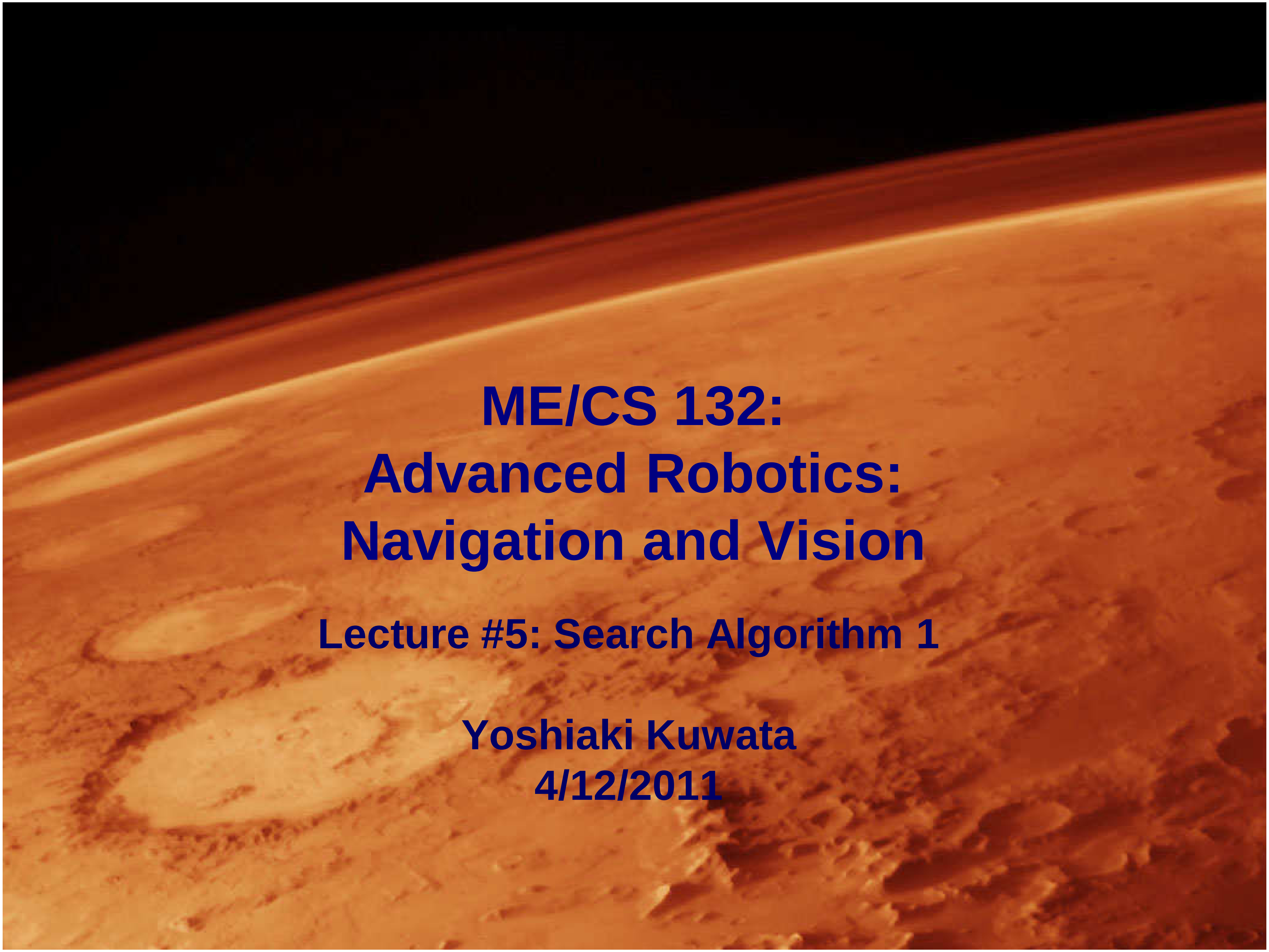


ME/CS 132: Advanced Robotics: Navigation and Vision

Lecture #5: Search Algorithm 1

Yoshiaki Kuwata

4/12/2011

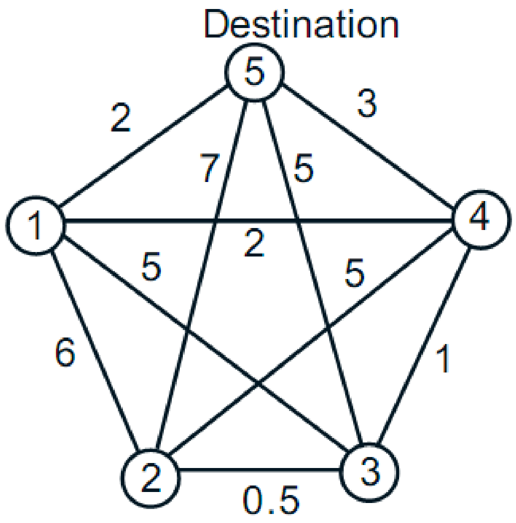
Lecture Overview

- Introduction
- Label Correcting Algorithm
  - Core idea
  - Depth-first search
  - Breadth-first search
  - Dijkstra
- More efficient search
  - A*
  - Advanced initialization

