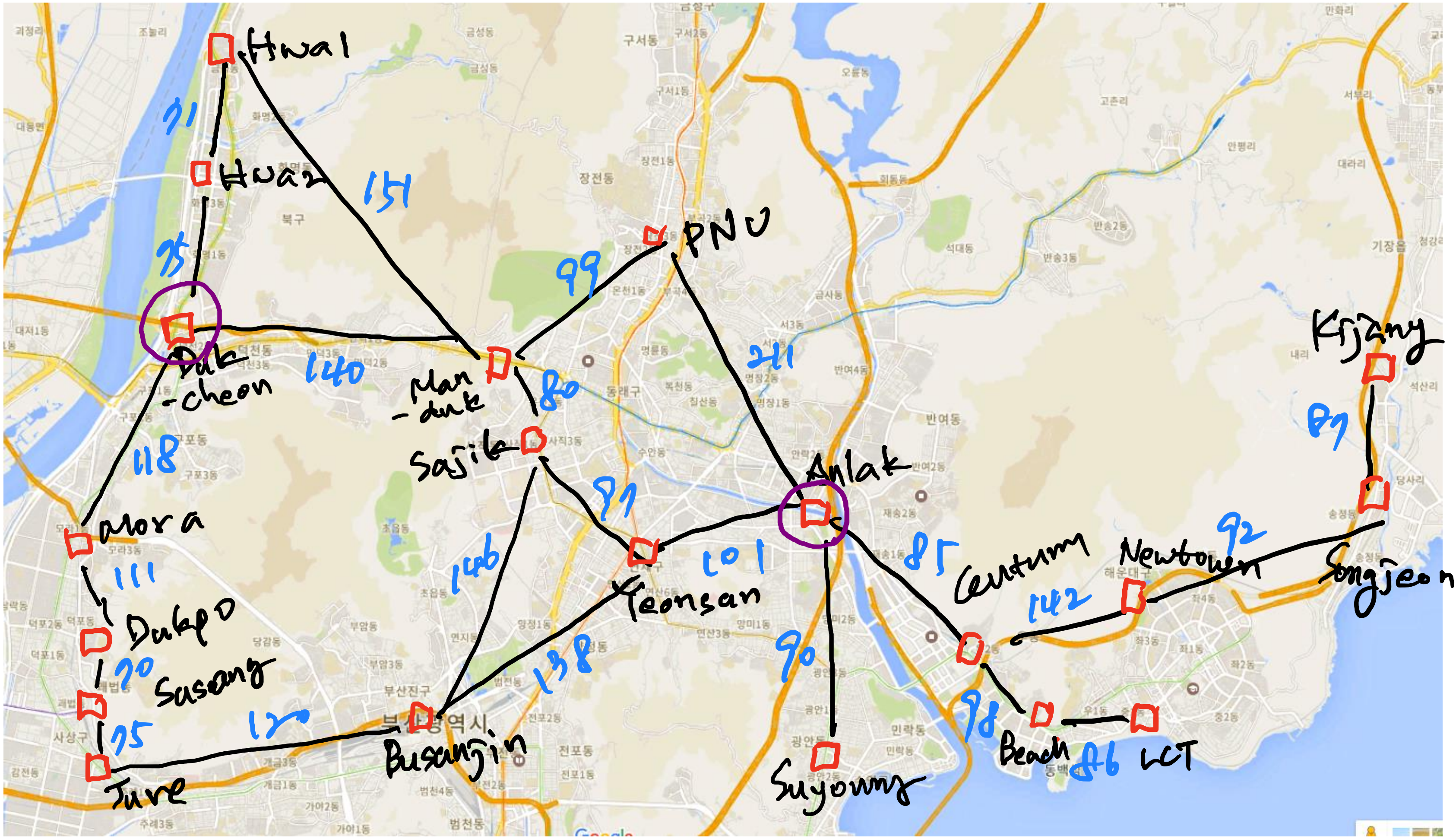
Search (I)

Lecture by Hyerim Bae, PhD.

Routing Finding



- Definition
  - Dictionary
    - ‘look for something whose presence is suspected’
  - In AI
    - a goal
    - The whole process of searching is usually presented as a number of steps that describe a connection between a start point and the goal.
- Problem description

  $\prod ::= (S, O, I, G)$
  - S: Domain state space
  - O: Operator on states
  - I: Initial state
  - G: Goal state
  - *cf*) cost

Traveling Salesperson::=(S, O, I, G)

$S=\{(M, P, C)\}$

$M=(N, \alpha)$ is the map

P is a path (may or may not be a circuit)

C=is a cost function defined on paths

O = create (M, P', C') as a sucessor of (M, P, C) by $P' \leftarrow P \cup \{n'\}$ where $n' \notin P$ and $(n, n') \in \alpha$ and $C' \leftarrow C + C(n')$

$I = \{(M, (n_i), 0) | n_i \in N\}$

$G=\{(M, P, C) \mid P$ is a circuit and $C(P)$ minimal$\}$

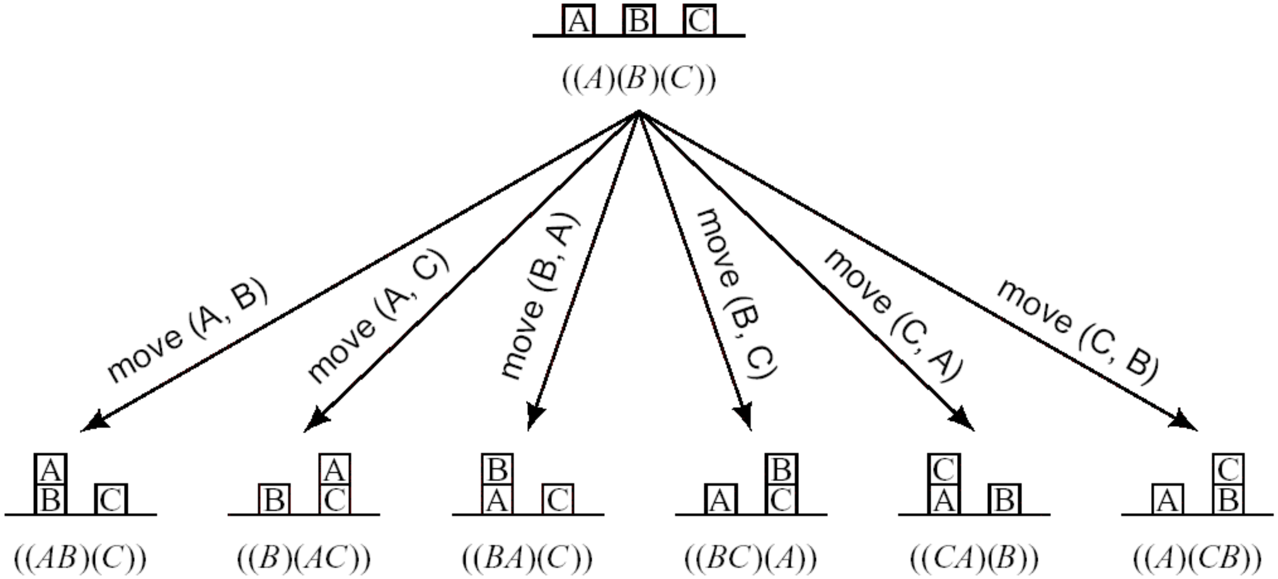
Agents That Plan

Memory vs. Computation

- Memory-based implementation
  - A competent reactive machine
    - Agent would require large amounts of memory.
    - Designer would require foresight in anticipating appropriate reactions for all possible situations.
- Computation-based implementation
  - It will reduce the agent’s memory requirements and the burden on the designer.
  - The designer specifies the computation instead of all possible situations.
  - These computations will take time
- Computations
  - Predict the consequences of the actions possible in any given situation.
  - If these consequence-predicting computations could be automatically learned or evolved, the agent would be able to select appropriate actions even in those environments that a designer might not have been able to foresee.
  - An agent must have a model of the world it inhabits and models of the effects of its actions on its model of the world.

