungleiche coverage, Sequenzierfehler etc. → dbG nicht mehr Eulersch**
- **Selbst wenn: Repeats führen zu vielen möglichen Eulerwegen**

In der Praxis betrachtet man eine Problemvariante
Leider auch NP-schwer! → Heuristiken.

Fazit



Praxis

- Beide Ansätze werden verwendet (OLC und dBG)
- Bei beiden: noch viel mehr Aspekte, die wir weggelassen haben
- Overlap-Graph hat bessere Auflösungsmöglichkeiten
- de Bruijn-Graph kleiner als Overlap-Graph
  - besser für großen Input