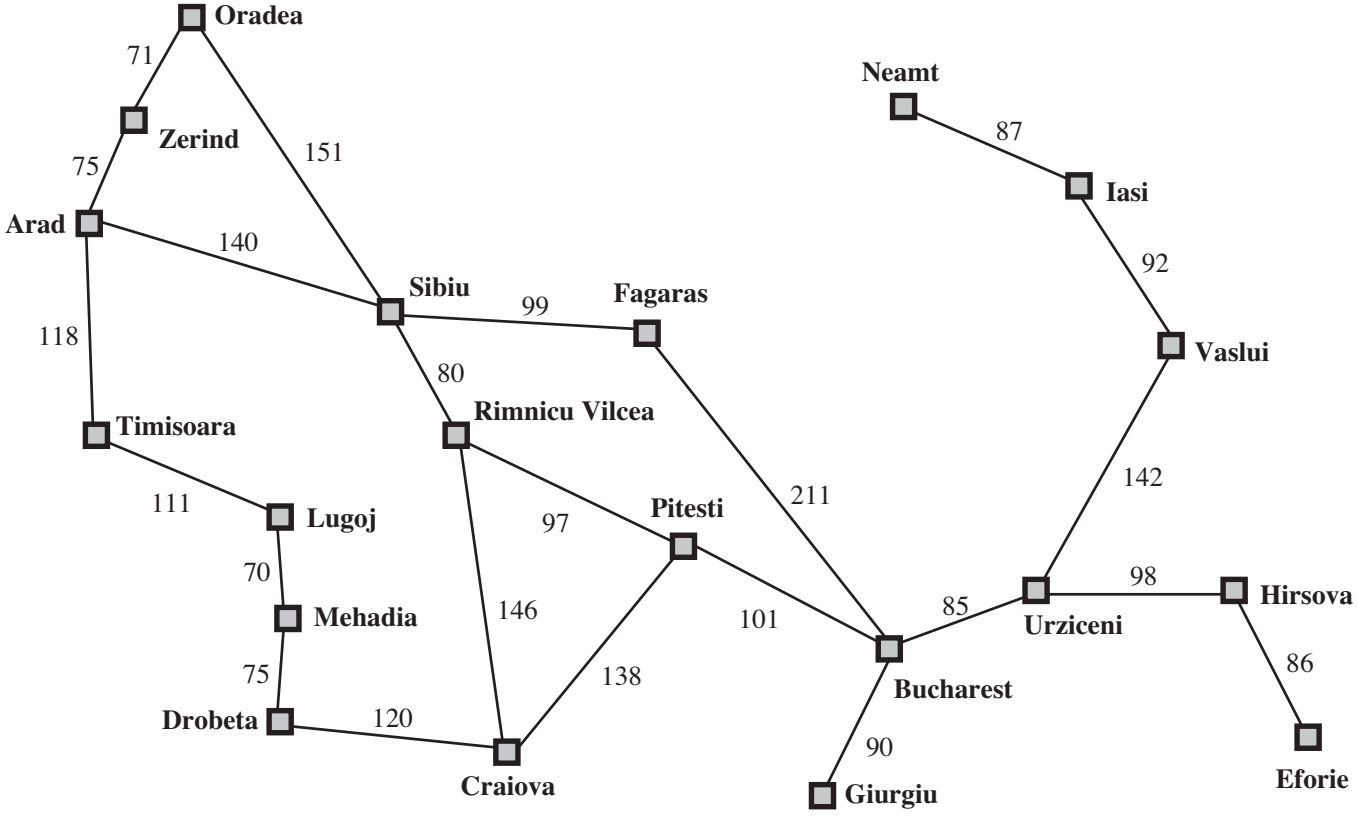
Künstliche Intelligenz

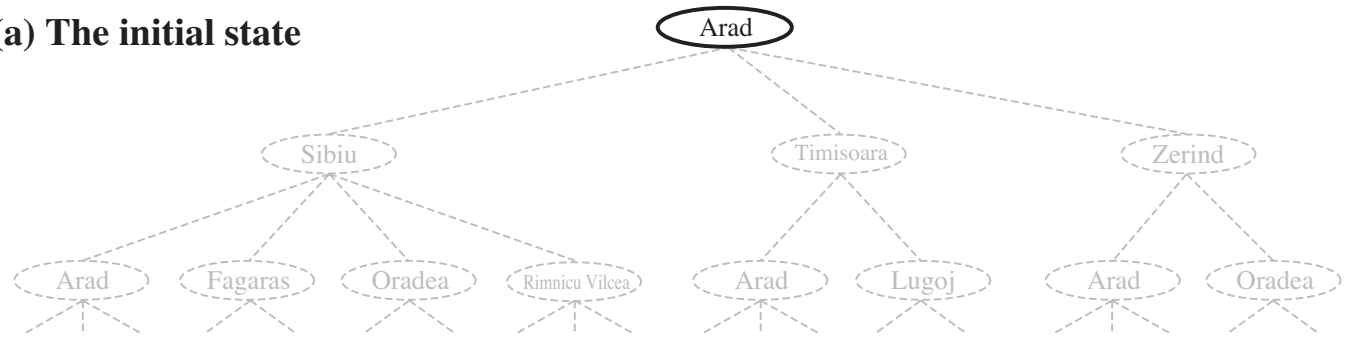
Vorlesung 2: Suchverfahren
Informierte Suche

WIEDERHOLUNG

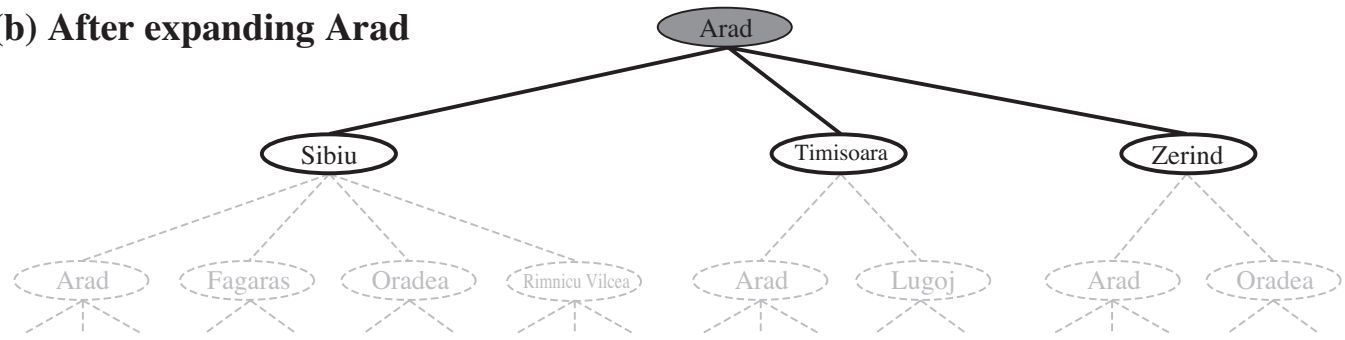
- Bislang uninformierte Strategien
- BFS, DFS, Iteratives Vertiefen, Bidirektionale Suche
- Wichtige Begriffe: Suchraum, Suchbaum, Grenze


Oradea
71
Neamt
87
Zerind
151
75
Iasi
Arad
140
92
Sibiu
99
Fagaras
118
Vaslui
80
Rimnicu Vilcea
Timisoara
142
111
211
Lugoj
97
Pitesti
70
98
Hirsova
Mehadia
146
85
101
Urziceni
75
86
138
Bucharest
Drobeta
120
90
Craiova
Giurgiu
Eforie

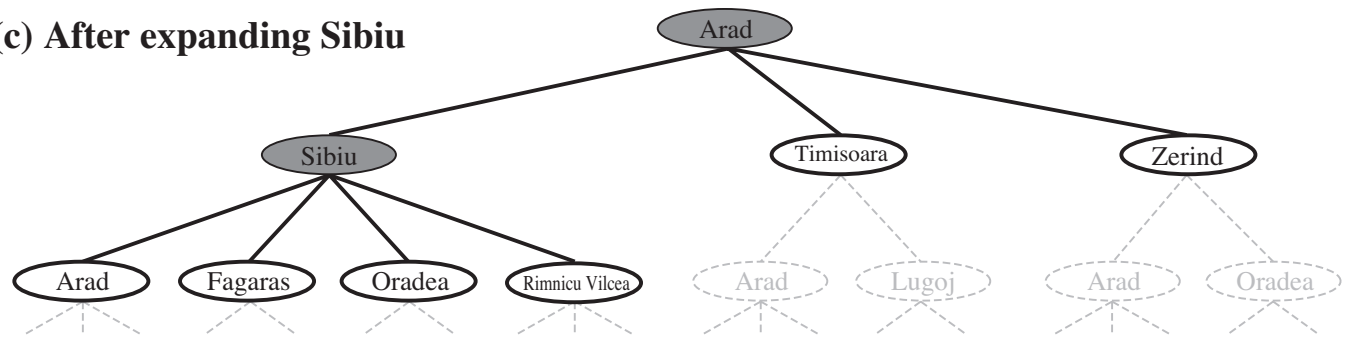
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu



function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

---

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

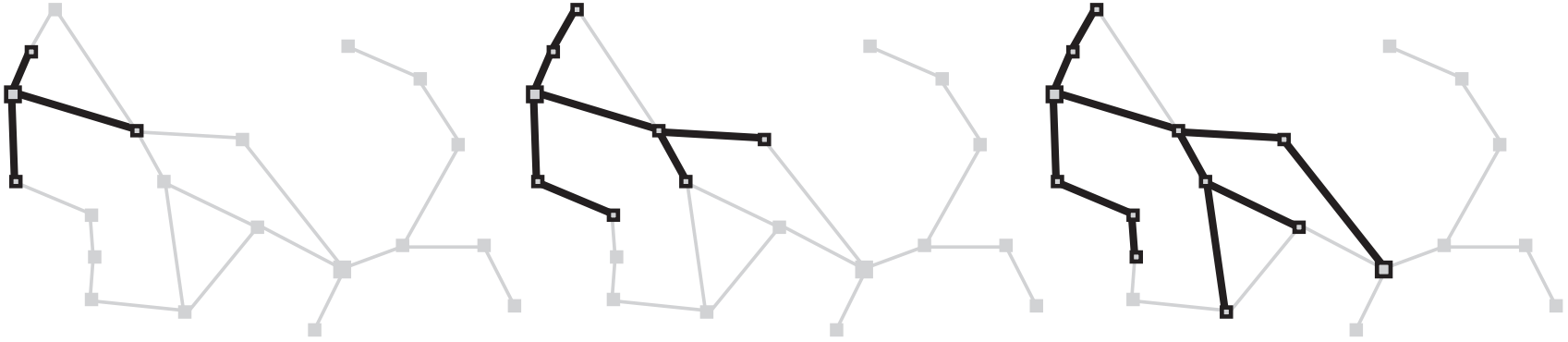


Abbildung 1: Reihenfolge von Suchbäume

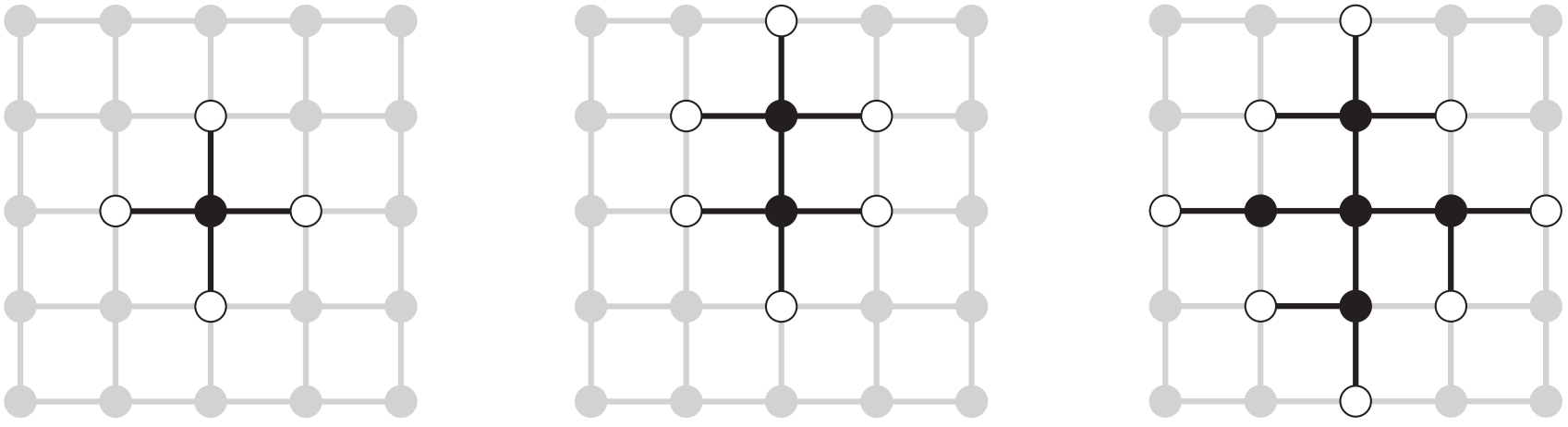


Abbildung 2: Trennungseigenschaft auf der Grenze

Probleme damit?

- Suchgrids sind wichtig in Computerspiele
- Jeder Zustand hat 4 Nachfolger
- Ein Suchbaum der Tiefe d hat 4^d Blätter
- Es existieren aber nur ungefähr $2d^2$ verschiedene Zustände nach d Schritte von irgendeinem Zustand ausgehend
- $d = 20$: 10^{12} Knoten aber nur 800 verschiedene Zustände
- Redundante Pfade machen lösbare Probleme unlösbar...

Infrastruktur für Suchalgorithmen

Für jeden Knoten n wird eine 4-elementige Datenstruktur erzeugt:

- n.State: Der Zustand der dem Knoten n im Zustandsraum (`Statespace`) entspricht.
- n.Parent: Der Vorgänger von n im Suchbaum.
- n.Actions: Die Handlung, die n aus seinem Vorgänger erzeugt hat.
- n.Path-Cost: Die Kosten, die entstehen, um aus dem initialen Zustand nach n zu gelangen.

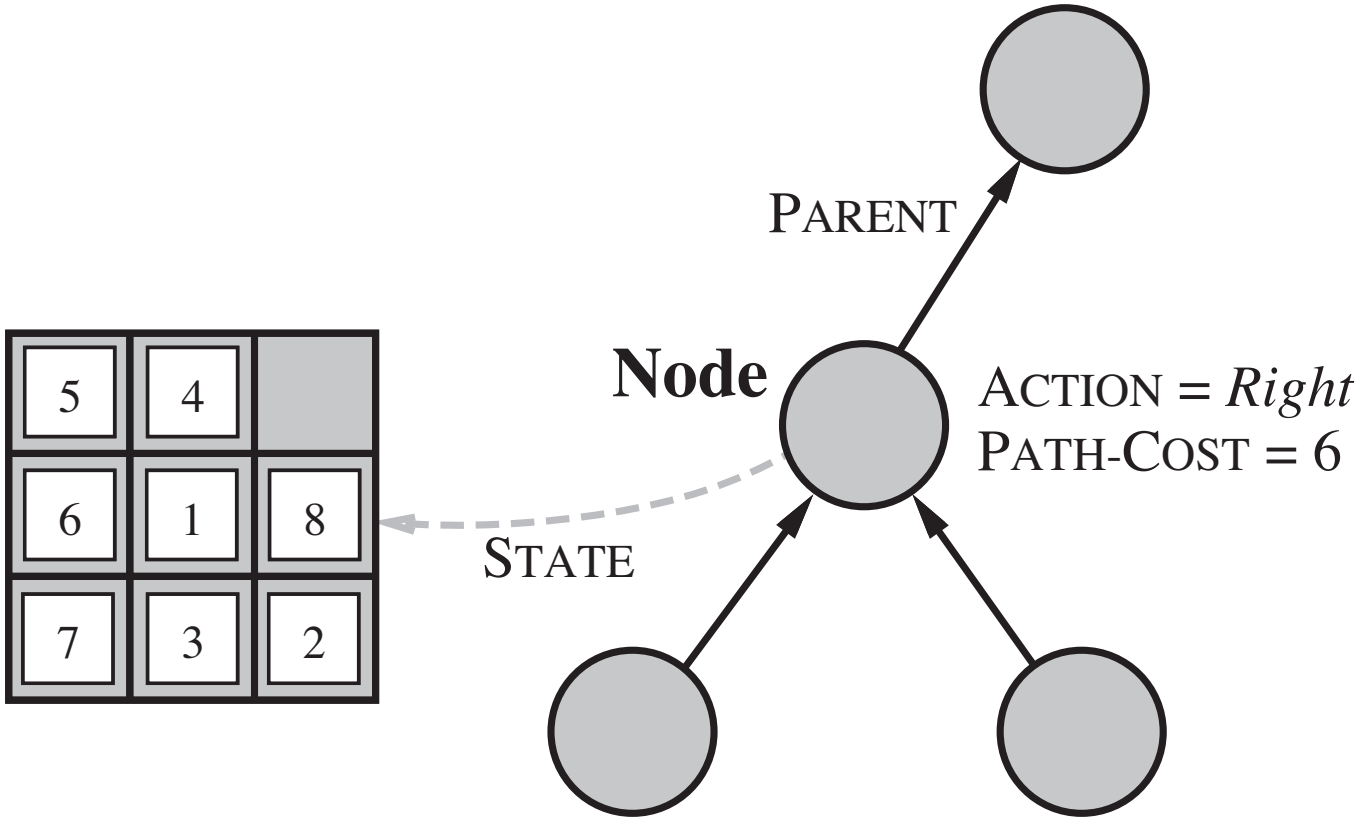


Abbildung 3: Knoten sind Datenstrukturen

```
function CHILD-NODE(problem, parent, action) returns a node
    return a node with
        STATE = problem.RESULT(parent.STATE, action),
        PARENT = parent, ACTION = action,
        PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

- Wichtig!!! Welcher ist der Unterschied zwischen Knoten und Zustand?

- Wichtig!!! Welcher ist der Unterschied zwischen Knoten und Zustand?
- Grenze wird als Warteschlange gespeichert
  - EMPTY?(*queue*)
  - POP(*queue*)
  - INSERT(*element*, *queue*)
- LIFO (Stapel), FIFO

BREITENSUCHE

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

Breitensuche

- Vollständige Suche für endlicher Verzweigungsfaktor
- Ist optimal falls die Pfadkosten eine stetig wachsende Funktion in der Tiefe des Knotens ist
- Zeitkomplexität (Anzahl der besuchten Knoten in der Tiefe d):

$$1 + b + b^2 + b^3 + \cdots + b^d + b(b^d - 1) = O(b^{d+1}).$$

- Speicherkomplexität: $O(b^{d+1})$.

Breitensuche

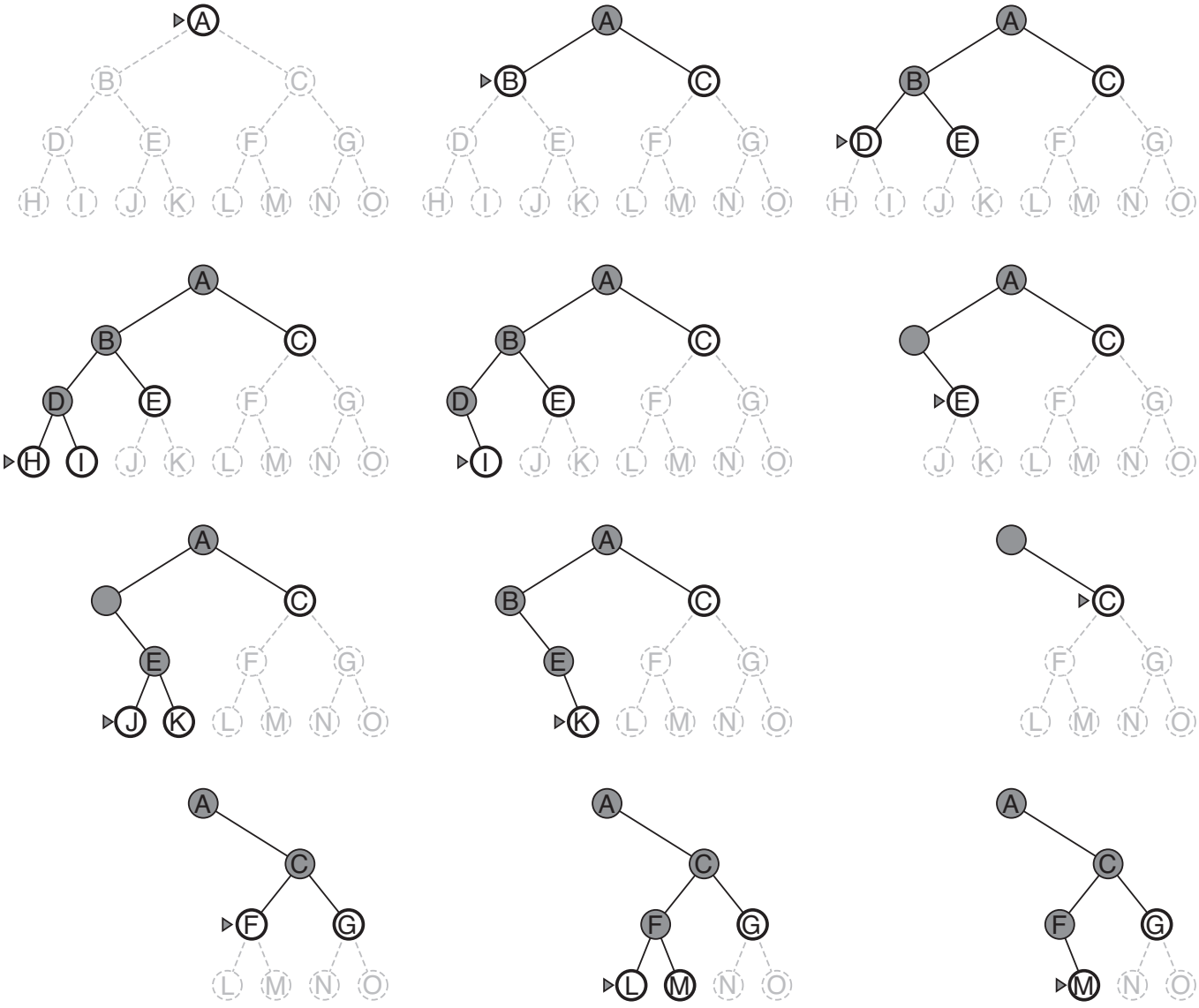
Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Abbildung 4: Verzweigungsfaktor $b = 10$, 1 Million Knoten/Sekunde, 1000 Bytes/Knoten

Gelernte Lektionen aus der BFS

- Der Speicherplatz ist ein viel größeres Problem als die Laufzeit...
- Exponentielle Suchprobleme können mit uninformierte Methoden nur für sehr kleine Instanzen gelöst werden.

Tiefensuche



Tiefensuche

- Falls der Algorithmus den falschen Pfad wählt, kann es sein, dass die Lösung nicht gefunden wird
  - nicht vollständig
  - kein Optimum
- Zeitkomplexität:
  - $O(b^m)$, wobei m die maximale Tiefe ist
  - Es kann vorkommen, dass $d << m$, wobei d die Tiefe des Optimum ist.
- Speicherkomplexität:
  - Speichert nur ein Pfad von der Wurzel zu einem Blatt, sowie die nicht expandierten Nachfolgerknoten
  - Expandierte Knoten werden entfernt, sobald alle Nachfolger erforscht wurden
  - $O(bm)$, wobei m die maximale Tiefe bezeichnet
  - Kann mittels Backtracking verringert werden
    - Erzeuge jeweils ein Nachfolger (statt alle) $\rightarrow O(m)$ Zustände
    - Modifiziere den laufenden Zustand, statt ihn zu kopieren $\rightarrow$ $O(1)$ Zustände und $O(m)$ Handlungen

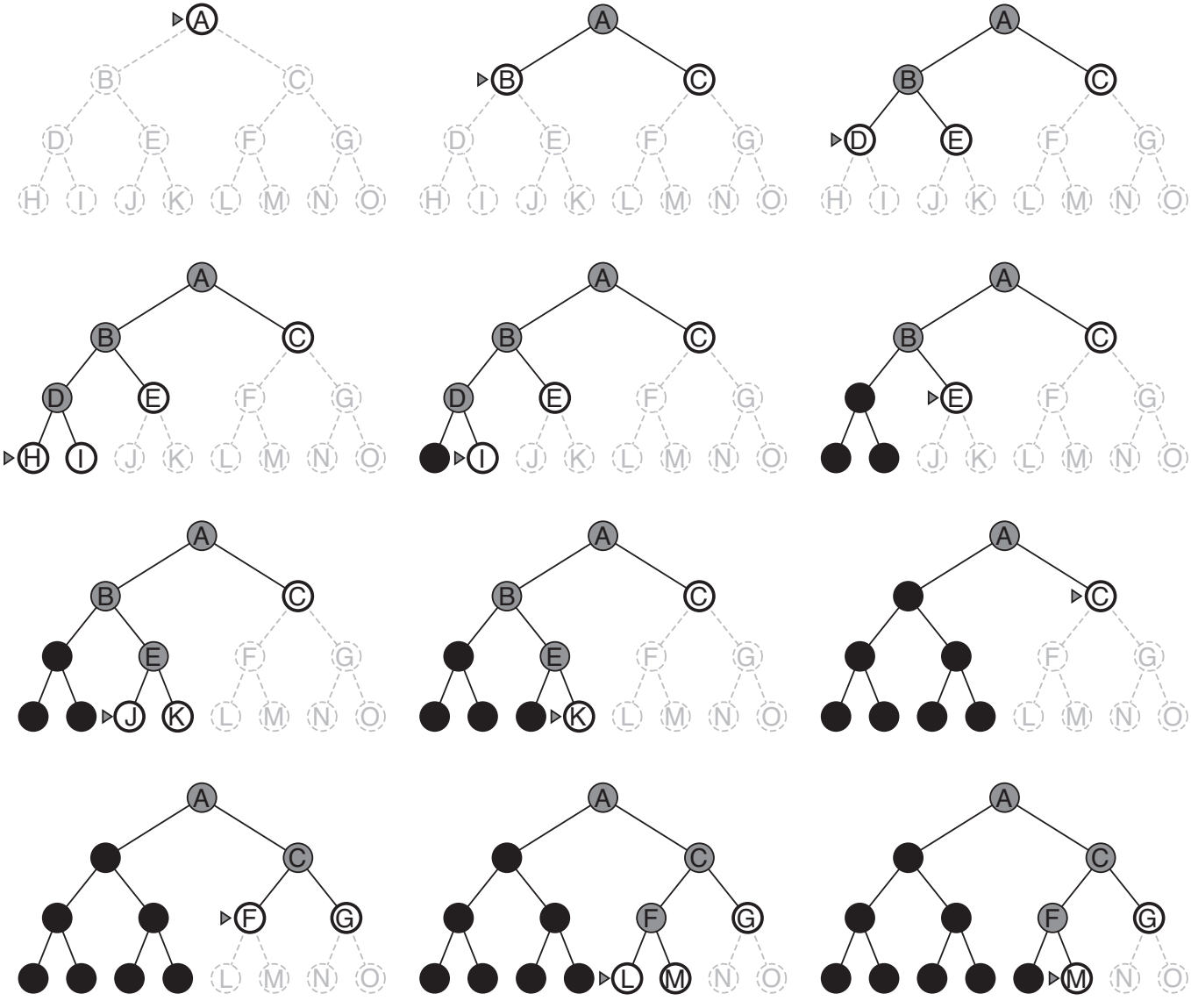
Eingeschränkte Tiefensuche

- Knoten in Tiefe I werden als nachfolgerfreie Knoten behandelt
- Wie bestimmen wir die Schranke?
  - Allgemeines Wissen über das Problem
  - Praktische Untersuchungen

Iterative Tiefensuche

- Tiefensuche ist nicht vollständig
- Wie macht man die Tiefensuche vollständig?
  - Suche ist vollständig
  - Suche ist optimal
  - Geringe Speicherkomplexität: $O(bd)$
  - Zeitkomplexität: $O(b^d)$ ist aber besser als BFS, welches ein weiteres Level untersucht
- Iterative Tiefensuche ist DIE Suchmetode für sehr große Suchräume und die Tiefe der Lösung ist unbekannt.

Iterative Tiefensuche



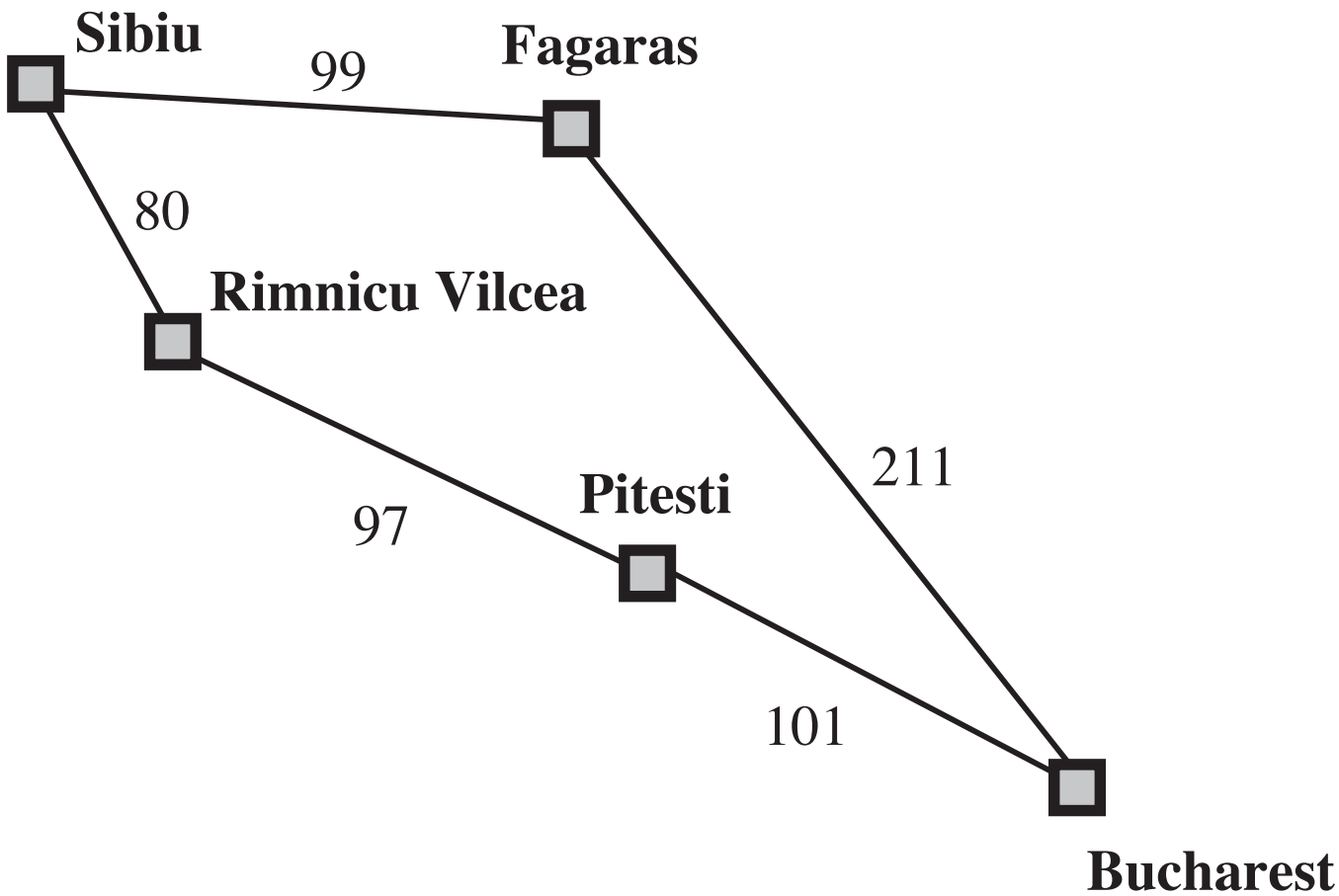
Uniforme Kostensuche

- Alle Handlungen haben gleichem Kost $\rightarrow$ BFS ist optimal
- Uniforme Kostensuche expandiert den Knoten mit dem kleinsten Pfadkosten $g(n)$.
- Dafür wird die Grenze als Prioritätsschlange gespeichert.

Uniforme Kostensuche

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

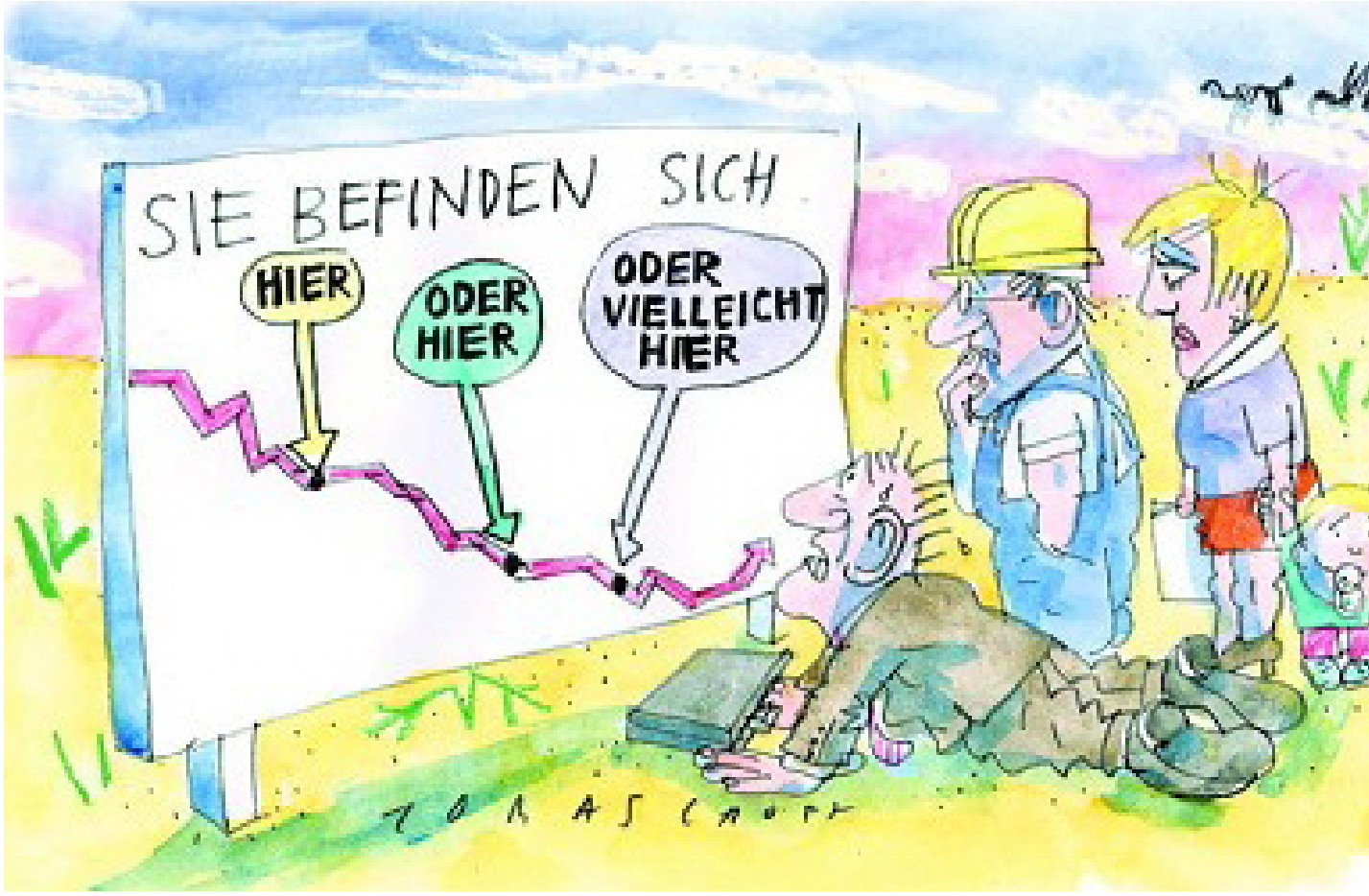
Uniforme Kostensuche



Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Abbildung 5: Vergleich der Suchstrategien: B Verzweigungsfaktor, d Tiefe der Lösung, m Tiefe der Suchbaums, l Schranke der Tiefensuche. [a] vollständig, falls b endlich, [b] vollständig für Kosten $\geq \epsilon > 0$, [c] optimal bei konstanten Kosten, [d] falls BFS in beiden Richtungen

Informierte Suchstrategien



Informierte Suchstrategien (ISS)

- Benutzt neben der Definition des Problems auch problemspezifisches Wissen.
- Findet Lösungen effizienter als eine uninformierte Strategie
- Der Suchraum ist eingeschränkt durch intelligente Auswahl der Knoten
- Dies funktioniert über einer Evaluierungsfunktion
- Diese Funktion ist immer problemspezifisch

Informierte Suchstrategien (ISS)

Definition

Die Suche nennt man informiert, wenn (zusätzlich) eine Bewertung aller Knoten des Suchraumes angegeben werden kann.

Knotenbewertung

- Schätzfunktion
- Ähnlichkeit zum Zielknoten oder auch Schätzung des Abstands zum Zielknoten
- Bewertung des Zielknotens: Sollte Maximum / Minimum der Schätzfunktion sein

Heuristische Suche

Definition

Eine Heuristik (Daumenregel) ist eine Schätzfunktion, die in vielen praktischen Fällen, die richtige Richtung zum Ziel angibt.

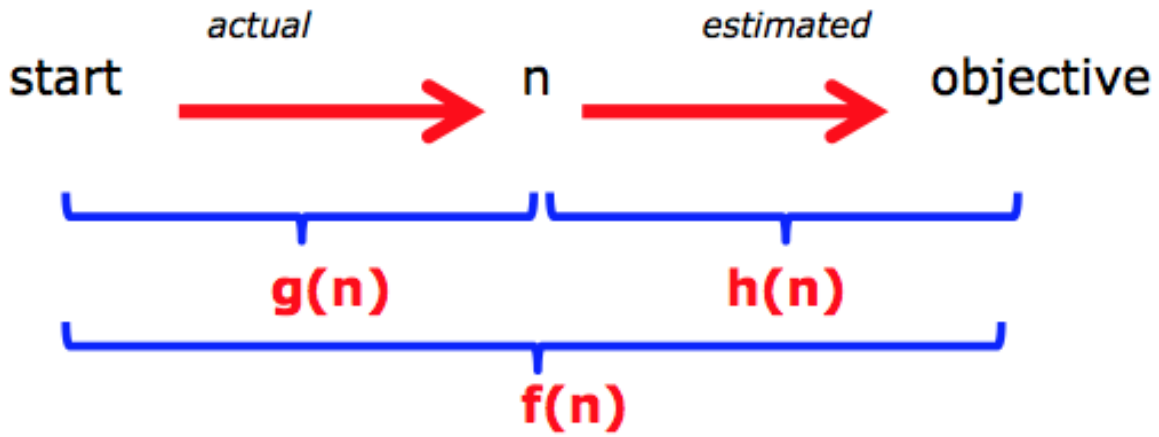
Suchproblem ist äquivalent zu:

- Minimierung (bzw. Maximierung) einer Knotenbewertung (einer Funktion) auf einem (implizit gegebenen) gerichteten Graphen
- Variante: Maximierung in einer Menge oder in einem n-dimensionaler Raum.

Suchstrategien in Bäumen

Evaluationsfunktionen

- $f(n)$ - Kosten durch Knoten (Zustand) n
- $h(n)$ - Kosten eines Lösungspfades von Knoten n zum Zielknoten
- $g(n)$ - Kosten eines Lösungspfades vom initialen Knoten zum Knoten n
- $f(n) = g(n) + h(n)$



8-Puzzle. Beispiel

Start:

8		1
6	5	4
7	2	3

Ziel:

1	2	3
4	5	6
7	8	

Bewertungsfunktionen (Beispiele):

1. $f_1()$ Anzahl der Plättchen an der falschen Stelle
2. $f_2()$ Anzahl der Züge (ohne Behinderungen zu beachten), die man braucht, um Endzustand zu erreichen.

8-Puzzle. Beispiel

$S_1 =$

2	3	1
8	7	6
	5	4

8-Puzzle. Beispiel

$$S_1 = \begin{array}{|c|c|c|}\hline 2 & 3 & 1 \\ \hline 8 & 7 & 6 \\ \hline & 5 & 4 \\ \hline \end{array} \qquad f_1(S_1) = 7$$

8-Puzzle. Beispiel

$$S_1 = \begin{array}{|c|c|c|}\hline 2 & 3 & 1 \\ \hline 8 & 7 & 6 \\ \hline & 5 & 4 \\ \hline \end{array}$$

$$f_1(S_1) = 7$$

$$f_2(S_1) = 2 + 1 + 1 + 3 + 1 + 0 + 2 + 2 = 12$$

8-Puzzle. Beispiel

$$S_1 = \begin{array}{|c|c|c|}\hline 2 & 3 & 1 \\ \hline 8 & 7 & 6 \\ \hline & 5 & 4 \\ \hline \end{array}$$

$$f_1(S_1) = 7$$

$$f_2(S_1) = 2 + 1 + 1 + 3 + 1 + 0 + 2 + 2 = 12$$

8-Puzzle. Beispiel

$$S_1 = \begin{array}{|c|c|c|}\hline 2 & 3 & 1 \\ \hline 8 & 7 & 6 \\ \hline & 5 & 4 \\ \hline \end{array}$$

$$f_1(S_1) = 7$$

$$f_2(S_1) = 2+1+1+3+1+0+2+2 = 12$$

8-Puzzle. Beispiel

$$S_1 = \begin{array}{|c|c|c|}\hline 2 & 3 & 1 \\ \hline 8 & 7 & 6 \\ \hline & 5 & 4 \\ \hline \end{array}$$

$$f_1(S_1) = 7$$
$$f_2(S_1) = 2+1+1+3+1+0+2+2 = 12$$

8-Puzzle. Beispiel

$$S_1 = \begin{array}{|c|c|c|}\hline 2 & 3 & 1 \\ \hline 8 & 7 & 6 \\ \hline & 5 & 4 \\ \hline \end{array}$$

$$f_1(S_1) = 7$$

$$f_2(S_1) = 2+1+1+3+1+0+2+2 = 12$$

8-Puzzle. Beispiel

$$S_1 = \begin{array}{|c|c|c|}\hline 2 & 3 & 1 \\ \hline 8 & 7 & 6 \\ \hline & 5 & 4 \\ \hline \end{array}$$

$$f_1(S_1) = 7$$

$$f_2(S_1) = 2+1+1+3+1+0+2+2 = 12$$

8-Puzzle. Beispiel

$$S_1 = \begin{array}{|c|c|c|}\hline 2 & 3 & 1 \\ \hline 8 & 7 & 6 \\ \hline & 5 & 4 \\ \hline \end{array}$$

$$f_1(S_1) = 7$$

$$f_2(S_1) = 2+1+1+3+1+0+2+2 = 12$$

8-Puzzle. Beispiel

$$S_1 = \begin{array}{|c|c|c|}\hline 2 & 3 & 1 \\ \hline 8 & 7 & 6 \\ \hline & 5 & 4 \\ \hline \end{array}$$

$$f_1(S_1) = 7$$

$$f_2(S_1) = 2+1+1+3+1+0+2+2 = 12$$

8-Puzzle. Beispiel

$$S_1 = \begin{array}{|c|c|c|}\hline 2 & 3 & 1 \\ \hline 8 & 7 & 6 \\ \hline & 5 & 4 \\ \hline \end{array}$$

$$f_1(S_1) = 7$$
$$f_2(S_1) = 2+1+1+3+1+0+2+2 = 12$$

$$S_2 = \begin{array}{|c|c|c|}\hline 2 & 3 & 1 \\ \hline & 7 & 6 \\ \hline 8 & 5 & 4 \\ \hline \end{array}$$

$$f_1(S_2) = 7$$
$$f_2(S_2) = 2+1+1+3+1+0+2+1 = 11$$

8-Puzzle. Beispiel

$S_1 =$

2	3	1
8	7	6
	5	4

$f_1(S_1) = 7$

$f_2(S_1) = 2+1+1+3+1+0+2+2 = 12$

$S_2 =$

2	3	1
	7	6
8	5	4

$f_1(S_2) = 7$

$f_2(S_2) = 2+1+1+3+1+0+2+1 = 11$

$\Rightarrow f_2$ ist genauer.

Suchstrategien in Bäumen: Tree Search

Wiederholung

```
Funktion Tree - Search (Problem) gibt eine
Lösung oder einen Fehler zur''uck
Die Grenzknoten mit dem Ausgangszustand von
Problem initialisieren
loop do
if die Grenze ist leer then return Fehler
Wähle einen Blattknoten aus und entferne ihn
aus den Grenzknoten
if Knoten enthält einen Zielzustand then
return die entsprechende Lösung
Knoten expandieren und der Grenze die
resultierenden Knoten hinzufügen
```

Suchstrategien in Bäumen: Graph-Search

Wiederholung