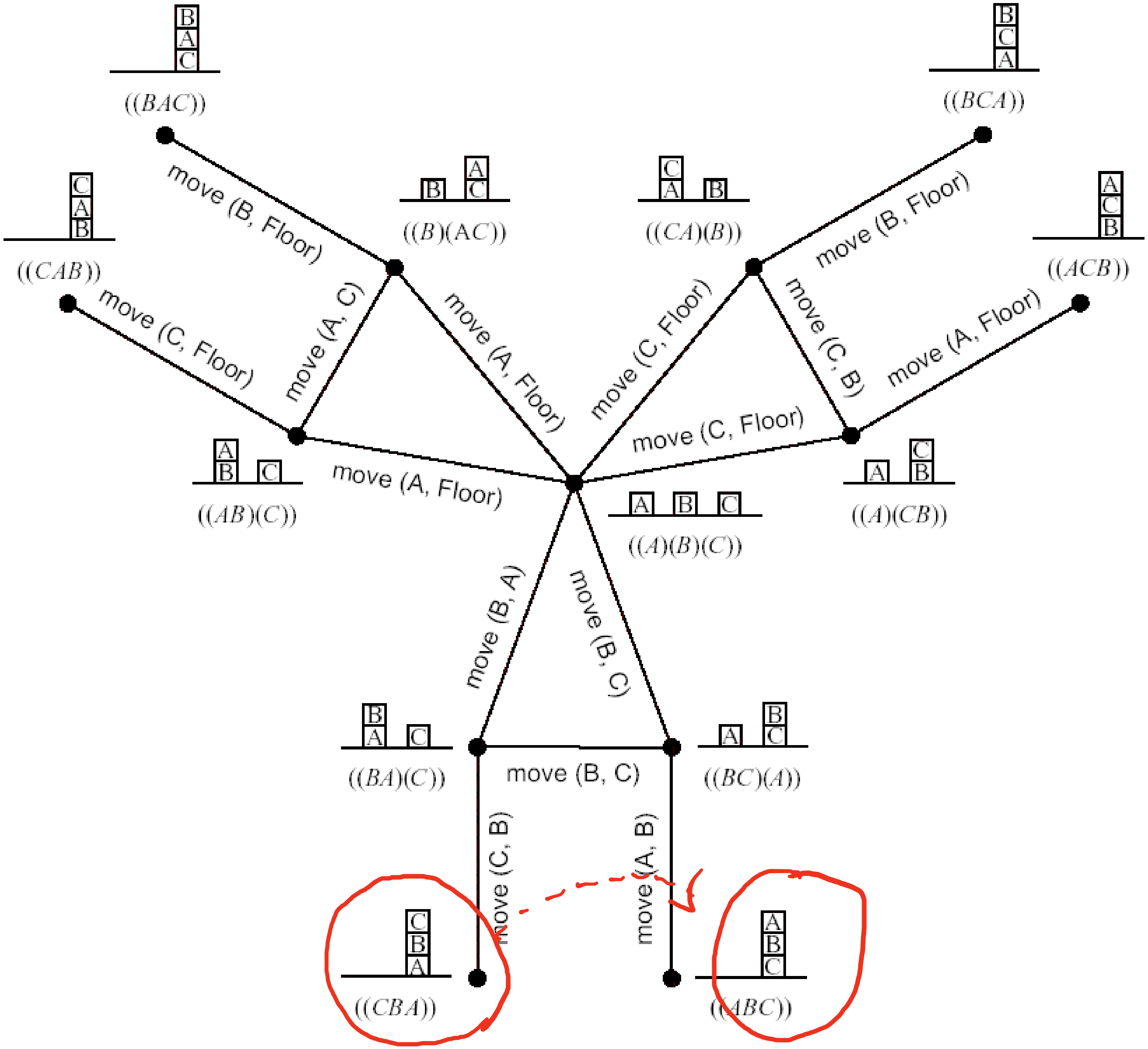
State-space graph

- An example
  - Grid-space world containing 3 toy blocks (A, B, C).
  - All initially are on the floor.
  - The task is to stack blocks so that A is on top of B and B is on top of C and C is on the floor.
  - Instances of a schema
    - move(x, y) – x can be A, B, C and y can be A, B, C and floor.
  - Operators
    - move(A, C) , move(A, B) …



- Directed graph
  - A most useful structure for keeping track of the effects of several alternative sequences of actions.
  - Node
    - Representations of the individual worlds
    - Iconic or feature
  - Arc
    - Operators
- State-space graph
  - A graph representing all of the possible actions and situations

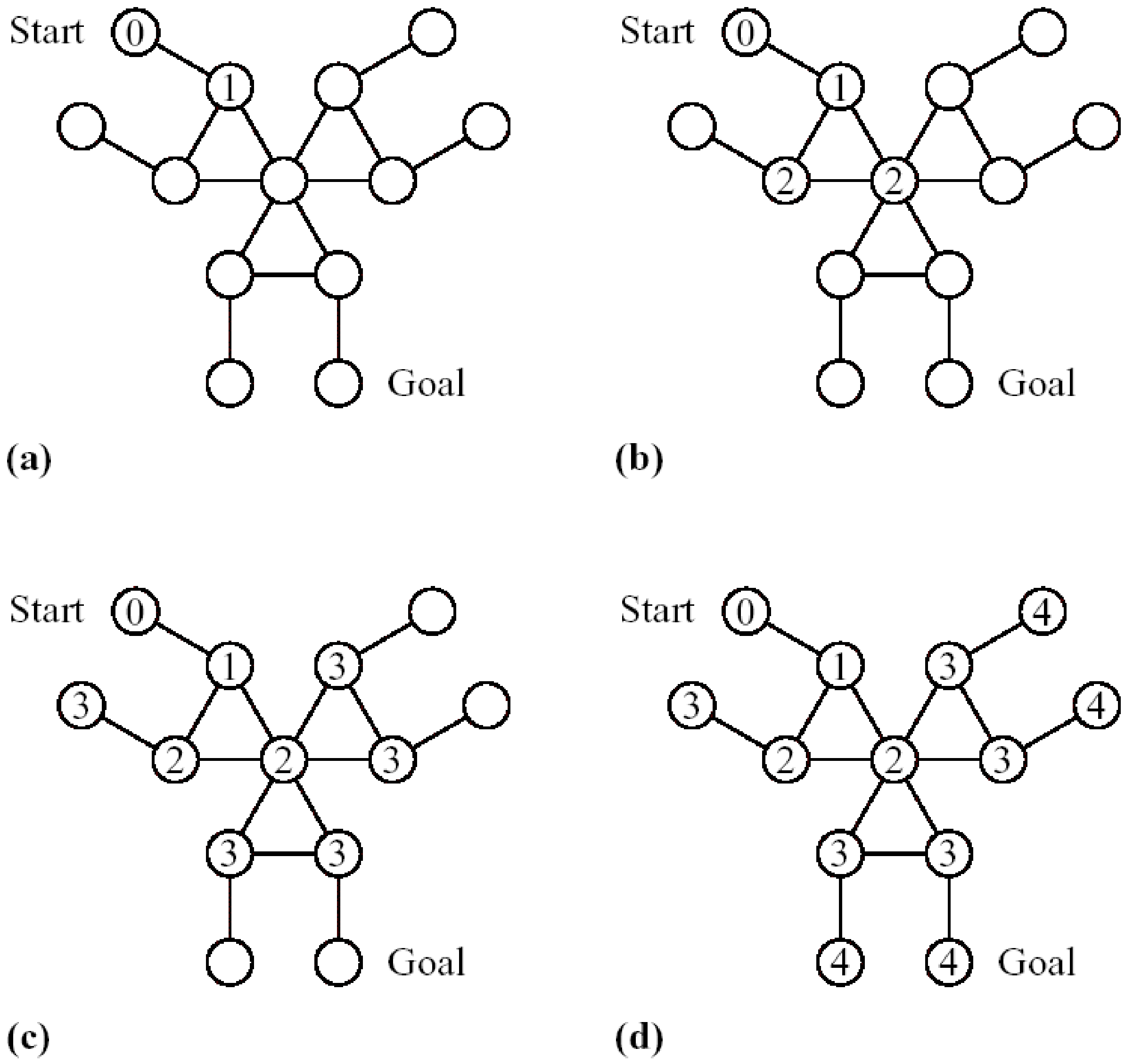
- How should it move?
  - Goal: (A,B,C)
  - Initial: (C, B, A)



- Plan
  - A sequence of the operators labeling the arcs along a path to a goal.
  - Planning is searching for such a sequence.
- Projecting
  - The process of predicting a sequence of world states resulting from a sequence of actions.
- Assumptions
  - Must be able to represent all of the relevant world situations by nodes
  - Can be no slipup or uncertainty in the agent’s effecter system
  - Must accurately identify the start node
  - Can be no other agents or dynamic processes that change the world

Searching explicit state spaces

- Search methods for explicit graphs
  - involve propagating markers over the nodes of the graph.
  - Label the start node with a 0.
  - Propagate successively larger integers out in waves along the arcs until an integer hits the goal.
  - Trace a path back from the goal to the start along a decreasing sequence of numbers.
  - Requires $O(n)$ steps
- Expansion
  - Puts marks on all of the marked node’s unmarked neighbors.
    - Which marked node should be expanded?
    - Breath-first search


