Karlsruher Institut für Technologie

Proseminar „Die P≠NP-Vermutung“
Space Complexity, Savitch‘s Theorem, PSPACE-completeness

Sommersemester 2012

INSTITUT FÜR THEORETISCHE INFORMATIK

KIT – Universität des Landes Baden-Württemberg und
nationales Forschungszentrum in der Helmholtz-Gemeinschaft

Simon Veith und Dominik Wagner

Gliederung

1. Kurze Wiederholung: Zeitkomplexitätsklassen
2. Space Complexity
    1. Klassenhierachie
    2. Beispielprobleme
3. Vollständigkeit
    1. NL-Vollständigkeit
    2. PSPACE-Vollständigkeit
4. Savitch‘s Theorem
    1. Konfigurationsgraphen
5. Probabilistische Algorithmen

Was bisher geschah

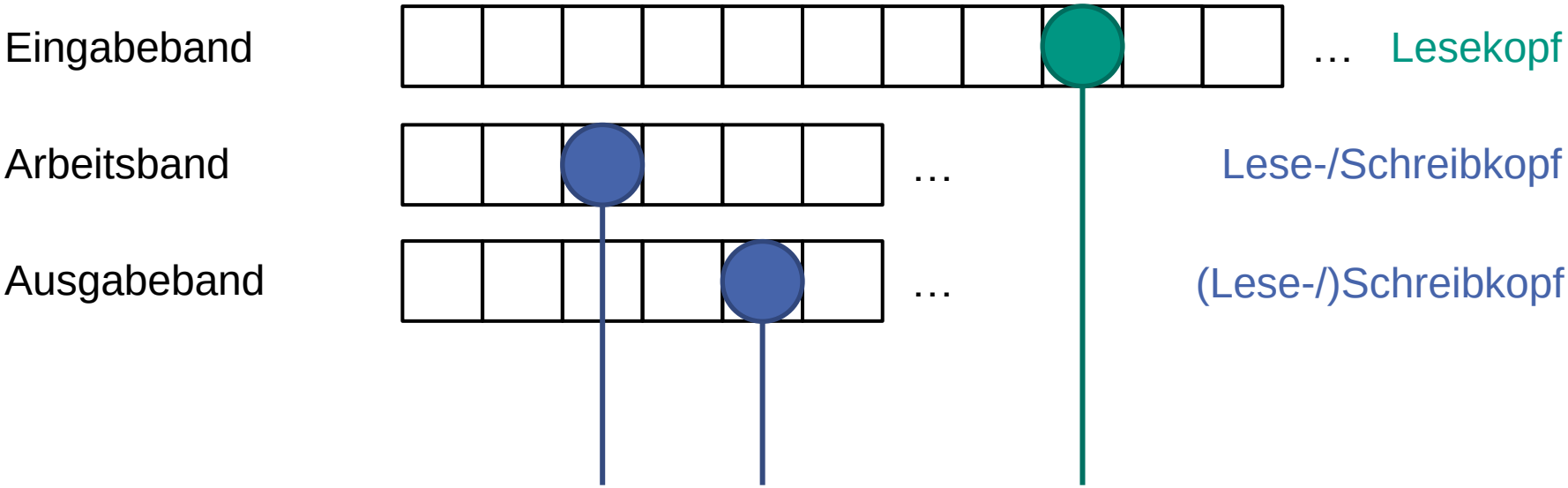
- Ein gängiges Komplexitätsmaß ist für uns die **Laufzeit** von Turing-Maschinen
- Gegeben eine Eingabe x mit Länge n
- Wie „lange“ läuft die Turing-Maschine bei dieser Eingabe?
- Was heißt „lange“? Wie messen wir das?
  - Anzahl Zustandsübergänge
  - Anzahl Bandbewegungen
- Wir definieren eine Zeitfunktion $T(n)$, die uns dies angibt
  - $T(n) = T(|x|) =$ Anzahl der Konfigurationen, die eine TM durchläuft, bis sie hält

Wiederholung: Zeitkomplexitätsklassen

- Klasse $\boldsymbol{DTIME}(T(n))$
  - Alle Sprachen, die von einer **deterministischen** Turing-Maschine in Zeit $O(T(n))$ entschieden werden
- Klasse $\boldsymbol{NTIME}(T(n))$
  - Alle Sprachen, die von einer **nicht**deterministischen Turing-Maschine in Zeit $O(T(n))$ entschieden werden
- Klasse $\boldsymbol{P}$
  - $\mathrm{U}_{c\geq 1}\, \boldsymbol{DTIME}(n^c)$
  - „Alle Sprachen, die von deterministischen TM in polynomieller Zeit entschieden werden“
- Klasse $\boldsymbol{NP}$ analog

Space Complexity

- Statt der **Laufzeit** einer Turing-Maschine kann man auch ihren **Platzbedarf** betrachten
  - Also: Anzahl der bei einer Berechnung besuchten Zellen
- Sinnvoll hierbei: Betrachtung einer *Mehrband*-Turing-Maschine
  - Eingabeband zählt *nicht* zum Platzbedarf
  - Erlaubt differenziertere Analyse
  - Nicht mächtiger als Einband-TM



Definitionen

- $\boldsymbol{SPACE}(S(n))$
  - Gegeben: S: $\mathbb{N} \to \mathbb{N}$, Sprache $L \subseteq \{0,1\}^*$ und eine Turing-Maschine M, die L entscheidet
  - Wenn eine Konstante c existiert, sodass M höchstens $c \cdot S(n)$ Zellen auf ihrem Arbeitsband besucht, dann liegt L in $\boldsymbol{SPACE}\big(S(n)\big)$.
- $\boldsymbol{NSPACE}\big(S(n)\big)$
  - Gegeben: S: $\mathbb{N} \to \mathbb{N}$, Sprache $L \subseteq \{0,1\}^*$ und eine Nichtdeterministische Turing-Maschine M, die L entscheidet
  - Wenn eine Konstante c existiert, sodass M unabhängig von ihren nichtdeterministischen Entscheidungen nie mehr als $c \cdot S(n)$ nichtleere Bandzellen bei Eingabelänge n verwendet, dann liegt L in $\boldsymbol{NSPACE}\big(S(n)\big)$.

Klassenhierarchie

- Eine TM kann in jedem Schritt nur eine Bandzelle besuchen
  - Bei $S(n)$ Schritten können also höchstens $S(n)$ Stellen genutzt werden
  - Also gilt $\boldsymbol{DTIME}\big(S(n)\big) \subseteq \boldsymbol{SPACE}\big(S(n)\big)$

Klassenhierarchie

- Zu einer DTM D, die mit einem Platzbedarf von $S(n)$ eine Lösung zu einem Problem findet, gibt es eine NDTM, die mit dem selben Platzbedarf ebenfalls eine Lösung zu dem Problem findet
  - Betrachte D als NDTM mit „Null-Orakel“
  - Also ist jedes Problem aus $\boldsymbol{SPACE}(S(n))$ auch mit einer NDTM bei gleichem Platzbedarf lösbar
  - Also gilt $\boldsymbol{SPACE}(S(n)) \subseteq \boldsymbol{NSPACE}(S(n))$
- Somit $\boldsymbol{DTIME}(S(n)) \subseteq \boldsymbol{SPACE}(S(n)) \subseteq \boldsymbol{NSPACE}(S(n))$.

Klassenhierarchie

- Der Laufzeitbedarf einer TM kann wesentlich höher sein als ihr Platzbedarf
  - Bandzellen können beliebig oft wiederverwertet werden
  - Beispiel: TM, die mit dem Arbeitsband einen binären Zähler realisiert
  - Zahlenbereich $[0, 2^{S(n)} - 1]$
  - Laufzeit $O(2^{S(n)})$
  - Vermutung: $\boldsymbol{SPACE}(S(n)) \subseteq \boldsymbol{DTIME}(2^{O(S(n))})$
- Somit ergibt sich
  - $\boldsymbol{DTIME}(S(n)) \subseteq \boldsymbol{SPACE}(S(n)) \subseteq \boldsymbol{NSPACE}(S(n)) \subseteq \boldsymbol{DTIME}(2^{O(S(n))})$

[Arora et al. ‘09]

Betrachtete Klassen

- Es ergeben sich einige interessante Platzkomplexitätsklassen
  - $\bigcup_{c>0}$ $\boldsymbol{SPACE}(n^c) = \boldsymbol{PSPACE}$
  - $\bigcup_{c>0}$ $\boldsymbol{NSPACE}(n^c) = \boldsymbol{NPSPACE}$
  - $\boldsymbol{SPACE}(\log n) = \boldsymbol{L}$
  - $\boldsymbol{NSPACE}(\log n) = \boldsymbol{NL}$
- Zu diesen gibt es sehr ähnliche Zeitkomplexitätsklassen
  - $\bigcup_{c>0}$ $\boldsymbol{DTIME}(n^c) = \boldsymbol{P}$
  - $\bigcup_{c>0}$ $\boldsymbol{NTIME}(n^c) = \boldsymbol{NP}$
  - $\boldsymbol{DTIME}(\log n) = \boldsymbol{LTIME}$
  - $\boldsymbol{NTIME}(\log n) = \boldsymbol{NLTIME}$

Betrachtete Klassen

	Zeit	Platz
polynomiell	P	$PSPACE$
	NP	$NPSPACE$
logarithmisch	$LTIME$	L
	$NLTIME$	NL

- Gefordert ist eine TM, die eine Eingabe der Länge n in $\log(n)$ Schritten verarbeitet
- Eingabe kann also nicht einmal komplett eingelesen werden
- $LTIME$ und $NLTIME$ in diesem Kontext daher nicht sehr sinnig*

