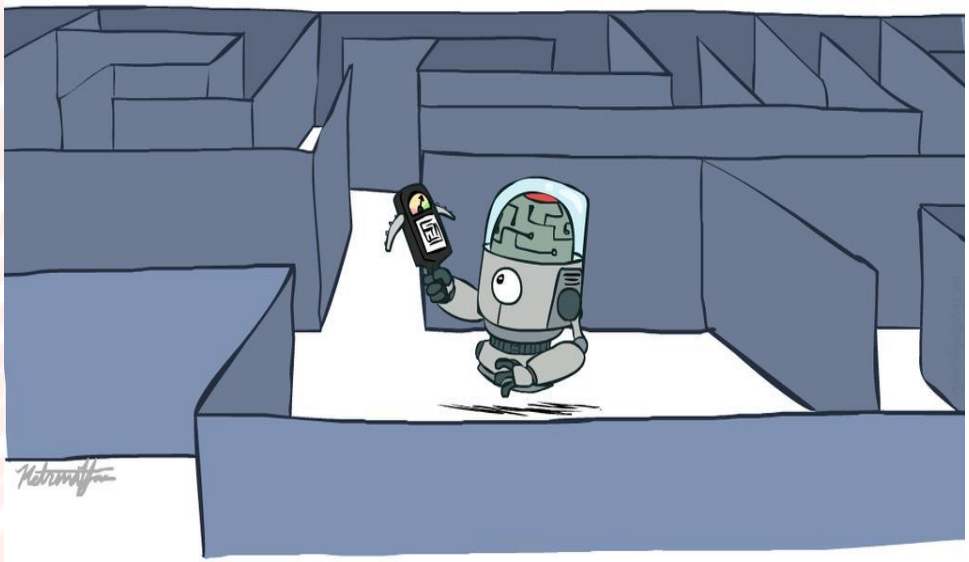
FACULTY OF INFORMATION TECHNOLOGY

Artificial Intelligence Fundamentals (NM TTNT)

Semester 1, 2023/2024

Chapter 4. Informed Search



Content

- Informed = use problem-specific knowledge
- Which search strategies?
  - Best-first search and its variants
- Heuristic functions?
  - How to invent them
- Local search and optimization
  - Hill climbing, local beam search, genetic algorithms,…
- …

Informed Search

- Informed searches *use domain knowledge to guide selection of the which path is best to continue searching.*

- Informed guesses, called *heuristics*, are used to guide the path selection.
  - Heuristic means "serving to aid discovery".

- All of the domain knowledge used to search is encoded in the *heuristic* function *h*.

Informed Search

- Define a heuristic function, h(n):
  - uses domain-specific information in some way
  - is computable from the current state description
  - it estimates:
    - the "goodness" of node n
    - how close node n is to a goal
    - the cost of minimal pathcost from node n to a goal state

h(n) >= 0	required for all nodes n
h(n) = 0	implies n is a goal node
h(n) = infinity	implies n is a dead end from which a goal cannot be reached

Previously: Tree search algorithm

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

- ***frontier*** contains generated nodes which are not yet expanded.

Previously: Graph search algorithm

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

- ***frontier*** contains generated nodes which are not yet expanded.
- ***explored*** list stores all expanded nodes

Best-first search

- A generic way of referring to the class of informed search methods.
- Idea: node is selected for expansion based on an evaluation function *f(n)*
  - estimate of "desirability"

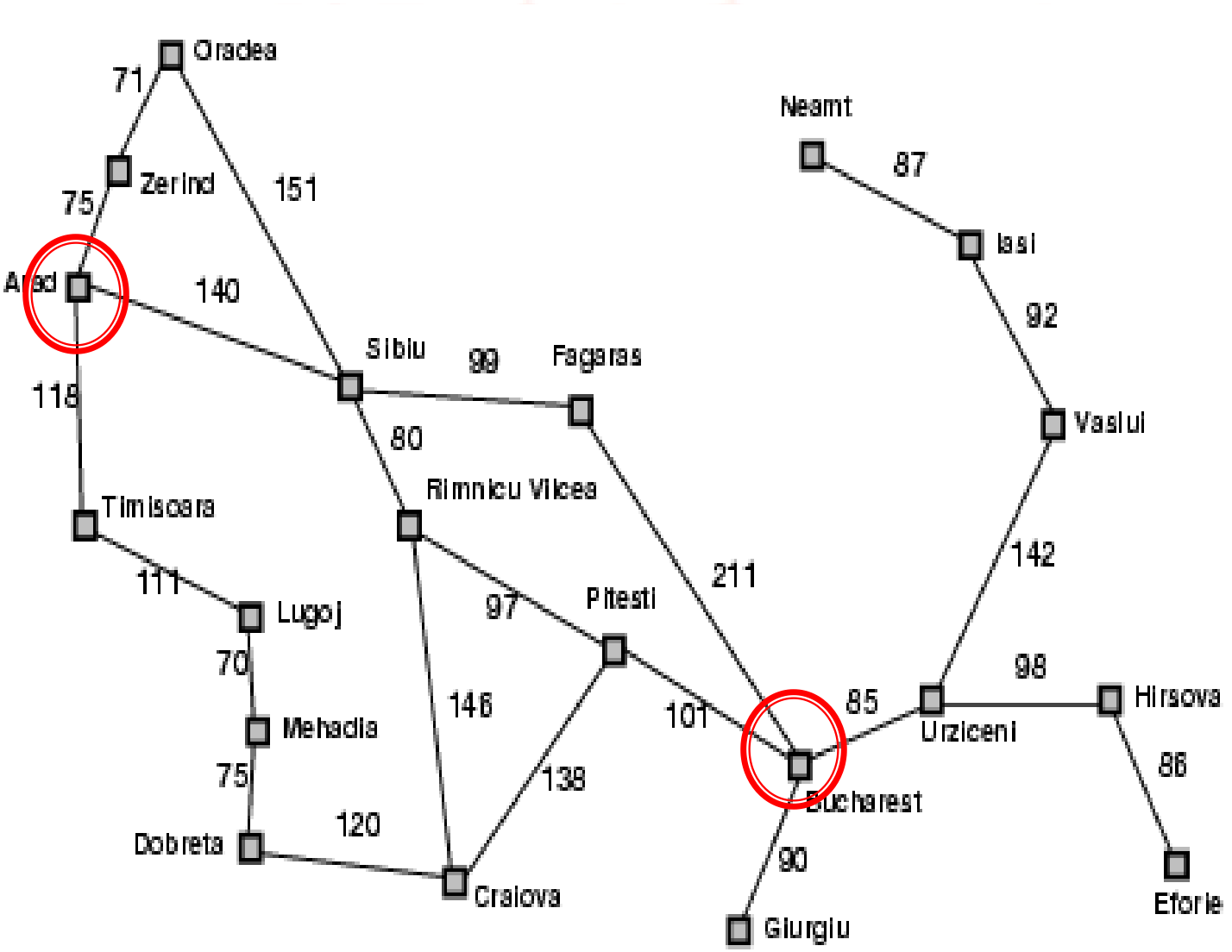
  →Expand most desirable unexpanded node

- Implementation: Order the nodes in frontier in decreasing order of desirability
- Special cases:
  - Greedy best-first search
  - A* search

Greedy best-first search

- One of the simplest best-first search strategies is to minimize the estimated cost to reach the goal
- Evaluation function $f(n) = h(n)$ (heuristic)
  - [dictionary] *"A rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood."*
  - **estimated cost of the cheapest path from n to goal**
- Greedy best-first search expands the node that appears to be closest to goal, e.g.
  - $h_{SLD}(n)$ = straight-line distance from n to Bucharest
  - If n is goal then $h(n)=0$
  - Expand node that is closest to goal: $h(n) \rightarrow \min$

Romania with step costs in km

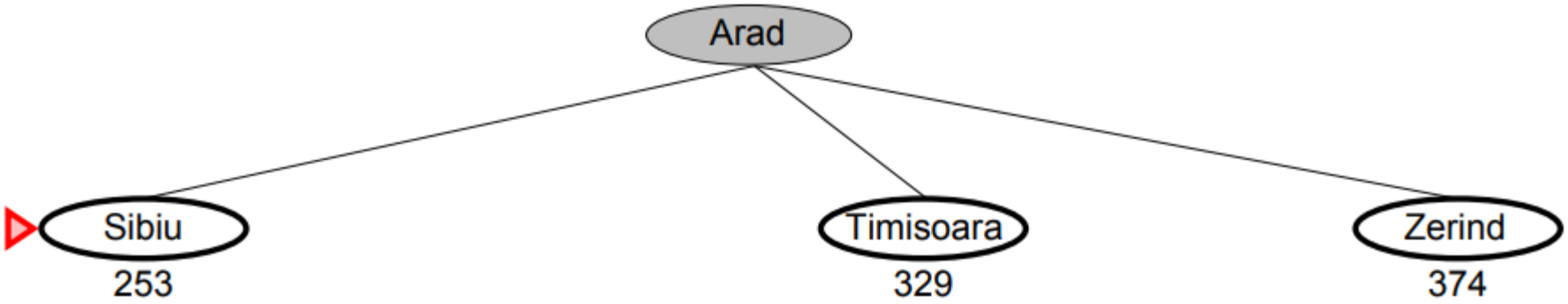


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example

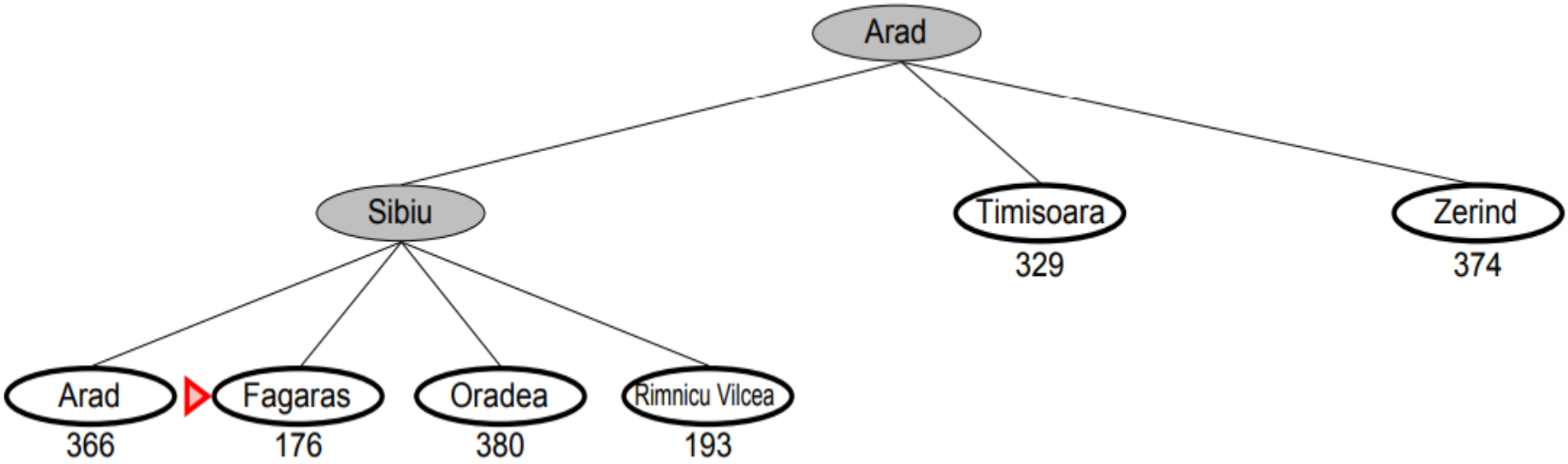
- Assume that we want to use greedy search to solve the problem of travelling from Arad to Bucharest.
- The initial state=Arad

Greedy best-first search example



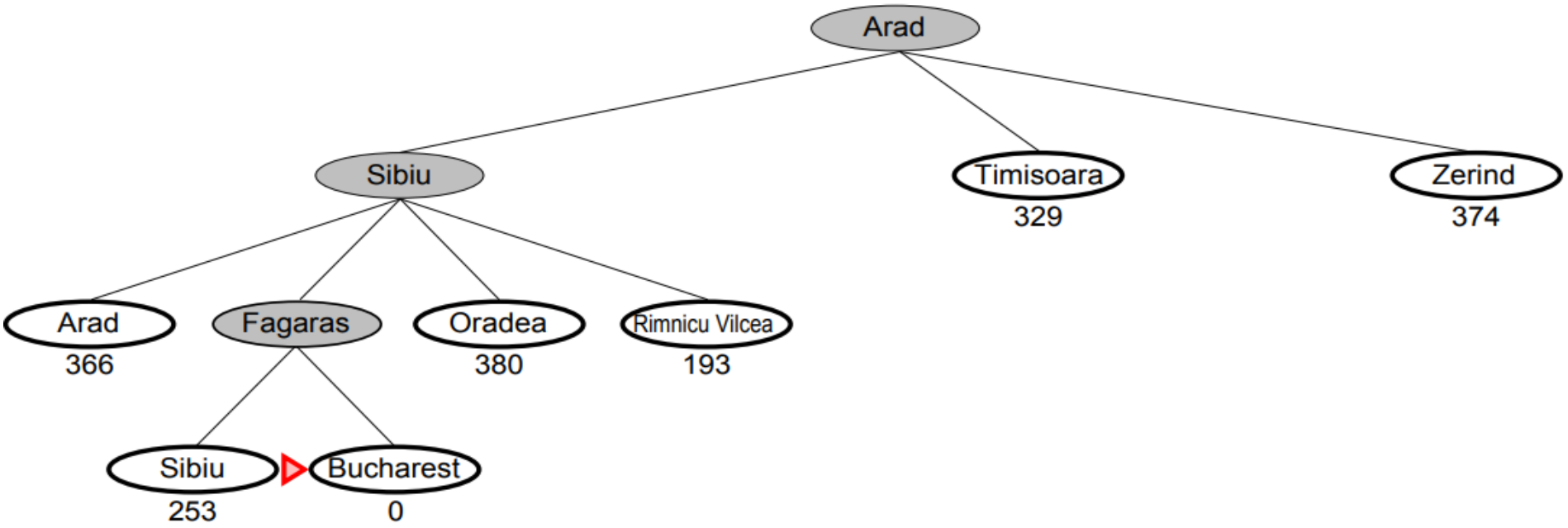
- The first expansion step produces:
  - Sibiu, Timisoara and Zerind
- Greedy best-first will select Sibiu.

Greedy best-first search example

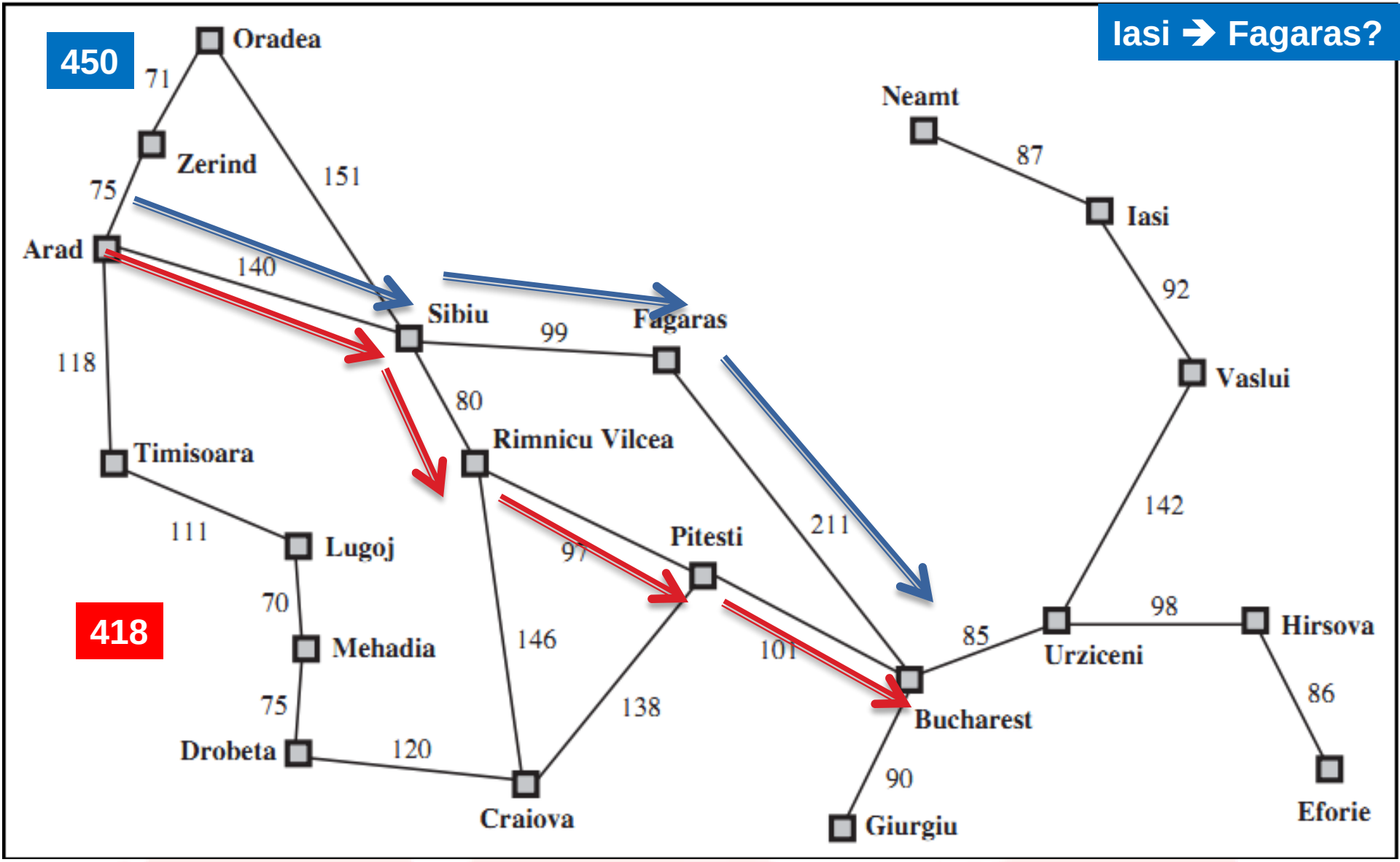


- If Sibiu is expanded we get:
  - Arad, Fagaras, Oradea and Rimnicu Vilcea
- Greedy best-first search will select: Fagaras

Greedy best-first search example



- If Fagaras is expanded we get:
  - Sibiu and Bucharest
- Goal reached !!
  - Yet not optimal (see Arad, Sibiu, Rimnicu Vilcea, Pitesti)


Iasi → Fagaras?
450
418
Oradea
71
Neamt
87
Zerind
151
Iasi
75
Arad
140
92
Sibiu
99
Fagaras
118
Vaslui
80
Rimnicu Vilcea
Timisoara
142
111
Lugoj
97
Pitesti
211
70
98
Hirsova
Mehadia
146
85
Urziceni
101
75
86
138
Bucharest
Drobeta
120
90
Craiova
Giurgiu
Eforie

Properties of greedy best-first search

- Complete: No – can get stuck in loops
  - e.g., Iasi → Neamt → Iasi → Neamt → …
  - Complete in finite space with repeated-state checking
- Time: $O(b^m)$, but a good heuristic can give dramatic improvement
- Space: $O(b^m)$ -- keeps all nodes in memory
- Optimal: No

A* algorithm

A* algorithm

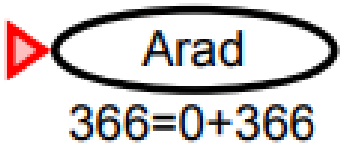
- Best-known form of best-first search
- **Idea**: avoid expanding paths that are already expensive.
- Evaluation function $f(n) = g(n) + h(n)$
  - $g(n)$ the path cost from start node (so far) to node n.
  - $h(n)$ estimated cost to get from node n to the goal.
  - $f(n)$ estimated total cost of path through n to goal.

A* algorithm

- A* search uses an **admissible heuristic** h(n)
  - A heuristic is admissible if it ***never overestimates*** the cost to reach the goal
    1. **h(n) <= h*(n)** where h*(n) is the true cost from n to goal
    2. **h(n) >= 0** so h(G)=0 for any goal G.

