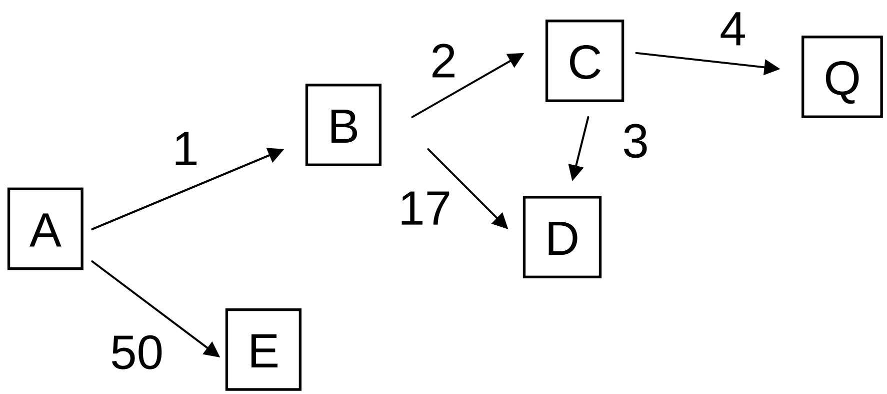
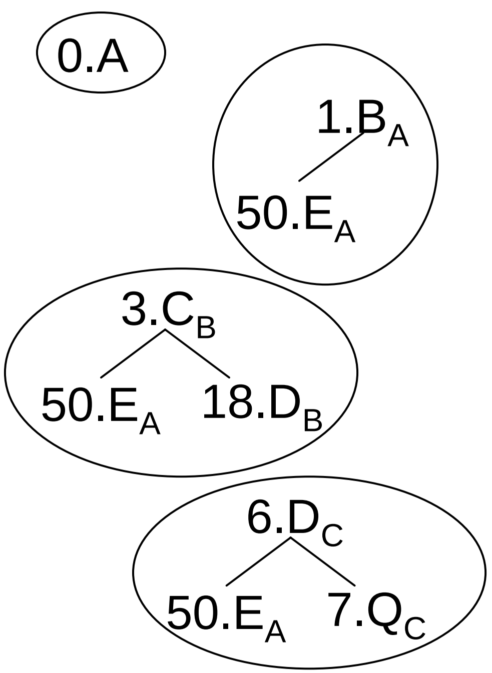
eppstein k-best

knight

Dijkstra



Push A
Pop A & go through A's arcs
  Push B.1 (from A)
  Push E.50 (from A)
Pop B & go through B's arcs
  Push C.3 (from B)
  Push D.18 (from B)
Pop C & go through C's arcs
  Push Q.7 (from C)
  Adjust D.6 (from C)



Analysis:

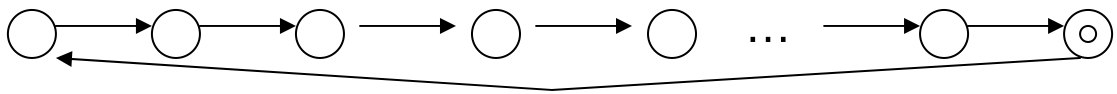
total pops = n
total pushes = m
maxheapsize = n

cost of pop = log n
cost of push = 1

O(m + n log n)

K-best

- Eppstein complexity is O(m + n log n + k log k)
- Great to keep n and k separate – both might be 100,000!
- But consider:



length of 1-best path: n

length of 2-best path: 2n

…

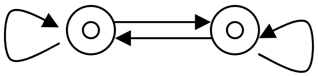
length of k-best path: kn

total of all these lengths = n(1+…+k) = O(n k^2)

- How can Eppstein do better? Well, printing paths is your problem.
- Eppstein computes "implicit representation of k best paths in O(m + n log n + k log k) time … from which we can read off the kth best path in time proportional to the number of edges in that path… and we can read off the length of that path in constant time."
- *k-best path lengths* problem solvable in O(m + n log n + k log k)

K-best

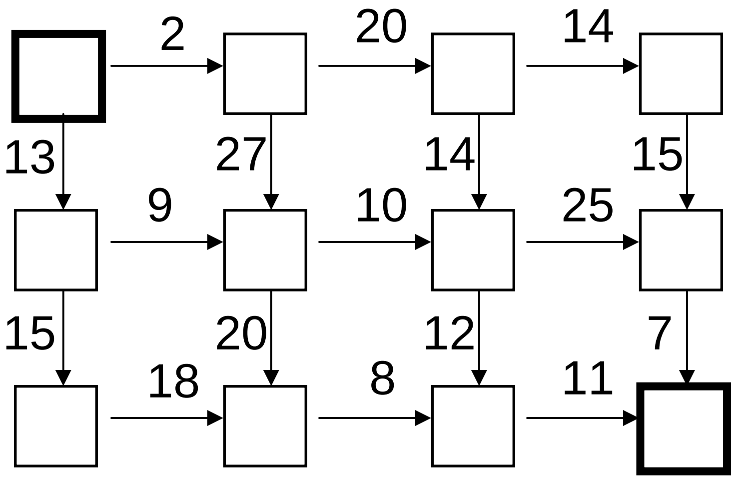
- Normal case for speech, MT, and NLG: k-best entries have roughly the same lengths
- Or even:



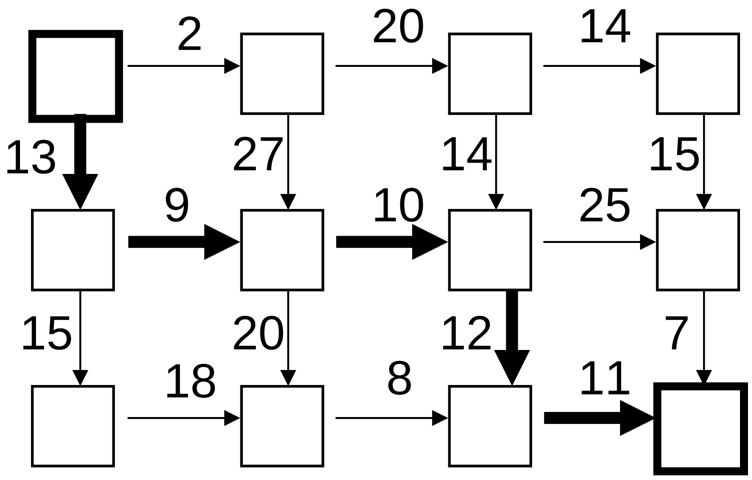
length of 1-best path: 0
length of 2-best path: 1
length of 3-best path: 1
length of 4-best path: 2
length of 5-best path: 2
length of 6-best path: 2
length of 7-best path: 2
length of 8-best path: 3
length of 9-best path: 3

- Summed lengths of k-best paths = O(k log k), not k^2 [homework]
- As k goes from 1m to 1b, lengths only increase from 20 to 30

Eppstein



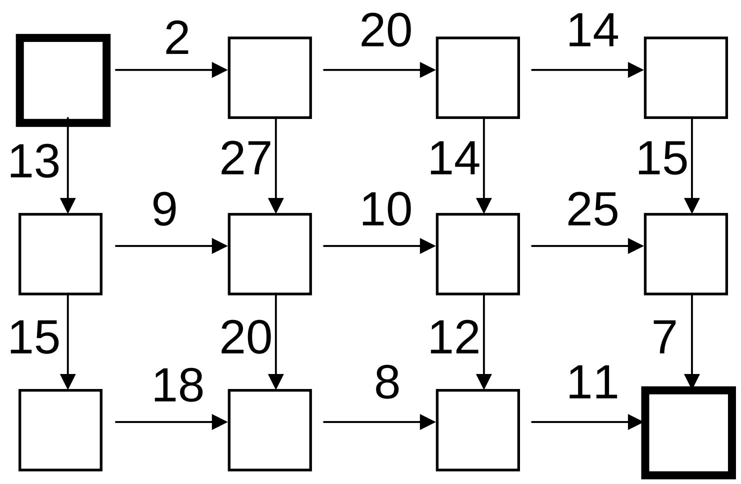
Eppstein

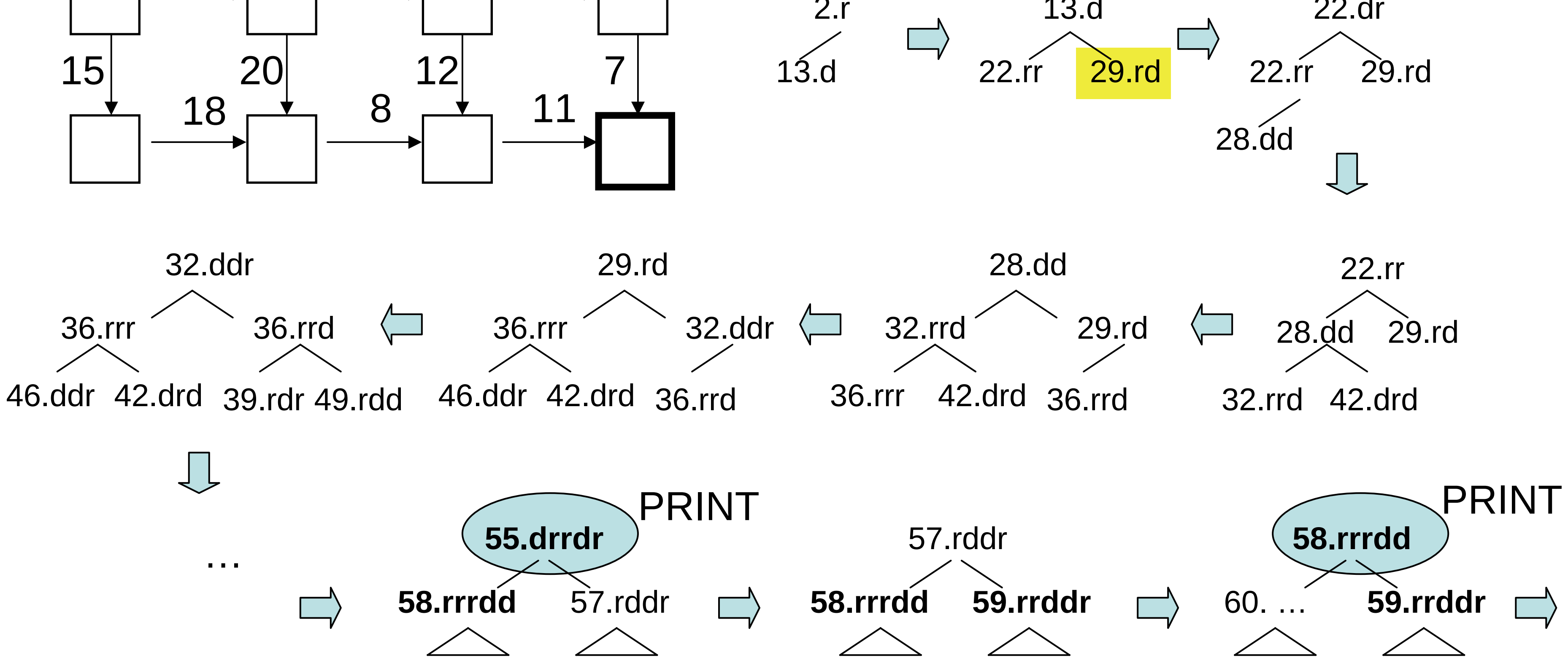


Best path = 55

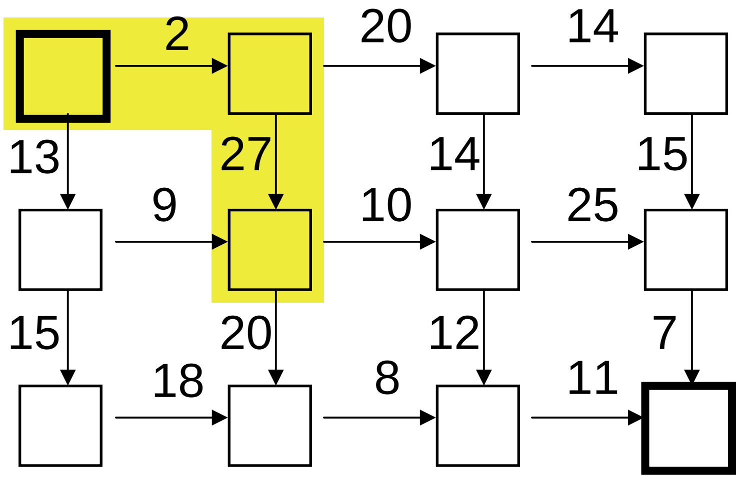
First Try: Breadth-First Search

Maintain a heap of <u>partial</u> paths.
Keep popping and pushing.
When a <u>complete path</u> is popped, print it.
Vast amount of work even before the 1-best.

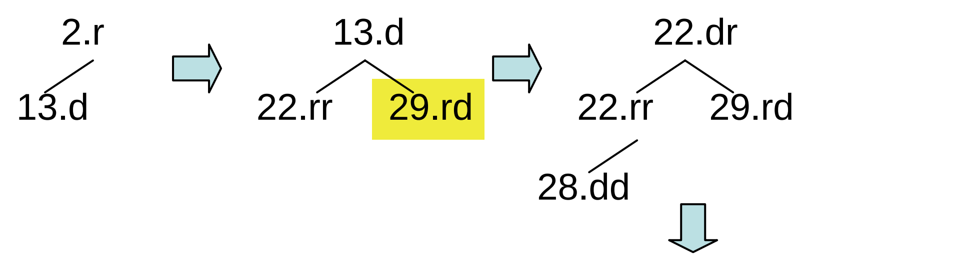


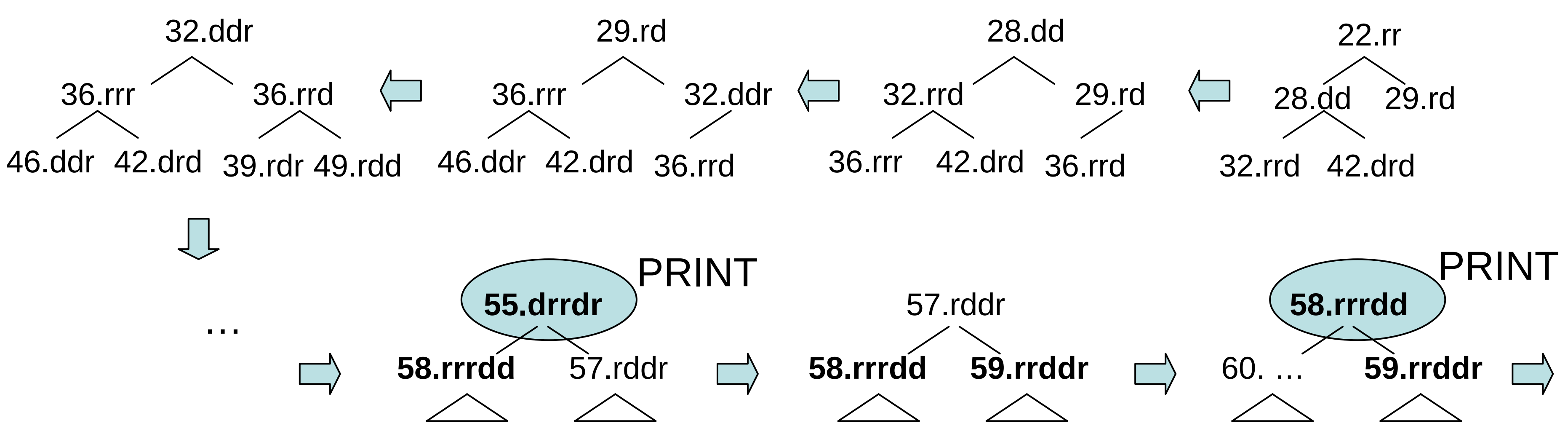


First Try: Breadth-First Search

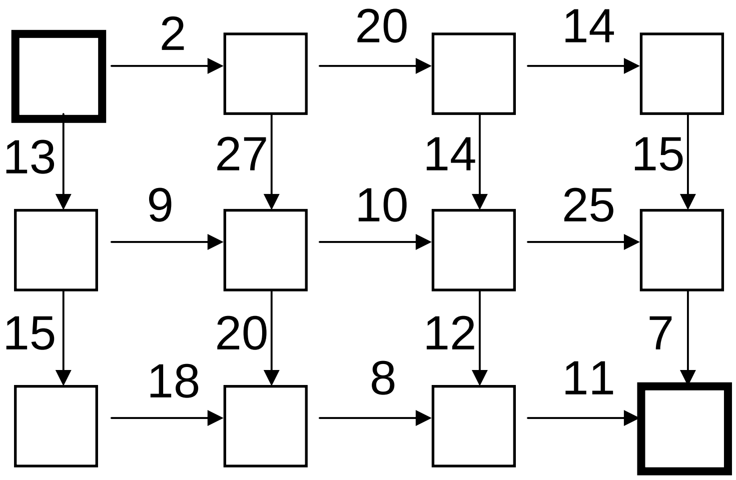


Maintain a heap of partial paths.
Keep popping and pushing.
When a complete path is popped, print it.
Vast amount of work even before the 1-best.