  * wenn wir vom Sequential Access Model ausgehen

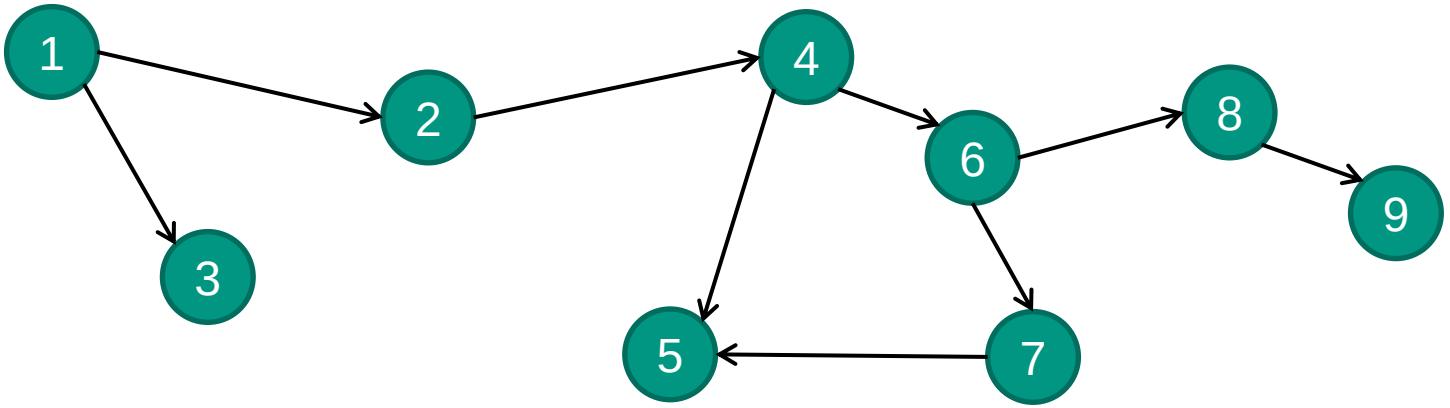
Beispielprobleme

- $\boldsymbol{EVEN} = \{x | x$ hat eine gerade Anzahl von Einsen$\}$
- Ist $\boldsymbol{EVEN} \in \boldsymbol{L}$?
  - DTM, die $\boldsymbol{EVEN}$ entscheidet
  - Zustände q_{even} und q_{odd}
  - q_{even} ist Startzustand und akzeptierender Zustand

Eingabe	q_{even}	q_{odd}
0	q_{even}	q_{odd}
1	q_{odd}	q_{even}
□	$HALT$	$HALT$

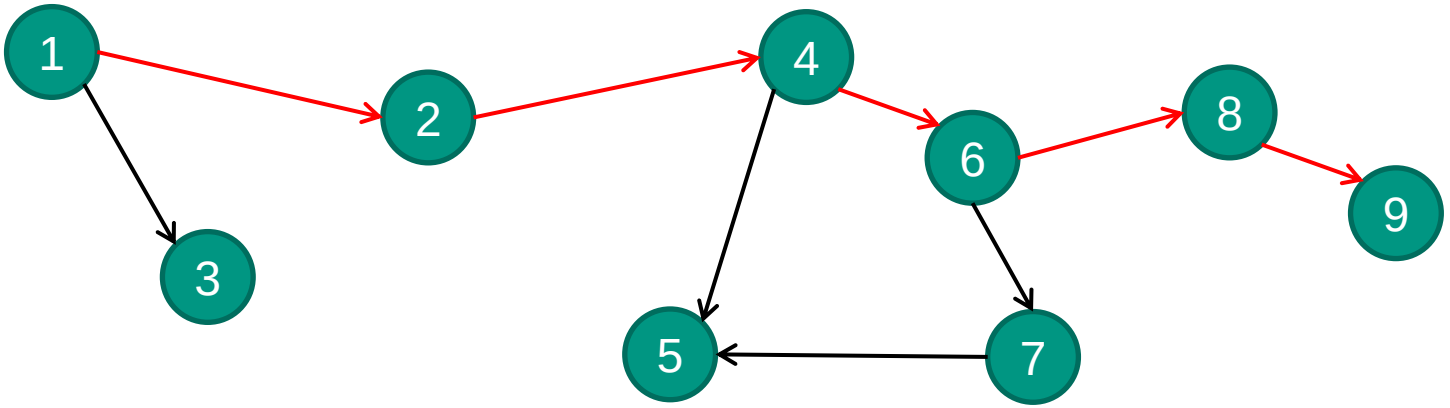
  - Speicherplatzbedarf $0 \in O(\log n) \Rightarrow \boldsymbol{EVEN} \in \boldsymbol{L}$

Beispielprobleme



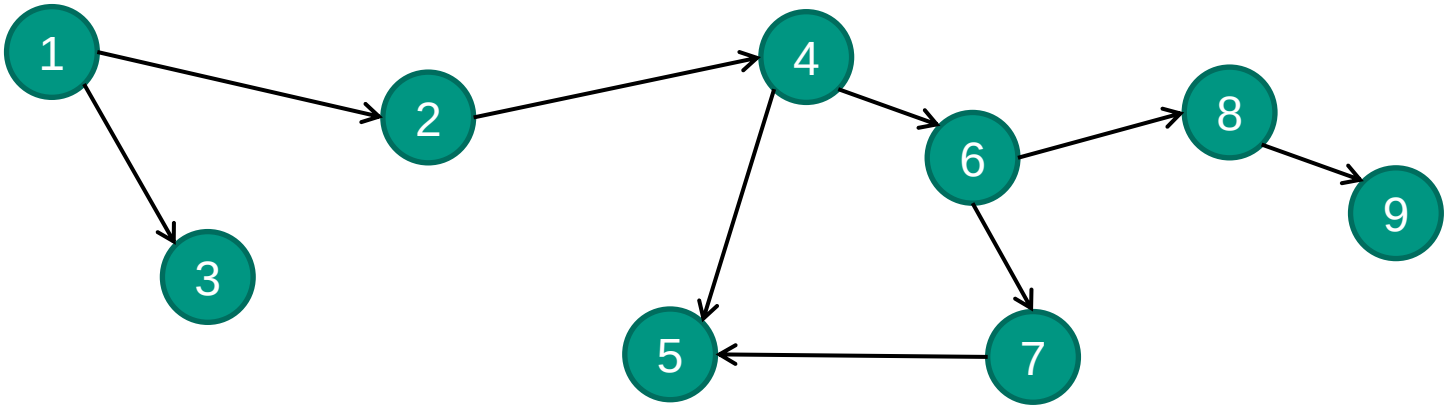
- $\boldsymbol{PATH} = \{< G, s, t >:$ Im gerichteten Graph G gibt es einen Pfad von s nach $t\}$
- Unternehme einen „nichtdeterministischen Spaziergang“ durch G
  - Starte bei Knoten s
  - Speichere Nummer des Knotens auf Band
  - Wähle von dort aus nichtdeterministisch eine ausgehende Kante
  - Wenn t erreicht wurde, akzeptiere
  - Wenn n Knoten besucht wurden, lehne ab

Beispielprobleme



- Warum funktioniert das?
  - Es muss nur *irgendeine* Abfolge von gewählten Knoten geben, die von s nach t führt
  - Nichtdeterminismus sorgt dafür, dass sie gefunden wird
- Wieviel Platz braucht das? (bei n Knoten)
  - Eine Knotennummer: $O(\log n)$
  - Einen Zähler für die Abbruchbedingung: $O(\log n)$
- Damit ist $\boldsymbol{PATH} \in NL$

Beispielprobleme



- Geht das auch deterministisch?
  - Klar, z.B. mit Tiefensuche
  - … die braucht aber mehr Platz
    - DFS muss sich alle vorhergehenden Knoten merken (für Backtracking)
- Geht das auch deterministisch *mit logarithmischem Platzbedarf*?
  - ⇔ Ist $\boldsymbol{PATH} \in L$?
  - Man weiß es nicht

NL-Vollständigkeit

- Ähnlich wie NP-vollständige Probleme lassen sich auch NL-vollständige Probleme ineinander überführen
- $\boldsymbol{PATH}$ ist NL-vollständig [Papadimitriou ‘94]
  - Probleme aus NL lassen sich auf $\boldsymbol{PATH}$ abbilden
  - Eben gesehen: $\boldsymbol{PATH} \in NL$
  - Wenn $\boldsymbol{PATH} \in L$ gilt, dann gilt auch $L = NL$
  - Ob $L = NL$ gilt, ist ein offenes Problem
- $\boldsymbol{PATH}$ ist für NL also das, was z.B. $\boldsymbol{3SAT}$ für NP darstellt!
- Die beiden „Familien“ von Komplexitätsklassen haben somit einige Gemeinsamkeiten

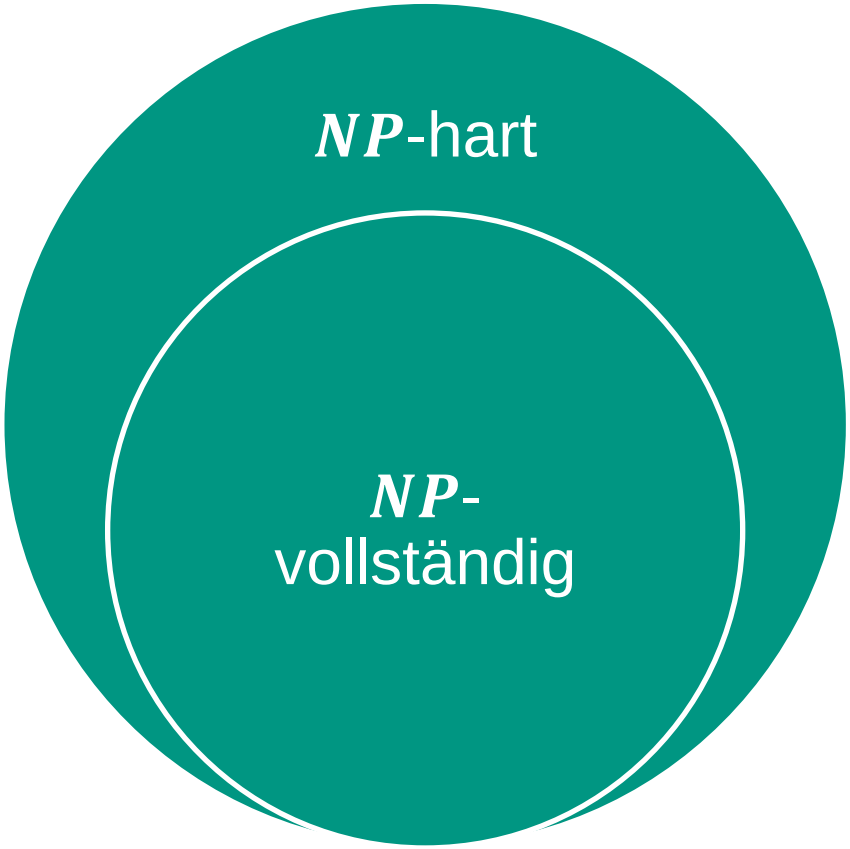
Logspace-Reduktion

- Wie überführt man die NL-Probleme ineinander?
- Bei NP: polynomielle Transformation
  - hier aber zu mächtig
- Die Reduktion sollte nicht mehr Platz haben als die Probleme selbst
- Aber: Wenn die Reduktion nur $\log(n)$ Platz hat, kann sie die reduzierte Instanz nicht einmal ausgeben!
- Daher: **Logspace-Reduktion**
  - Berechnet immer nur ein einzelnes Bit der Ausgabe
  - Alternativ: Transformations-TM kann auf Ausgabeband nur schreiben und nicht zurückbewegen