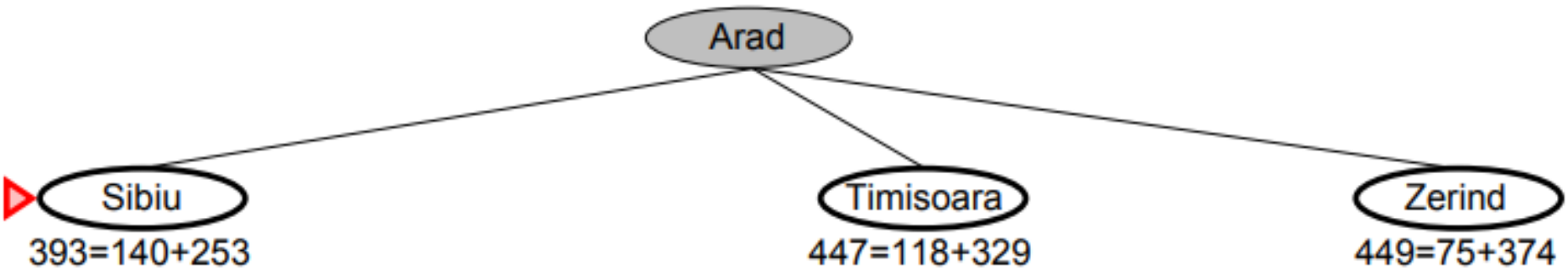
    e.g. $h_{SLD}(n)$ never overestimates the actual road distance
  - If h is admissible, f(n) never overestimates the actual cost of the best solution through n

A* algorithm example



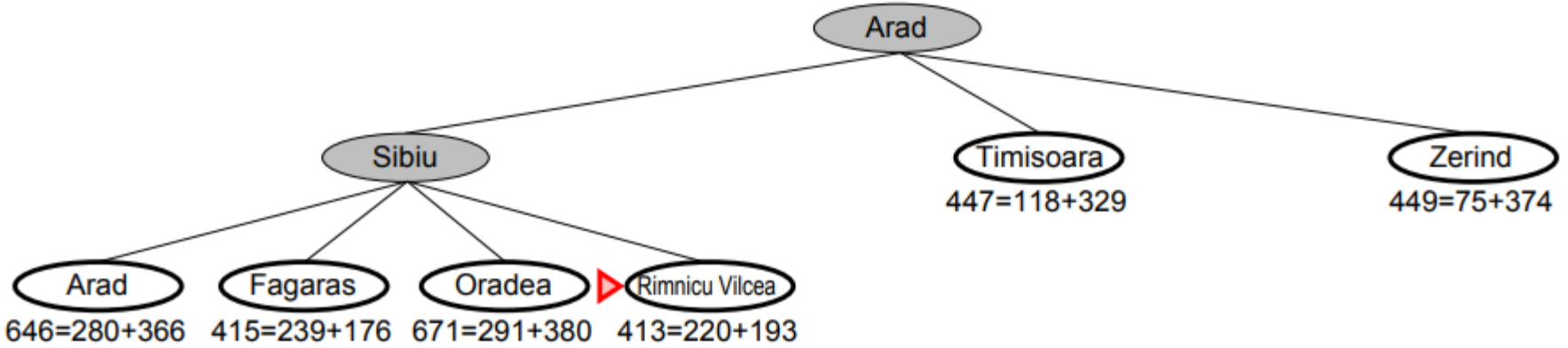
- Find Bucharest starting at Arad
  - *f(Arad) = c(??, Arad) + h(Arad) = 0 + 366 = 366*

A* algorithm example



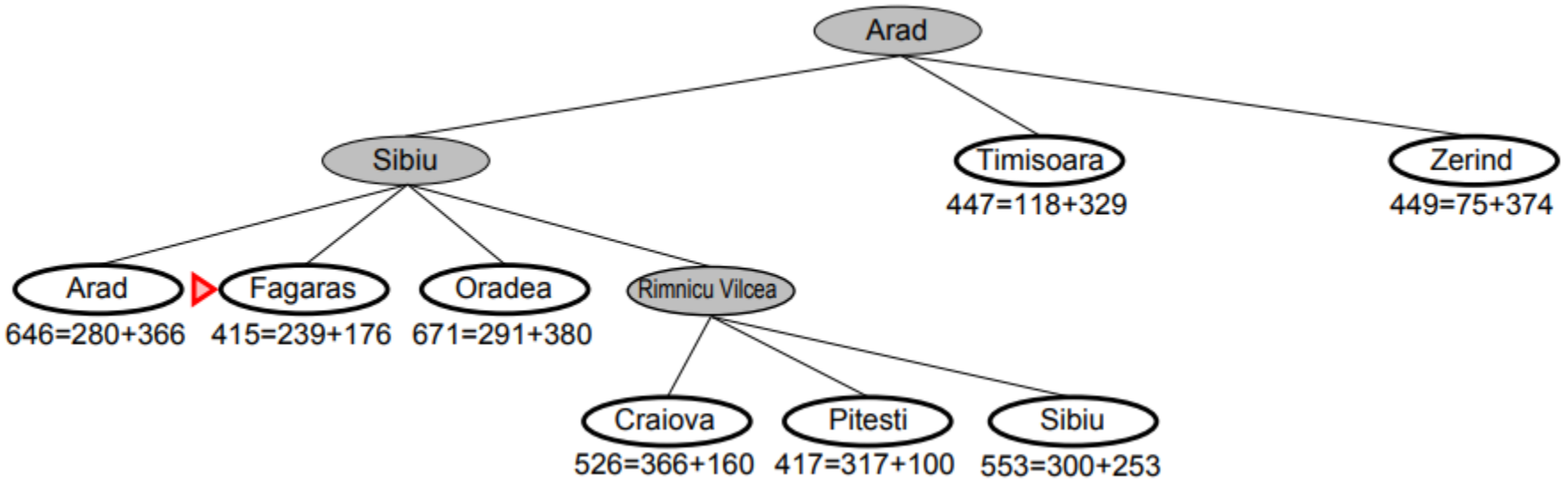
- Expand Arad and determine *f(n)* for each node
  - *f(Sibiu) = c(Arad, Sibiu) + h(Sibiu) = 140 + 253 = 393*
  - *f(Timisoara) = c(Arad, Timisoara) + h(Timisoara) = 118 + 329 = 447*
  - *f(Zerind) = c(Arad, Zerind) + h(Zerind) = 75 + 374 = 449*
- Best choice is Sibiu

A* algorithm example



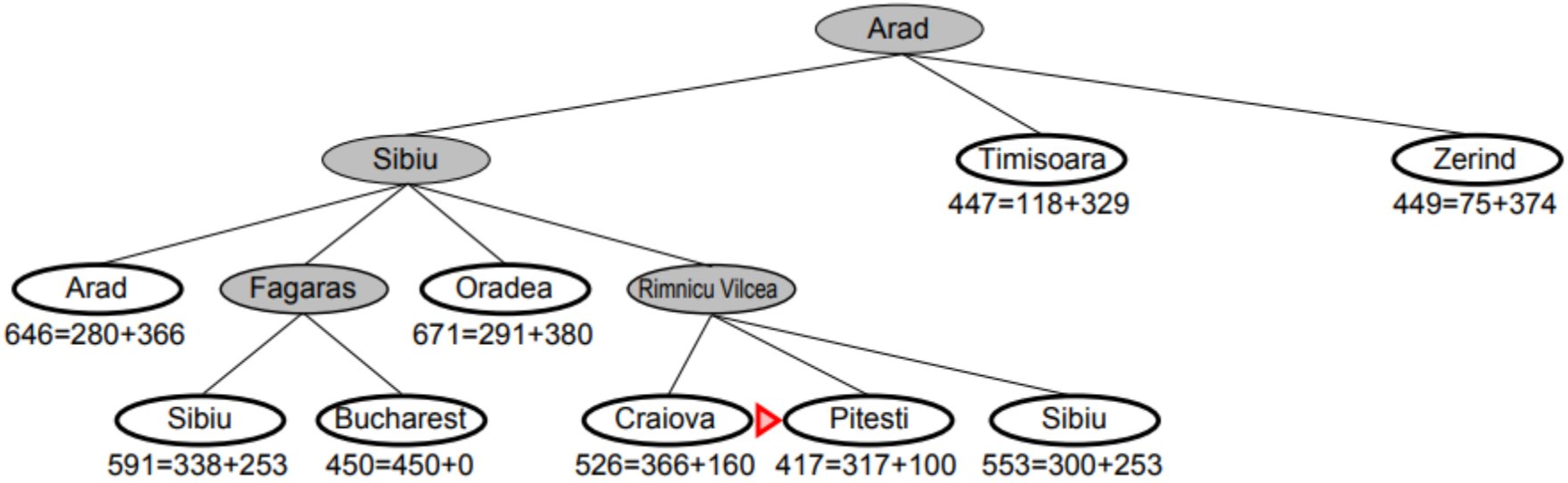
- Expand Sibiu and determine f(n) for each node
  - *f(Arad) = g(Sibiu) + c(Sibiu, Arad) + h(Arad) = 646*
  - *f(Fagaras) = g(Sibiu) + c(Sibiu, Fagaras) + h(Fagaras) = 415*
  - *f(Oradea) = g(Sibiu) + c(Sibiu, Oradea) + h(Oradea) = 671*
  - *f(Rimnicu Vilcea)= g(Sibiu) + c(Sibiu, Rimnicu Vilcea) + h(Rimnicu Vilcea) = 413*
- Best choice is Rimnicu Vilcea

A* algorithm example



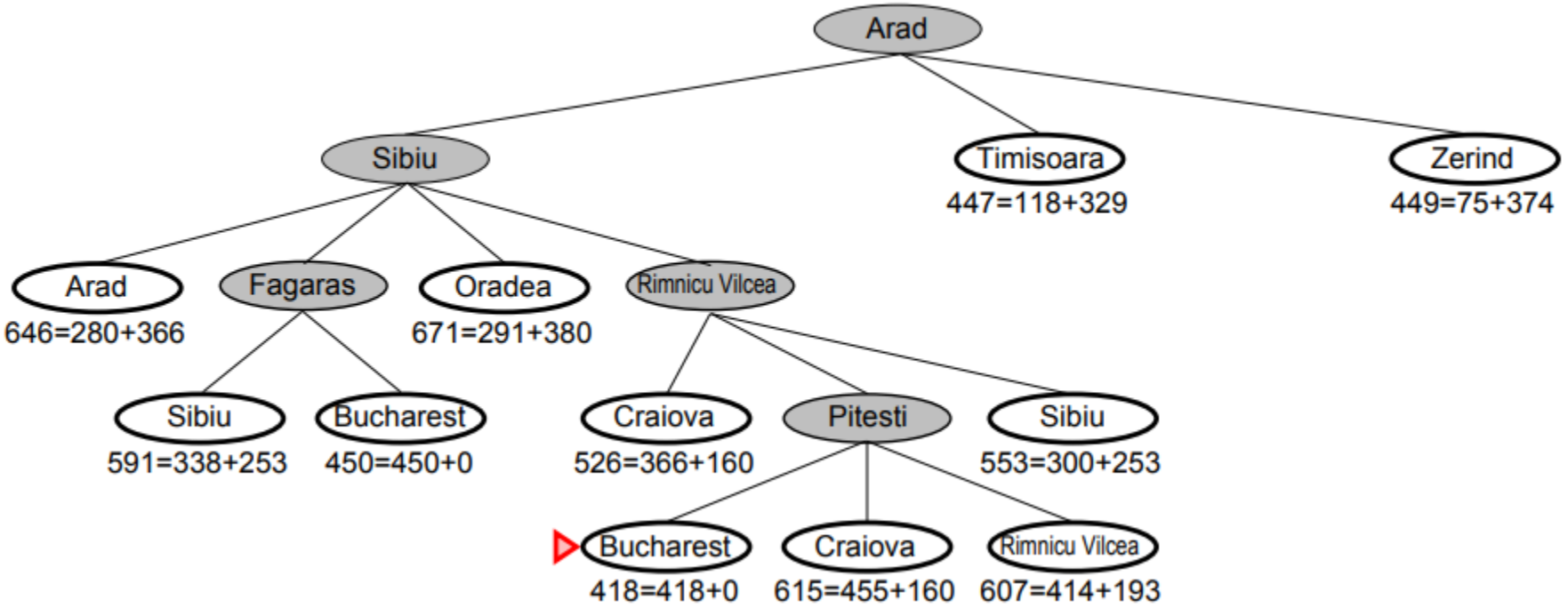
- Expand Rimnicu Vilcea and determine f(n) for each node
  - *f(Craiova) = g(Rimnicu Vilcea) + c(Rimnicu Vilcea,Craiova) + h(Craiova) = 526*
  - *f(Pitesti) = g(Rimnicu Vilcea) + c(Rimnicu Vilcea, Pitesti)+ h(Pitesti) = 417*
  - *f(Sibiu) = g(Rimnicu Vilcea) + c(Rimnicu Vilcea, Sibiu) + h(Sibiu) = 553*
- Best choice is Fagaras

A* algorithm example



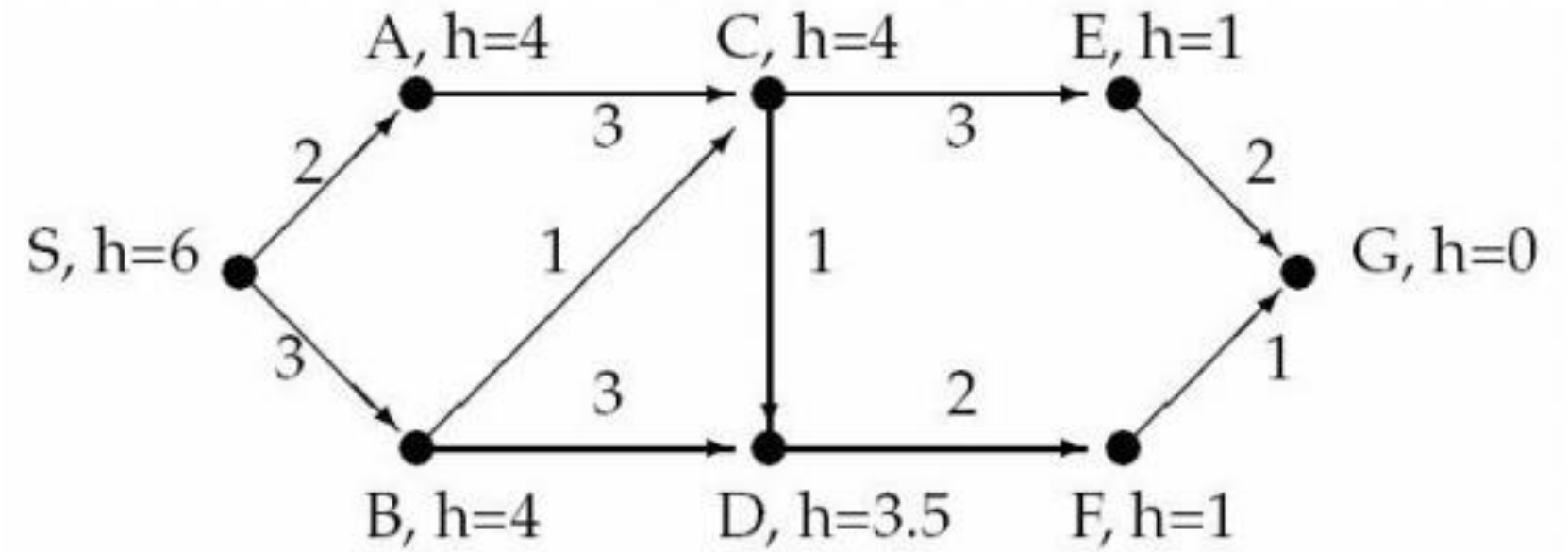
- Expand Fagaras and determine *f(n)* for each node
  - *f(Sibiu) = g(Fagaras) + c(Fagaras, Sibiu) + h(Sibiu) = 591*
  - *f(Bucharest) =* g(*Fagaras*) *+ c(Fagaras, Bucharest) + h(Bucharest) = 450*
- Best choice is Pitesti !!!

A* algorithm example



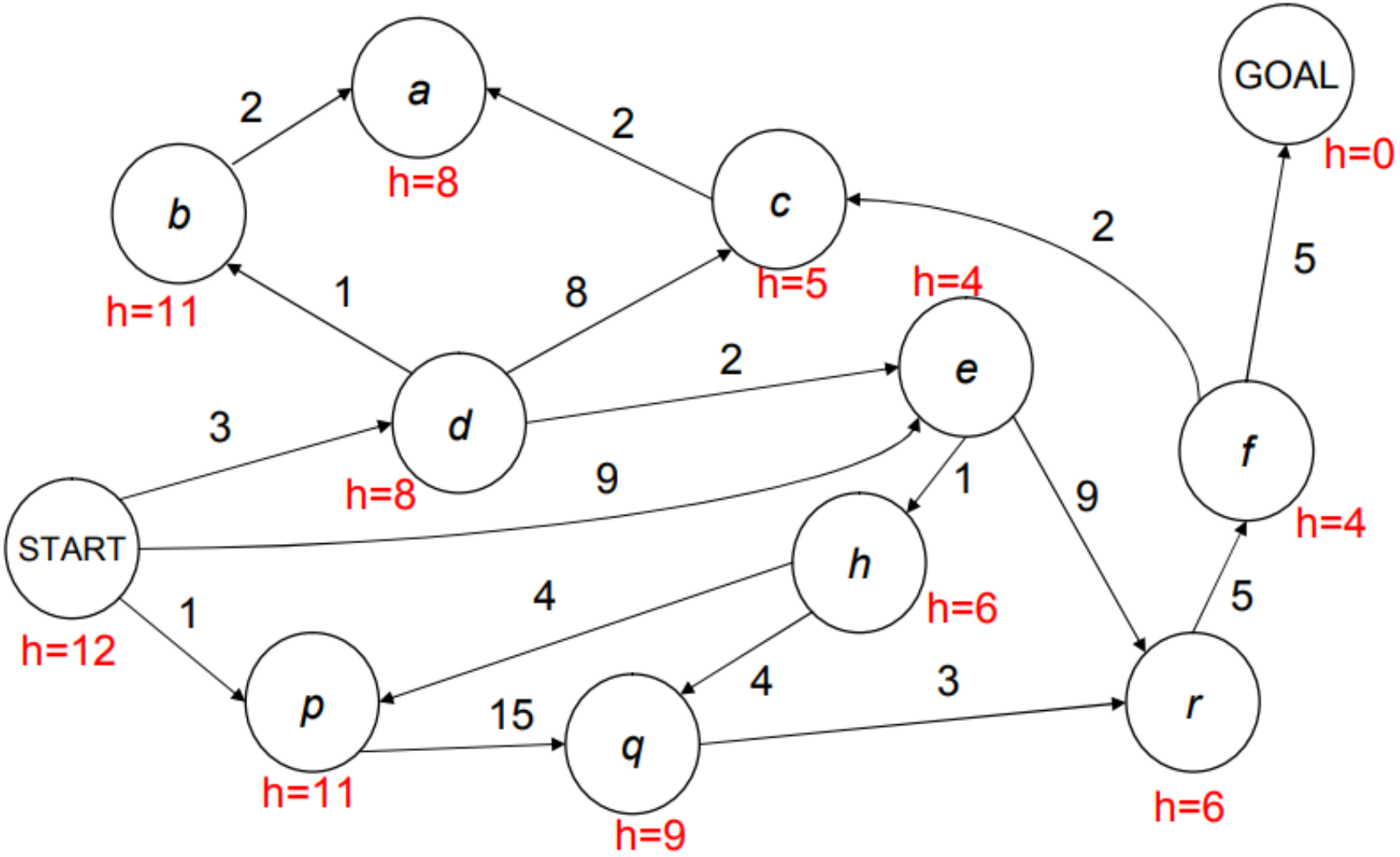
- Expand Pitesti and determine *f(n)* for each node
  - *f(Bucharest) = g(Pitesti) + c(Pitesti, Bucharest) + h(Bucharest) = 418*
- Best choice is Bucharest !!!
  - Optimal solution (only if *h(n)* is admissable)
- Note values along optimal path !!