```
Funktion Graph-Search(Problem) gibt eine
Lösung oder einen Fehler zur''uck
Die Grenzknoten mit dem Ausgangszustand von
Problem initialisieren
Die untersuchte Menge als leer initialisieren
if die Grenze ist leer then return Fehler
Wähle einen Blattknoten aus und entferne ihn
aus den Grenzknoten
if Knoten enthält einen Zielzustand then
return die entsprechende Lösung
Füge den Knoten zur untersuchten Menge hinzu
Knoten expandieren und der Grenze die
resultierenden Knoten hinzufügen nur, wenn
nicht in der Grenze der untersuchten Menge
```

ISS: BESTENSUCHE (BEST-FIRST)

GREEDY SUCHE

- Ist eine Instanz des TREE-SEARCH oder GRAPH-SEARCH Algorithmus und wählt einen Knoten auf Basis einer Evaluierungsfunktion $f(n)$ zur Expandierung aus.
- Evaluierungsfunktion: Kostenabschätzungsfunktion - der Knoten mit der *geringsten* Bewertung wird zuerst erweitert.
- Die Wahl von f bestimmt die Suchstrategie
  - Beispiel: Suche mit einheitlichen Kosten $f = Pfadkosten$
- Strategien
  - Expandiere den Knoten der dem Ziel am nächsten liegt
  - Suche die billigste/teuerste Lösung

ISS: Bestensuche (Best-first)

Algorithmus **Best-First Search**

Datenstrukturen:
Sei L Liste von Knoten, markiert mit dem Weg dorthin.
h sei die Bewertungsfunktion der Knoten

Eingabe: L Liste der initialen Knoten, sortiert, so dass die besseren Knoten vorne sind.

Algorithmus:

1. Wenn L leer ist, dann breche ab
2. Sei K der erste Knoten von L und R die Restliste.
3. Wenn K ein Zielknoten ist, dann gebe K und den Weg dahin aus.
4. Sei $N(K)$ die Liste der Nachfolger von K. Entferne aus $N(K)$ die bereits im Weg besuchten Knoten mit Ergebnis $\mathcal{N}$
5. Setze $L := \mathcal{N}$ ++ R
6. Sortiere L, so dass bessere Knoten vorne sind und gehe zu 1.

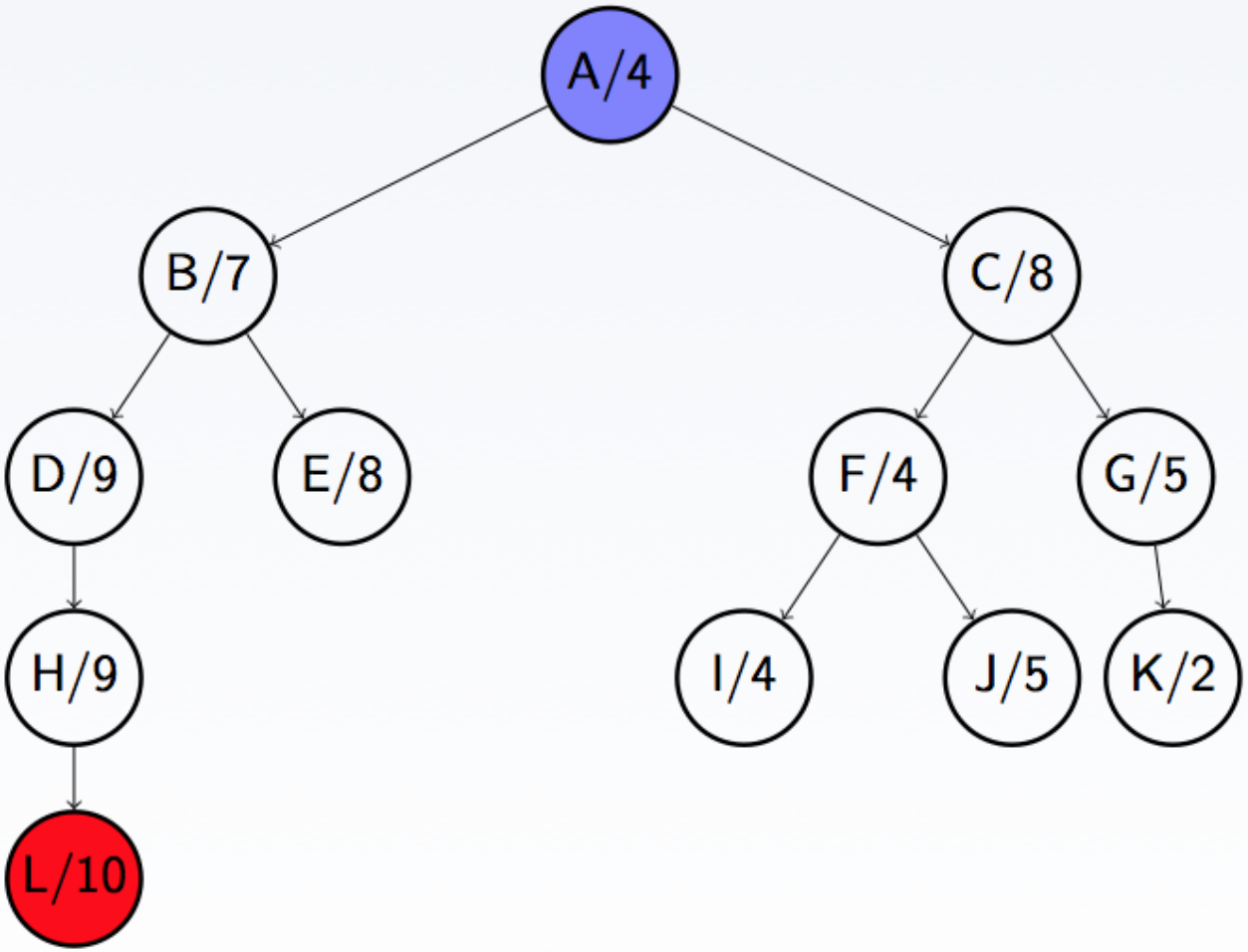
ISS: BESTENSUCHE (BEST-FIRST)

- Komplexität:
  - b Verzweigungsfaktor, d maximale Tiefe der Lösung
  - Zeitkomplexität $T(n) = 1 + b + b^2 + \cdots + b^d \Rightarrow O(b^d)$
  - Speicherkomplexität: $S(n) = T(n)$
  - Vollständigkeit: NEIN - unendliche Pfade, falls jeder einzelne Knoten als beste Lösung evaluiert wird
  - Optimalität - abhängig von der Heuristik
- Vorteile:
  - Spezifische Information macht die Suche schneller
  - Gute Geschwindigkeit
- Nachteile: Evaluierung der Zustände erfordert Ressoursen (komputationelle, physische...)
- Anwendungen: Web crawler (automatic indexer), Spiele

ISS: BESTENSUCHE (BEST-FIRST)

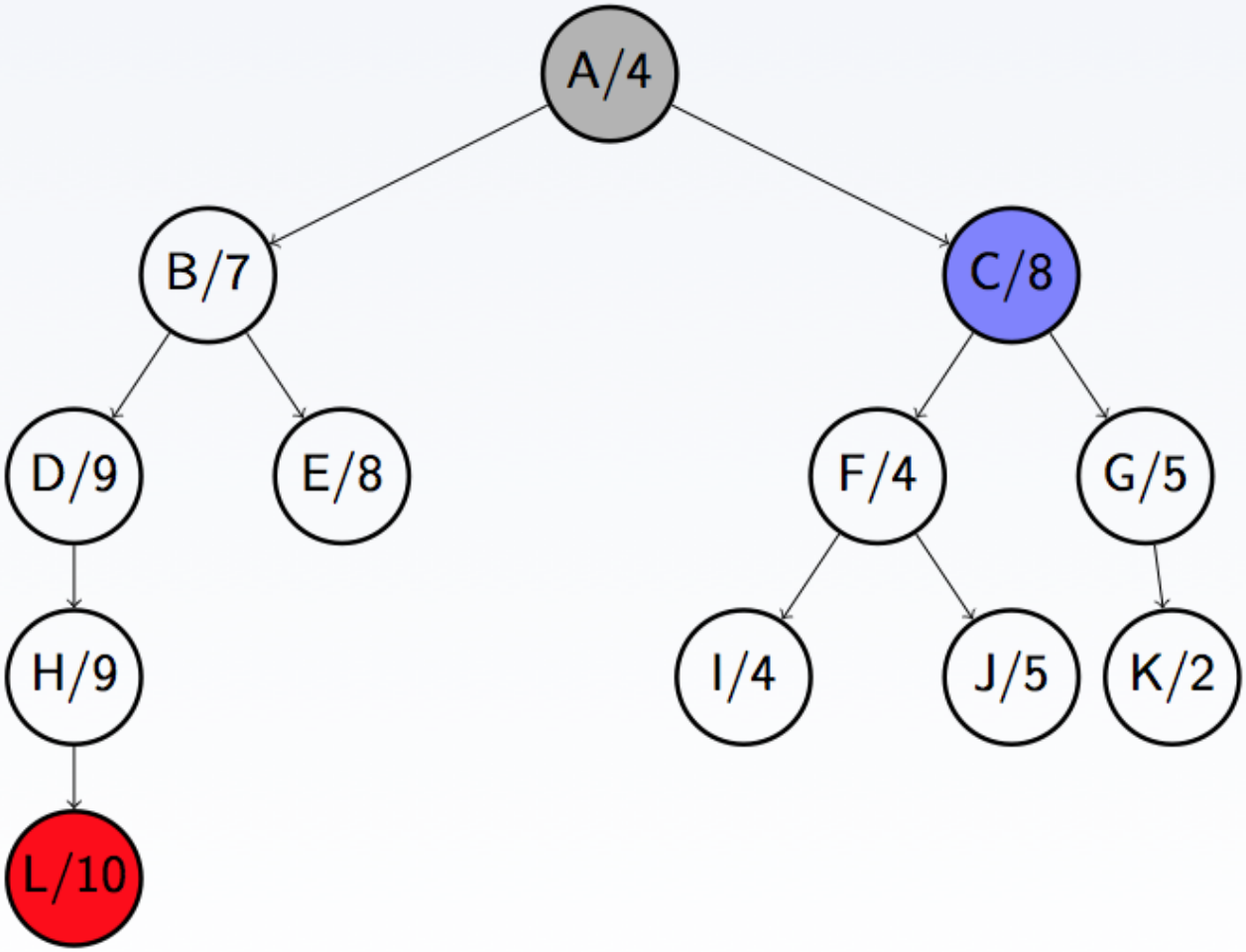
```
bool BestFS(elem, list){
     found = false;
     visited = Φ;
     toVisit = {start};   //FIFO sorted list (priority queue)
     while((toVisit != Φ) && (!found)){
          if (toVisit == Φ)
            return false
          node = pop(toVisit);
          visited = visited U {node};
          if (node == elem)
            found = true;
          else
            aux = Φ;
          for all unvisited children of node do{
            aux = aux U {child};
          }
          toVisit = toVisit U aux; //adding a node into the FIFO list based on its
                                   // evaluation (best one in the front of list)