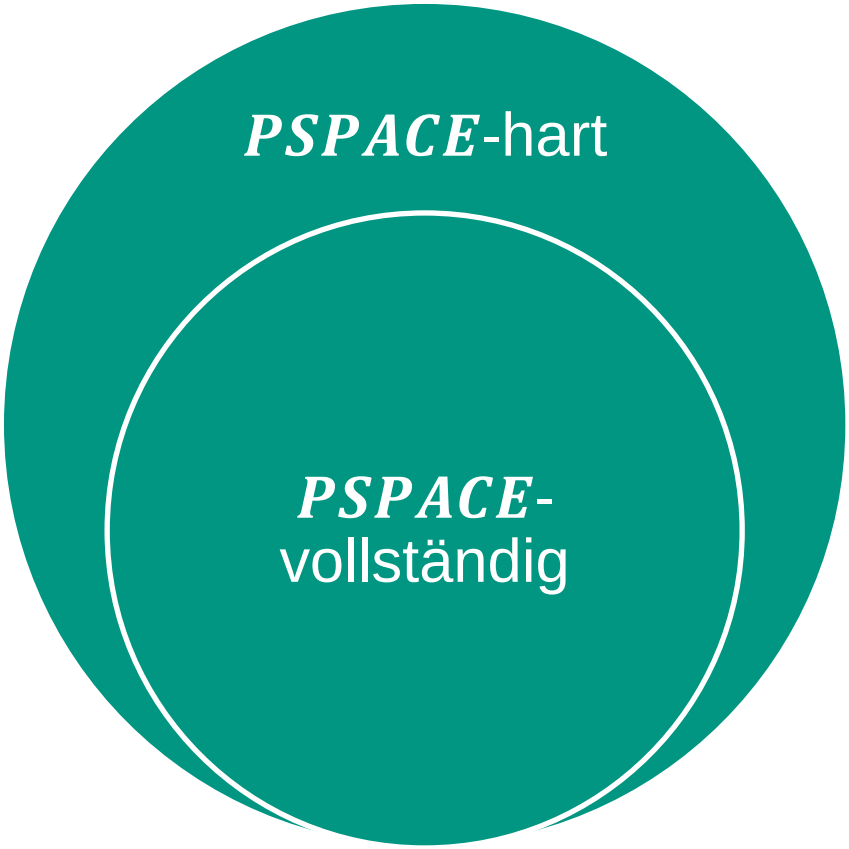
Vollständigkeit

- Bisher haben wir kennengelernt:
  - NP-Vollständigkeit
    - nichtdeterministisch, polynomieller Zeitbedarf
  - NL-Vollständigkeit
    - nichtdeterministisch, logarithmischer Platzbedarf
- Es gibt auch noch:
  - $PSPACE$-Vollständigkeit
    - nichtdeterministisch, polynomieller Platzbedarf

$PSPACE$-Vollständigkeit



Jedes $\pi' \in \boldsymbol{NP}$ polynomiell reduzierbar auf $\pi \in \boldsymbol{NPC}$



Jedes $\pi' \in \boldsymbol{PSPACE}$ polynomiell reduzierbar auf $\pi \in \boldsymbol{PSPACEC}$

$PSPACE$-Vollständigkeit

- Ist $P = PSPACE$?
  - Lässt sich jeder Algorithmus, der polynomiell viel Platz braucht, auch in polynomieller Zeit berechnen?
- Vermutlich nicht, denn:
  - Es gilt $NP \subseteq PSPACE$, da Platzbedarf durch Laufzeit begrenzt ist
  - Wäre $P = PSPACE$, dann wäre $NP \subseteq P$ und damit $P = NP$.

Quantified Boolean Formula (QBF)

- $*_1\ x_1 : *_2\ x_2 : *_3\ x_3 : *_4\ x_4 \ldots : *_n\ x_n : \varphi(x_1, x_2, x_3, x_4, \ldots, x_n)$
  - $*_n, \ldots, *_n$ sind Quantoren ∃ oder ∀
  - φ ist eine Prädikatsfunktion mit Parametern $x_1, \ldots, x_n \in \{\text{true}, \text{false}\}$
- Beispiel: $\forall\ x : \exists\ y : (x \wedge y) \vee (\bar{x} \wedge \bar{y})$
- Problem $\boldsymbol{QBF}$:
  - Gibt es zu diesen Quantoren eine erfüllende Belegung der Parameter?
- Im Beispiel:
  - Durch „scharfes Hinsehen“ sieht man
  - $\forall x\ \exists y : (x == y) = \text{true}$
- $\boldsymbol{TQBF}$: alle erfüllbaren $\boldsymbol{QBF}$-Instanzen
  - $\boldsymbol{TQBF}$ ist $\boldsymbol{PSPACE}$-vollständig [Arora et al. ‘09]

Von Zertifikaten und Spielen

- Von $\boldsymbol{NP}$-vollständigen Problemen sind wir gewohnt:
  - Unsere Zertifikate sind „kurz“ (können in Polynomialzeit überprüft werden)
  - Alle nichtdeterministischen Entscheidungen zu simulieren dauert „lange“ (exponentiell viel Zeit)
- Bei $\boldsymbol{PSPACE}$-vollständigen Problemen ist es umgekehrt:
  - Alle nichtdeterministischen Entscheidungen zu simulieren braucht nur polynomiell viel Platz (Beweis folgt!)
  - Unsere Zertifikate sind „lang“ (brauchen Exponentialzeit zur Überprüfung)

Von Zertifikaten und Spielen

- Was ist ein Zertifikat für ein $\boldsymbol{PSPACE}$-vollständiges Problem?
- Anschaulich: Eine Gewinnstrategie für ein Spiel mit perfekter Information
- Beispiel: Schach
  - Zwei Spieler führen abwechselnd Züge aus
  - Beide können jederzeit das ganze Spielfeld einsehen (perfekte Information)
  - Man sagt, Spieler 1 hat eine Gewinnstrategie, wenn gilt:
    - Es gibt **einen** Spielzug von Spieler **1**, sodass gilt…
    - … für **alle** möglichen Spielzüge von Spieler **2** gilt…
    - … es gibt **einen** Spielzug von Spieler **1**, sodass gilt …
    - … für **alle** möglichen Spielzüge von Spieler **2** gilt …
    - …
    - Spieler 1 gewinnt das Spiel.

Von Zertifikaten und Spielen

- Da Schach ziemlich komplex ist, konstruieren wir das einfachere **QBF-Spiel**.
- Wir wählen eine Boolesche Funktion $\varphi(x_1, x_2, \ldots, x_{2n})$ mit $2n$ Parametern
- Beide Spieler wählen nun abwechselnd die Quantoren ∃ oder ∀
- Spieler 1 muss erreichen, dass die resultierende Formel true wird
- Zertifikat, dass Spieler 1 eine Gewinnstrategie hat
  - Muss für **alle** möglichen Spielzüge von Spieler 2 zeigen, dass Spieler 1 gewinnt
- Ein solches Zertifikat kann exponentiell lang sein!
  - Hier unterscheidet es sich deutlich von $\boldsymbol{NP}$

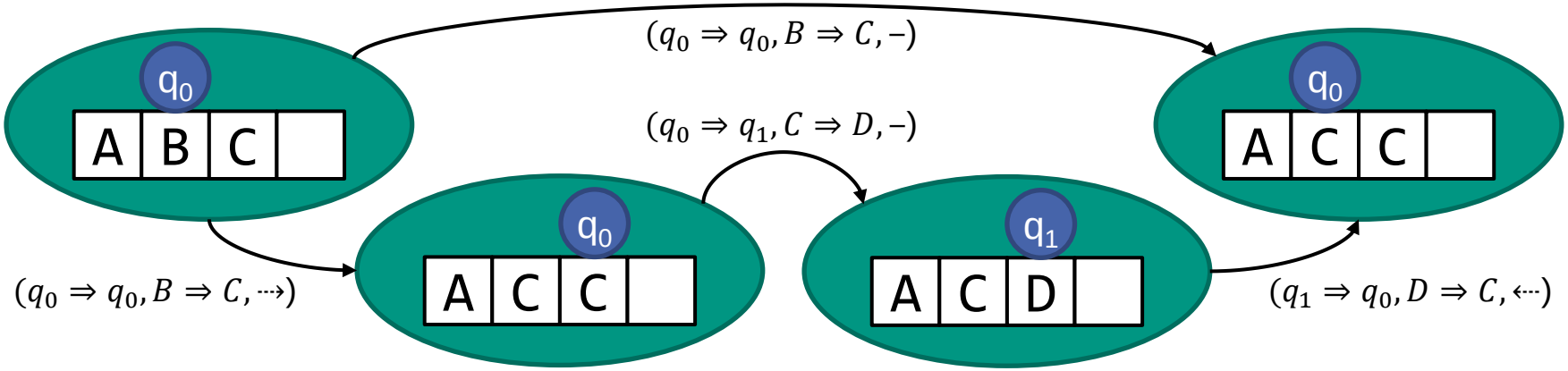
Savitch‘s Theorem

- Gängige Fragestellung im Kontext $\boldsymbol{P \neq NP}$:
  - Wenn eine NDTM eine bestimmte Sprache erkennt, *wie lange* braucht dafür eine DTM?
- Die gleiche Frage kann man sich für den Platzbedarf stellen:
  - Wenn eine NDTM eine bestimmte Sprache erkennt, *wie viel Platz* braucht dafür eine DTM?
- Diese Frage ist bereits geklärt!
- **Savitch‘s Theorem** [Savitch ‘70]
  - Eine nichtdeterministische Turing-Maschine mit Platzbedarf $S(n)$ kann auf einer deterministischen TM simuliert werden mit Platzbedarf $[S(n)^2]$ (vorausgesetzt $S(n) \geq \log n$).