A* example 1

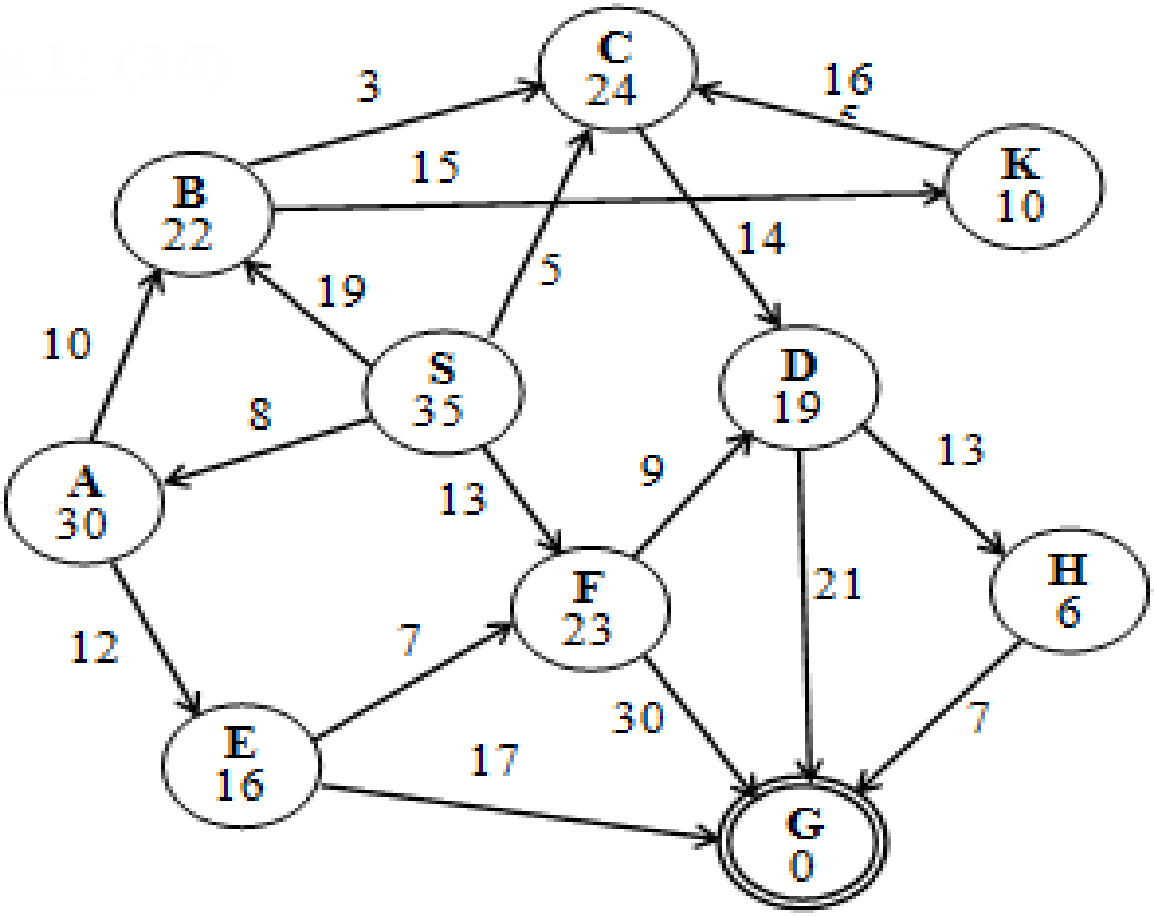


- Using the A* algorithm to find the shortest path from node S to node G. Each node is labeled by a capital letter and the value of a heuristic function. Each edge is labeled by the cost to traverse that edge.
- Assume that if you find a path to a node already on the queue that you update its cost (using the lower f value) instead of adding another copy of that node to the queue.

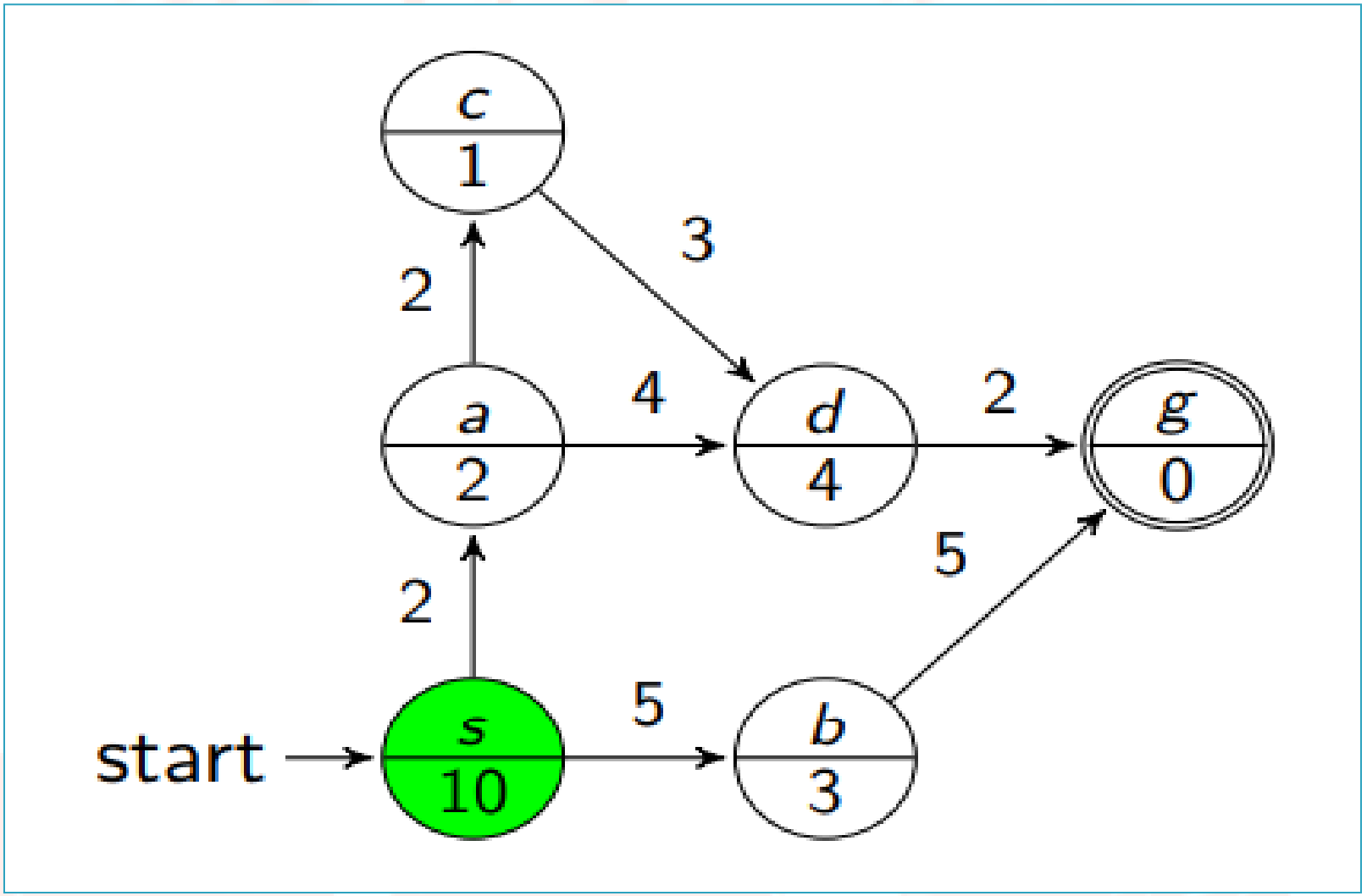
A* example 2


a
h=8
b
h=11
c
h=5
GOAL
h=0
d
h=8
e
h=4
f
h=4
START
h=12
h
h=6
p
h=11
q
h=9
r
h=6
2
2
2
5
1
8
2
3
9
1
9
1
4
5
4
3
15

Exercise 1

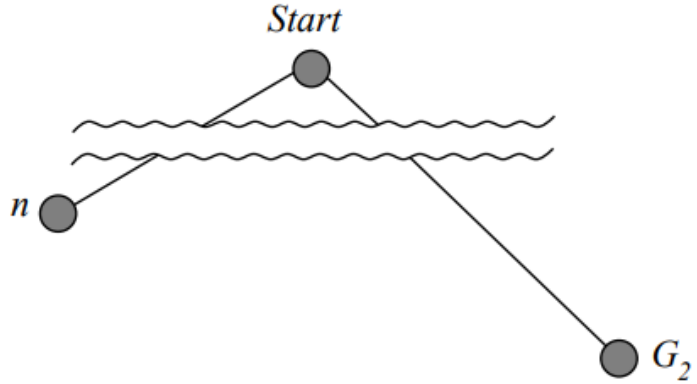

C
24
3
16
K
10
B
22
15
14
19
5
10
S
35
D
19
8
A
30
13
9
13
F
23
21
H
6
12
7
30
7
E
16
17
G
0

Exercise 2



Optimality of A* (standard proof)

- Suppose some suboptimal goal G_2 has been generated and is in the queue.
- Let n be an unexpanded node on a shortest path to an optimal goal G.



$f(G_2) = g(G_2)$ since $h(G_2) = 0$

$> g(G)$ since G_2 is suboptimal

$\geq f(n)$ since h is admissible

Since f(G2) > f(n), A* will never select G_2 for expansion

Consistent (monotonicity) heuristics

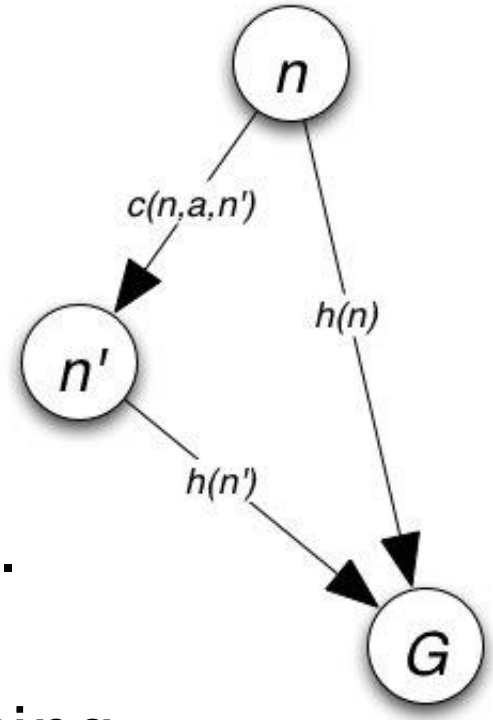
- A heuristic is consistent if for every node n, every successor n' of n generated by any action a,
  - $h(n) \leq c(n,a,n') + h(n')$
- If h is consistent, we have

  $f(n') = g(n') + h(n')$

  $= g(n) + c(n,a,n') + h(n')$

  $\geq g(n) + h(n) = f(n)$

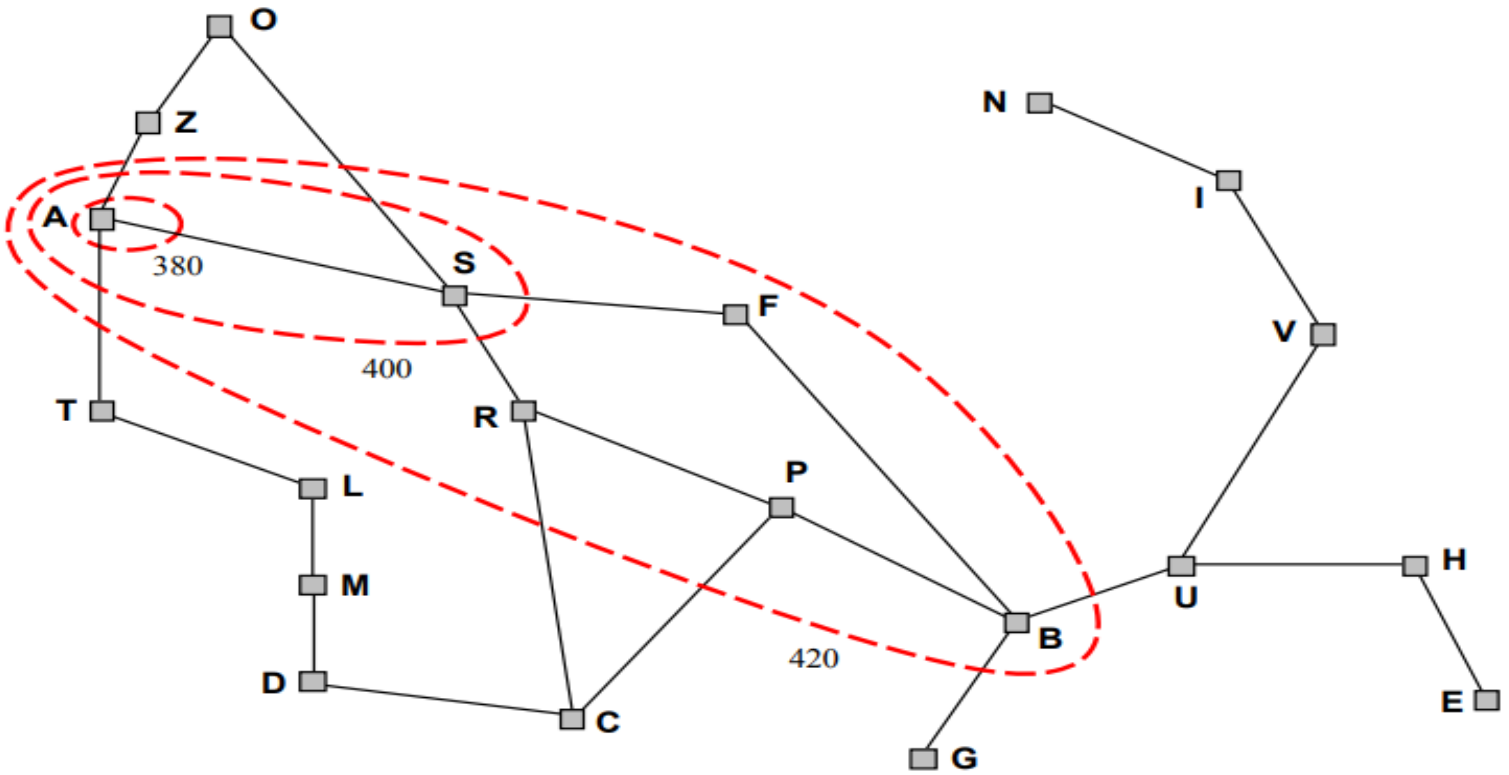
  i.e., f(n) is non-decreasing along any path.



- **Theorem:** If h(n) is consistent, A* using GRAPH-SEARCH is optimal

Optimality of A* (more useful)

- **Lemma:** A * expands nodes in order of increasing f value*
- Gradually adds "f-contours" of nodes (cf. breadth-first adds layers) Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$



Properties of A*

- Completeness: YES
  - Since bands of increasing f are added
  - Unless there are infinitly many nodes with $f<f(G)$
- Time complexity: $O(b^d)$
  - Number of nodes expanded is still exponential in the length of the solution.
- Space complexity: $O(b^d)$
  - It keeps all generated nodes in memory
  - Hence space is the major problem not time
- Optimality: YES

  A* expands all nodes with $f(n)< C^*$

  A* expands some nodes with $f(n)=C^*$

  A* expands no nodes with $f(n)>C^*$

UCS vs Greedy BFS vs A*

- **Uniform-cost search**: expand lowest path cost

$$f(n) = g(n)$$

- **Greedy best first search**: expand the node that is closest to the goal

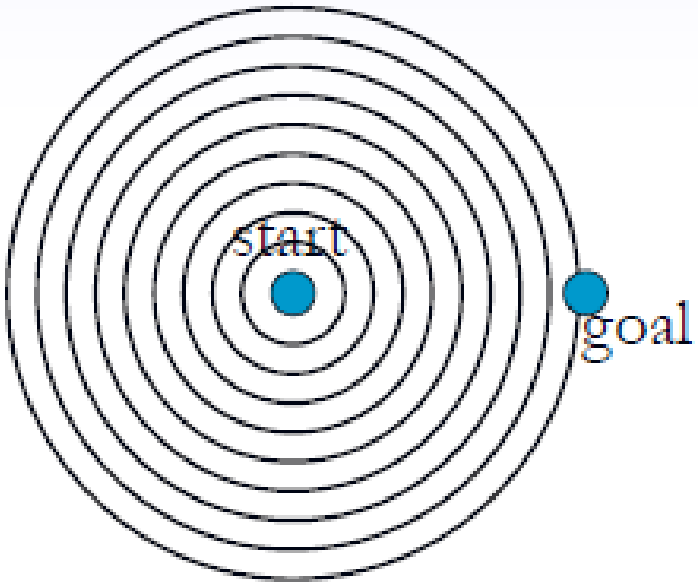
$$f(n) = h(n)$$

- **A* search**: combine UCS and Greedy (minimizing the total estimated solution cost)

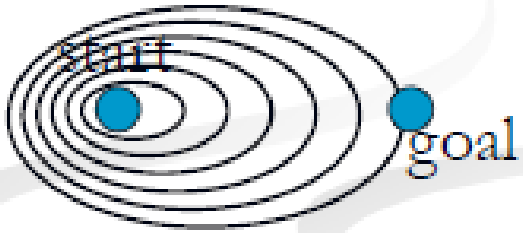
$$f(n) = g(n) + h(n)$$

Uninformed vs Informed search

Uninformed search



Informed search



Memory-bounded heuristic search

Some solutions to A* space problems (*maintain completeness and optimality*)

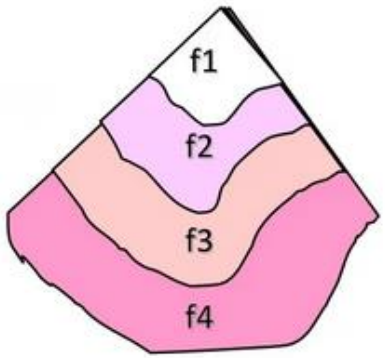
- Iterative-deepening A* (IDA*)
  - Here cutoff information is the **f-cost (g+h)** instead of depth
- Recursive best-first search(RBFS)
  - Recursive algorithm that attempts to mimic standard best-first search with linear space (using **f_limit** variable to keep track of the f-value of the best alternative path available from any ancestor of the current node).
- (Simplified) Memory-bounded A* ((S)MA*)
  - Drop the worst-leaf node when memory is full

IDA* Algorithm

f-bound ⟵ f(S)

Algorithm:

- **WHILE** (goal is not reached) **DO**
  - ***f-bound ⟵ f-limitted_search(f-bound)***
    - Perform **f-limited search** with ***f-bound***



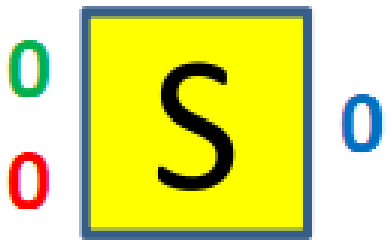
IDA* Algorithm (cont.)