Shortest Path Applications

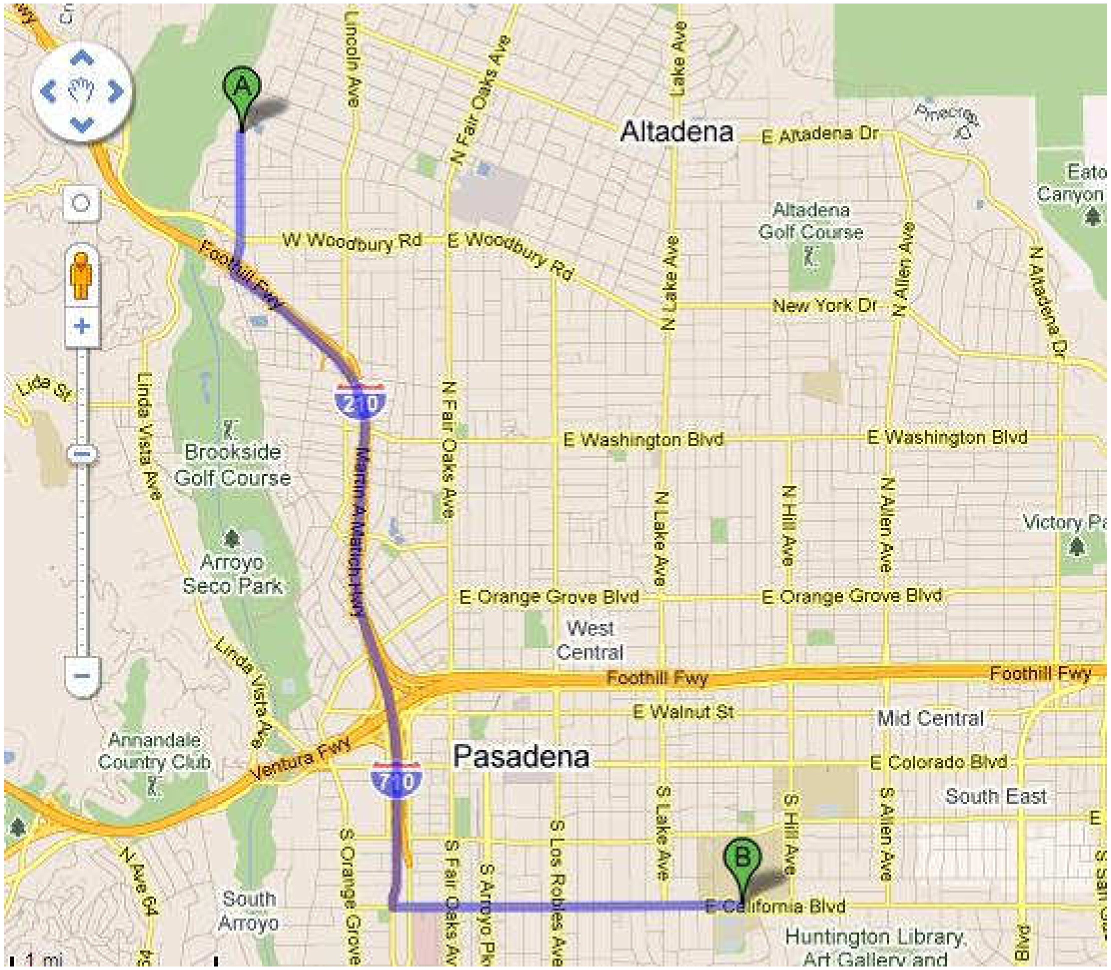
- From Chapter 2 of “Dynamic Programming and Optimal Control” by Dimitri Bertsekas

- What is the minimum cost of getting to node 5?



Shortest Path Applications

- Road Network

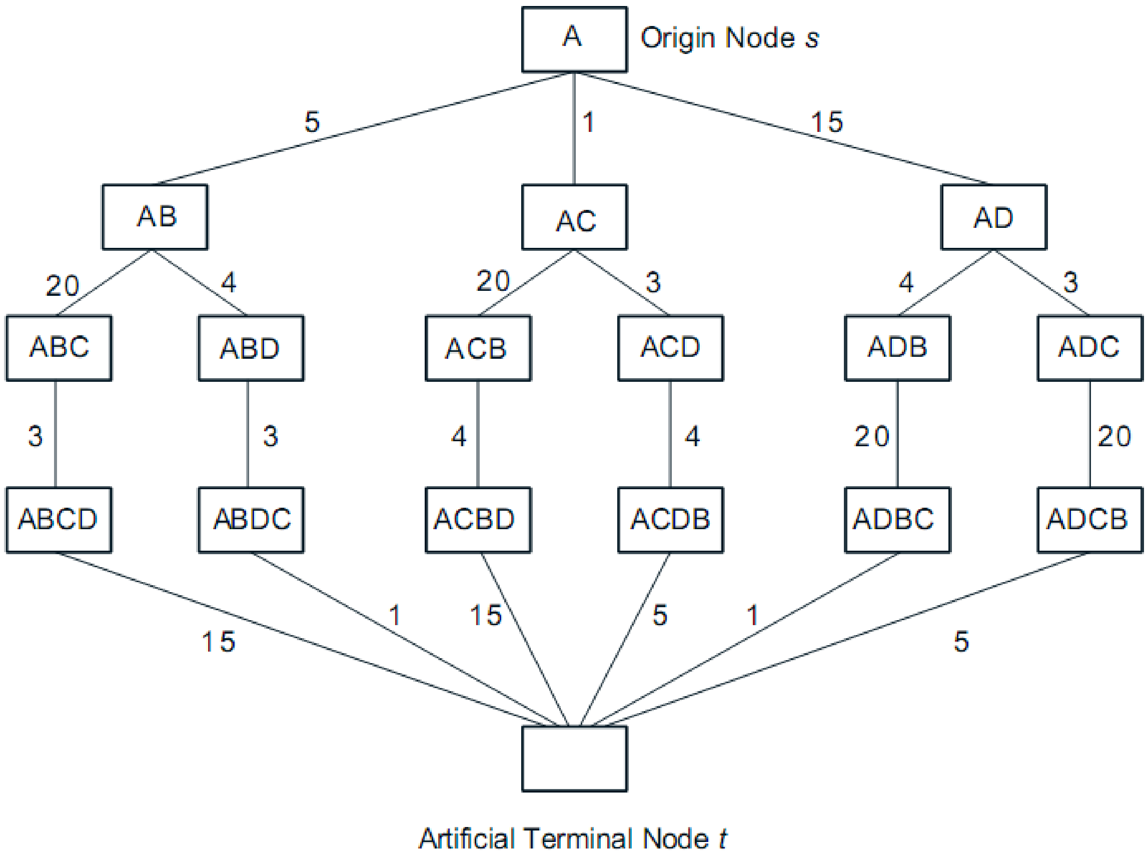


Traveling Salesman Problem (TSP)