Start
0
1
2
2
Goal
Start
0
1
3
3
2
2
3
3
3
Goal
Start
0
4
1
3
3
4
2
2
3
3
3
4
4
Goal

(a)

(b)

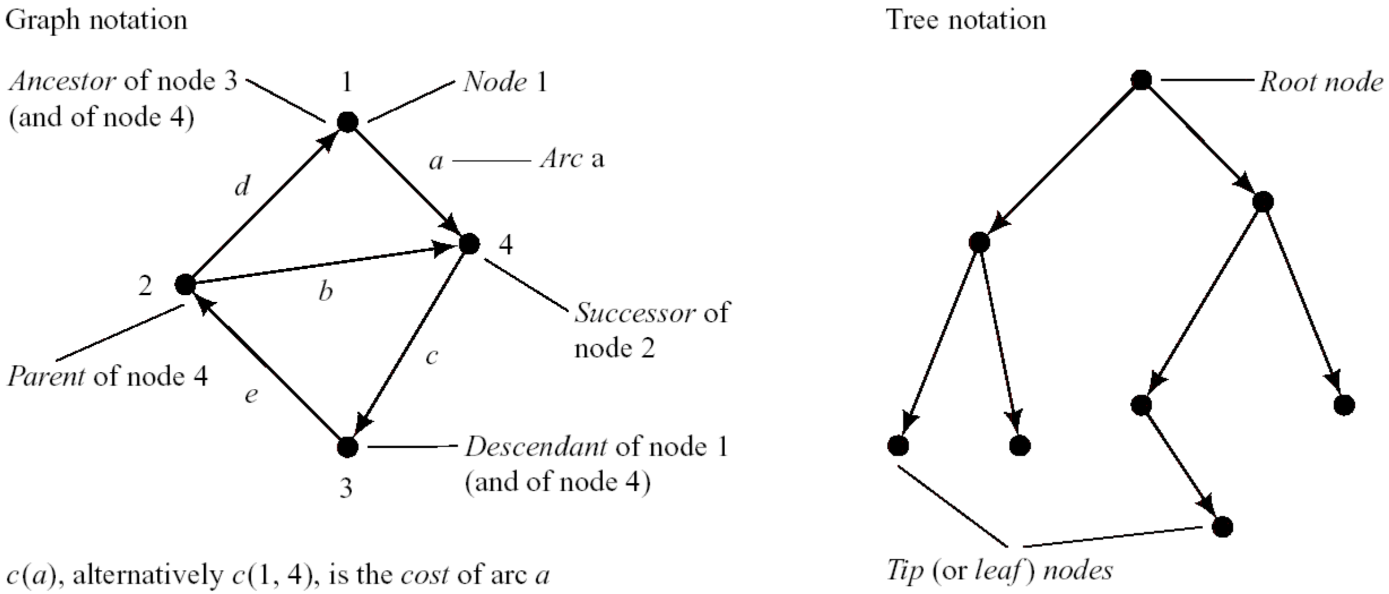
(c)

(d)

Feature-based state spaces

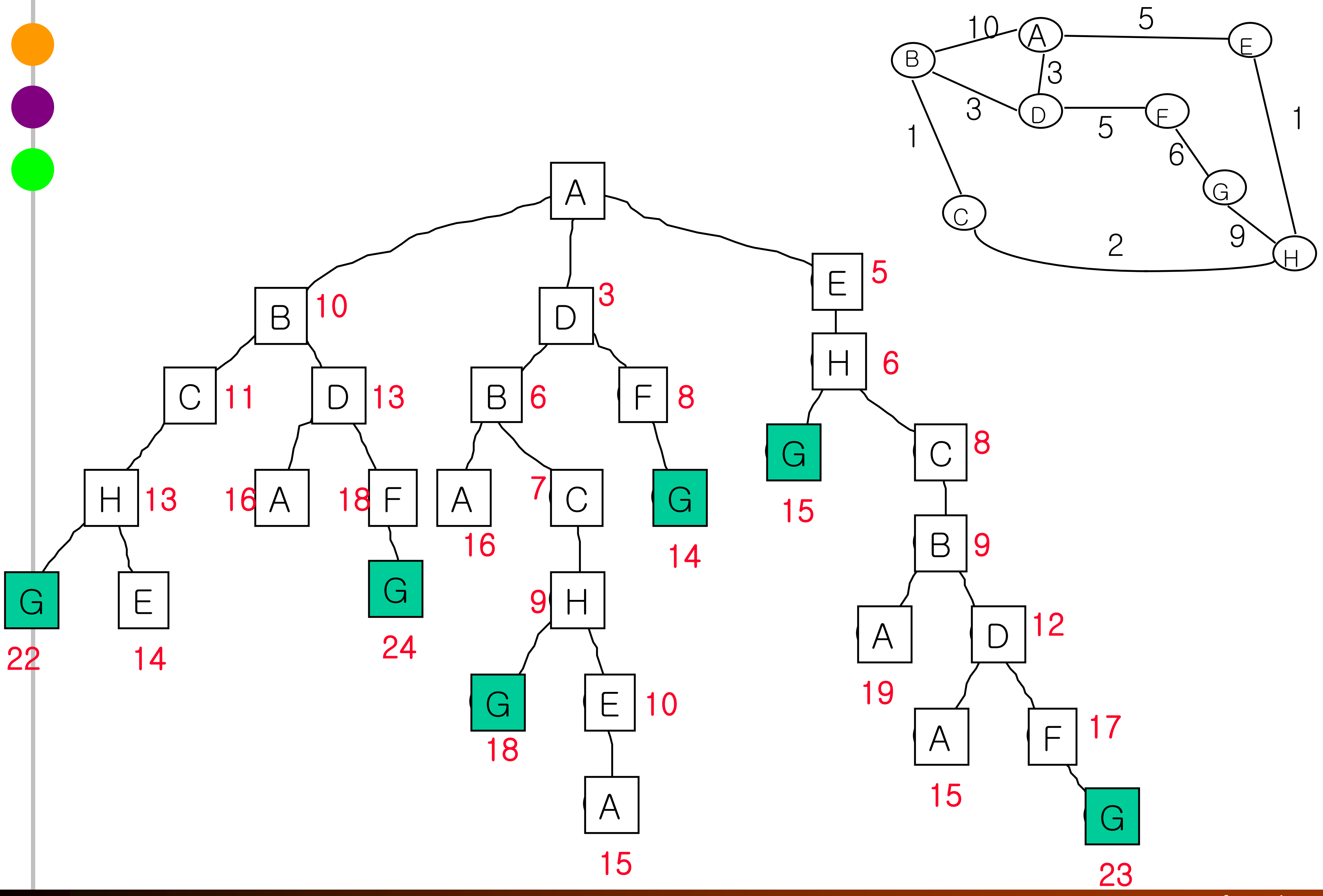
- Feature-Based graphs need a way to describe how an action affects features.
  - STRIPS [Fikes & Nilsson, 1971]
    - Define an operator by 3 lists
    - Precondition list specifies those features that must have value 1 and 0 in order that the action can be applied.
    - Delete list specifies those features that will have their values changed from 1 to 0.
    - Add list specifies those features that will have their values changed from 0 to 1
  - Neural Networks
    - Train a neural network to learn to predict the value of a feature vector at time *t* from its value at time *t*-1 and the action taken at time *t*-1. [Jordan & Rumelhart 1992]
    - After training, the prediction network can be used to compute the feature vectors that would result from various actions.
    - Computed features in turn could be used as new inputs to the network to predict the feature vector two steps ahead, and so on.