Konfigurationsgraphen

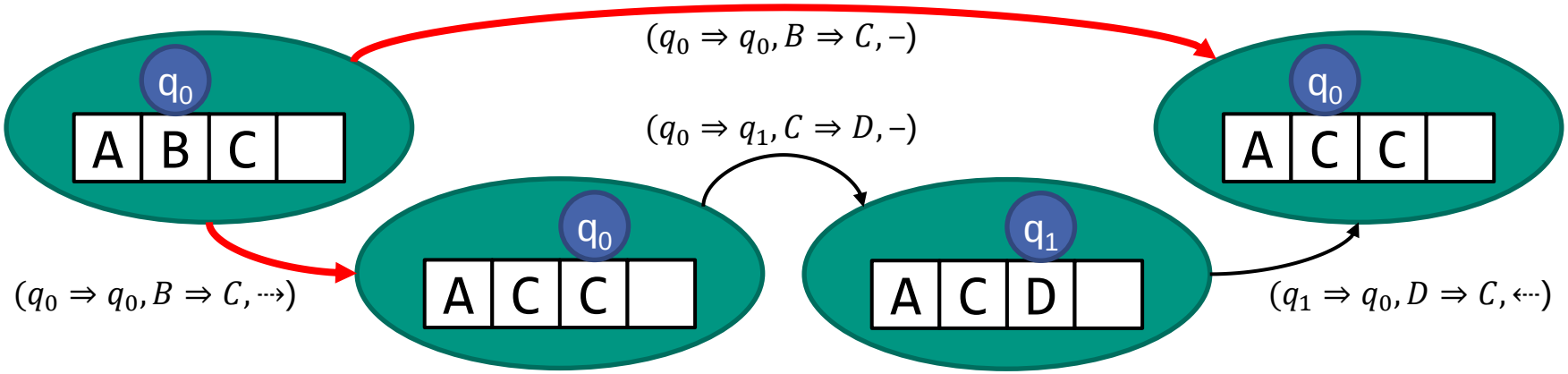
- **Konfiguration** einer TM M
  - Bandinhalt (nicht-leer)
  - Zustand
  - Position des Lese-/Schreibkopfes

  zu einem gegebenen Ausführungszeitpunkt
- **Konfigurationsgraph** $G_{M,x}$ auf einer Eingabe x
  - Ein Knoten entspricht einer Konfiguration
  - Jede gerichtete Kante entspricht einem Ausführungsschritt entsprechend der Zustandsübergangsfunktion



Konfigurationsgraphen

- Deterministische Turing-Maschinen
  - Zu jedem Zeitpunkt genau eine, deterministisch festgelegte Möglichkeit für den nächsten Schritt
  - Konfigurationsgraph hat Ausgangsgrad = 1
- Nichtdeterministische Turing-Maschinen
  - Mehrere nichtdeterministische Übergänge möglich
  - Konfigurationsgraph hat Ausgangsgrad 1 oder 2



Konfigurationsgraphen

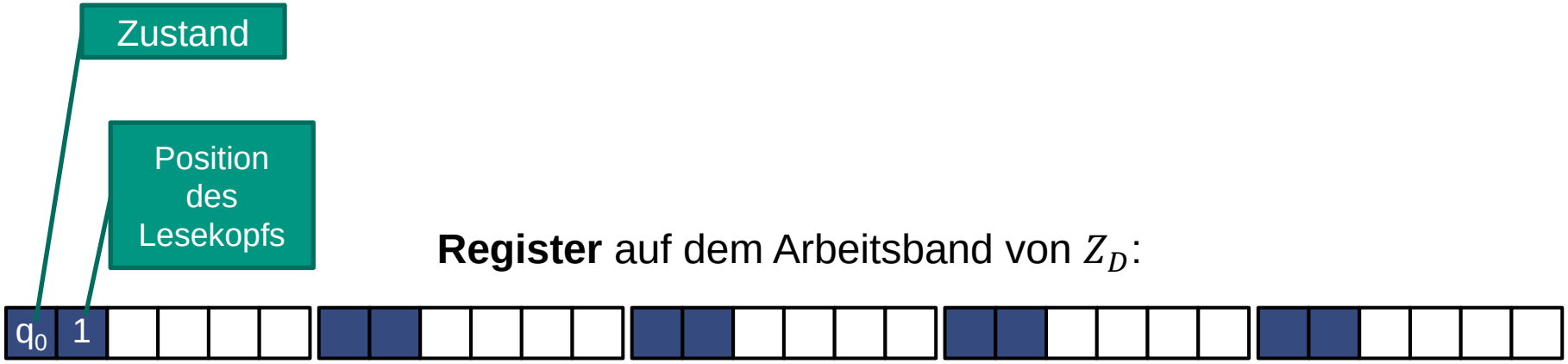
- Bezeichne Startkonfiguration mit C_{start}
- Annahme: genau eine akzeptierende Konfiguration C_{accept}
  - Ansonsten: modifiziere TM so, dass sie bei Erreichen eines akzeptierenden Zustands alle Bänder löscht und in neuen Zustand C_{accept} wechselt
- Turing-Maschine M akzeptiert Eingabe x
  $\Leftrightarrow$ Es gibt im Konfigurationsgraphen $G_{M,x}$ einen gerichteten Pfad von C_{start} nach C_{accept}

Beweiskonstruktion von Savitch‘s Theorem

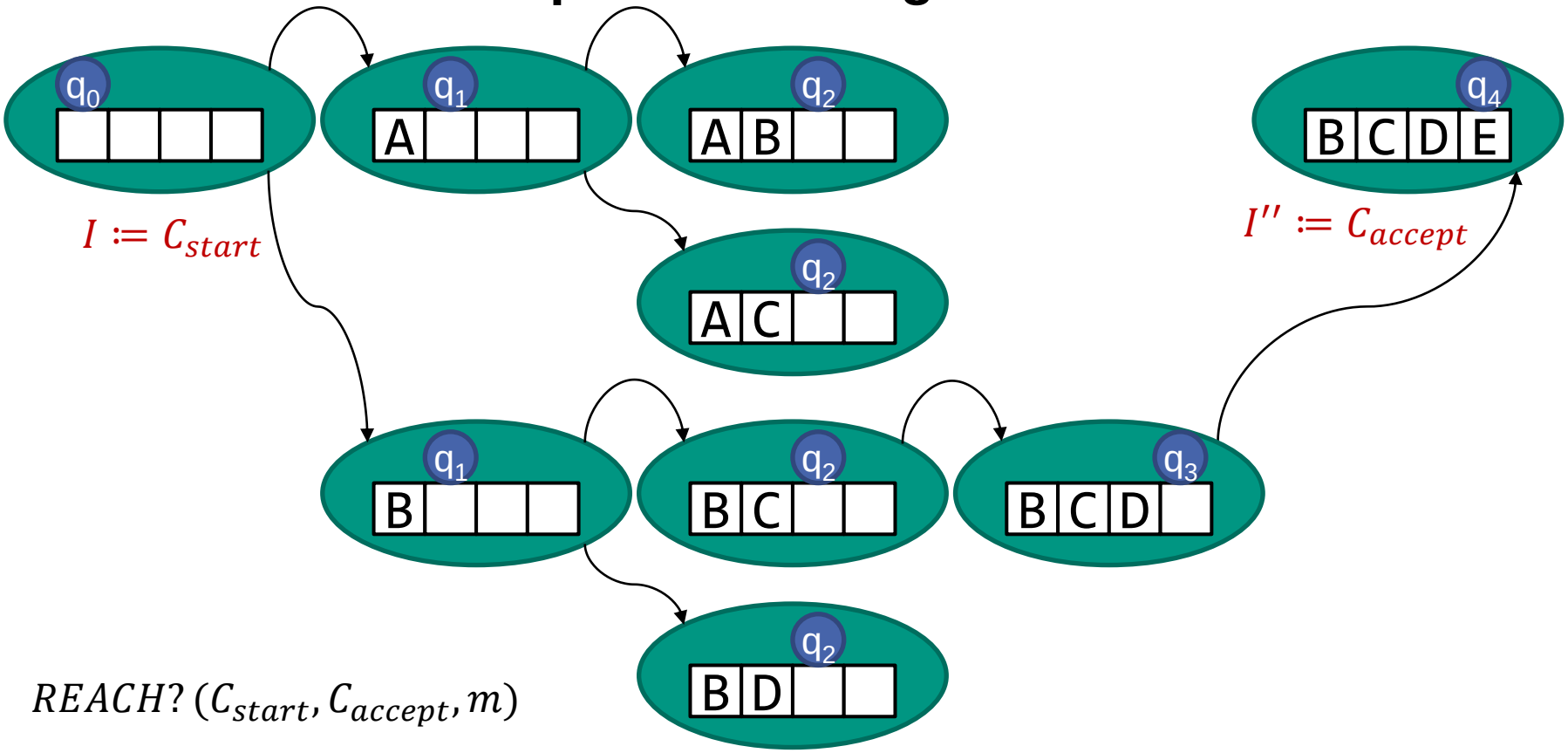
- Gegeben eine Turing-Maschine Z_N
  - nichtdeterministisch
  - entscheidet eine Sprache L
  - Platzbedarf $S(n) \geq \log n$
- Wir konstruieren uns daraus eine deterministische TM Z_D
  - Z_D sucht auf dem (gedanklichen) Konfigurationsgraph einen Pfad von C_{start} nach C_{accept}
  - Hierzu: rekursive Funktion $REACH?\,(u, v, i)$
    - Gibt es einen Pfad von u nach v der Länge höchstens 2^i?