First Try: Breadth-First Search



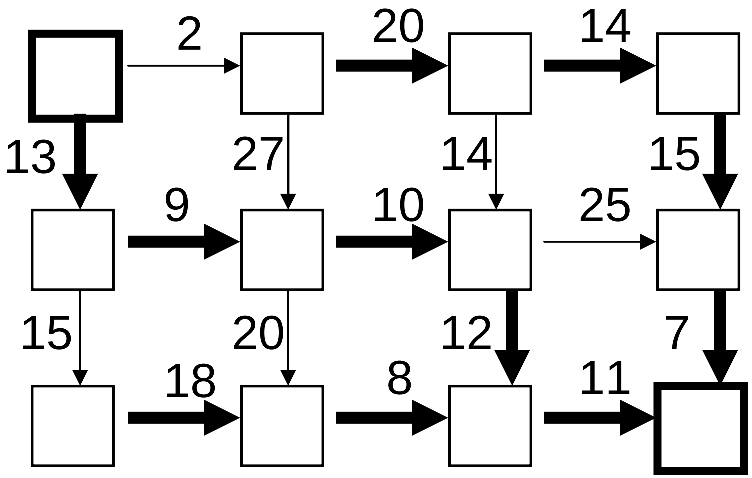
Maintain a heap of partial paths.
Keep popping and pushing.
When a complete path is popped, print it.
Vast amount of work even before the 1-best.

Core Eppstein idea:

Make a heap where all elements are complete paths.

Every pop gives you the next longest complete path.

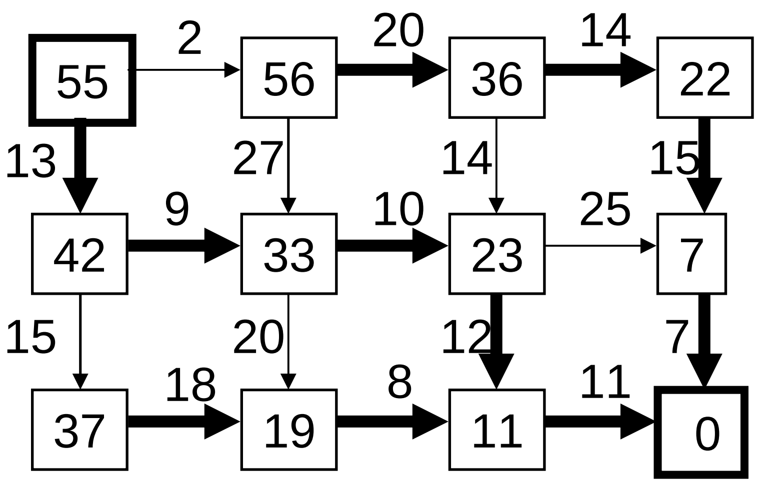
Eppstein



Best path = 55

Every node has its own best edge

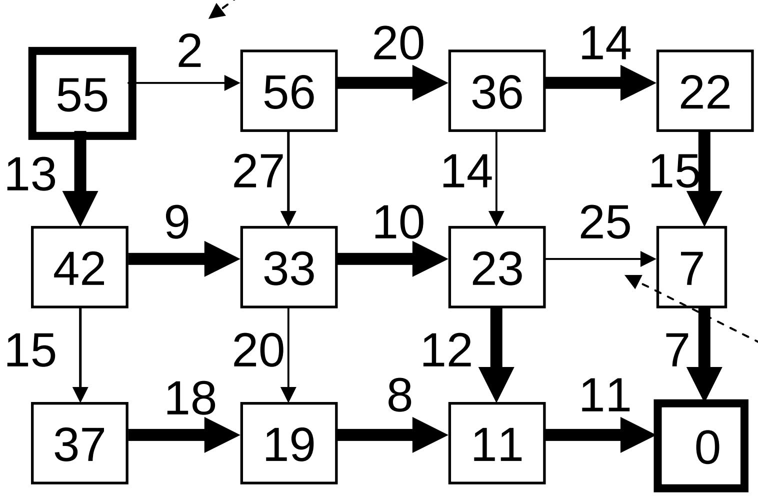
Eppstein



Best path = 55
Every node has its own best edge
Every node has its best cost to goal

Eppstein

What if we took this edge instead of 13? We'd wind up with a worse path, worse by 2 + (56-55) = 3.

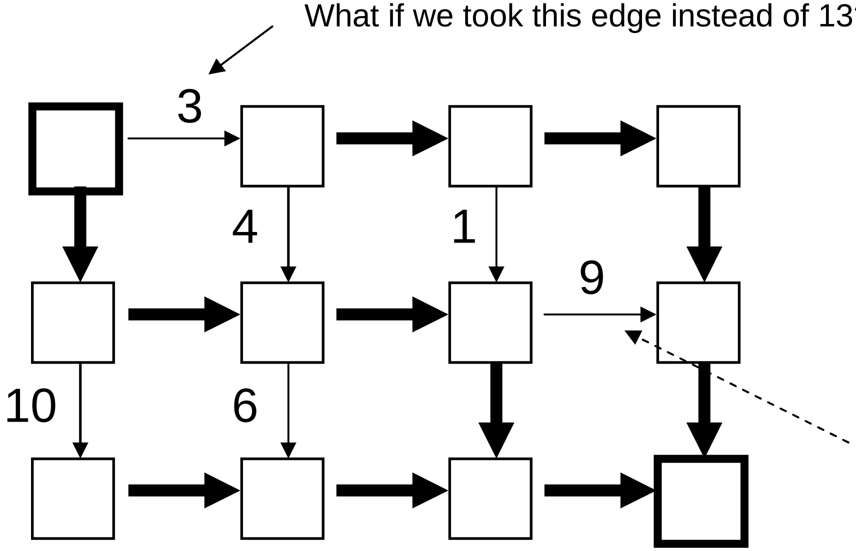


What if we took this edge instead of 12?
Worse by 25 + (7-23) = 9.

Best path = 55
Every node has its own best edge
Every node has its best cost to goal

Eppstein

What if we took this edge instead of 13? We'd wind up with a worse path, worse by 2 + (56-55) = 3.



SIDETRACK EDGES

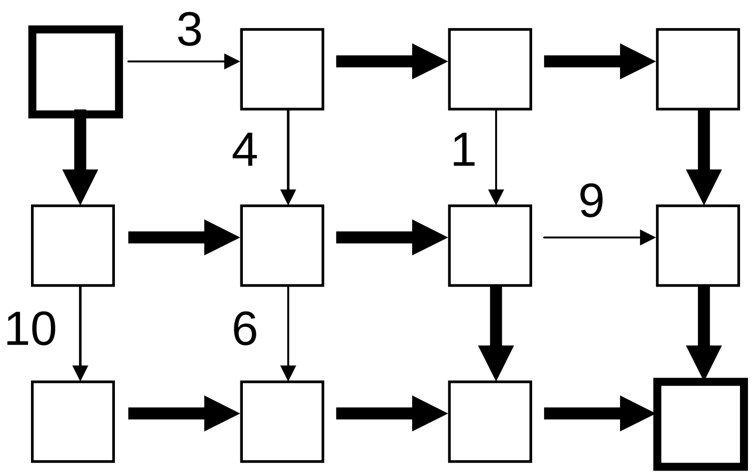
What if we took this edge instead of 12?
Worse by 25 + (7-23) = 9.

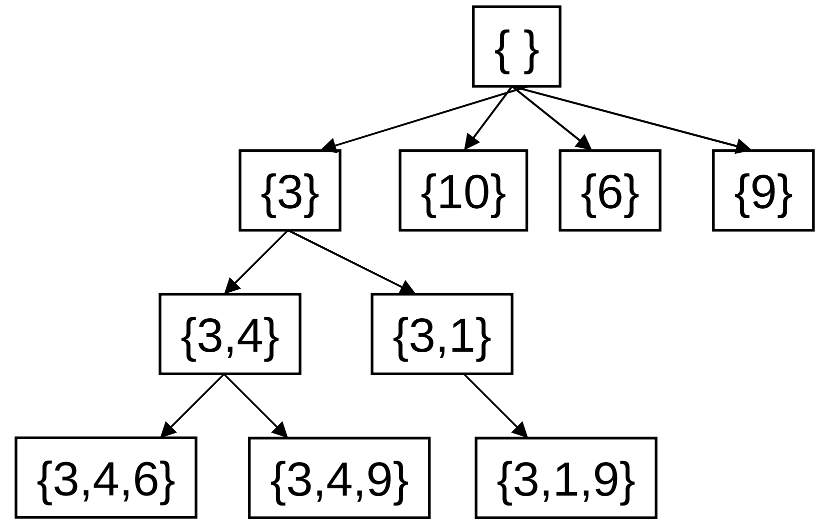
Best path = 55
Every node has its own best edge
Every node has its best cost to goal

Every path can be represented as a sequence of sidetrack edges.

So { }, {3}, {9}, {3,1}, {3, 4, 9}, etc. all correspond to paths in original graph.

Eppstein





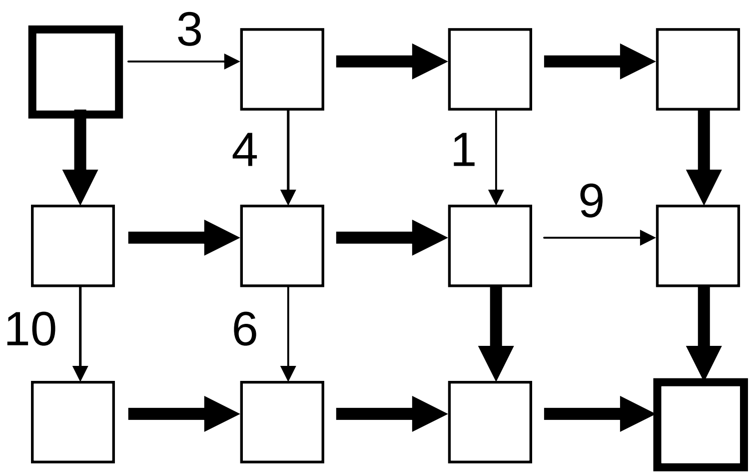
Tree of sidetrack sequences
(Yes, these are all 10 paths!)
Costs O(m + n log n) to set this up.

Best path = 55
Every node has its own best edge
Every node has its best cost to goal

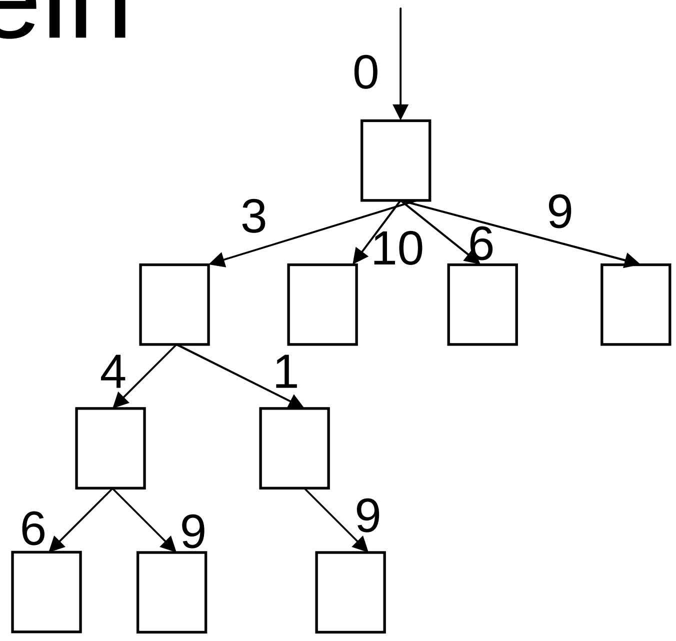
Every path can be represented as a sequence of sidetrack edges.

So { }, {3}, {9}, {3,1}, {3, 4, 9}, etc. all correspond to paths in original graph.

Eppstein



Best path = 55

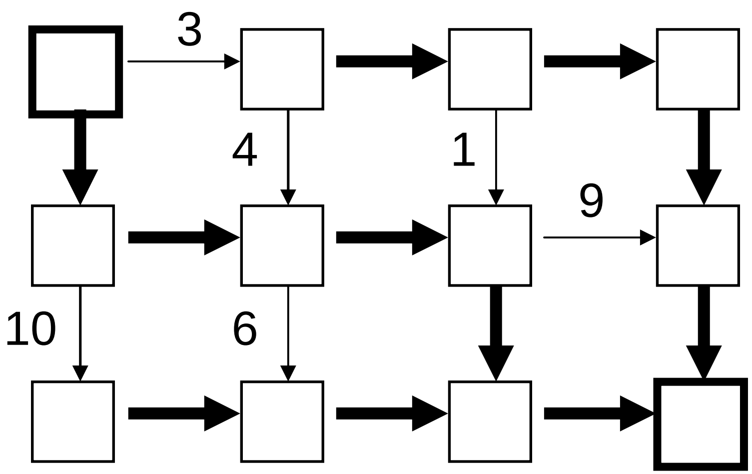


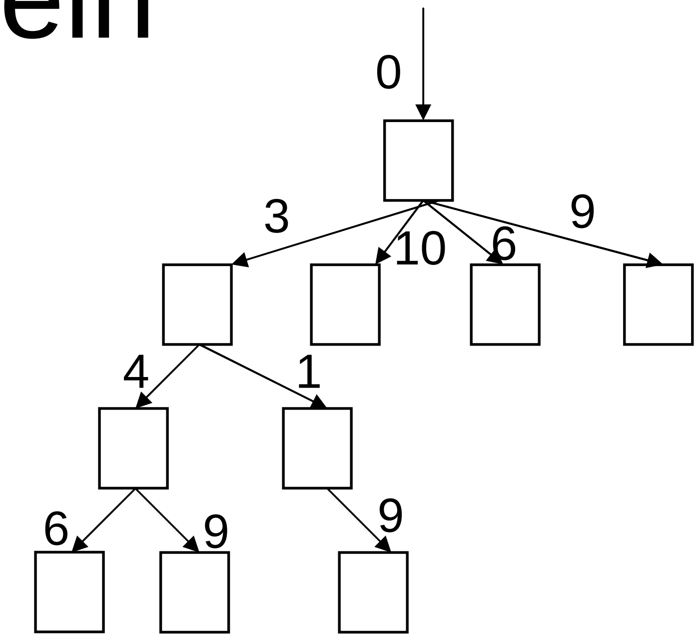
Taking yet each successive sidetrack incurs some incremental cost.

Cost of sidetrack sequence is sum of those incremental costs.

Now, enumerate sidetrack sequences in order of cost.
(& add 55 to each sequence's cost!)