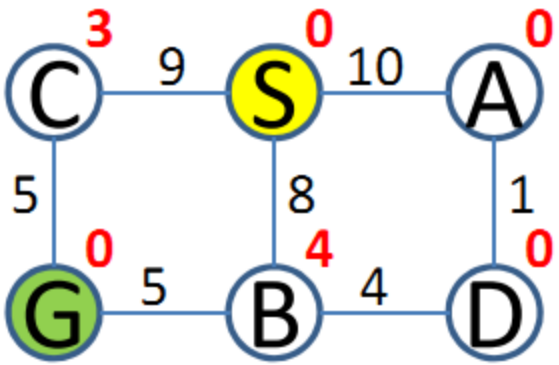
f-limitted Search Algorithm

- ***Input:***
  - QUEUE ⟵ Path only containing root
  - f-bound ⟵ Natural number
  - f-new ⟵ ∞
- ***Algorithm:***
  - **WHILE** (QUEUE not empty && goal not reached) **DO**
    - Remove **first path** from QUEUE
    - Create paths to children
    - Reject paths with loops
    - Add paths with **f(path) ≤ f-bound** to **front** of QUEUE *(depth-first)*
    - f-new ⟵ minimum({f-new} ∪ {f(P) | P is rejected path})
  - **IF** goal reached **THEN** success **ELSE** report f-new

IDA* Example – 1st iteration

- Find a path from S to G



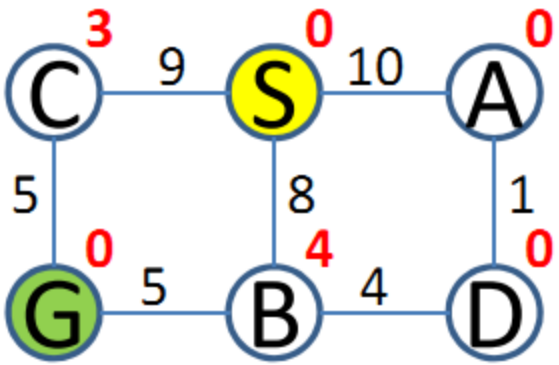


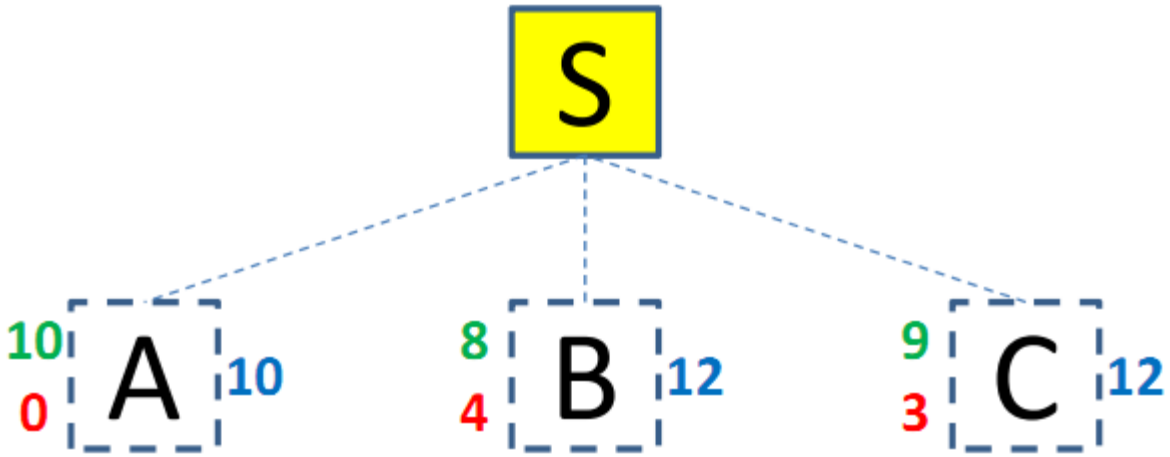
f_bound = f(S)=0
f_new = ∞

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example (cont.)

- Find a path from S to G



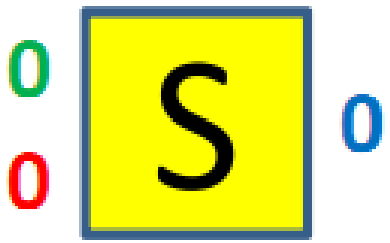


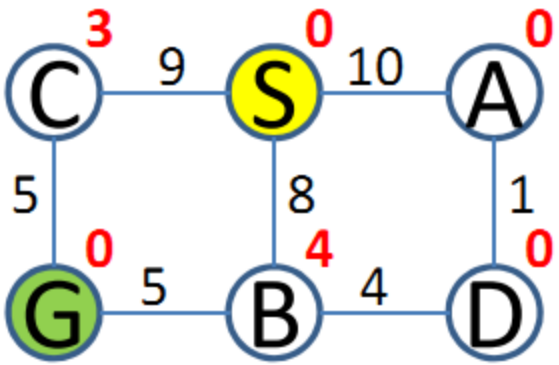
f_bound = 0
f_new = 10

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example – 2nd iteration

- Find a path from S to G



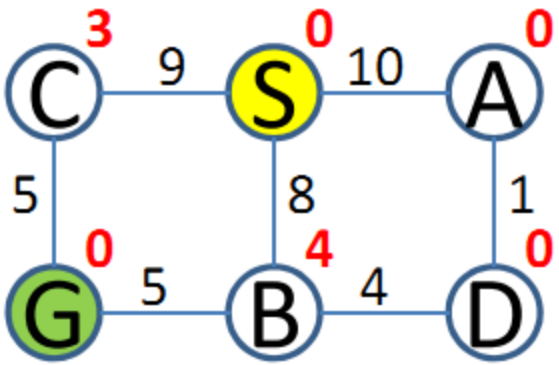


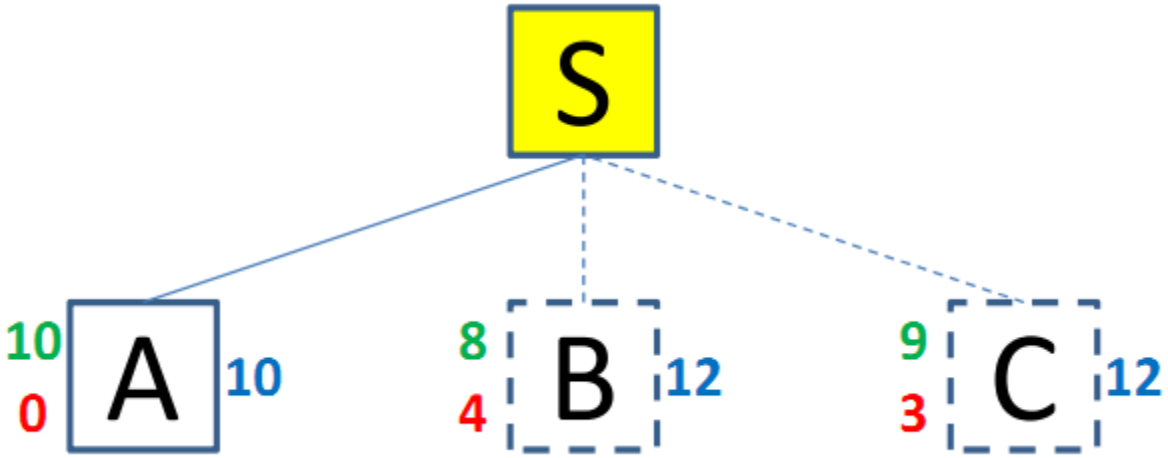
f_bound = 10
f_new = ∞

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example (cont.)

- Find a path from S to G



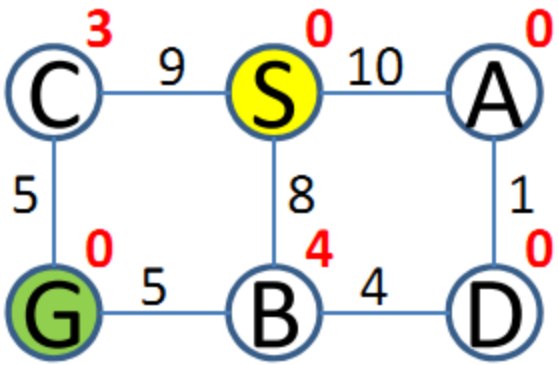


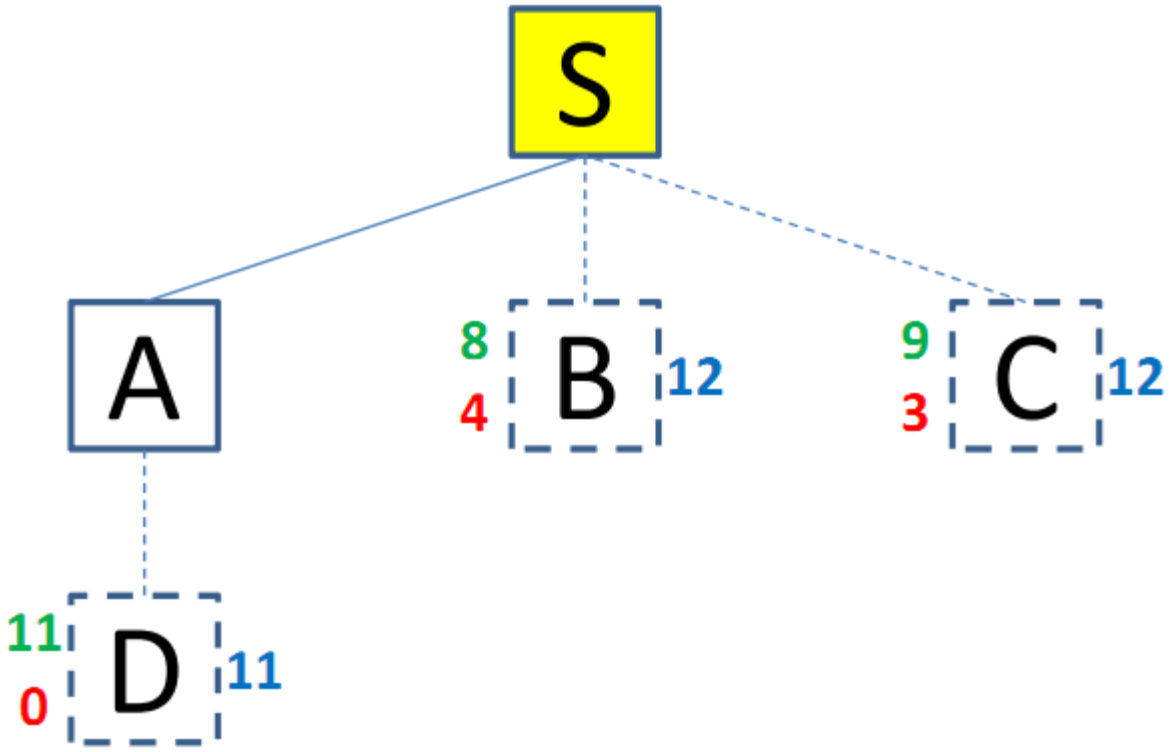
f_bound = 10
f_new = 12

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example (cont.)

- Find a path from S to G



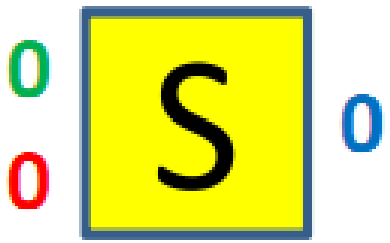


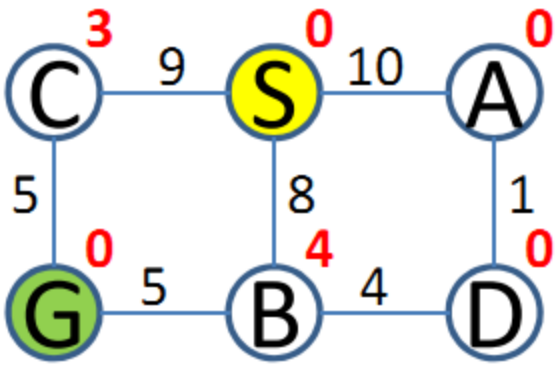
f_bound = 10
f_new = 11

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example – 3rd iteration

- Find a path from S to G



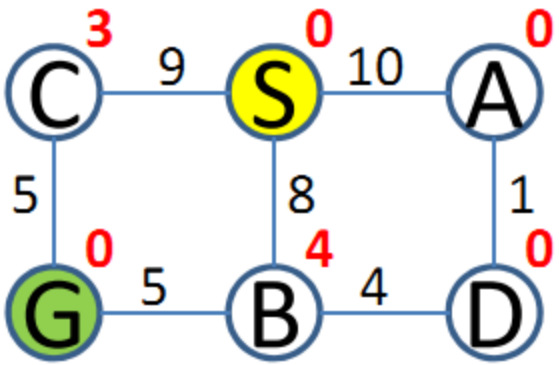


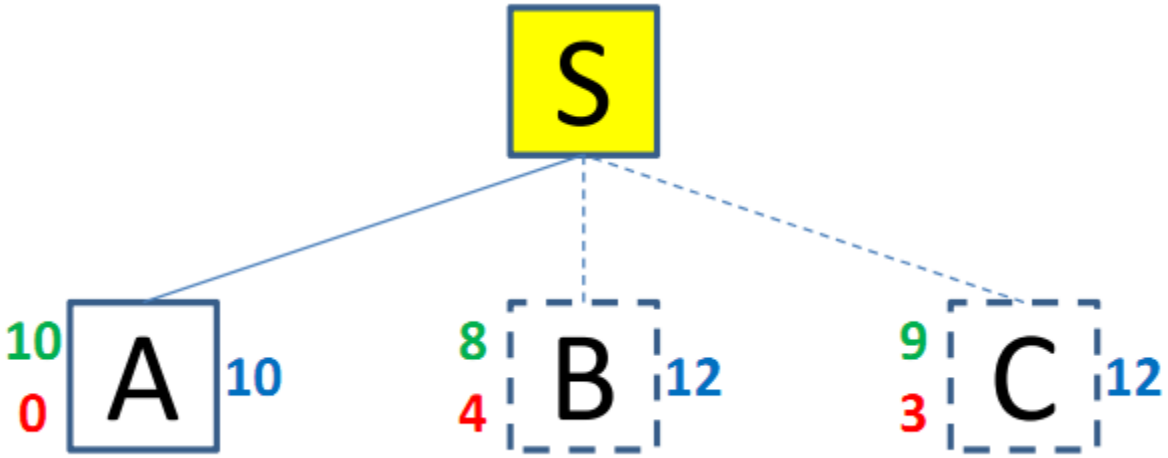
f_bound = 11
f_new = ∞

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example (cont.)

- Find a path from S to G



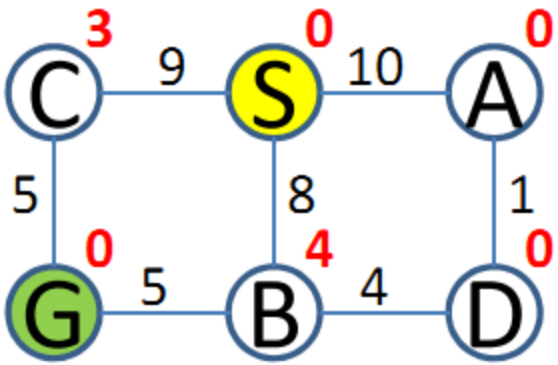


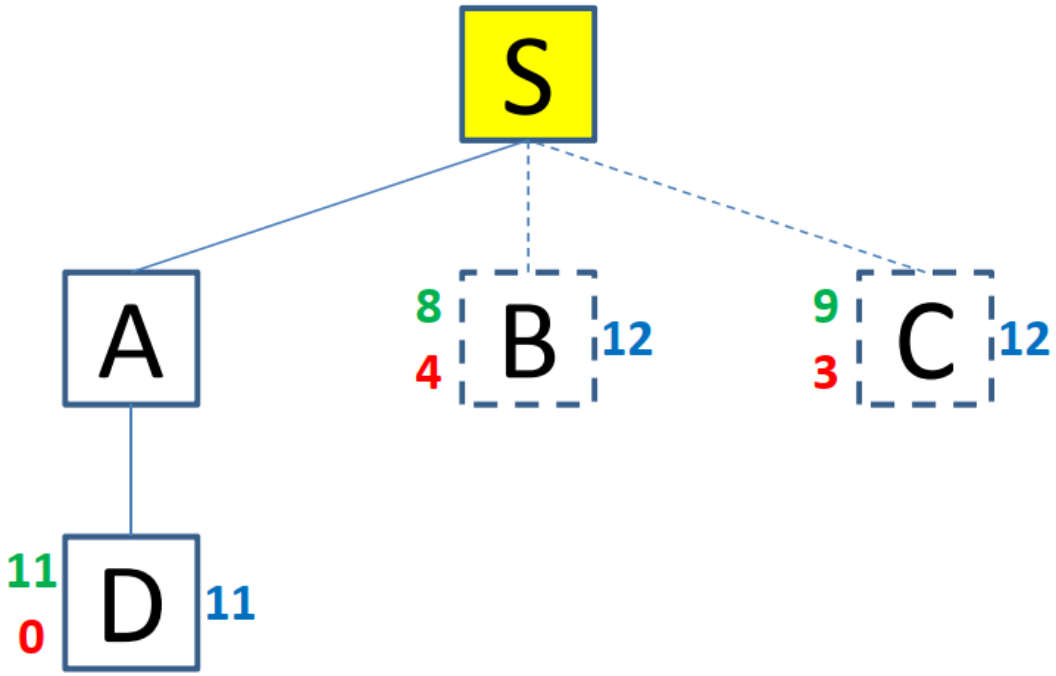
f_bound = 11
f_new = 12

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example (cont.)

- Find a path from S to G



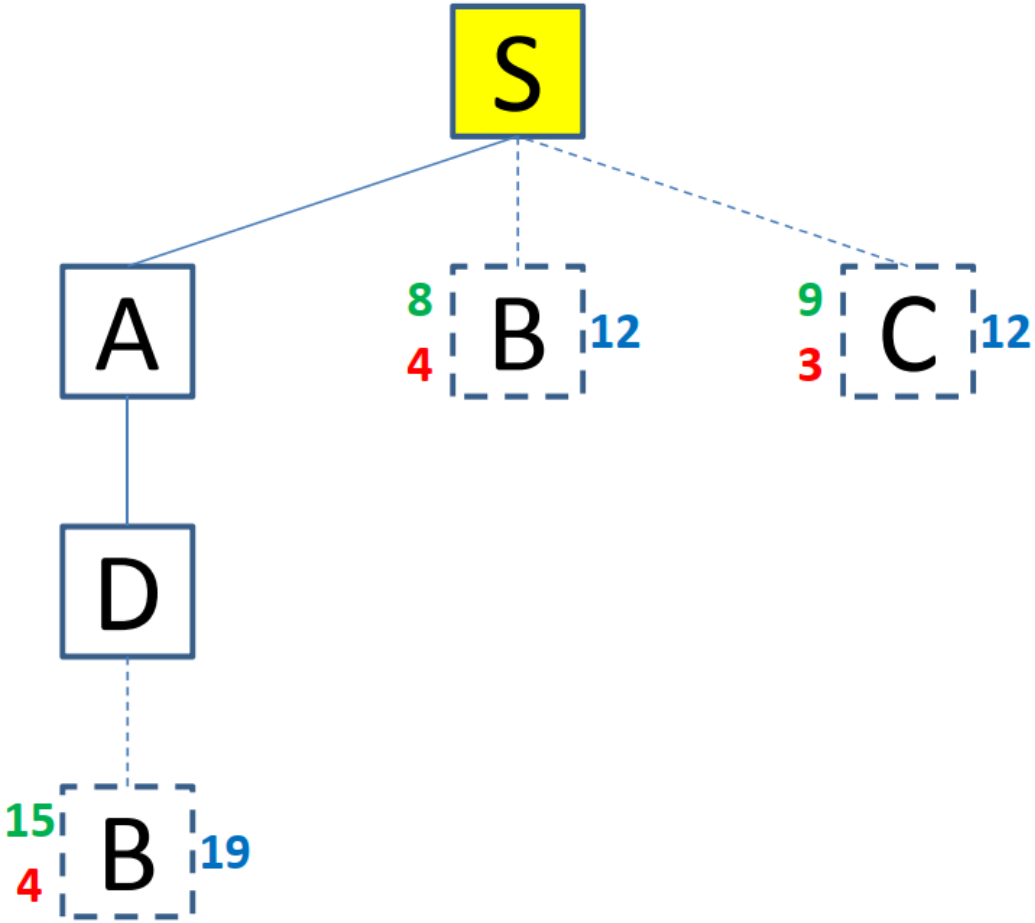


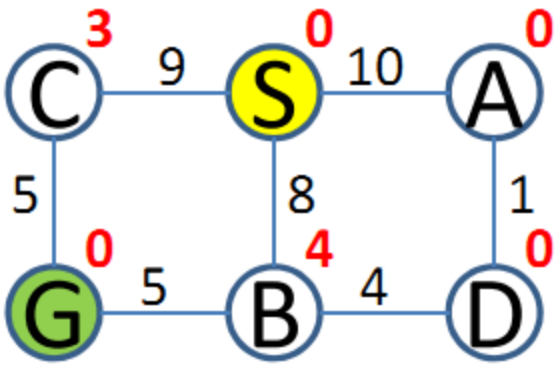
f_bound = 11
f_new = 12

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example (cont.)