Beweiskonstruktion von Savitch‘s Theorem

- Given an input of length n, Z_D initializes its computation by **dividing one of its storage tapes into** $[c'S(n)] + 1$ **blocks (called *registers*)**, each of length $[cS(n)]$
- […] Note that each register is capable of holding any ID of Z_N in which the nonblank portion of every storage tape is at most $S(n)$. […]

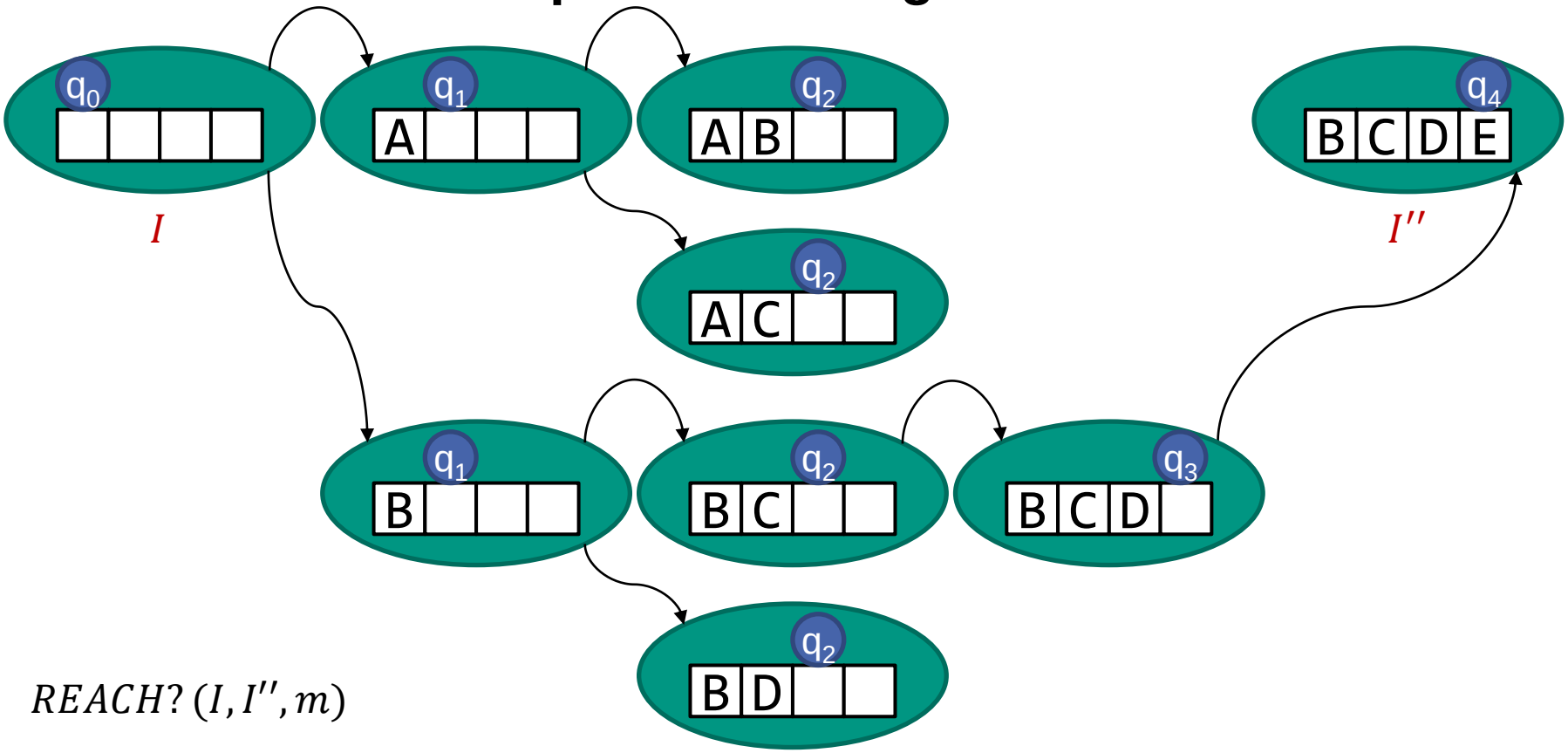


Deterministische Implementierung



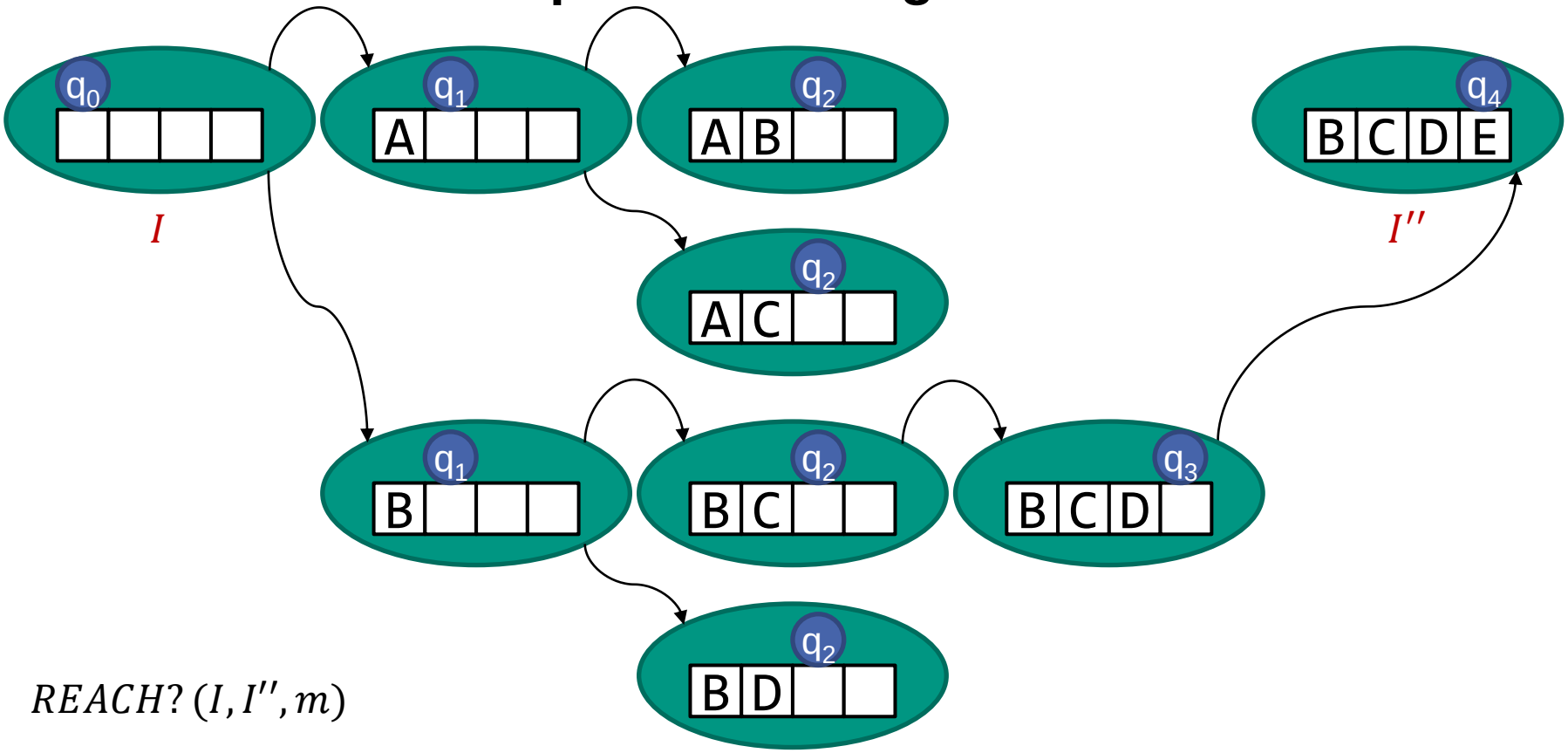
„For a prechosen m, it will need to check if […] Z_N can in 2^m steps change its configuration from I to I''. Furthermore, Z_D will have to perform this task using only m registers.“