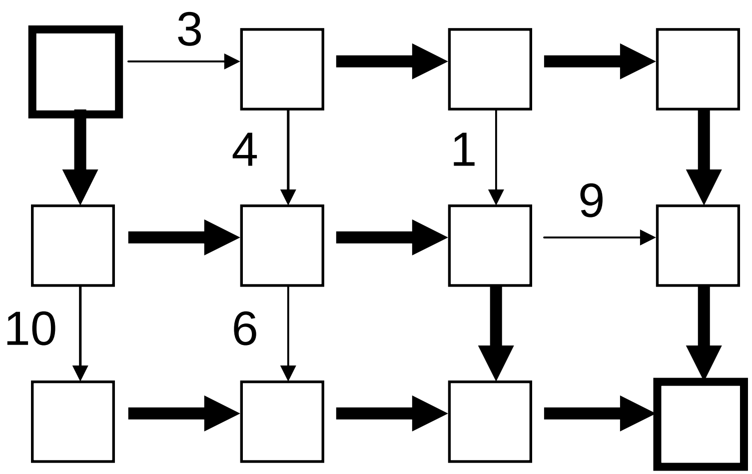
Eppstein

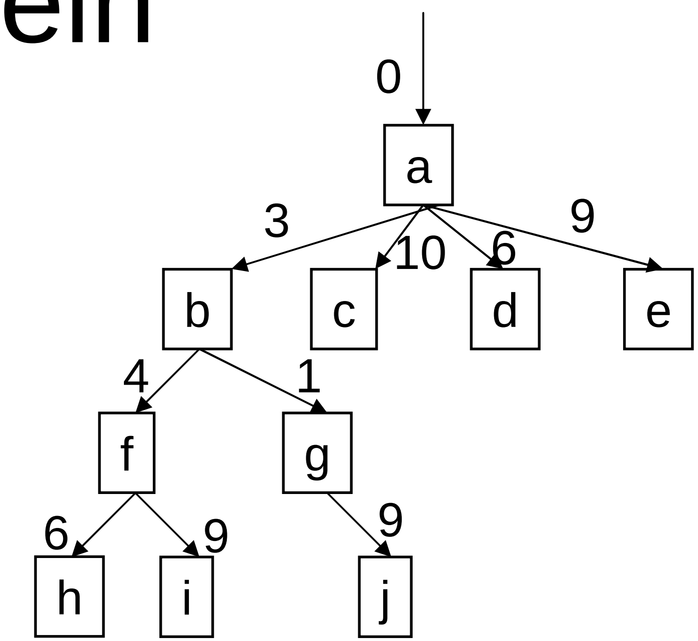




Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H

Eppstein

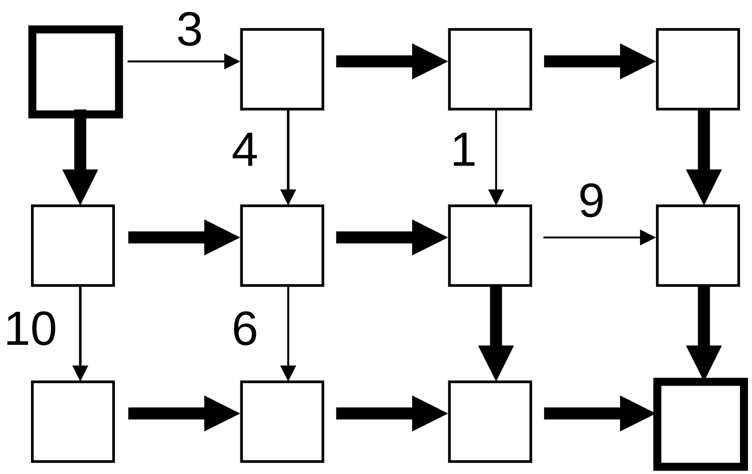


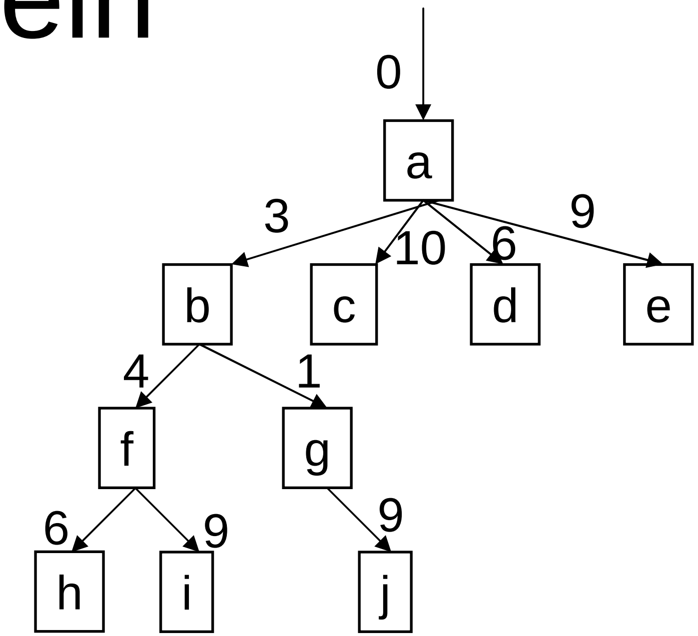


Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H

0.a

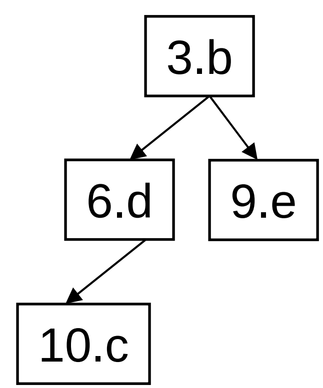
Eppstein



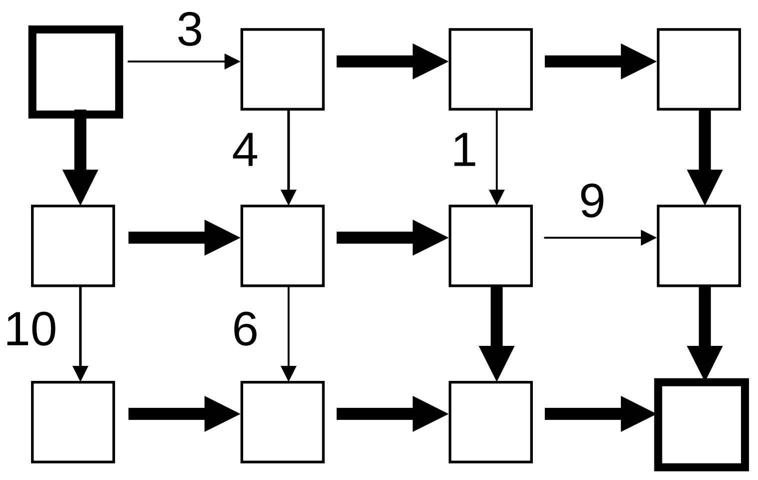


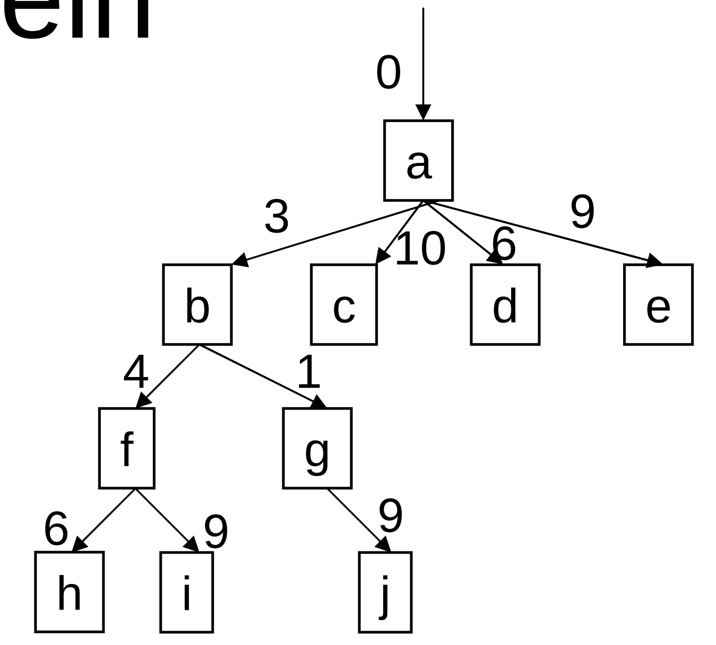
55

Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H



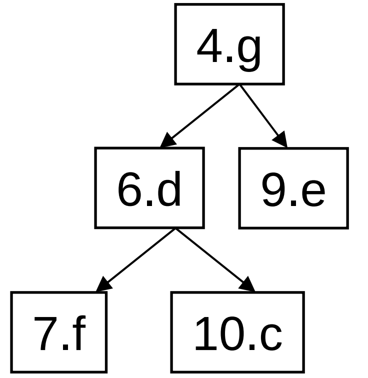
Eppstein



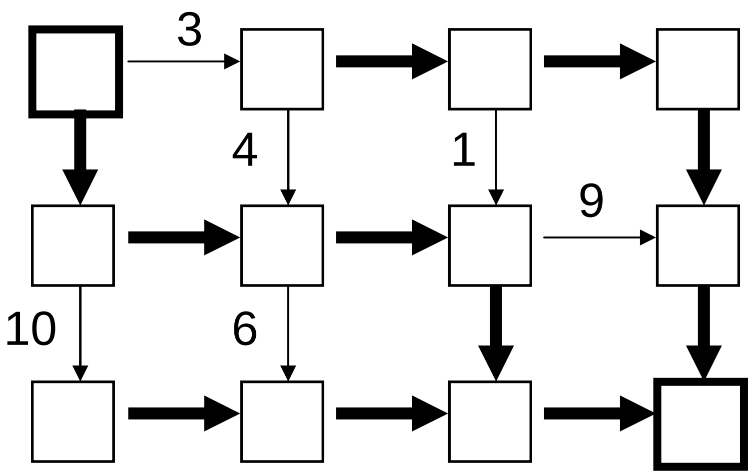


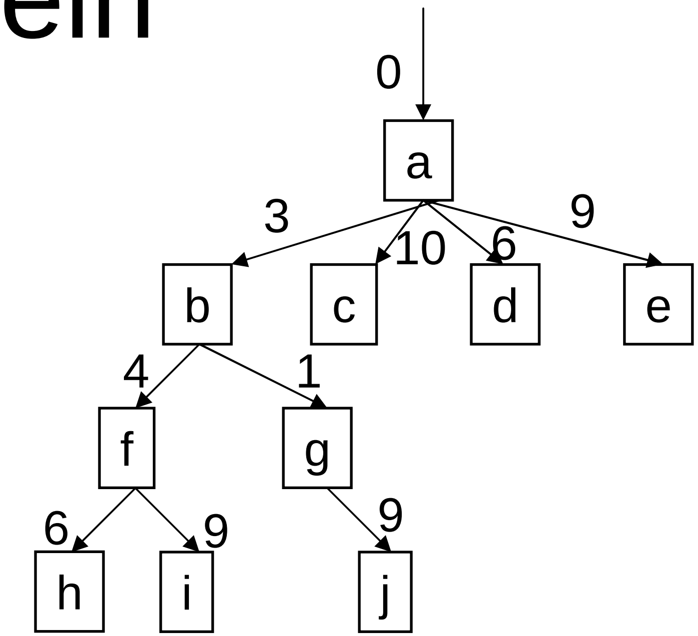
55
58

Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H



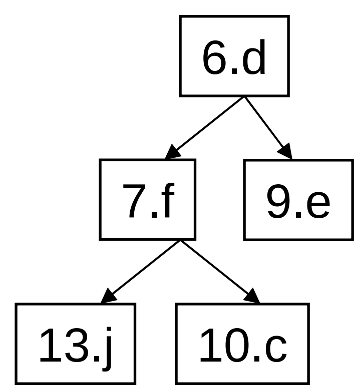
Eppstein



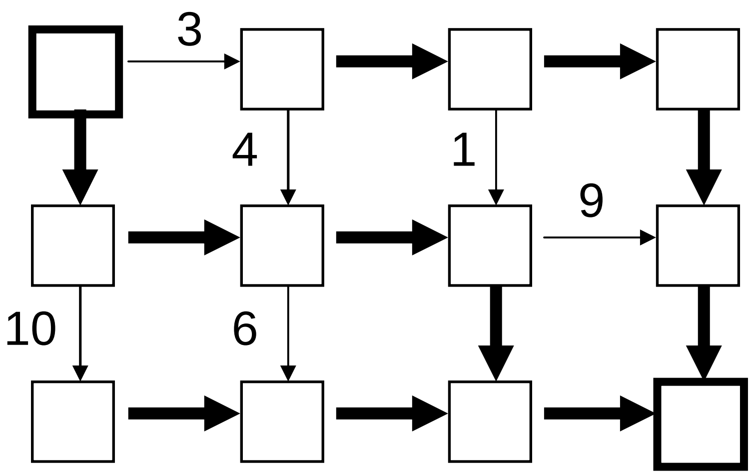


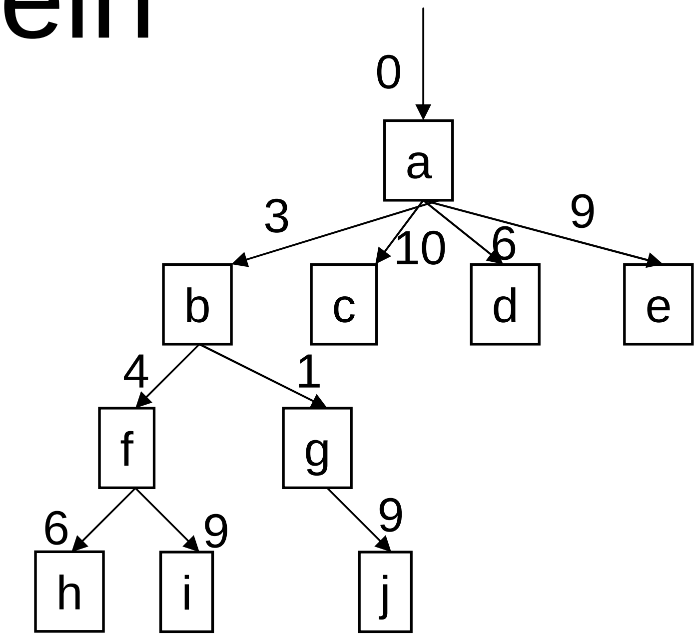
55
58
59

Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H

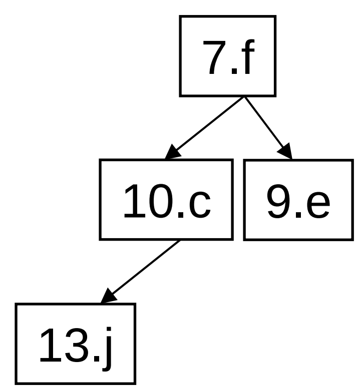


Eppstein



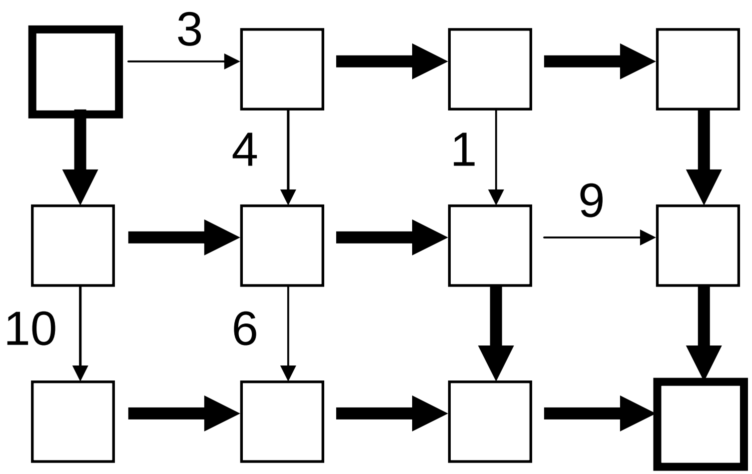


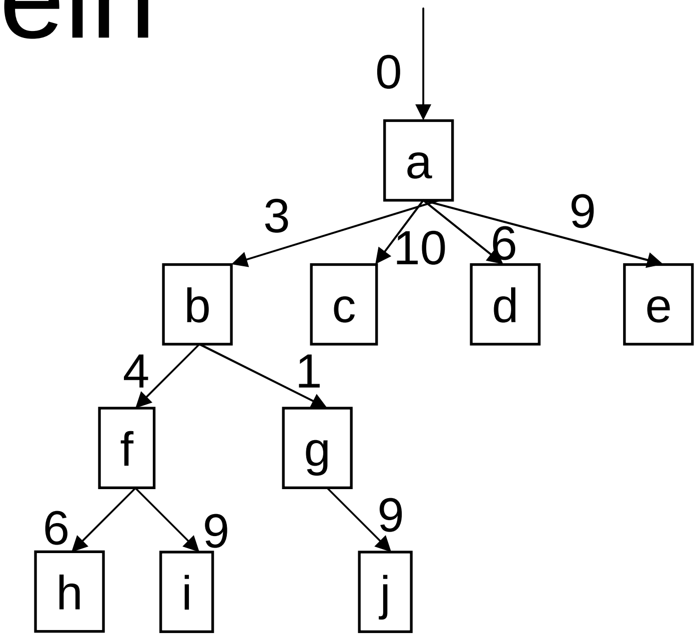
Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H