- A graph consists of a set of nodes.
  - Arcs connect certain pairs of nodes.
  - A directed graph
    - Arcs are directed from one member of the pair to the other.
    - Successor (child)
    - Parent
  - An undirected graph
    - Edges are undirected arcs.
    - Contain only edges.



c(*a*), alternatively *c*(1, 4), is the *cost* of arc *a*
(*d*, *a*), alternatively (2, 1, 4), is a *path* from node 2 to node 4

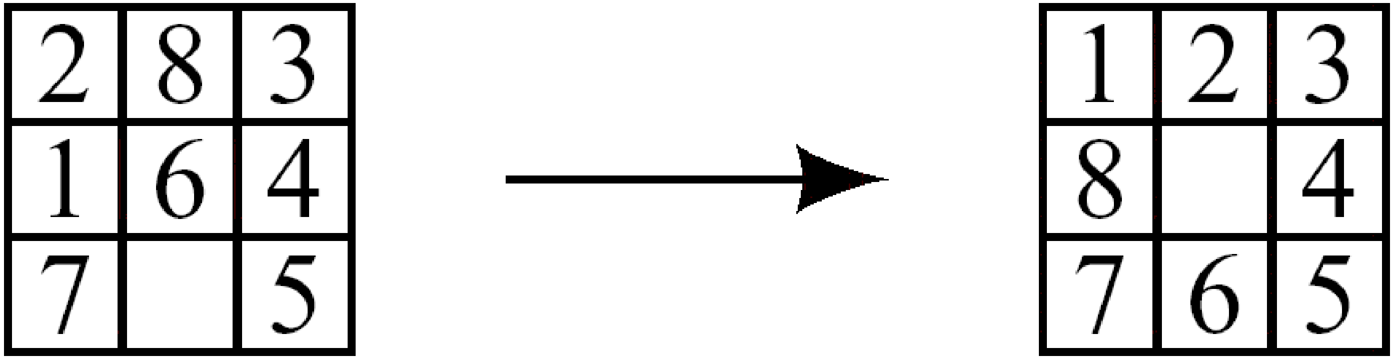
- Tree is a special case of a graph.
  - A directed tree
    - A root node has no parent.
    - Each has exactly one parent except root.
    - A leaf node has no successors.
    - Depth of any node is defined to be the depth of its parent plus 1. (The root node is of depth zero.)
  - An undirected tree
    - There is only one path along edges between any pair of nodes.
  - Branching factor
    - All nodes have the same number (b)
- A path of length *k* from node n_1 to node n_k.
  - A sequence of nodes with each n_{i+1} a successor of n_i for i=1,…,k-1
- Accessible
  - Exist a path from one node to other node.
- Descendent, Ancestor
- Optimal path
  - Path having minimal cost between two nodes.
- A spanning tree
  - A tree including all nodes in a graph


B
A
E
D
F
C
G
H
10
5
3
3
5
1
1
6
9
2
A
B 10
D 3
E 5
C 11
D 13
B 6
F 8
H 6
H 13
16 A
18 F
A
7 C
G
G
C 8
G
E
G
16
14
15
B 9
22
14
24
9 H
A
D 12
G
E 10
19
18
A
F 17
A
15
G
15
23

Uninformed Search

- Many problems of practical interest
  - Have large search spaces
  - Cannot be represented by explicit graphs
    - Some elaborations are required
  - Elaboration
    - How we formulate search problem
    - Have methods for representing large search graphs implicitly
    - Use efficient methods for searching large graphs
  - Setting up problems for search
    - Art that still requires human participation

- e.g.



2	8	3
1	6	4
7		5

1	2	3
8		4
7	6	5

- state description
  - 3-by-3 array: each cell contains one of 1-8 or blank symbol
- two state transition descriptions
  - 8×4 moves: one of 1-8 numbers moves up, down, right, or left
  - 4 moves: blank symbol moves up, down, right, or left
- How many states are there?
- State space for 8-puzzle is
  - Divided into two separate graphs : not reachable from each other

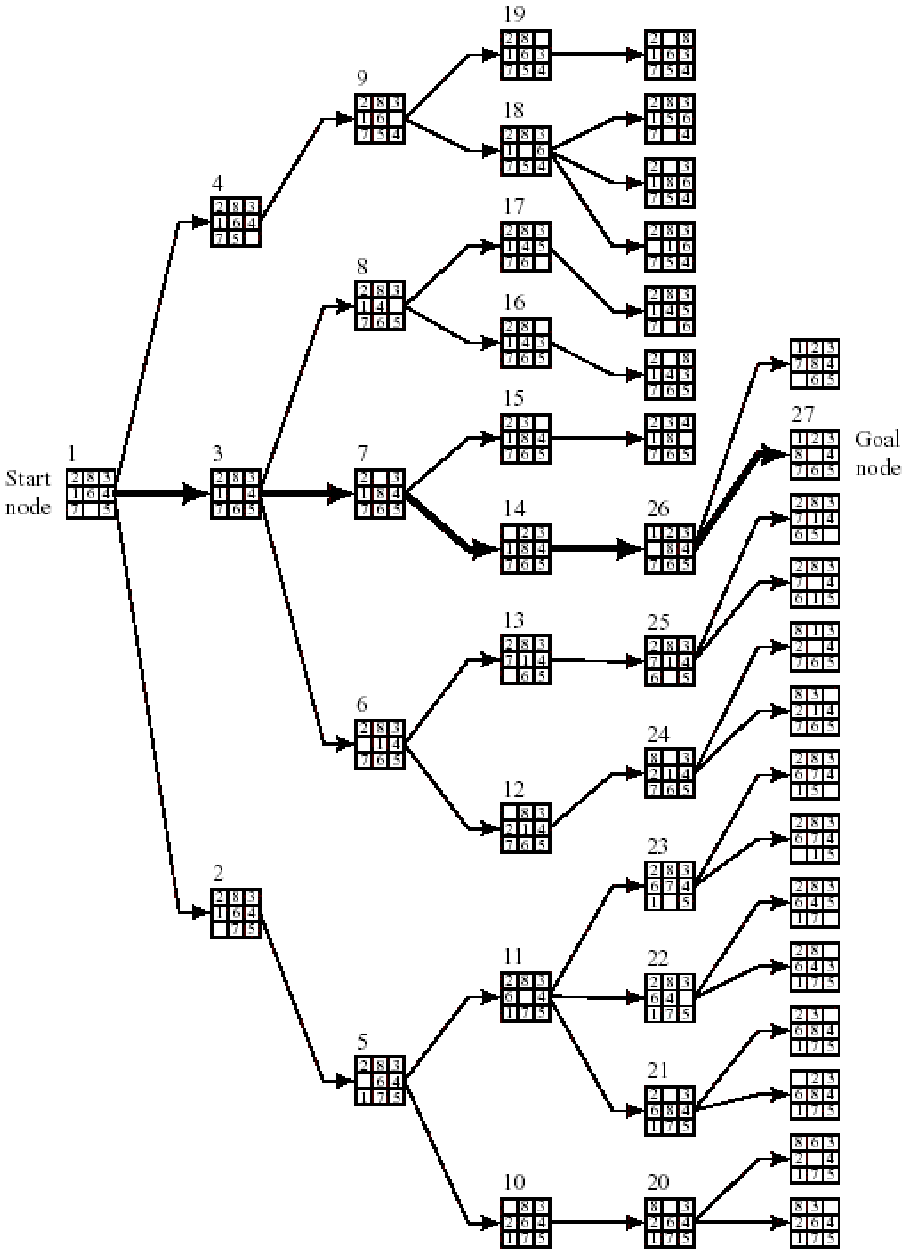
Components of Implicit State-space Graphs

- It’s possible to transform an implicit rep. to an explicit one
  - Generates all of the nodes that are successors of the start node
  - Then generates all of their successors, and so on
- 3 basic components to an implicit representation of a state-space graph
  1. Description of start node
  2. *Actions*: Functions of state transformation
  3. *Goal condition*: *true-false* valued function
- 2 classes of search process
  1.Uninformed (blind) search: no problem specific reason to prefer one to any other
  2.Heuristic search: existence of problem-specific information

Breath-First search

- BFS
  - The simplest uninformed search procedure
  - Successor function
    - Applying all possible operators to all the direct successors
  - Expanding
    - Each application of a successor function to a node
- Procedure
  1. Apply all possible operators (*successor function)* to the start node.
  2. Apply all possible operators to all the direct successors of the start node.
  3. Apply all possible operators to their successors till goad node found.

```
PROCEDURE BreadthFirstSearch(Tree tree, Node root, Goal goal, List alreadyVisited)

        Node current
        Queue toVisit

        add root to toVisit

        while toVisit is not empty
                current <- first node on toVisit
                remove first node from toVisit
                if current = goal
                        add current to alreadyVisited
                        return true
                end if
                for each child node C of current
                        add C to toVisit
                end for
                add current to alreadyVisited
        end while
```

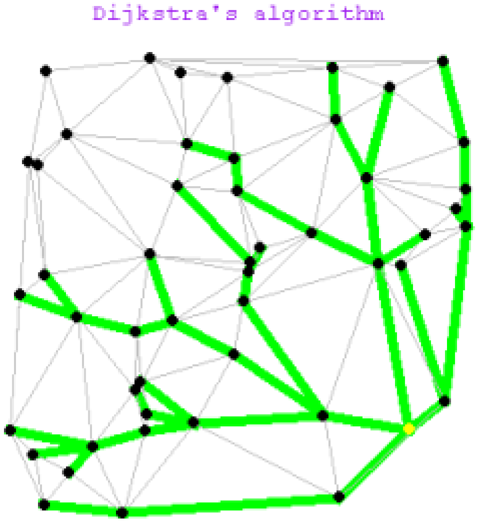

Start node
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
Goal node

- Advantage
  - Finds the path of minimal length to the goal.
- Disadvantage
  - Requires the generation and storage of a tree whose size is exponential the depth of the shallowest goal node
- *Uniform-cost* search [Dijkstra 1959]
  - Expansion by *equal cost* rather than equal depth
- Complete?
  - Yes (if *b* is finite)
- Time ? (If the goal is in depth *d*)

- Space ?
  - O(b^d)?