Deterministische Implementierung



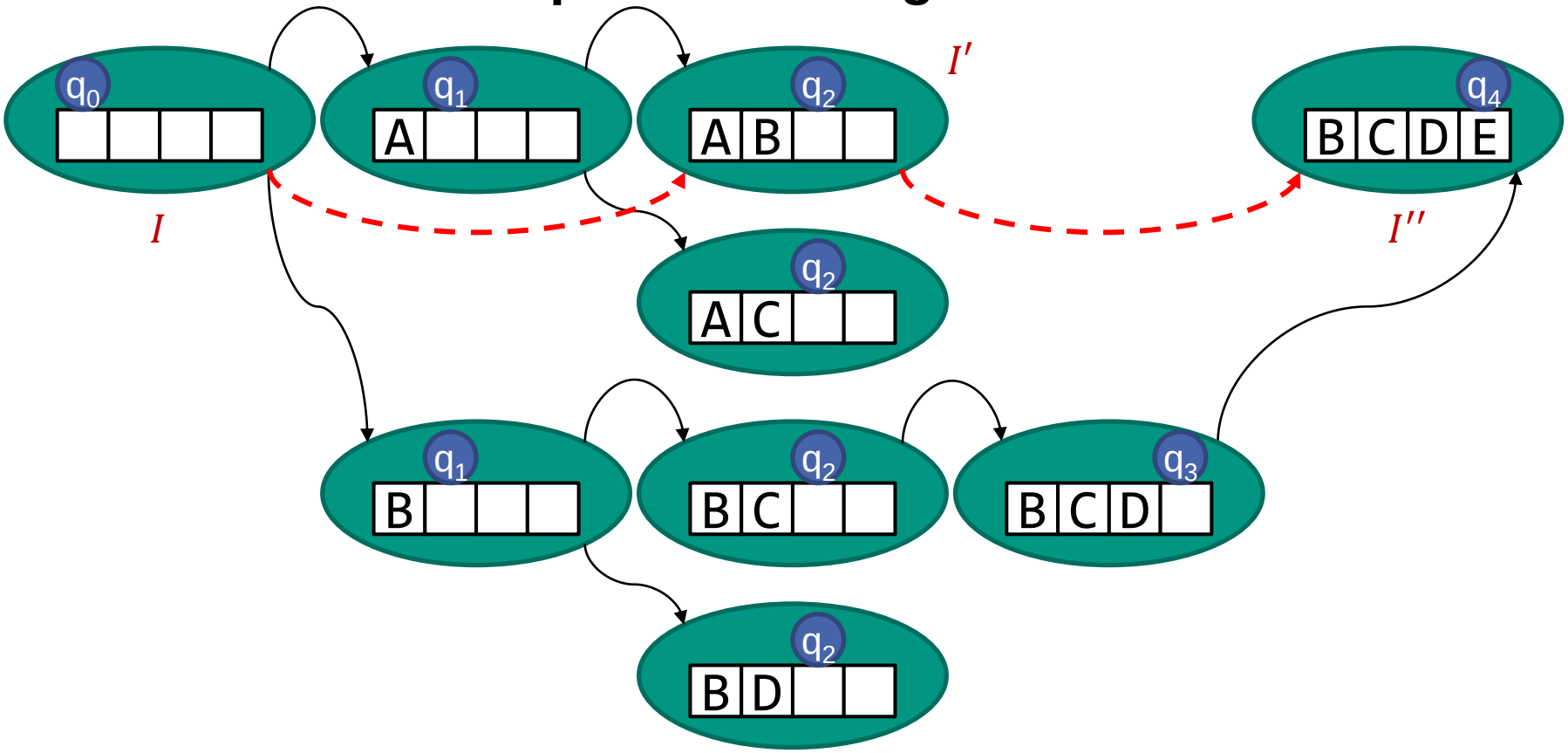
„Z_D proceeds as follows. It runs through (in a systematic way) all ID's I' of Z_N and for each I', checks to see if there is a computation of at most 2^{m-1} steps from I to I', and a computation of at most 2^{m-1} steps from I' to I''."

Deterministische Implementierung

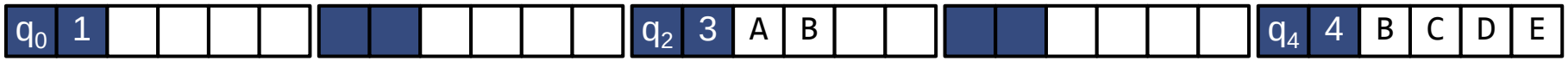


Register auf dem Arbeitsband von Z_D:

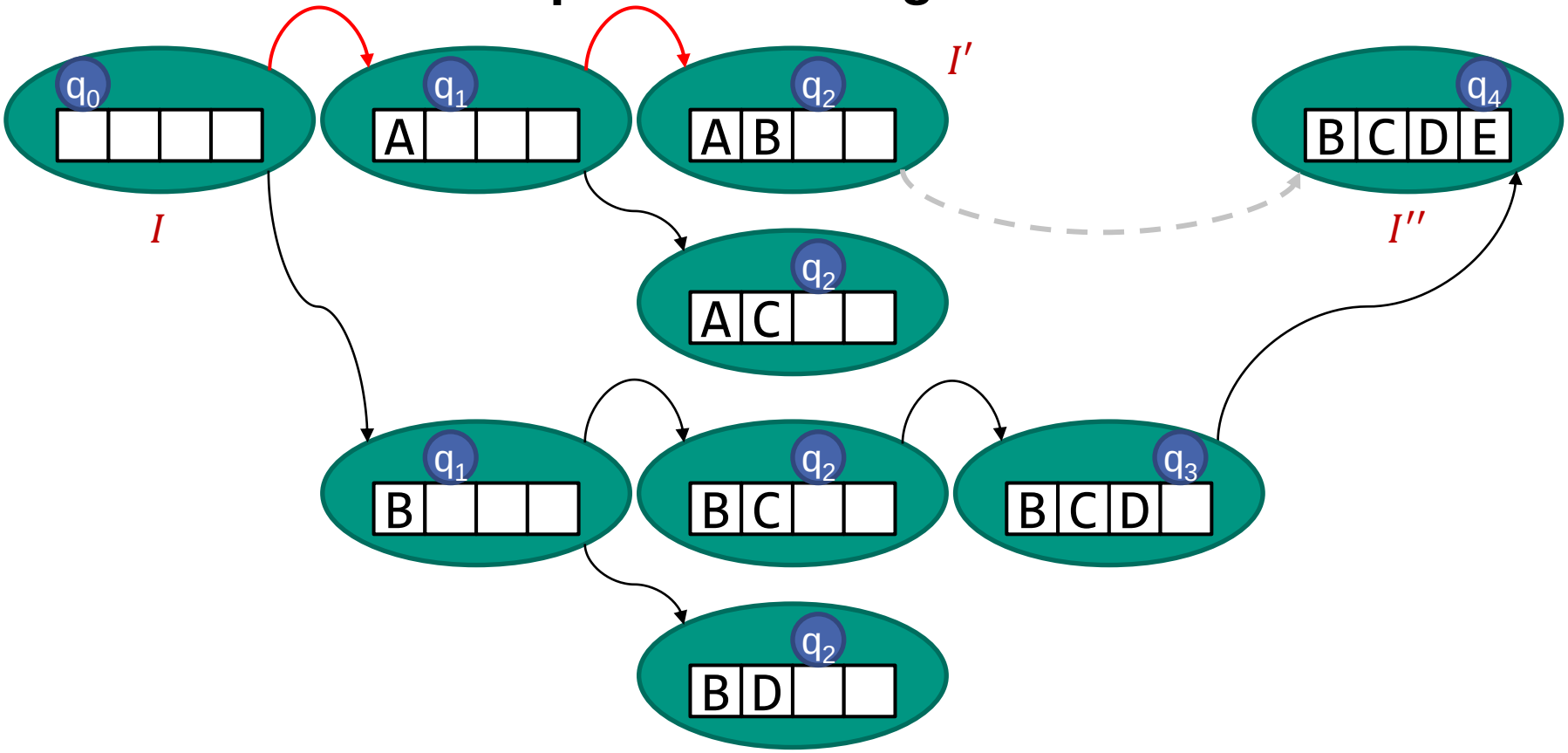
Deterministische Implementierung



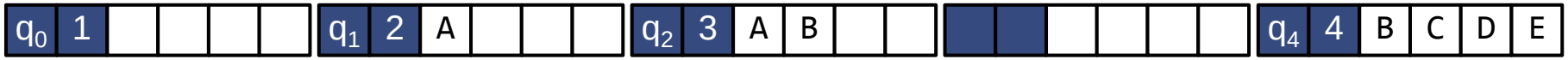
$$REACH?\,(I, I'', m)$$
$$= REACH?\,(I, I', m-1) \wedge REACH?\,(I', I'', m-1)$$



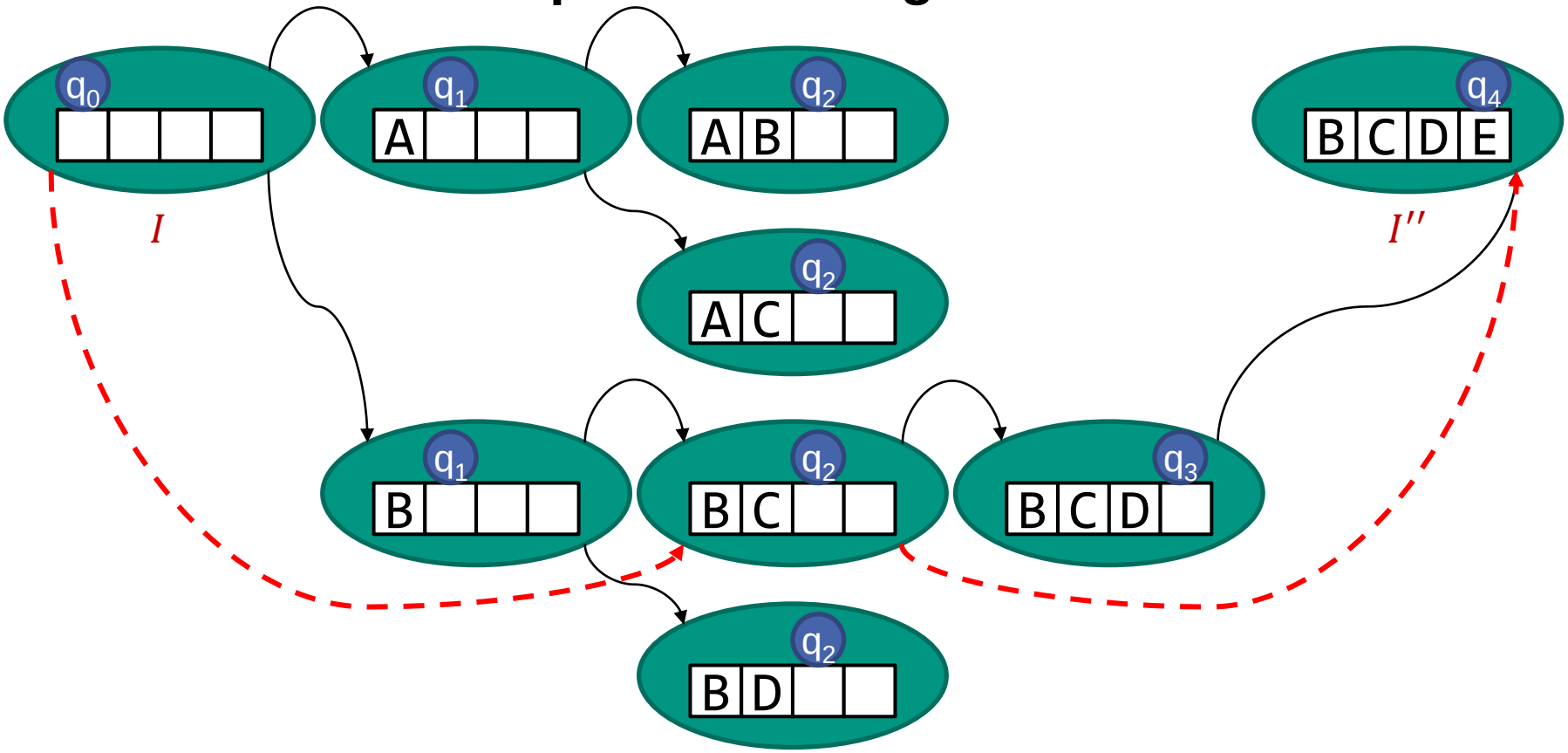
Deterministische Implementierung



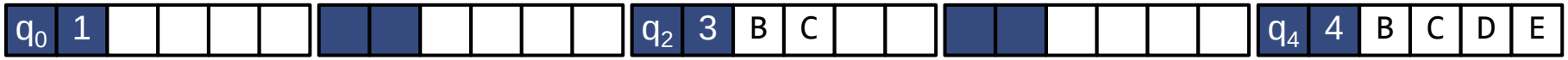
$$REACH?\,(I, I'', m)$$
$$= REACH?\,(I, I', m-1) \wedge REACH?\,(I', I'', m-1)$$