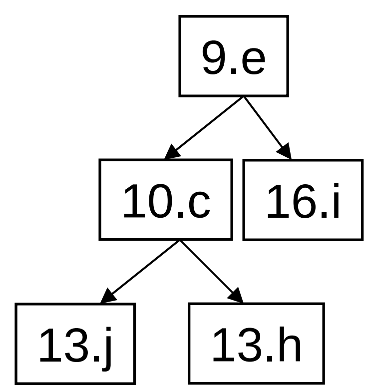
55
58
59
61

Eppstein



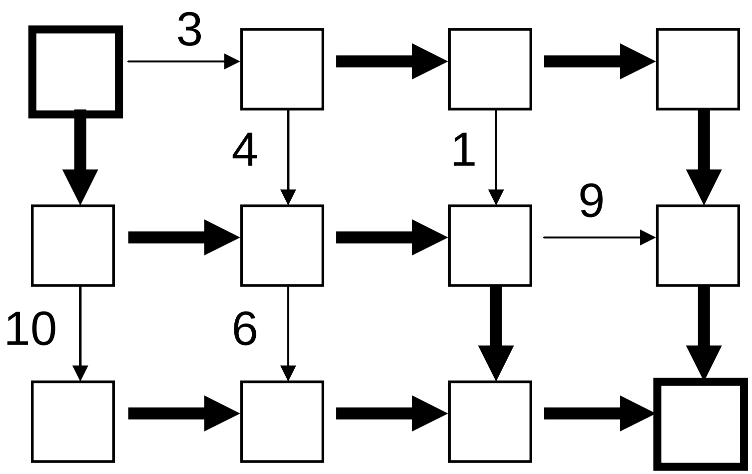


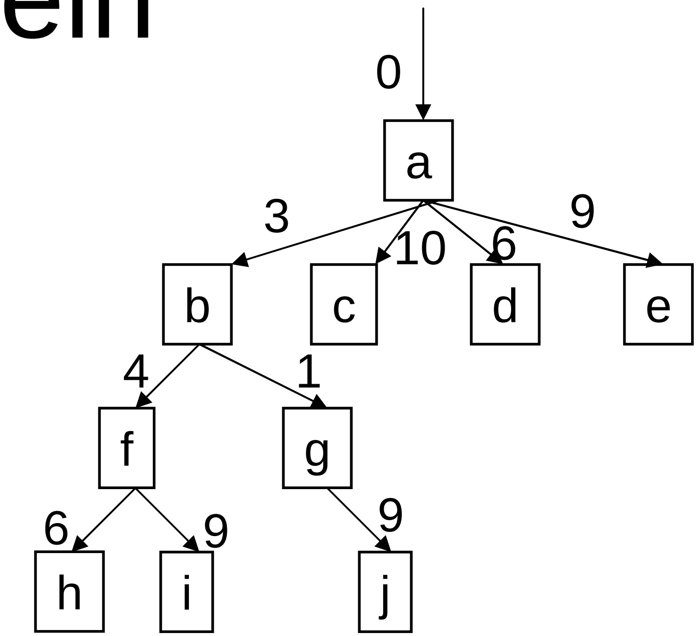
Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H



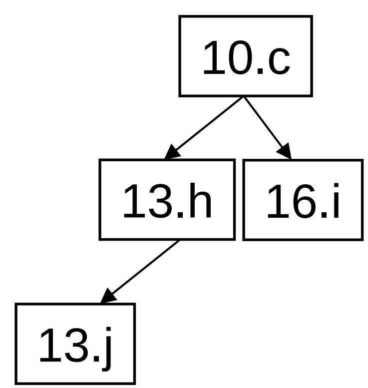
55
58
59
61
62

Eppstein



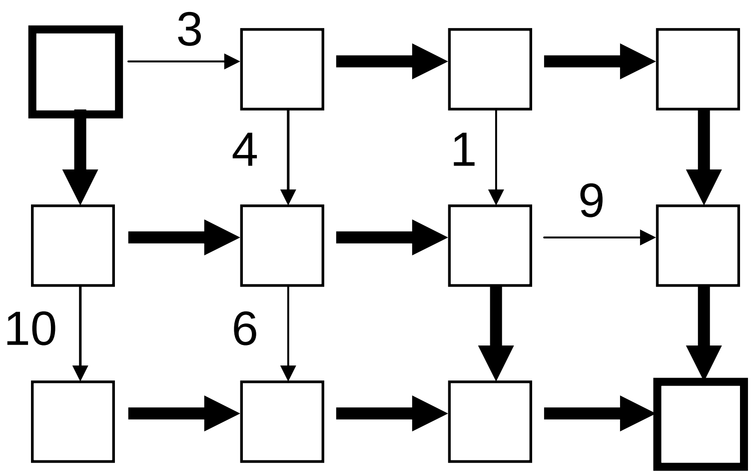


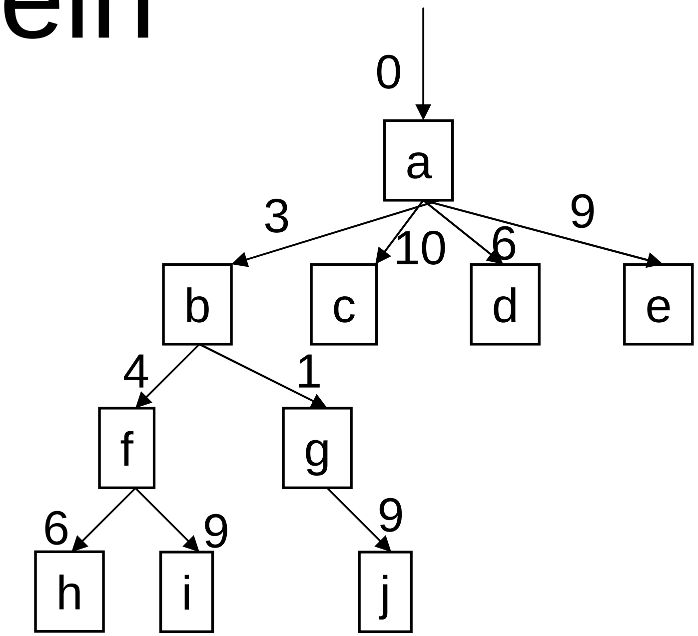
Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H



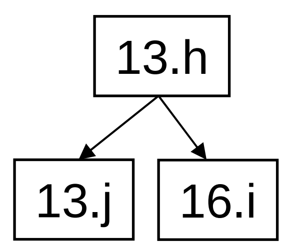
55
58
59
61
62
64

Eppstein



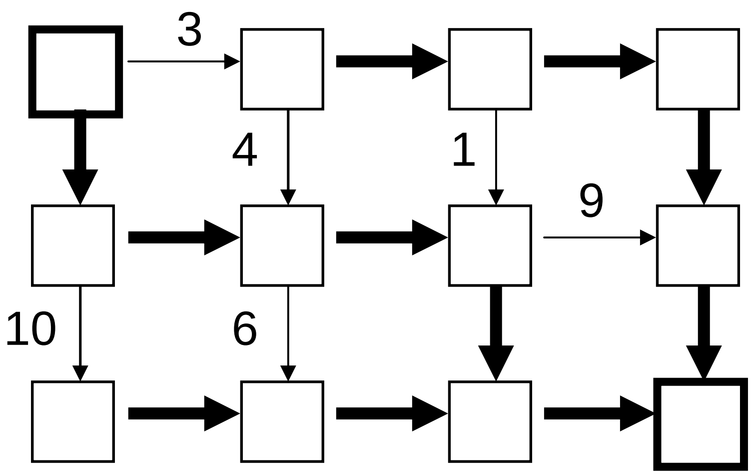


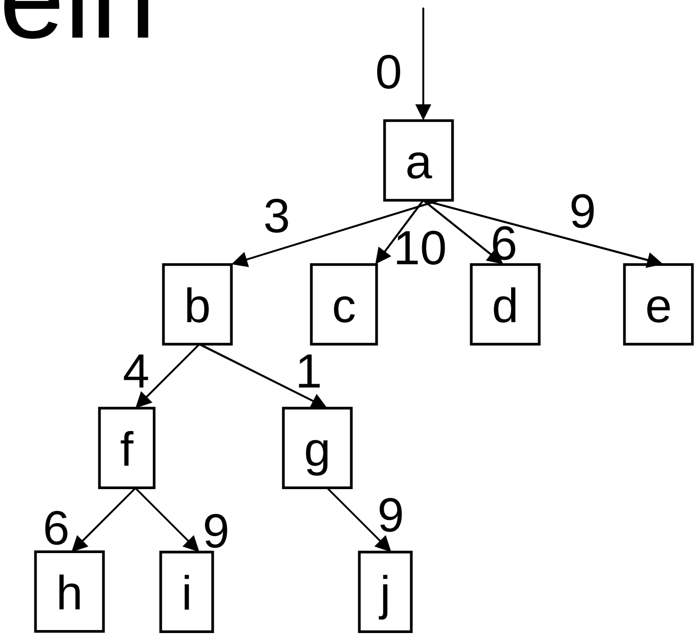
Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H



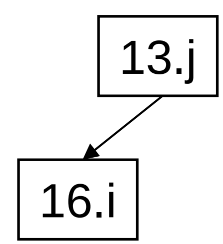
55
58
59
61
62
64
65

Eppstein



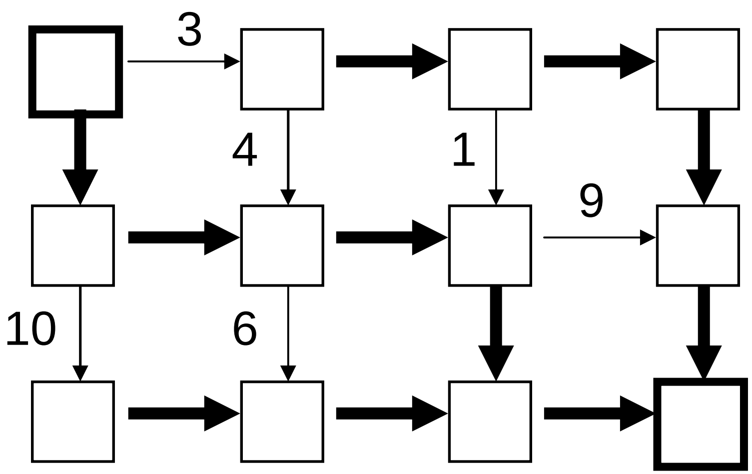


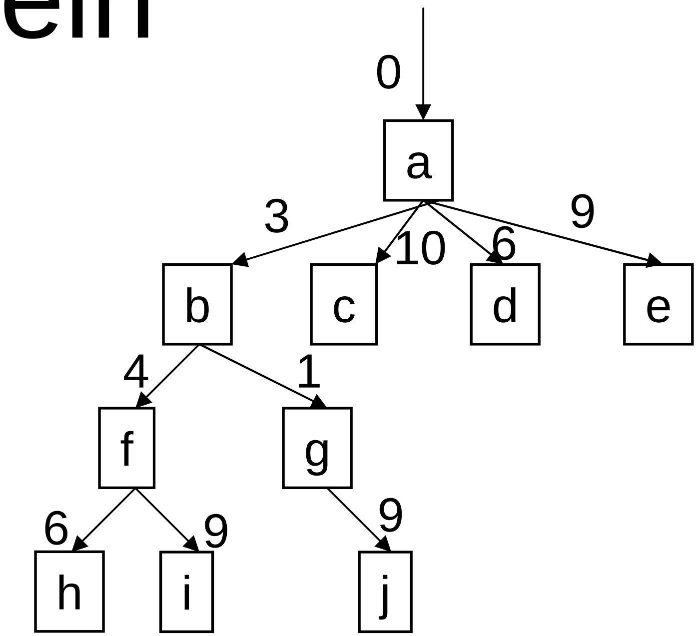
Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H



55
58
59
61
62
64
65
68

Eppstein



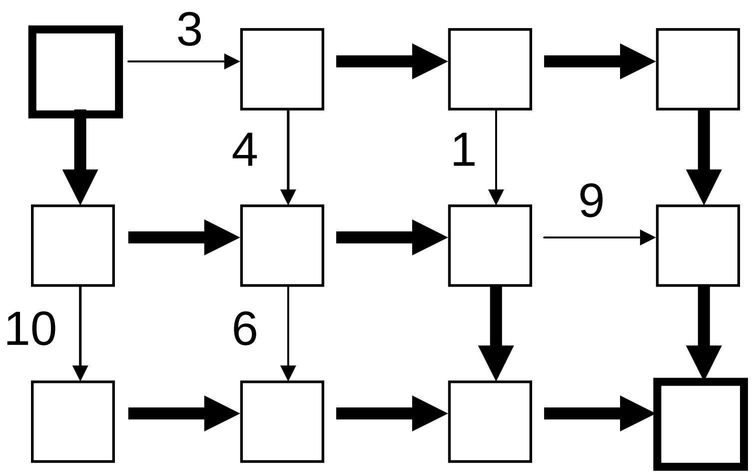


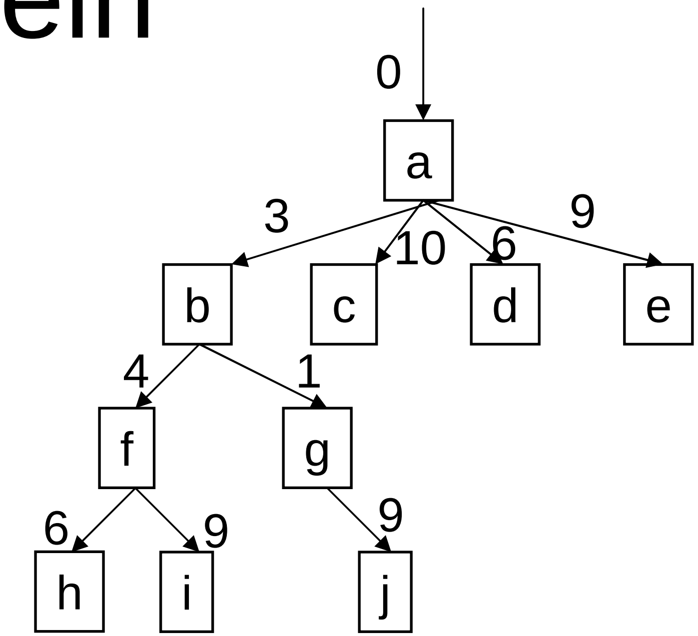
Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H

16.i

55
58
59
61
62
64
65
68
68

Eppstein



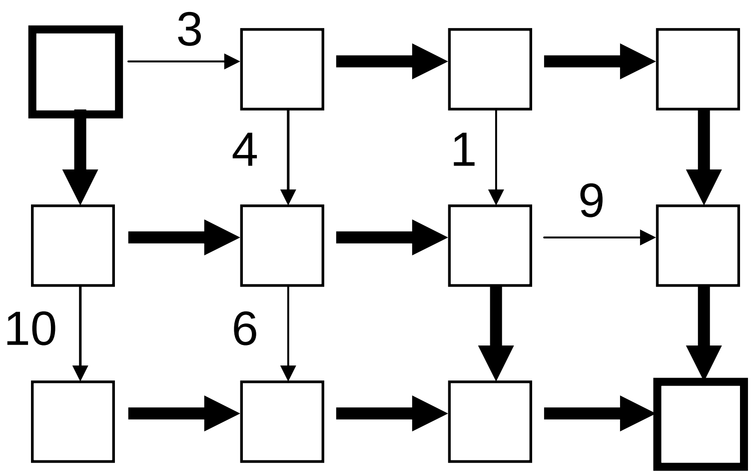


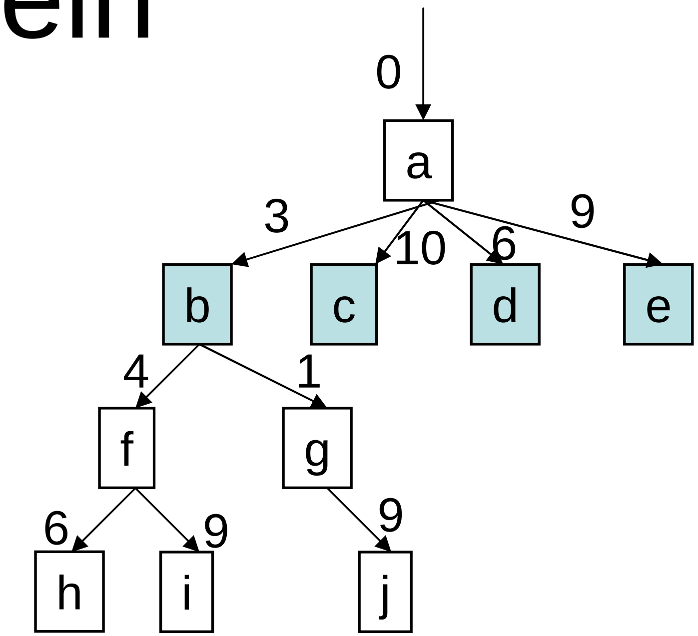
Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H