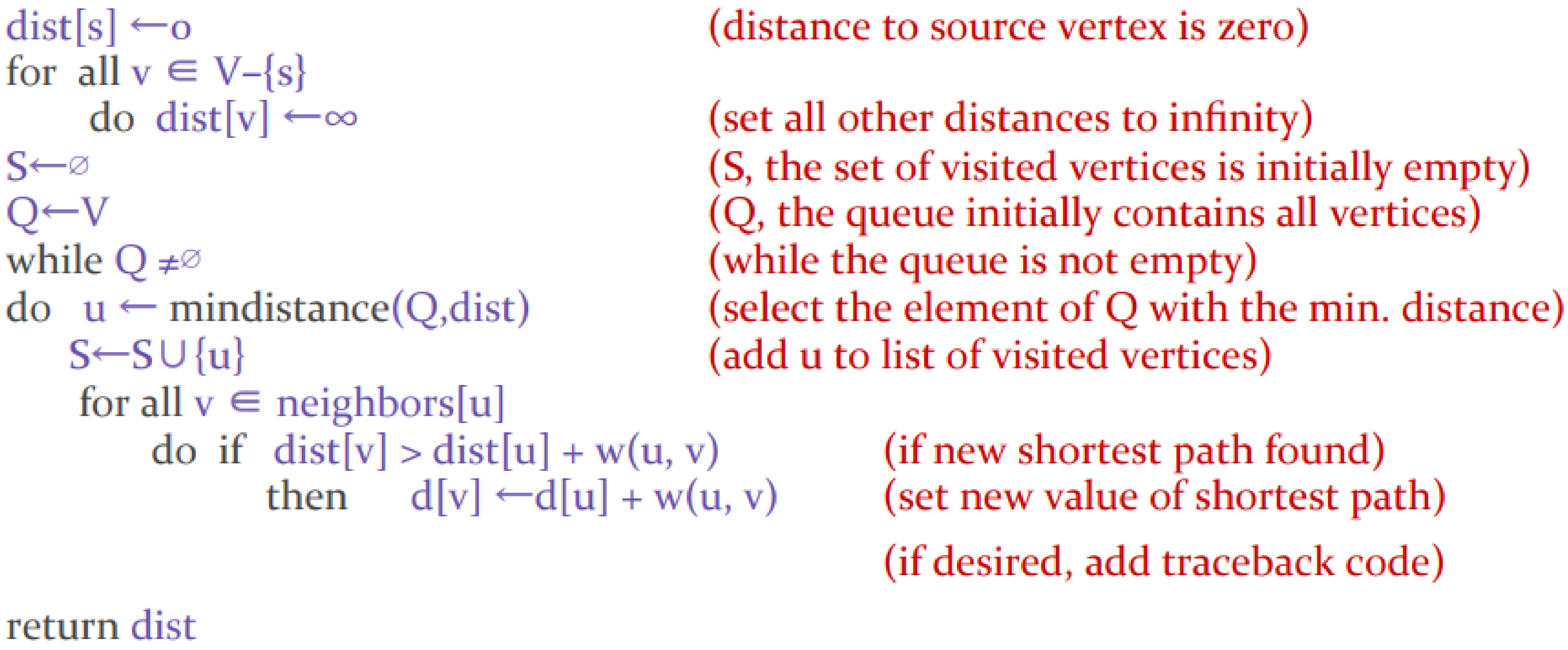
Dijkstra

- Dijkstra algorithm is algorithm for finding the shortest paths between nodes in weighted graph. For unweighted graph that all edge have same value, BFS is preferable to use because for this situation BFS has running time less than Dijkstra
- Dijkstra is conceived by computer scientist Edgar W Dijkstra
- The algorithm exist in many variants: dijkstra original variant found the shortest path between two nodes
- Weighted graph $G = (E, V)$
- Sources vertex $s \in V\ \mathit{to\ all\ vertices}\ v\ \in V$



Dijkstra

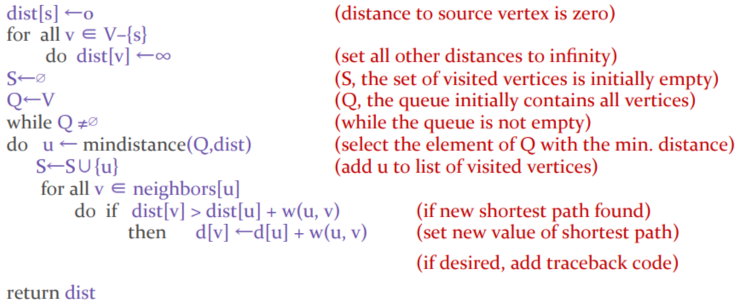
- Pseudocode of Dijkstra algorithm



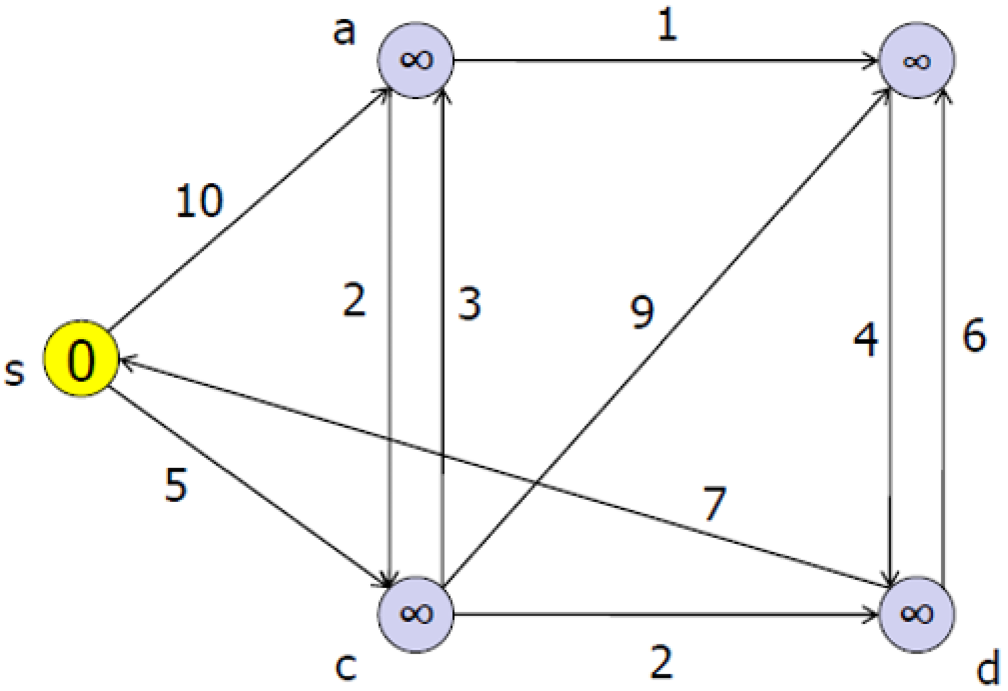
```
dist[s] ← 0                                     (distance to source vertex is zero)
for  all v ∈ V–{s}
        do  dist[v] ← ∞                         (set all other distances to infinity)
S ← ∅                                           (S, the set of visited vertices is initially empty)
Q ← V                                           (Q, the queue initially contains all vertices)
while Q ≠ ∅                                     (while the queue is not empty)
do   u ← mindistance(Q,dist)                    (select the element of Q with the min. distance)
      S ← S ∪ {u}                               (add u to list of visited vertices)
       for all v ∈ neighbors[u]
              do  if   dist[v] > dist[u] + w(u, v)         (if new shortest path found)
                       then      d[v] ← d[u] + w(u, v)     (set new value of shortest path)

                                                           (if desired, add traceback code)

return dist
```

Dijkstra

- Demo (1)