     }  //while
     return found;
}
```

Beispiel: Bestensuche (Best-first)



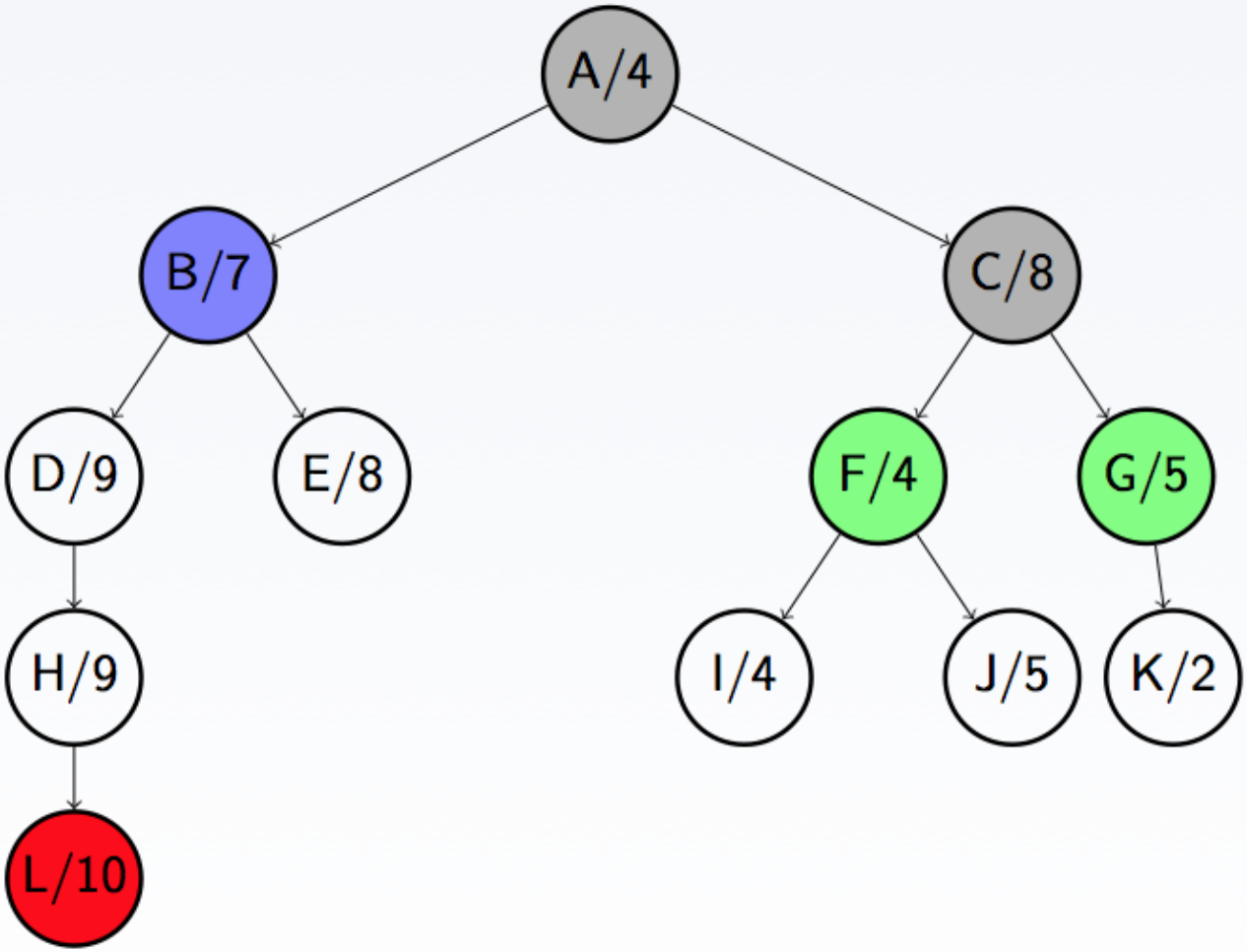
$L = [A]$

Beispiel: Bestensuche (Best-first)



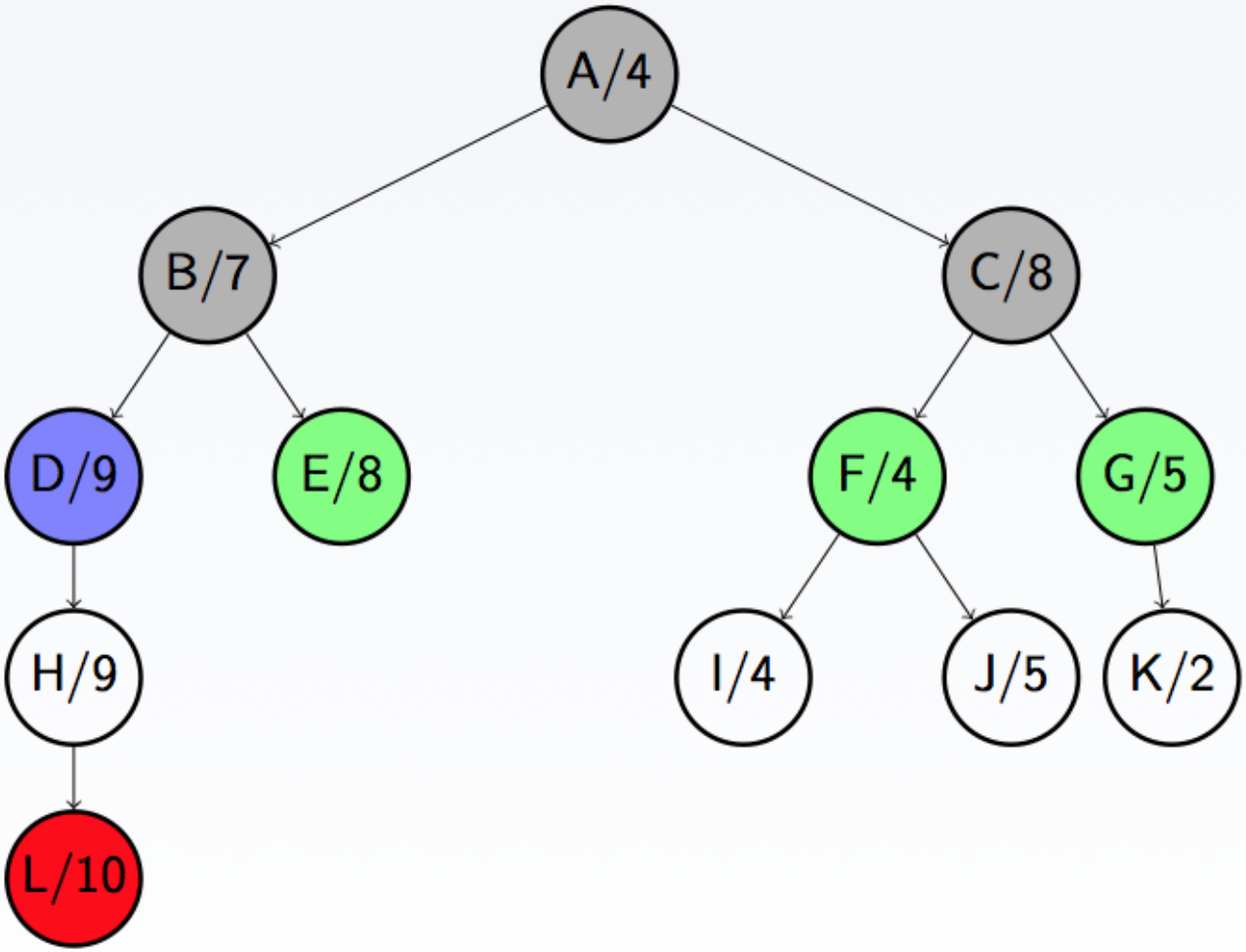
L = sort ([C,B] ++ []) = [C,B]

Beispiel: Bestensuche (Best-first)



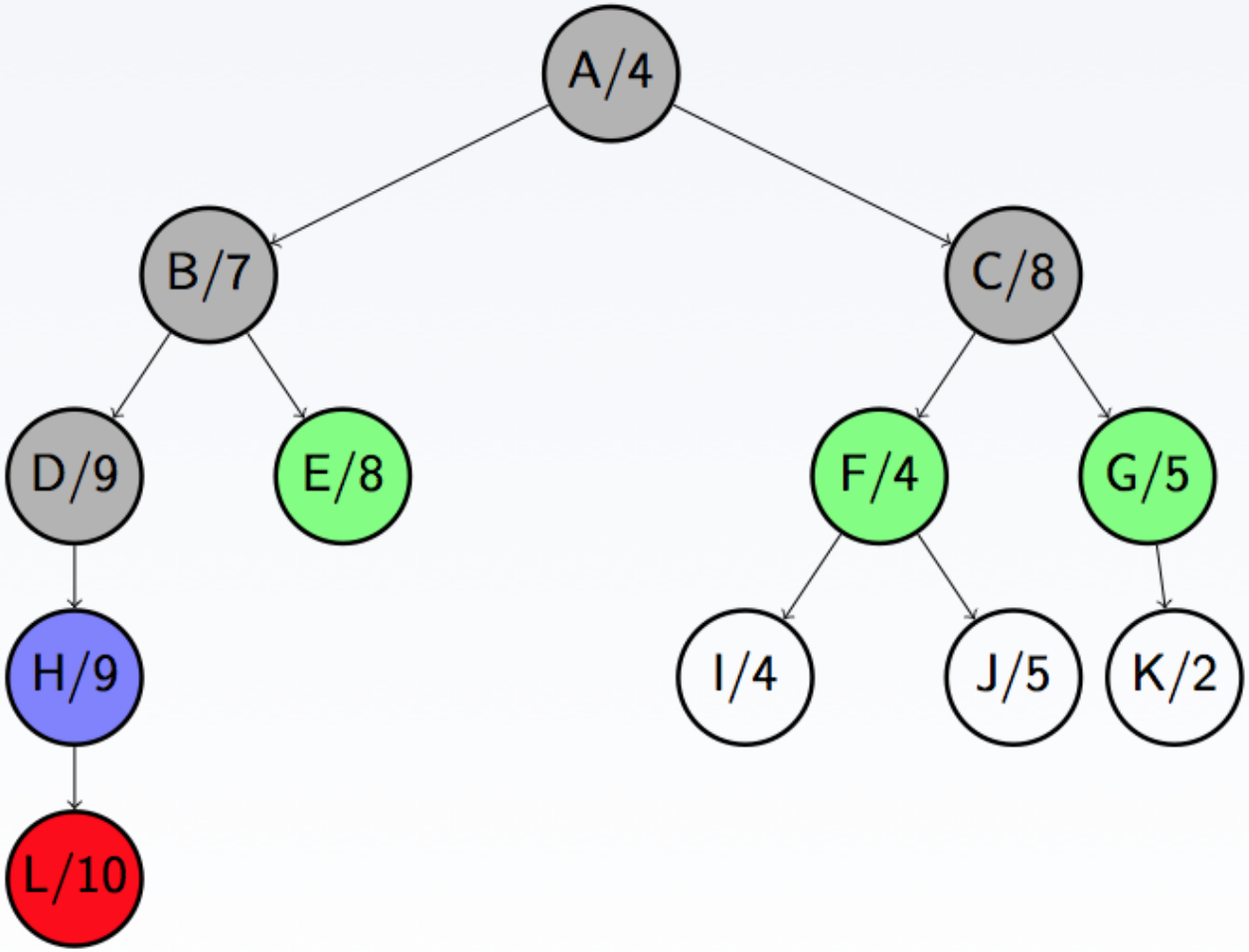
L = sort ([G,F] ++ [B]) = [B,G,F]

Beispiel: Bestensuche (Best-first)



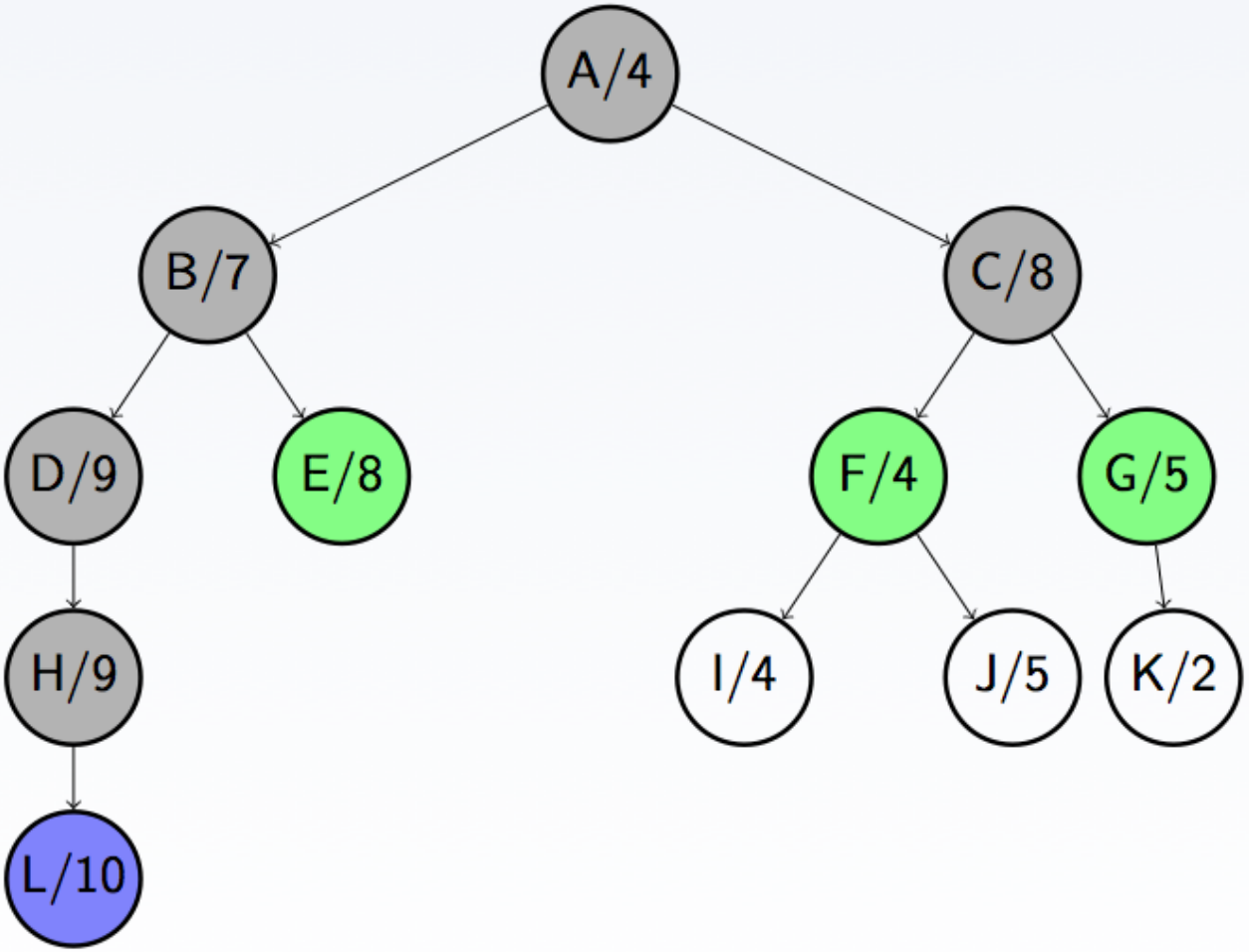
L = sort ([D,E] ++ [G,F]) = [D,E,G,F]

Beispiel: Bestensuche (Best-first)



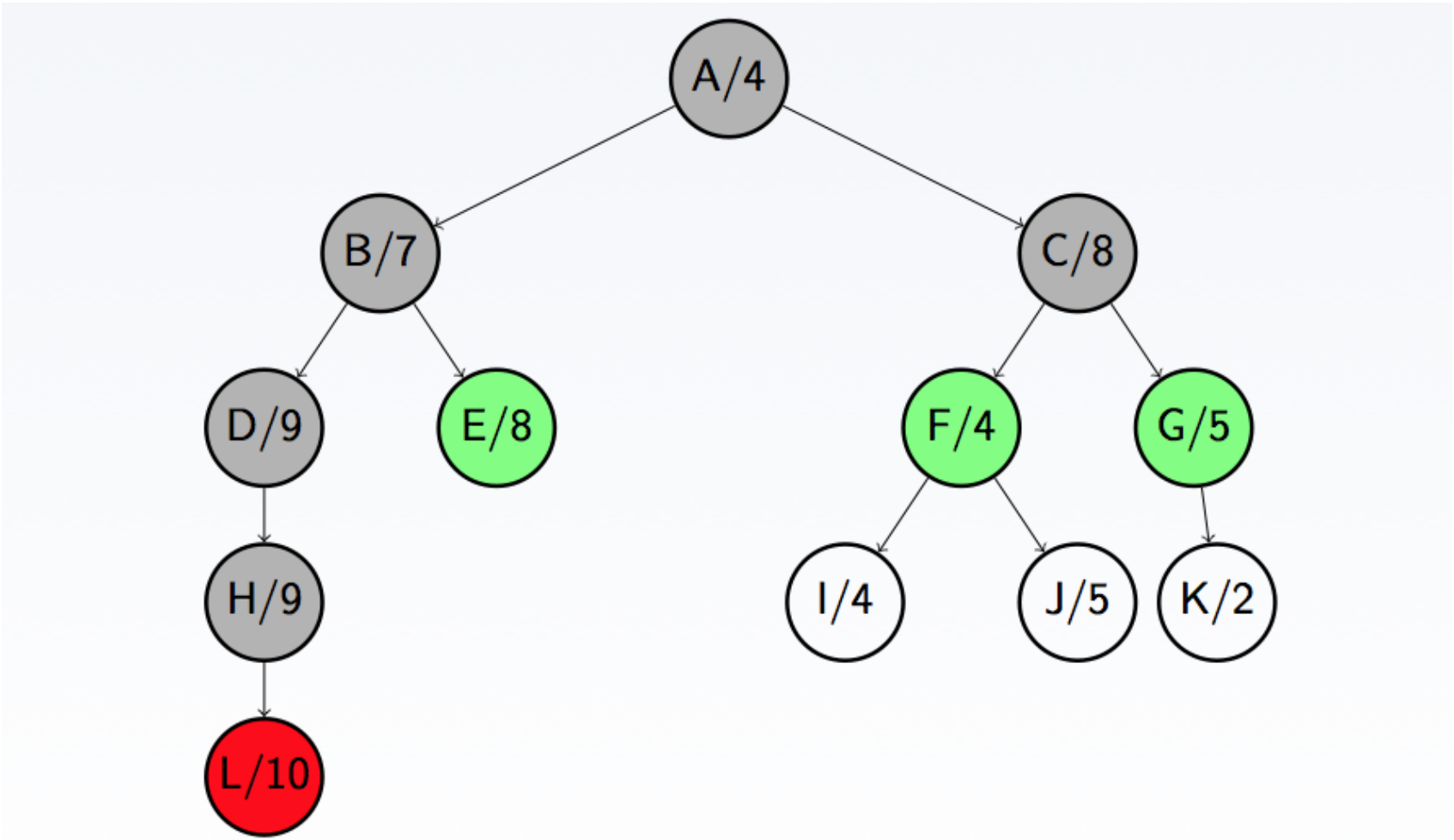
L = sort ([H,E] ++ [G,F] = [H,E,G,F]

Beispiel: Bestensuche (Best-first)



L = sort ([L] ++ [E,G,F] = [L,E,G,F]

8-Puzzle. Beispiel



Zielknoten L gefunden

Best-First-Suche: Eigenschaften

- entspricht einer gesteuerten Tiefensuche daher unvollständig
- Platzbedarf ist durch die Speicherung der besuchten Knoten exponentiell in der Tiefe.
- Durch Betrachtung aller Knoten auf dem Stack können lokale Maxima schneller verlassen werden, als beim Hill-Climbing

BFS - WEITERE BEISPIELE

- Missionäre und Kanibale: $h(n)$ Anzahl der Personen auf dem initialen Ufer
- 8-Puzzle: $h(n)$ Anzahl der Steine am falschen Platz ODER $h(n)$ - Manhatten Entfernung (Entfernung zum Ziel)
- Problem des Handlungsreisenden: $h(n)$ Entfernung zum nächsten Nachbar
- Auszahlen einer Summe mit minimaler Anzahl von Münzen: $h(n)$ wähle einen Münzwert kleiner als die auszuzahlende Summe

GREEDY ISS

- Komplexität
  - Zeit Komplexität → DFS
  - b Verzweigungsfaktor, d_{max} maximale Tiefe der Suche
  - $T(n) = 1 + b + b^2 + \cdots + b^{d_{max}}$
  - $T(n) = 1 + b + b^2 + \cdots + b^{d_{max}} \Rightarrow O(b^{d_{max}})$
- Speicherkomplexität → BFS: d - Tiefe der Lösung, dann $S(n) = 1 + b + b^2 + \cdots + b^d \Rightarrow O(b^{d_{max}})$
- Vollständigkeit: NEIN; Optimalität: möglich
- Vorteile: findert schnell eine Lösung (nicht immer optimal), insbesondere für kleine Datensätze
- Nachteile: Die Summe der optimalen lokalen Lösungen $\neq$ globale optimale Entscheidung. Ex. TSP
- Anwendungen:
  - Planungsaufgaben
  - partielle Summen: Münzen, Rucksack
  - Puzzles
  - Optimale Pfade in Graphen

Greedy ISS: Beispiel

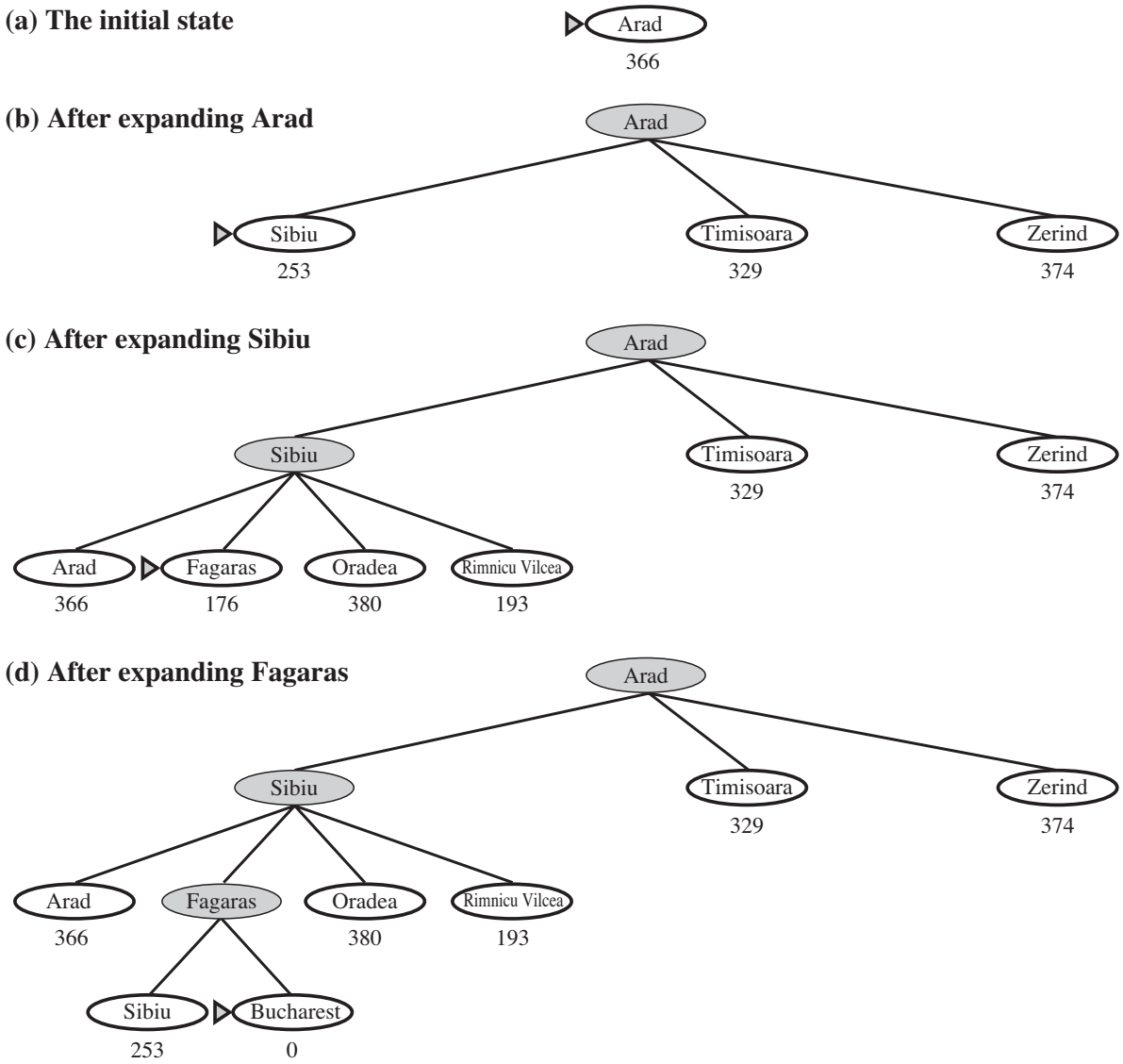
Luftlinien Distanz Heuristik

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

Abbildung 6: h_{SLD} Heuristikwerte - Entfernung nach Bukarest

Greedy ISS: Beispiel

Luftlinien Distanz Heuristik



Optimalitätsbedingungen

- Zulässigkeit: Die Heuristik überschätzt nie die Kosten.
- Konsistenz (Monotonie): Für jeden Knoten n und jeden Nachfolger n' erzeugt durch eine Handlung a, die abgeschätzten Kosten um das Goal aus n zu erreichen sind nicht größer als die Kosten, die notwendig sind um n' zu erreichen plus der Abschätzung der Kosten um das Goal von n' aus zu erreichen:

$$h(n) \leq c(n, a, n') + h(n').$$

Bergsteigerprozedur (Hill-climbing)

- Auch als Gradientenaufstieg bekannt
- Gradient: Richtung der Vergrößerung einer Funktion (Berechnung durch Differenzieren)
- Parameter der Bergsteigerprozedur
  - Menge der initialen Knoten
  - Nachfolgerfunktion (Nachbarschaftsrelation)
  - Bewertungsfunktion der Knoten, wobei wir annehmen, dass Zielknoten maximale Werte haben (Minimierung erfolgt analog)
  - Zieltest

Bergsteigen

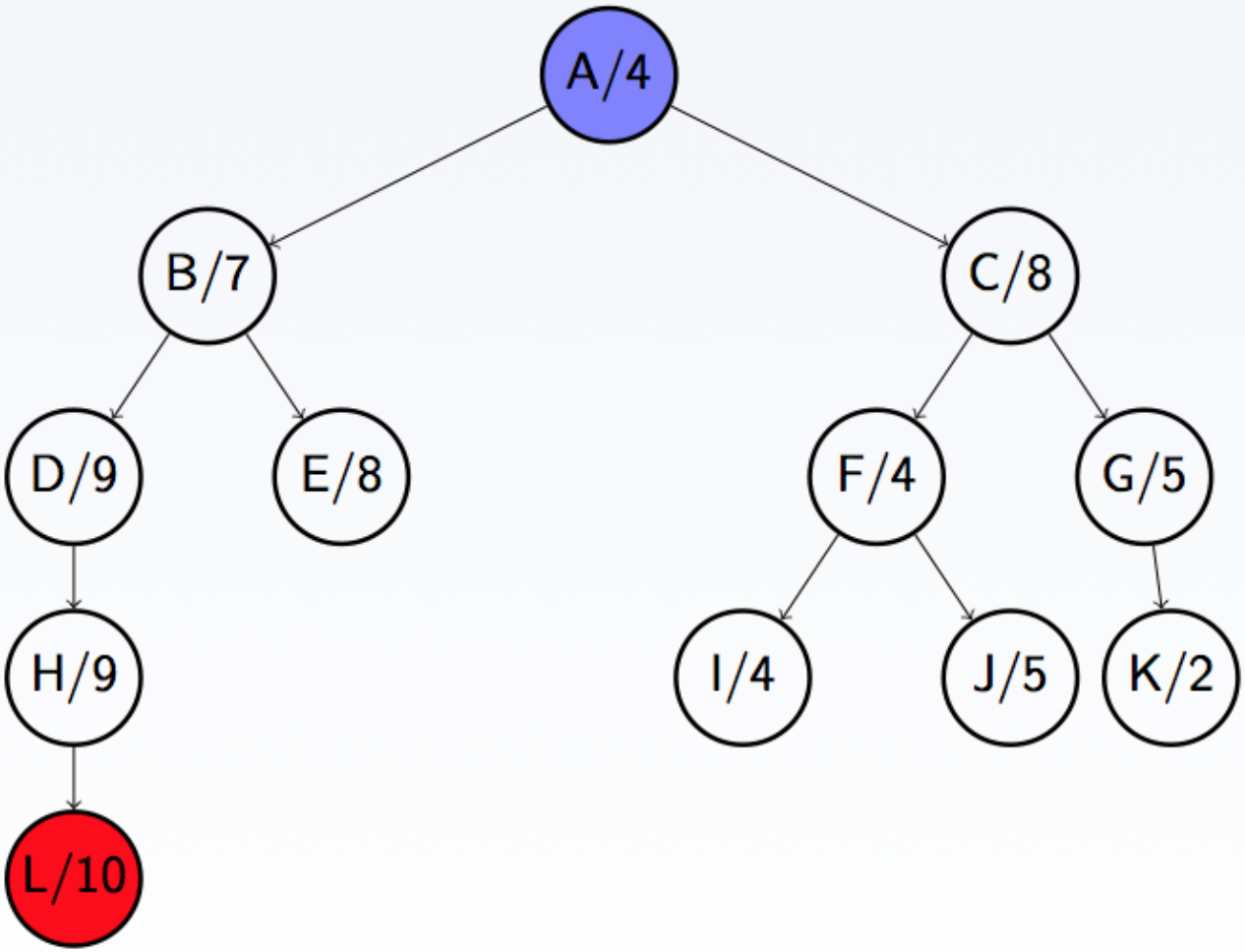
Algorithmus **Bergsteigen**

Datenstrukturen: L : Liste von Knoten, markiert mit Weg dorthin