<empty heap>

55
58
59
61
62
64
65
68
68
71

Eppstein

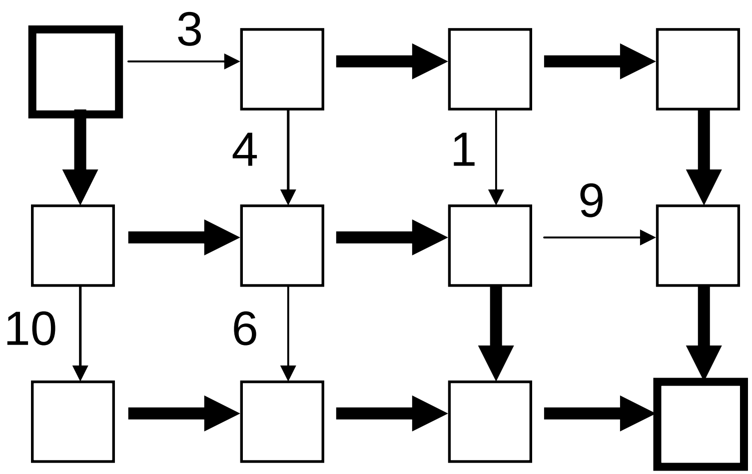


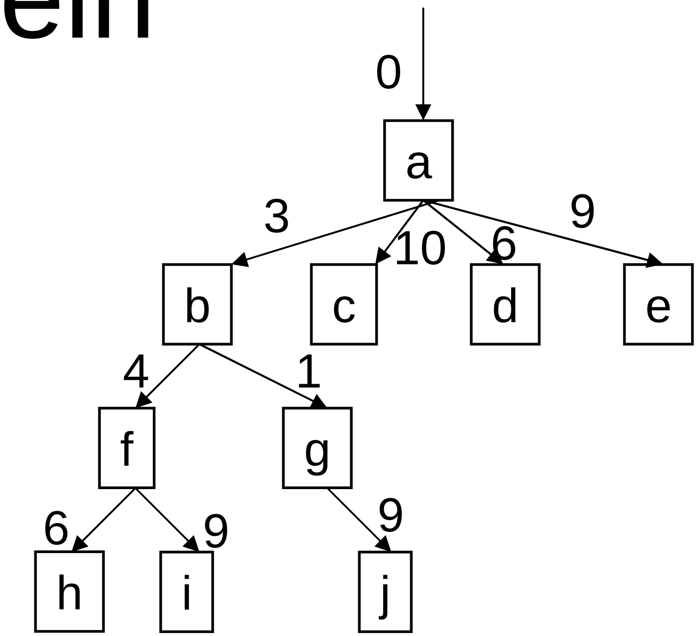


Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H

<u>Analysis</u>:
total pops = k
total pushes = km (if fully connected)
maxheapsize = O(km)
cost of pop = O(log km) = O(log k + log m)
cost of push = 1
→ O(km + k log k + k log m)
→ O(km + k log k)
→ km = kn^2 = bad term to have!!

Eppstein

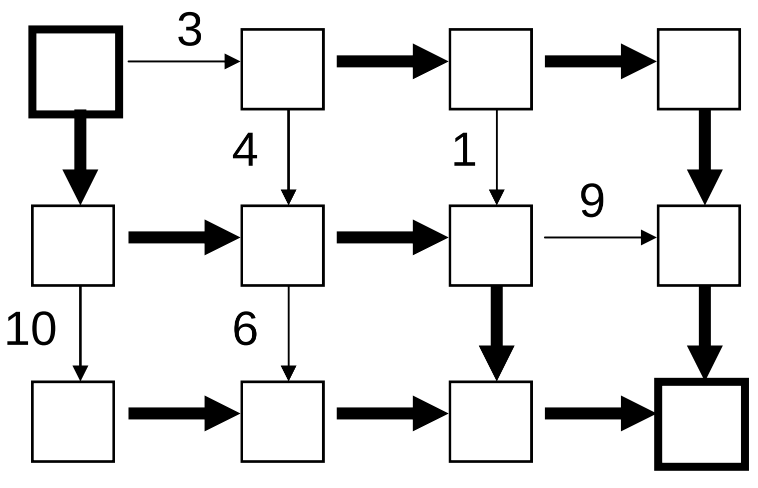


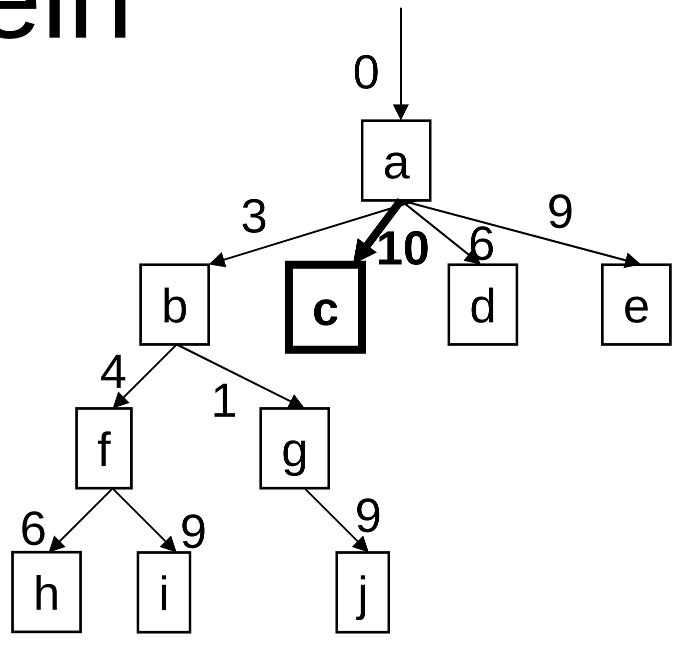


Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H

<u>Analysis</u> for <u>bounded outdegree tree</u>:
total pops = k
total pushes = k * constant
maxheapsize = O(k)
cost of pop = O(log k)
cost of push = 1
→ O(k + k log k)
→ O(k log k), no n factor