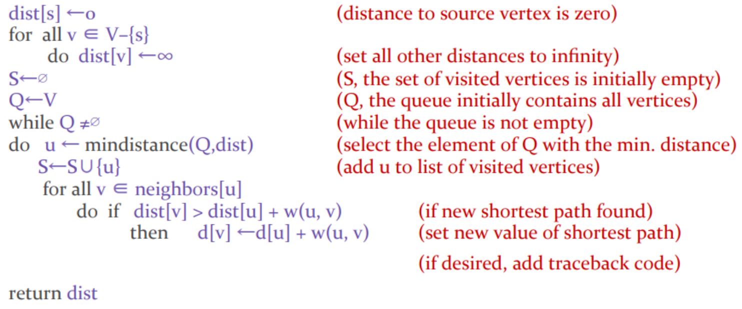
```
dist[s] ←0                                   (distance to source vertex is zero)
for  all v ∈ V–{s}
      do  dist[v] ←∞                          (set all other distances to infinity)
S←∅                                          (S, the set of visited vertices is initially empty)
Q←V                                          (Q, the queue initially contains all vertices)
while Q ≠∅                                   (while the queue is not empty)
do   u ← mindistance(Q,dist)                 (select the element of Q with the min. distance)
     S←S∪{u}                                 (add u to list of visited vertices)
     for all v ∈ neighbors[u]
          do  if   dist[v] > dist[u] + w(u, v)          (if new shortest path found)
                   then     d[v] ←d[u] + w(u, v)        (set new value of shortest path)
                                                        (if desired, add traceback code)
return dist
```



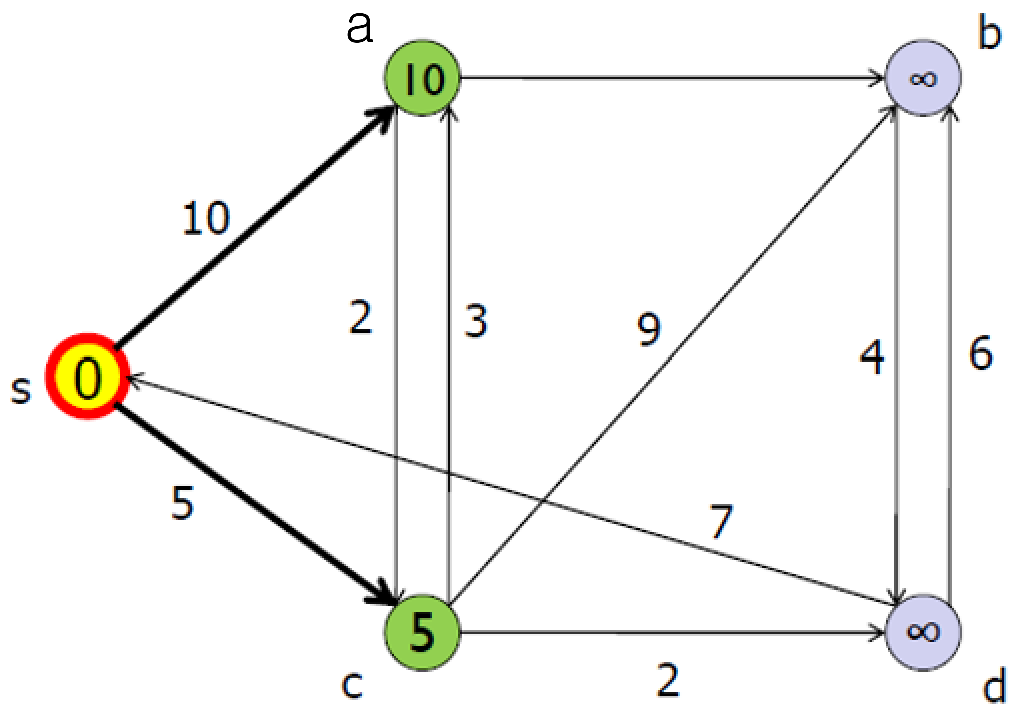
Q = {s,a,b,c,d}
S = {}
Dist_S_To = {(s,0), (a,∞), (b,∞), (c,∞), (d,∞)}
Predecessor = {(a,null), (b,null), (c,null), (d,null)}

Dijkstra

- Demo (2)



```
dist[s] ←o                                (distance to source vertex is zero)
for  all v ∈ V–{s}
      do  dist[v] ←∞                      (set all other distances to infinity)
S←∅                                       (S, the set of visited vertices is initially empty)
Q←V                                       (Q, the queue initially contains all vertices)
while Q ≠∅                                (while the queue is not empty)
do   u ← mindistance(Q,dist)              (select the element of Q with the min. distance)
     S←S∪{u}                              (add u to list of visited vertices)
     for all v ∈ neighbors[u]
          do  if   dist[v] > dist[u] + w(u, v)          (if new shortest path found)
                   then     d[v] ←d[u] + w(u, v)        (set new value of shortest path)
                                                        (if desired, add traceback code)

return dist
```



Q = {a,b,c,d}

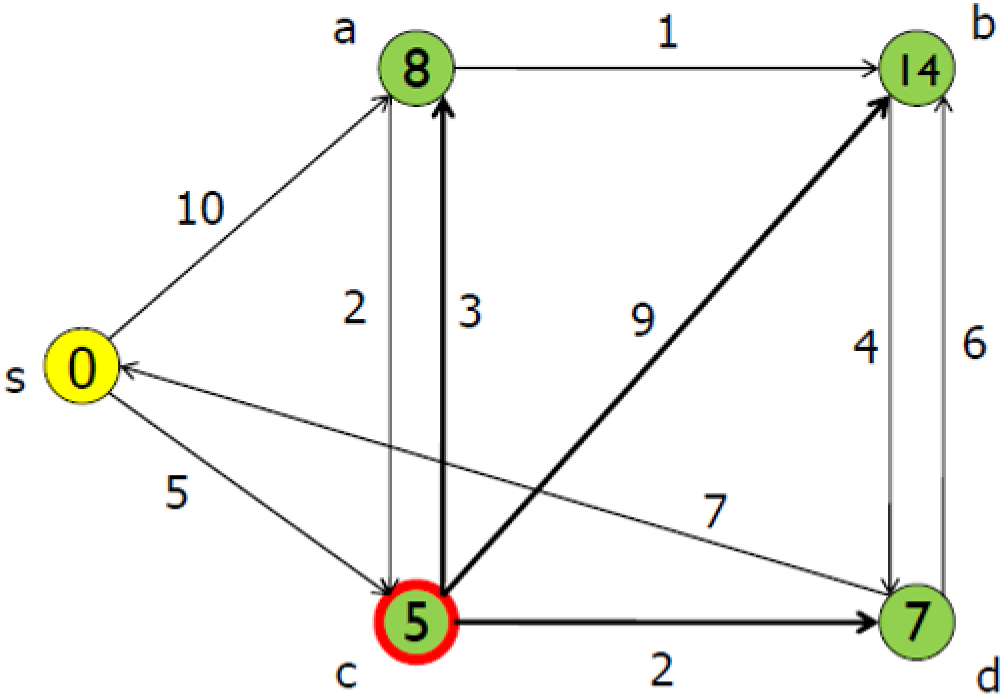
S = {s}

Dist_S_To = {(a,10), (b,∞), (c,5), (d,∞)}

Predecessor = {(a,s), (b,null), (c,s), (d,null)}

Dijkstra

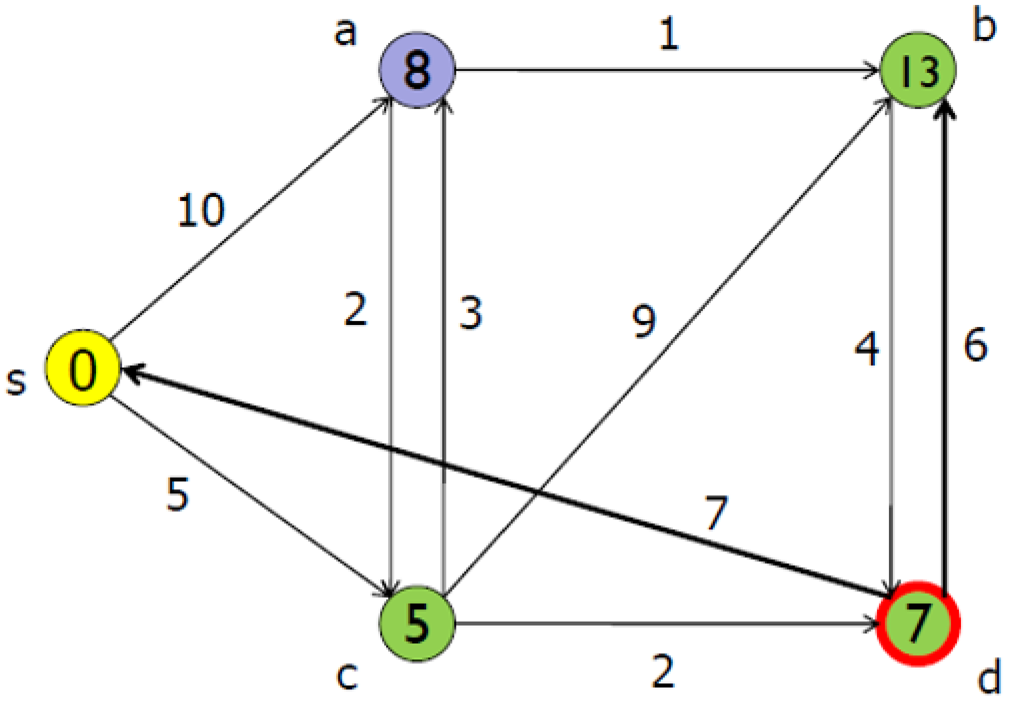
- Demo (3)