- Visit all cities with the minimum traveling cost

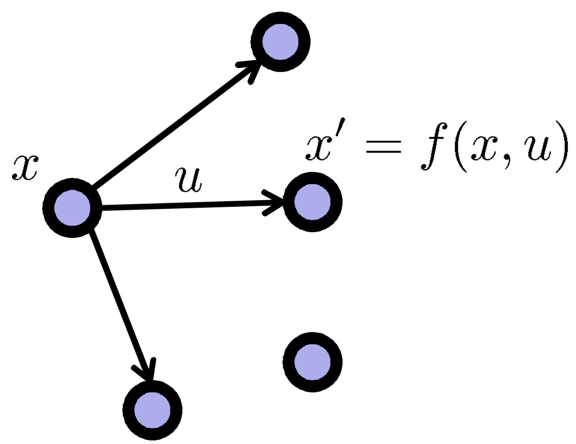
- Can pose it as a shortest path problem

	A	B	C	D
A		5	1	15
B	5		20	4
C	1	20		3
D	15	4	3	



Shortest-path Applications

- Typical search spaces for robot navigation
  - Regular grid
  - State lattice
  - PRM
- For shortest path algorithms, they are represented as graphs
  - Node/Vertex
  - Arc/Edge
- LaValle's book
  - focus on the planning aspect
  - Node: x
  - Edge connection from node: $u \in U(x)$
  - Edge cost: $l(x, u)$
  - Child node of x: $x' = f(x, u)$



Label Correcting Algorithm

- Many discrete search algorithms belong to this
- Given:
  - Origin/start/initial node: s
  - Destination/target/goal node: t
  - Edge cost from node i to node j: a_{ij} (≥ 0)
- Find:
  - The minimum cost of going from s to t
  - The path (sequence of nodes)
- Rough idea:
  - Put a **label** d_i on each node
    - d_i **: Length of the shortest path found so far from s to i ("cost-to-come")**
    - Initially, $d_i = \infty$ for all i's, except $d_s = 0$
  - Correct the label as it explores the graph

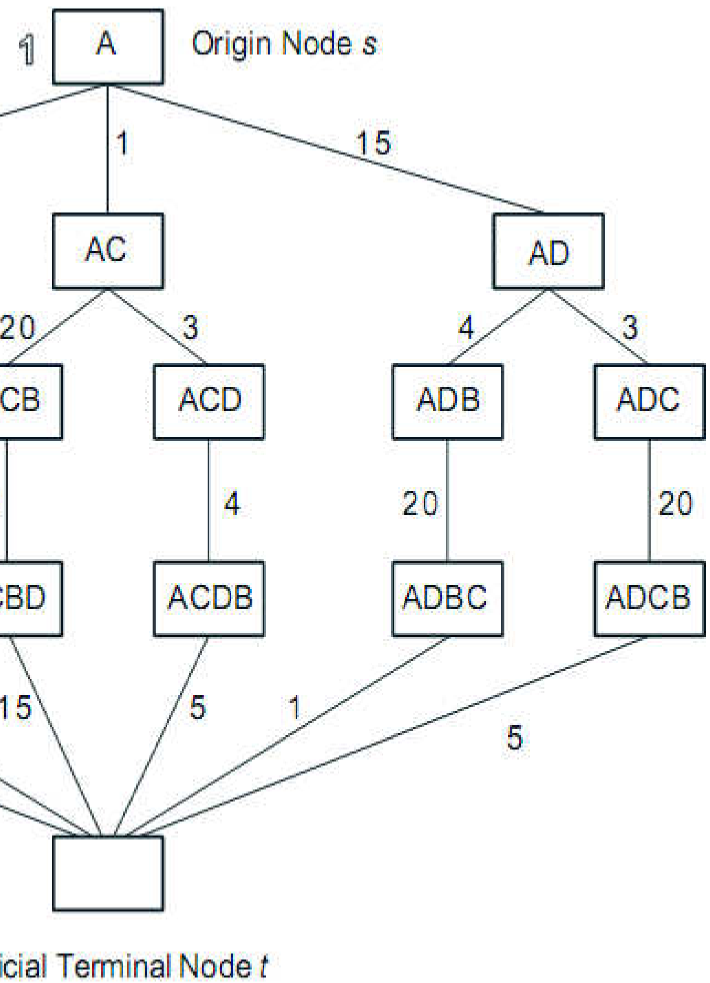
Label Correcting Algorithm

- Terminology
  - Child node: if there is an arc (i , j), then j is a child of i
  - Parent node: sometimes called "back-pointer"
  - **Open list**: contains visited nodes that are still "active" (for further examination)
- Algorithm
  - Initialize: OPEN = {s}
  1. Remove a node i from OPEN
  2. For each child j of i,
     - If $d_i+a_{ij} < \min\{d_j, d_t\}$, then set $d_j = d_i + a_{ij}$ and set i to be the parent of j.
       - Also, if $j \neq t$, place j in OPEN
  3. If OPEN is empty, terminate. Otherwise, go to step 1.

Found a better way of reaching *j* (via *i*)