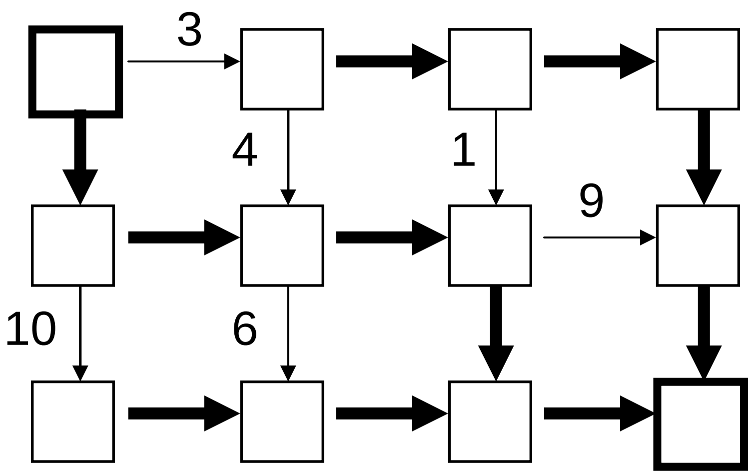
Eppstein

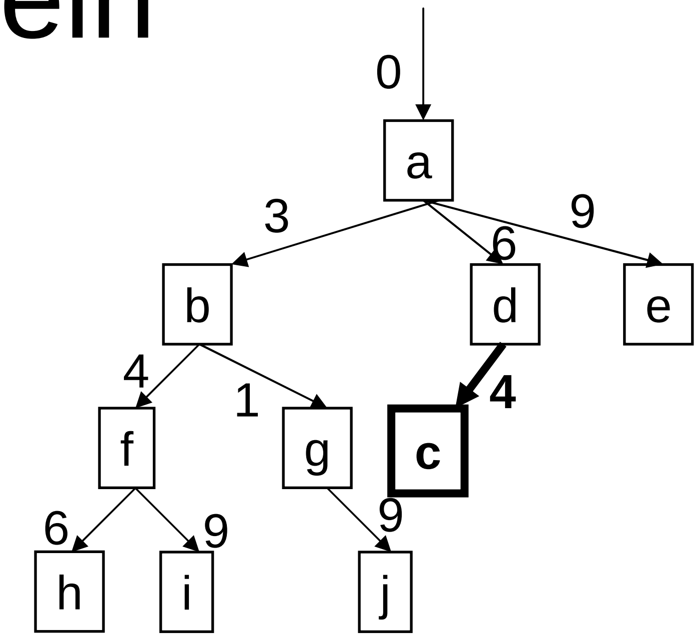




Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H

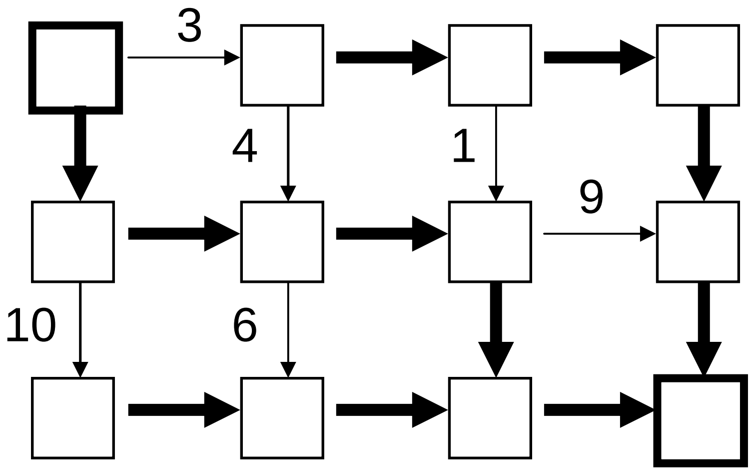
Eppstein

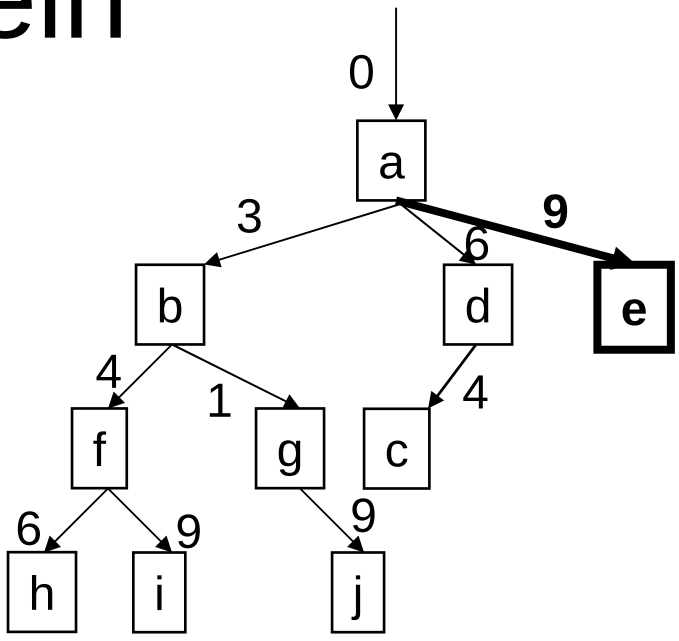




Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H

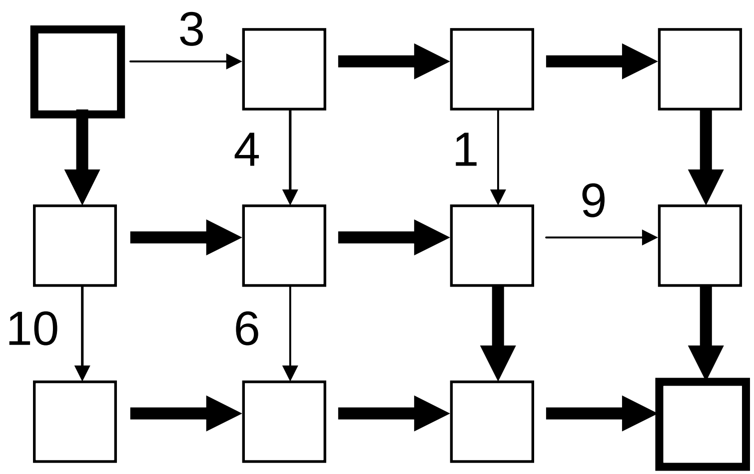
Eppstein

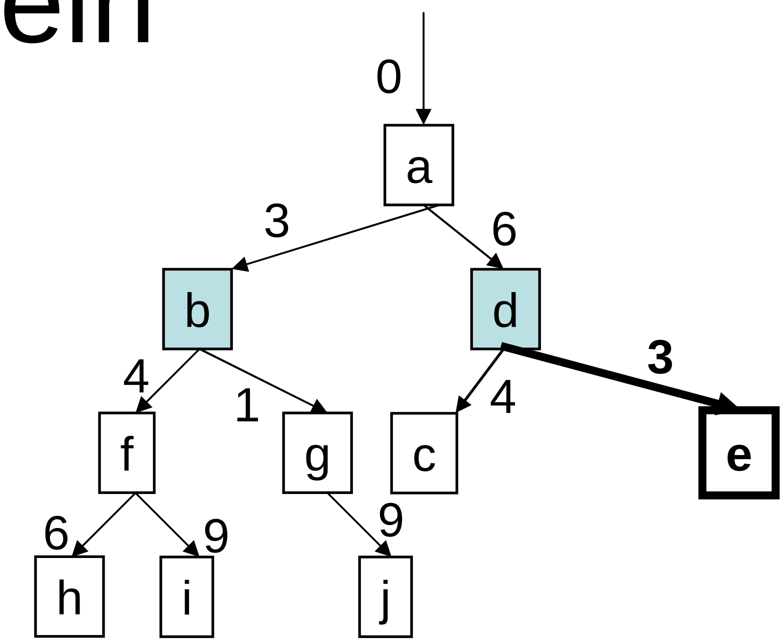




Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H

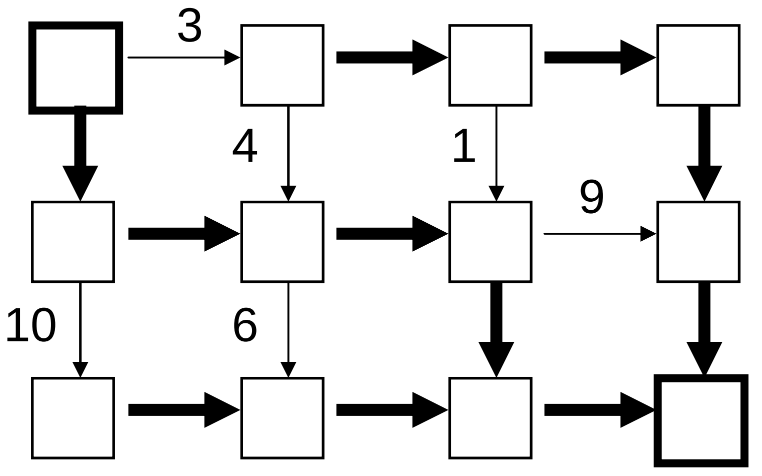
Eppstein

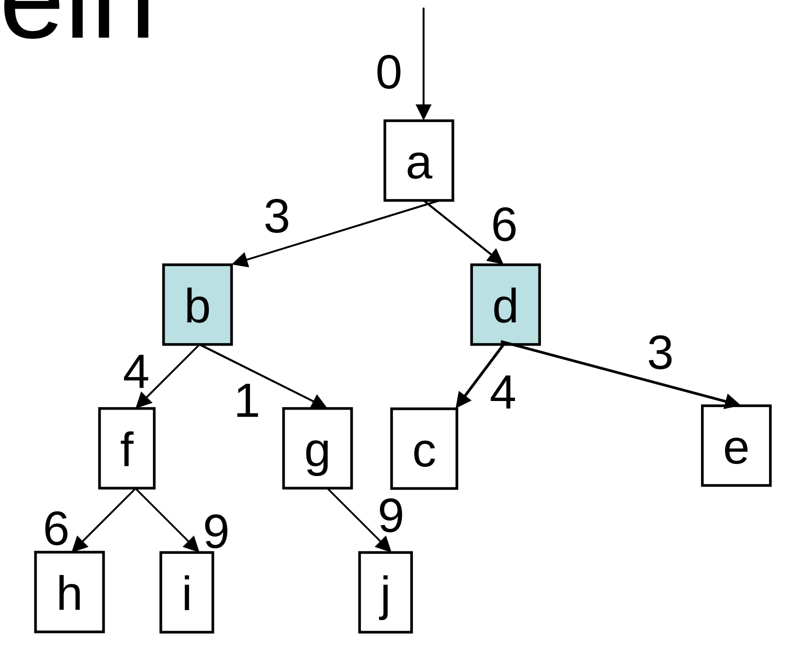




Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H

Eppstein

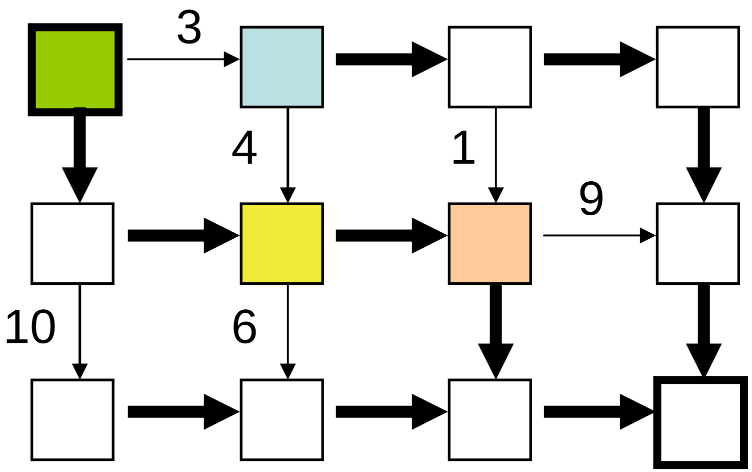


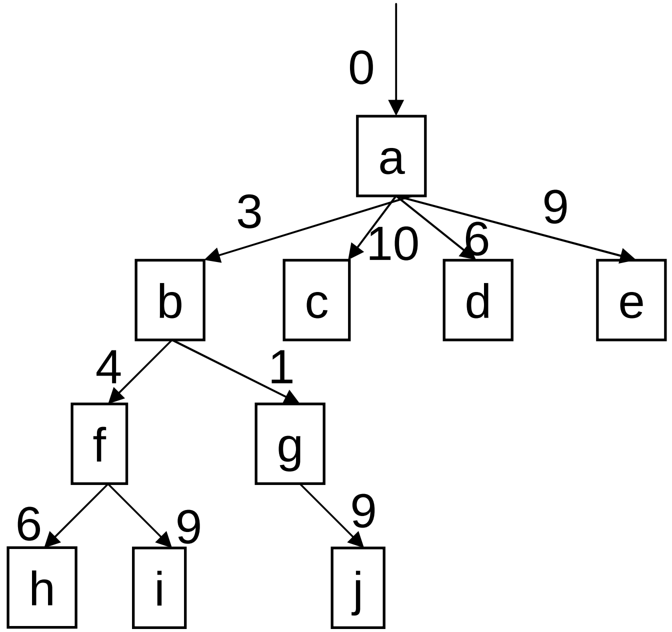


Algorithm (Breadth-First Search):
push root.0 onto heap H
for i = 1 to k
  pop node r.c1 from top of H
  print cost c1 + 55
  for j = each edge from r → s with cost c2
    push node s.(c1+c2) onto H

So we might be able to reduce the outdegree…

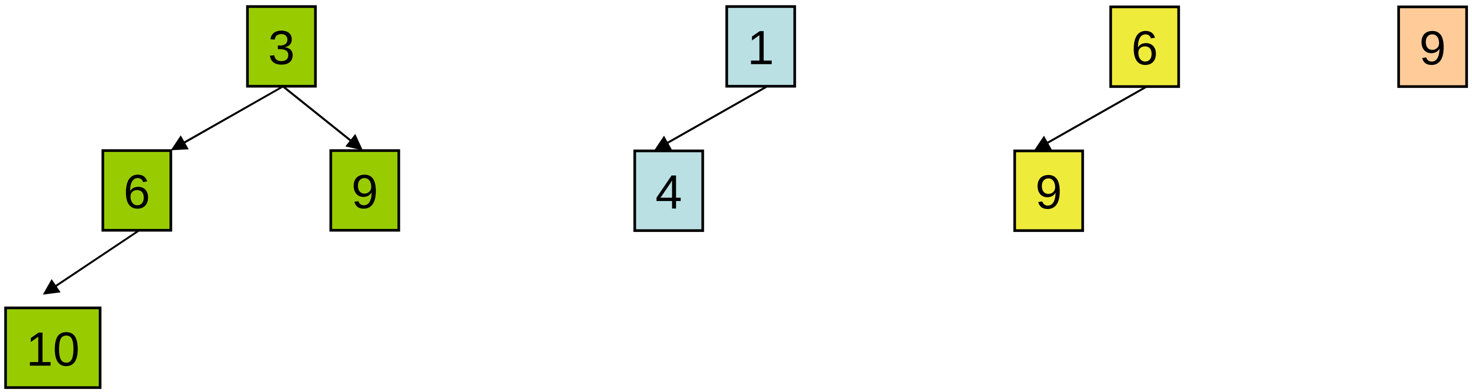
Goals:
- bounded outdegree sidetrack tree
- maintain 1-to-1 correspondence of
    - sidetrack tree paths
    - original graph start→final paths

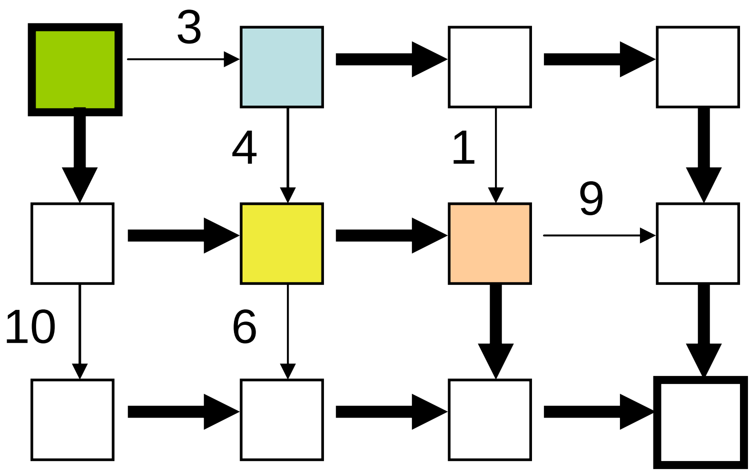




1. Pick best sidetrack of each node from start to final. Arrange all in binary heap.

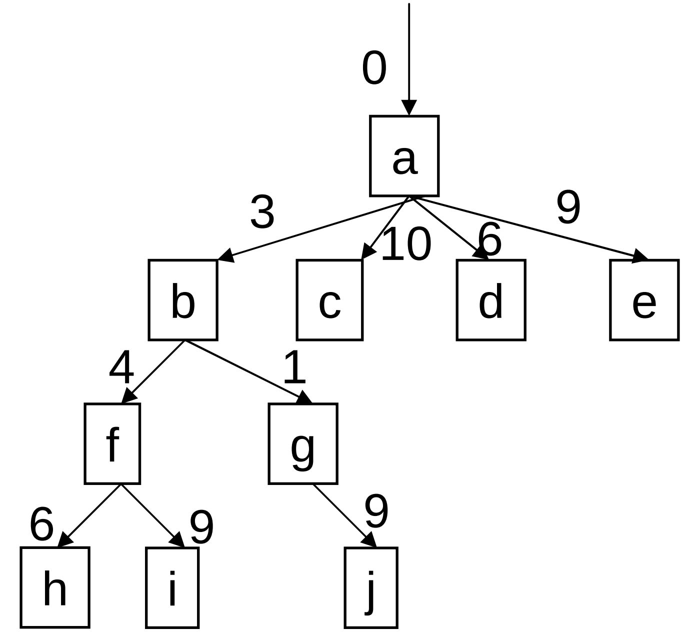
2. Repeat for other nodes besides start.

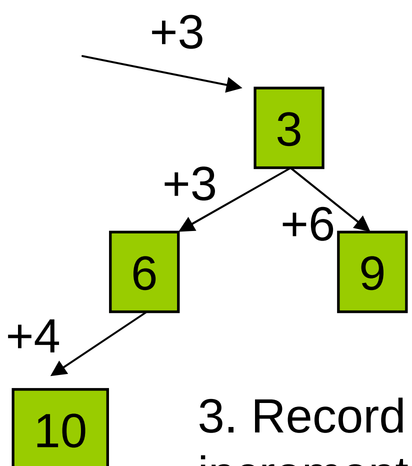


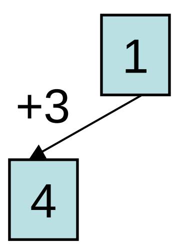


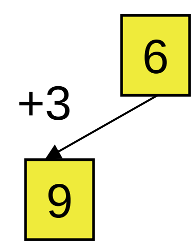
1. Pick best sidetrack of each node from start to final. Arrange all in binary heap.

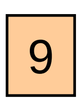
2. Repeat for other nodes besides start.



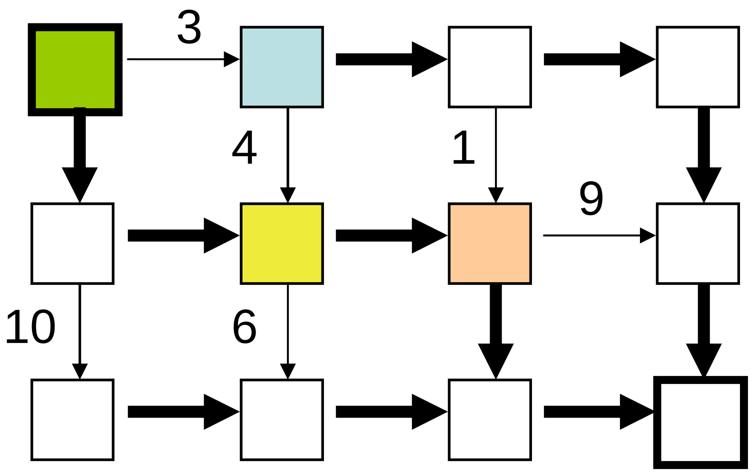


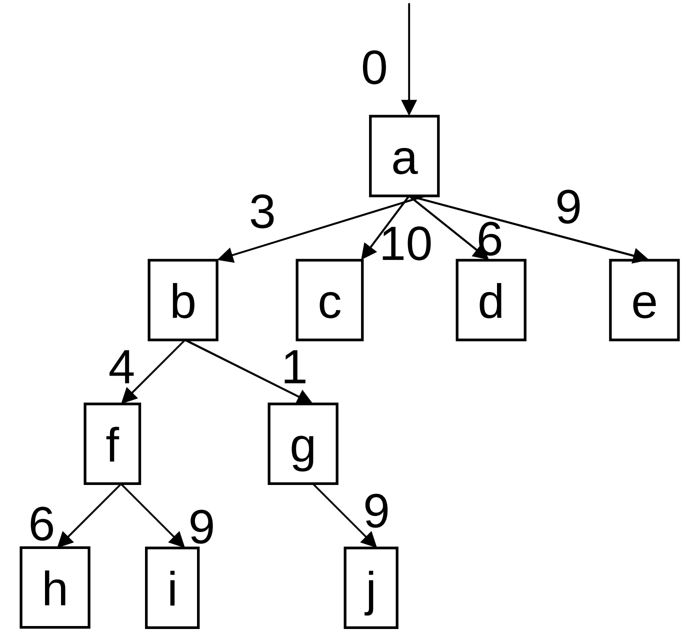






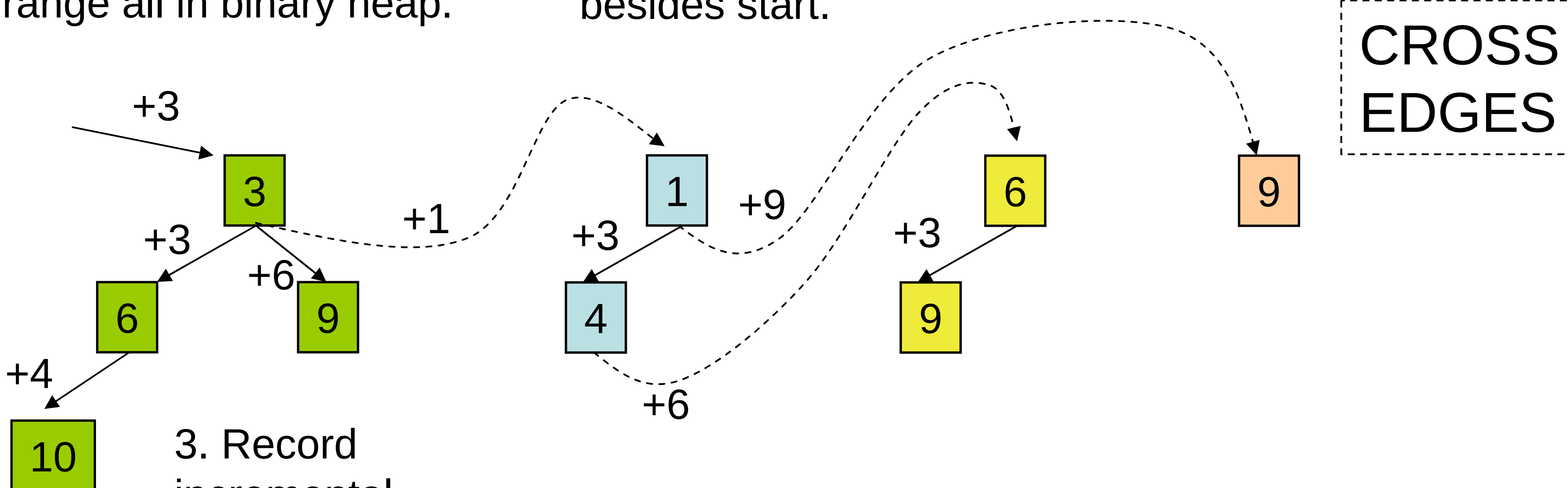
3. Record incremental costs.





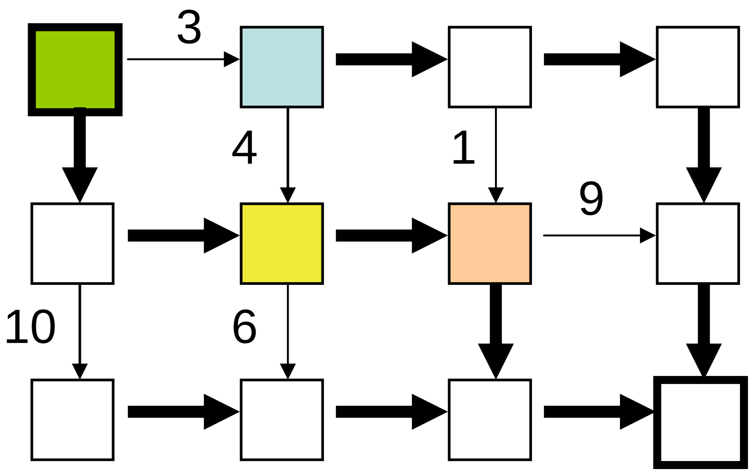
1. Pick best sidetrack of each node from start to final. Arrange all in binary heap.