h sei die Bewertungsfunktion der Knoten
Eingabe: L sei die Liste der initialen Knoten, absteigend sortiert entsprechend h

Algorithmus:

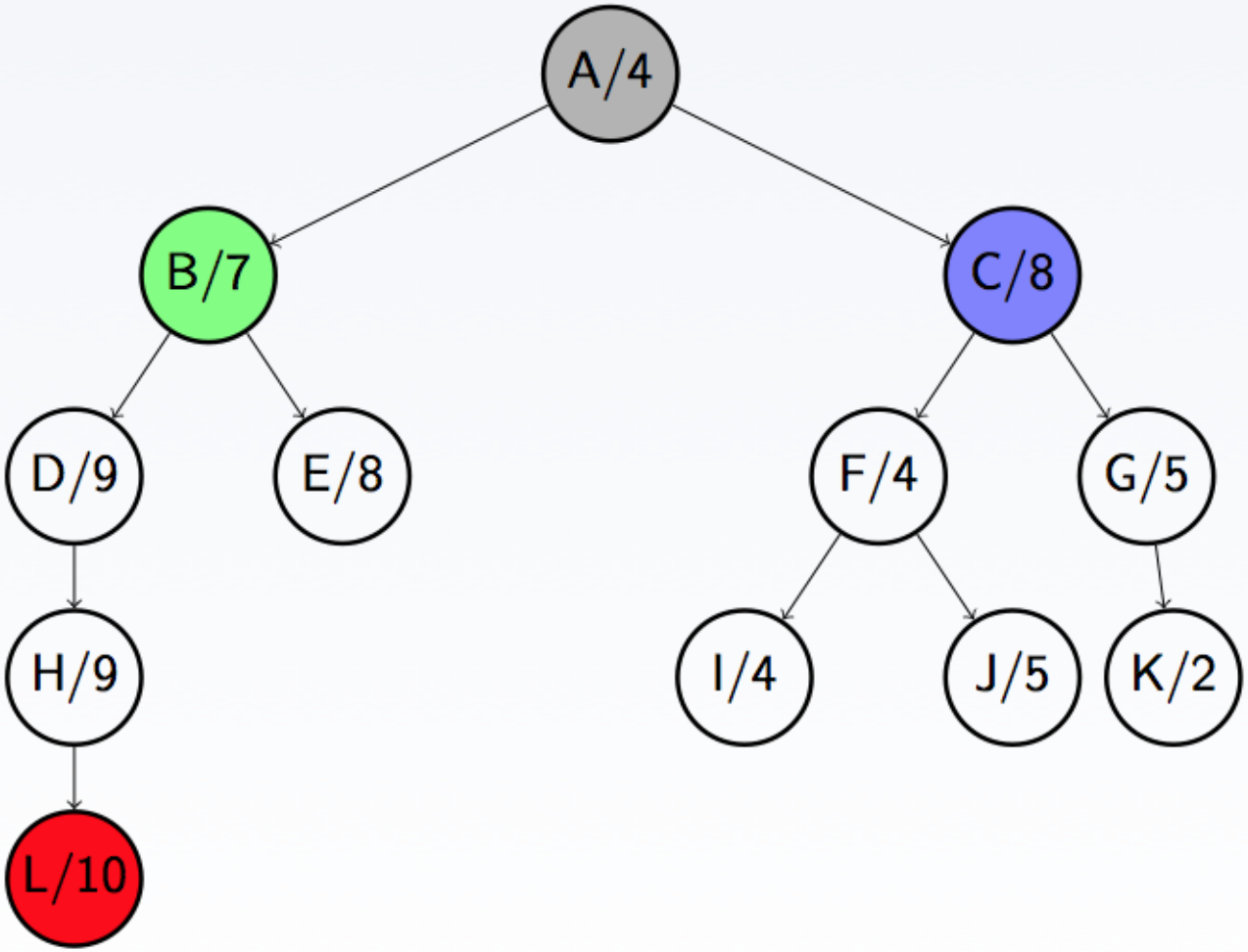
1. Sei K das erste Element von L und R die Restliste
2. Wenn K ein Zielknoten, dann stoppe und gebe K markiert mit dem Weg zurück
3. Sortiere die Liste $NF(K)$ absteigend entsprechend h und entferne schon besuchte Knoten aus dem Ergebnis. Sei dies die Liste L'.
4. Setze $L := L'$++R und gehe zu 1.

Beispiel: Bergsteigen



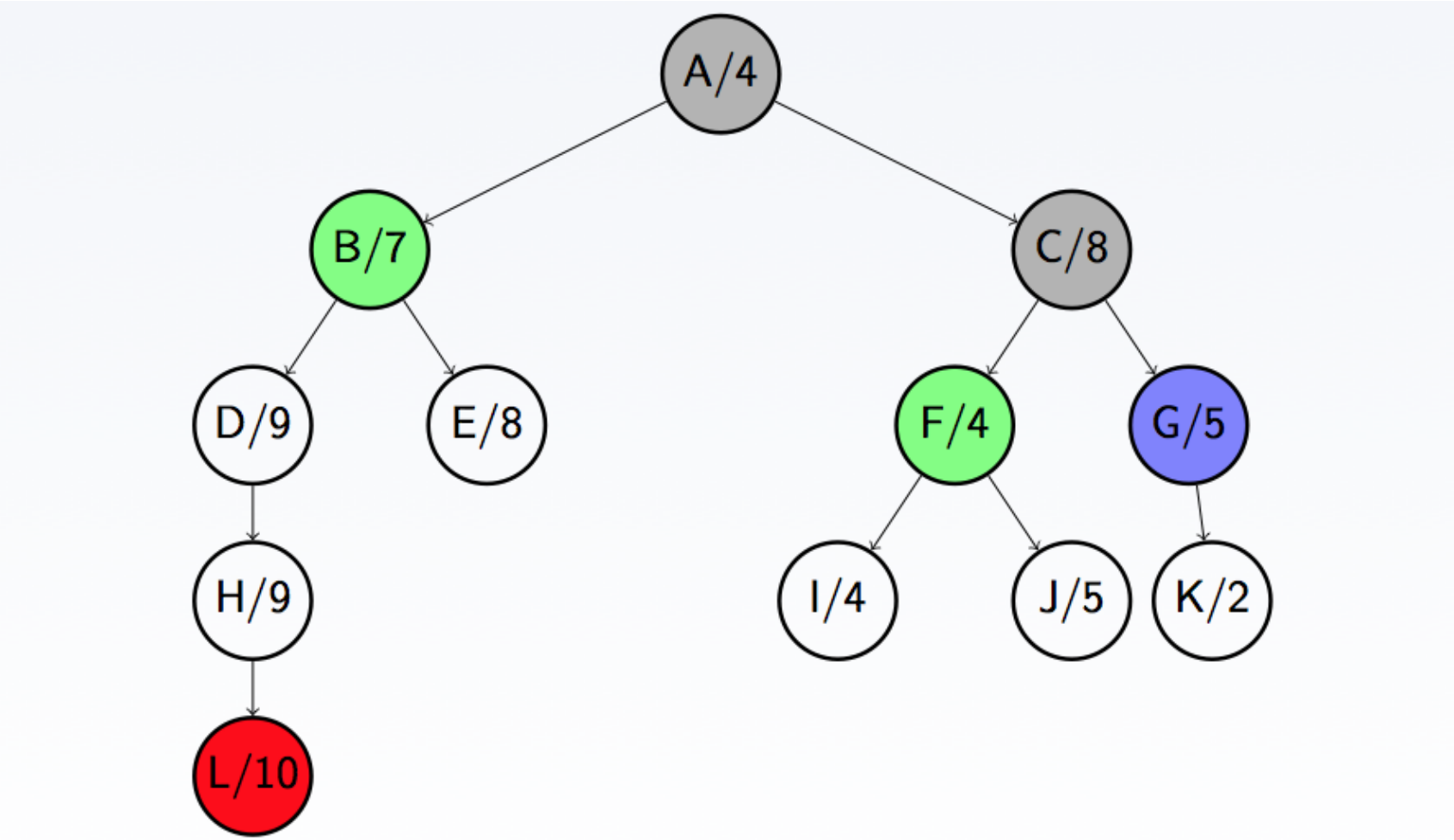
$L = [A]$

Beispiel: Bergsteigen



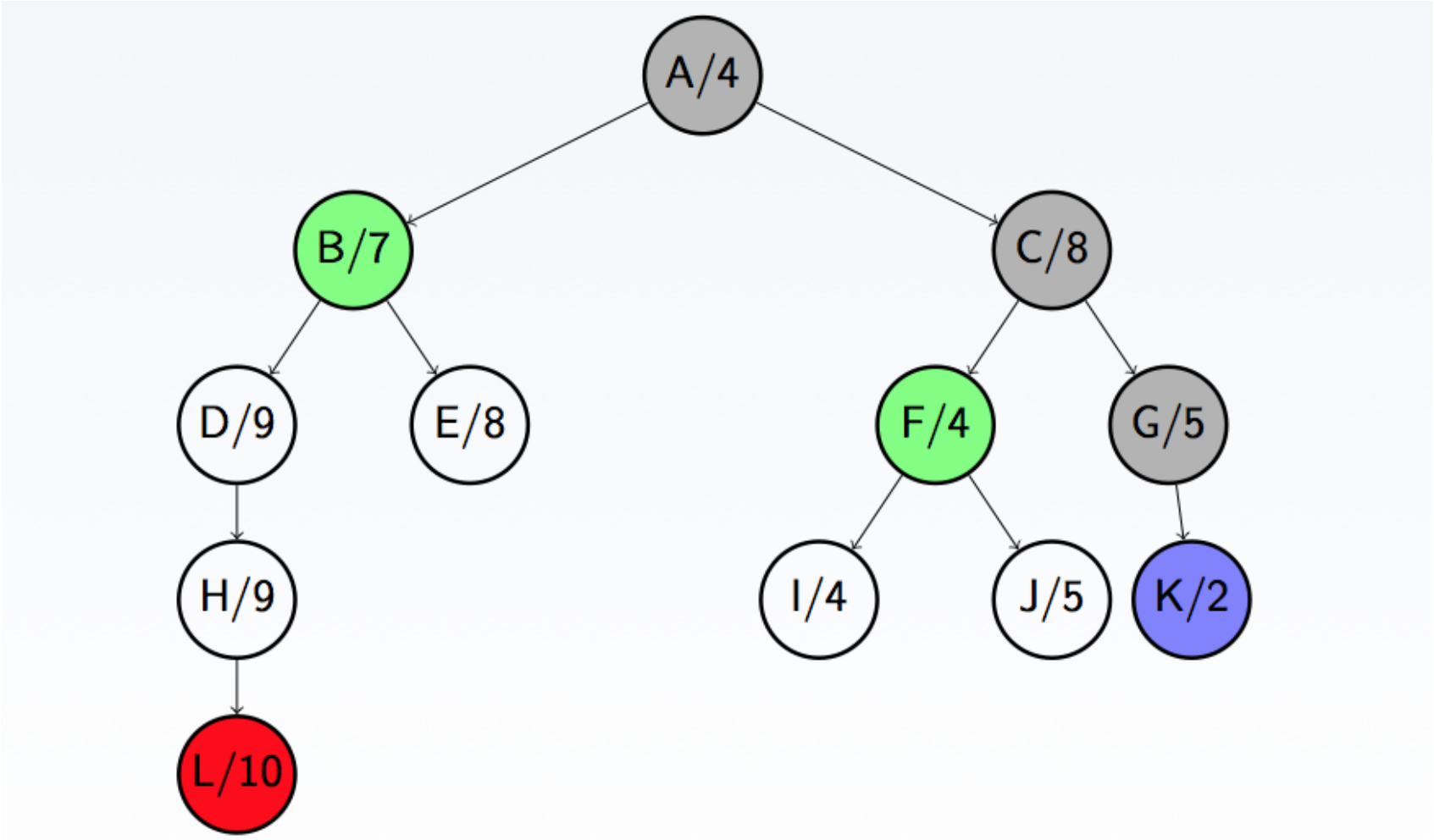
$$L = [C,B] \text{ ++ } [] = [C,B]$$

Beispiel: Bergsteigen



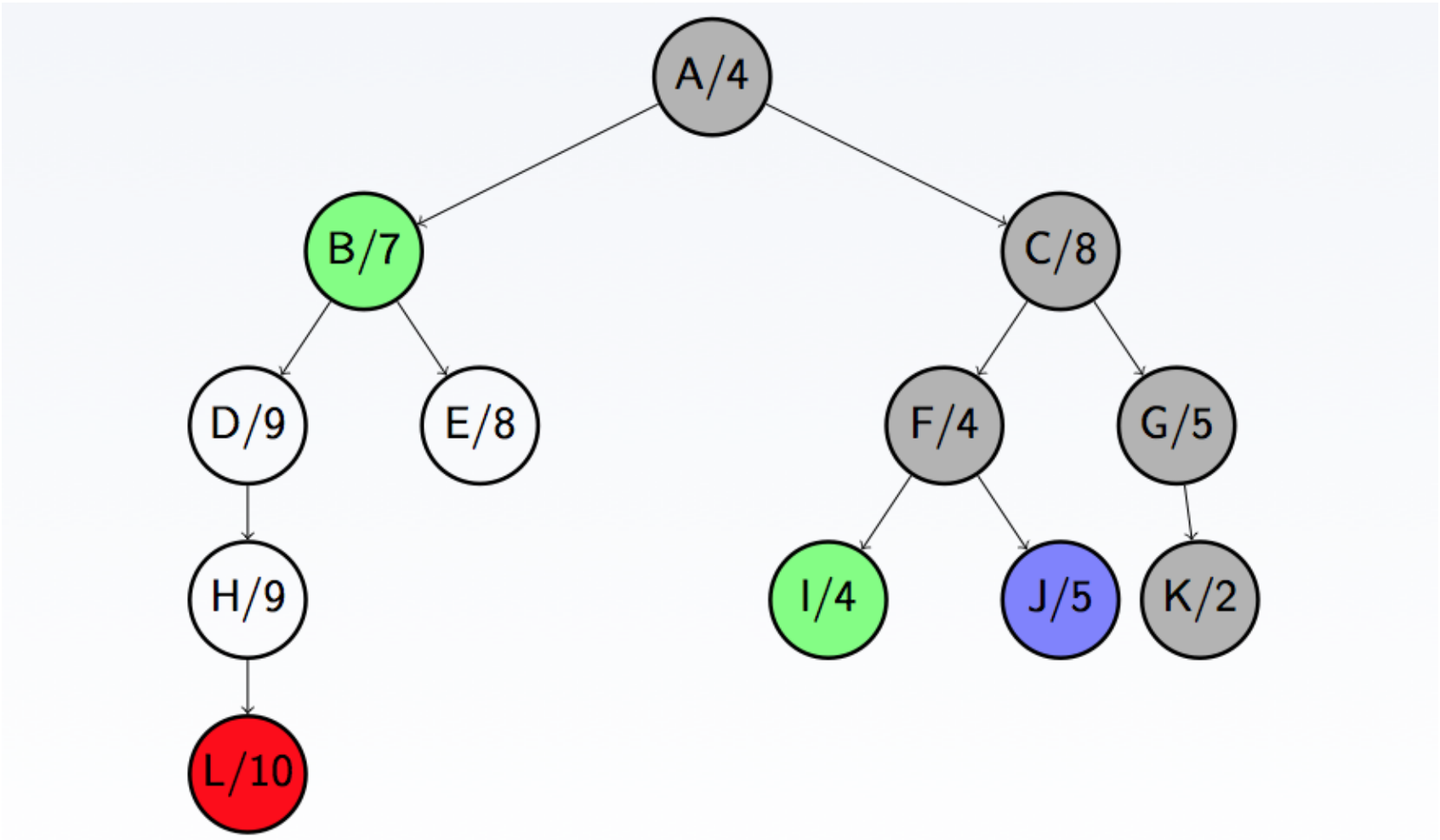
$$L = [G,F] \mathbin{++} [B] = [G,F,B]$$

Beispiel: Bergsteigen



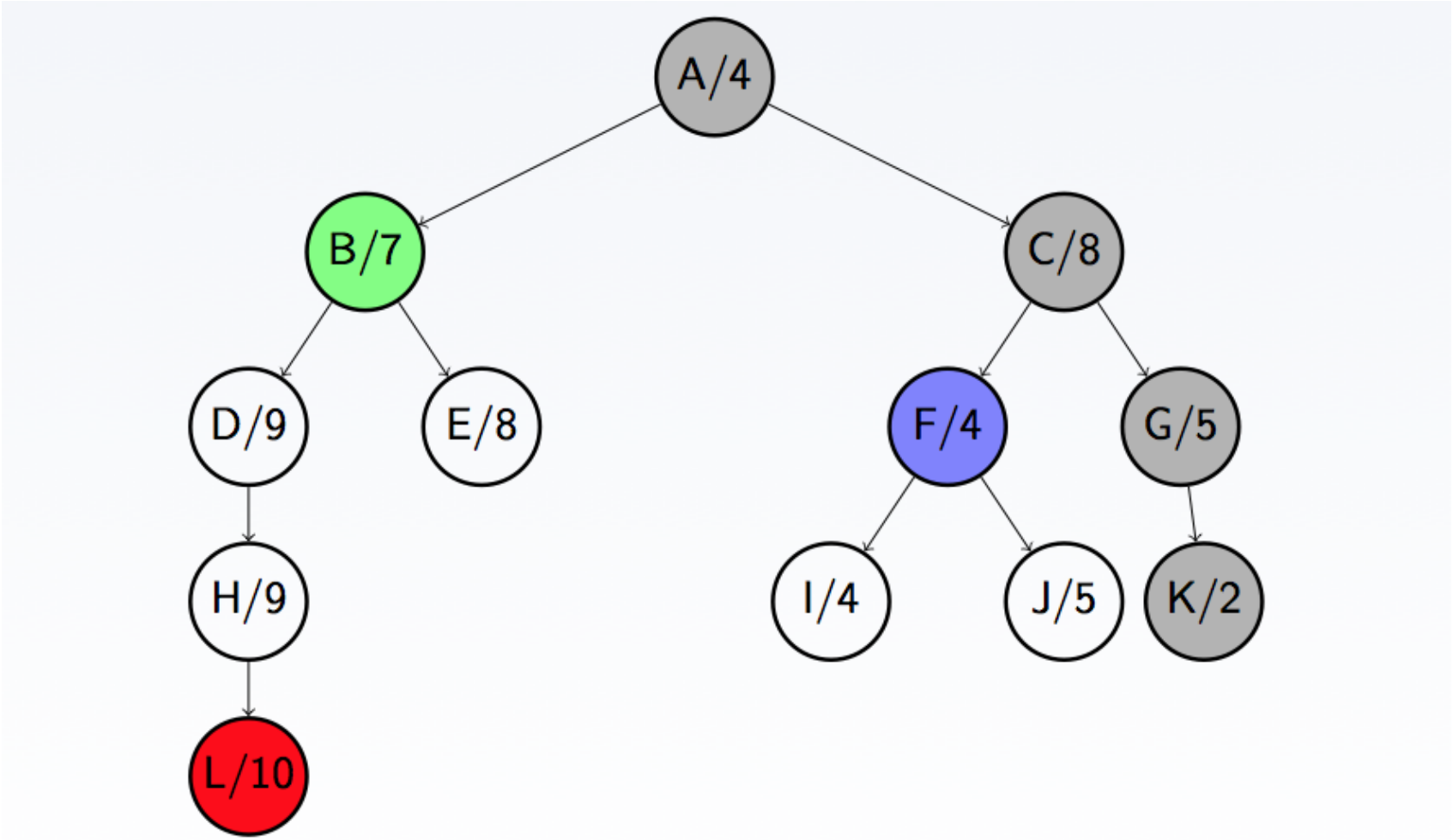
$$L = [K] \mathbin{++} [F,B] = [K,F,B]$$

Beispiel: Bergsteigen



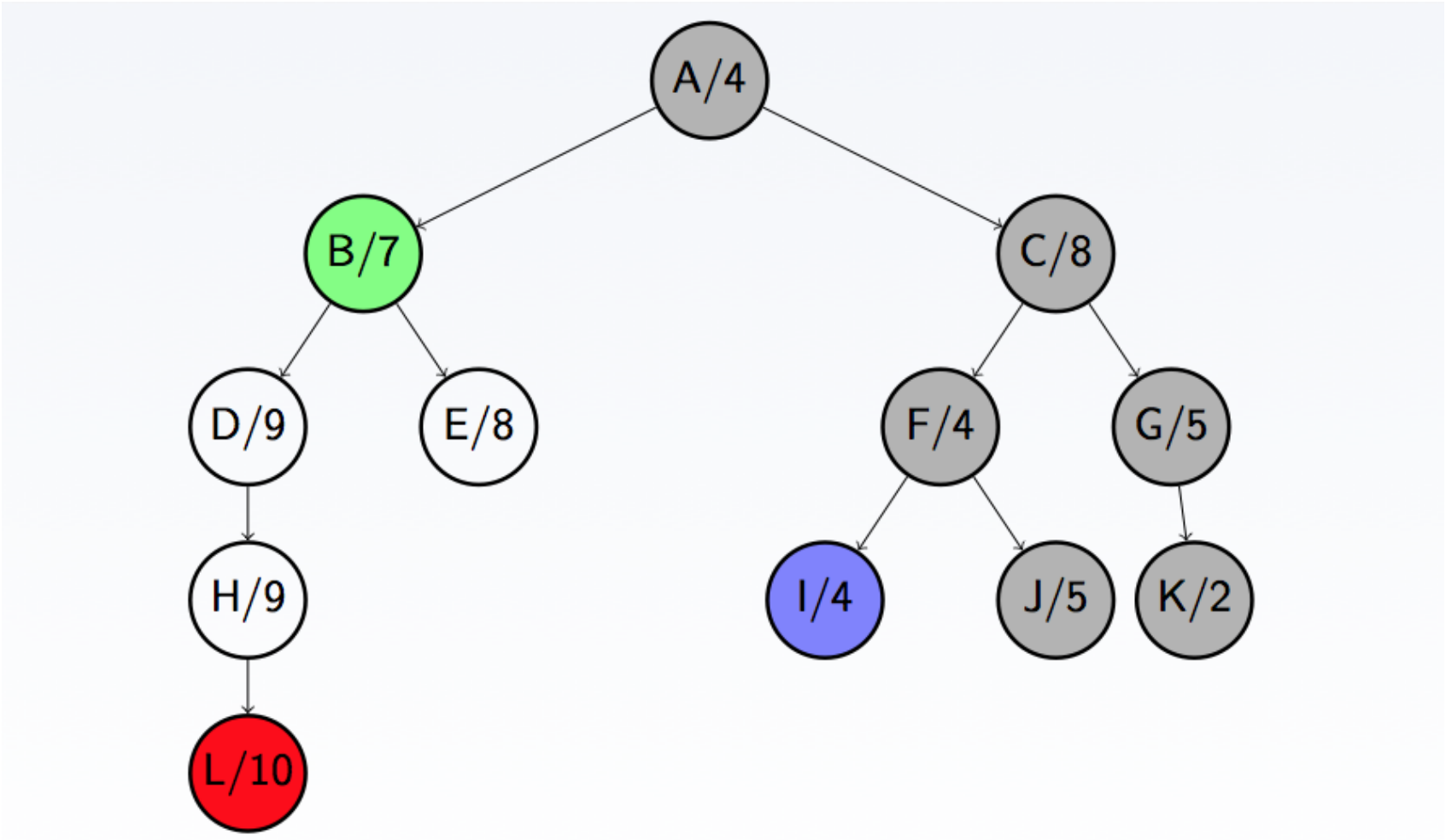
$$L = [J,I] \mathbin{++} [B] = [J,I,B]$$

Beispiel: Bergsteigen



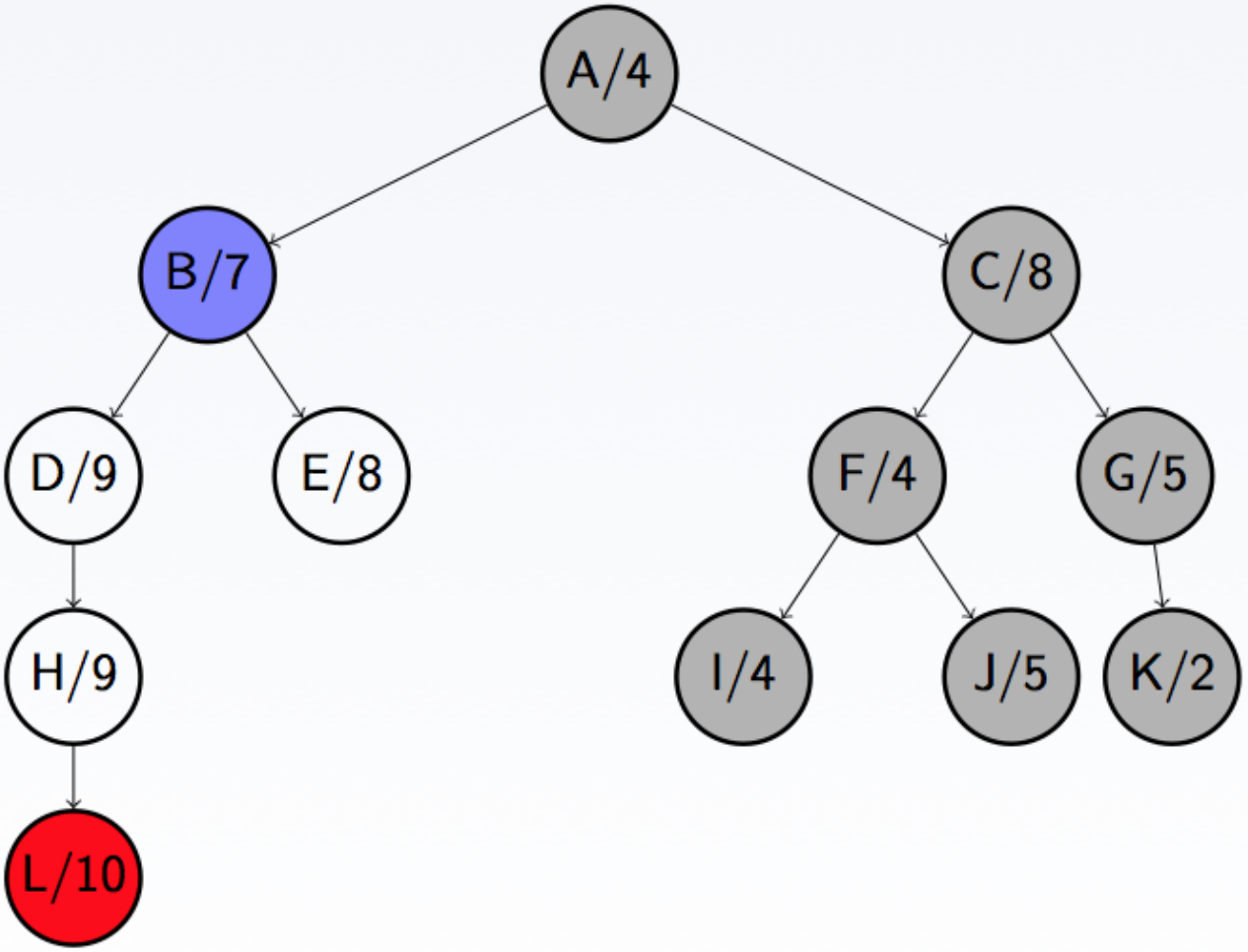
$$L = [] \text{ ++ } [F,B] = [F,B]$$

Beispiel: Bergsteigen



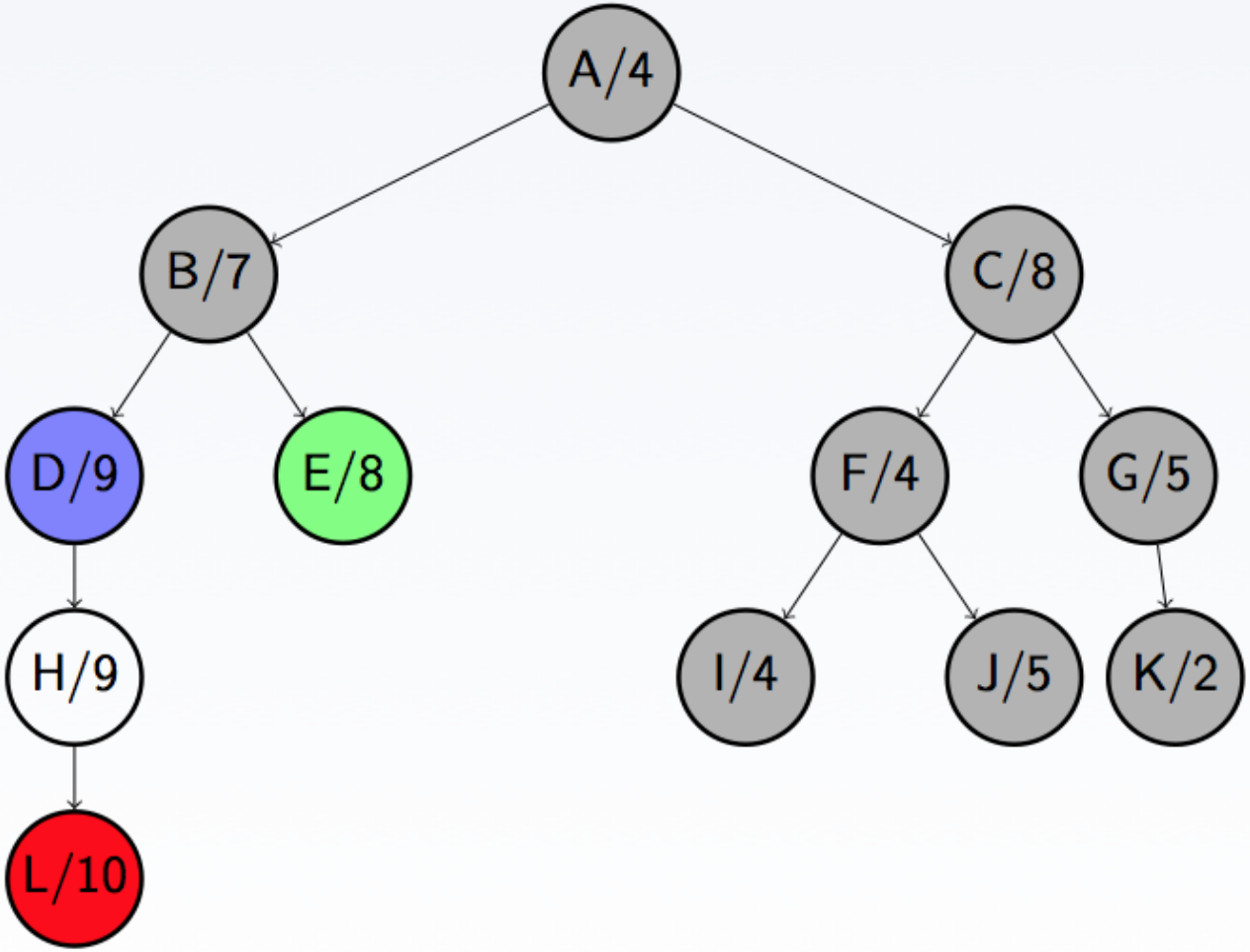
$$L = [] \text{ ++ } [I,B] = [I,B]$$

Beispiel: Bergsteigen



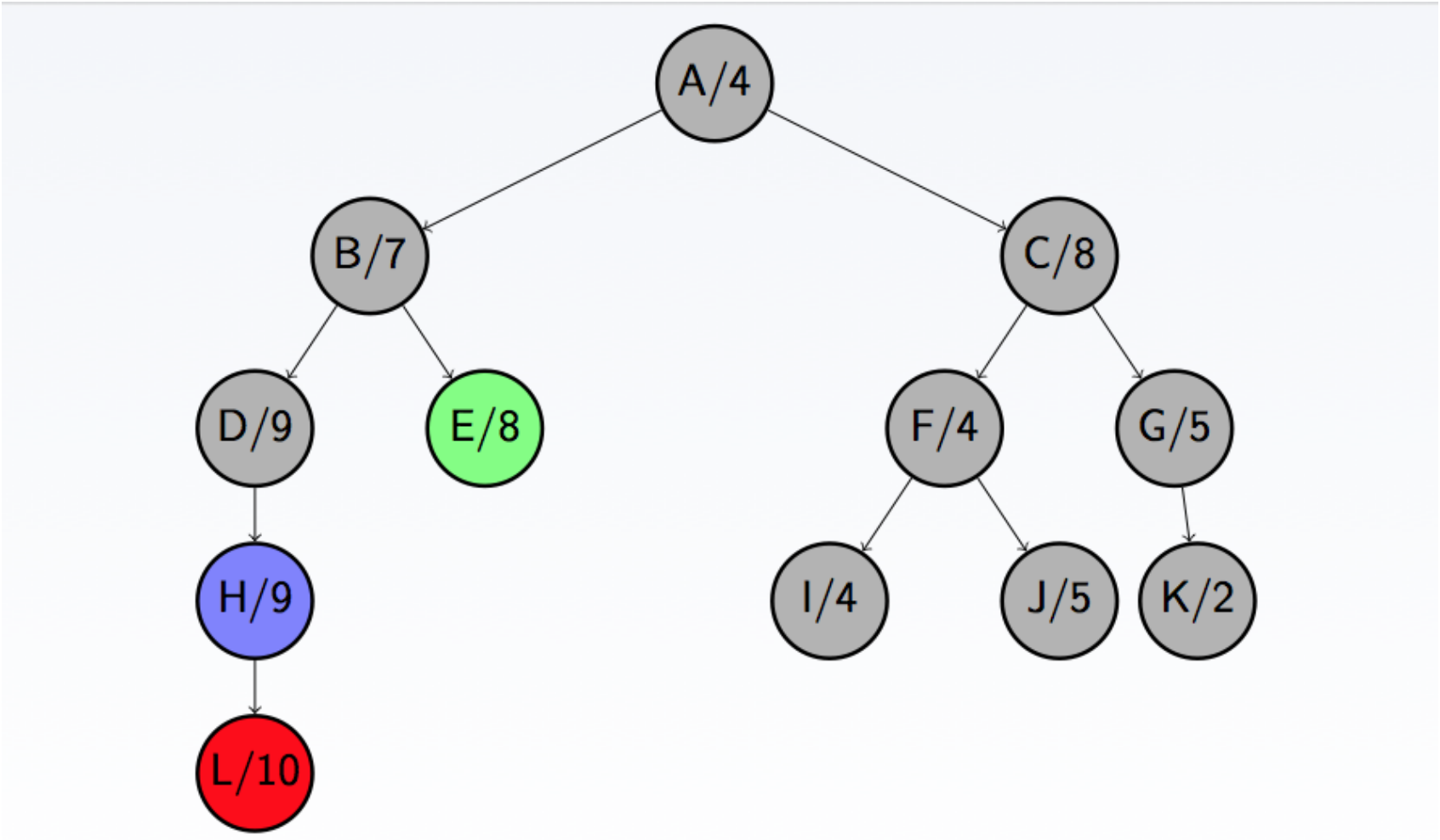
$$L = [] \text{ ++ } [B] = [B]$$

Beispiel: Bergsteigen



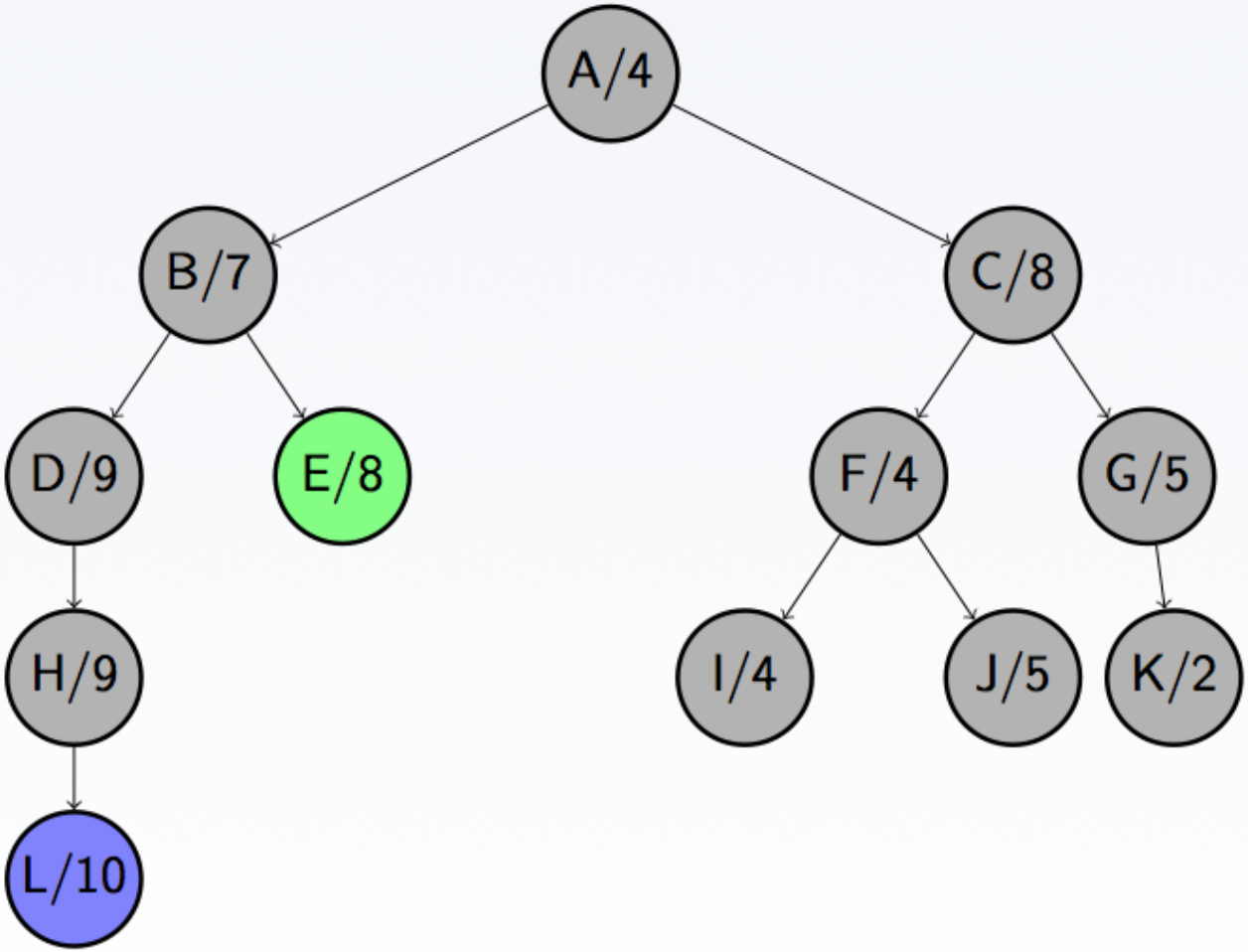
$$L = [D,E] \mathbin{++} [] = [D,E]$$

Beispiel: Bergsteigen



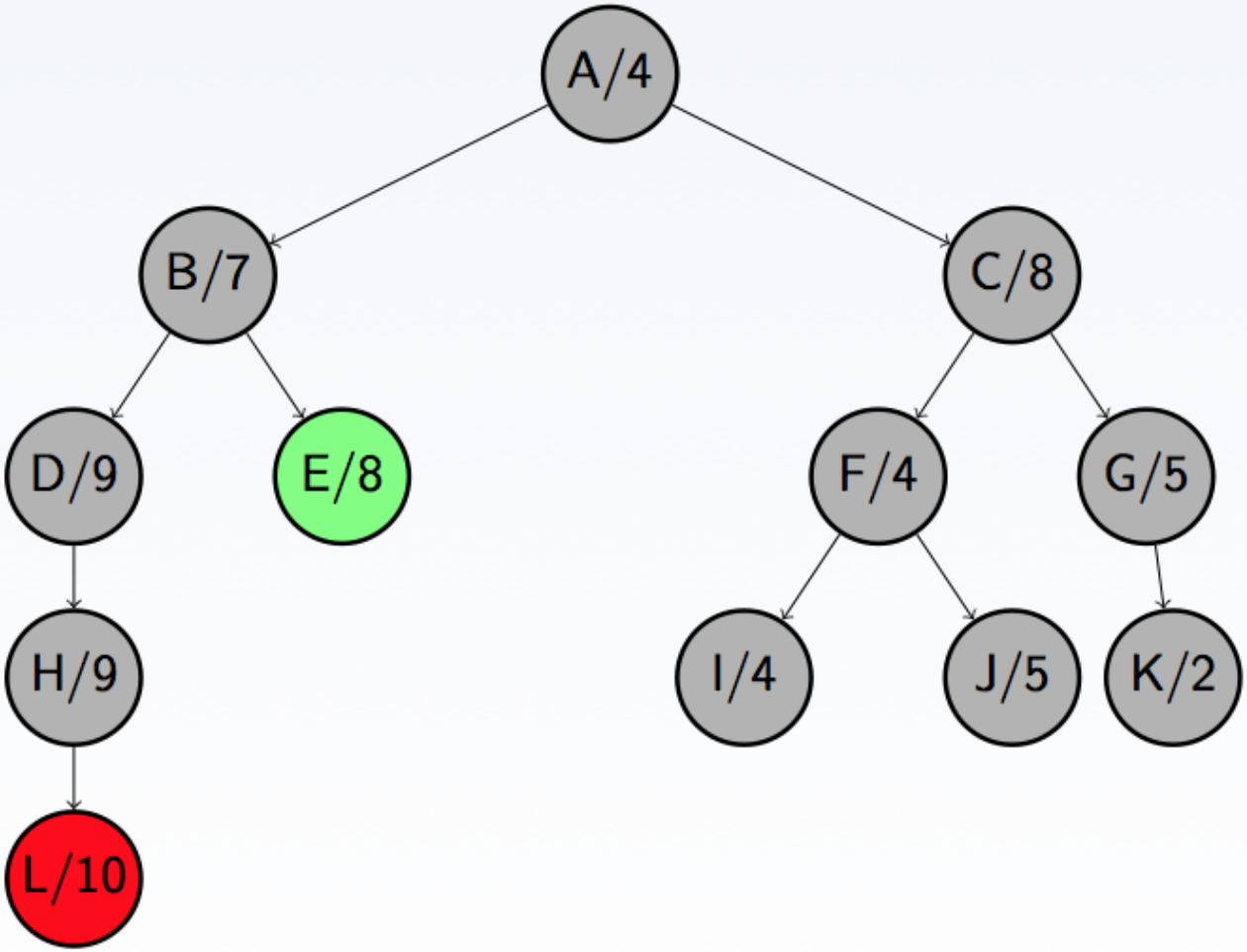
$$L = [H] \mathbin{++} [E] = [H,E]$$

Beispiel: Bergsteigen



$$L = [L] \mathbin{++} [E] = [L,E]$$

Beispiel: Bergsteigen



Zielknoten L gefunden

Eigenschaften der Bergsteigerprozedur

- Entspricht einer gesteuerten Tiefensuche mit Sharing daher nicht-vollständig
- Platzbedarf ist durch die Speicherung der besuchten Knoten exponentiell in der Tiefe.
- Varianten
  - Optimierung einer Funktion ohne Zieltest:
  - Bergsteige ohne Stack, stets zum nächst höheren Knoten
  - Wenn nur noch Abstiege möglich sind, stoppe und gebe aktuellen Knoten aus
- Findet lokales Maximum, aber nicht notwendigerweise globales

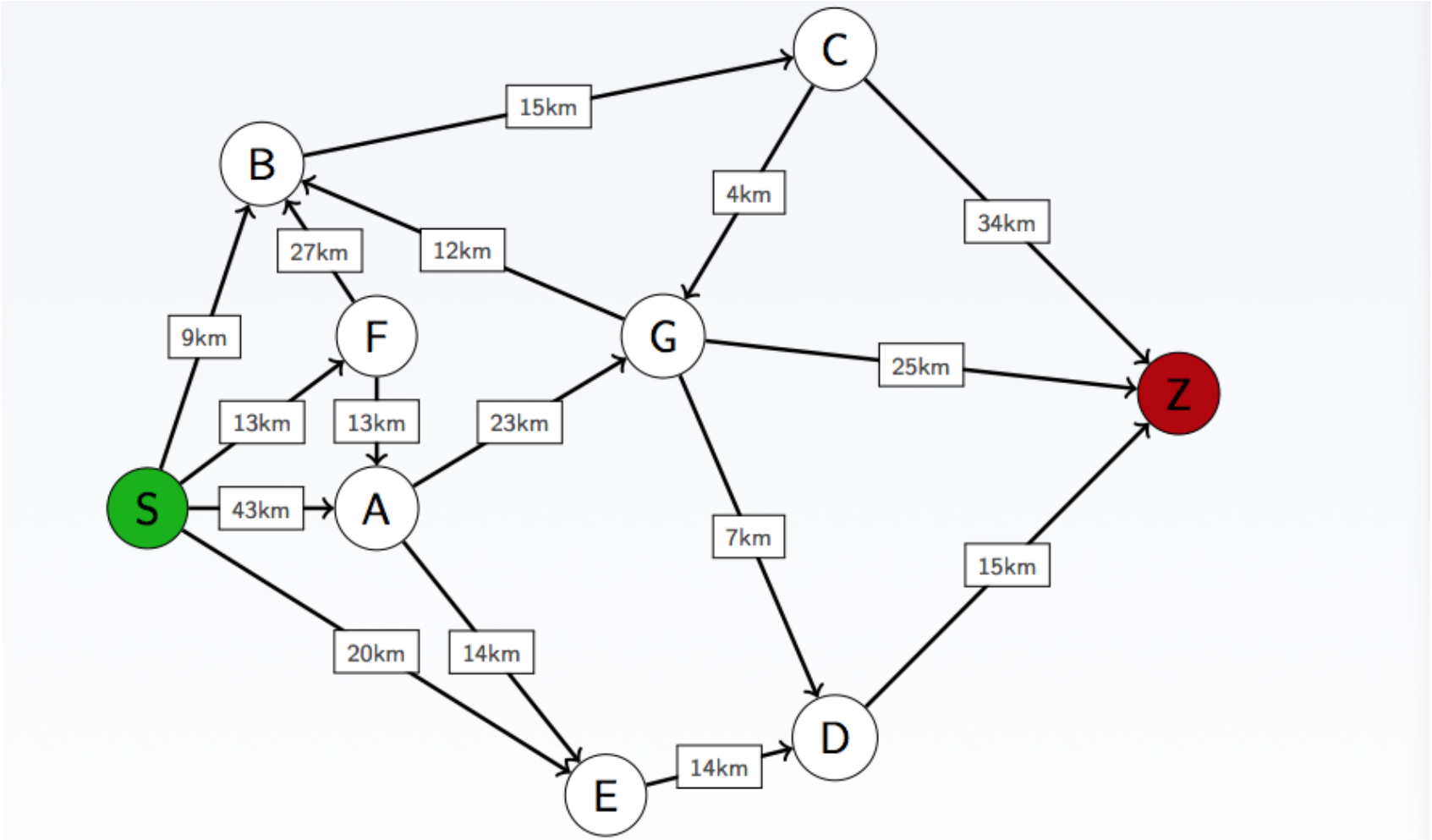
Best-First-Suche und Climbing

- Ähnlich zum Hillclimbing, aber:
- Wählte stets als nächsten zu expandierenden Knoten, den mit dem besten Wert
- Änderung im Algorithmus: sortiere alle Knoten auf dem Stack

A^*-Algorithmus

- Suchproblem
  - Startknoten
  - Zieltest Z
  - Nachfolgerfunktion NF.
    Annahme: Es gibt nur eine Kante zwischen zwei Knoten.
    (Graph ist schlicht)
  - Kantenkosten $g(N_1, N_2) \in \mathbb{R}$.
  - Heuristik h schätzt Abstand zum Ziel
- Ziel: Finde kostenminimalen Weg vom Startknoten zu einem Zielknoten

Beispiel: Routensuche



Heuristik z.B. Luftliniendistanz

A^*-Algorithmus

- Kombination positiver Fakten der Suche mit einheitlichen Kosten
  - Optimalität und Vollständigkeit
  - Sortierte Schlangen
- Greedy Suchverfahren
  - Geschwindigkeit
  - Sortierung anhand Evaluierung
- Evaluierungsfunktion $f(n)$
  - Kostenabschätzung für den Pfad durch Knoten n: $f(n) = g(n) + h(n)$
  - $g(n)$ - Kostenfunktion vom initialen Zustand bis zum Zustand n
  - $h(n)$ - Heuristische Kostenfunktion vom Zustand n bis zum Zielzustand
- Minimisierung der gesamt Kosten für ein Pfad

Beispiel: Das Rucksack-Problem

- Volumen W, n Objekte $(o_1, o_2, \ldots, o_n)$, jedes Objekt bringt ein Gewinn p_i, $i = 1, 2, \ldots, n$.