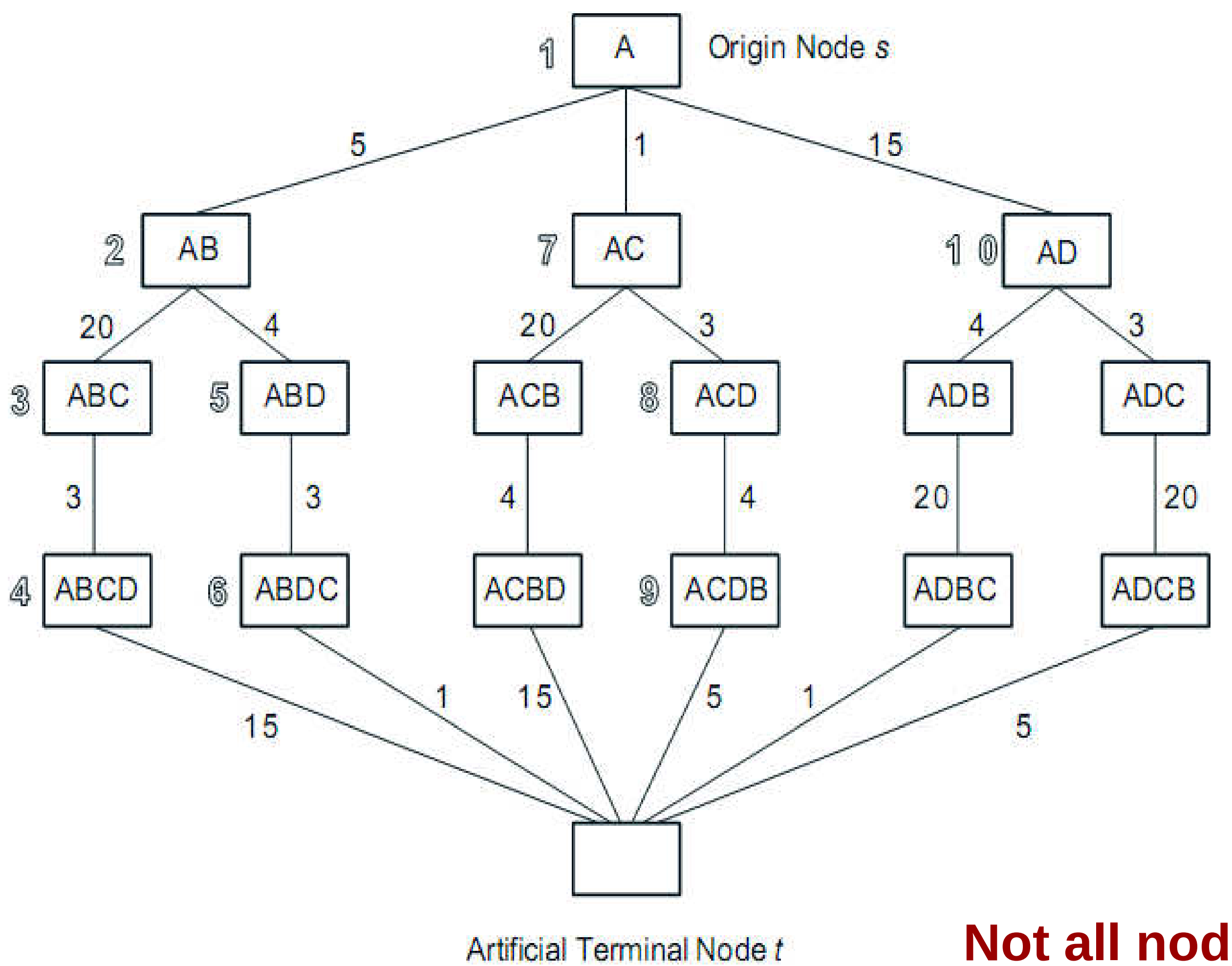
Path through *j* can improve the path to *t*

Example: 4x4 TSP



Node exiting OPEN	OPEN after iteration	d_t
-	1	∞

Example: 4x4 TSP



Node exiting OPEN	OPEN after iteration	d_t
-	1	∞
1	2, 7,10	∞
2	3, 5, 7, 10	∞
3	4, 5, 7, 10	∞
4	5, 7, 10	43
5	6, 7, 10	43
6	7, 10	13
7	8, 10	13
8	9, 10	13
9	10	13
10	Empty	13

Not all nodes are examined!

Properties

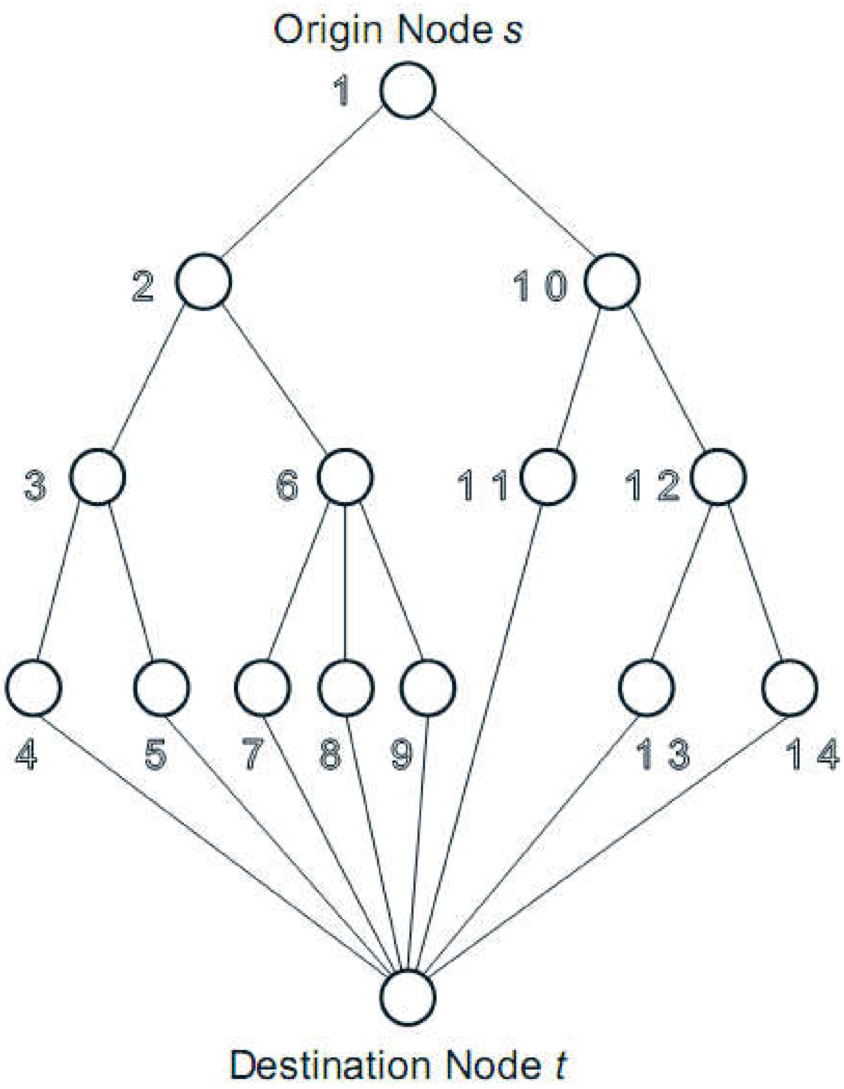
- "If there exists at least one path from the origin to the destination, the algorithm terminates with d_t equal to the shortest distance from the origin to the destination"
- The algorithm is called "*complete*"
  - Guaranteed to find a solution (in finite time) when there is one
  - Related terms
    - **Resolution complete**: if a solution exists at the resolution, it will find it. Otherwise, the algorithm could run forever
    - **Probabilistically complete**: probability of finding a solution converges to 1 with enough points
- The algorithm is called "*optimal*"
  - Guaranteed to find an optimal solution