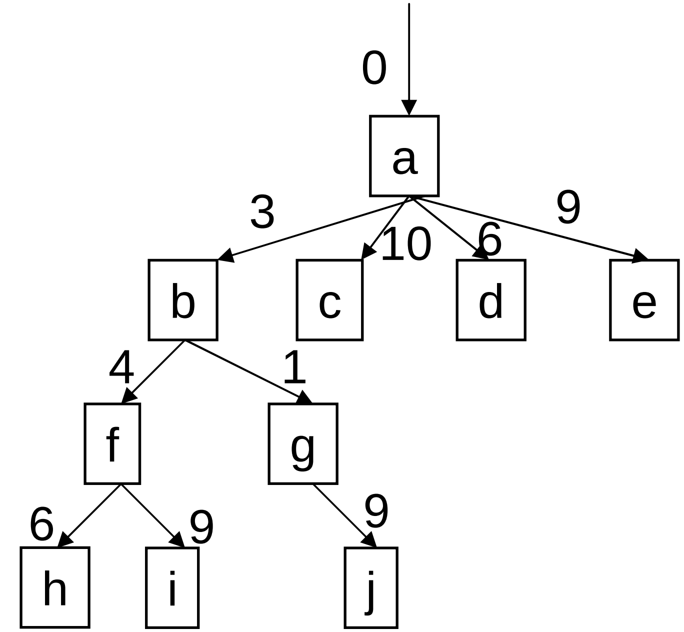
2. Repeat for other nodes besides start.



3. Record incremental costs.

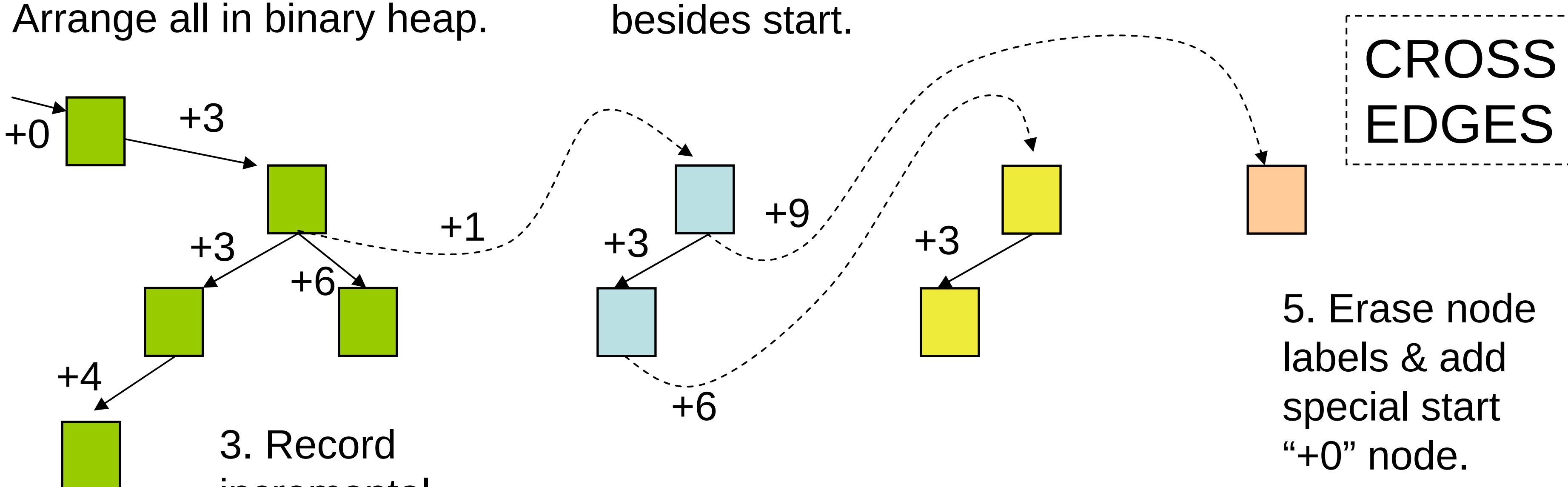
4. Connect each sidetrack node to the heap of its tail. Outdegree = 3.





1. Pick best sidetrack of each node from start to final. Arrange all in binary heap.

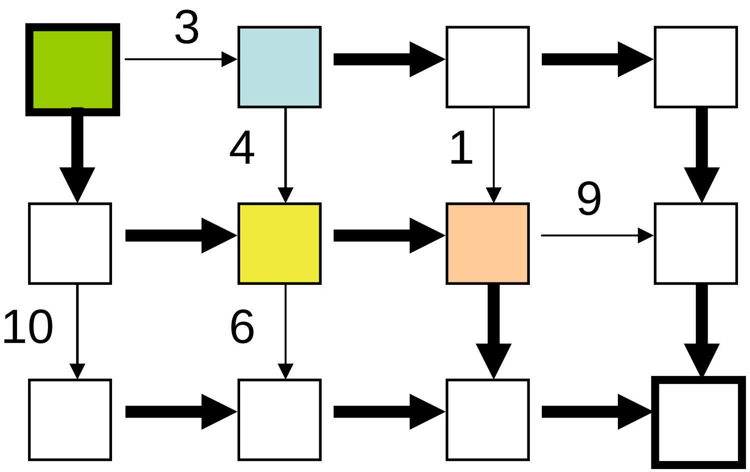
2. Repeat for other nodes besides start.

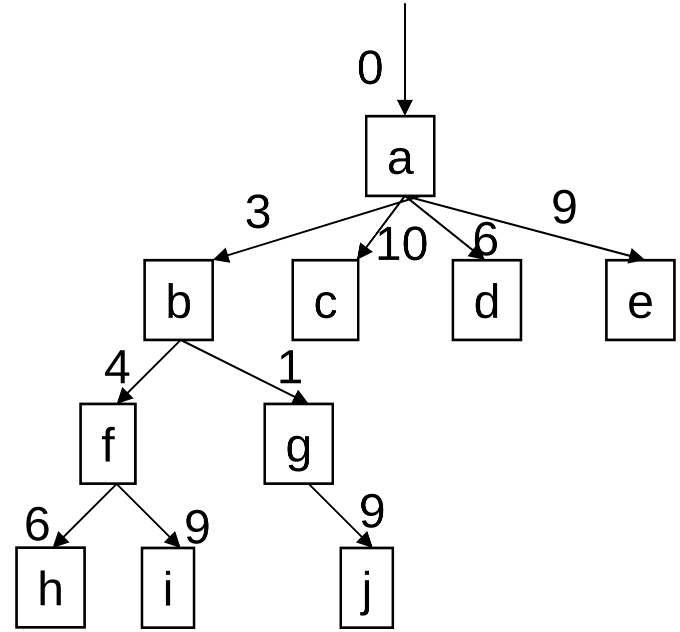


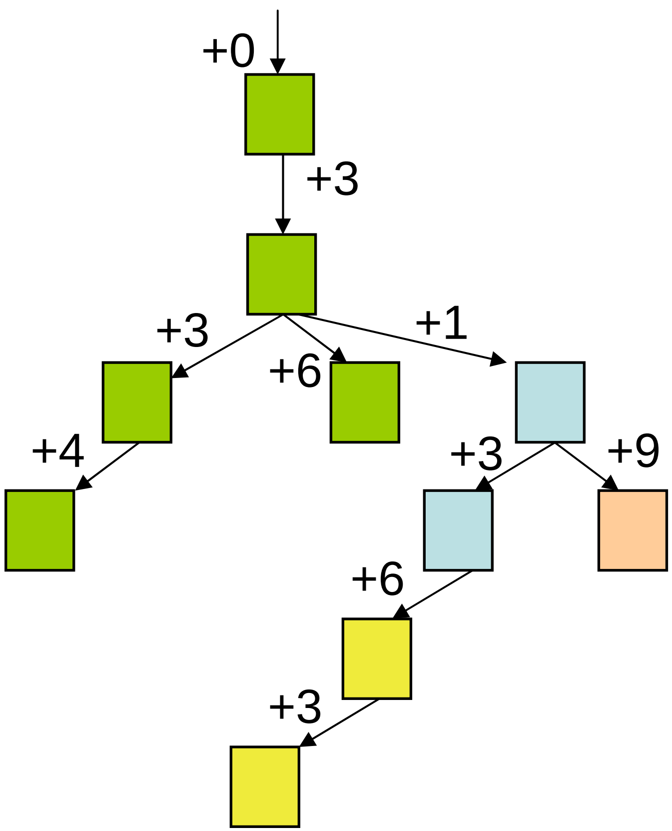
3. Record incremental costs.

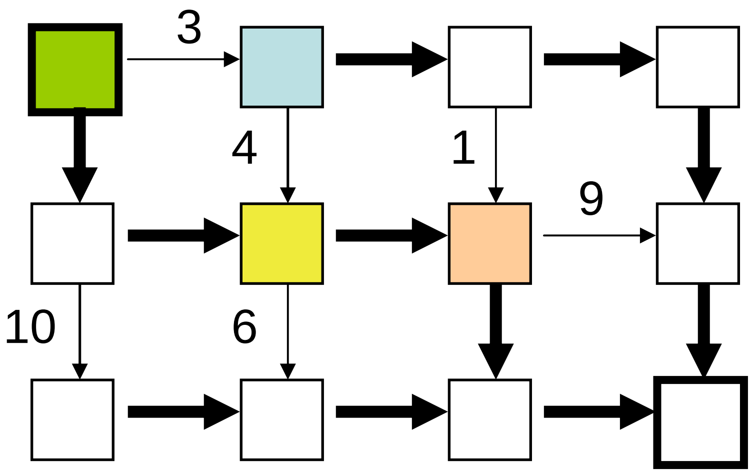
4. Connect each sidetrack node to the heap of its tail. Outdegree = 3.

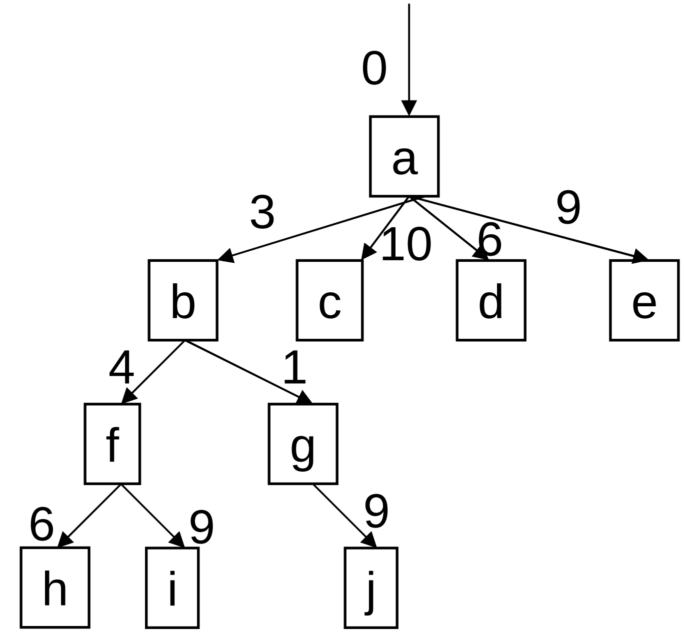
5. Erase node labels & add special start "+0" node.

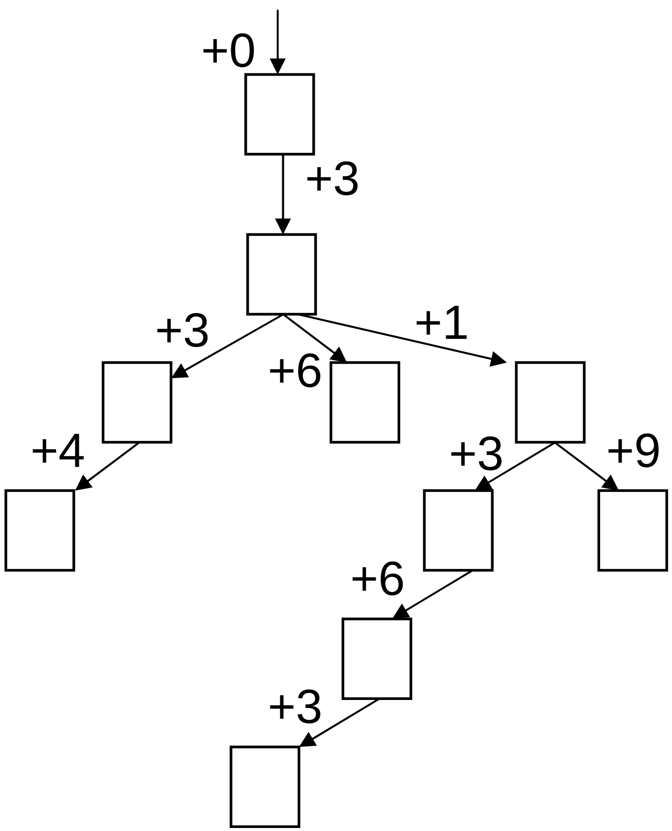

3
4
1
9
10
6


0
a
3
10
6
9
b
c
d
e
4
1
f
g
6
9
9
h
i
j


+0
+3
+3
+6
+1
+4
+3
+9
+6
+3







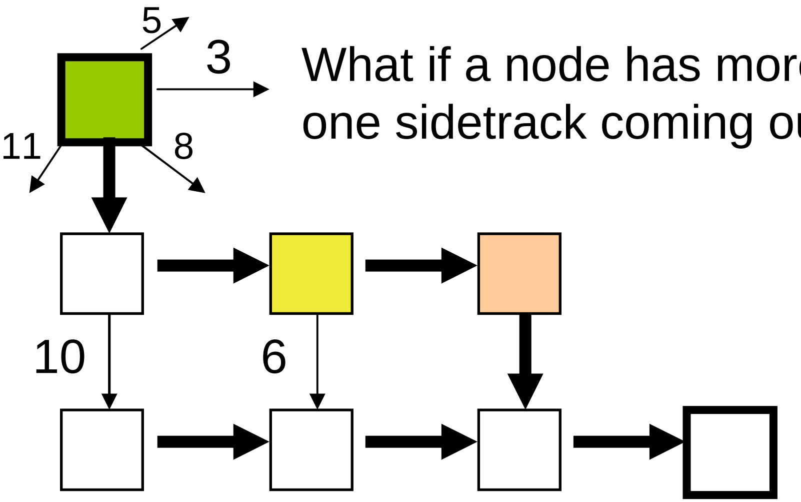
This new tree has the same 10 paths!
But it has max outdegree = 3.
Now same algorithm (BFS) works better!

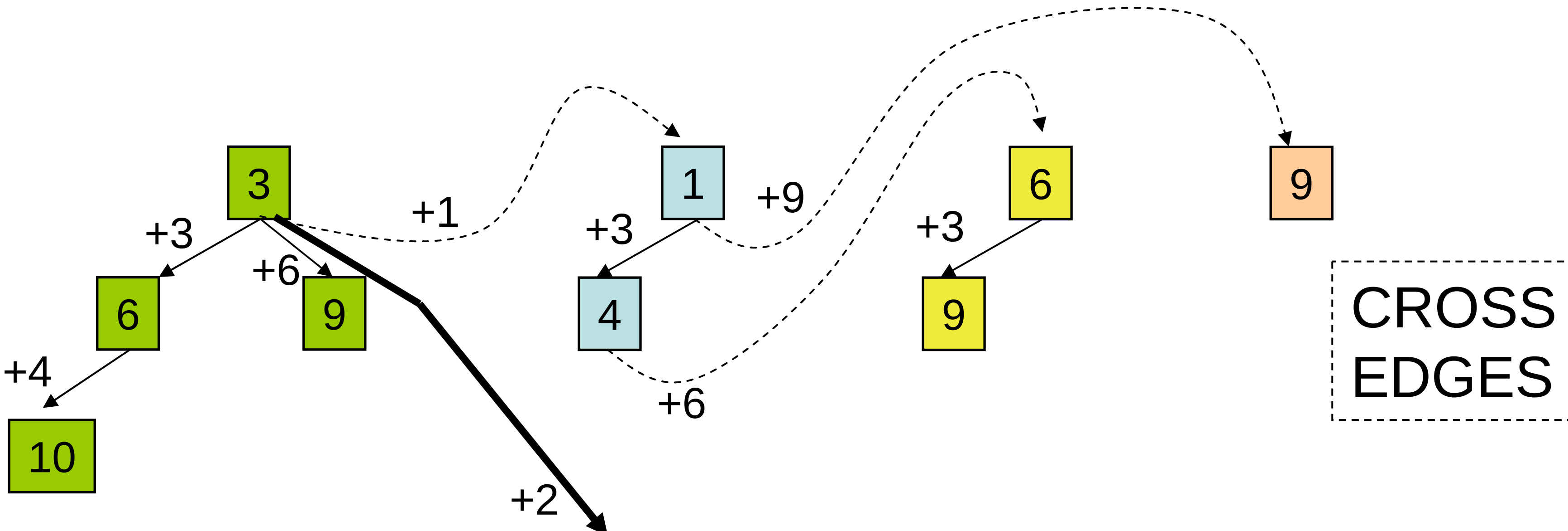
DONE!