Q = {a,b,d}
S = {c,s}
Dist_S_To = {(a,8), (b,14), (c,5), (d,7)}
Predecessor = {(a,c), (b,c), (c,s), (d,c)}

Dijkstra

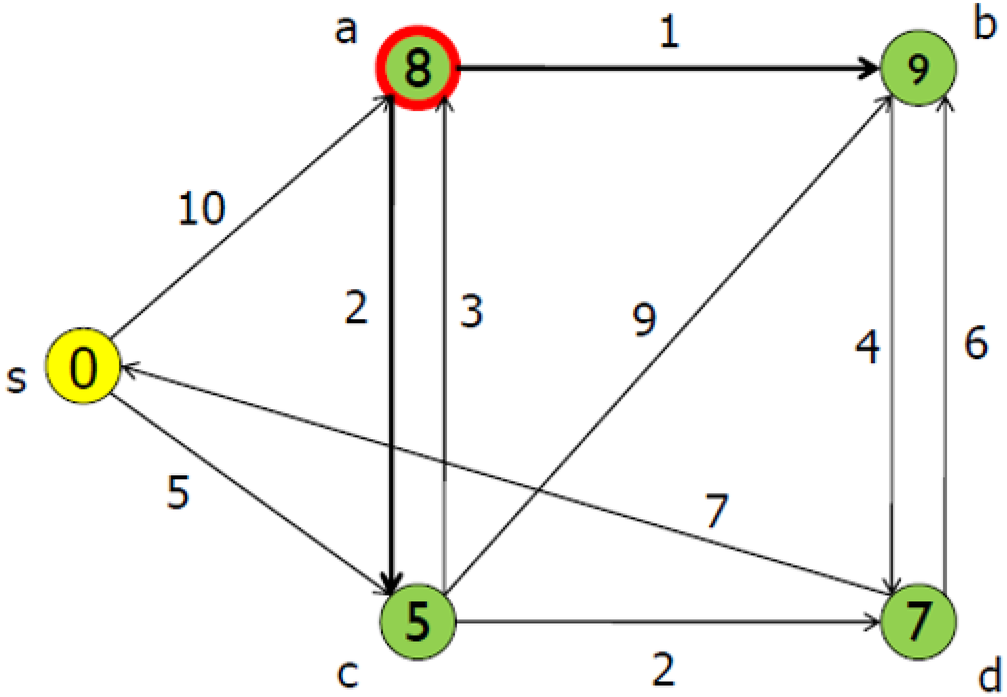
- Demo (4)



Q = {a,b}
S = {c,d,s}
Dist_S_To = {(s,0), (a,8), (b,13), (c,5), (d,7)}
Predecessor = {(a,c), (b,d), (c,s), (d,c)}

Dijkstra

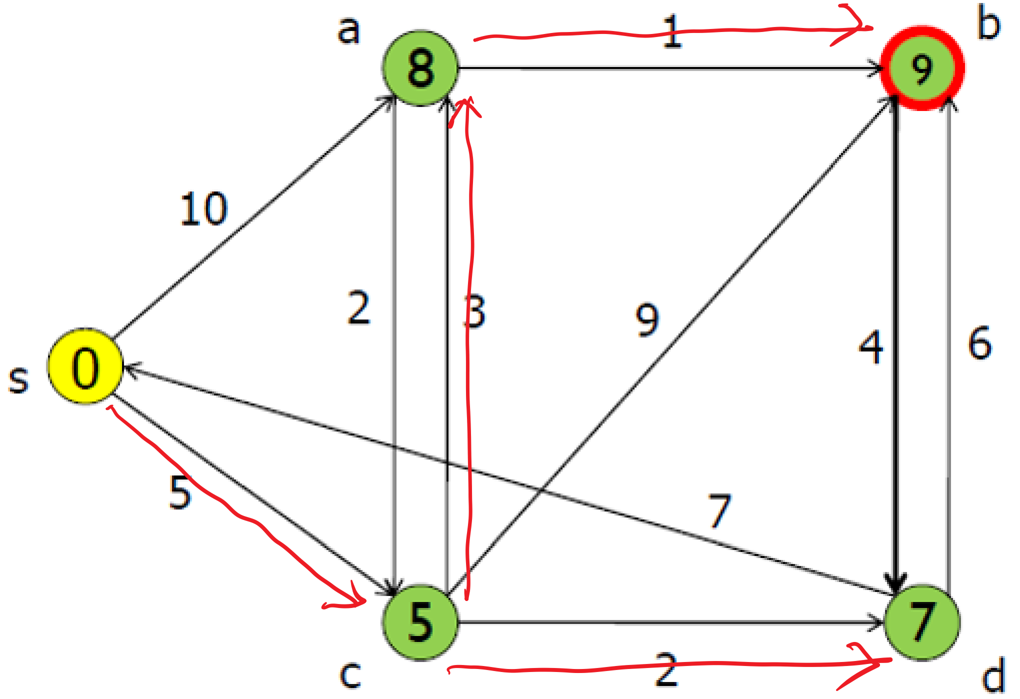
- Demo (5)



Q = {b}
S = {a,c,d,s}
Dist_S_To = {(s,0), (a,8), (b,9), (c,5), (d,7)}
Predecessor = {(a,c), (b,a), (c,s), (d,c)}

Dijkstra

- Demo (6)



Q = {}
S = {a,b,c,d,s}
Dist_S_To = {(s,0), (a,8), (b,9), (c,5), (d,7)}
Predecessor = {(a,c), (b,a), (c,s), (d,c)}

Dijkstra

- Demo (7)

Q = {}
S = {a,b,c,d,s}
Dist_S_To = {(a,8), (b,9), (c,5), (d,7)}
Predecessor = {(a,c), (b,a), (c,s), (d,c)}

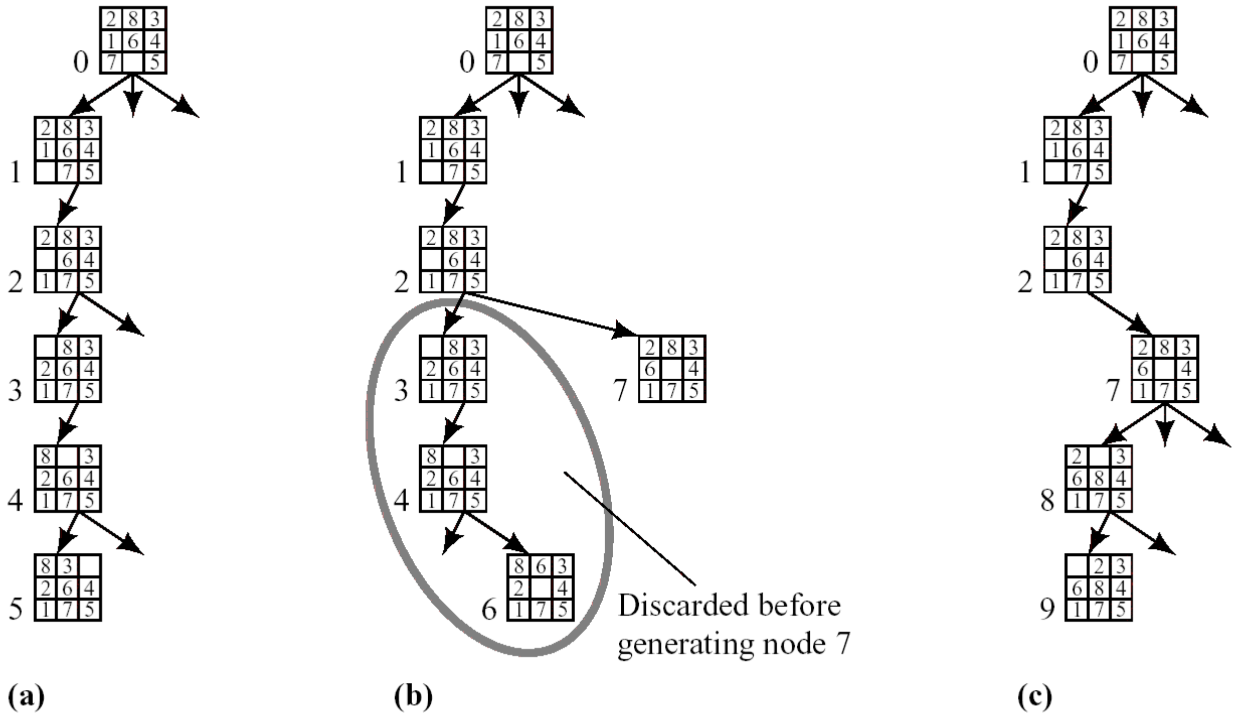
- After we get information above, we can determine any shortest path from the source node to any node in the graph by following predecessor set
- Running time complexity for Dijkstra algorithm is O(V^2) for matrix representation and O(E Log V) for adjacency list representation