- Find a path from S to G



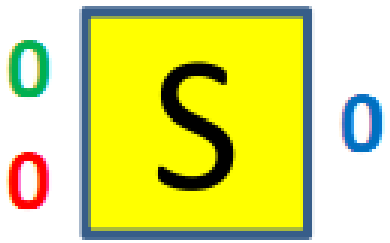


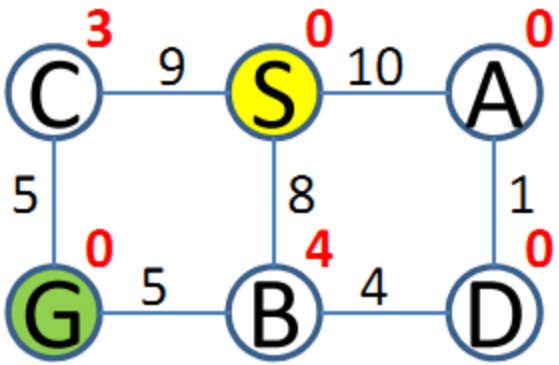
f_bound = 11
f_new = 12

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example – 4th iteration

- Find a path from S to G



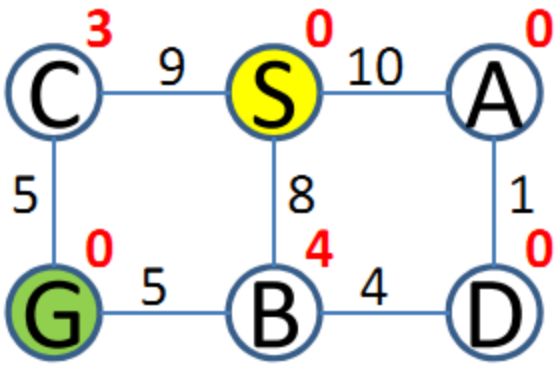


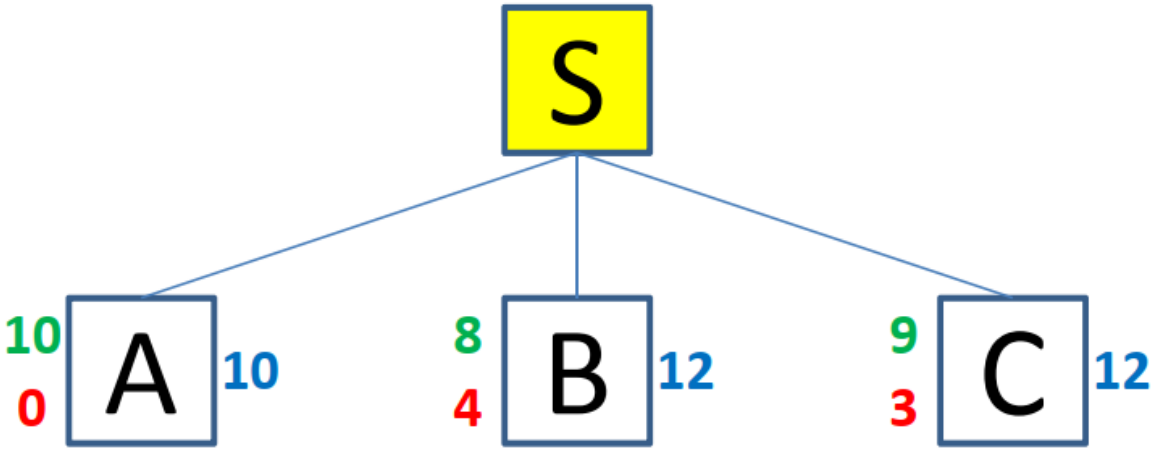
f_bound = 12
f_new = ∞

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example (cont.)

- Find a path from S to G



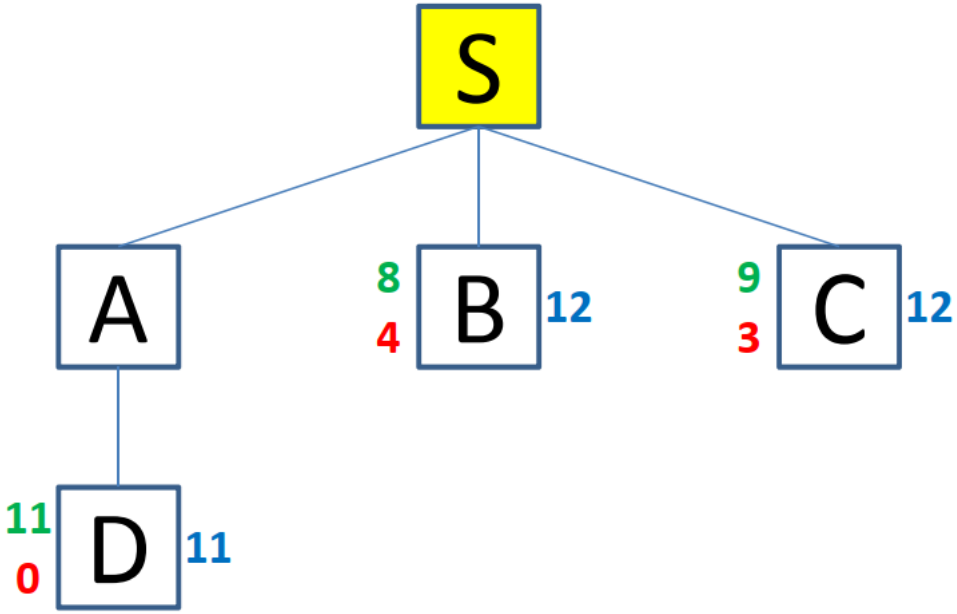


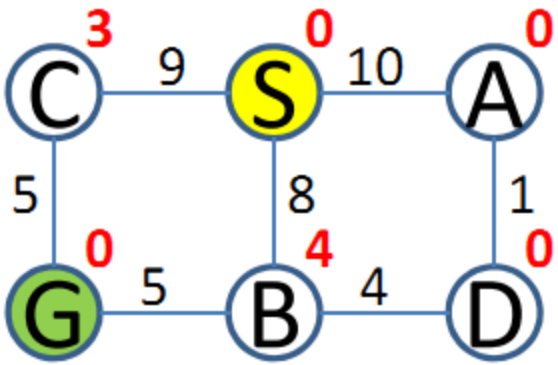
f_bound = 12
f_new = ∞

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example (cont.)

- Find a path from S to G



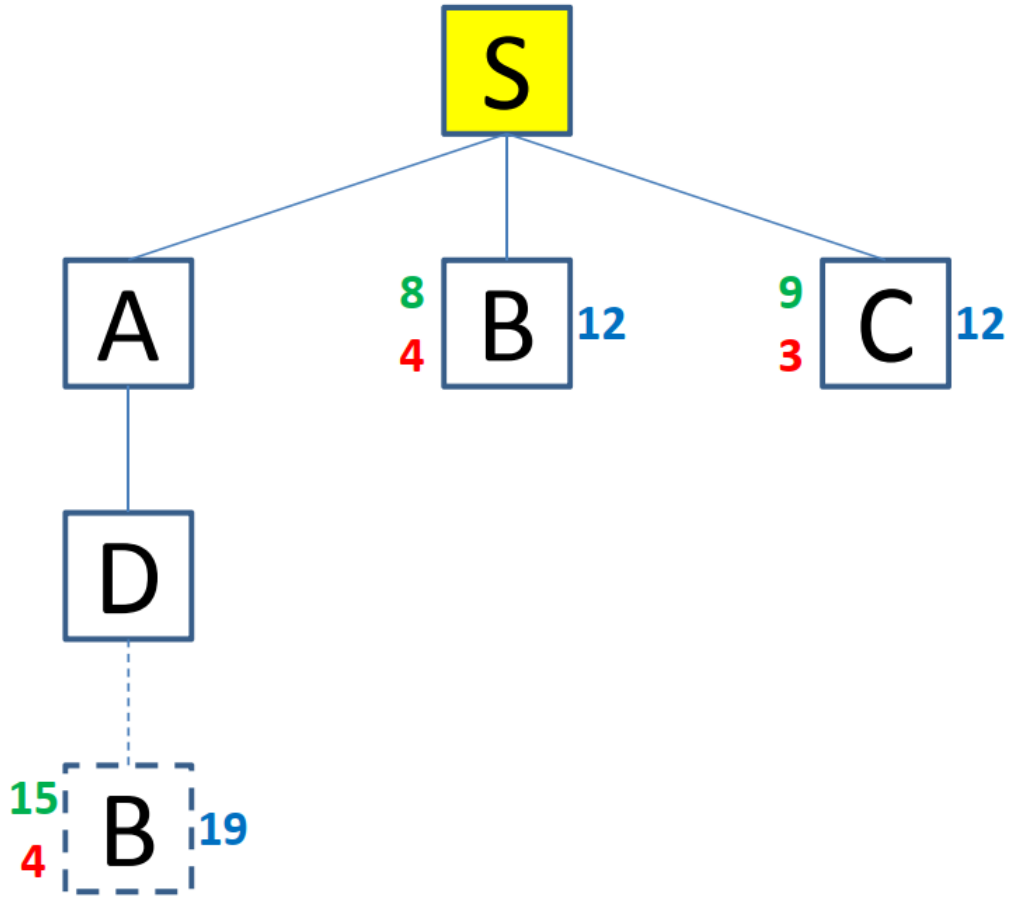


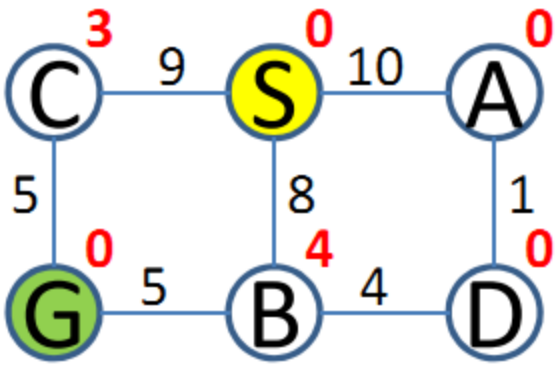
f_bound = 12
f_new = ∞

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example (cont.)

- Find a path from S to G



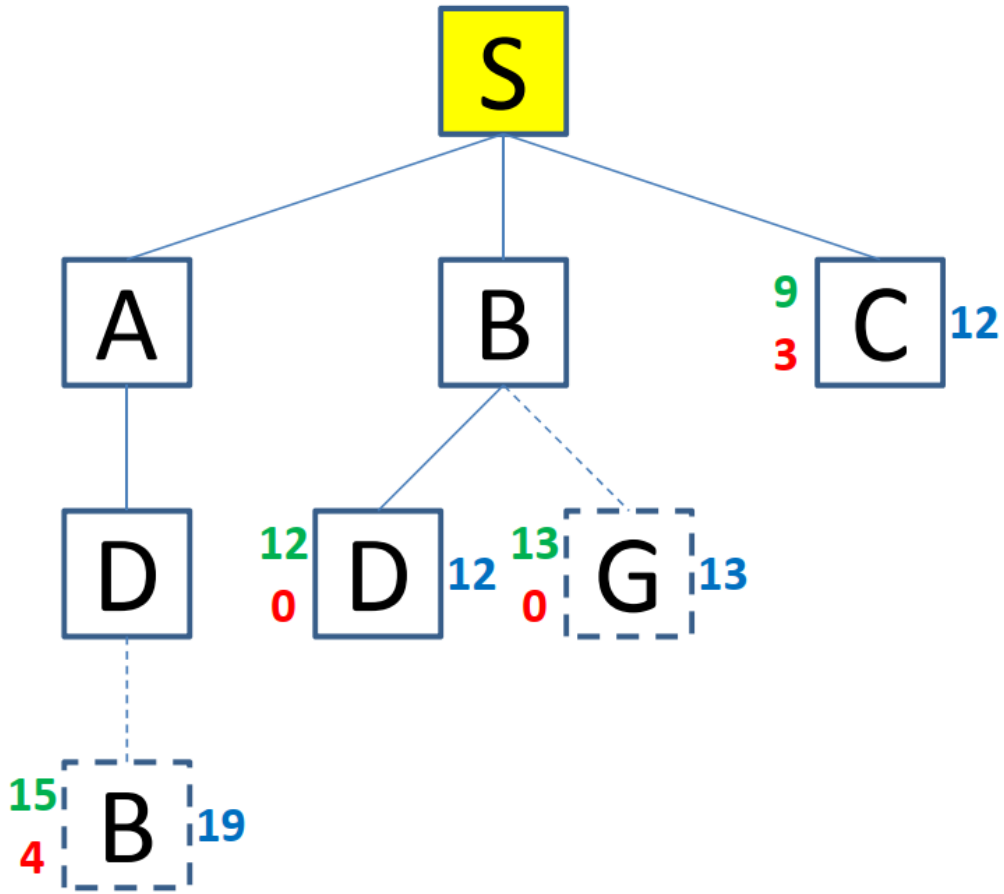


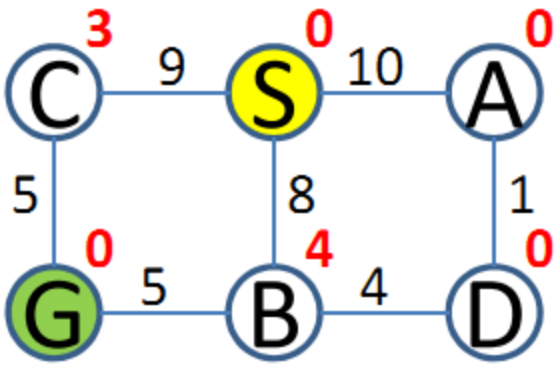
f_bound = 12
f_new = 19

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example (cont.)

- Find a path from S to G



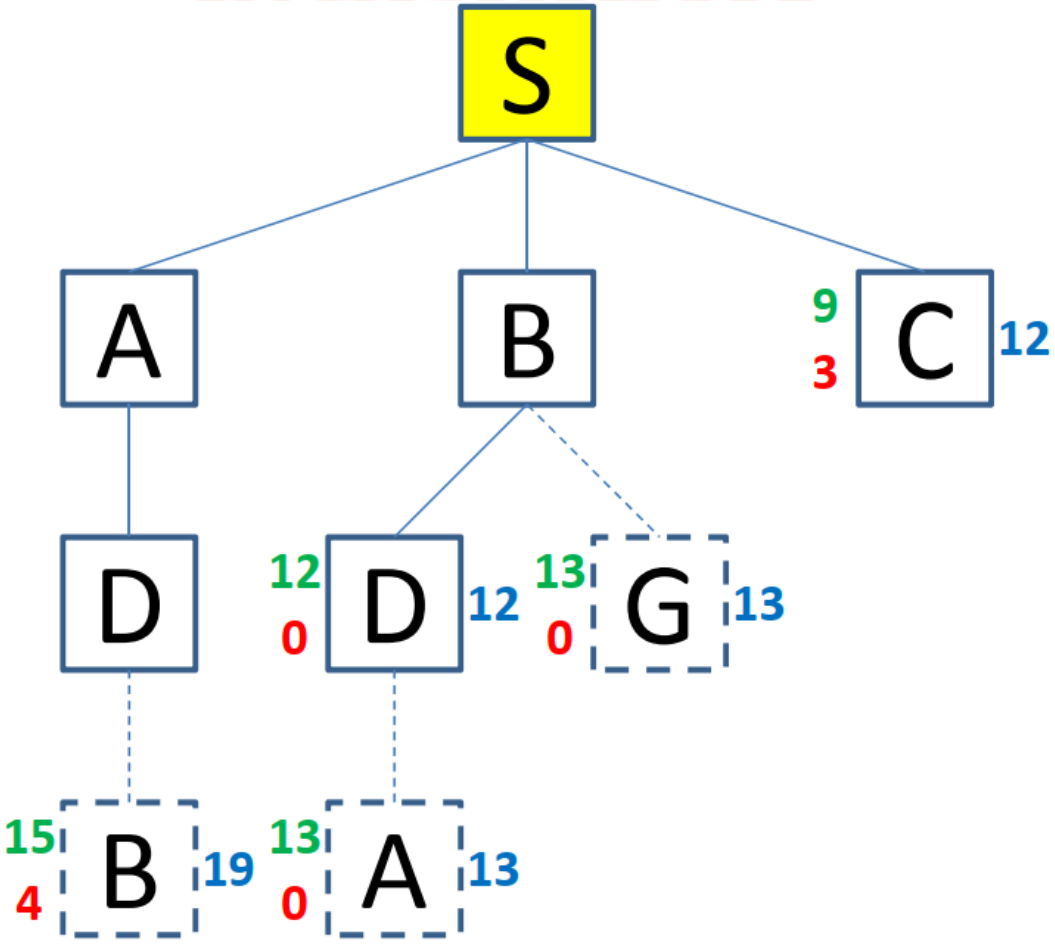


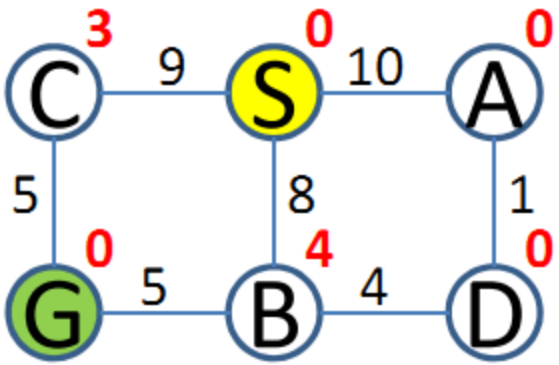
f_bound = 12
f_new = 13

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example (cont.)