-

	o_1	o_2	o_3	o_4
p_i	10	18	32	14
w_i	1	2	4	3

- Lösung: für $W = 5 \rightarrow o_1$ und o_3
- $g(n) = \sum p_i$ für ausgewählte Objekte o_i
- $h(n) = \sum p_j$ für nicht ausgewählte Objekte und $\sum w_j \leq W - \sum w_i$
- Jeder einzelne Knoten ist ein Tupel (p, w, p^*, f) mit:
  - p - Gewinn der ausgewählten Objekte (Funktion $g(n)$)
  - w - Gewicht der ausgewählten Objekte
  - p^* - maximaler Gewinn der erreicht werden kann aus dem jetzigen Zustand unter Betrachtung des freien Platzes im Rucksack (Funktion $h(n)$)

A^*-Algorithmus

Algorithmus A^*-**Algorithmus**

Datenstrukturen:

- Menge `Open` von Knoten
- Menge `Closed` von Knoten
- Wert $g(N)$ für jeden Knoten (markiert mit Pfad vom Start zu N)
- Heuristik h
- Zieltest Z
- Kantenkostenfunktion c

Eingabe:

- `Open` $:= \{S\}$, wenn S der Startknoten ist
- $g(S) := 0$, ansonsten ist g nicht initialisiert
- `Closed` $:= \emptyset$

A^*-Algorithmus

Algorithmus:
repeat
  Wähle N aus `Open` mit minimalem $f(N) = g(N) + h(N)$
  if $Z(N)$ **then**
    break; // Schleife beenden
  else
    Berechne Liste der Nachfolger $\mathcal{N} := NF(N)$
    Schiebe Knoten N von `Open` nach `Closed`
    for $N' \in \mathcal{N}$ **do**
      if $N' \in$ `Open` $\cup$ `Closed` und $g(N) + c(N, N') > g(N')$ **then**
        skip // Knoten nicht verändern
      else
        g(N') := g(N) + c(N,N'); // neuer Minimalwert für g(N')
        Füge N' in `Open` ein und (falls vorhanden) lösche N' aus `Closed`;
      end-if
    end-for
  end-if
until `Open` $= \emptyset$
if `Open` $= \emptyset$ **then** Fehler, kein Zielknoten gefunden
else N ist der Zielknoten mit $g(N)$ als minimalen Kosten
end-if

A^*-Algorithmus

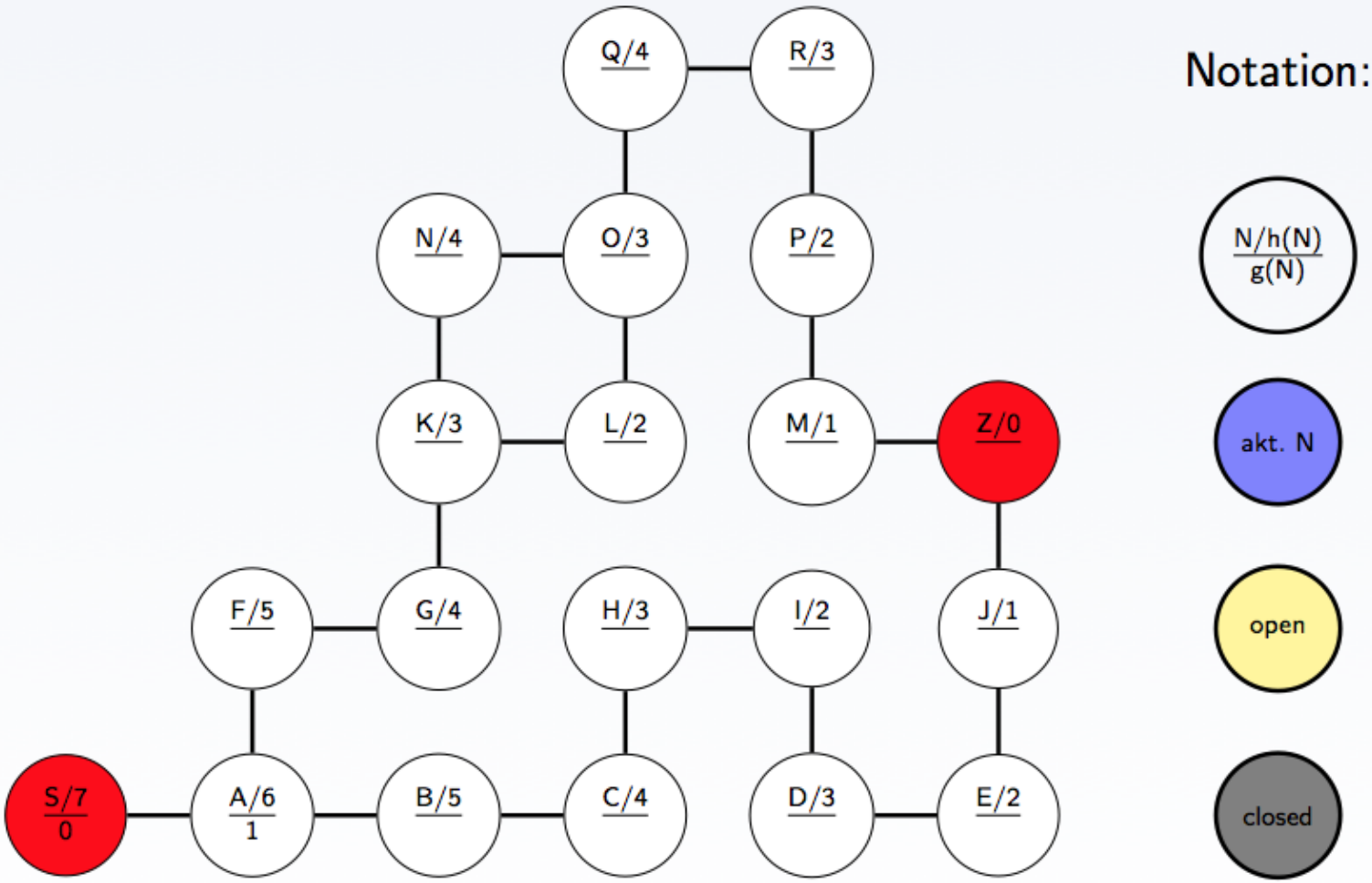
- Komplexitätsanalyse:
  - Zeitkomplexität: b Verzweigungsfaktor, d^{max} maximale Tiefe, $T(n) = 1 + b + b^2 + \cdots + b^{d^{max}} \Rightarrow O(b^{d^{max}})$
  - Speicherkomplexität: d Tiefe der Lösung, $S(n) = 1 + b + b^2 + \cdots + b^d \Rightarrow O(b^d)$
  - Vollständigkeit: Ja
  - Optimalität: Ja
- Vorteile: Expandiert die kleinste Anzahl von Knotem im Baum
- Nachteile: Verbraucht viel Speicherplatz
- Anwendungen: Planungsprobleme, Probleme mit partielle Summen (Rucksack, Münzen), Puzzles, Optimale Pfade in Graphen

A^*-Algorithmus

Versionen

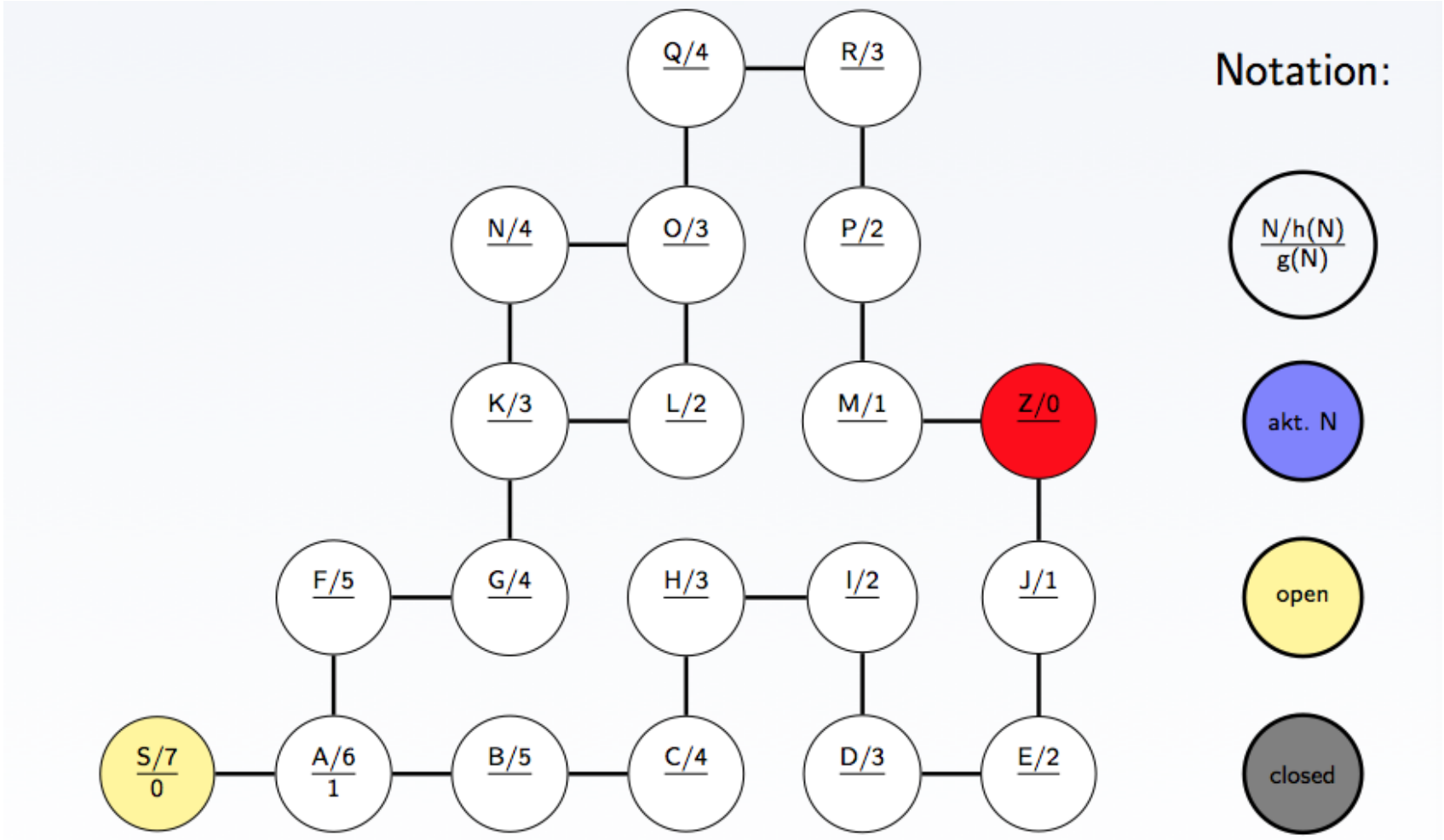
- iterative deepening $A^*(IDA^*)$
- memory-bounded $A^*(MA^*)$
- simplified memory bounded $A^*(SMA^*)$
- recursive best-first search (RBFS)
- dynamic $A^*(DA^*)$
- real time A^*
- hierarchical A^*

Beispiel



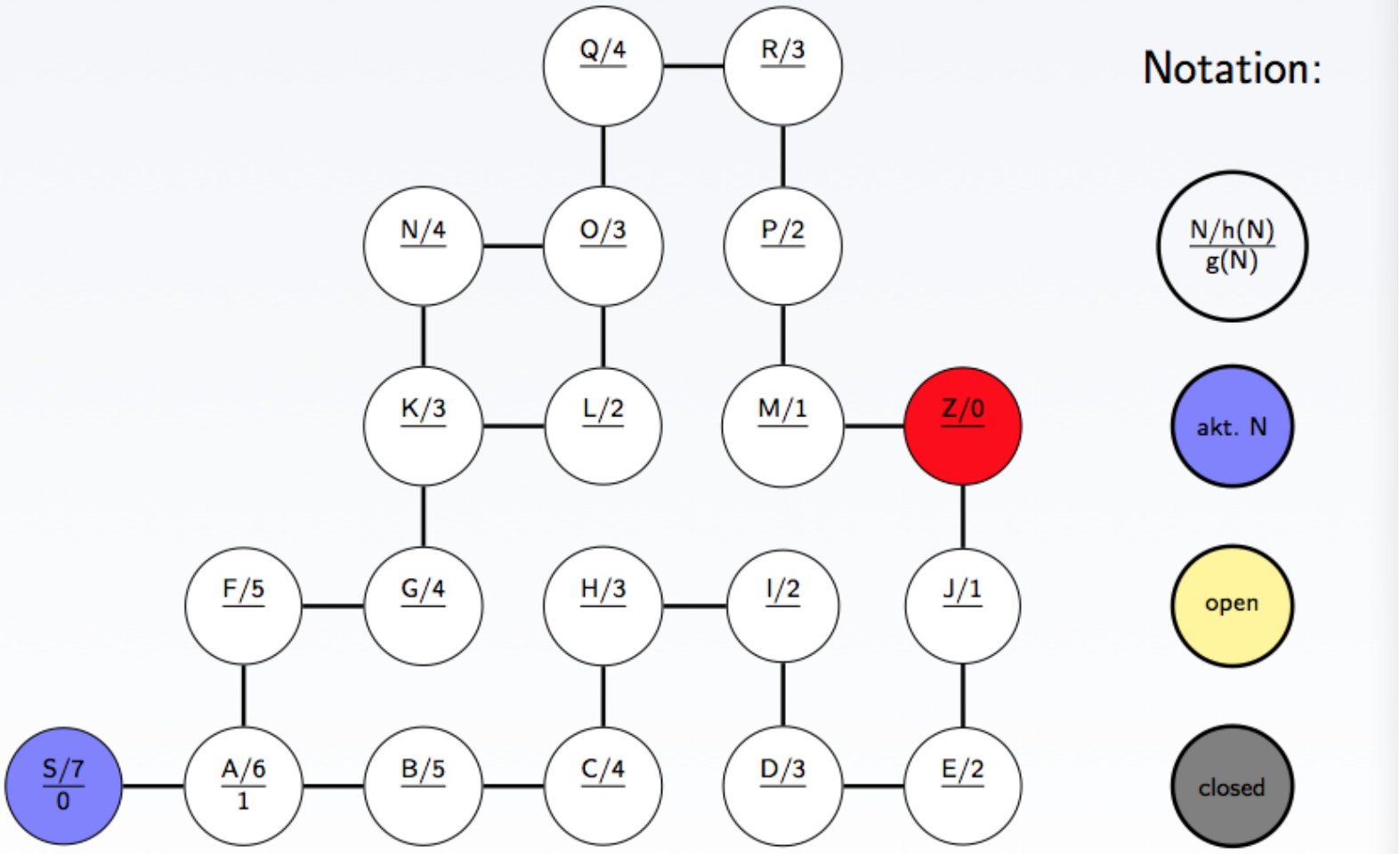
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



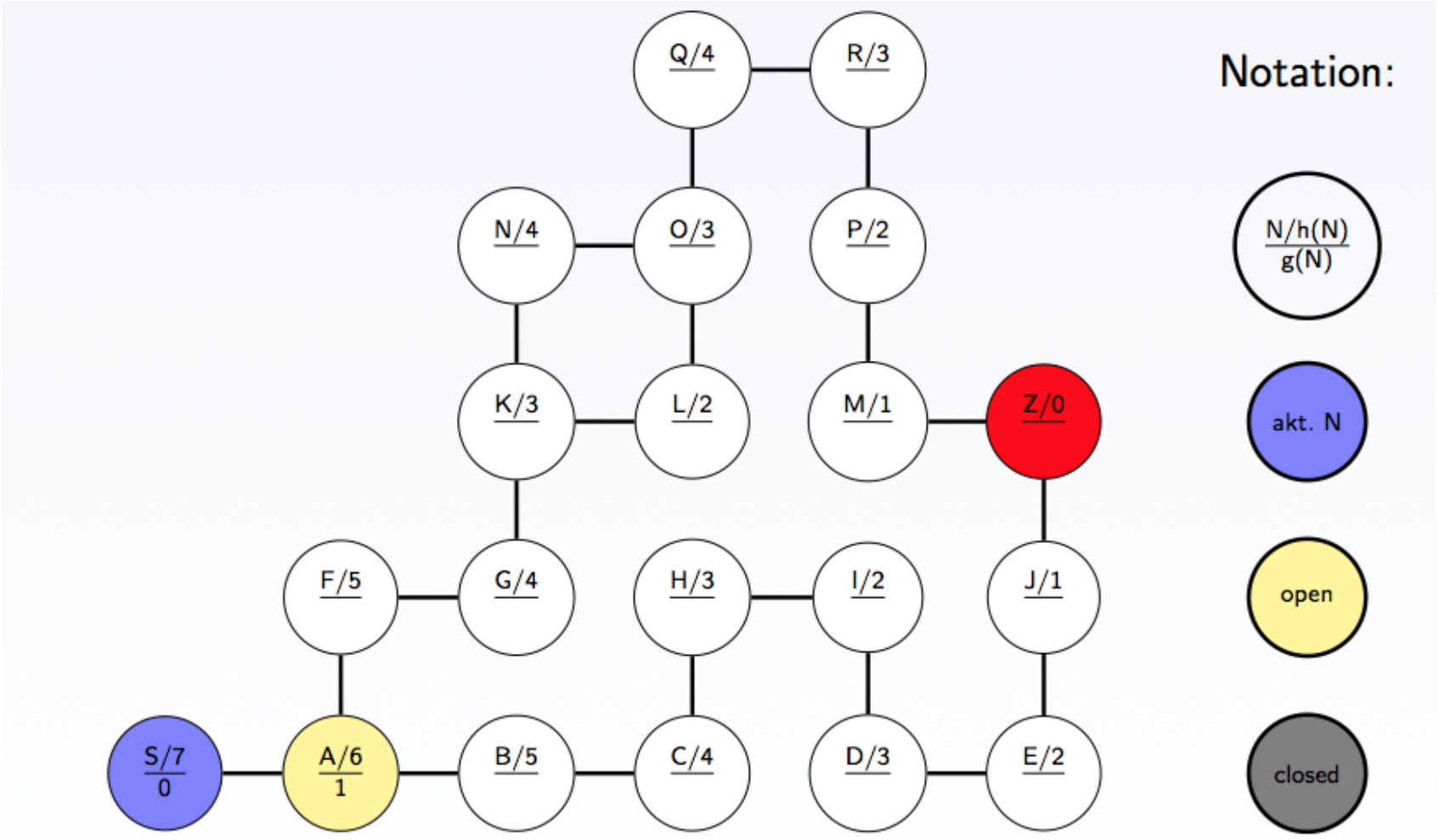
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

BEISPIEL



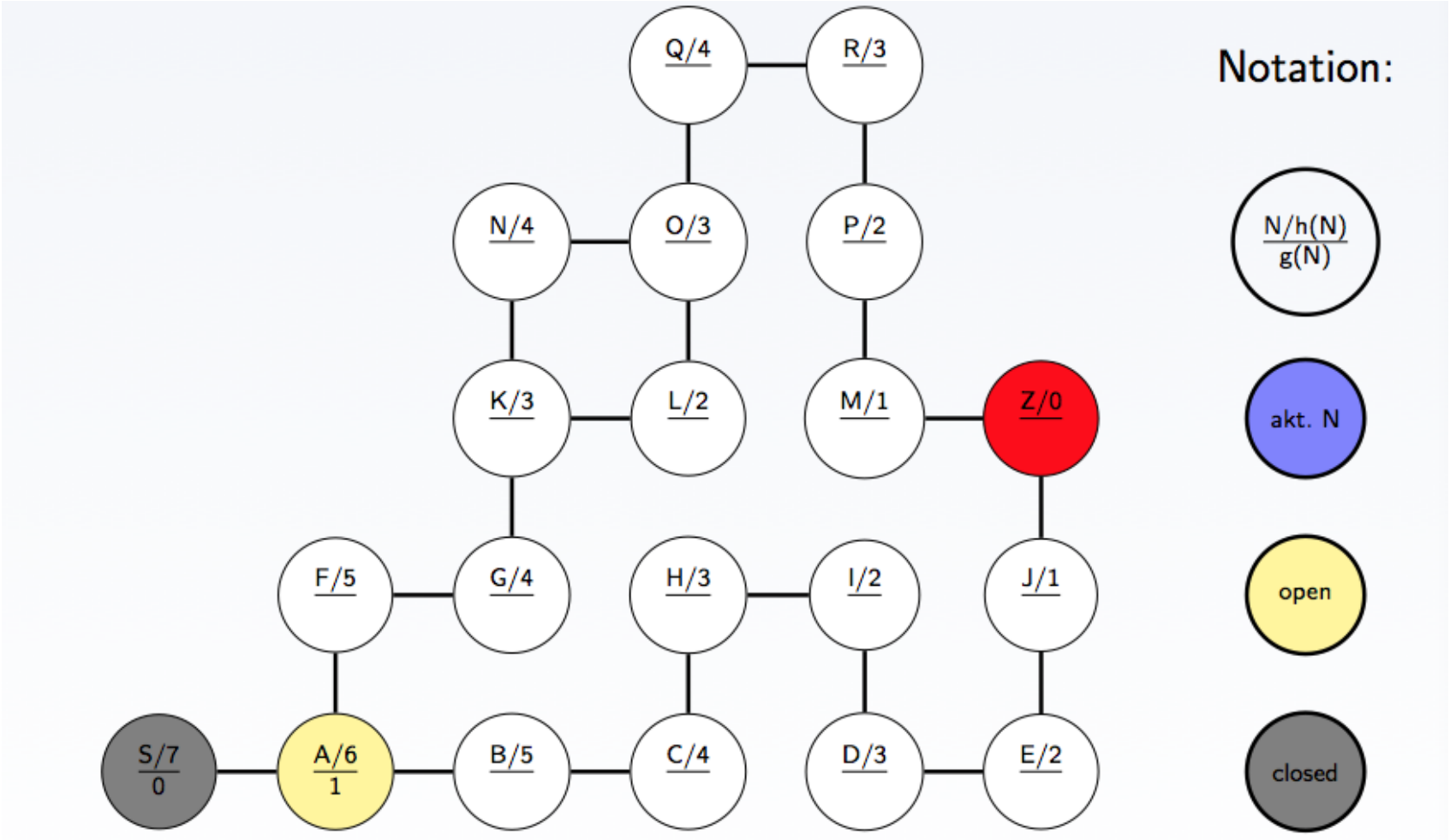
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

BEISPIEL



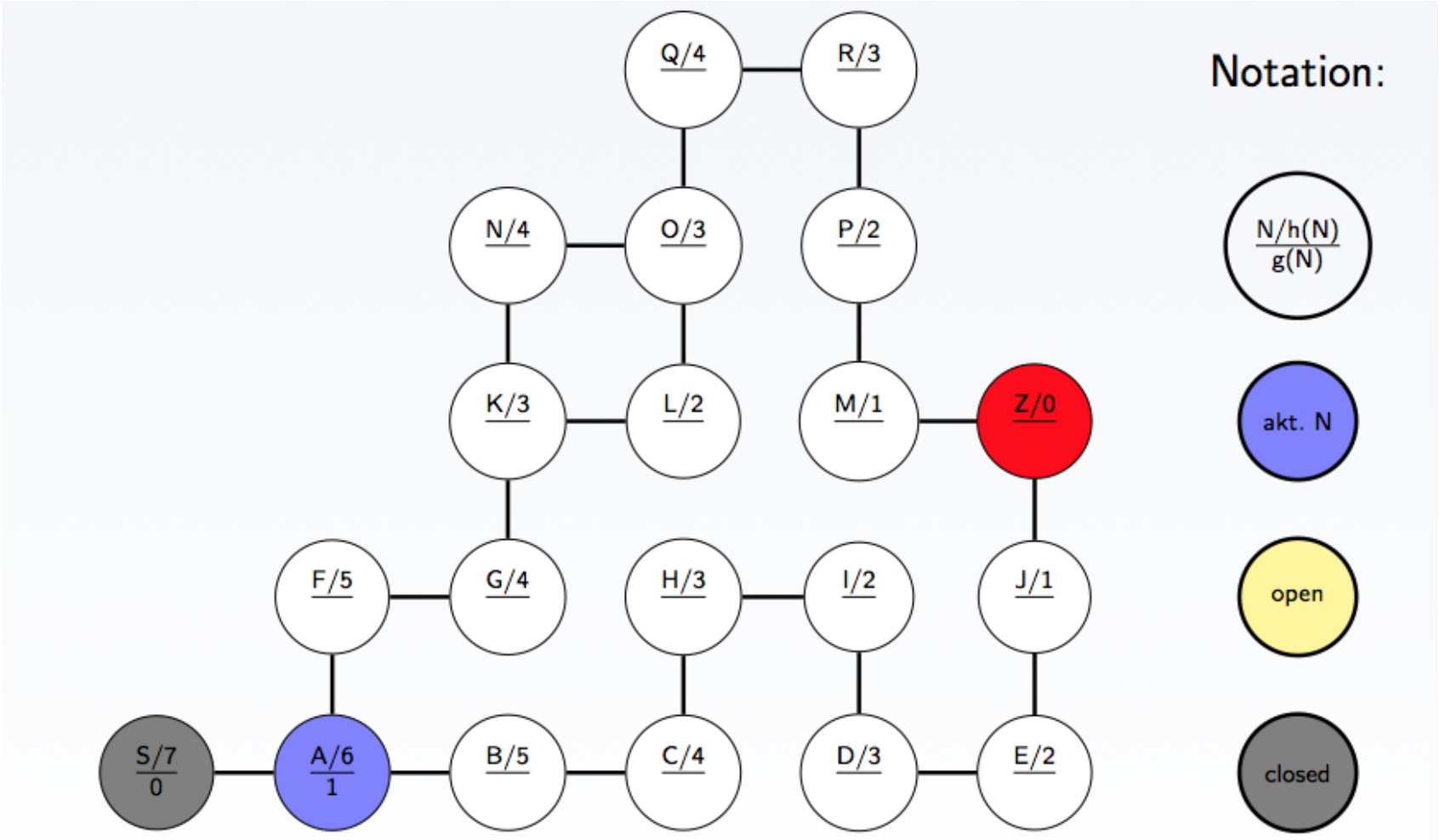
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



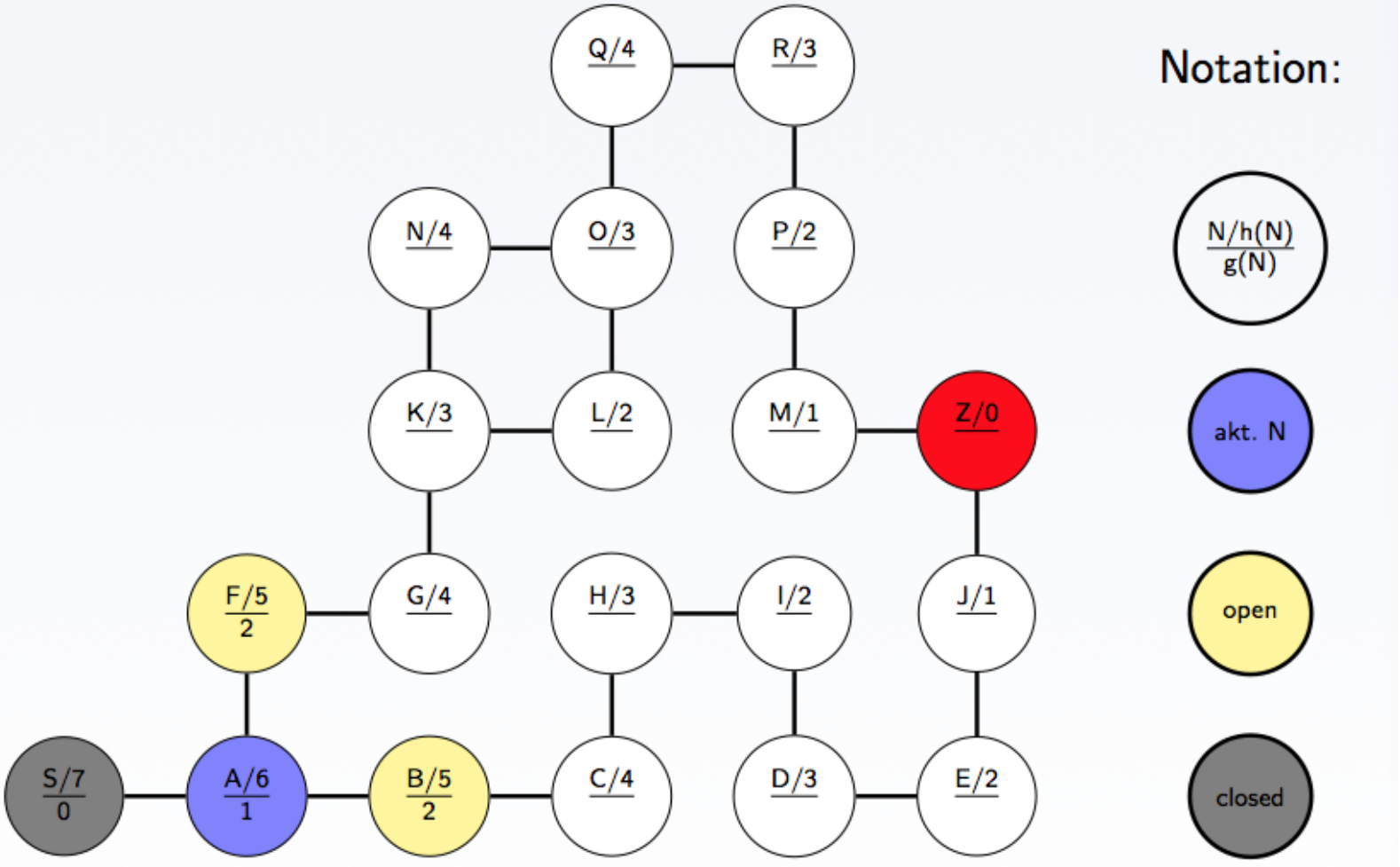
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



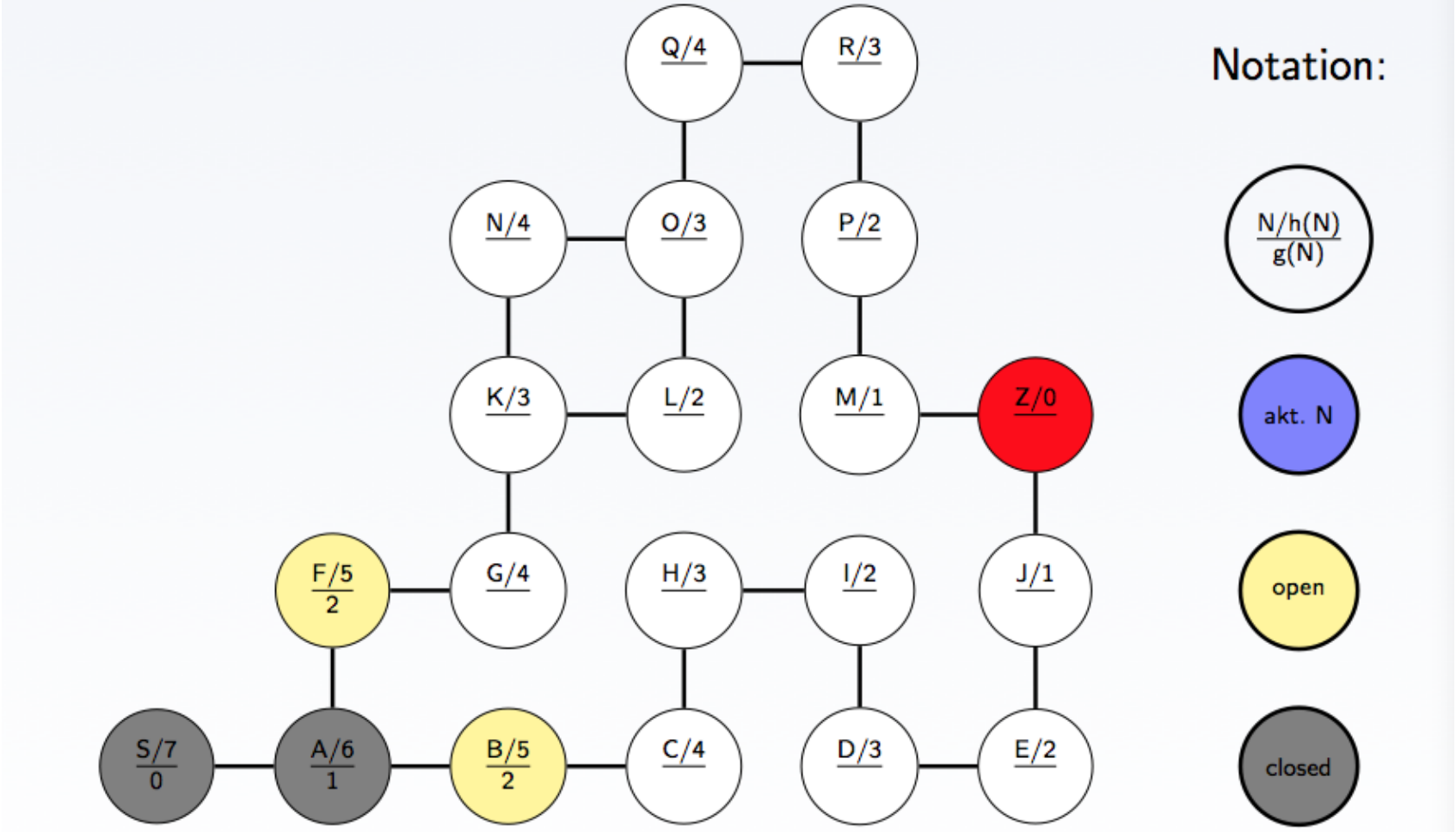
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



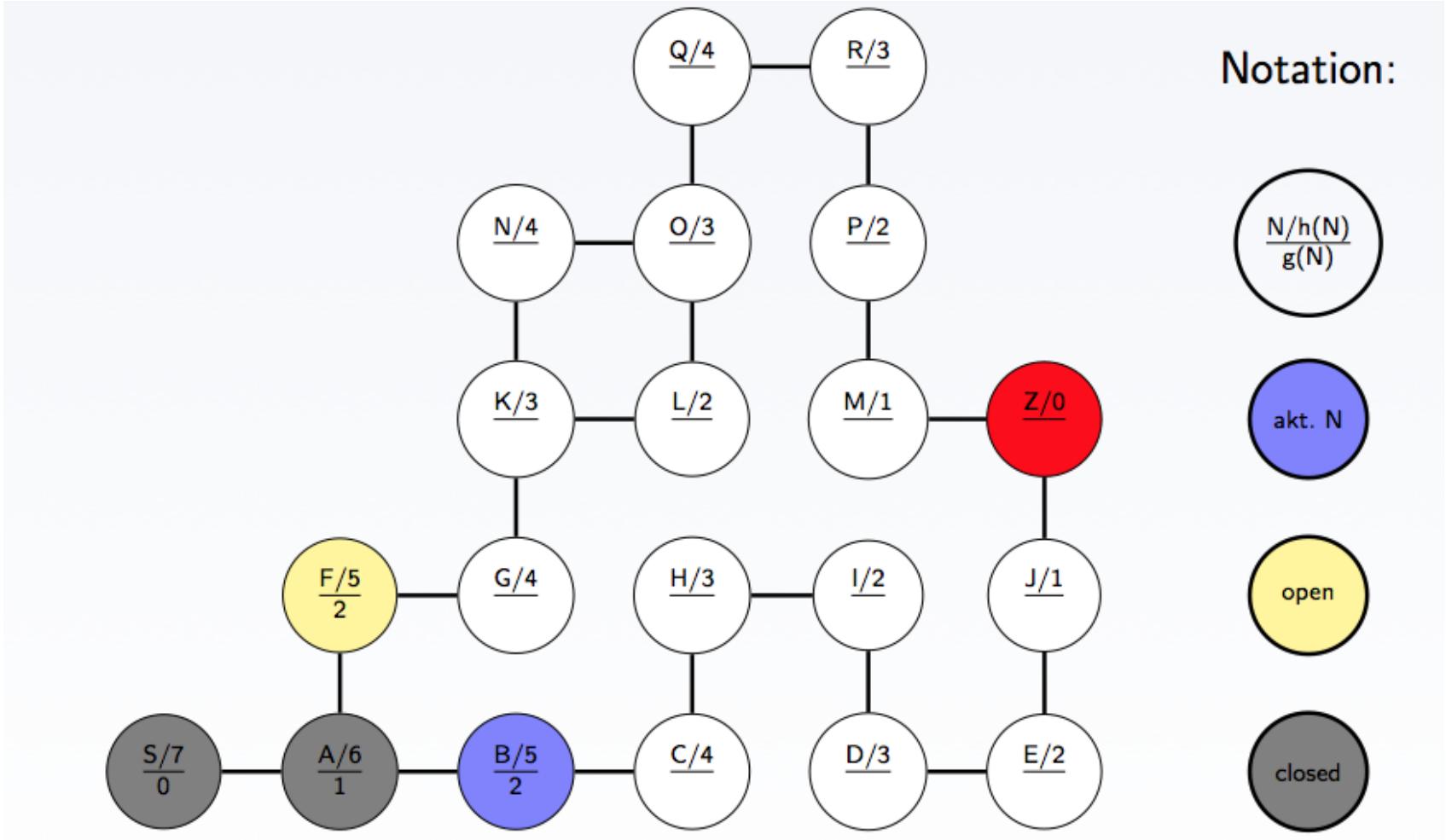
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



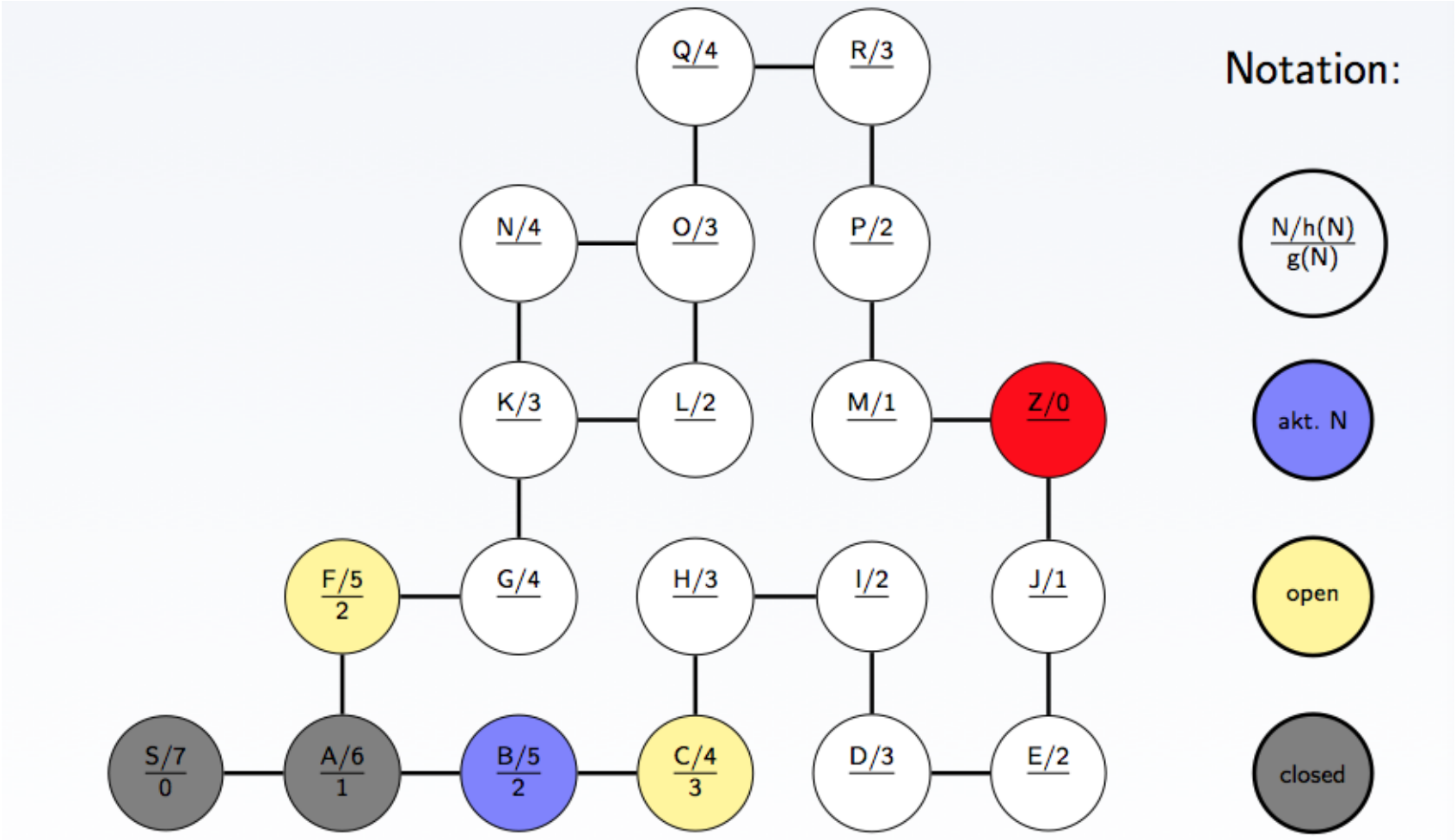
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