Different Node Selection Methods

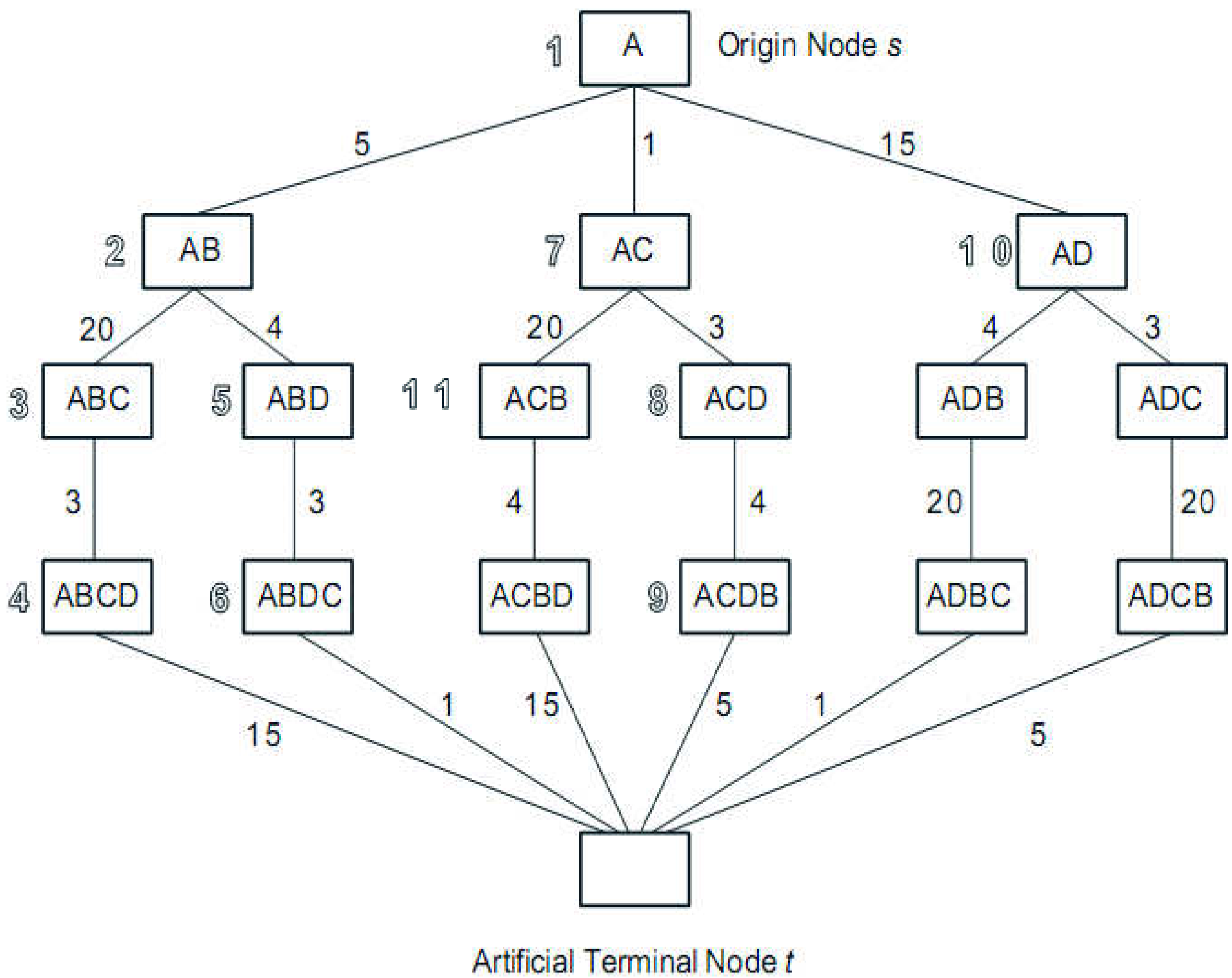
- Various strategies in step 1: Remove a node i from OPEN
- Breadth-first search (a.k.a. Bellman-Ford method)
  - First-in First out ("queue")
  - Run time O(|V|+|E|)
- Depth-first search
  - Last-in First out ("stack")
  - Requires relatively little memory
  - Run time O(|V|+|E|)
- Dijkstra's algorithm (1959)
  - Fewer the nodes enter OPEN, faster the search would be
  - Choose a node with minimum value of label: $i = \operatorname{argmin}_{j \text{ in OPEN}} d_j$
    - This "min" operation could get computationally expensive for large graphs
  - Property: a node will enter OPEN at most once
  - Run time O(|V|ln|V|+|E|) using Fibonacci heap

Example: Depth-first Search (LIFO)

- Open list
  - Initial: {1}
  - Remove 1, add 2 & 10: {2, 10}
  - Remove 2, add 3 & 6: {10, 3, 6}
  - Remove 3, add 4 & 5: {10, 6, 4, 5}
  - Remove 4: {10, 6, 5}
  - Remove 5: {10, 6}
  - Remove 6, add 7,8,9: {10, 7,8,9}
  - Remove 7: {10, 8, 9}
  - Remove 8: {10, 9}
  - Remove 9: {10}
  - Remove 10, add 11 & 12: {11, 12}
  - Remove 11: {12}
  - Remove 12, add 13 & 14: {13, 14}
  - Remove 13: {14}
  - Remove 14: {}