- Find a path from S to G



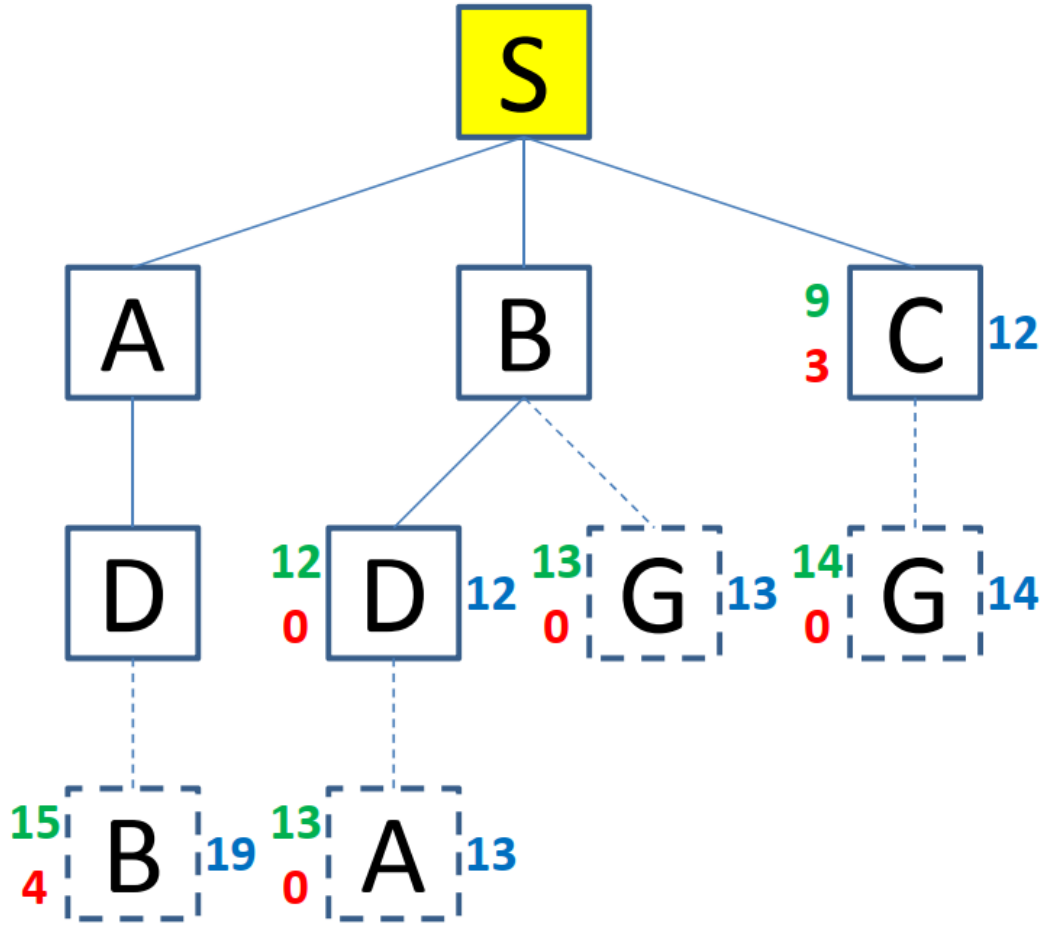


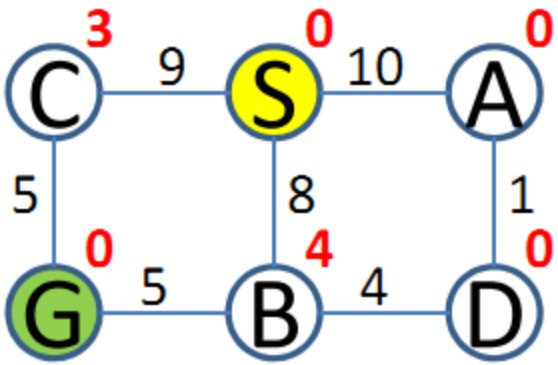
f_bound = 12
f_new = 13

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example (cont.)

- Find a path from S to G



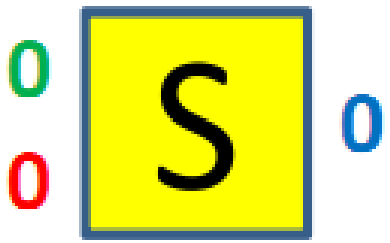


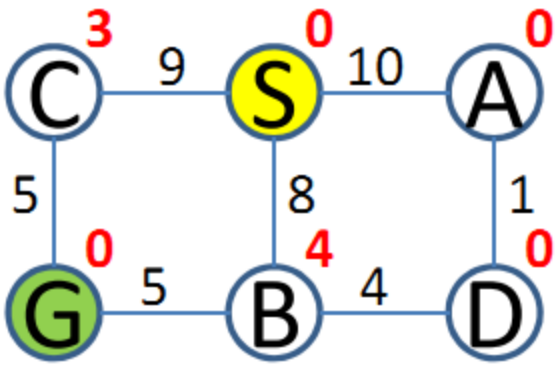
f_bound = 12
f_new = 13

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example – 5th iteration

- Find a path from S to G



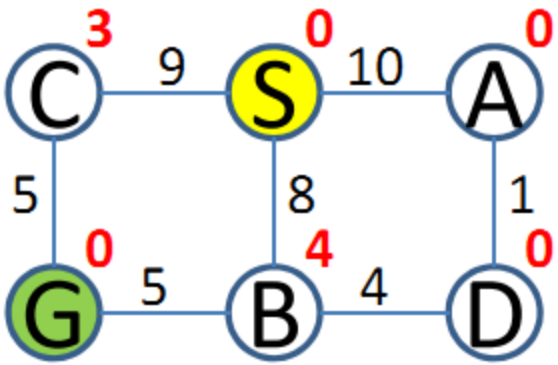


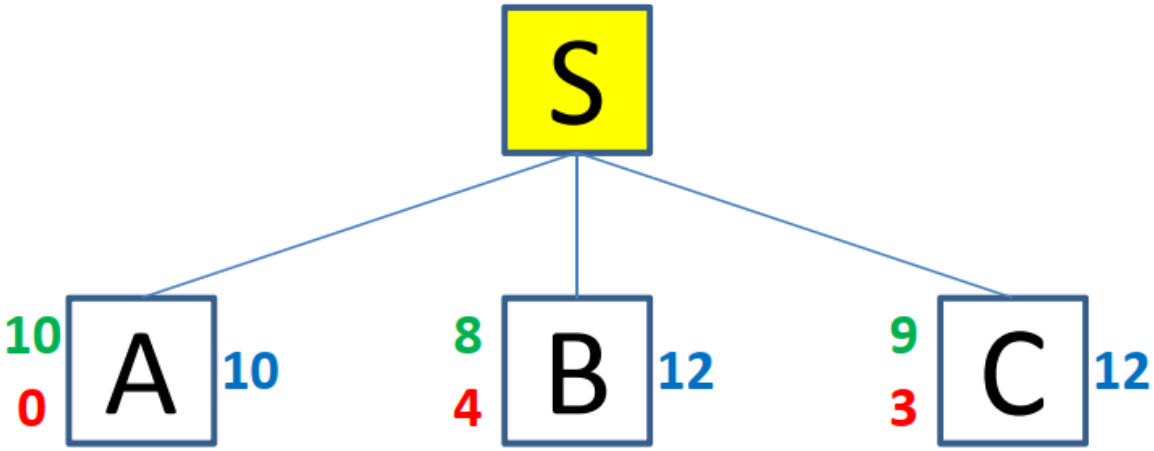
f_bound = 13
f_new = ∞

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example (cont.)

- Find a path from S to G



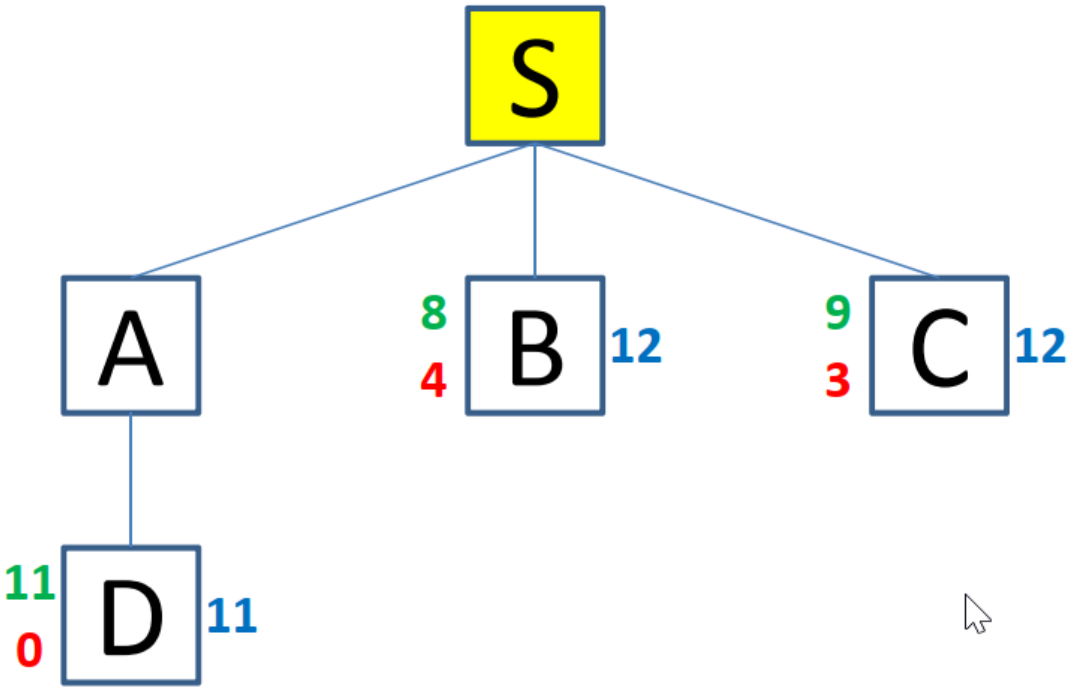


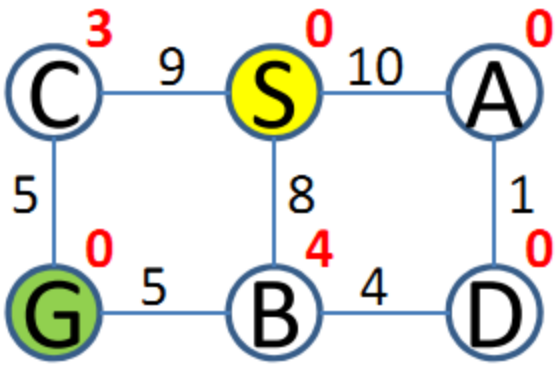
f_bound = 13
f_new = ∞

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example (cont.)

- Find a path from S to G



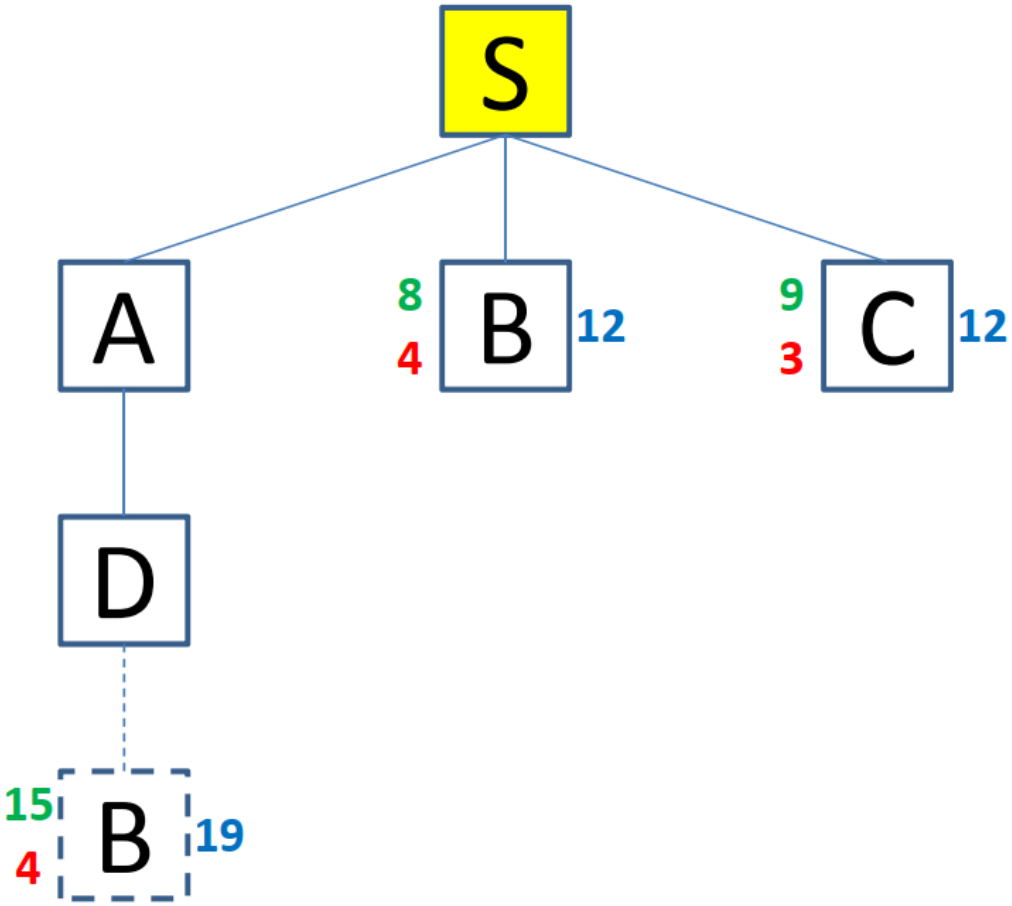


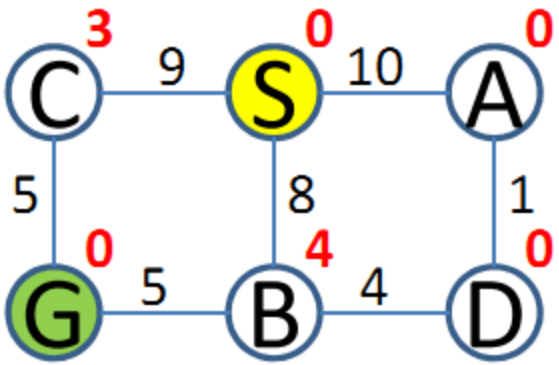
f_bound = 13
f_new = ∞

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example (cont.)

- Find a path from S to G



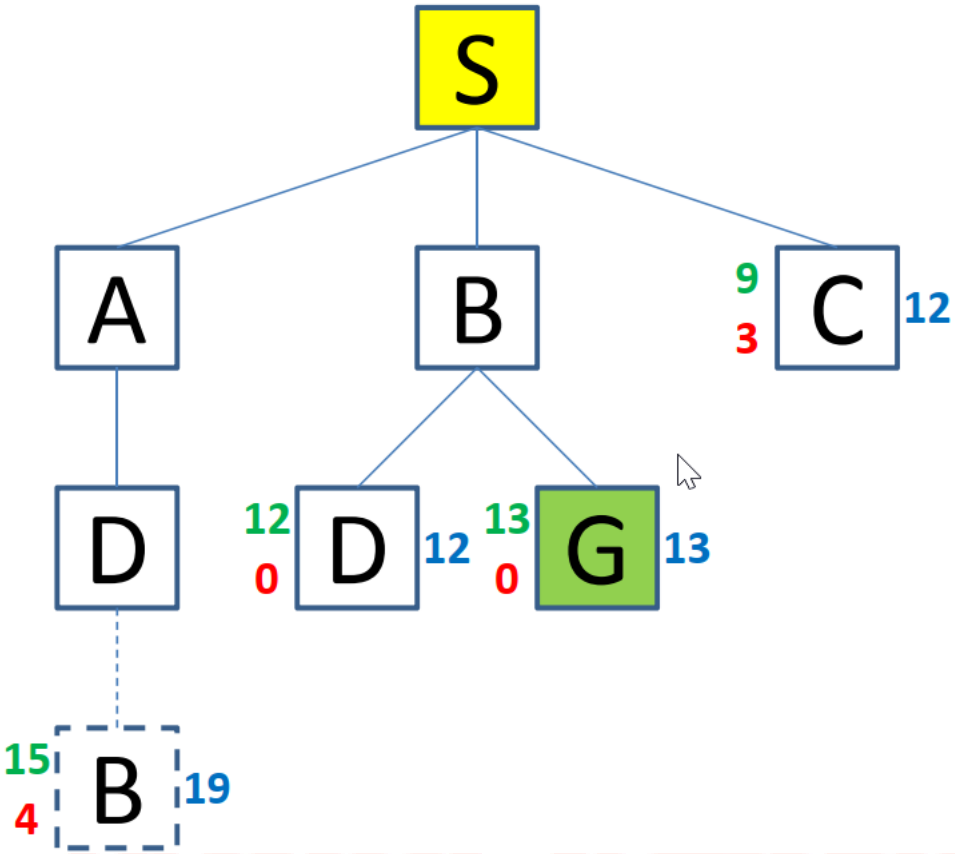


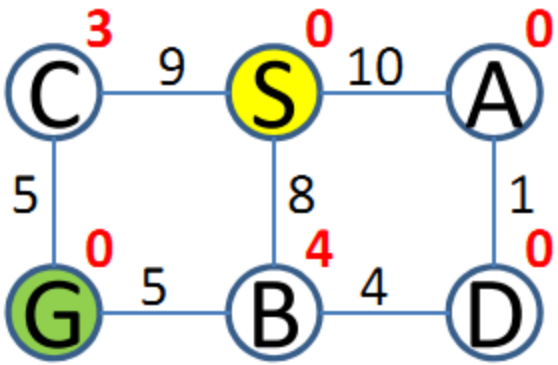
f_bound = 13
f_new = 19

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

IDA* Example (cont.)

- Find a path from S to G



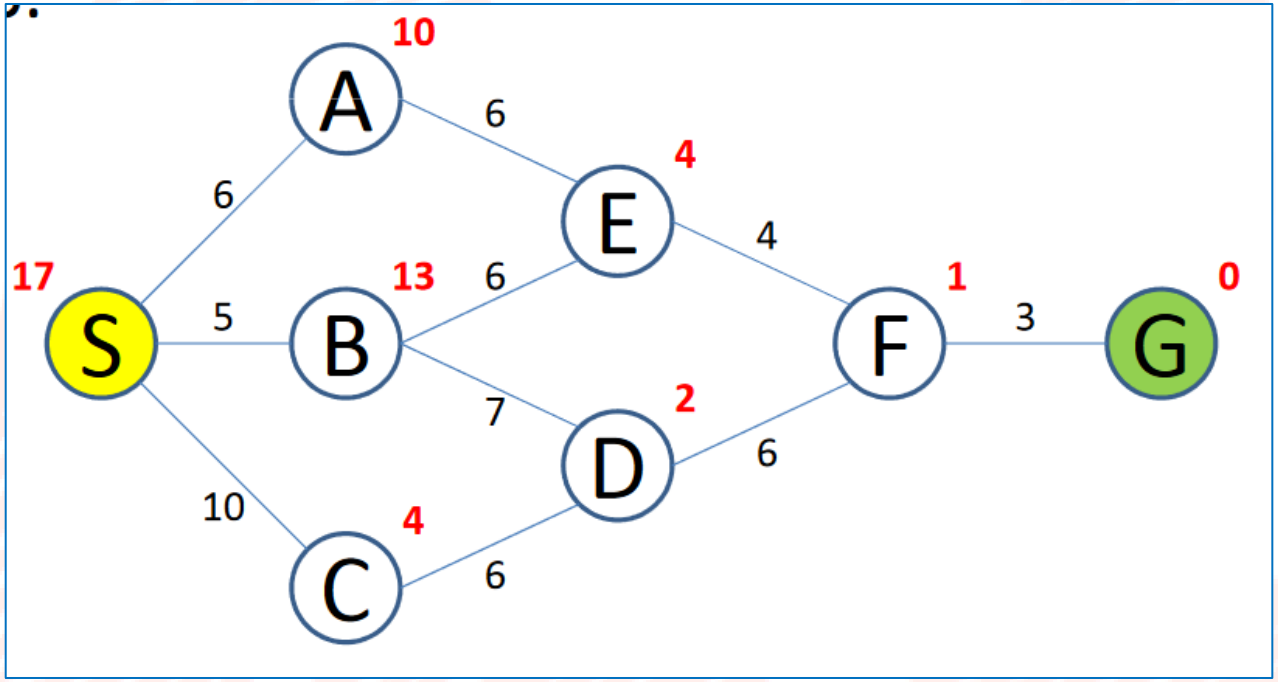


f_bound = 13
f_new = 19

Cutoff value: the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

Exercise

- Perform the IDA* Algorithm on the following figure.