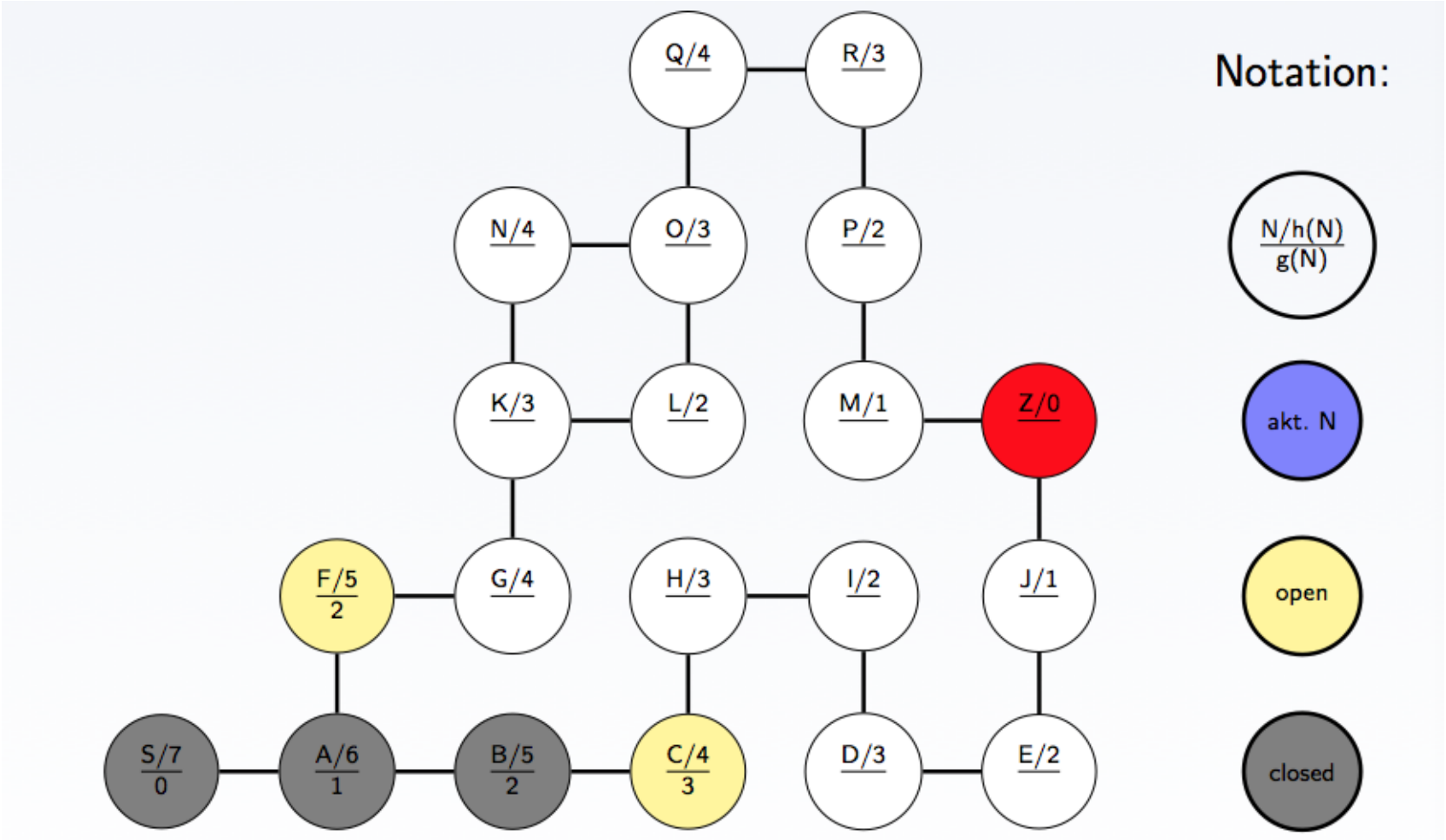
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

BEISPIEL



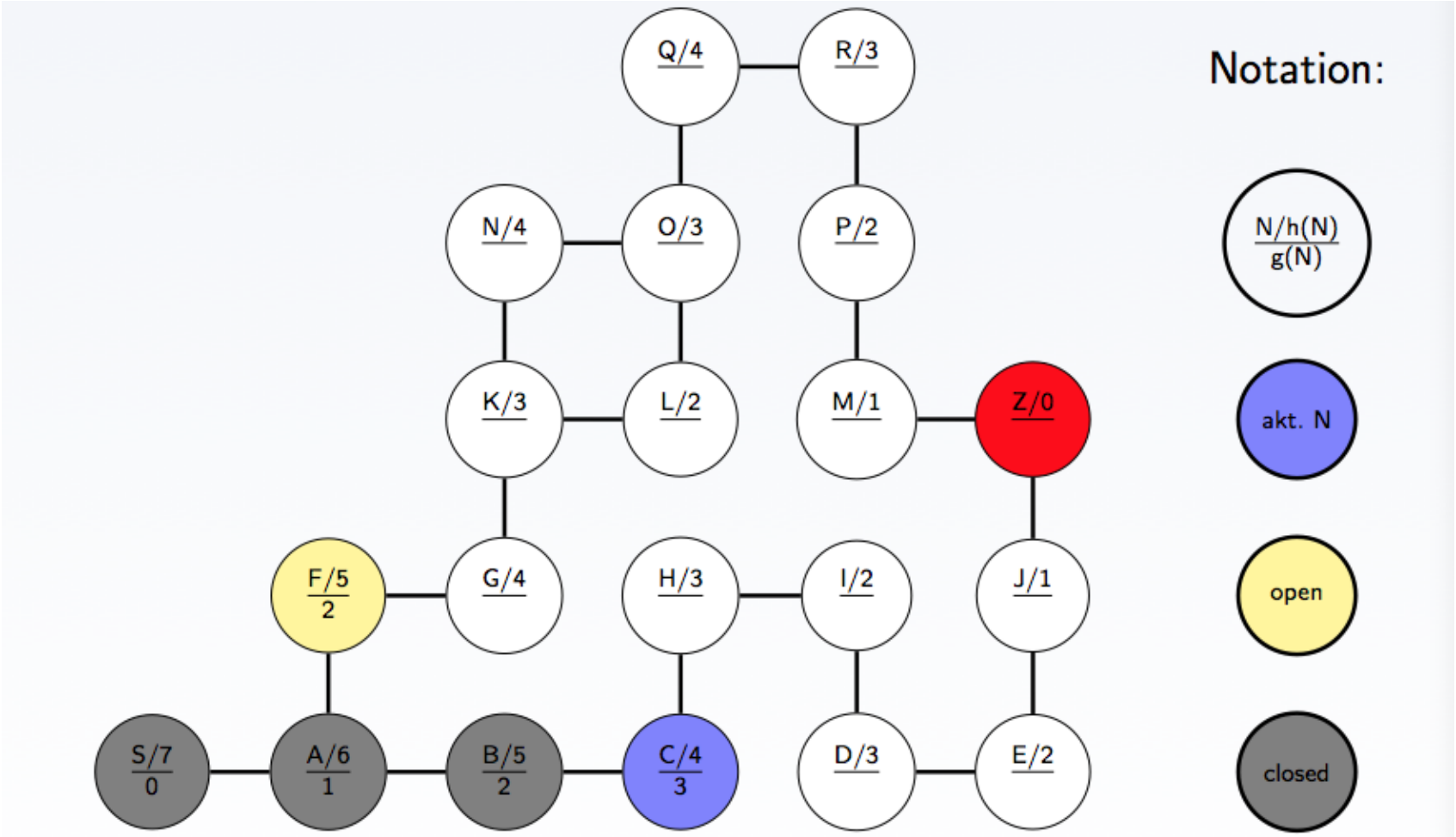
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



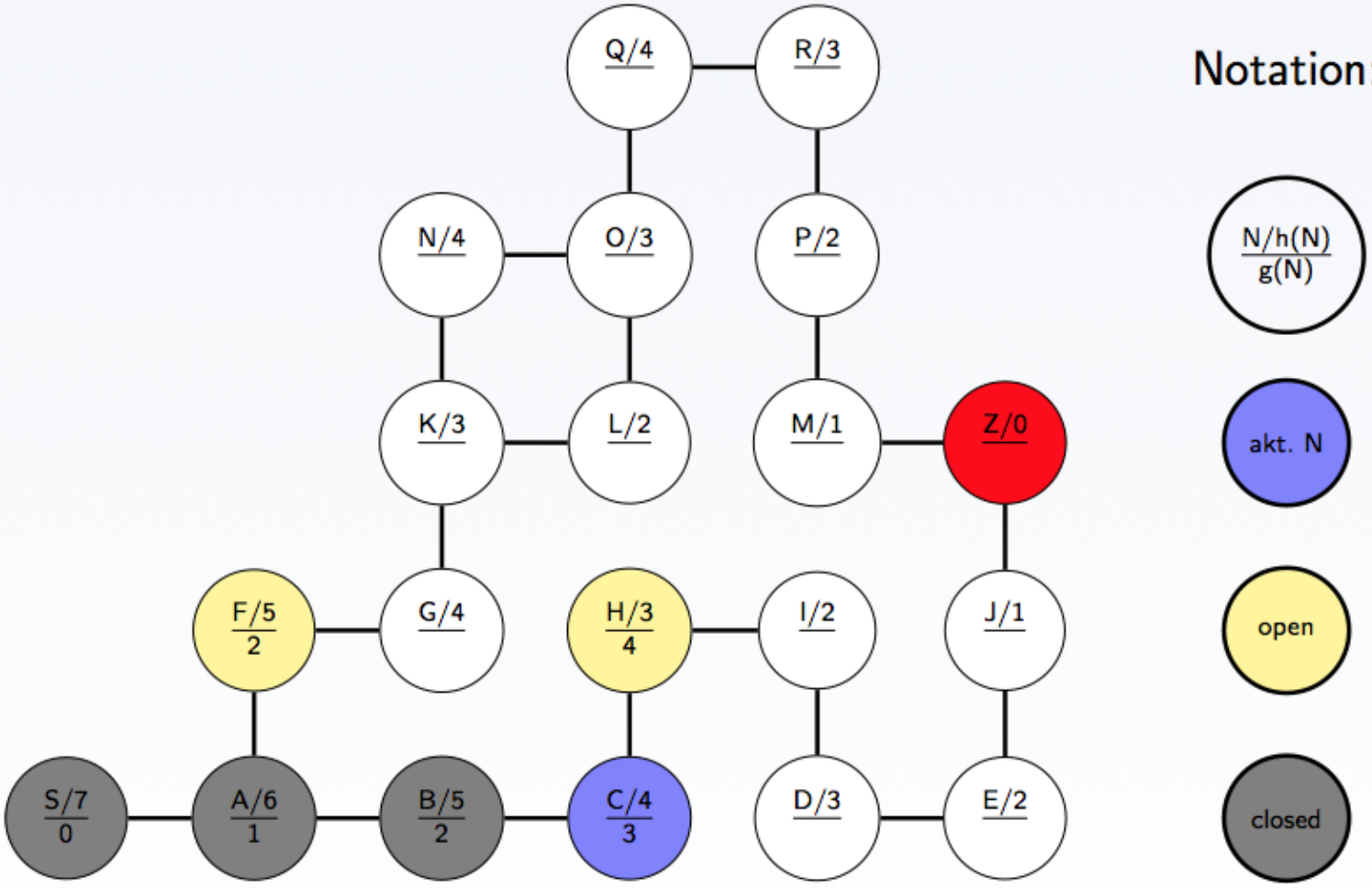
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



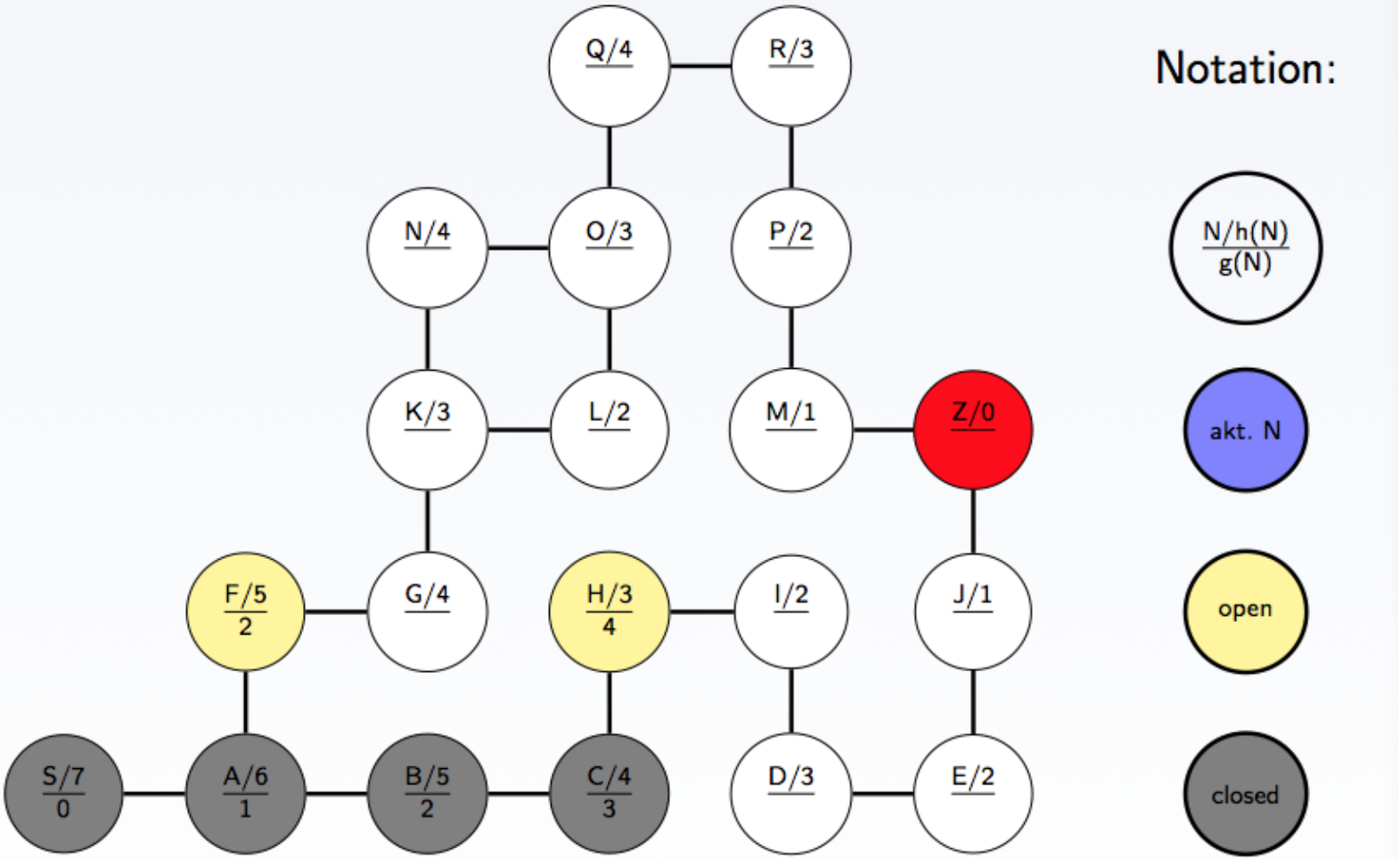
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

BEISPIEL



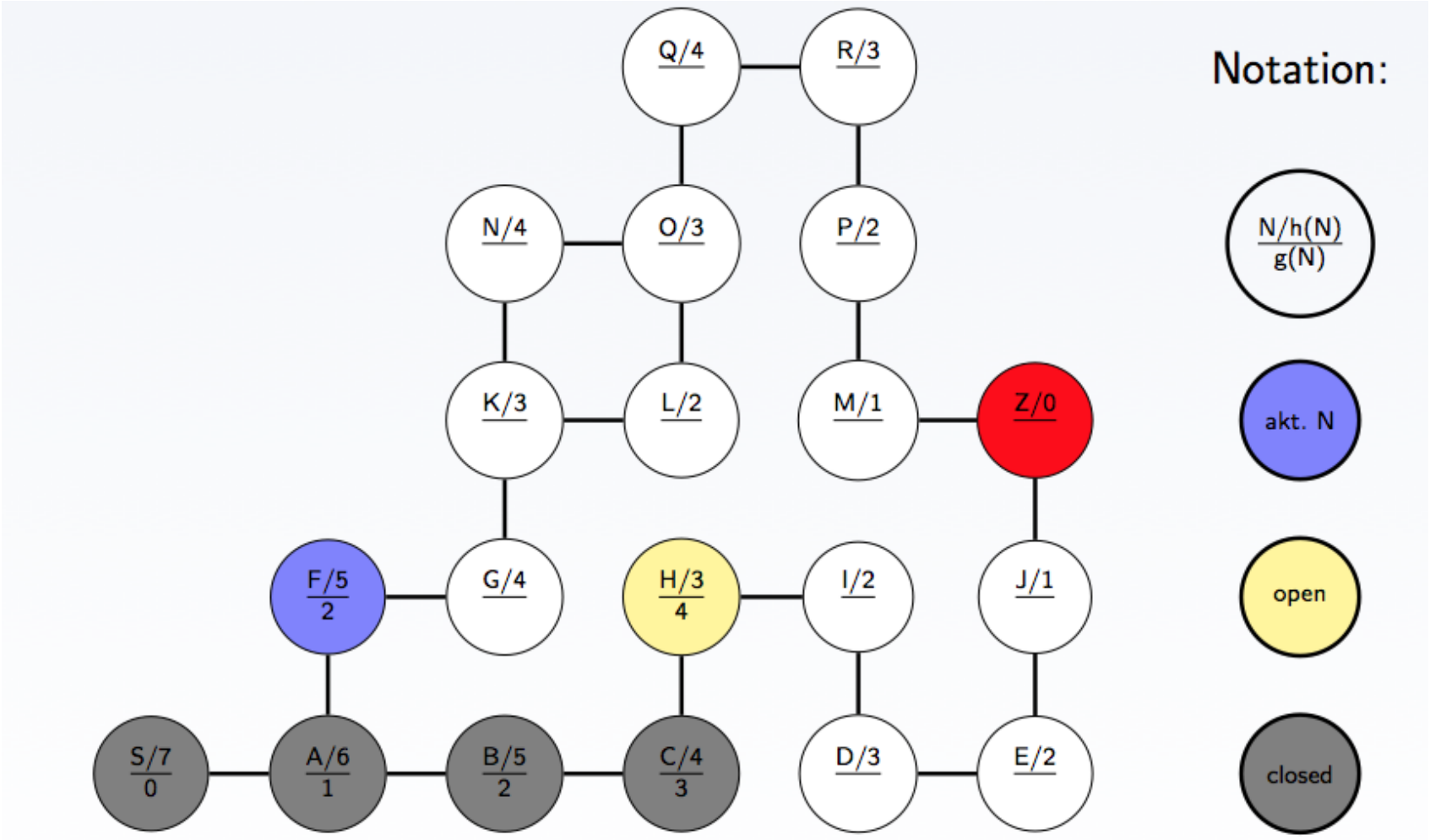
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

BEISPIEL



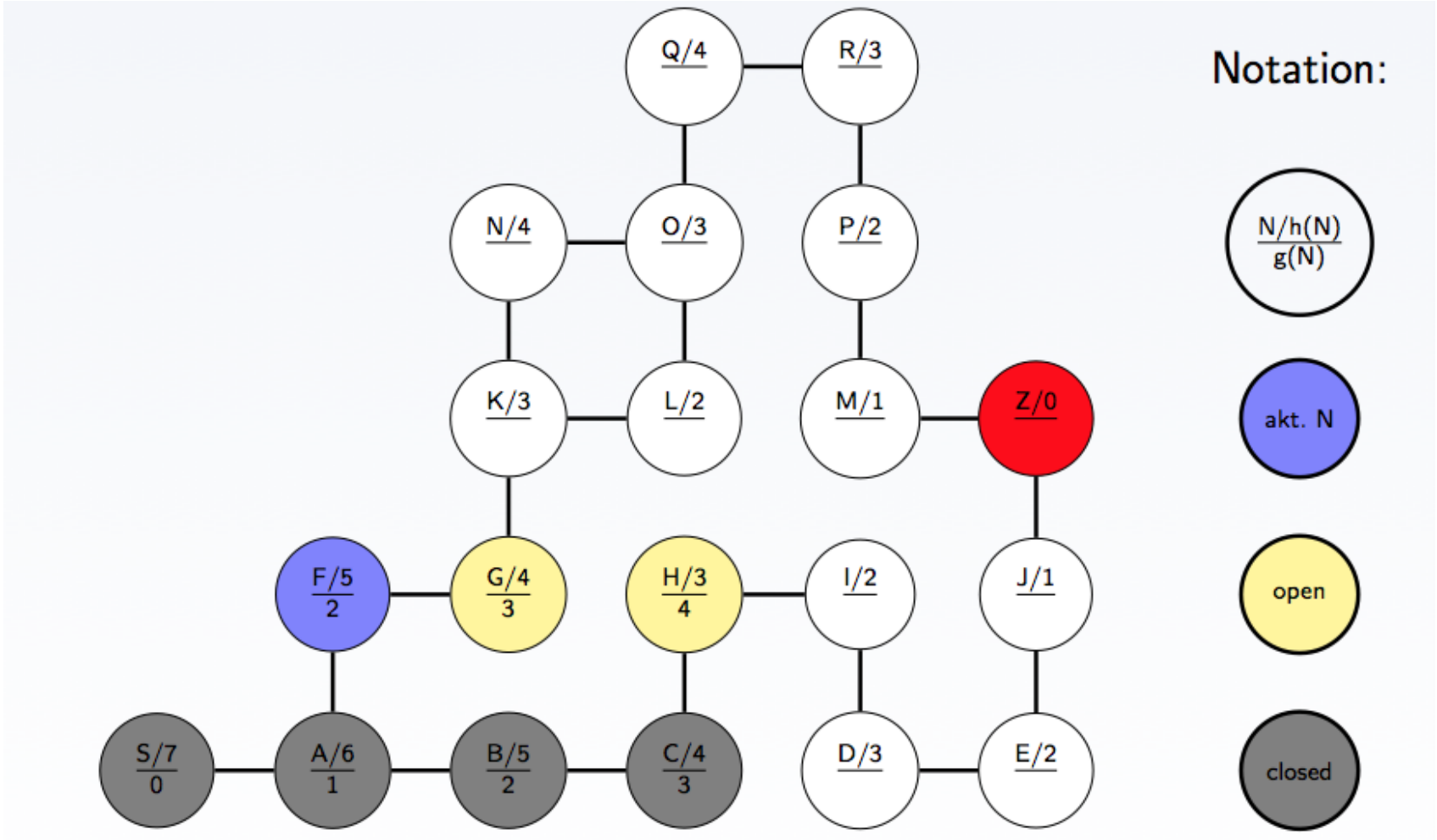
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



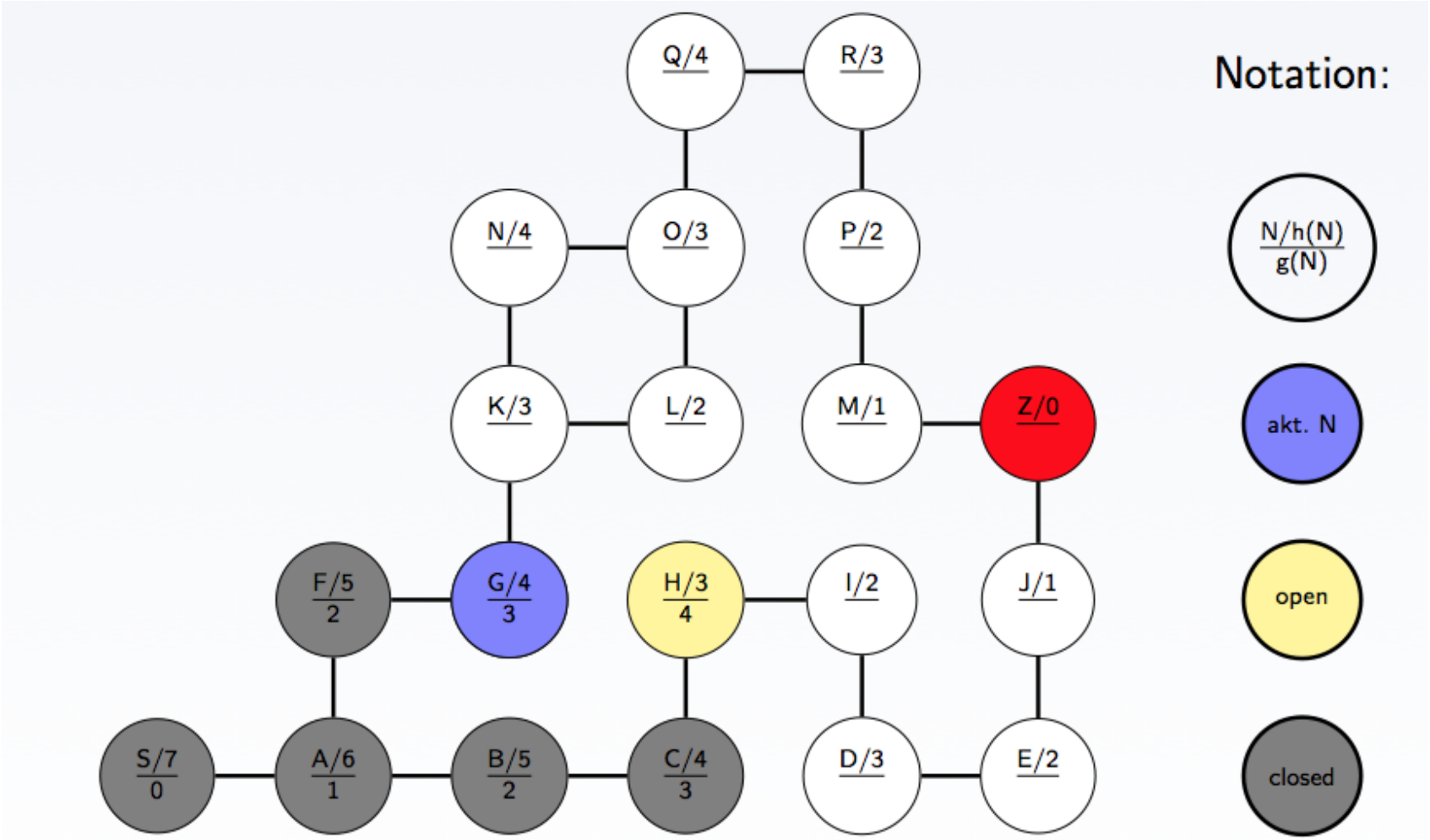
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



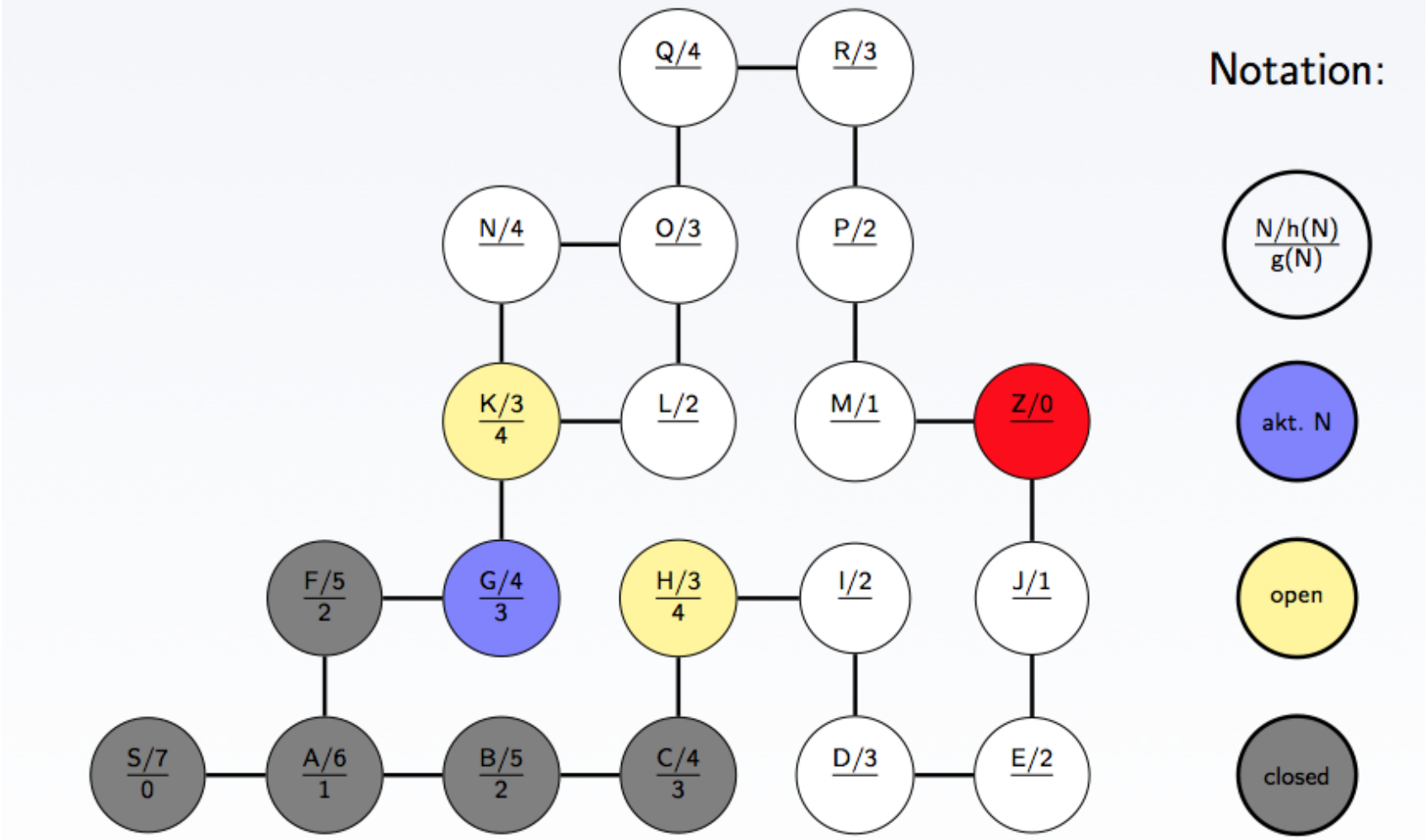
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



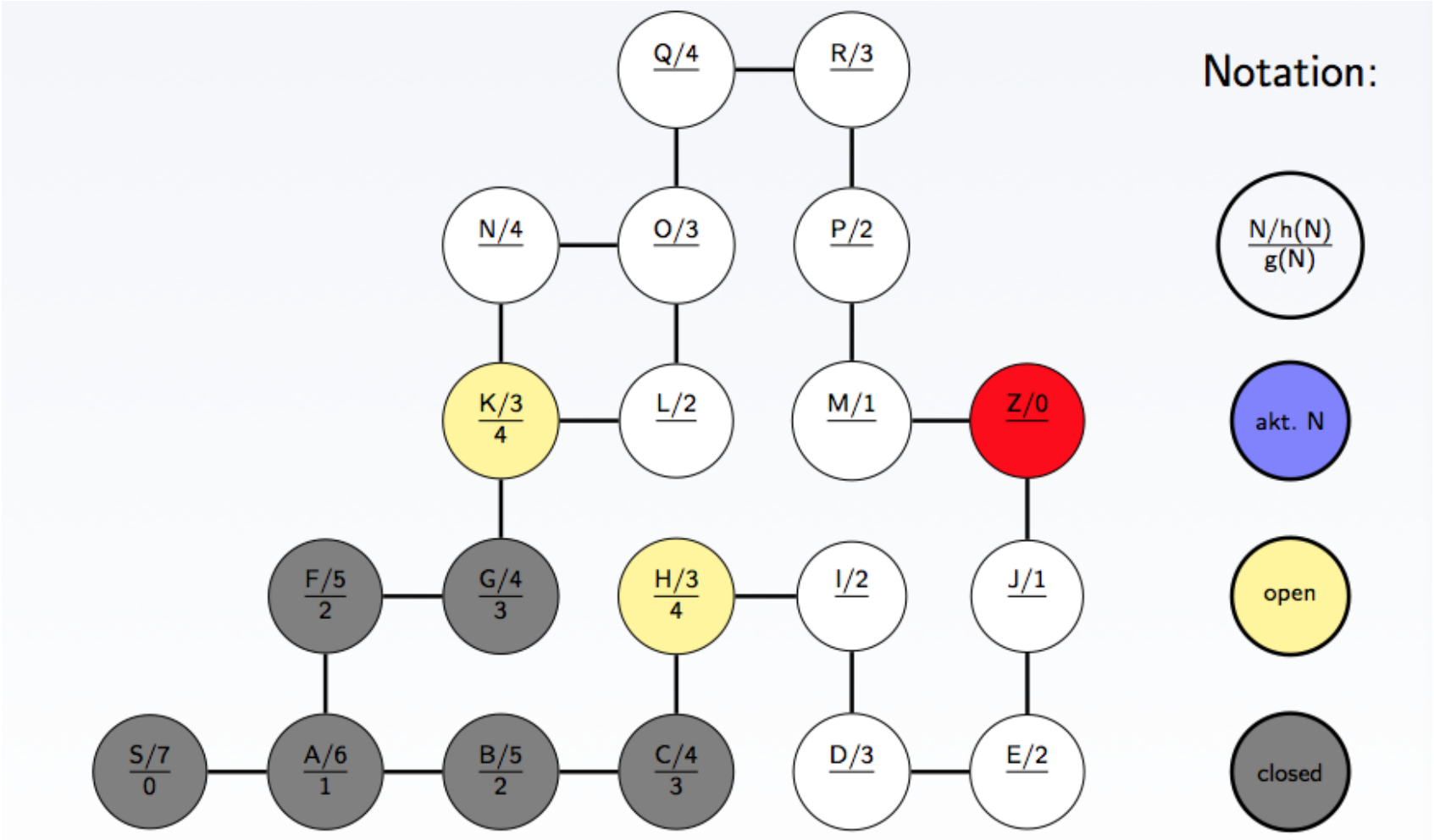
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



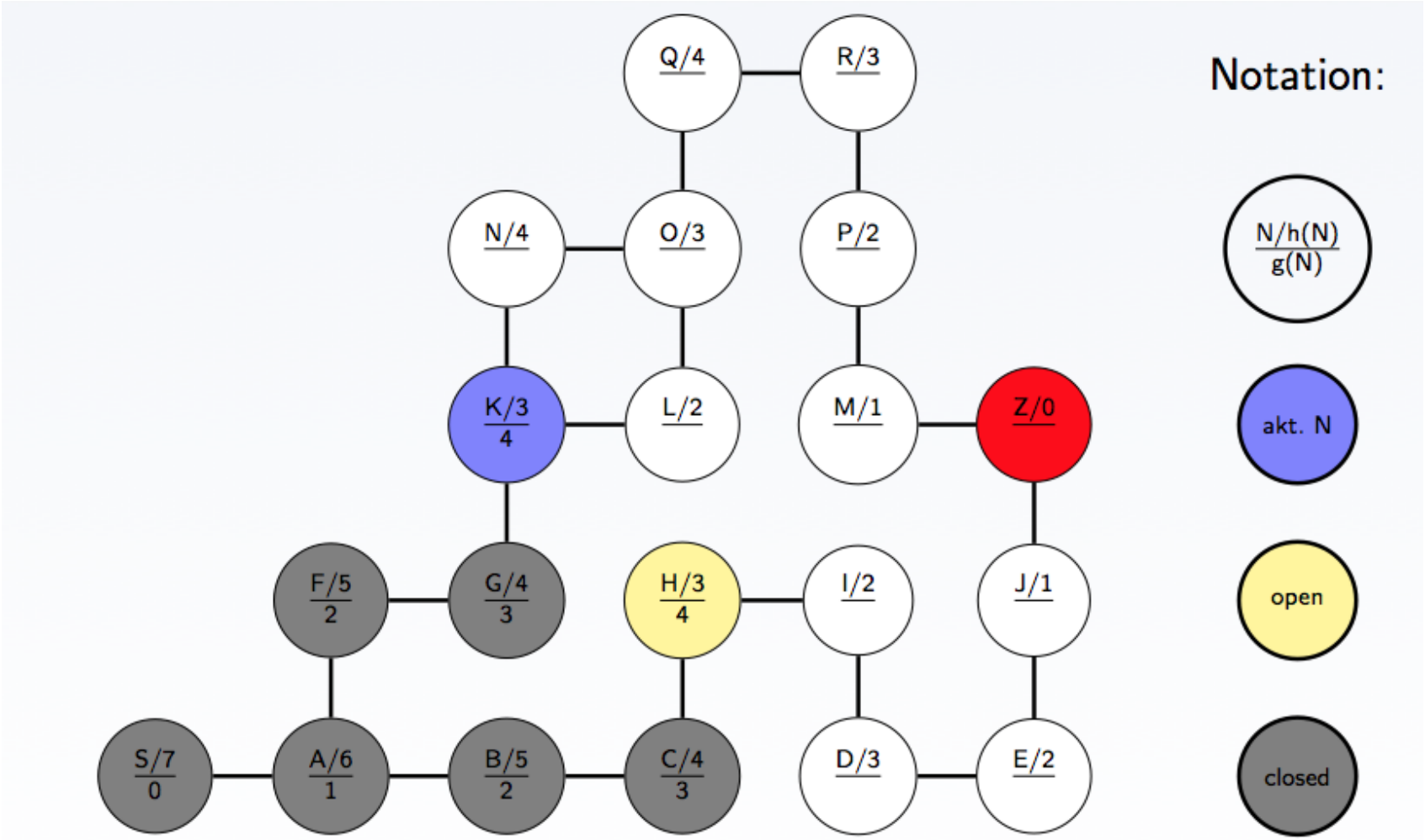
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



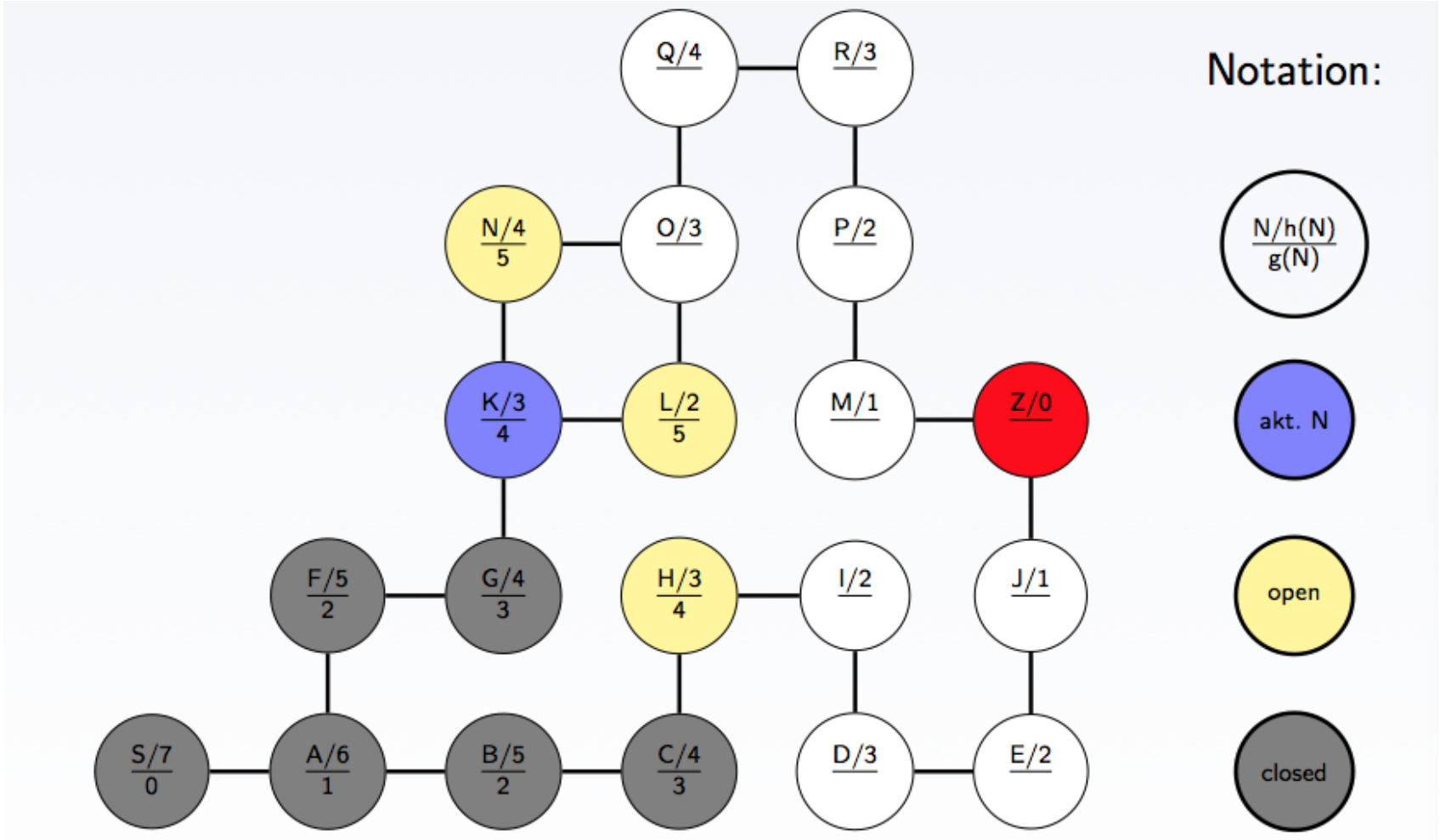
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



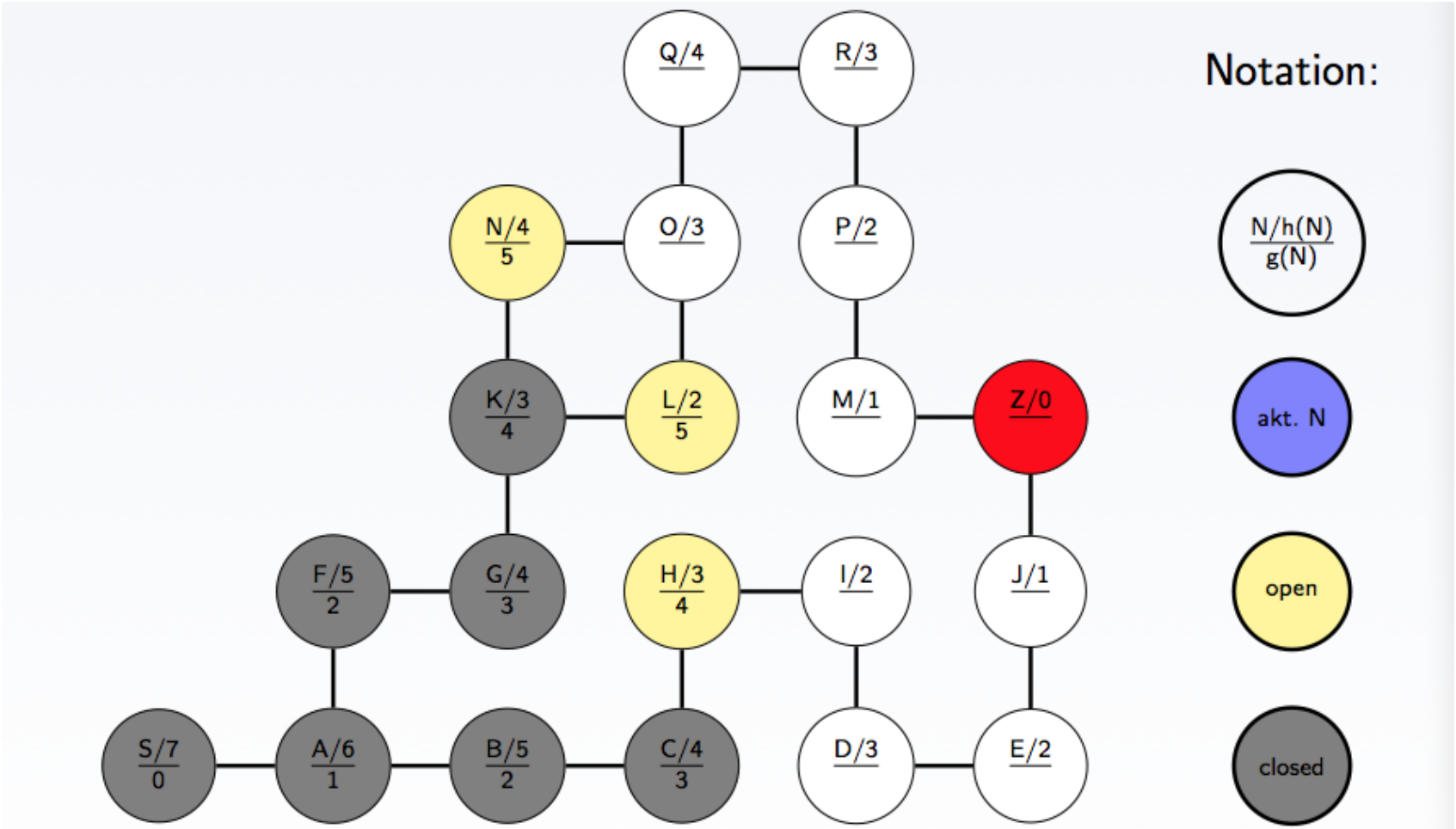
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



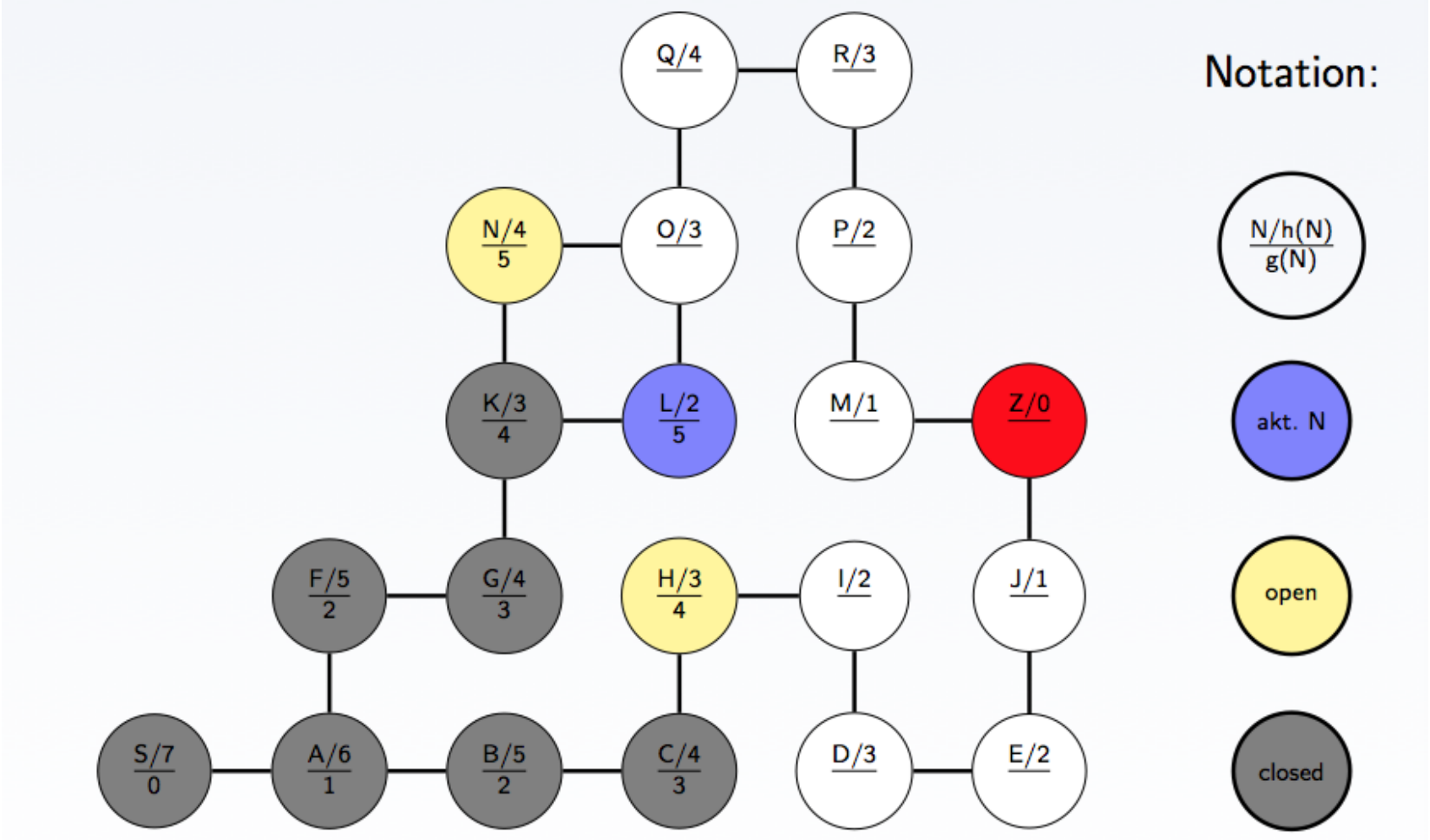
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



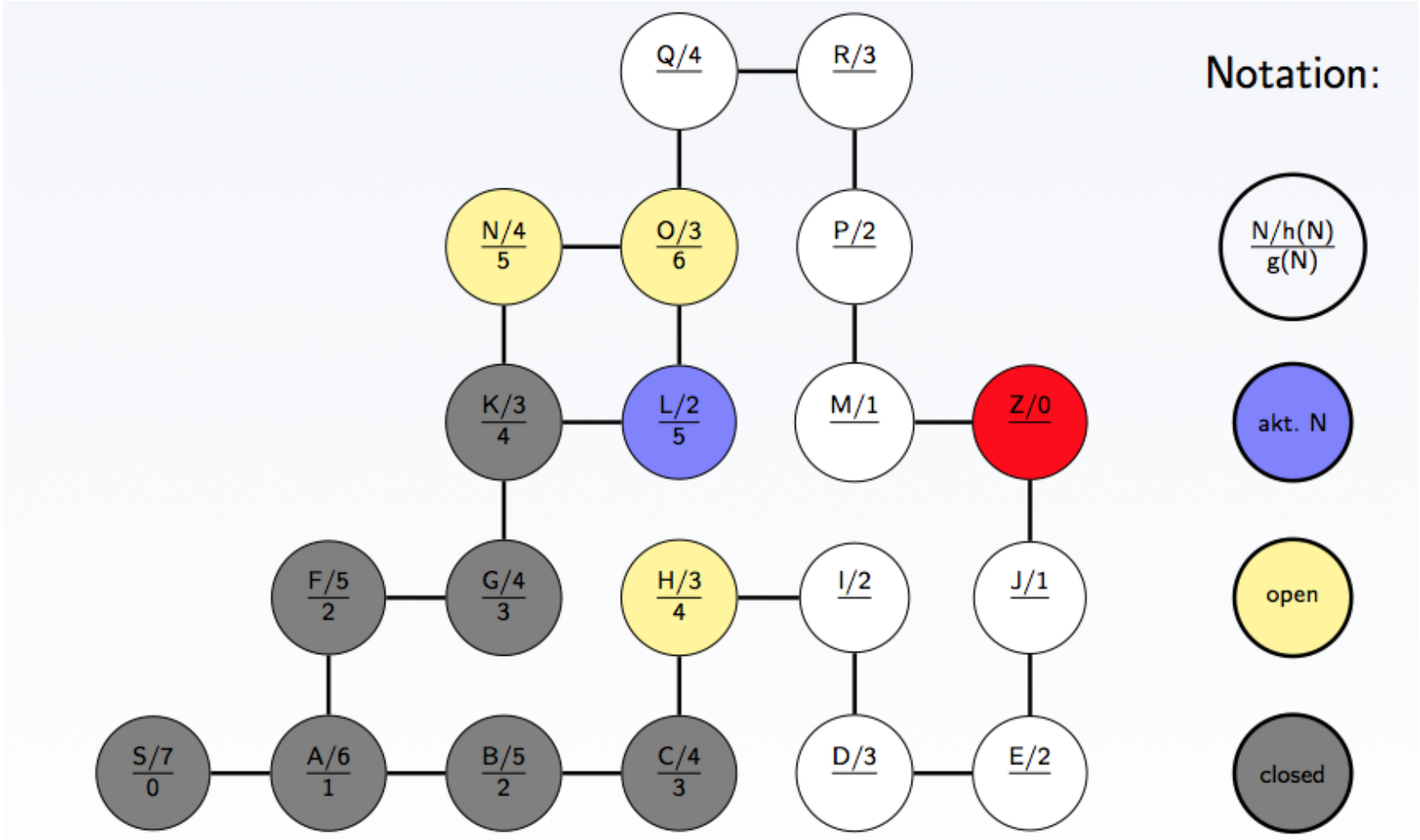
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



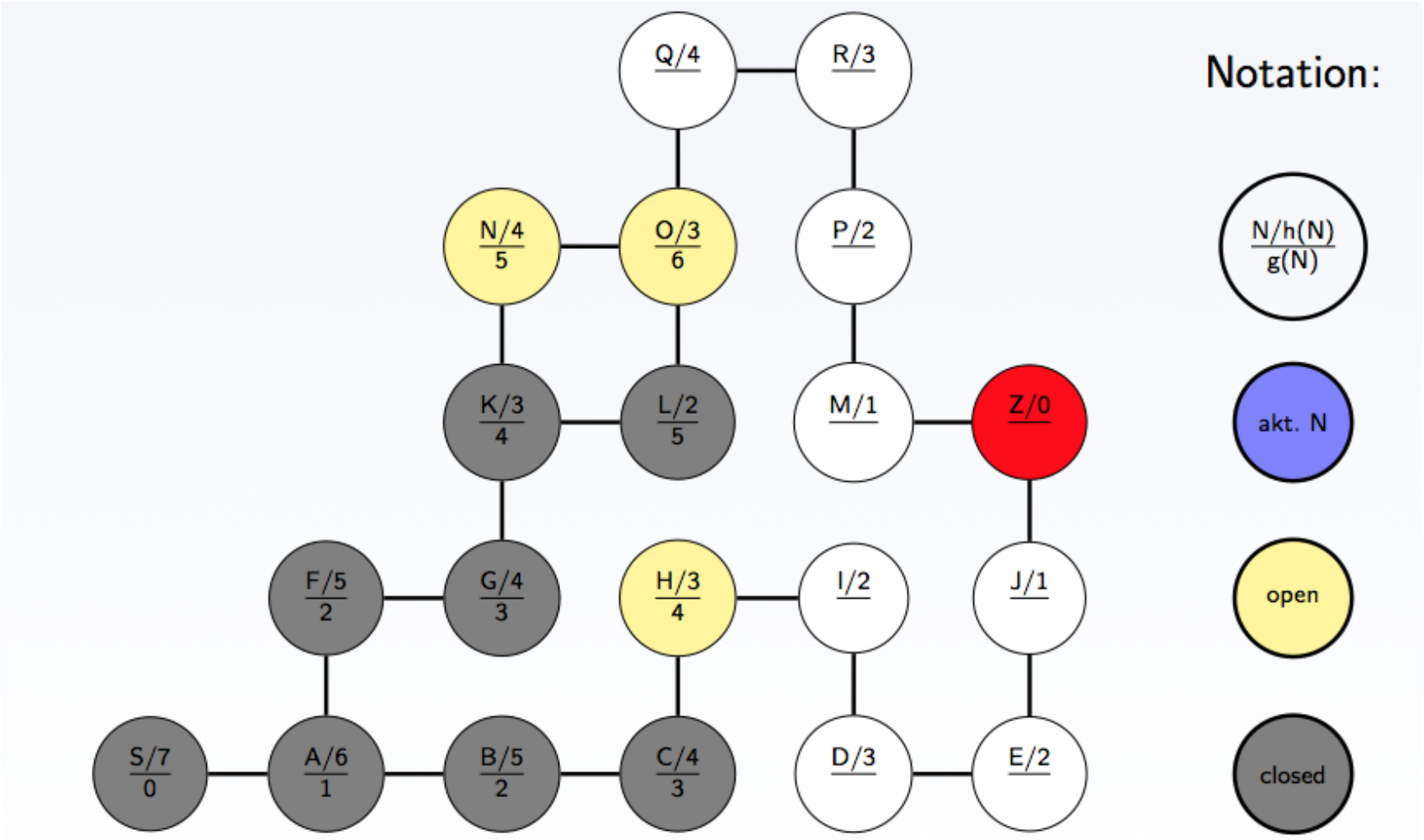
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



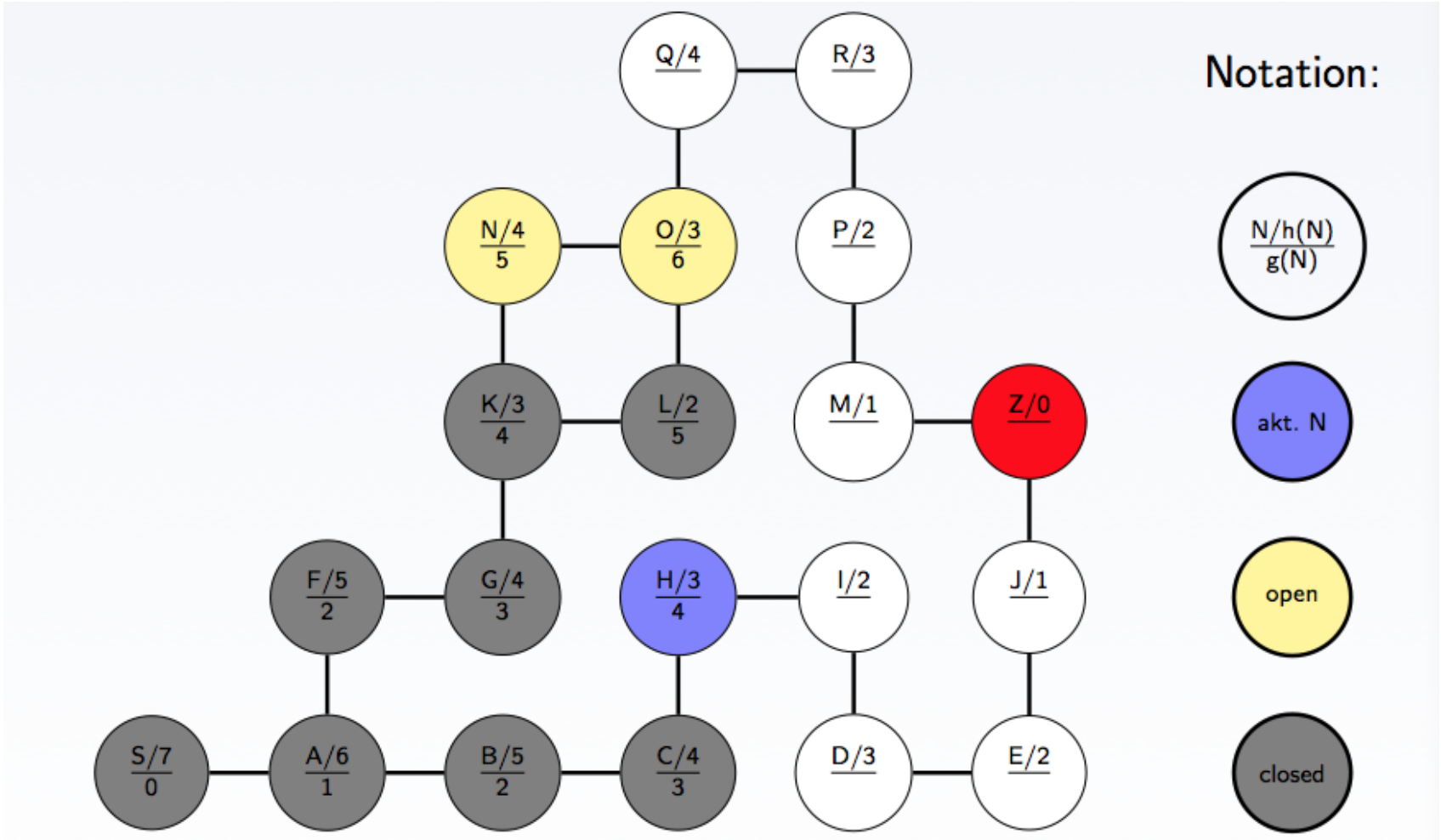
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



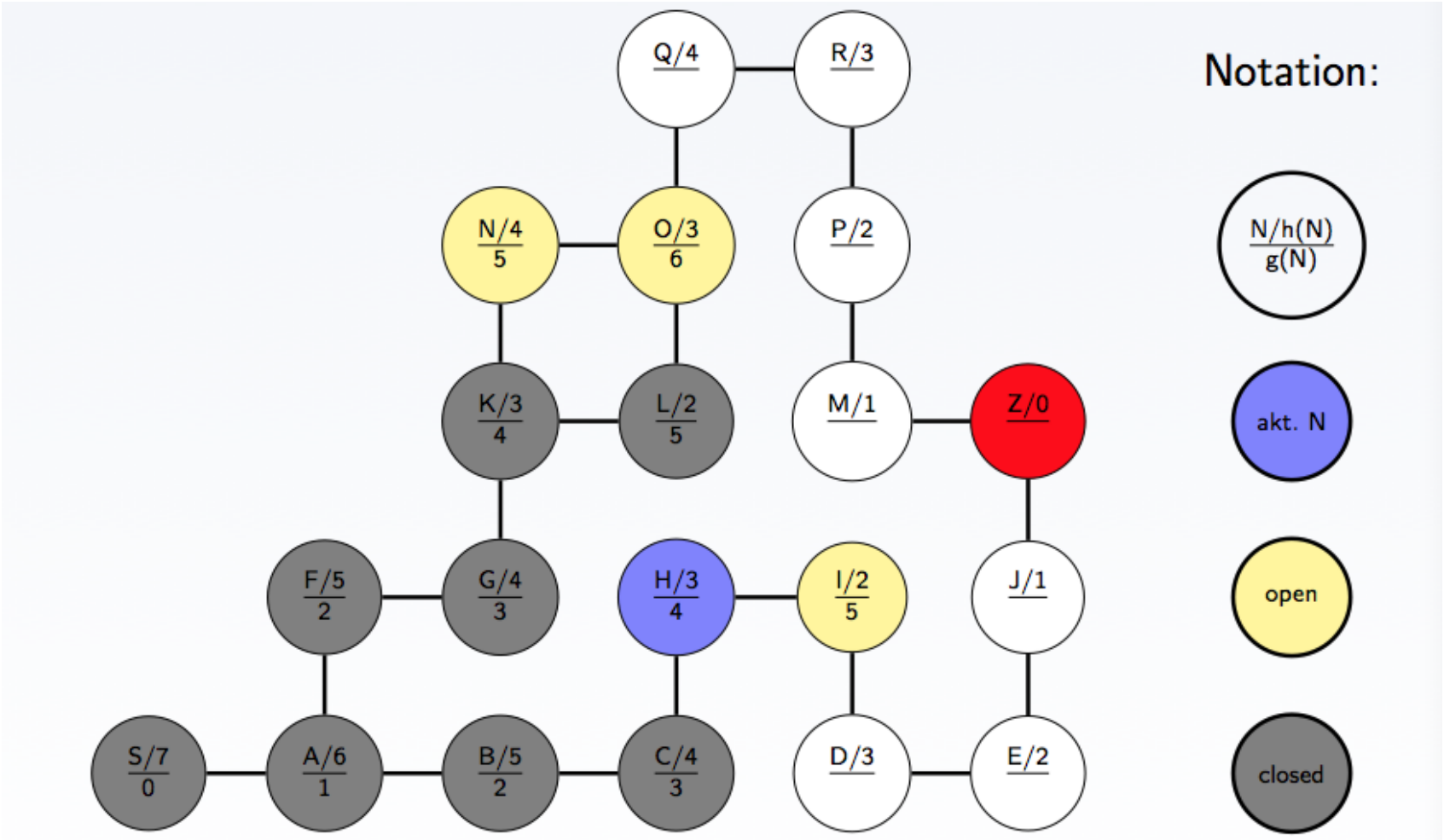
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



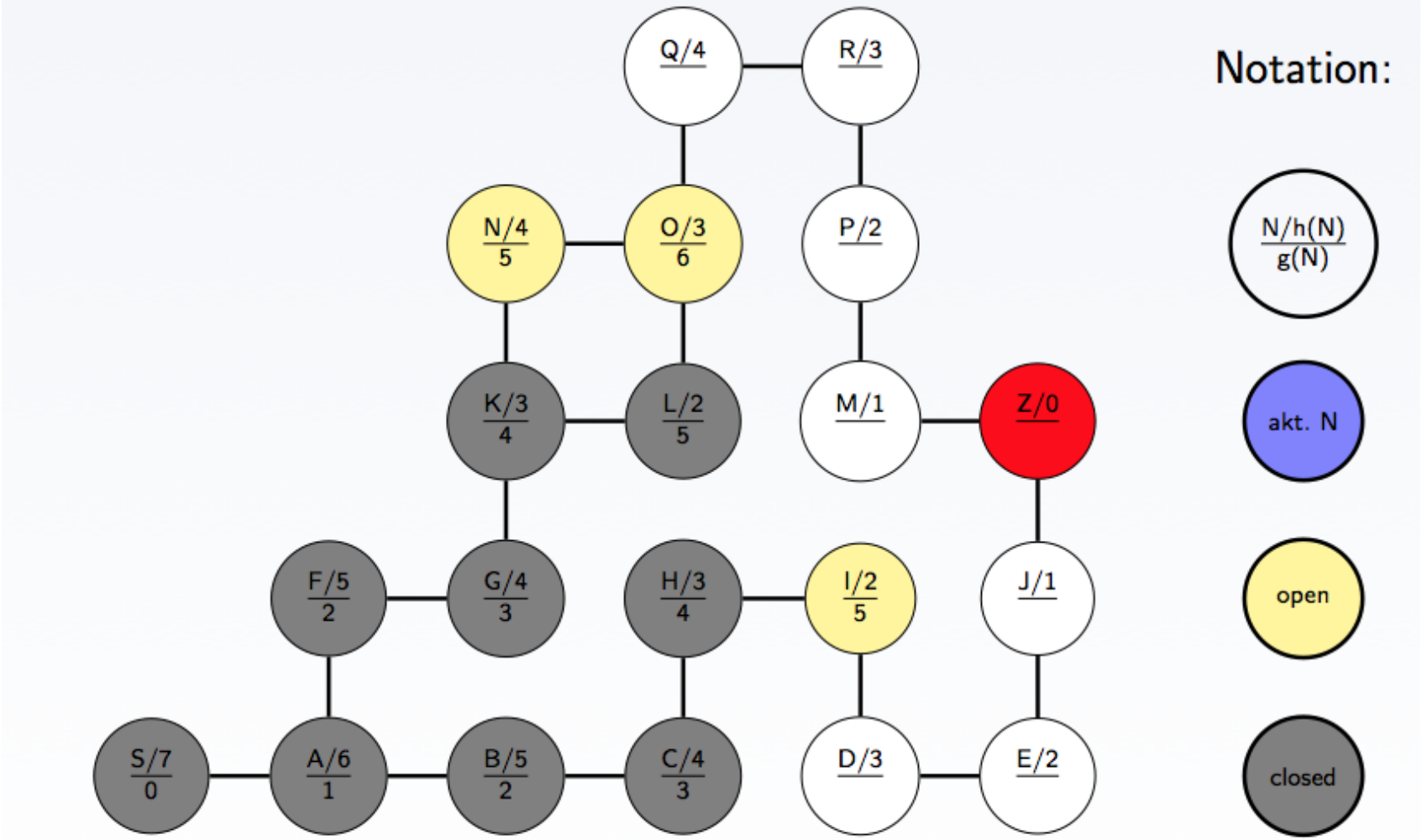
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



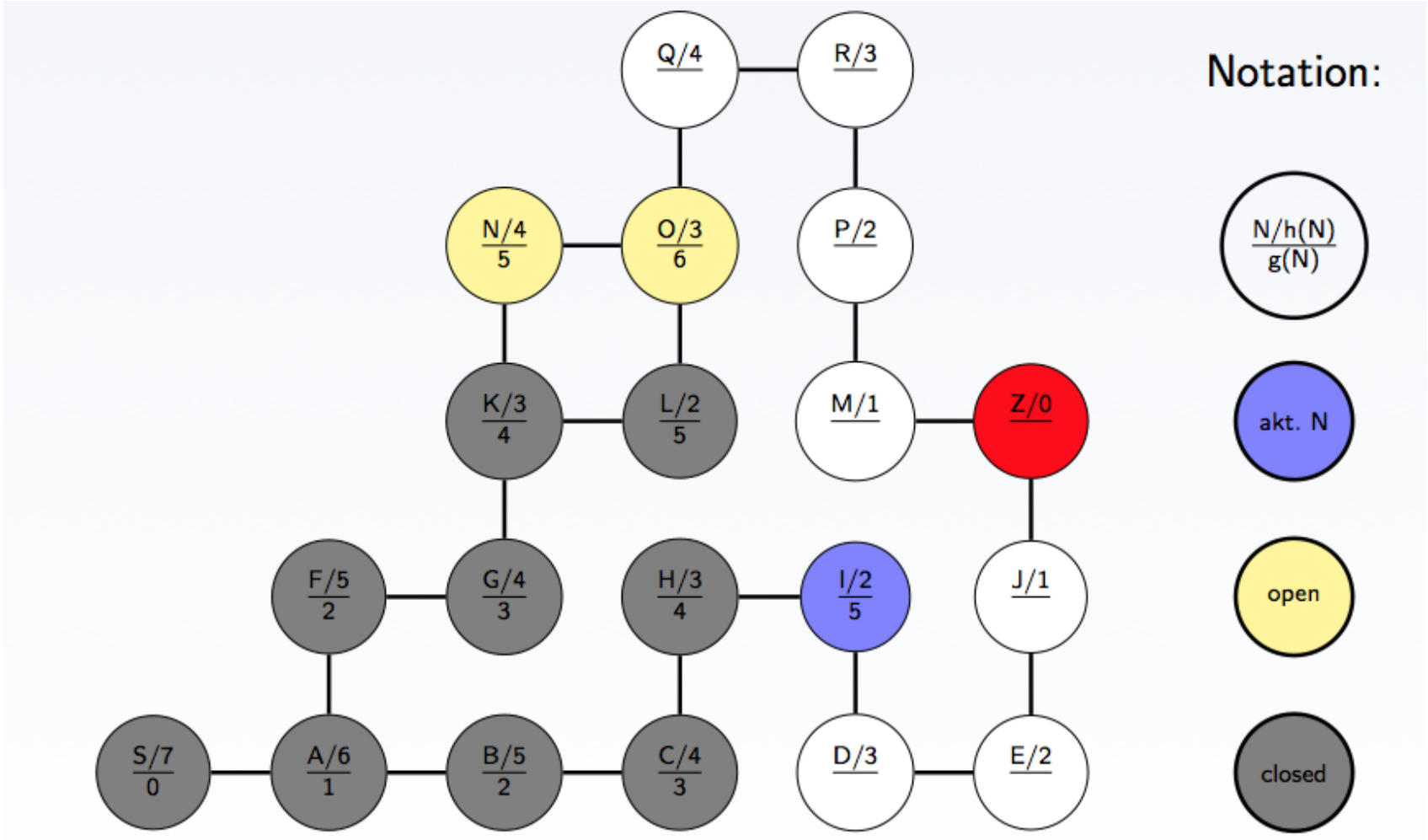
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



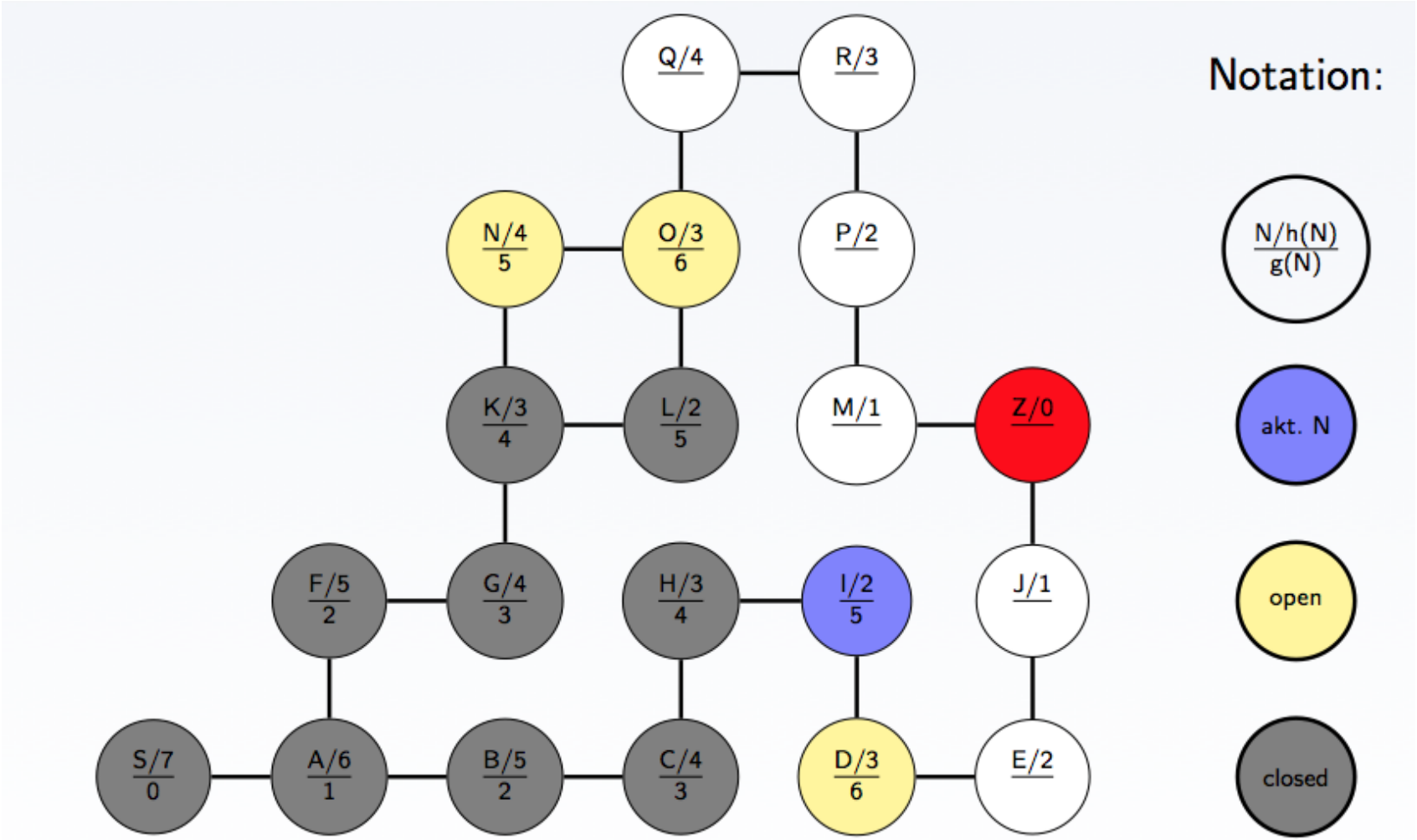
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



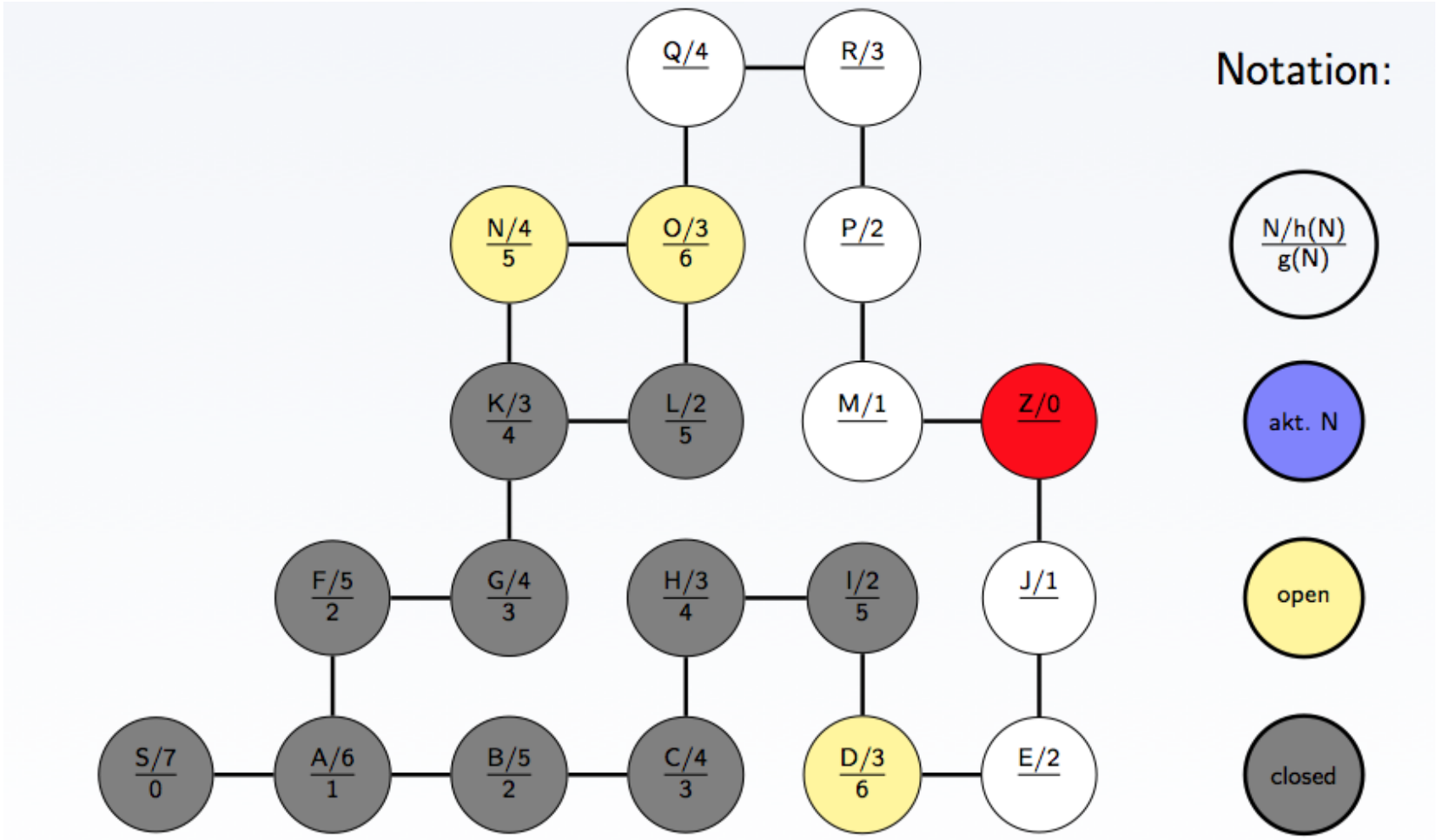
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



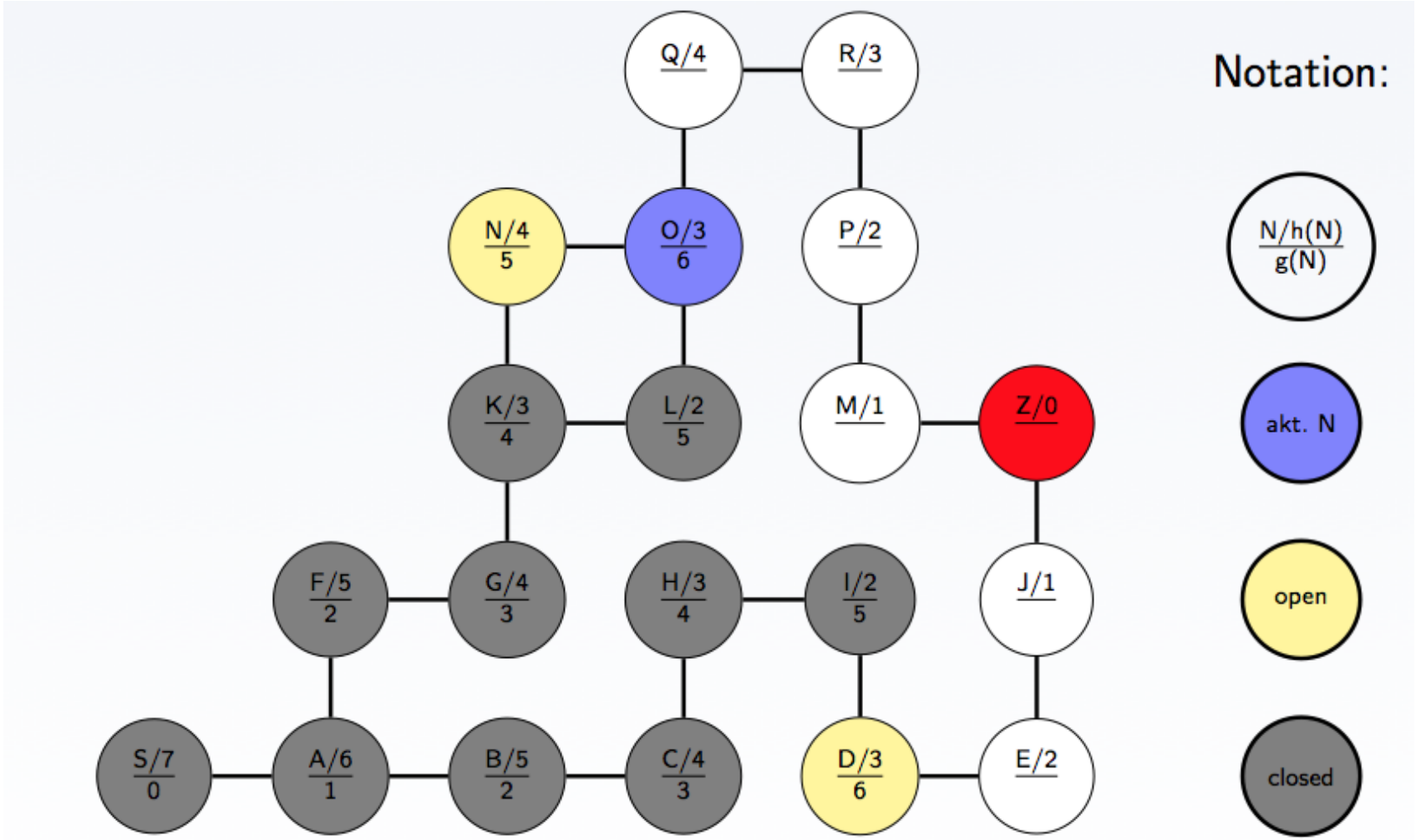
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



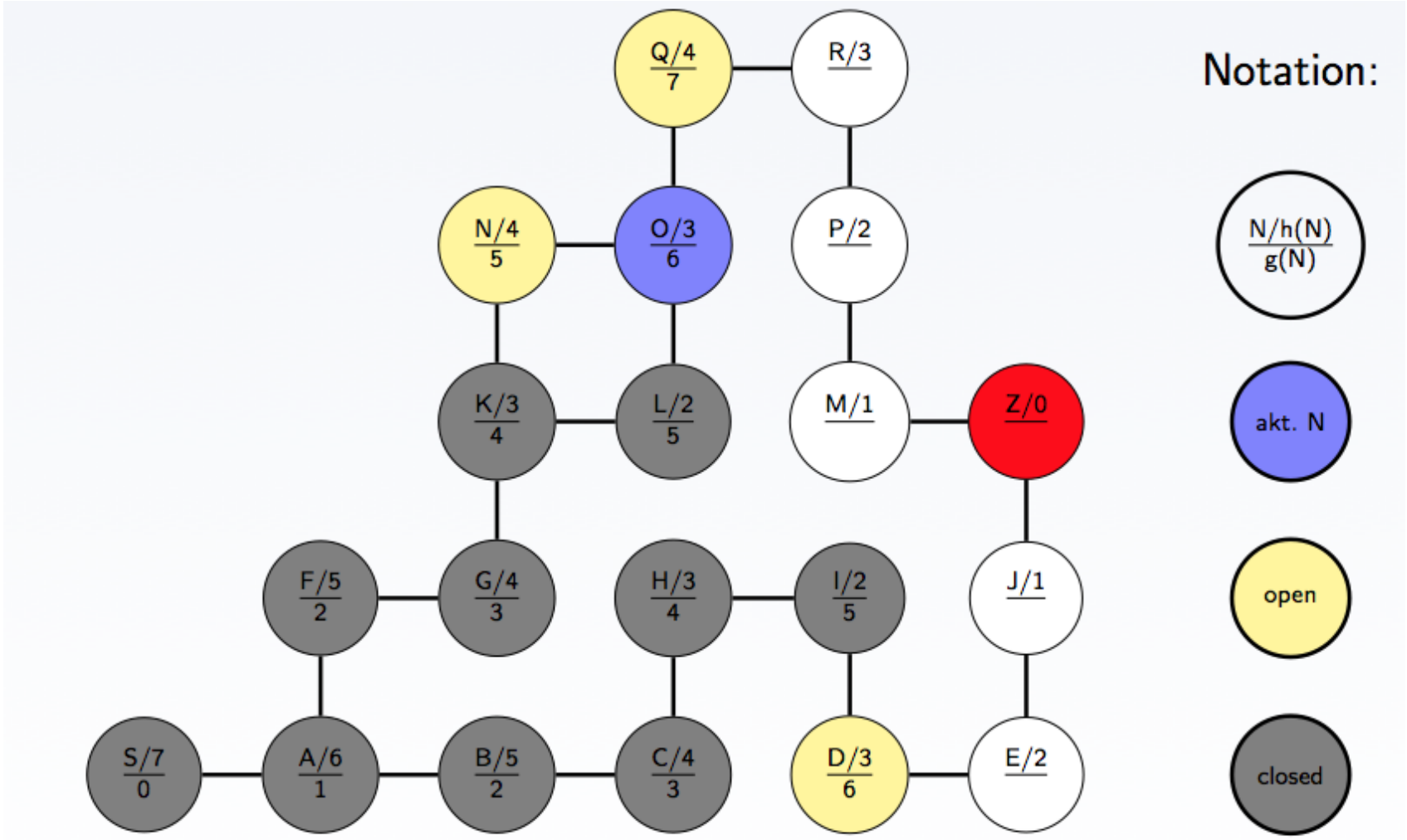
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



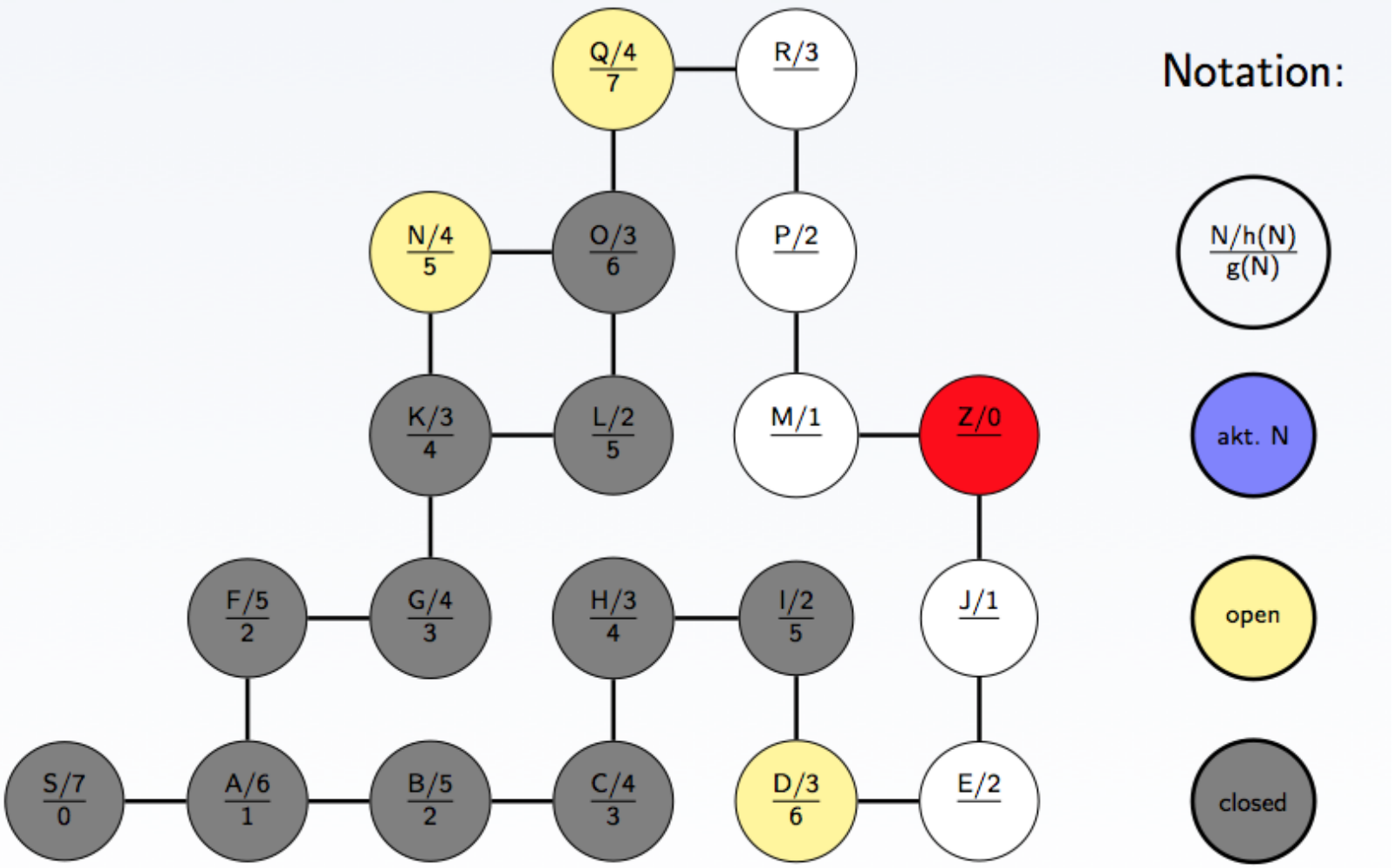
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

BEISPIEL



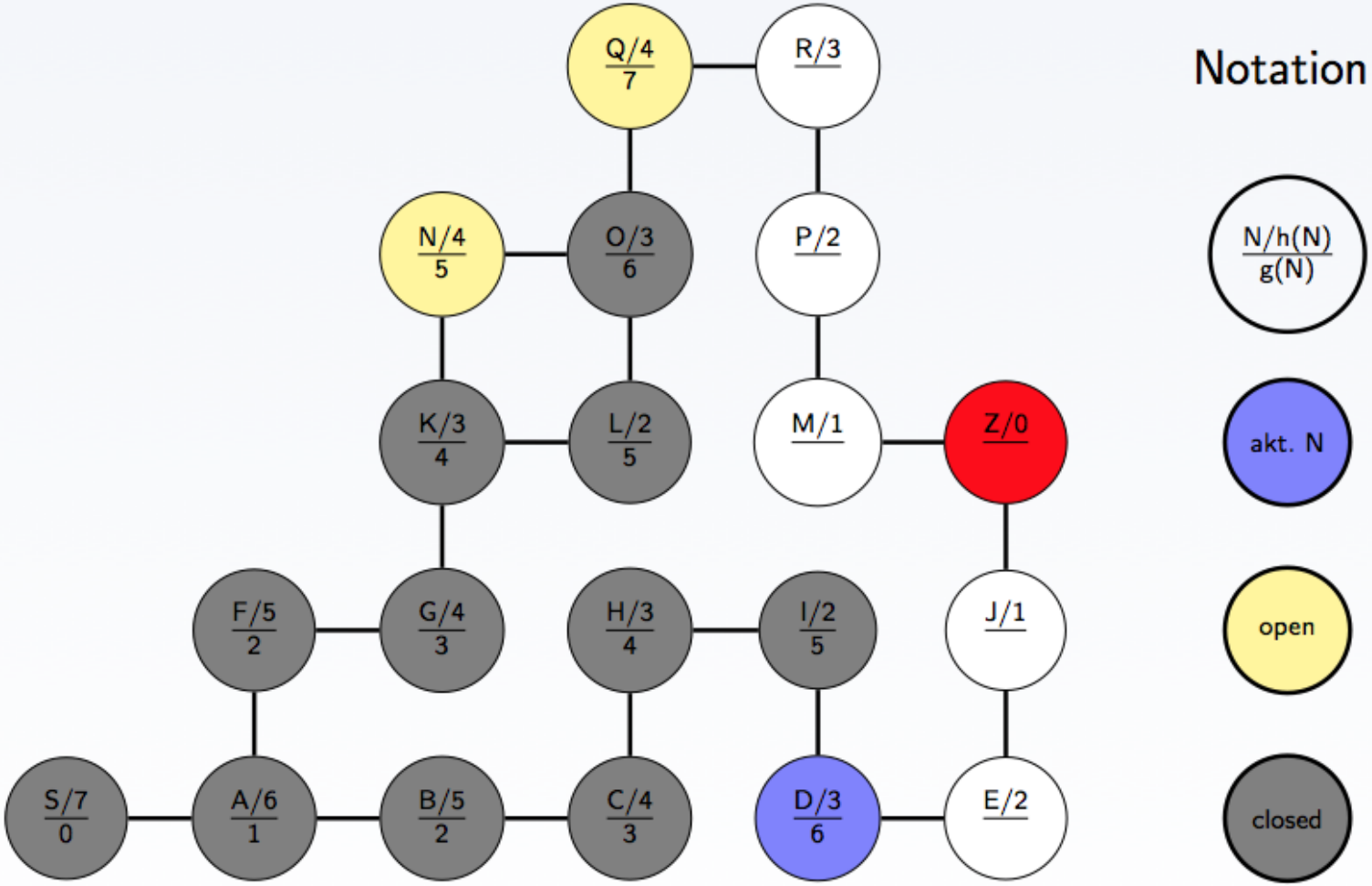
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



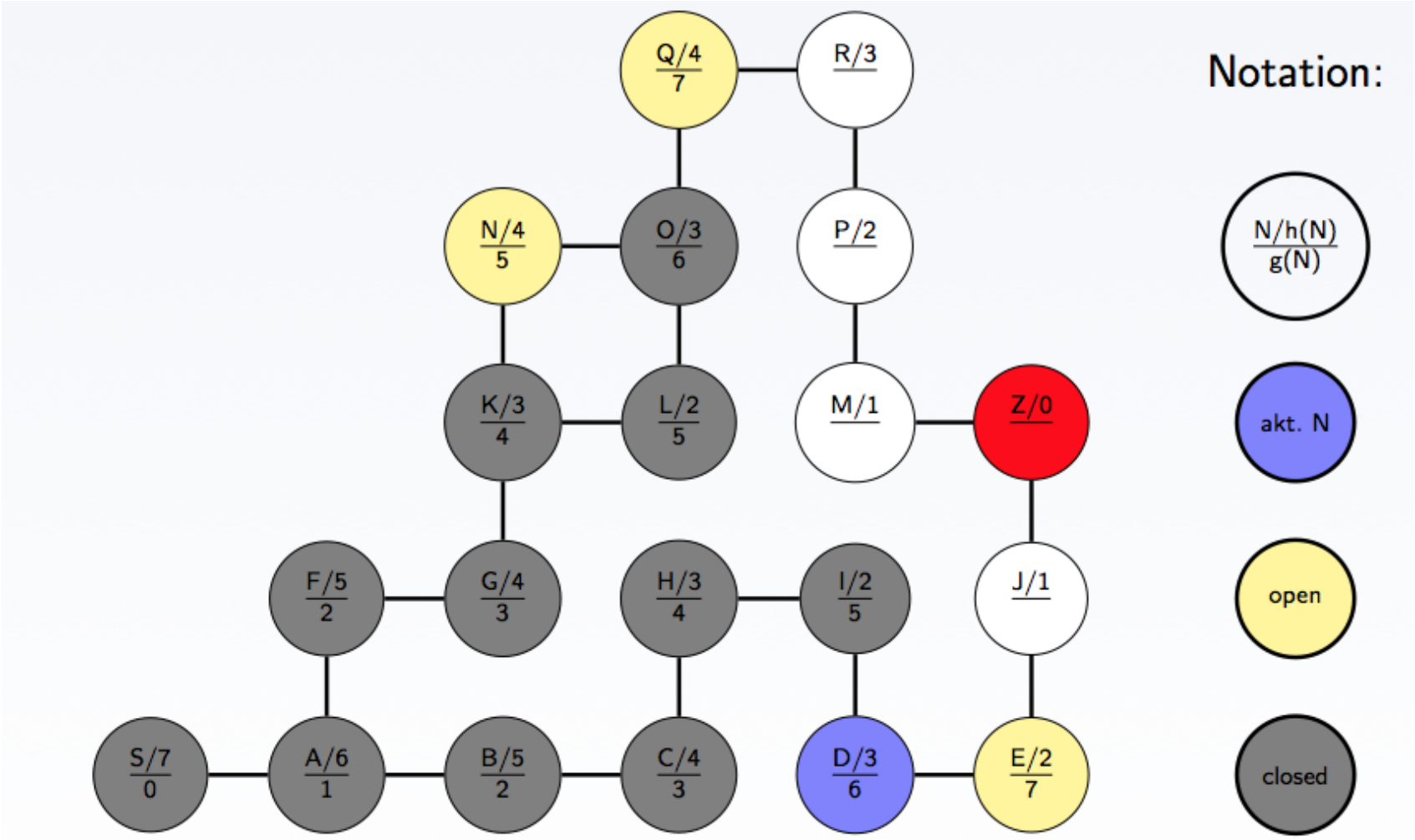
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



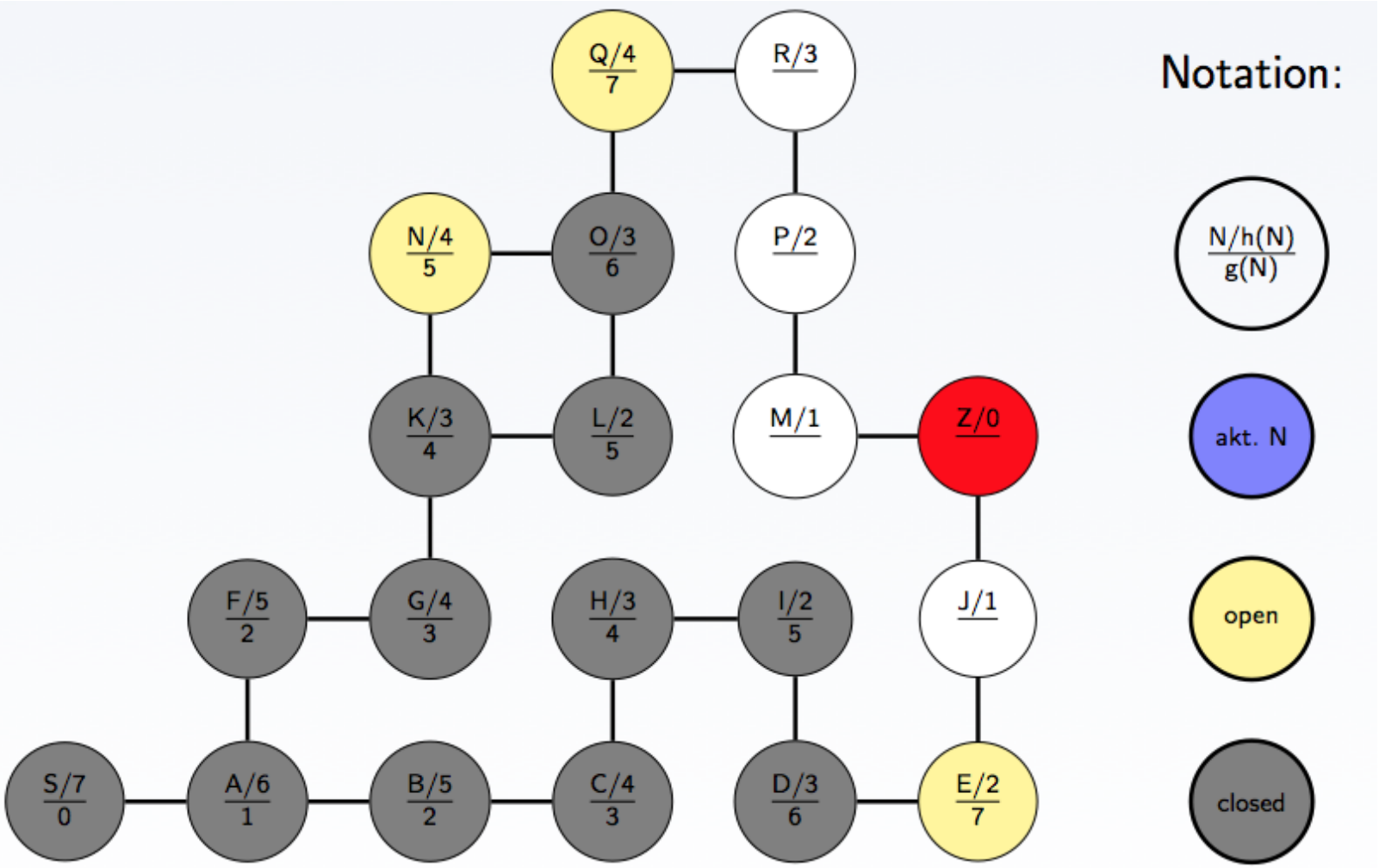
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

BEISPIEL



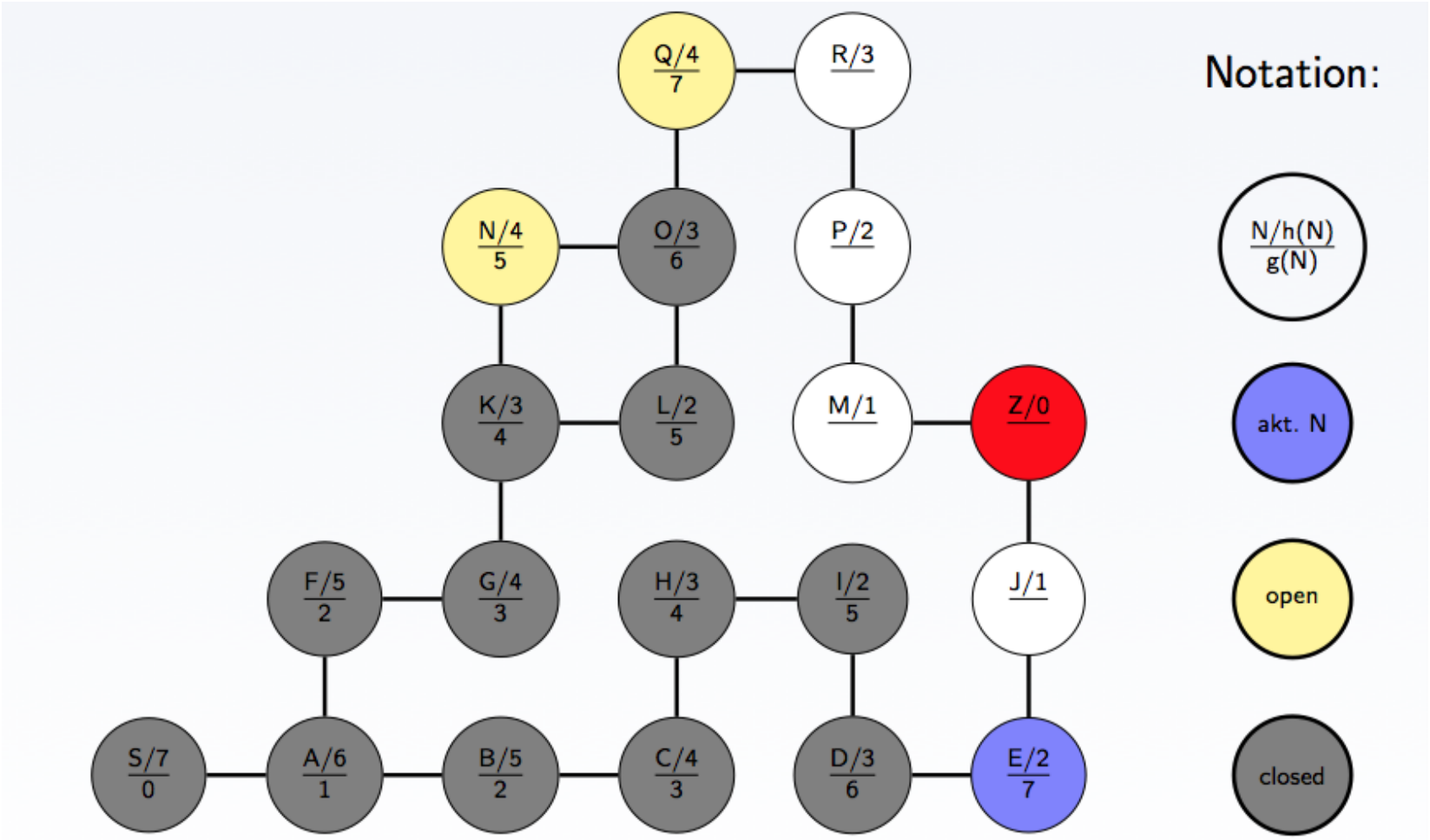
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



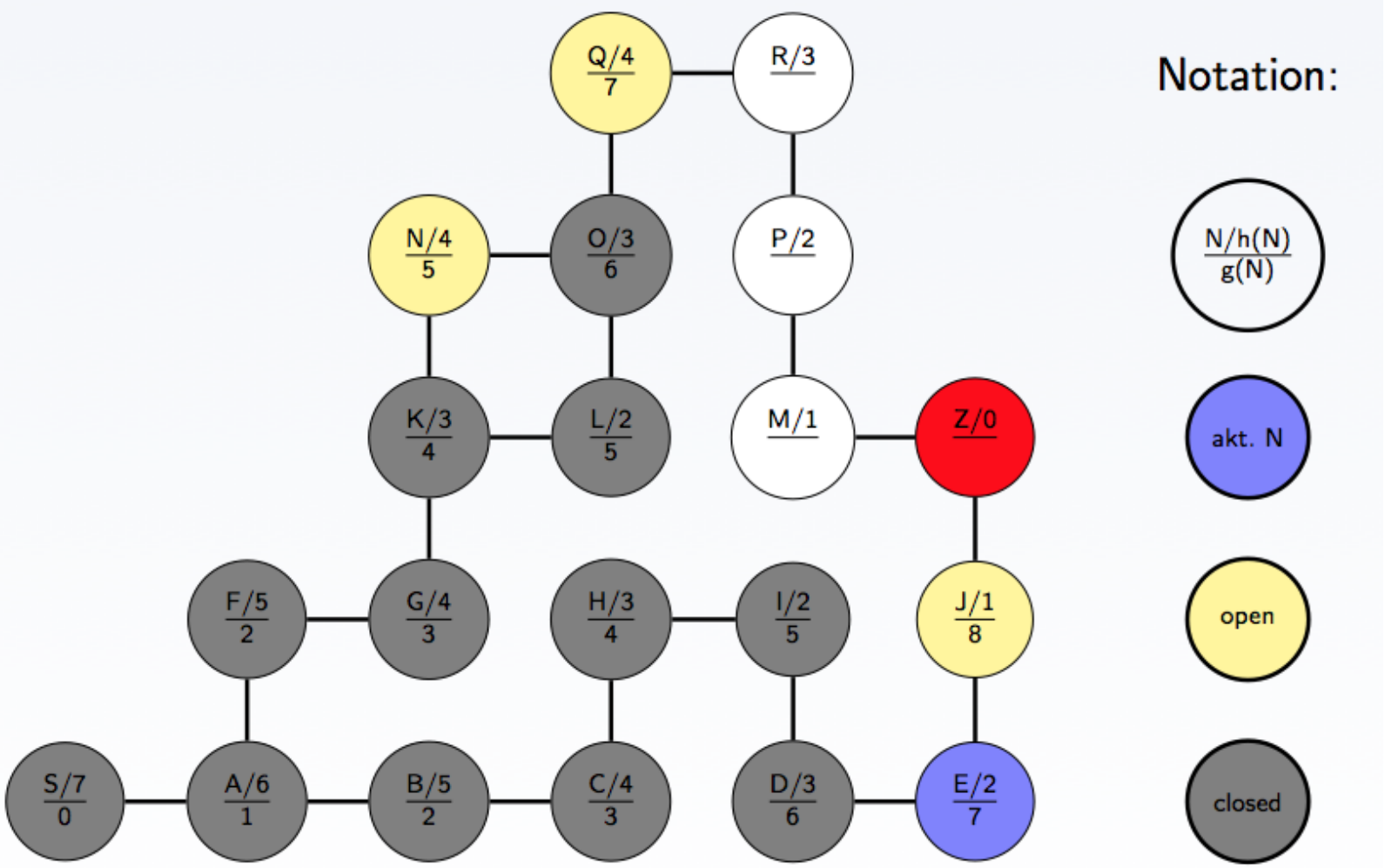
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



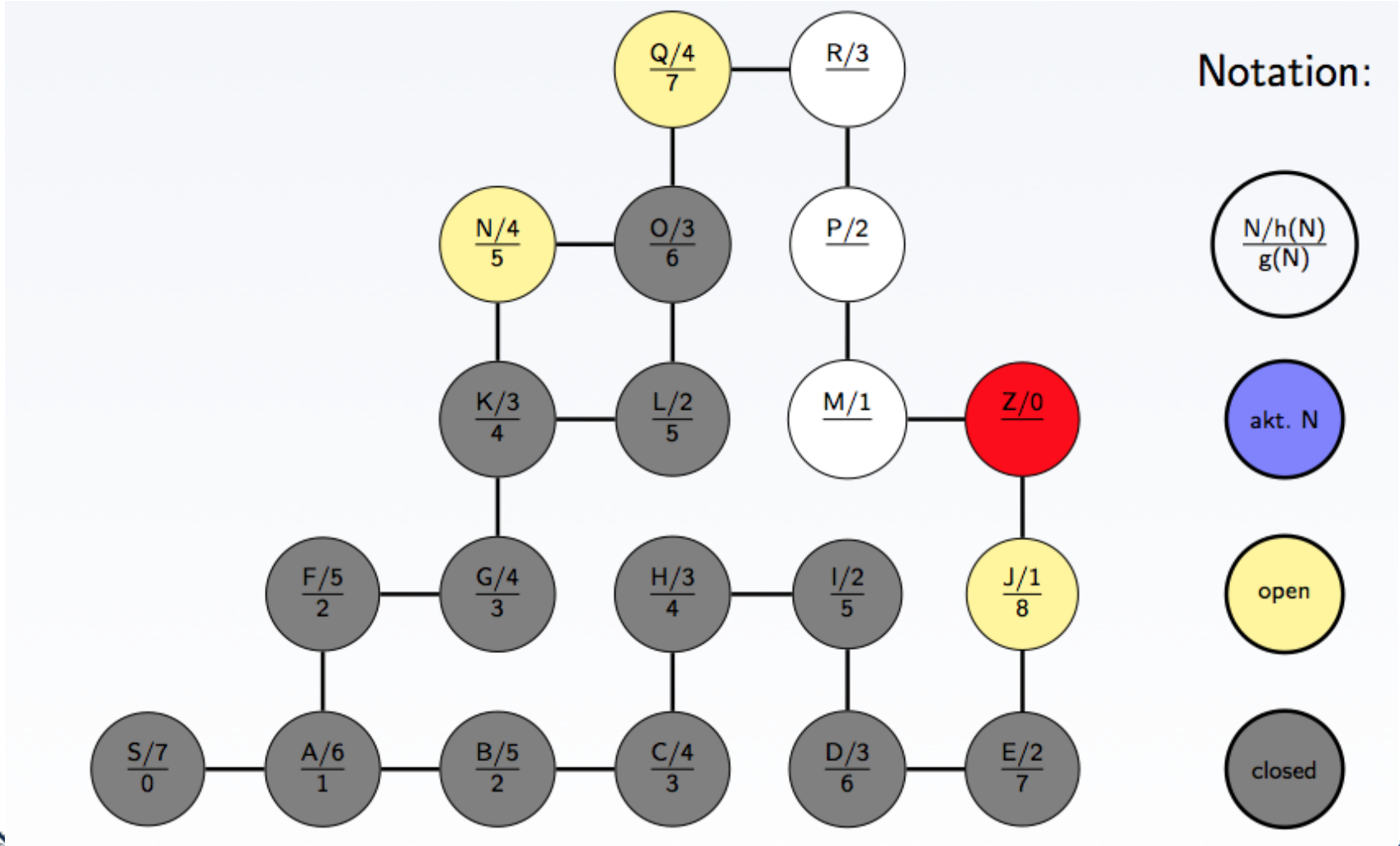
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



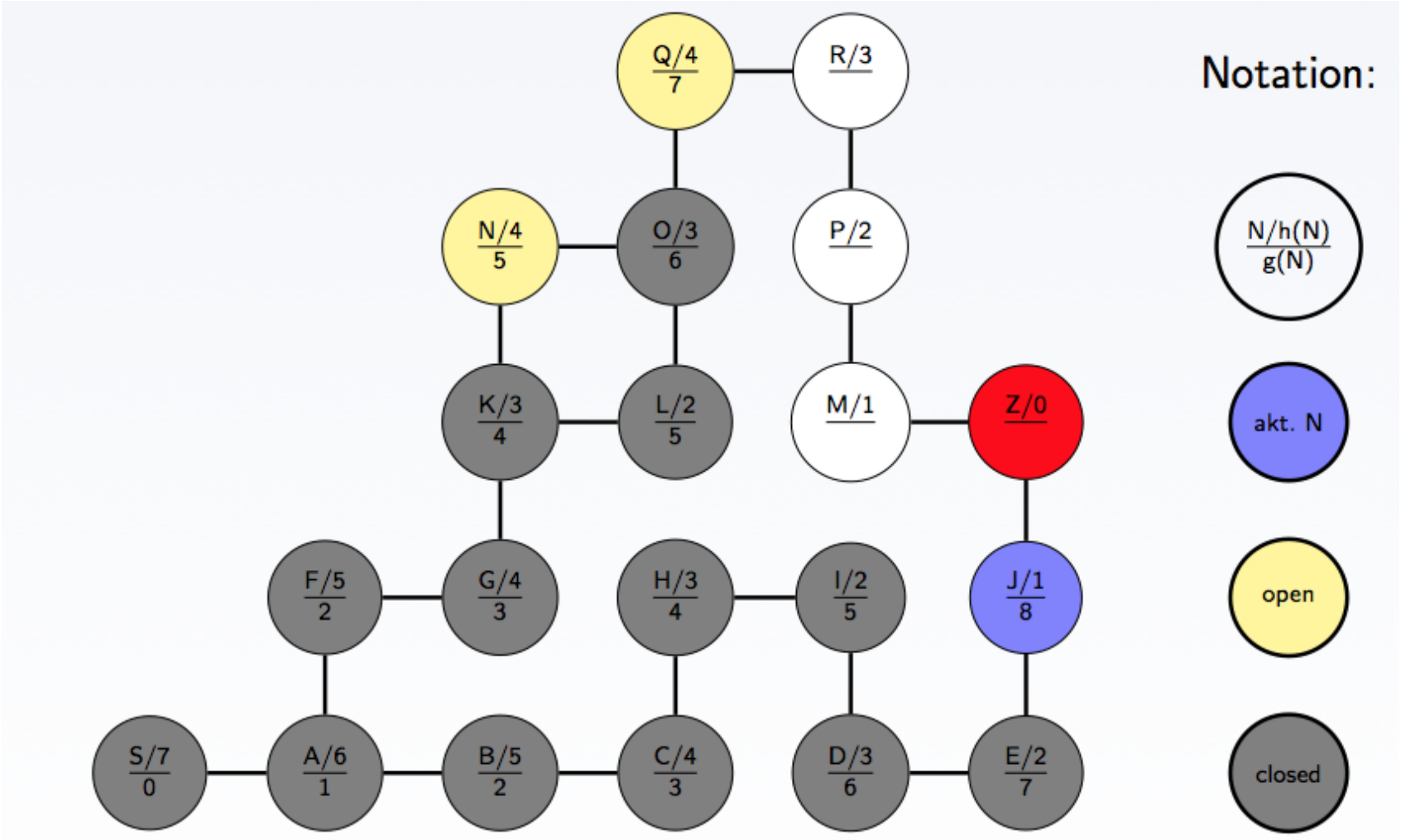
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



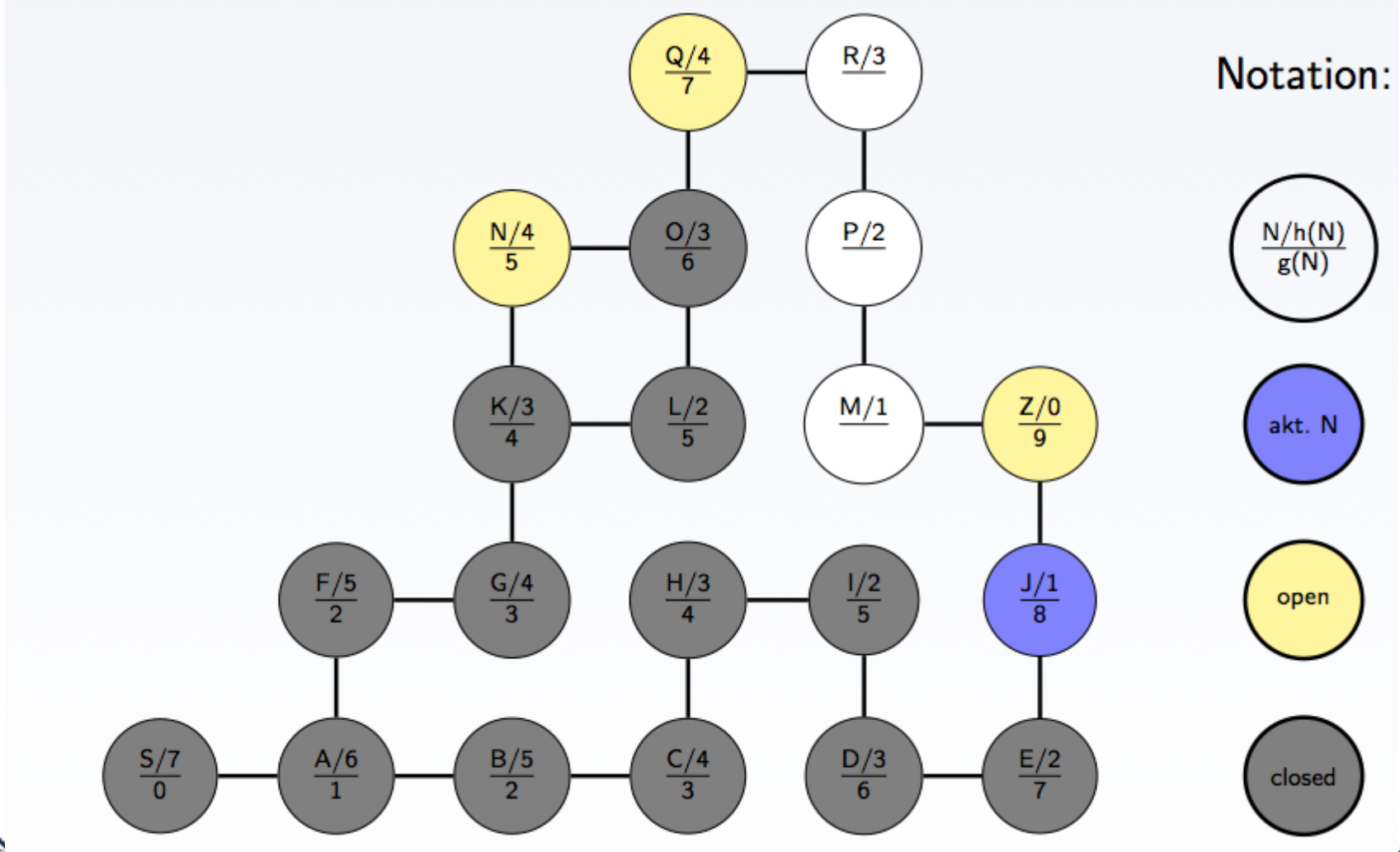
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



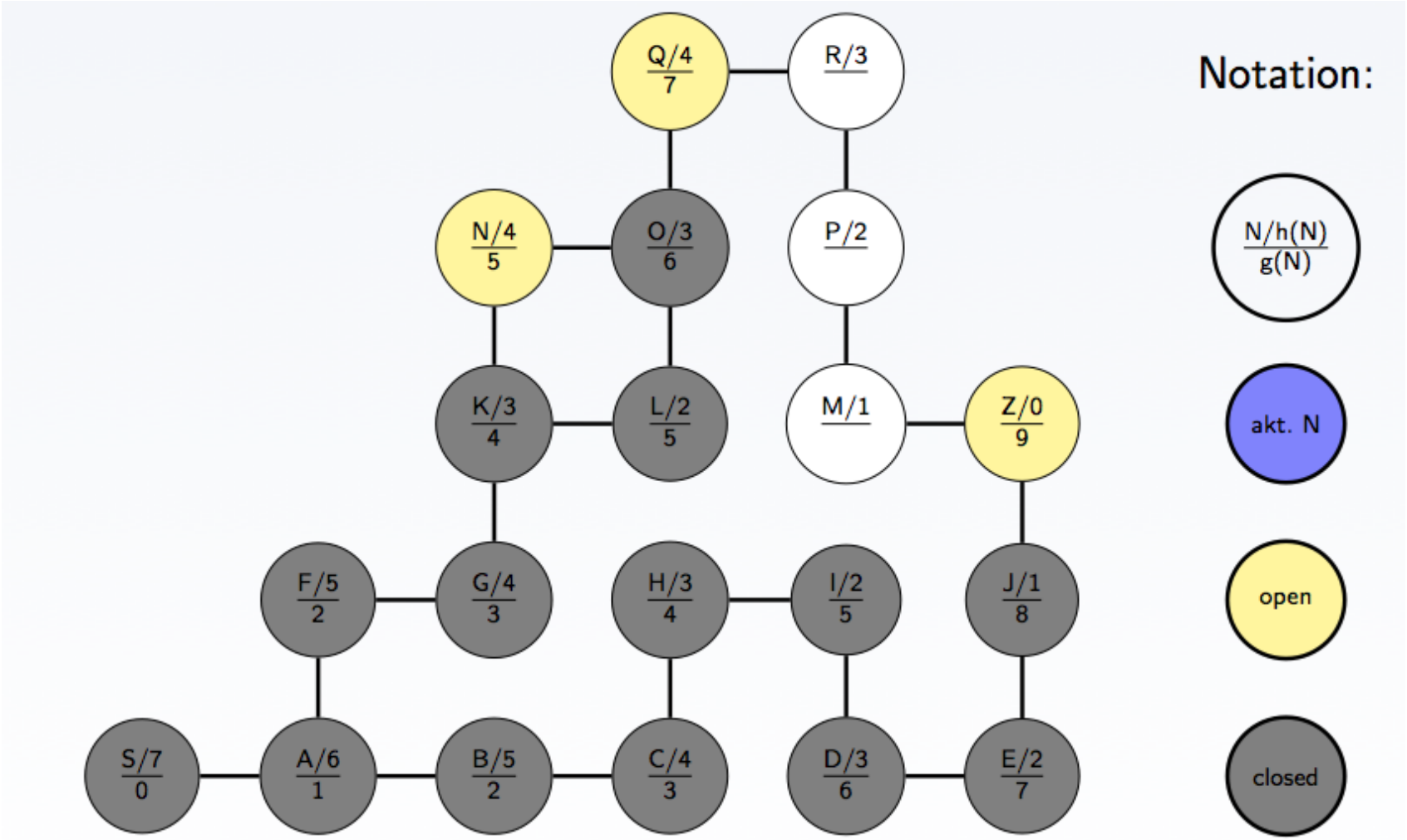
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



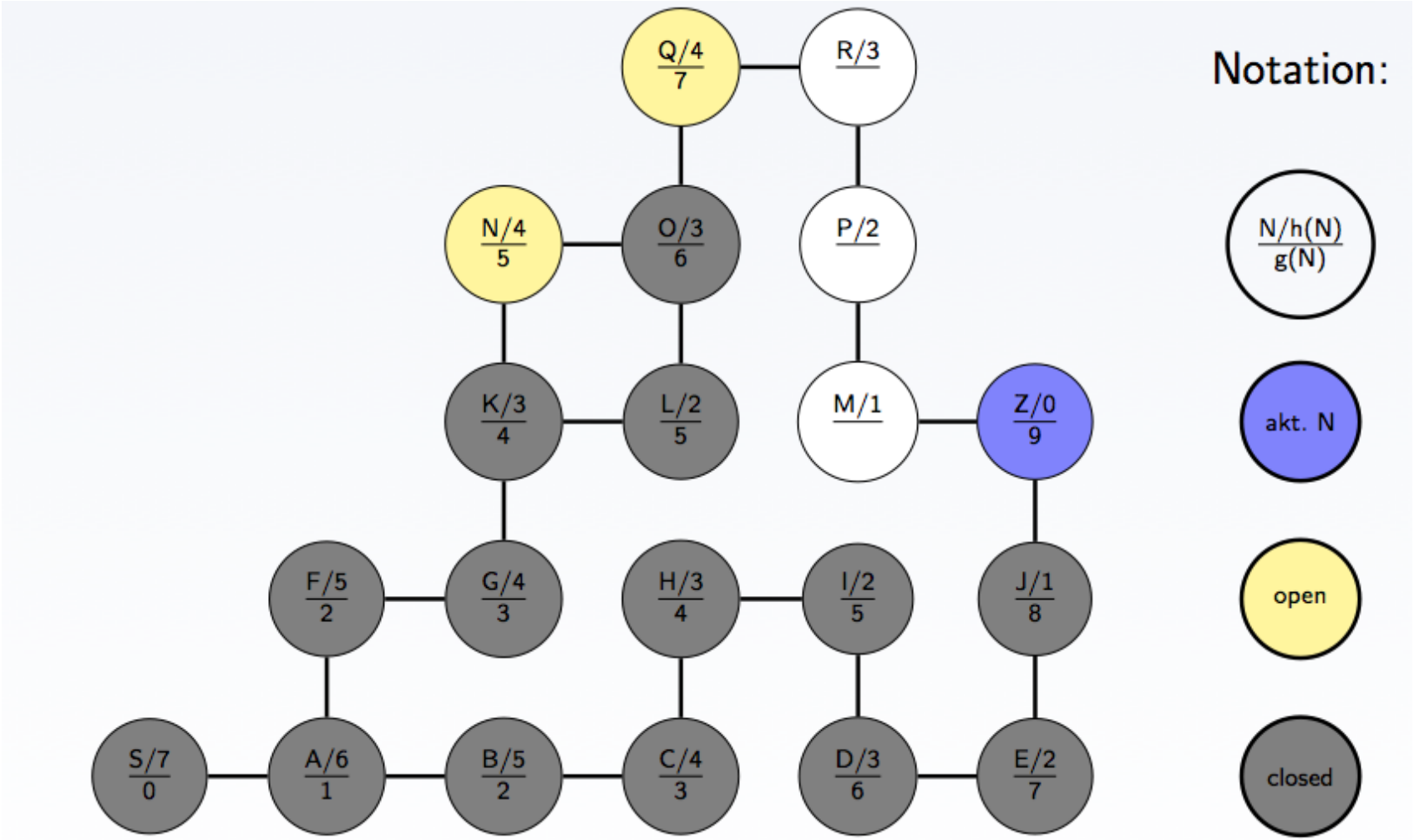
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