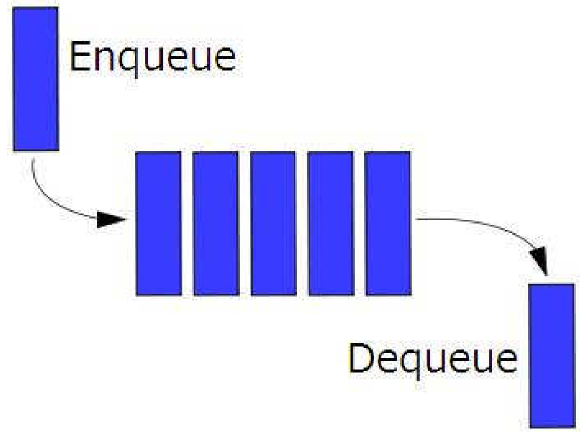


Example: Dijkstra's algorithm

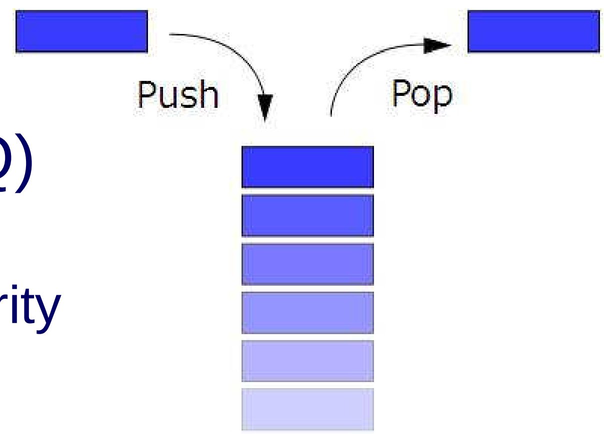


Node exiting OPEN	OPEN ID (d_i)	d_t
-	1(0)	∞
1	2(5), 7(1), 10(15)	∞
7	8(4),11(21), 2(5),10(15)	∞
8	9(8),11(21), 2(5),10(15)	∞
2	3(25), 5(9), 9(8),11(21),10(15)	∞
9	3(25), 5(9), 11(21),10(15)	13
5	6(12), 3(25), 11(21),10(15)	13
6	3(25), 11(21),10(15)	13
10	3(25), 11(21)	13
11	3(25)	13
12	Empty	13

Implementation of OPEN



- FIFO → Queue
  - "enqueue": insert the item at the bottom
  - "dequeue": remove the item at the top
- LIFO → Stack
  - "push": insert the item at the top
  - "pop": remove the item at the top

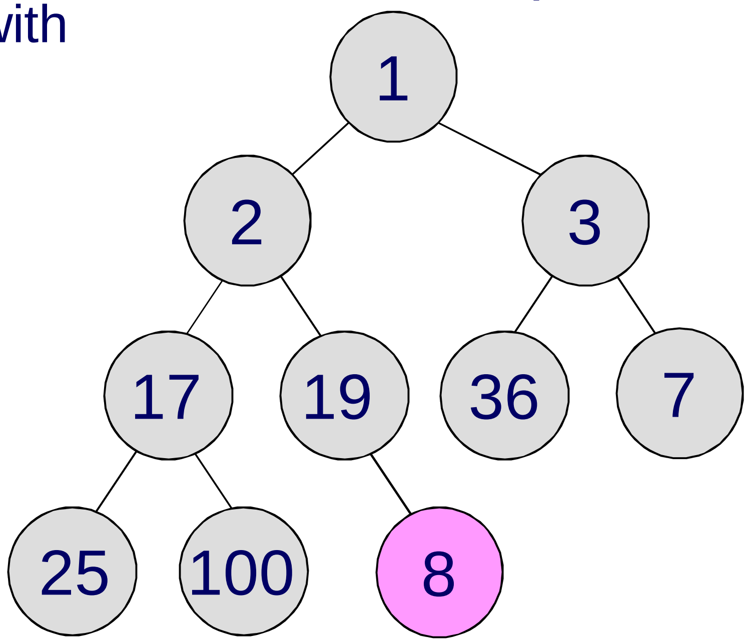


- Dijkstra → Priority queue (denoted as Q)
  - "push": insert the item with some priority
  - "pop": remove the item with the highest priority
  - Various data structures
    - Linear array: O(n) for insert, O(1) for removal
    - Binary heap: O(log n) for insert & removal
    - Fibonacci heap: O(1) for insert, O(log n) for removal. Most efficient.

Priority Queue as a Binary Heap

- "push" – add an element
  1. Add on the bottom level of the heap
  2. Compare the added element with its parent; if they are in the correct order, stop.
  3. If not, swap the element with its parent and go to step 2

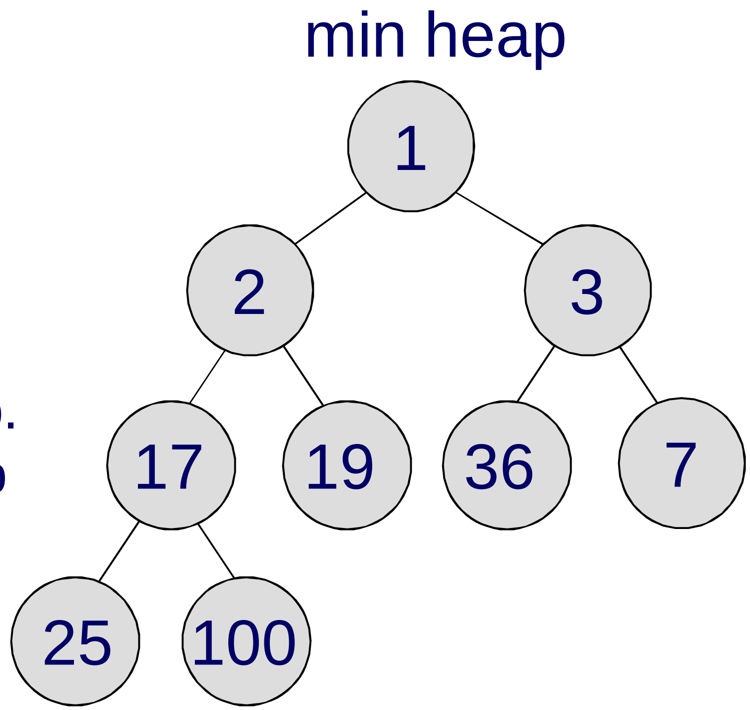
min heap



[wikipedia]