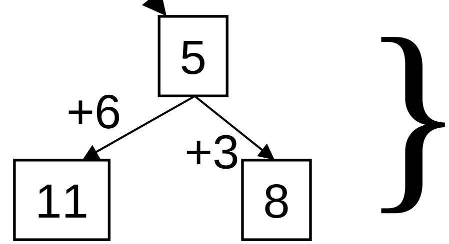
Detail #1

multiple sidetracks

What if a node has more than one sidetrack coming out?



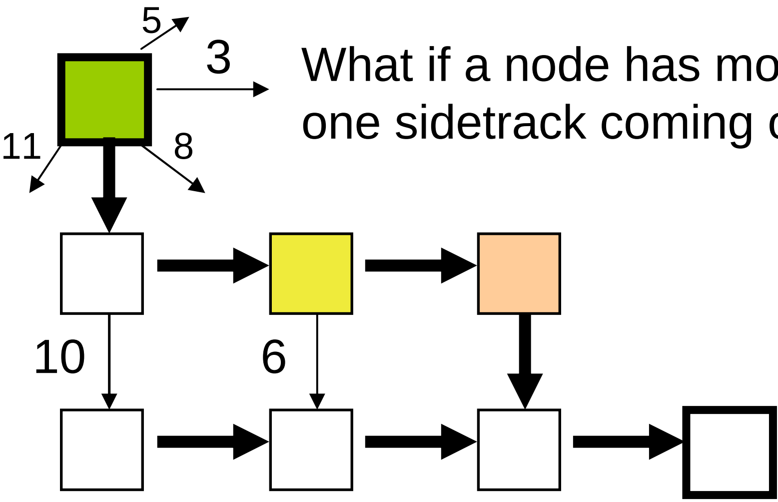




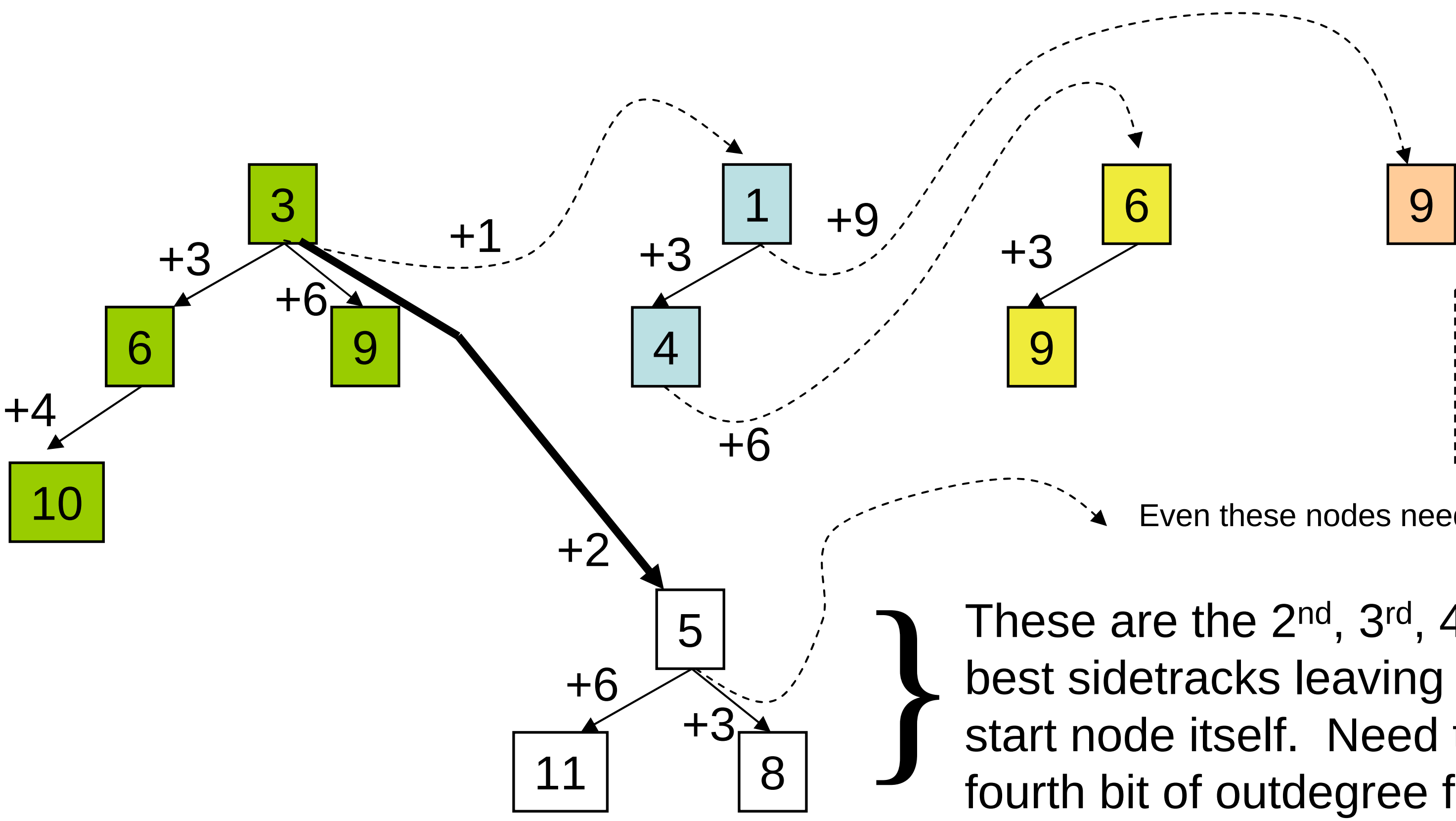
These are the 2nd, 3rd, 4th, etc best sidetracks leaving the start node itself. Need to add this fourth outdegree for all nodes in the new structure.

Detail #1

multiple sidetracks



What if a node has more than one sidetrack coming out?



CROSS EDGES

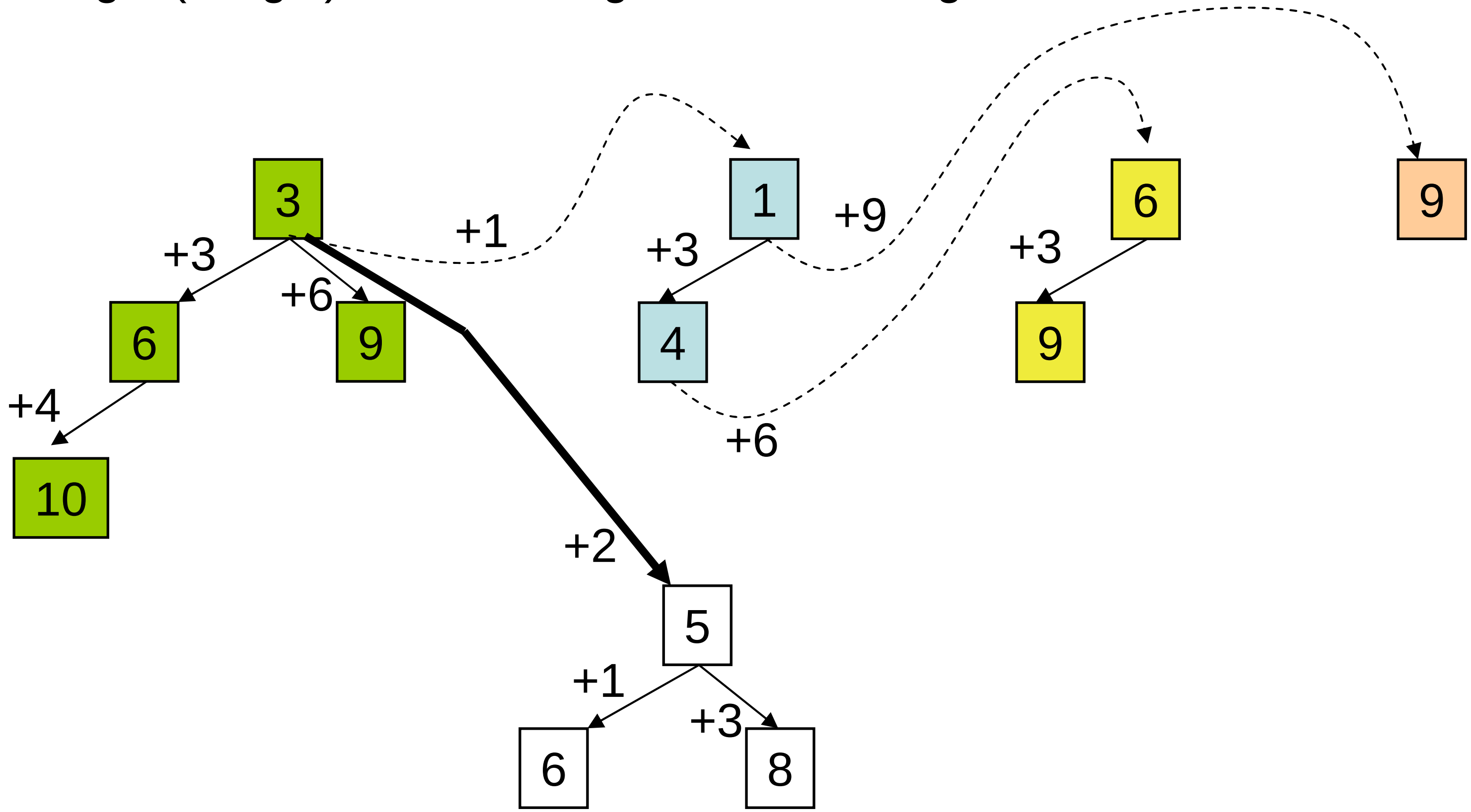
Even these nodes need cross edges.

These are the 2nd, 3rd, 4th, etc best sidetracks leaving the start node itself. Need to add this fourth bit of outdegree for all nodes in the new structure.

Detail #2

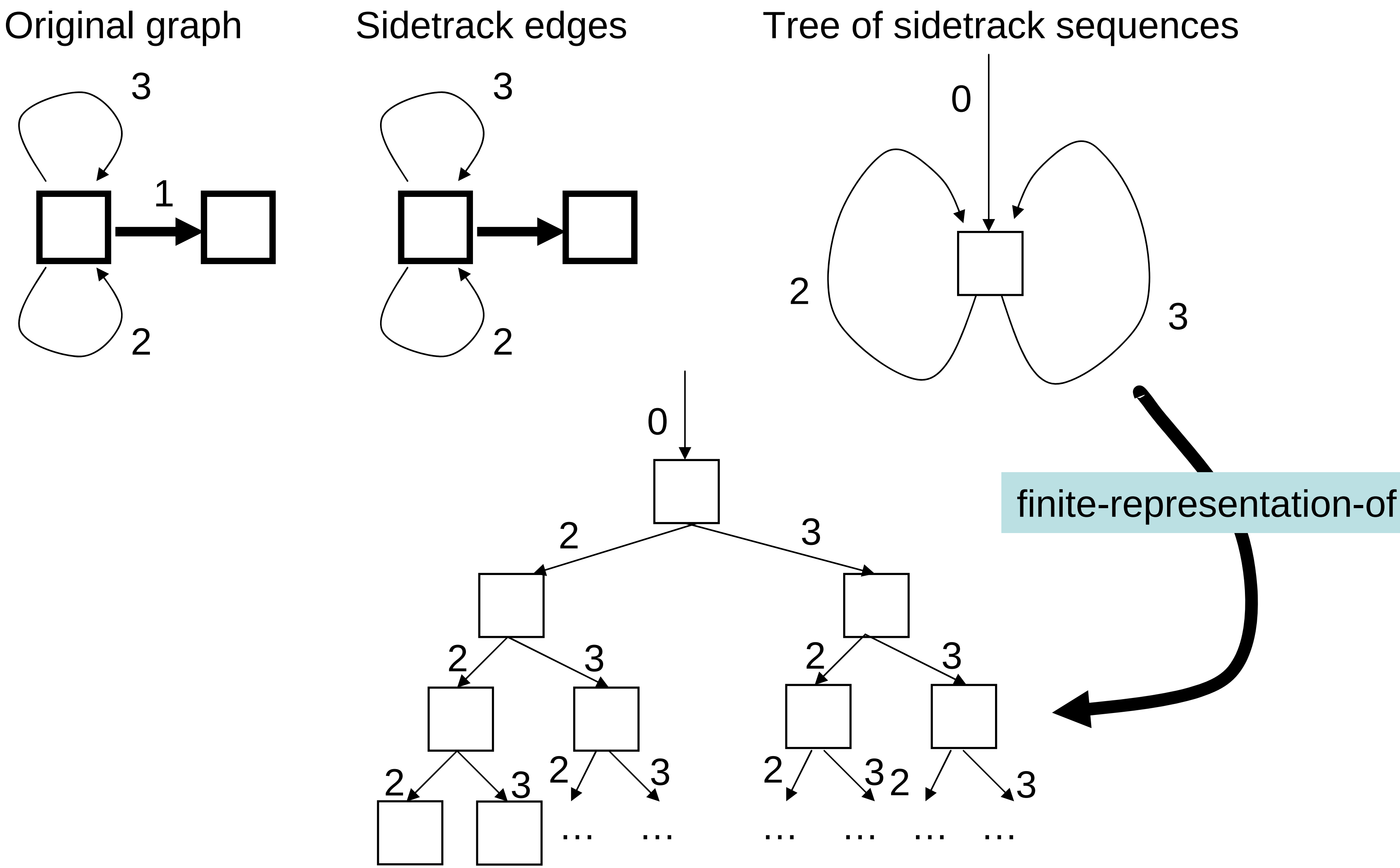
structure sharing

Notice that 6 and 9 appear many times.
Heaps need to be built so that they share a bunch of sub-structure.
Done carefully, the whole structure can be built in O(m + n log n) time, leaving O(k log k) for extracting the items using BFS.

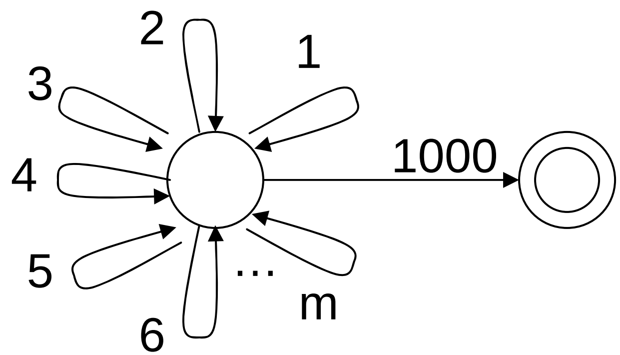


Detail #3

loops are no problem



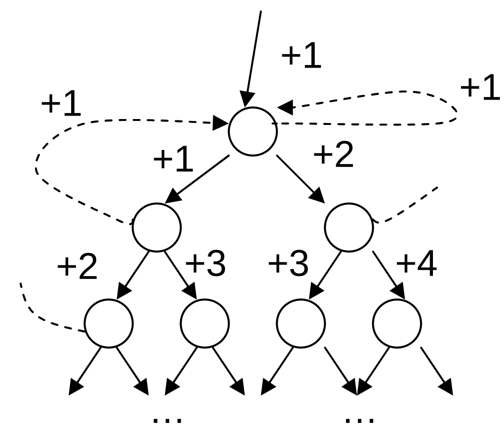
K-Best Dijkstra v. Eppstein



K-Best Dijkstra

Push (S,0)
Pop (S,0)
Do m+1 pushes
Pop (S,1)
Do m+1 pushes
Pop (S,1)
Do m+1 pushes
…
At least O(km) work before any complete path is generated…

Eppstein



Each BFS-pop corresponds to a complete path:

1-1000, 2-1000, 1-1-1000, 3-1000, 1-2-1000, 2-1-1000, 1-1-1-1000, 4-1000, 3-1-1000, 1-3-1000, 2-2-1000, 1-1-2-1000, …

the end