BEISPIEL



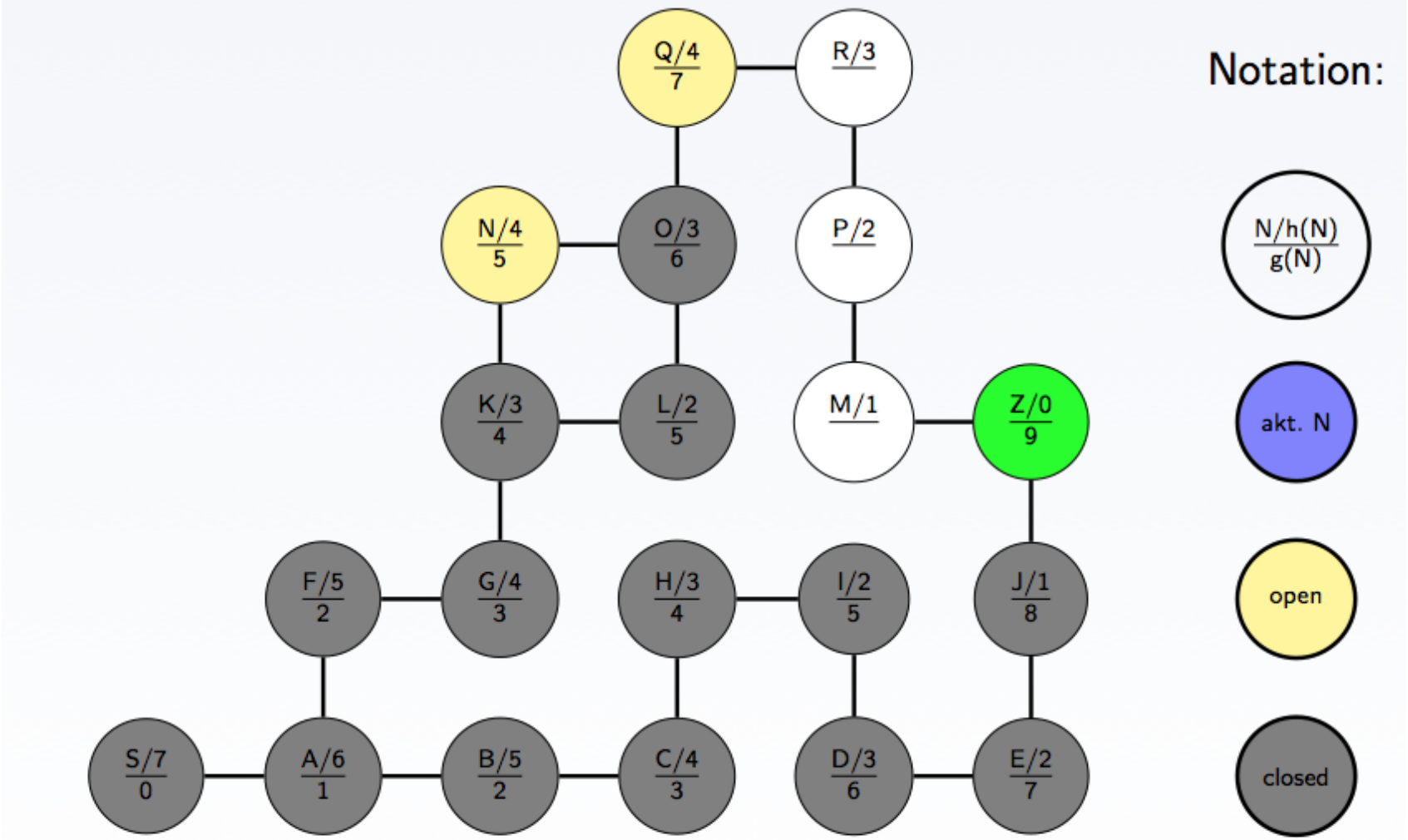
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



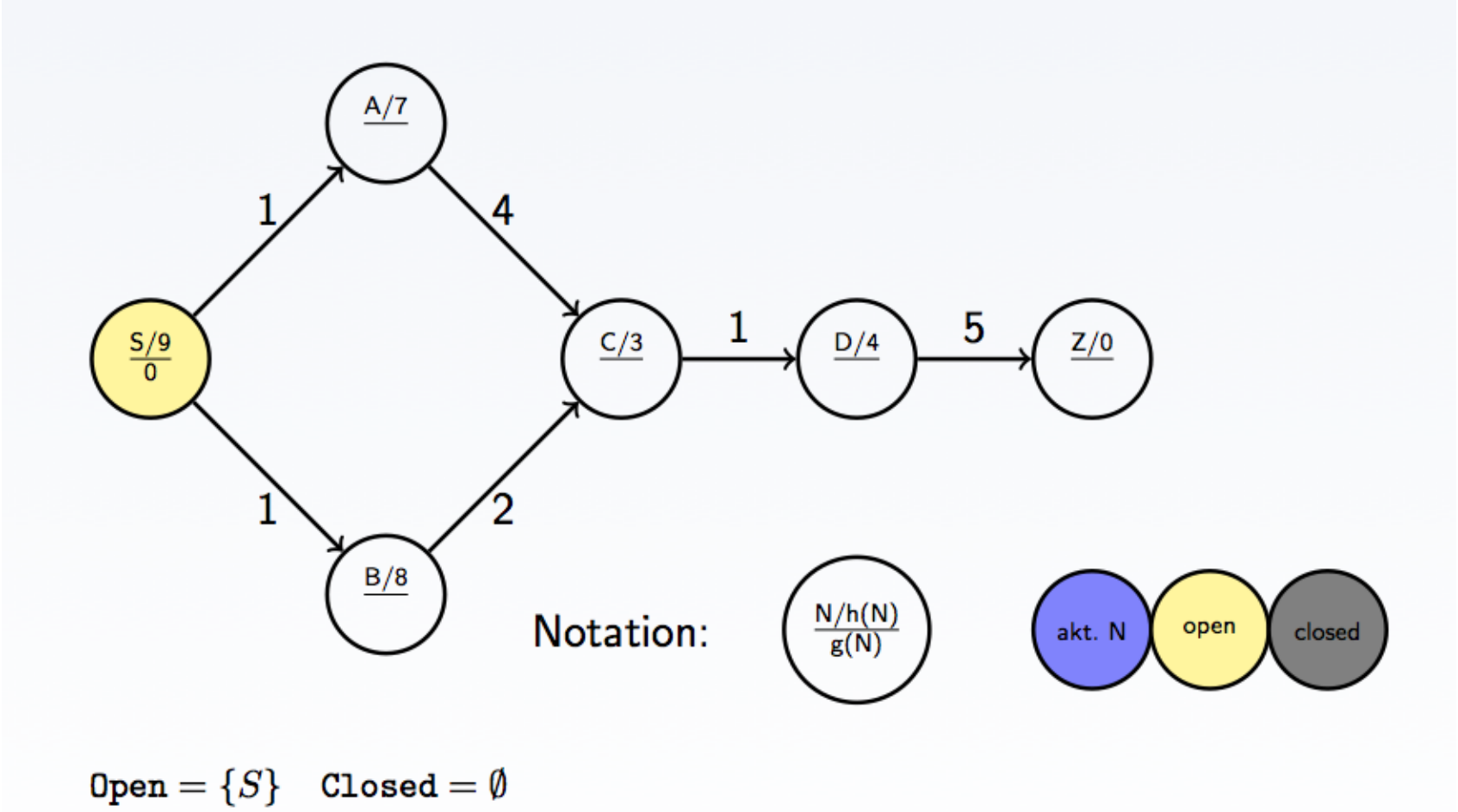
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



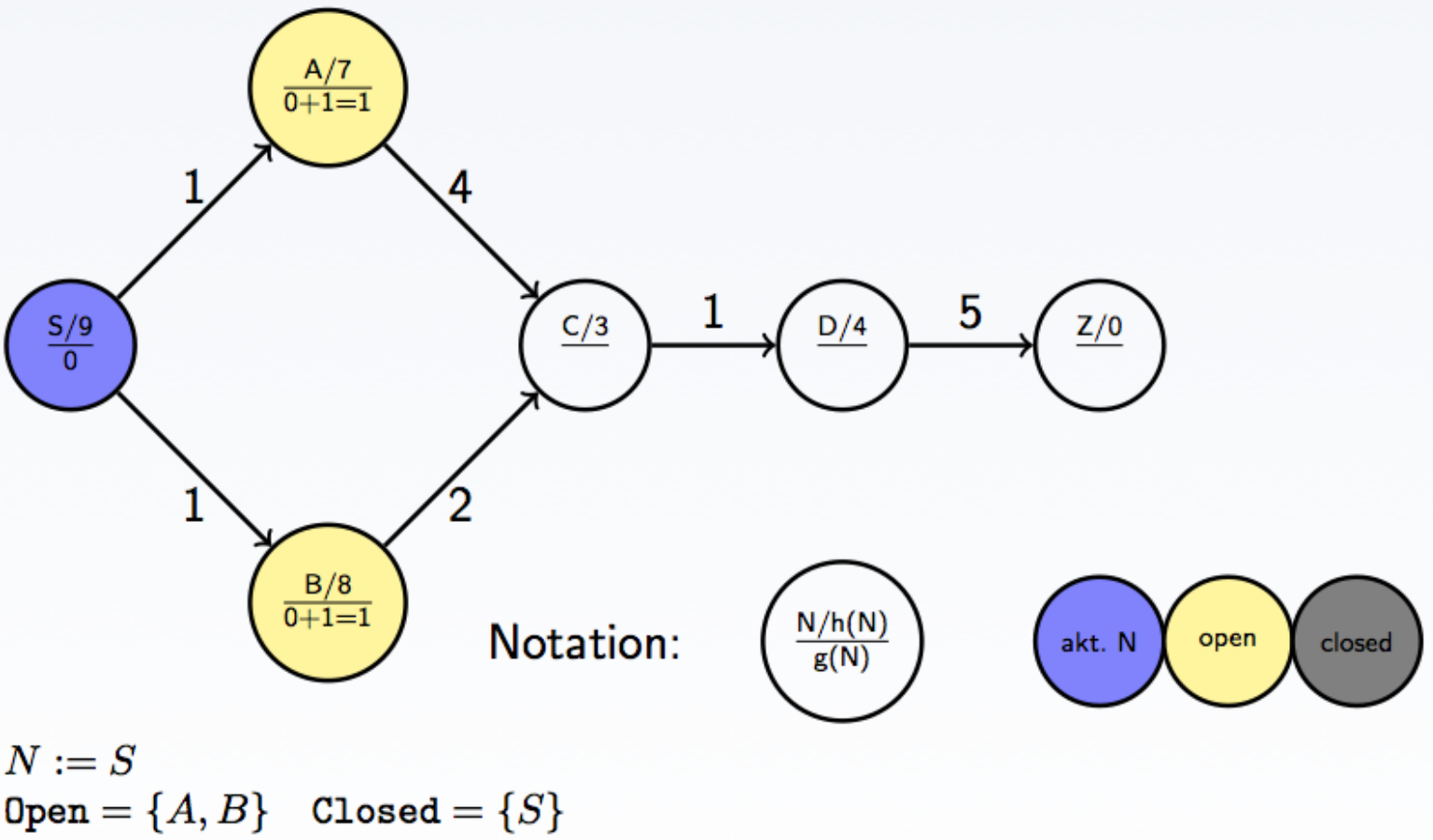
Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

Beispiel



$\texttt{Open} = \{S\} \quad \texttt{Closed} = \emptyset$

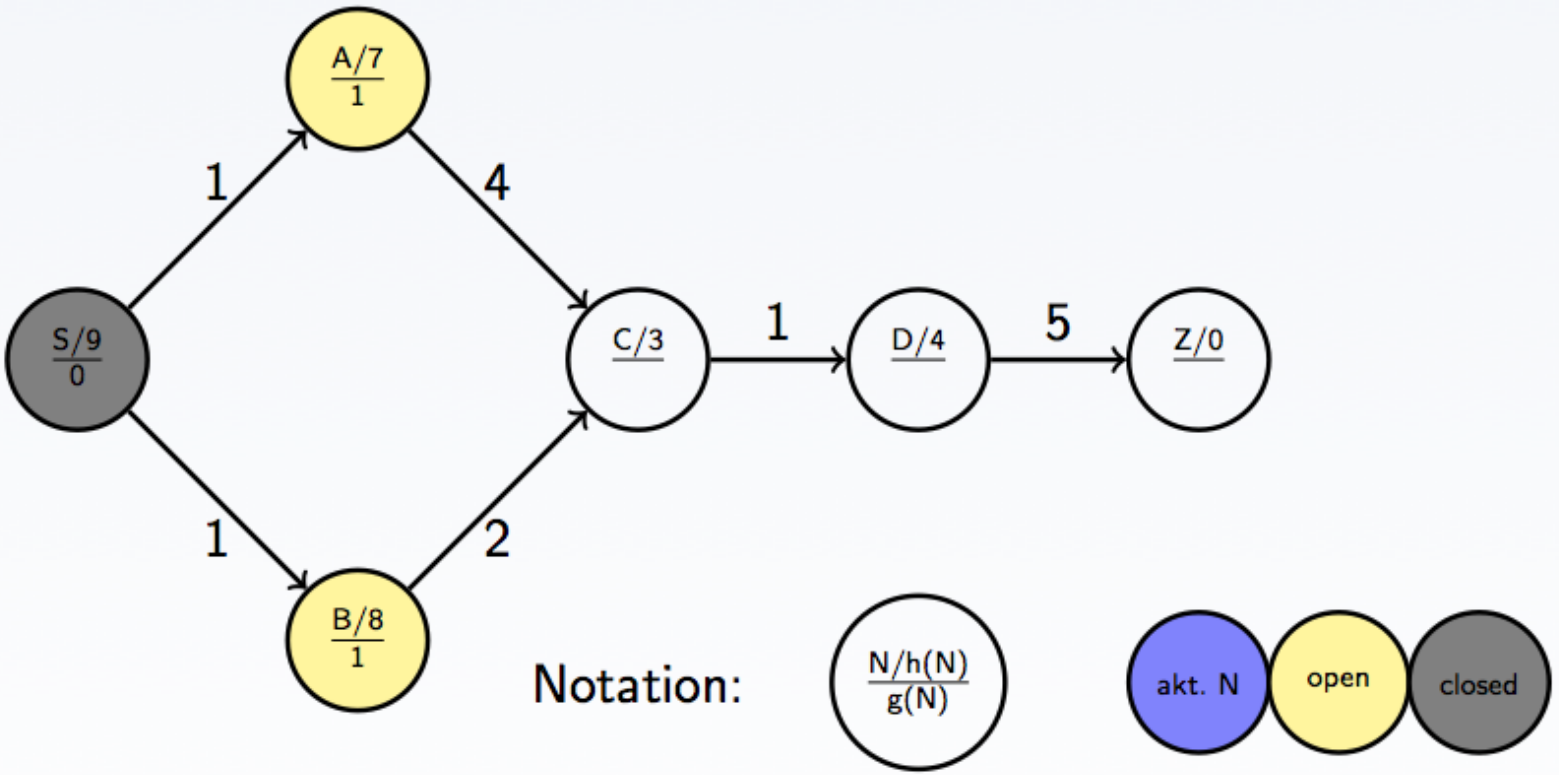
Beispiel



$N := S$

$\texttt{Open} = \{A, B\} \quad \texttt{Closed} = \{S\}$

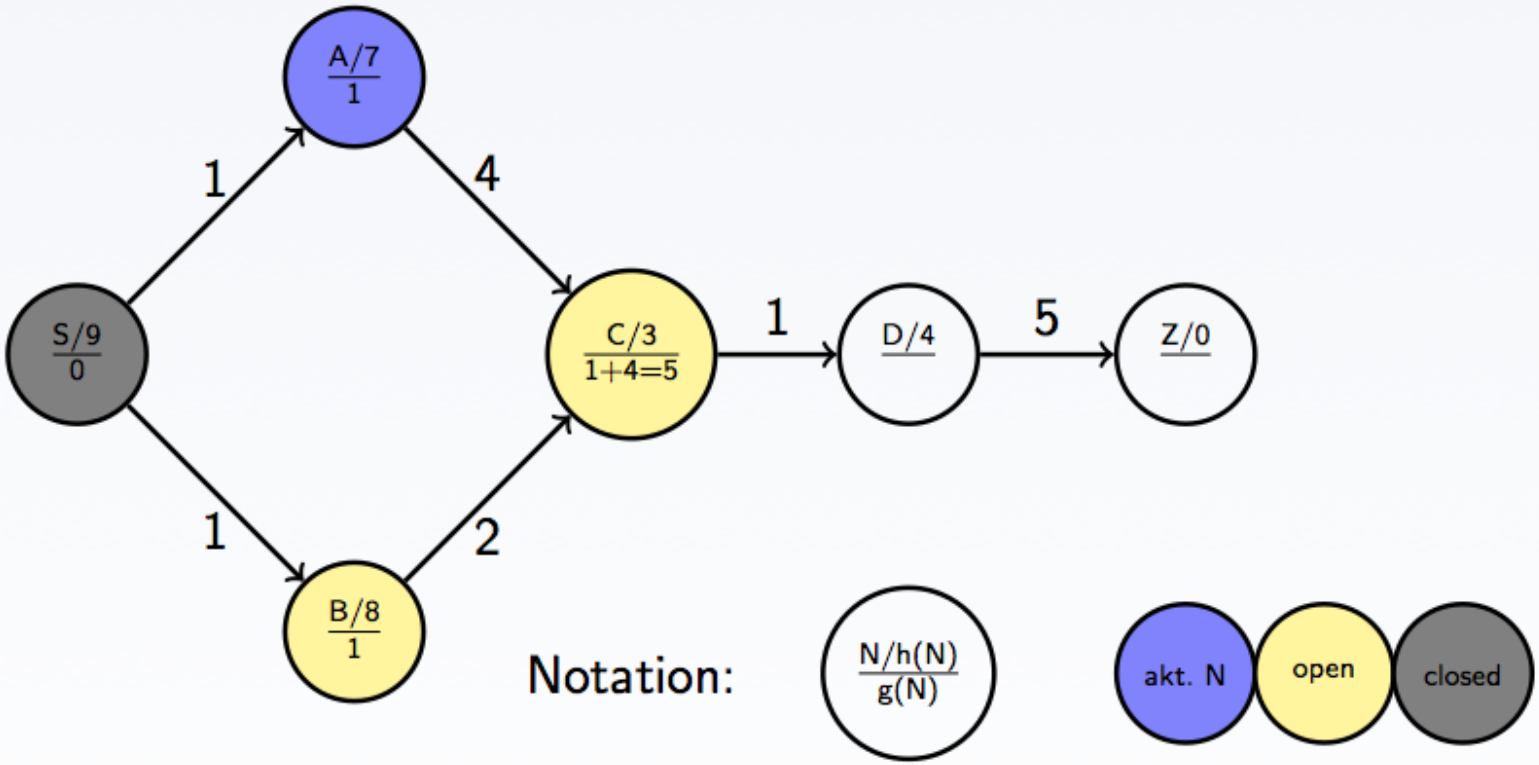
Beispiel



$N := S$

$\texttt{Open} = \{A, B\} \quad \texttt{Closed} = \{S\}$

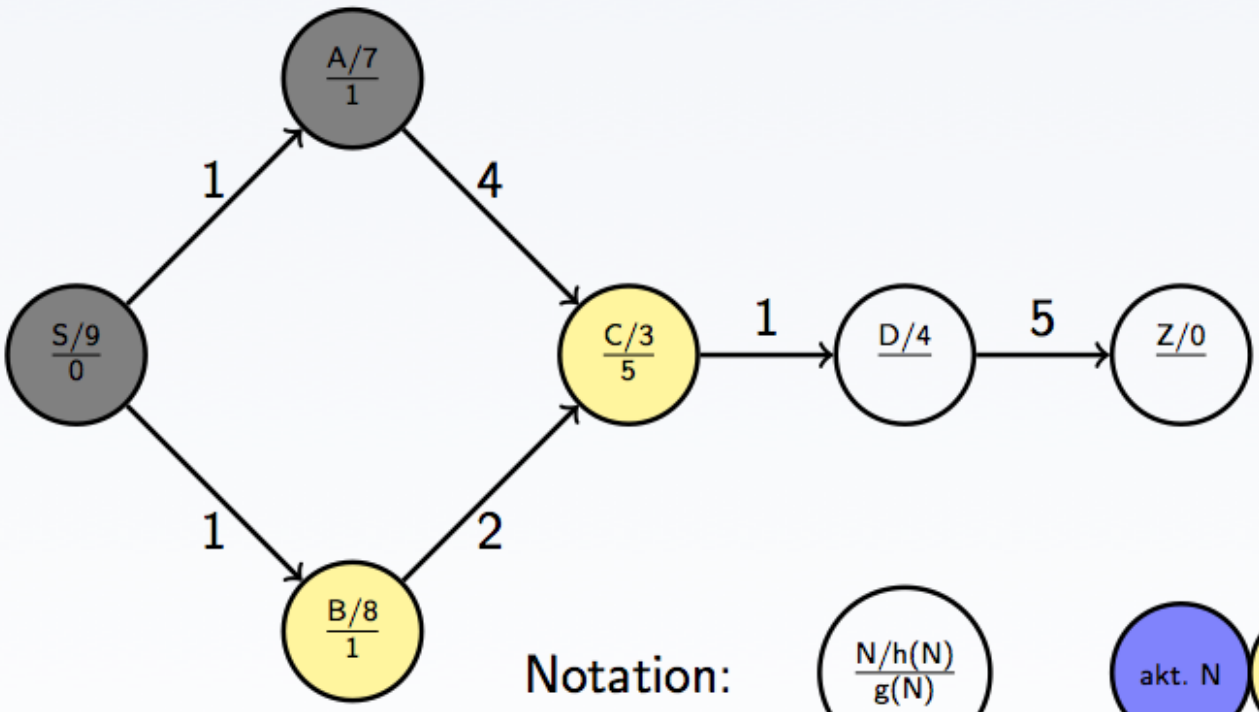
Beispiel



$f(A) = 1 + 7 = 8 \quad f(B) = 1 + 8 = 9 \quad N := A$

$\texttt{Open} = \{B, C\} \quad \texttt{Closed} = \{A, S\}$

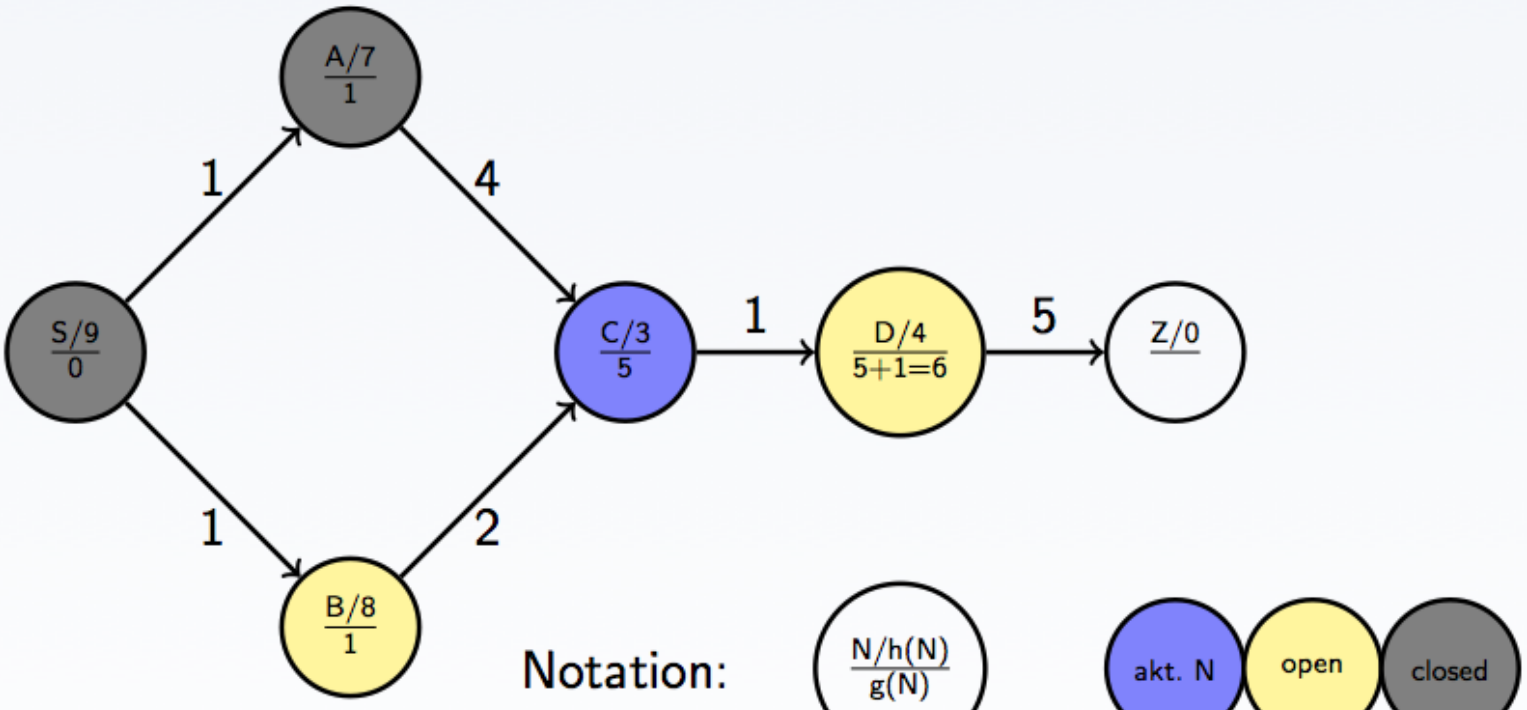
Beispiel



$f(A) = 1 + 7 = 8 \quad f(B) = 1 + 8 = 9 \quad N := A$

$\texttt{Open} = \{B, C\} \quad \texttt{Closed} = \{A, S\}$

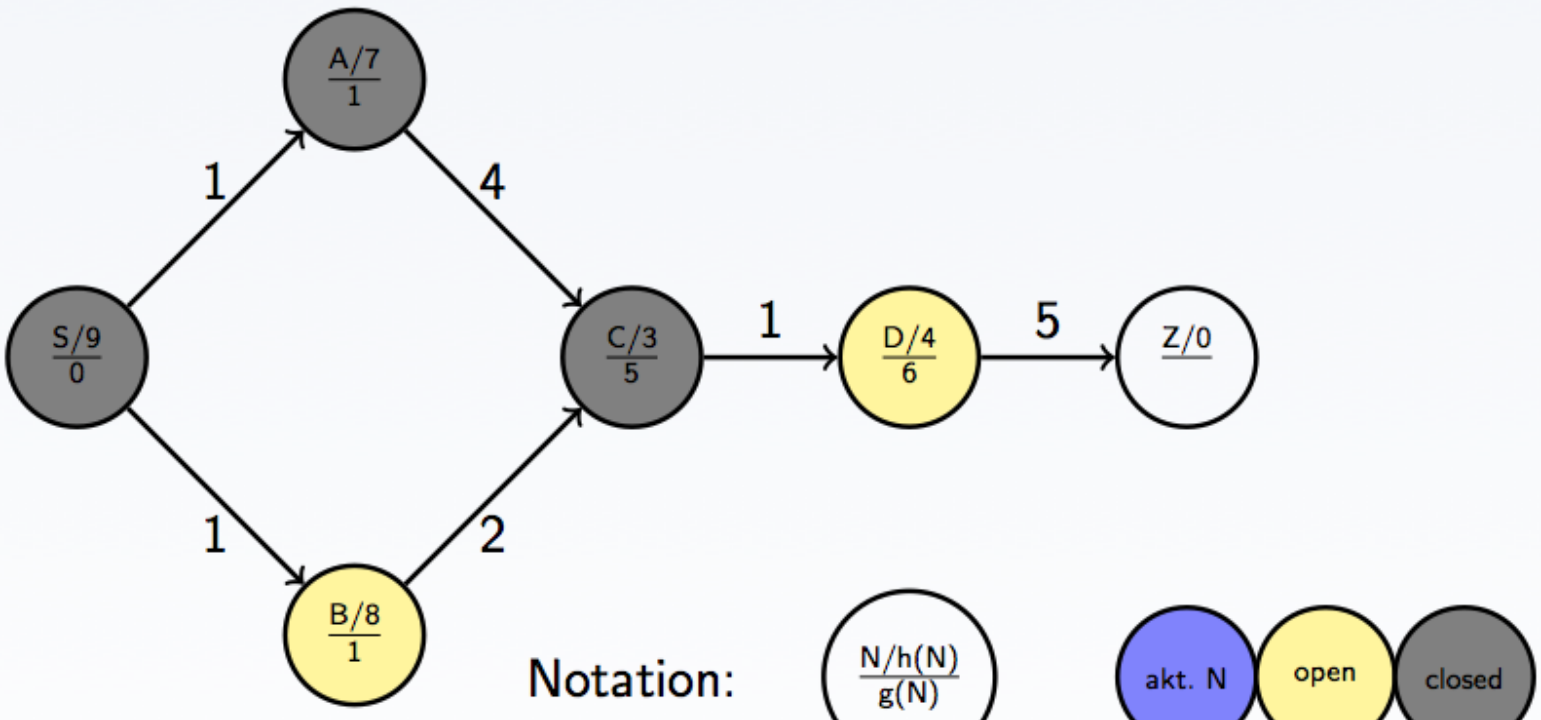
Beispiel



$f(B) = 1 + 8 = 9 \quad f(C) = 5 + 3 = 8 \quad N := C$

$\texttt{Open} = \{B, D\} \quad \texttt{Closed} = \{A, C, S\}$

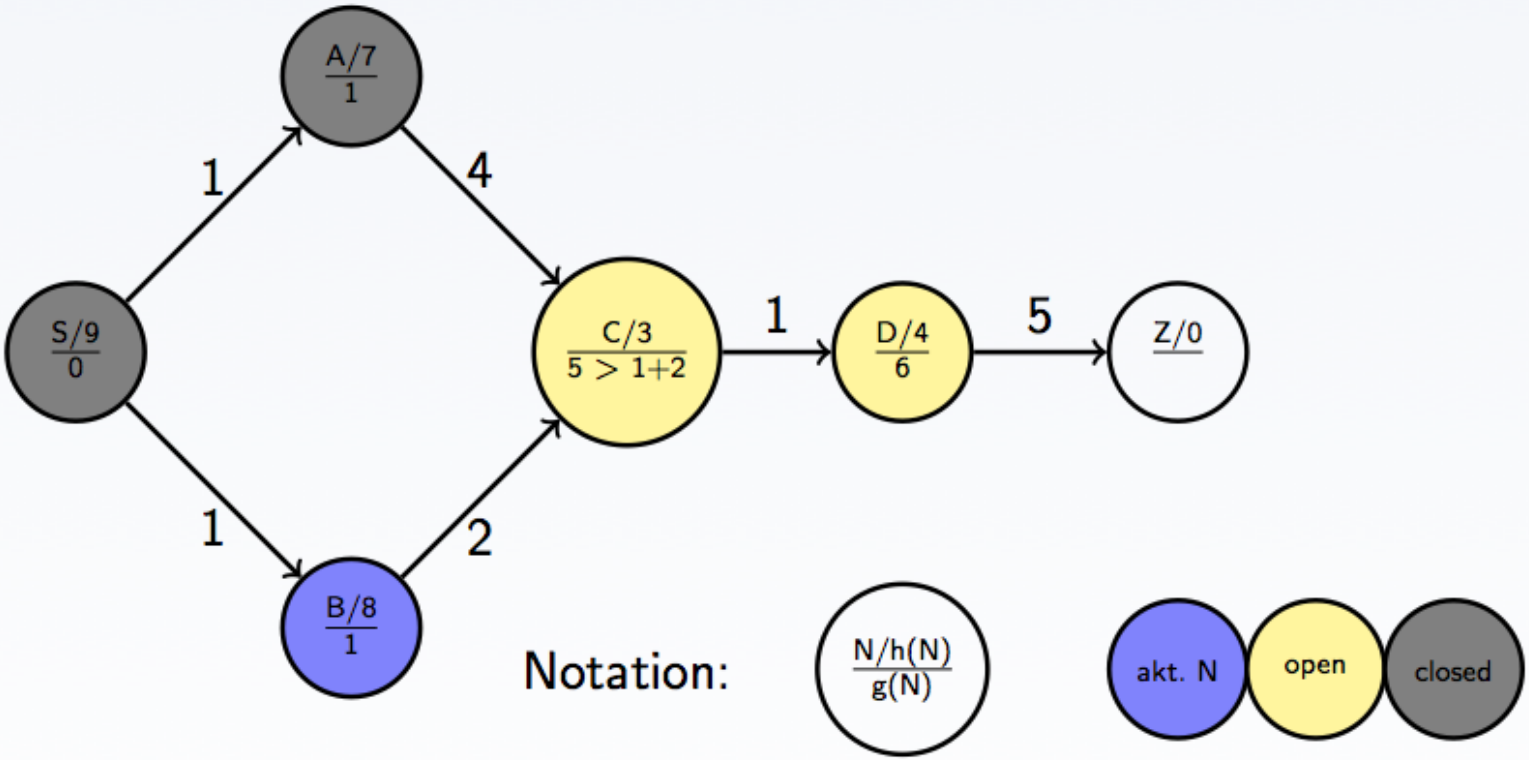
Beispiel



$f(B) = 1 + 8 = 9 \quad f(C) = 5 + 3 = 8 \quad N := C$

$\texttt{Open} = \{B, D\} \quad \texttt{Closed} = \{A, C, S\}$

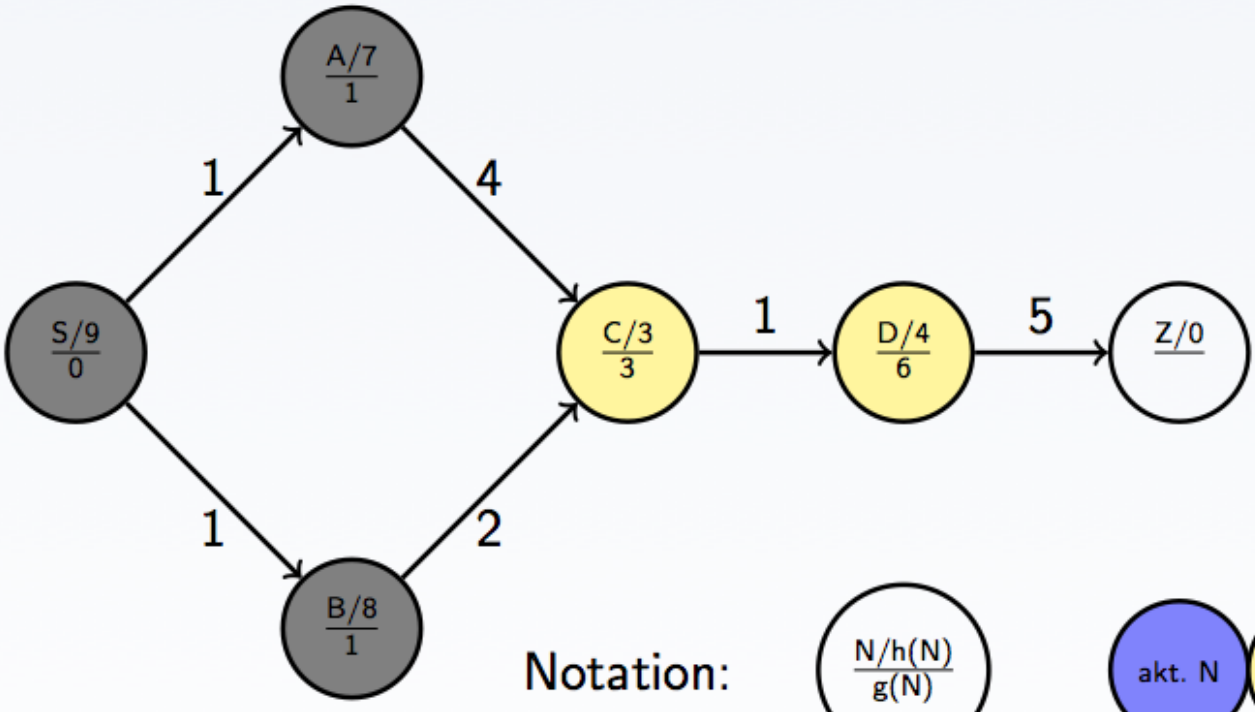
Beispiel



$f(B) = 1 + 8 = 9 \quad f(D) = 6 + 4 = 10 \quad N := B$

$\texttt{Open} = \{C, D\} \quad \texttt{Closed} = \{A, B, S\}$

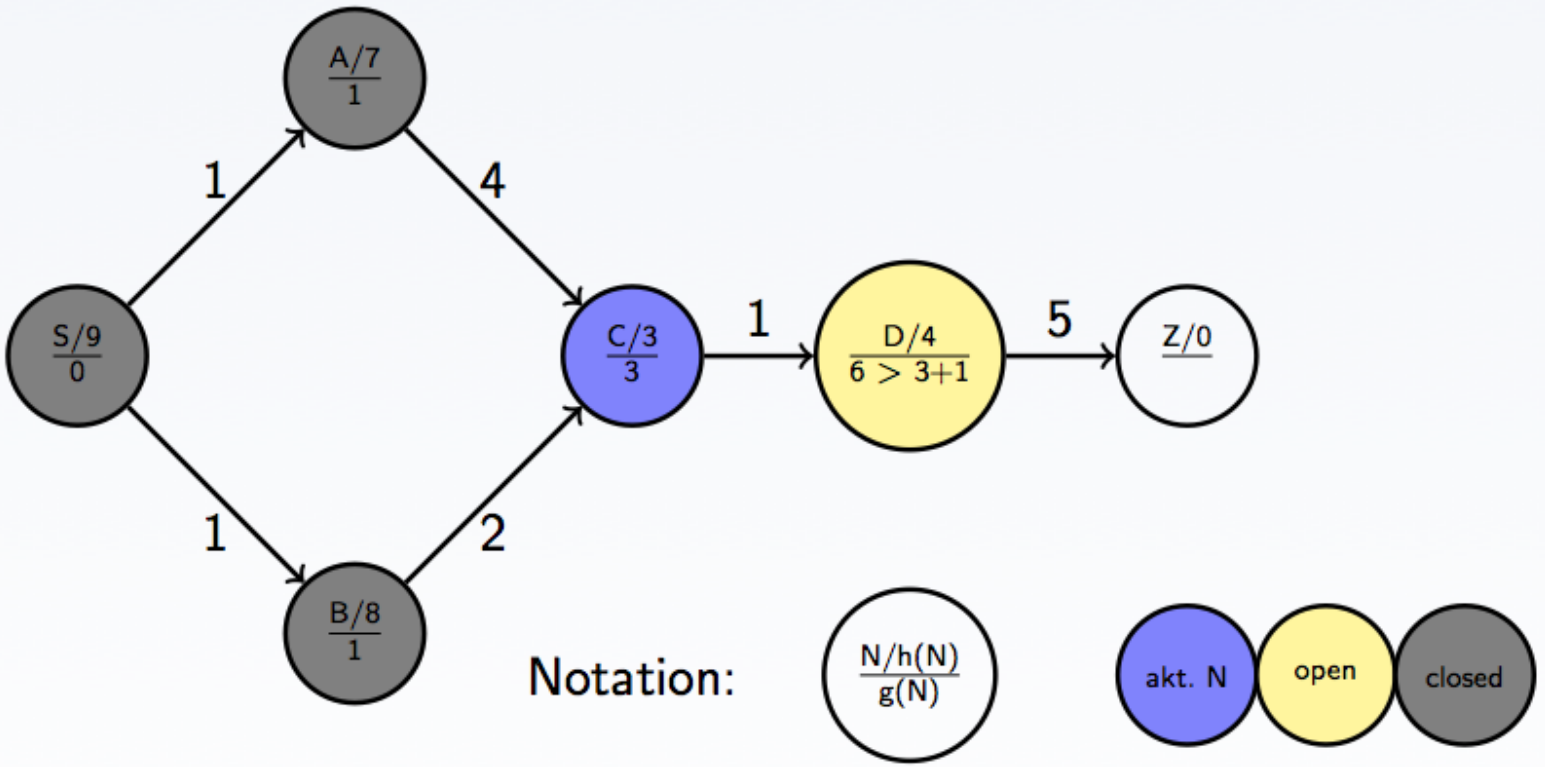
Beispiel



$f(B) = 1 + 8 = 9 \quad f(D) = 6 + 4 = 10 \quad N := B$

$\texttt{Open} = \{C, D\} \quad \texttt{Closed} = \{A, B, S\}$

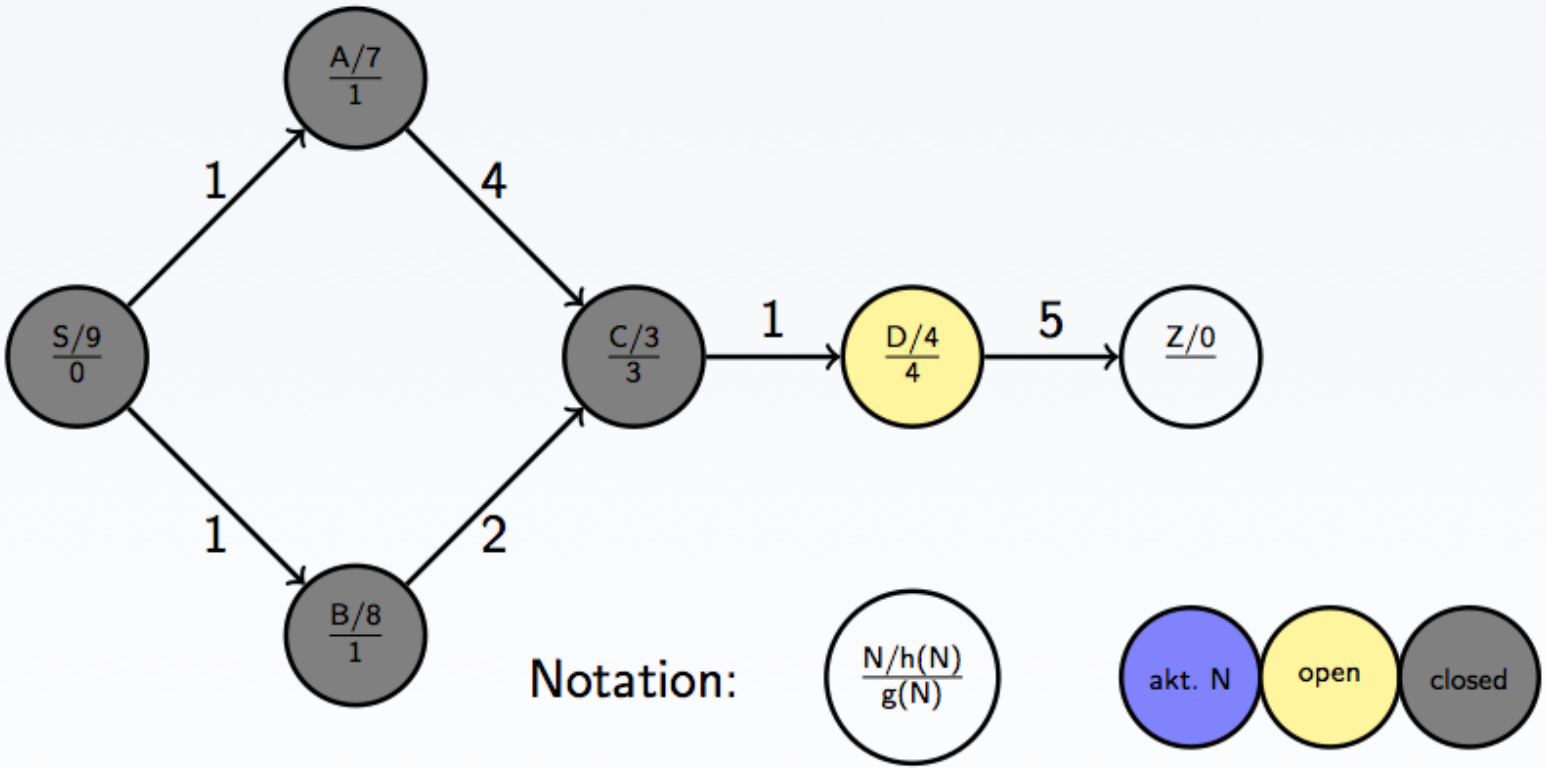
Beispiel



$f(C) = 3 + 3 = 6 \quad f(D) = 6 + 4 = 10 \quad N := C$

$\texttt{Open} = \{D\} \quad \texttt{Closed} = \{A, B, C, S\}$

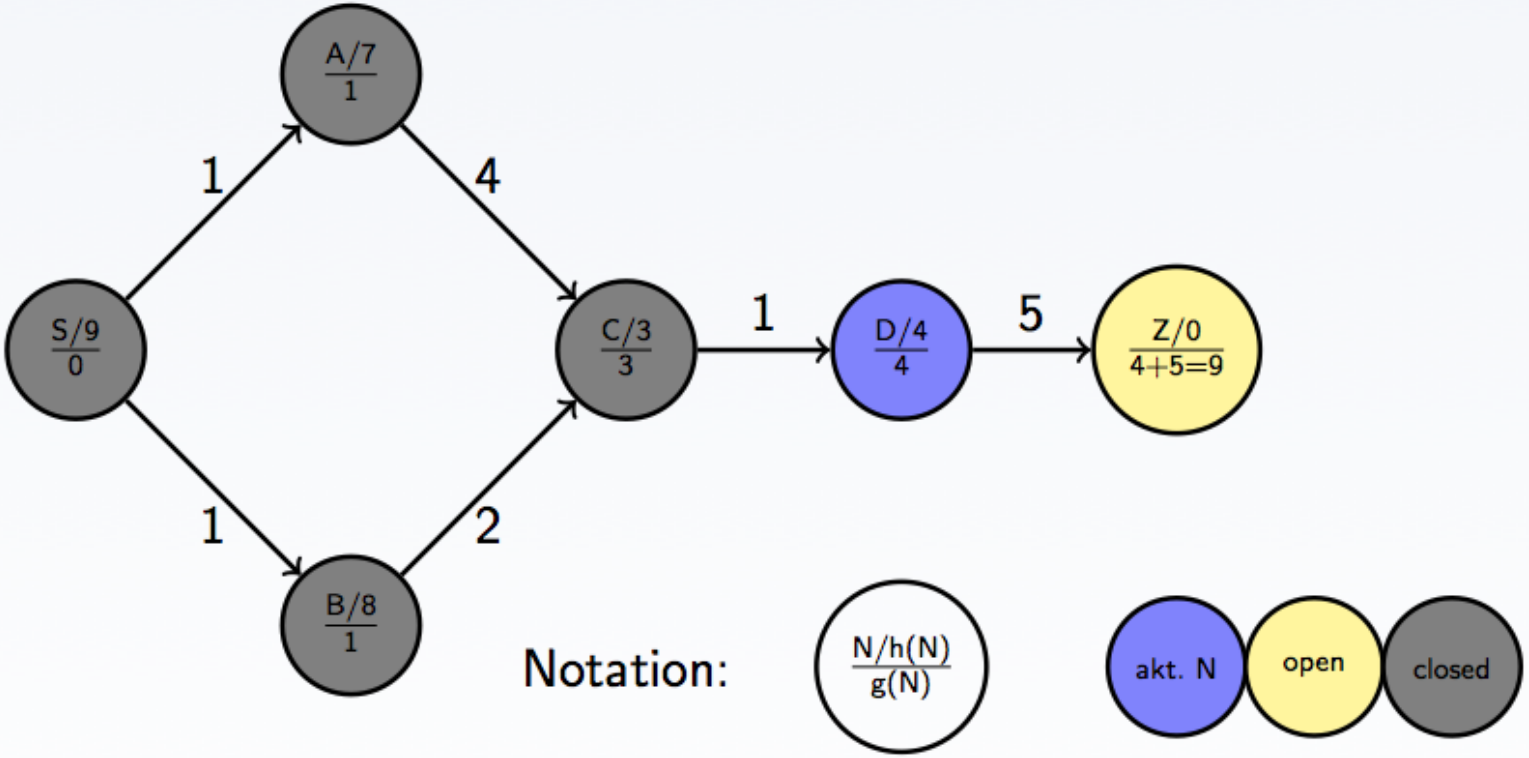
Beispiel



$f(C) = 3 + 3 = 6 \quad f(D) = 6 + 4 = 10 \quad N := C$

$\texttt{Open} = \{D\} \quad \texttt{Closed} = \{A, B, C, S\}$

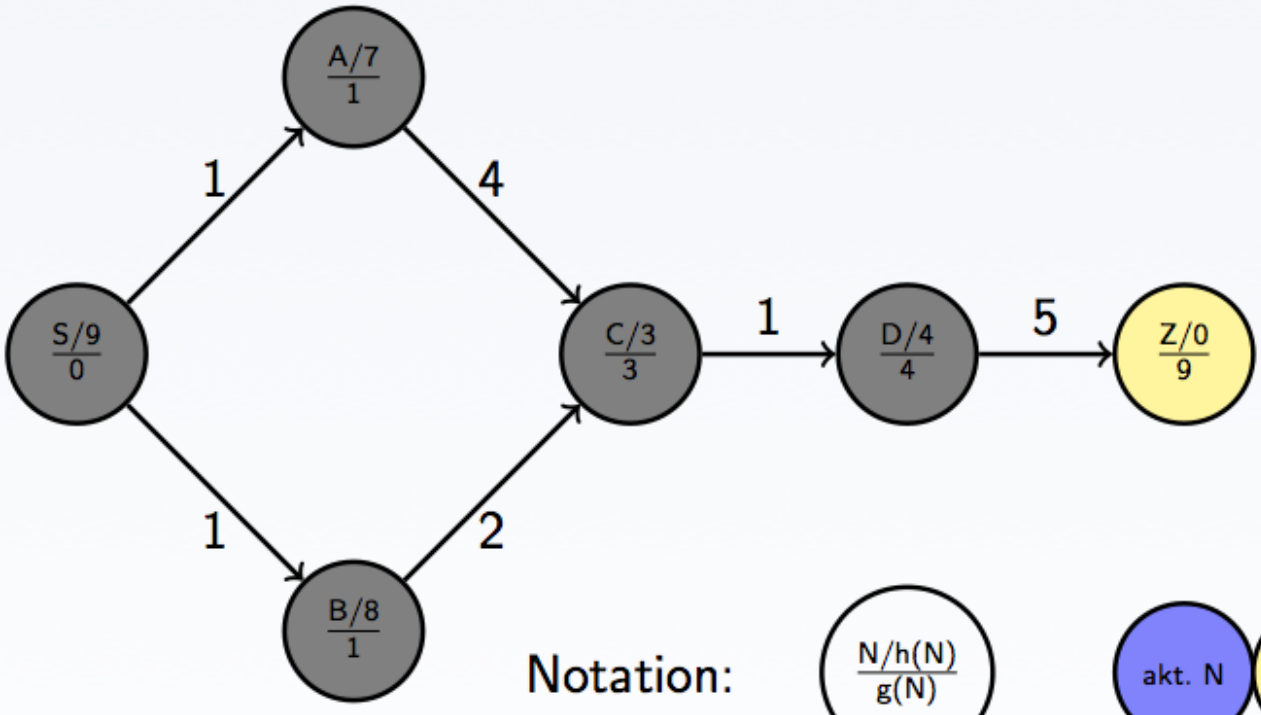
Beispiel



$f(D) = 4 + 4 = 8 \quad N := D$

$\texttt{Open} = \{Z\} \quad \texttt{Closed} = \{A, B, C, D, S\}$

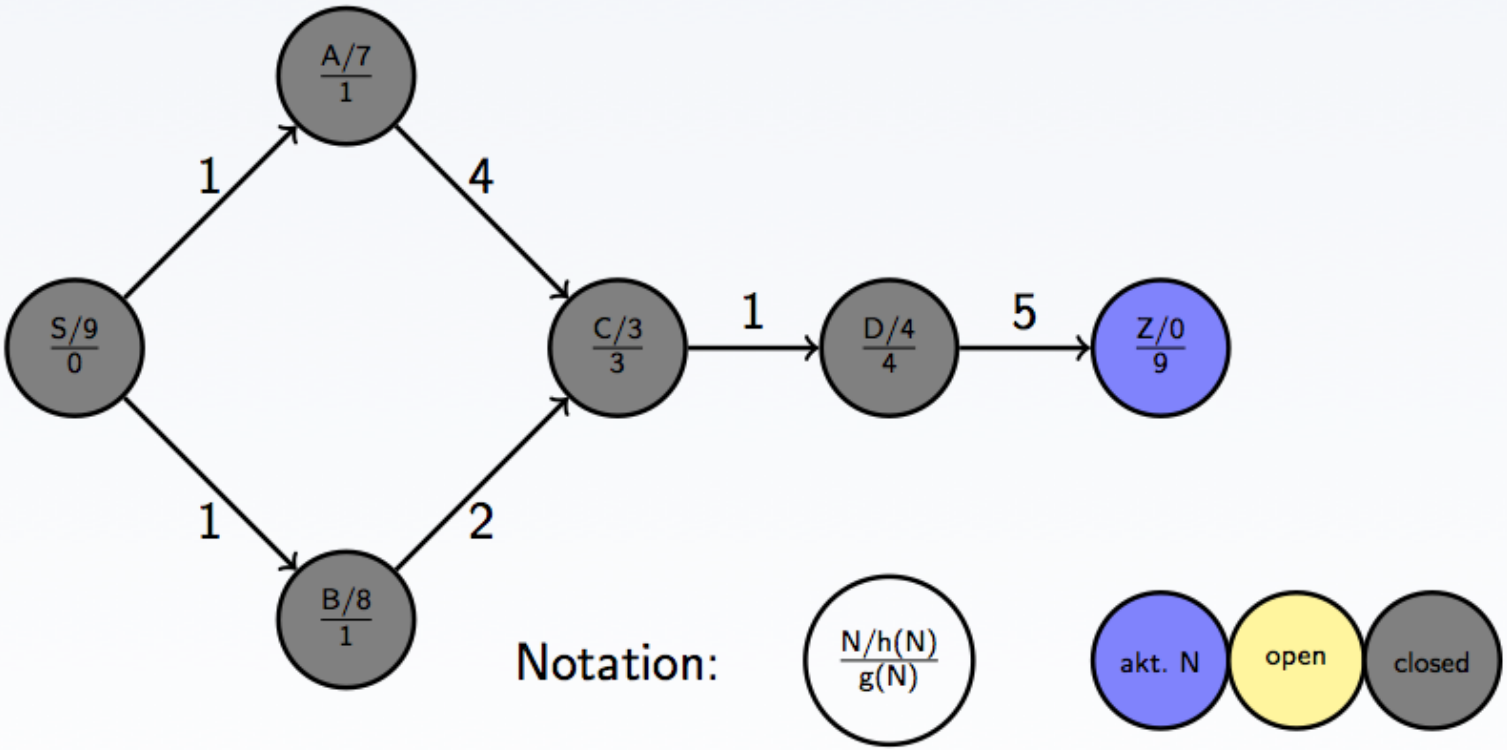
Beispiel



$f(D) = 4 + 4 = 8 \quad N := D$

$\texttt{Open} = \{Z\} \quad \texttt{Closed} = \{A, B, C, D, S\}$

Beispiel



$f(Z) = 9 + 0 = 9 \quad N := Z$

$\texttt{Open} = \{Z\} \quad \texttt{Closed} = \{A, B, C, D, S\}$ Zielknoten Z

Beispiel

- Beispiel zeigt, dass i.A. notwendig:
- Knoten aus `Closed` wieder in `Open` einfügen
- Beispiel extra so gewählt!
- Beachte: Auch Kanten werden mehrfach betrachtet
- Mehr Anforderungen an die Heuristik verhindern das!

Notationen für die Analyse

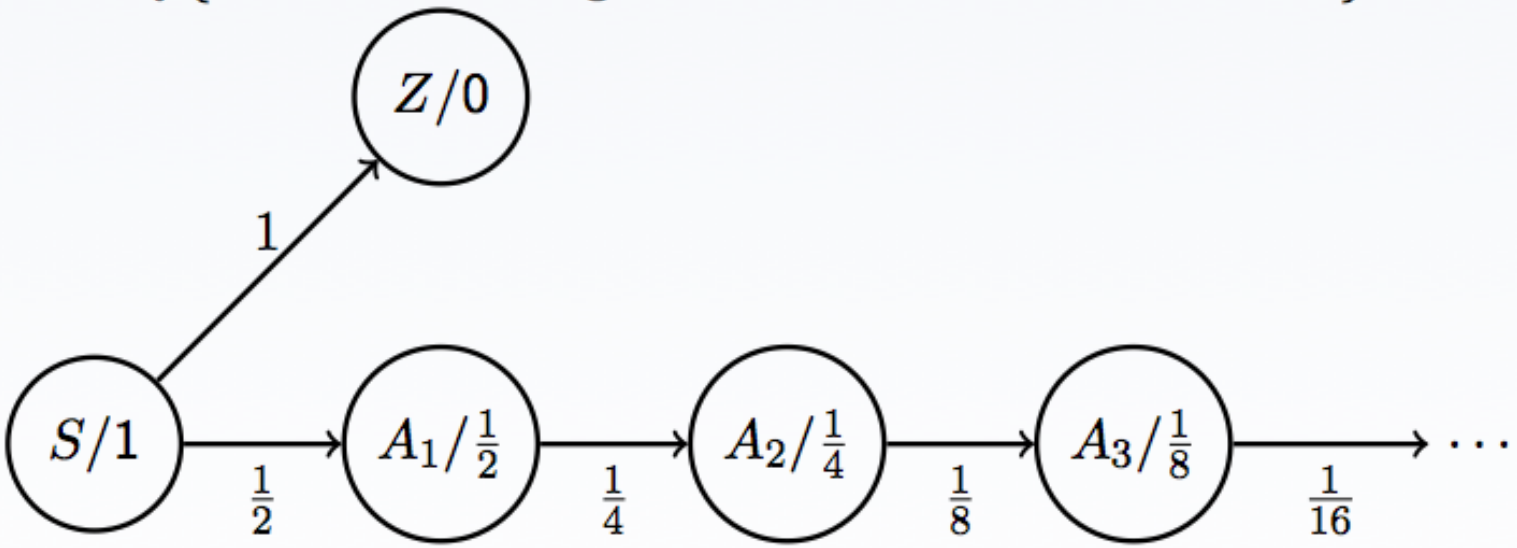
- $g^*(N, N')$ = Kosten des optimalen Weges von N nach N'
- $g^*(N)$ =Kosten des optimalen Weges vom Start bis zu N
- $c^*(N)$ =Kosten des optimalen Weges von N bis zum nächsten Zielknoten Z.
- $f^*(N) = g^*(N) + c^*(N)$ (Kosten des optimalen Weges durch N bis zu einem Ziel Z)

Voraussetzungen für den A^*-Algorithmus

1. es gibt nur endlich viele Knoten N mit $g^*(N) + h(N) \leq d$, wobei
   $d = \inf\{\text{Kosten aller Wege von } S \text{ zu einem Zielknoten } Z\}$.
2. Für jeden Knoten N gilt: $h(N) \leq c^*(N)$, d.h. die Schätzfunktion ist unterschätzend.
3. Für jeden Knoten N ist die Anzahl der Nachfolger endlich.