Priority Queue as a Binary Heap

- "pop" – delete a root
  1. Replace the root of the heap with the last element on the last level.
  2. Compare the new root with its children; if they are in the correct order, stop.
  3. If not, swap the element with one of its children and return to the previous step. (swap w/ its smaller child in a min-heap and its larger child in a max-heap.)



LaValle's book

FORWARD_LABEL_CORRECTING(x_G)

Set $C(x) = \infty$ for all $x \neq x_I$, and set $C(x_I) = 0$
$Q.Insert(x_I)$
while Q not empty **do**
  $x \leftarrow Q.GetFirst()$
  forall $u \in U(x)$
    $x' \leftarrow f(x, u)$
    if $C(x) + l(x, u) < \min\{C(x'), C(x_G)\}$ **then**
      $C(x') \leftarrow C(x) + l(x, u)$
      if $x' \neq x_G$ **then**
        $Q.Insert(x')$

- Other notations to note
  - Unvisited
  - Closed (Dead)
  - Open (Alive)

Extensions of
Label Correcting Algorithm

Better Test to Add a Node to OPEN

- Step 2:

  "If $d_i + a_{ij} < \min\{d_j, d_t\}$, then set $d_j = d_i + a_{ij}$ and place j in OPEN"

- Can make this test tighter
- If **a lower bound h_j** of the true shortest distance from j to t (i.e., an underestimate of cost-to-go) is known
  - **"If $d_i + a_{ij} < \min\{d_j, d_t\}$" ➔ "If $d_i + a_{ij} < d_j$ and $d_i + a_{ij} + h_j < d_t$"**

    The path going through *i* and *j* can improve the cost of reaching *t*
  - Called **A* algorithm** (1968). Very popular
  - h: is sometimes called "heuristics function"
    - Neglect the structure of the regular grid: 2-norm distance to target
    - Obstacle-free path length: Dubin's distance
    - If $h_i = 0$ (loosest lower bound), A* reduces to Dijkstra
  - Choose a node with minimum value of estimated cost:

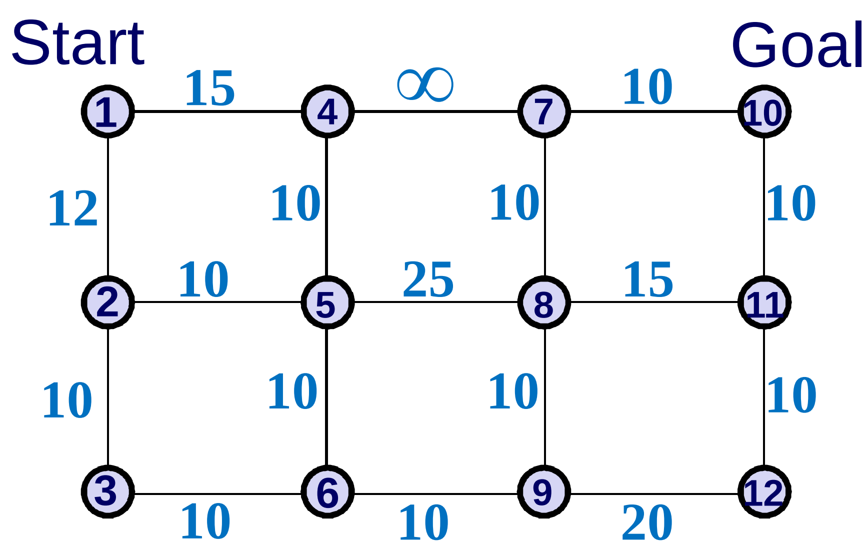
    $$i = \underset{j \text{ in OPEN}}{\operatorname{argmin}} (d_j + h_j)$$
  - In general much fewer nodes to expand compared to Dijkstra

Some Notes on A* algorithm

- Other notations
  - f_j: $g_j + h_j$
  - g_j: distance from s to j (the label d_j in the label correcting algorithm)
  - h_j: heuristic value from j to t
  - Then, use f_j in sort the ndoes
- Sometimes called "informed search" as opposed to "uninformed search" in AI
- "Optimally efficient"
  - For any given heuristic function, A* expands the fewest nodes of any admissible search algorithm
- Heuristic function
  - Admissible: $h_i \leq h_i^*$ (underestimates the cost-to-go)
  - c.f. Consistent: $h_i \leq a_{ij} + h_j$ (go incrementally without going back)
  - If consistent, then admissible

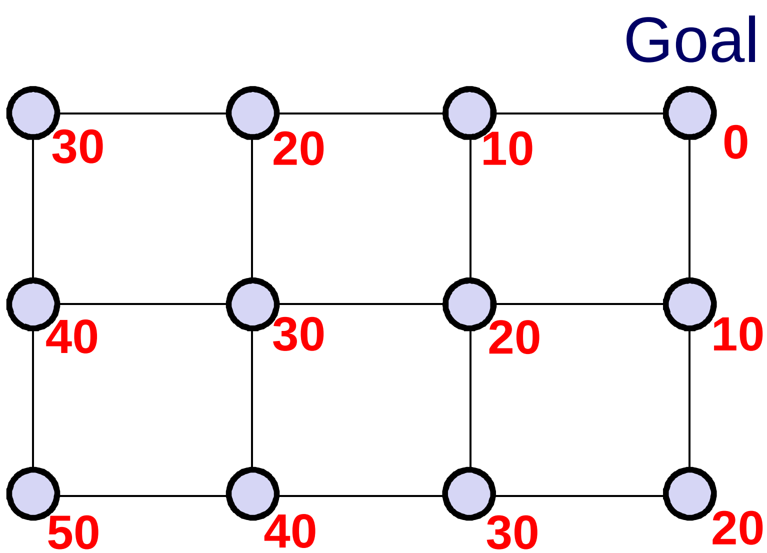
A* Example: 4-connected grid

- Grid of size 3 x 4
- Start at node #1, goal at node #10
- Physical distance of each edge is 10
- **Edge cost** = distance + some terrain penalty