Depth First or Backtracking Search

- DFS
  - Generates the successor of a node just one at a time.
  - A trace is left at each node
    - indicate that additional operators can be applied there if needed.
  - At each node a decision must be made
    - which operator to apply first, which next, and so on.
  - Repeats this process until the *depth bound.*
  - *chronological Backtrack* when search depth is depth bound.
- 8-puzzle example
  - Depth bound: 5
  - Operator order: left → up → right → down

```
PROCEDURE DepthFirstSearch(Tree tree, Node root, Goal goal, List alreadyVisited)

        Node current
        Stack toVisit

        add root to toVisit

        while toVisit is not empty
                current <- first node on toVisit
                remove first node from toVisit
                if current = goal
                        add current to alreadyVisited
                        return true
                end if
                for each child node C of current
                        add C to toVisit
                end for
                add current to alreadyVisited
        end while
```

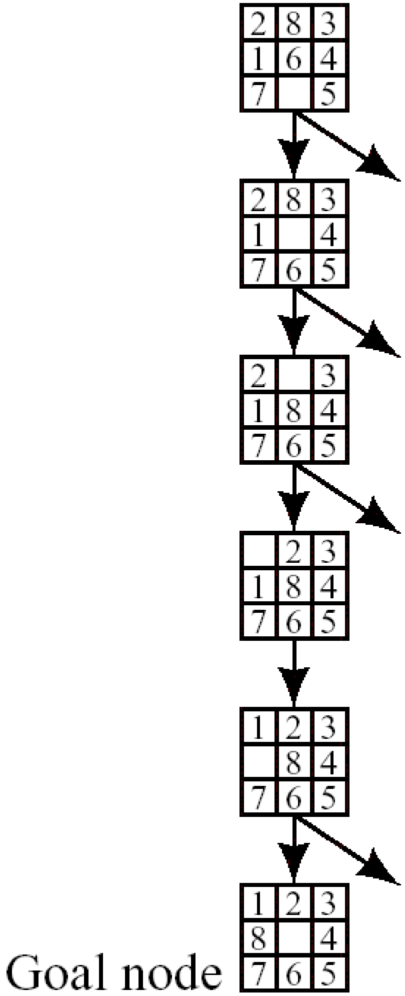
“if the depth is within the depth bound”


0
1
2
3
4
5
0
1
2
3
4
6
7
Discarded before generating node 7
0
1
2
7
8
9

(a)

(b)

(c)


Goal node

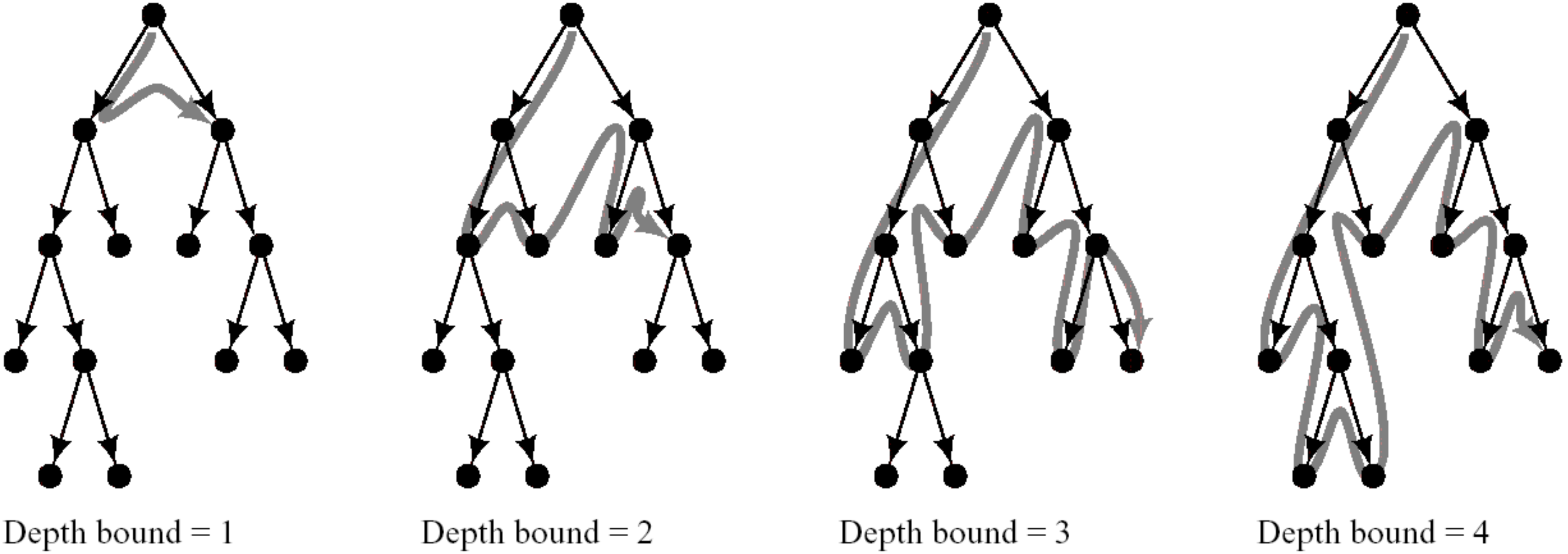
- Advantage
  - Low memory size: linear in the depth bound
    - saves only that part of the search tree consisting of the path currently being explored plus traces
- Disadvantage
  - No guarantee for the minimal state length to goal state
  - The possibility of having to explore a large part of the search space
- Complete??
  - No: fails in infinite-depth spaces, spaces with loops

- Time (Worst case): $=1 + b + b^2 + \text{.........} + b^d = O(b^{d+1})$

- Space: O(bm), i.e., linear space!

Iterative Deepening

- Advantage
  - Linear memory requirements of depth-first search
  - Guarantee for goal node of minimal depth
- Procedure
  - Successive depth-first searches are conducted – each with depth bounds increasing by 1



- The number of nodes
  - In case of breadth-first search

$$N_{\mathrm{bf}} = 1 + b + b^2 + \cdots + b^d = \frac{b^{d+1}-1}{b-1} \quad (b : \text{branching factor},\ d : \text{depth})$$

  - In case of iterative deepening search

$$N_{\mathrm{df}_j} = \frac{b^{j+1}-1}{b-1} \ :\ \text{number of nodes expanded down to level } j$$

$$N_{\mathrm{id}} = \sum_{j=0}^{d} \frac{b^{j+1}-1}{b-1}$$

$$= \frac{1}{b-1}\left[\left(b\left(\sum_{j=0}^{d} b^j\right) - \sum_{j=0}^{d} 1\right)\right] = \frac{1}{b-1}\left[b\left(\frac{b^{d+1}-1}{b-1}\right) - (d+1)\right]$$

$$= \frac{b^{d+2} - 2b - bd + d + 1}{(b-1)^2}$$

- For large *d* the ratio N_{id}/N_{df} is *b*/(*b-1*)
- For a branching factor of 10 and deep goals, 11% more nodes expansion in iterative-deepening search than breadth-first search
- Related technique *iterative broadening* is useful when there are many goal nodes