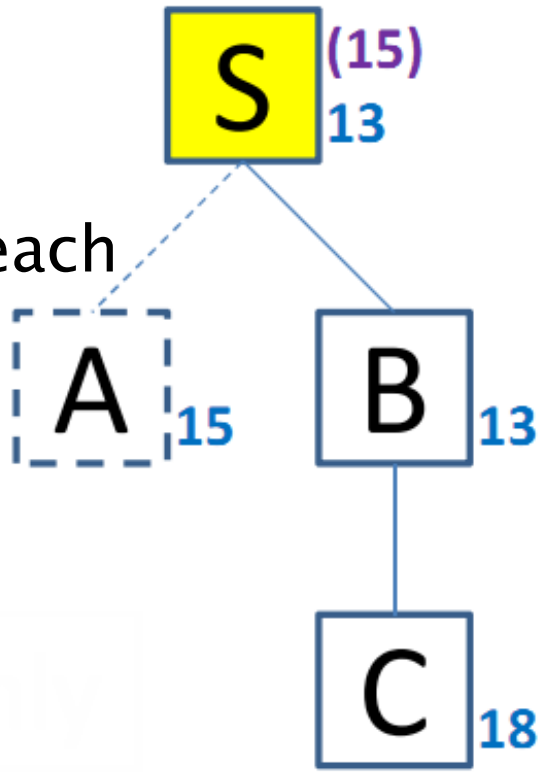
(Simplified) Memory-Bounded A*

- Use all available memory.
  - I.e. expand best leafs until available memory is full
  - When full, SMA* drops worst leaf node (highest f-value)
  - Like RFBS backup forgotten node to its parent

(Simplified) Memory-Bounded A*

- **Key Idea**:
  - IF memory full for extra node (C)
  - Remove highest f-value leaf (A)
  - Remember best-forgotten child in each parent node (15 in S)

(E.g. Memory of 3 nodes only)



(Simplified) Memory-Bounded A*

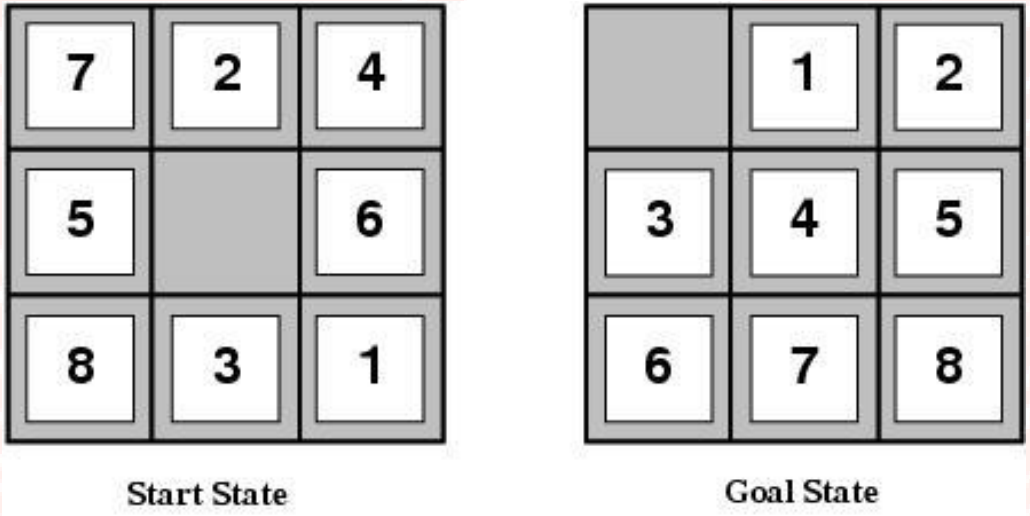
- What if all leaf nodes have the same f-value?
  - Same node could be selected for expansion and deletion.
  - SMA* solves this by expanding newest best leaf and deleting oldest worst leaf.
- SMA* is complete if solution is reachable, optimal if optimal solution is reachable.

Learning to search better

- All previous algorithms use *fixed strategies*.
- Agents can learn to improve their search by exploiting the *meta-level state space*.
  - Each meta-level state is a internal (computational) state of a program that is searching in *the object-level state space*.
  - In A* such a state consists of the current search tree
- A meta-level learning algorithm from experiences at the meta-level.

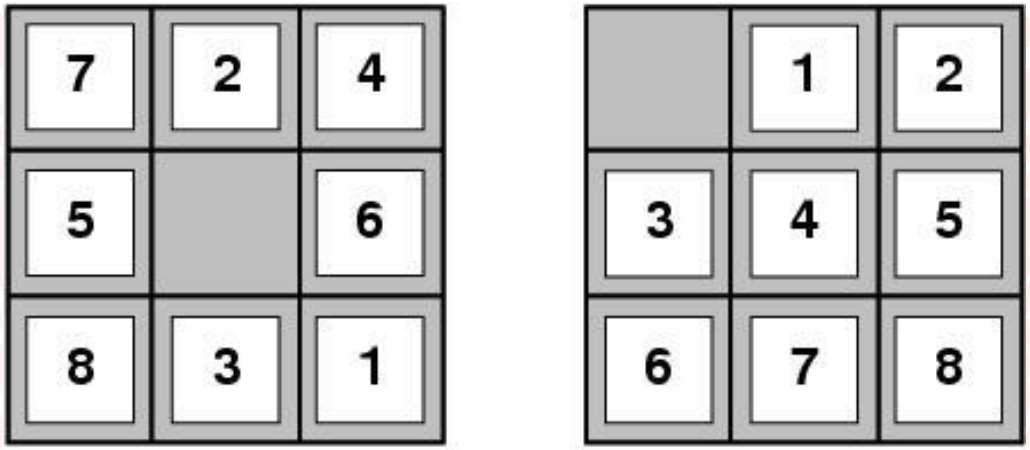
Heuristic functions

Invent Heuristic functions



- E.g. for the 8-puzzle
  - Avg. solution cost is about 22 steps (branching factor +/- 3)
  - Exhaustive search to depth 22: 3.1 x 10^{10} states.
  - A good heuristic function can *reduce the search process*.

Heuristic functions



- E.g. for the 8-puzzle knows two commonly used heuristics
  - H_1 = the number of misplaced tiles
    - $H_1(s)=8$
  - H_2 = the sum of the distances of the tiles from their goal positions (Manhattan distance).
    - $H_2(s)=3+1+2+2+2+3+3+2=18$

Heuristics in A*

- Use A* to solve the 8-puzzle:

Initial state:

1	2	3
4	6	8
7	5	

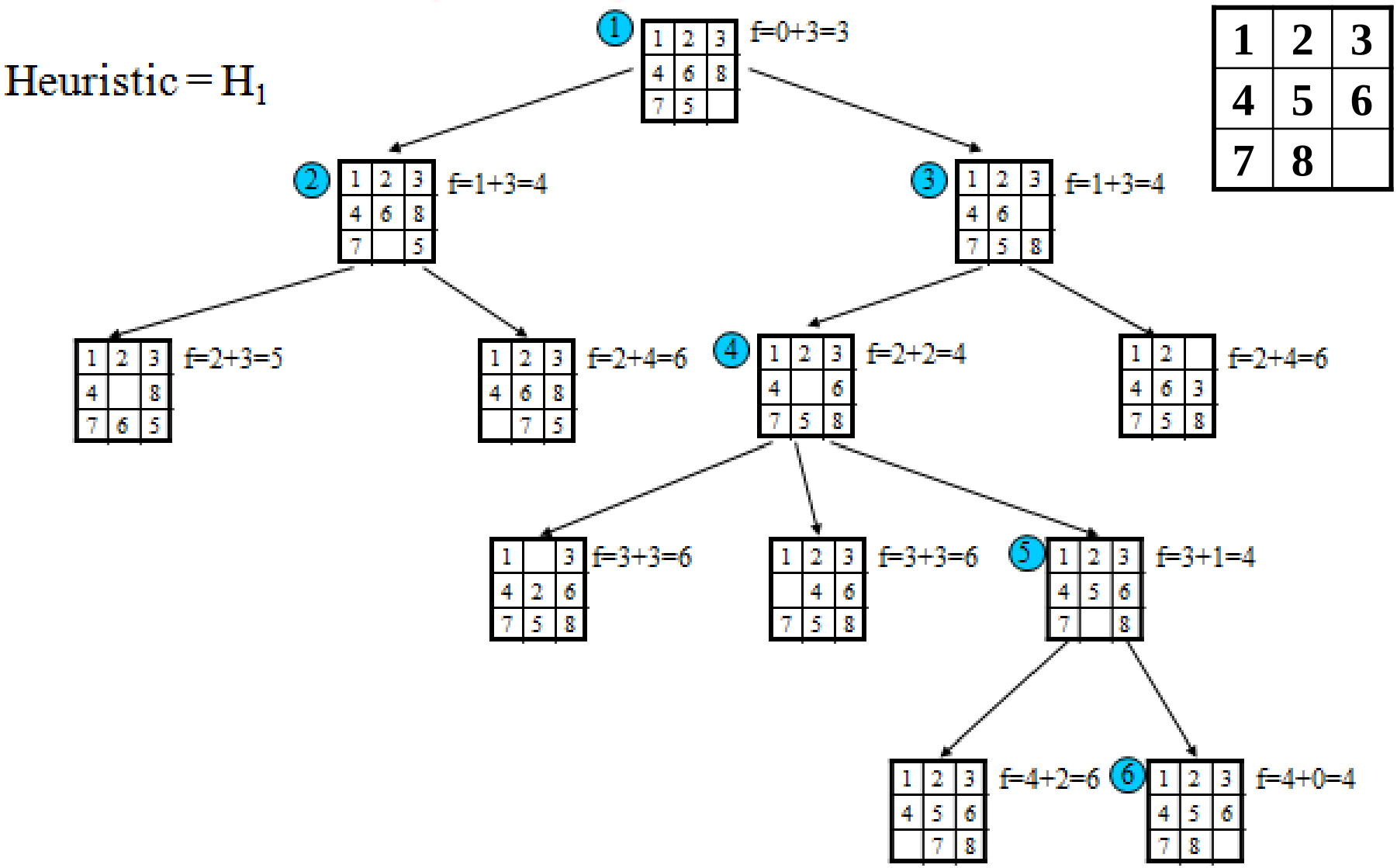
Goal state:

1	2	3
4	5	6
7	8	

path cost (g): the total number of moves made

- Consider these heuristics
  - H_1: The number of tiles out of place
  - H_2: Sum of distances of tiles from goal positions
- Ignore moves that return you to the previous state

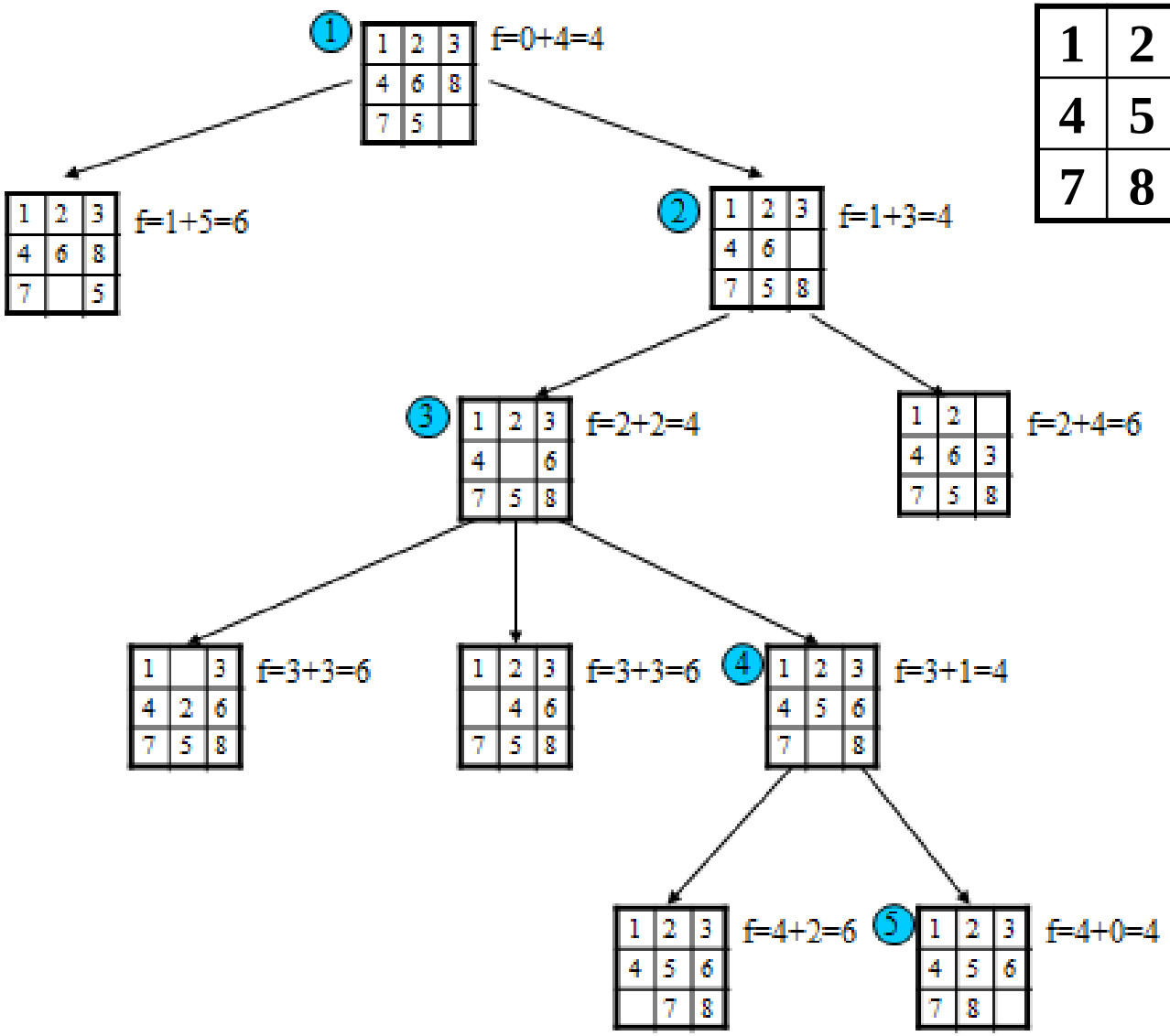
A* on 8-puzzle


Heuristic = H_1
1 2 3
4 5 6
7 8
1
f=0+3=3
2
f=1+3=4
3
f=1+3=4
f=2+3=5
f=2+4=6
4
f=2+2=4
f=2+4=6
f=3+3=6
f=3+3=6
5
f=3+1=4
f=4+2=6
6
f=4+0=4

A* on 8-puzzle

1	2	3
4	5	6
7	8	

Heuristic = H_2



Heuristic quality

- Effective branching factor b*
  - Is the branching factor that a uniform tree of depth d would have in order to contain N+1 nodes.

$$N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d.$$

- Measure is fairly constant for sufficiently hard problems.
  - Can thus provide a good guide to the heuristic's overall usefulness.
  - A good value of b* is 1.

Heuristic quality and dominance

- Typical search costs (average number of nodes expanded)

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

- If $h_2(n) \geq h_1(n)$ for all n (both admissible) then h_2 dominates h_1 and h_2 is better for search

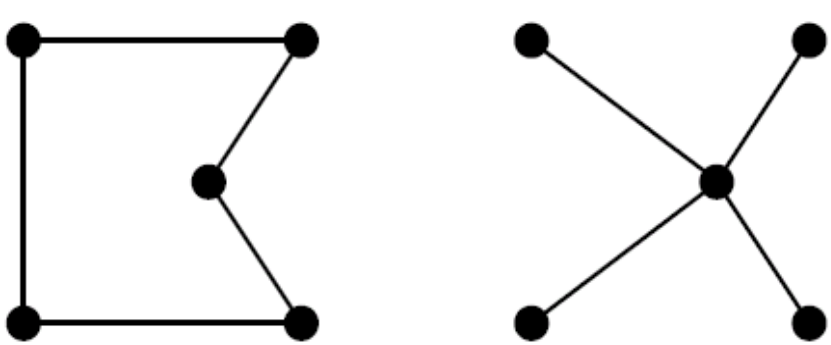
- Given any admissible heuristics ha(n), hb(n) h(n) = max(ha(n), hb(n)) is also admissible and dominates ha, hb

Admissible heuristics – Relaxed problem

- A problem with fewer restrictions on the actions is called a relaxed problem
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem:
  - Relaxed 8-puzzle for h_1 : a tile can move anywhere
    As a result, $h_1(n)$ gives the shortest solution
  - Relaxed 8-puzzle for h_2 : a tile can move to any adjacent square. As a result, $h_2(n)$ gives the shortest solution.
- The optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem.

Relaxed problems (con't)

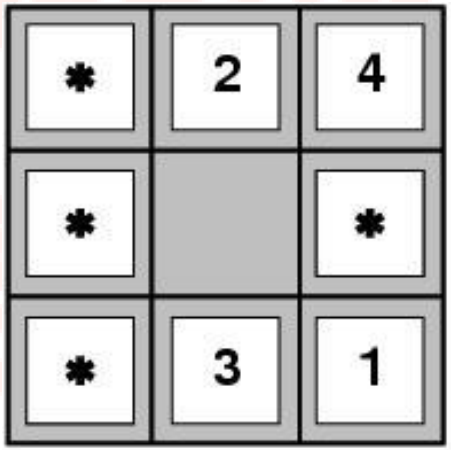
- Well-known example:
  Travelling Salesperson Problem (TSP)
- Find the shortest tour visiting all cities exactly once



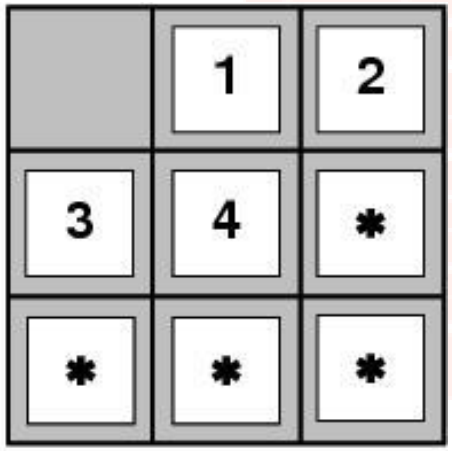
- Minimum spanning tree can be computed in $O(n^2)$ and is a lower bound on the shortest (open) tour

Inventing admissible heuristics

- Admissible heuristics can also be derived from the solution cost of a subproblem of a given problem.
- This cost is a lower bound on the cost of the real problem.
- Pattern databases store the exact solution to for every possible subproblem instance.
  - The complete heuristic is constructed using the patterns in the DB



Start State



Goal State

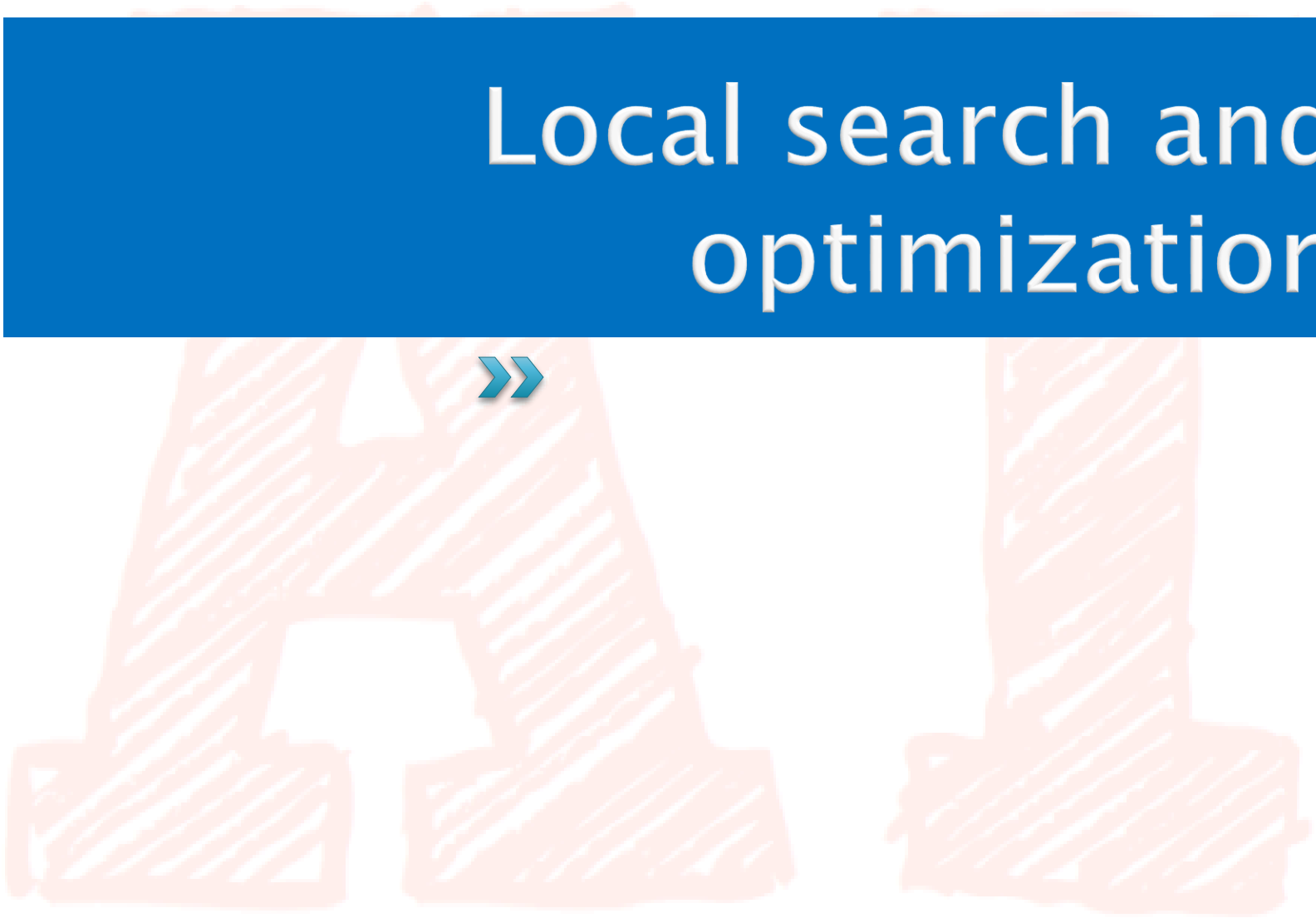
Inventing admissible heuristics

- Another way to find an admissible heuristic is through learning from experience:
  - Experience = solving lots of 8-puzzles
  - An inductive learning algorithm can be used to predict costs for other states that arise during search.