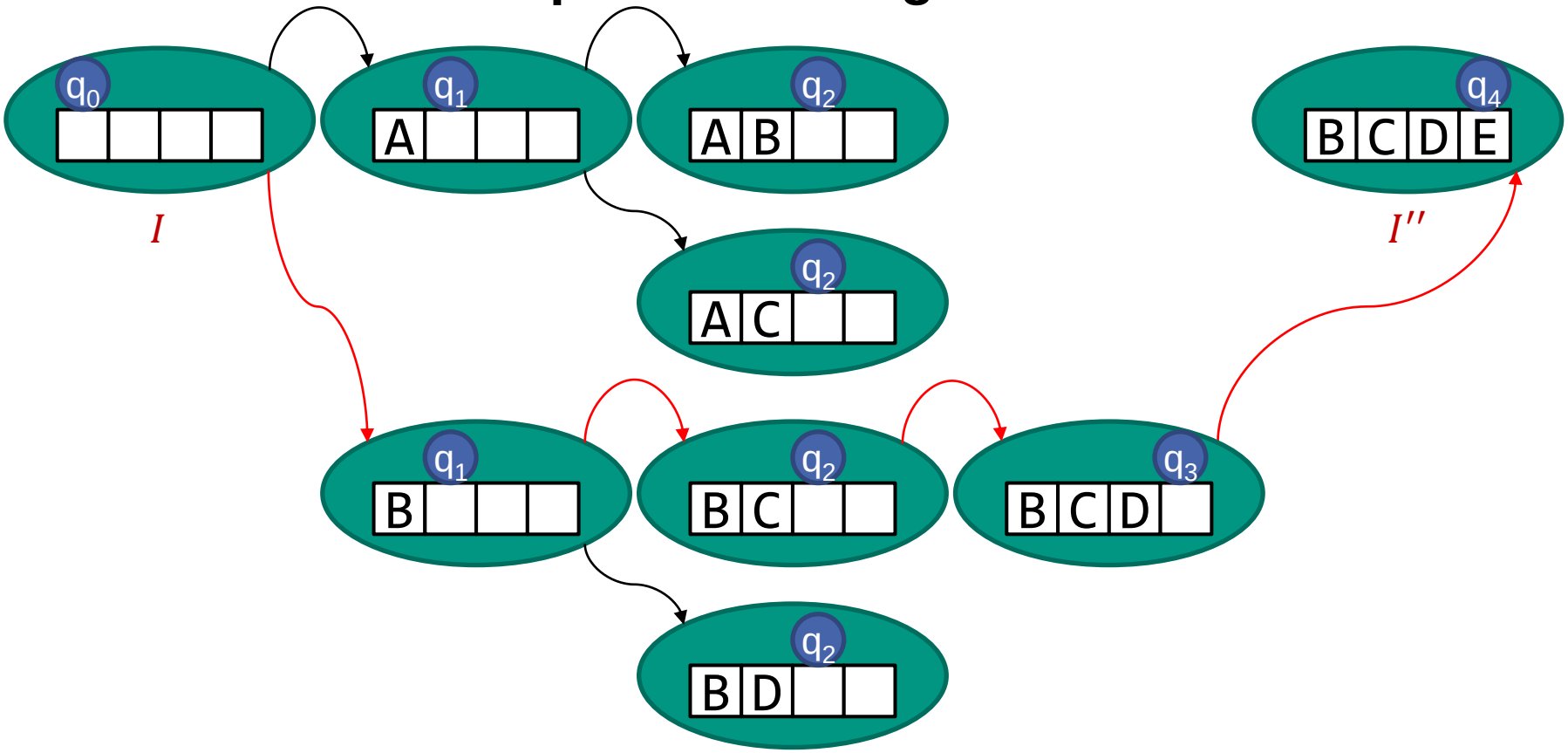
Deterministische Implementierung



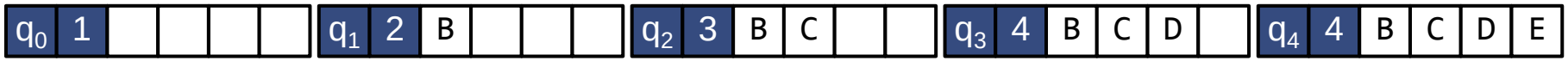
$$REACH?\,(I, I'', m)$$
$$= REACH?\,(I, I', m-1) \wedge REACH?\,(I', I'', m-1)$$



Deterministische Implementierung



$$REACH?\,(I, I'', m)$$
$$= REACH?\,(I, I', m-1) \wedge REACH?\,(I', I'', m-1) = \cdots$$



Deterministische Implementierung

- Funktion $REACH?$ arbeitet rekursiv
- Zu suchende Pfadlänge wird in jedem Rekursionsschritt um Faktor 2 verringert
- Basisfall: ein einzelner Schritt von Maschine Z_N
  - Dieser braucht dann keinen zusätzlichen Platz mehr, kann durch die endliche Steuerung von Z_D überprüft werden
- Z_D geht alle möglichen Konfigurationen systematisch durch

Warum reicht der Speicherplatz dafür aus?

(oder: Wo kommt eigentlich dieses m her?)

- Ursprüngliche TM Z_N braucht höchstens Platz $S(n)$
- Schon gesehen: $\boldsymbol{SPACE}\big(S(n)\big) \subseteq \boldsymbol{DTIME}\big(2^{O(S(n))}\big)$
  - Laufzeit (!) von Z_N also begrenzt durch $2^{O(S(n))}$
  - Kann maximal $2^{O(S(n))}$ Schritte durchführen
  - Somit maximal $M = 2^{O(S(n))}$ Knoten im Konfigurationsgraph
- $REACH?$ hat damit höchstens $\log M = O\big(S(n)\big) =: m$ Rekursionsstufen
- Jede Stufe muss sich nur eine Konfiguration merken
  - Also Platzbedarf $O\big(S(n)\big)$ pro Stufe
- Insgesamt: Platzbedarf von Z_D ist $O\big(S(n)\big) \cdot O\big(S(n)\big) = O(S(n)^2)$

Bedeutung für die Komplexitätsklassen

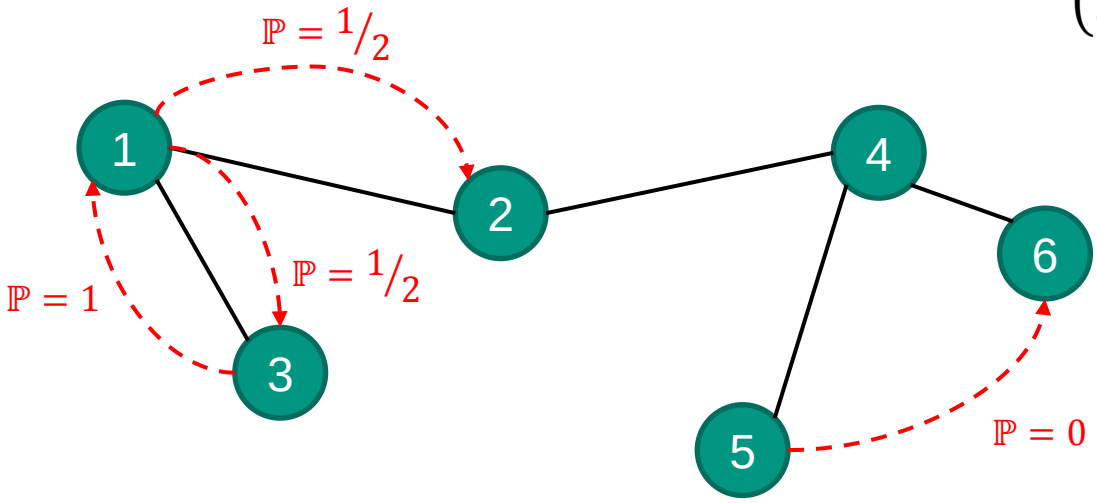
- Eben gezeigt:
  Nichtdeterministische TM können durch deterministische TM simuliert werden, wobei sich der Platzbedarf höchstens quadriert
- Also $\boldsymbol{NSPACE}\big(S(n)\big) \subseteq \boldsymbol{SPACE}(S(n)^2)$
- Da der Zuwachs polynomiell ist, fallen die Klassen zusammen:
  $$\boldsymbol{PSPACE} = \boldsymbol{NPSPACE}$$ [Arora et al. ‘09]
- Korollar von Savitch‘s Theorem
  - Jede kontextsensitive Sprache wird von einer deterministischen Turing-Maschine in Platz $O(n^2)$ erkannt

[Savitch ‘70]

Probabilistische Algorithmen

- Schon gesehen: $\boldsymbol{PATH}$ für gerichtete Graphen $\in \boldsymbol{NL}$
- Was ist mit ungerichteten Graphen?
- Probabilistischer Polynomialzeit-Algorithmus in $\boldsymbol{SPACE}(\log n)$
  - Algorithmus bewegt sich „zufällig“ durch ungerichteten Graph G
  - Übergangswahrscheinlichkeit von Knoten i nach j hängt vom Knotengrad $d(i)$ ab

$$\mathbb{P}_{i\to j} = \begin{cases} 0 & \nexists \text{ Kante } (i,j) \\ \dfrac{1}{d(i)} & \text{sonst} \end{cases}$$