4. Alle Kanten kosten etwas: $c(N, N') > 0$ für alle N, N'.
5. Der Graph ist schlicht, d.h. zwischen zwei Knoten gibt es höchstens eine Kante

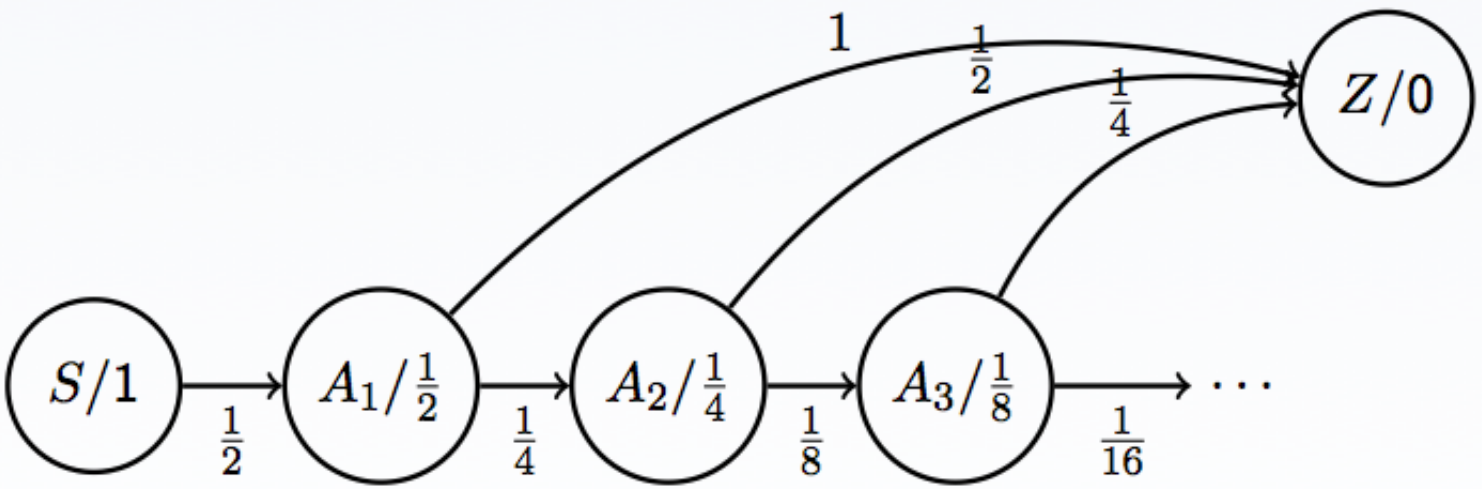
Bedingung 1 ist notwendig: Beispiele

Bedingung 1: es gibt nur endlich viele Knoten N mit
$g^*(N) + h(N) \leq d$, wobei
$d = inf\{$Kosten aller Wege von S zu einem Zielknoten $Z\}$.



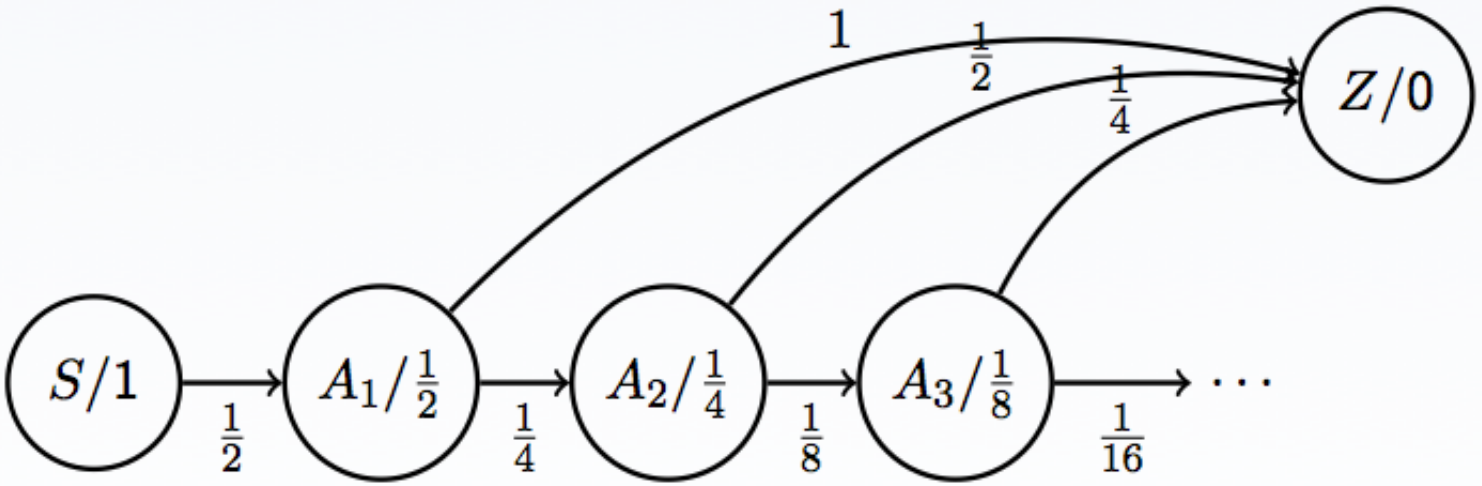
Bedingung 1 ist notwendig: Beispiele

Bedingung 1: es gibt nur endlich viele Knoten N mit $g^*(N) + h(N) \leq d$, wobei
$d = inf\{$Kosten aller Wege von S zu einem Zielknoten $Z\}$.
zum Infimum d muss es nicht notwendigerweise auch einen endlichen Weg im Suchgraphen geben:



Bedingung 1: hinreichende Bedingungen

Bedingung 1: es gibt nur endlich viele Knoten N mit $g^*(N) + h(N) \leq d$, wobei
$d = inf\{$Kosten aller Wege von S zu einem Zielknoten $Z\}$.
zum Infimum d muss es nicht notwendigerweise auch einen endlichen Weg im Suchgraphen geben:



Korrektheit und Vollständigkeit der A^*-Suche

Wenn Voraussetzungen für den A^*-Algorithmus erfüllt, dann existiert zum Infimum d stets ein endlicher Weg mit Kosten
Notation:

$$infWeg(N) := \inf\{\text{Kosten aller Wege von } N \text{zu einem Ziel}\}$$

Satz

Es existiere ein Weg vom Start S bis zu einem Zielknoten. Sei $d = infWeg(S)$. Die Voraussetzungen für den A^*-Algorithmus seien erfüllt. Dann existiert ein optimaler Weg von S zum Ziel mit Kosten d.

Korrektheit und Vollständigkeit der A^*-Suche

Ein optimaler Knoten ist stets in `Open`:

Lemma

Die Voraussetzungen zum A^*-Algorithmus seien erfüllt. Es existiere ein optimaler Weg $S = K_0 \to K_1 \to K_2 \to \cdots \to K_n = Z$ vom Startknoten S bis zu einem Zielknoten Z. Dann ist während der Ausführung des A^*-Algorithmus stets ein Knoten K_i in `Open`, markiert mit $g(K_i) = g^*(K_i)$, d.h. mit einem optimalen Weg von S zu K_i.

Korrektheit und Vollständigkeit der A^*-Suche

Expandierte Zielknoten sind optimal:

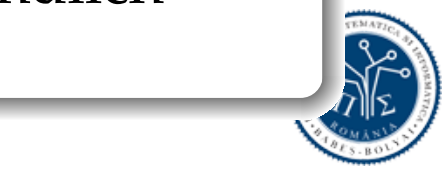
Lemma

Wenn die Voraussetzung für den A^*-Algorithmus erfüllt sind, gilt: Wenn A^* einen Zielknoten expandiert, dann ist dieser optimal.

Ein Zielknoten wird nach endlicher Zeit expandiert:

Lemma

Die Voraussetzungen zum A^*-Algorithmus seien erfüllt. Wenn ein Weg vom Start zum Zielknoten existiert gilt: Der A^*-Algorithmus expandiert einen Zielknoten nach endlich vielen Schritten.



Korrektheit und Vollständigkeit der A^*-Suche

Zusammenfassend ergibt sich:

Theorem

Es existiere ein Weg vom Start bis zu einem Zielknoten. Die Voraussetzungen zum A^*-Algorithmus seien erfüllt. Dann findet der A^*-Algorithmus einen optimalen Weg zu einem Zielknoten.

Spezialfälle

- Wenn $h(N) = 0$ für alle Knoten N, dann ist A^*-Algorithmus dasselbe wie die sogenannte Gleiche-Kosten-Suche
- Wenn $c(N_1, N_2) = k$ für alle Knoten N_1, N_2 und $h(N) = 0$ für alle Knoten N, dann ist A^*-Algorithmus gerade die Breitensuche.

Variante: A^{*o}-Algorithmus

A^{*o} -Algorithmus

- findet alle optimalen Wege
- Abänderung am A^*-Algorithmus:
  sobald erster Zielknoten mit Wert d expandiert wurde:
  - Füge in `Open` nur noch Knoten mit $g(N) + h(N) \leq d$ ein
  - Andere Knoten kommen in `Closed`
- Stoppe erst, wenn `Open` leer ist

Theorem

Wenn die Voraussetzungen für den A^*-Algorithmus gelten, dann findet der Algorithmus A^{*o} alle optimalen Wege von S zum Ziel.