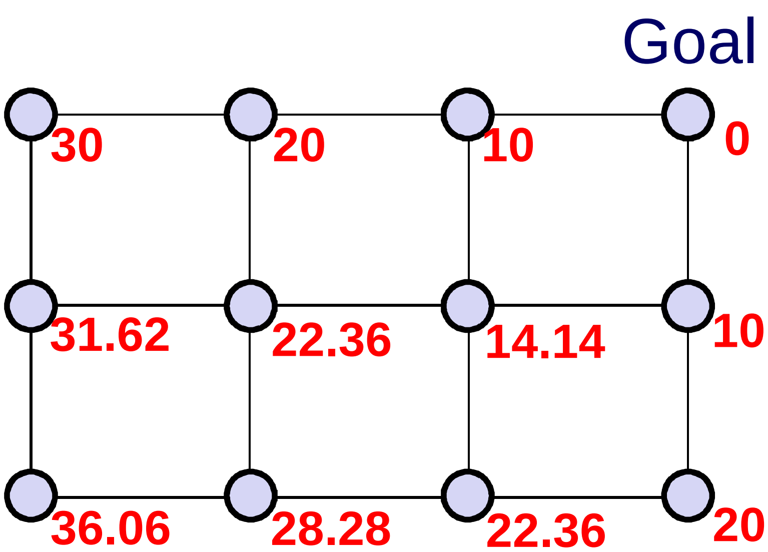


A* Example: 4-connected grid

- Physical distance of each edge is 10
- Different heuristics
  - Manhattan vs Euclidean distance
    → which one is better & why?

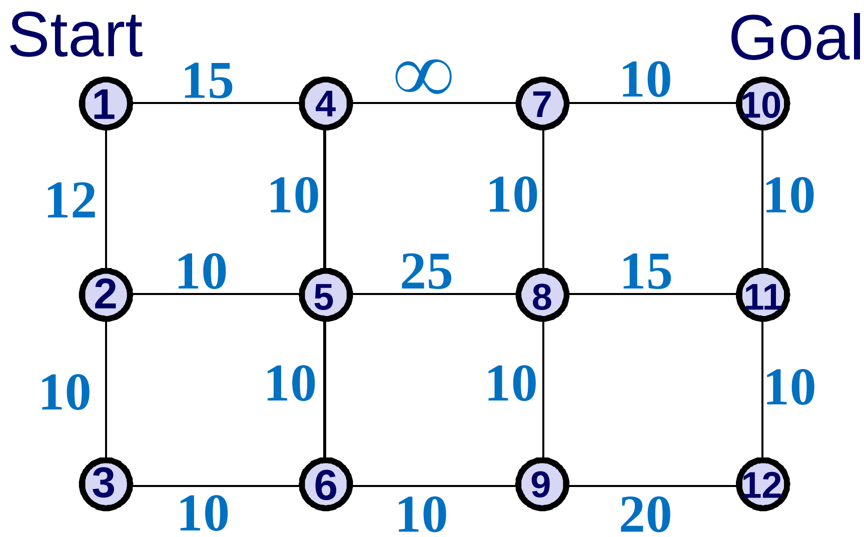


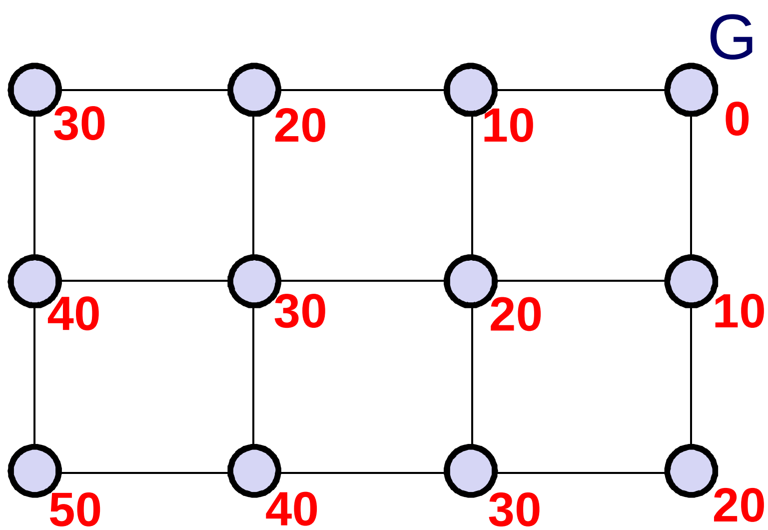
Manhattan distance



Euclidean distance

A* Example: 4-connected grid





Heuristic value

Node exiting OPEN	OPEN # (d, d+h)	d_t

Other Improvements

- Advanced initialization
  - Normally labels are initialized as "$d_i = \infty$ for all i's, except $d_s = 0$"
  - If there is some good starting path (obtained heuristically), initialize the labels d_i with length of some path from s to i (so that $d_i < \infty$).
  - The test "$d_i + a_{ij} < \min\{d_j, d_t\}$" of adding nodes to OPEN becomes tighter ➔ fewer nodes would enter OPEN
- Upper bound
  - If an upper bound m_j of the cost-to-go (j to t) is known, then, reduce dt faster. When $d_j + m_j < d_t$, then $d_t := d_j + m_j$
- Bidirectional planning
  - Start the search from start and the target at the same time
  - Terminate when they "meet" in the middle with some conditions
- Incremental version (next lecture)
  - Do not start from scratch when a small part of the environment changes.