Probabilistische Algorithmen

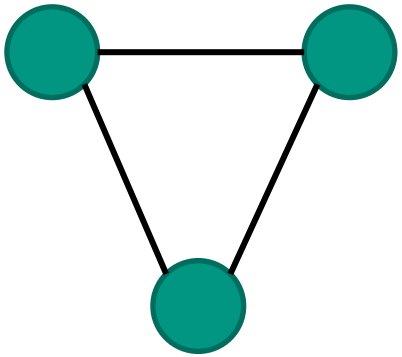
- Hier: Nichtdeterminismus „ersetzt“ durch Zufall
- Laufzeit polynomiell
- Platzbedarf weiterhin $O(\log n)$
- Aber: Algorithmus kann fehlschlagen
  - durch unglückliche Zufallsentscheidungen

„It should be noted that each of these algorithms will have a small probability of failure which can be made arbitrarily small – in a sense one can trade accuracy for space.“

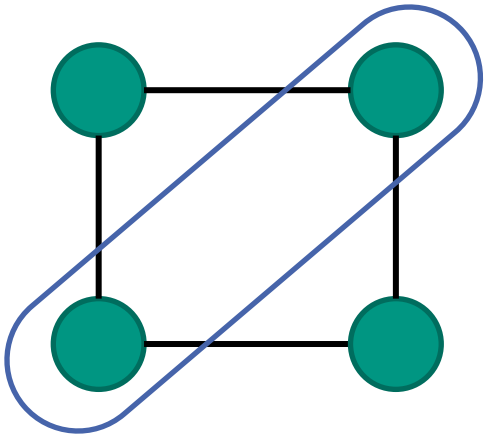
[Aleliunas et al. ‘79]

Probabilistische Algorithmen

- Neues Problem:
  - Gegeben Graph G, Knoten l
  - Ist die Zusammenhangskomponente, in der l liegt, **nicht** bipartit?
  - $\Leftrightarrow$ Ist l Teil eines ungeraden Zyklus?
- Gedächtnisstütze
  - Ein Graph heißt **bipartit**, wenn sich dessen Knoten in zwei Mengen aufteilen lassen, sodass innerhalb jeweils einer Menge keine Knoten verbunden sind



nicht bipartit



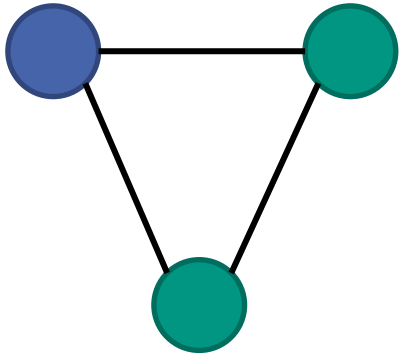
bipartit

Probabilistische Algorithmen

- Annahme: Algorithmus kann in Logspace…
  - einen einzigen Knoten markieren
  - Nachbarn eines Knoten bestimmen
- Ablauf
  - Markiere Ausgangsknoten
  - Initialisiere Paritätszähler mit 0
  - Führe zufällige Traversierung durch
    - Nach jedem Schritt: Invertiere Paritätszähler
    - Falls zurück am markierten Knoten angekommen und Parität = 1: akzeptiere (d. h. G ist nicht bipartit)

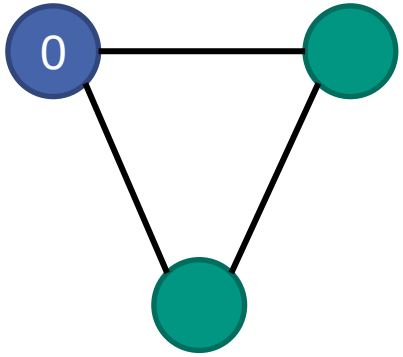
Probabilistische Algorithmen

- Annahme: Algorithmus kann in Logspace…
  - einen einzigen Knoten markieren
  - Nachbarn eines Knoten bestimmen
- Ablauf
  - **Markiere Ausgangsknoten**
  - Initialisiere Paritätszähler mit 0
  - Führe zufällige Traversierung durch
    - Nach jedem Schritt: Invertiere Paritätszähler
    - Falls zurück am markierten Knoten angekommen und Parität = 1: akzeptiere (d. h. G ist nicht bipartit)



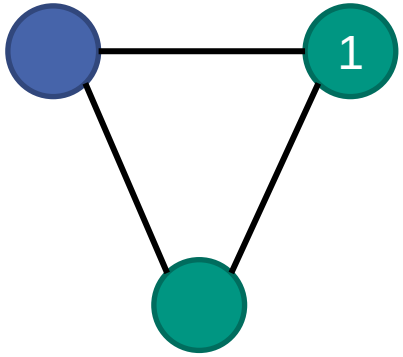
Probabilistische Algorithmen

- Annahme: Algorithmus kann in Logspace…
  - einen einzigen Knoten markieren
  - Nachbarn eines Knoten bestimmen
- Ablauf
  - Markiere Ausgangsknoten
  - **Initialisiere Paritätszähler mit 0**
  - Führe zufällige Traversierung durch
    - Nach jedem Schritt: Invertiere Paritätszähler
    - Falls zurück am markierten Knoten angekommen und Parität = 1: akzeptiere (d. h. G ist nicht bipartit)



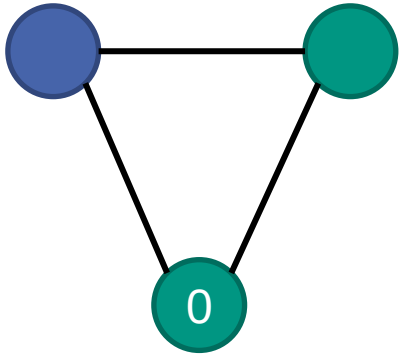
Probabilistische Algorithmen

- Annahme: Algorithmus kann in Logspace…
  - einen einzigen Knoten markieren
  - Nachbarn eines Knoten bestimmen
- Ablauf
  - Markiere Ausgangsknoten
  - Initialisiere Paritätszähler mit 0
  - **Führe zufällige Traversierung durch**
    - Nach jedem Schritt: Invertiere Paritätszähler
    - Falls zurück am markierten Knoten angekommen und Parität = 1: akzeptiere (d. h. G ist nicht bipartit)



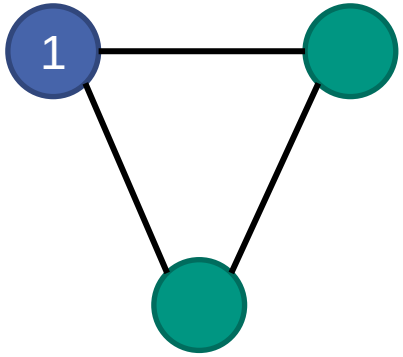
Probabilistische Algorithmen

- Annahme: Algorithmus kann in Logspace…
  - einen einzigen Knoten markieren
  - Nachbarn eines Knoten bestimmen
- Ablauf
  - Markiere Ausgangsknoten
  - Initialisiere Paritätszähler mit 0
  - **Führe zufällige Traversierung durch**
    - Nach jedem Schritt: Invertiere Paritätszähler
    - Falls zurück am markierten Knoten angekommen und Parität = 1: akzeptiere (d. h. G ist nicht bipartit)



Probabilistische Algorithmen

- Annahme: Algorithmus kann in Logspace…
  - einen einzigen Knoten markieren
  - Nachbarn eines Knoten bestimmen
- Ablauf
  - Markiere Ausgangsknoten
  - Initialisiere Paritätszähler mit 0
  - Führe zufällige Traversierung durch
    - Nach jedem Schritt: Invertiere Paritätszähler
    - **Falls zurück am markierten Knoten angekommen und Parität = 1: akzeptiere**
      (d. h. G ist nicht bipartit)



Weitere Klassen

- Diese probabilistischen Algorithmen motivieren die Betrachtung weiterer Klassen von Sprachen:
  - $\boldsymbol{RSPACE}(S(n))$
    - Von probabilistischen Algorithmen erkannt
    - Platzbedarf $S(n)$
  - $\boldsymbol{RSPACE}^{\boldsymbol{POLY}}(S(n))$
    - zusätzliche Einschränkung: in Polynomialzeit
- Ungerichtetes $\boldsymbol{PATH}$ ist in $\boldsymbol{RSPACE}^{\boldsymbol{POLY}}(\log n)$.
- Noch unklar, ob gerichtetes $\boldsymbol{PATH}$ in $\boldsymbol{RSPACE}^{\boldsymbol{POLY}}(\log n)$.
  $\Leftrightarrow \boldsymbol{RSPACE}^{\boldsymbol{POLY}}(\log n) = \boldsymbol{RSPACE}(\log n)$
- Ob die beiden Klassen zusammenfallen, ist offen
- $\boldsymbol{RSPACE}(\log n) = \boldsymbol{NSPACE}(\log n)$
  - Mit genügend Zeit lässt sich der Nichtdeterminismus durch Zufall simulieren

[Aleliunas et al. ‘79]

Wichtige Punkte

- $\boldsymbol{DTIME}(S(n)) \subseteq \boldsymbol{SPACE}(S(n)) \subseteq \boldsymbol{NSPACE}(S(n)) \subseteq \boldsymbol{DTIME}(2^{O(S(n))})$
- Für $\boldsymbol{NL}$ und $\boldsymbol{NPSPACE}$ gibt es ähnliche Vollständigkeitsbegriffe wie für $\boldsymbol{NP}$
- $\boldsymbol{PSPACE} = \boldsymbol{NPSPACE}$ (Savitch‘s Theorem)
- $\boldsymbol{RSPACE} = \boldsymbol{NSPACE}$ (Zufall statt Nichtdeterminismus)
- Wäre $\boldsymbol{P} = \boldsymbol{PSPACE}$, dann wäre $\boldsymbol{NP} \subseteq \boldsymbol{P}$ und damit $\boldsymbol{P} = \boldsymbol{NP}$

Vielen Dank für die Aufmerksamkeit ☺