Summary

- Heuristic functions estimate costs of shortest paths
- Good heuristics can dramatically reduce search cost
- Greedy best-first search expands lowest h
  - incomplete and not always optimal
- A* search expands lowest g + h
  - complete and optimal
  - also optimally eficient (up to tie-breaks, for forward search)
- Admissible heuristics can be derived from exact solution of relaxed problems

Local search and optimization



Local search algorithms

- In many optimization problems,
  - the path to the goal is irrelevant;
  - the goal state itself is the solution
- State space = *set of "complete" configurations*
  - Find optimal configuration, e.g., Travel Saleman Problem
  - or, find configuration satisfying constraints,
    - e.g., timetable, n-queens
- In such cases, we can use local search algorithms
  keep a single "current" state, try to improve it

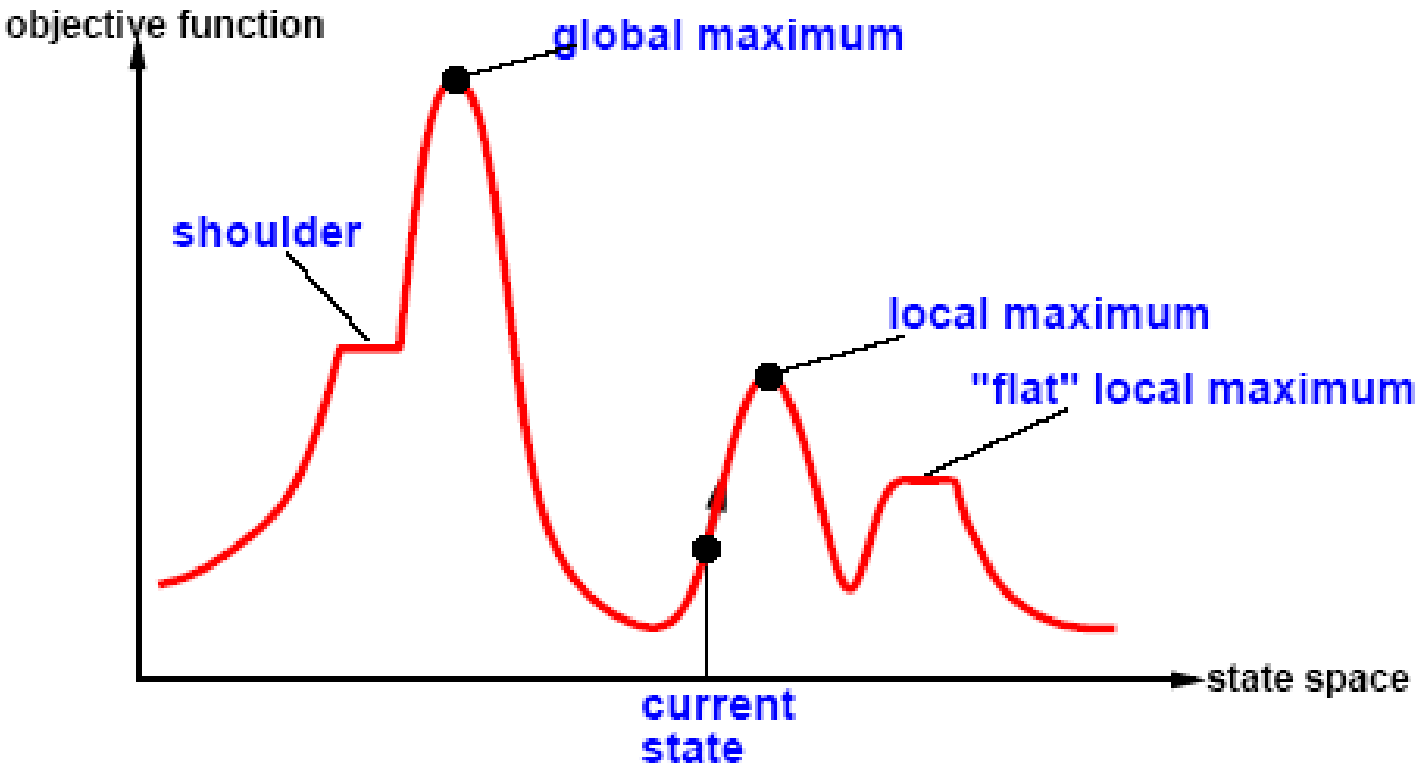
Hill-climbing search

- "is a loop that continuously moves in the direction of increasing value"
  - It terminates when a **peak is reached**.

- Hill climbing does not look ahead beyond the immediate neighbors of the current state.
  - Hill-climbing chooses randomly among the set of best successors, if there is more than one.
  - Hill-climbing a.k.a. greedy local search

Hill-climbing search (cont.)

- Problem: depending on initial state, can get stuck in local maxima



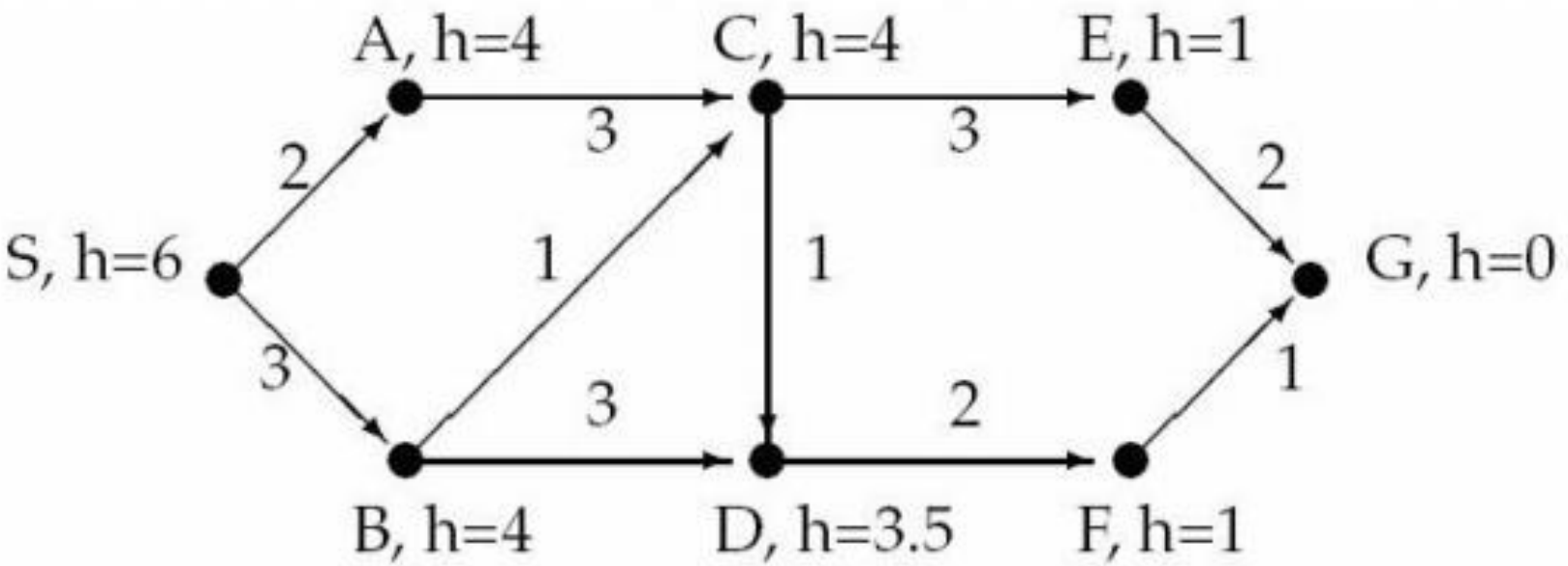
Hill-climbing search (cont.)

"Like climbing Everest in thick fog with amnesia"

```
function HILL-CLIMBING( problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
    end
```

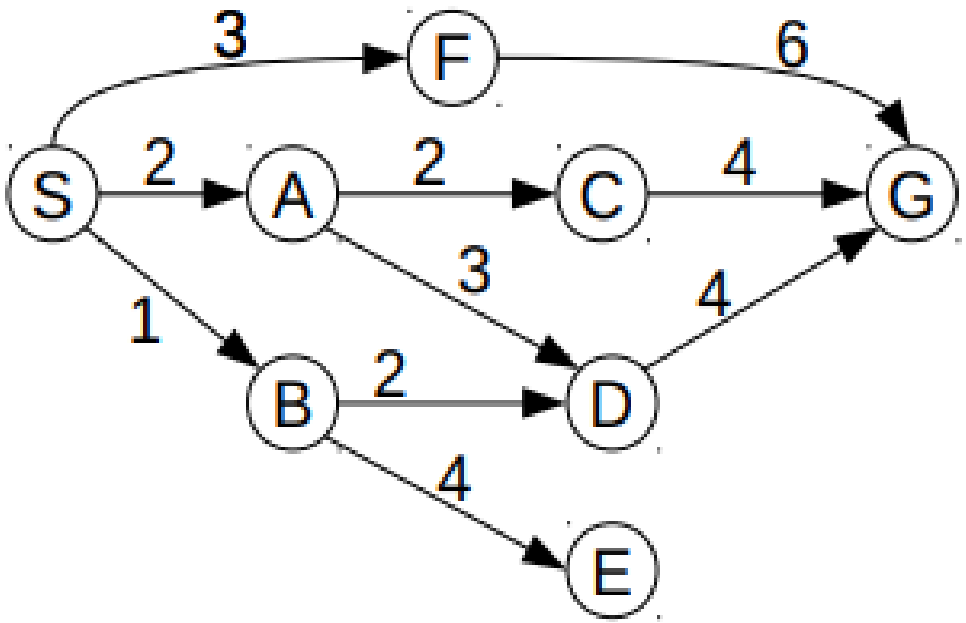
Example: Hill climbing

- Using Hill climbing to find a path from S to G (using h values)



Example: Hill climbing (cont.)

- Using Hill climbing to find a path from S to G (using h values)

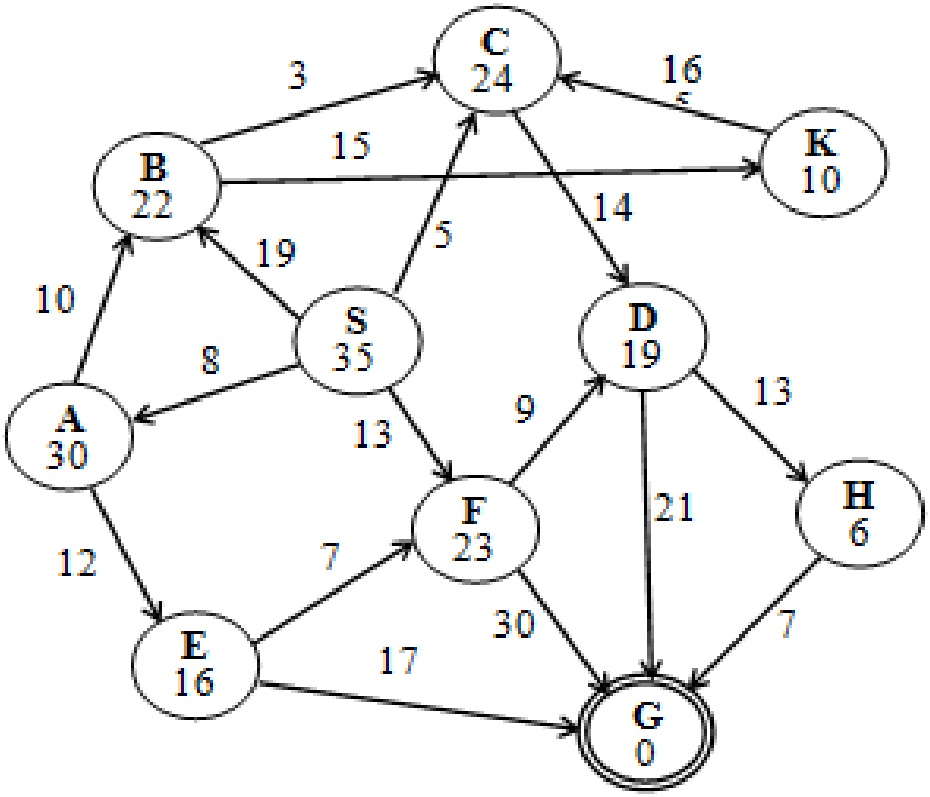


heuristic function (goal state: G)

S	A	B	C	D	E	F	G
6	4	5	2	2	8	4	0

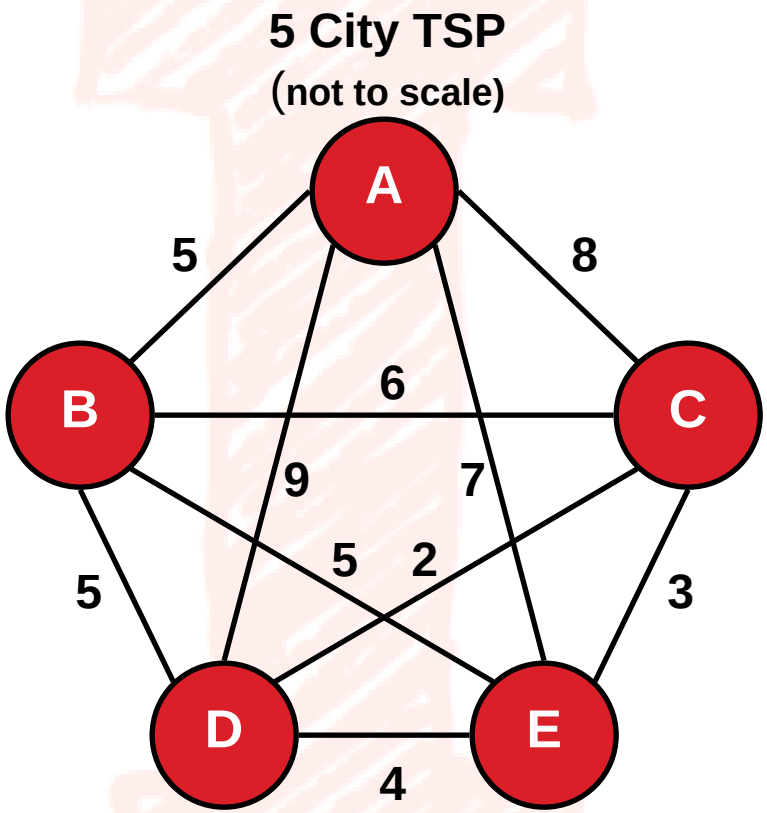
Example: Hill climbing (cont.)

- Using Hill climbing to find a path from S to G (using h values)



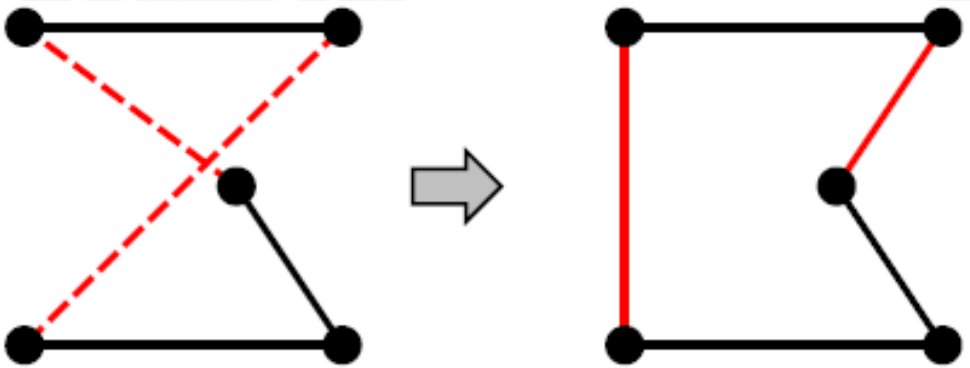
Example: Traveling Salesperson Problem

- A salesman wants to visit a list of cities
  - stopping in each city only once
  - returning to the first city
  - traveling the shortest distance
- A solution is a permutation of cities, called a tour
  - e.g. A – B – C – D – E – A length 24
- How many solutions exist?
  - (n–1)!/2 where n = # of cities
  - n = 5 results in 12 tours
  - n = 10 results in 181440 tours
  - n = 20 results in ~$6*10^{16}$ tours



Example: TSP (cont.)

- Start with any complete tour, perform pairwise exchanges



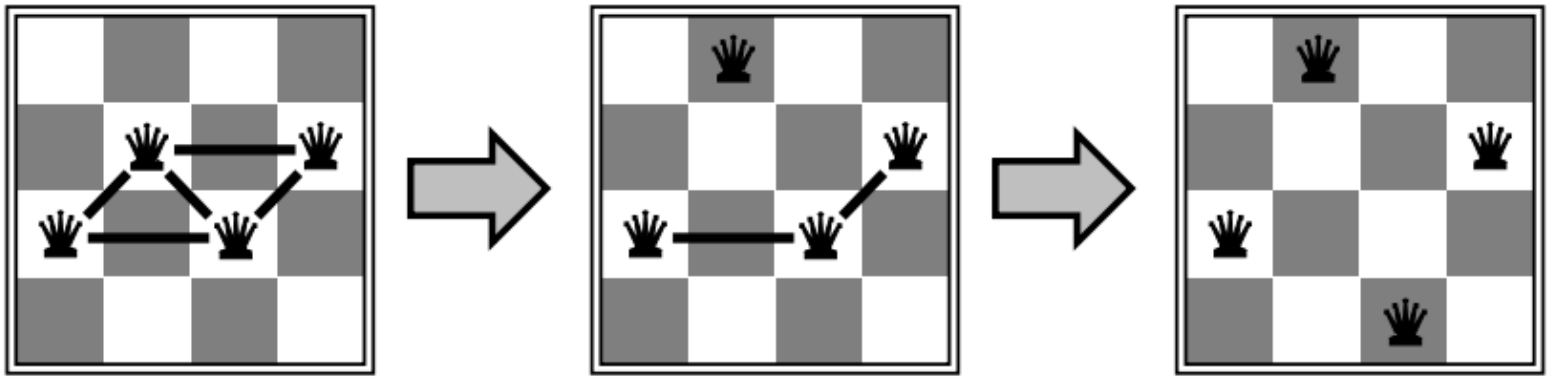
- Variants of this approach get within 1% of optimal very quickly with thousands of cities

Example: Travelling Salesperson Problem

- An operator is needed to transform one solution to another.
- What operators work for TSP?
  - two-swap (common)
    - take two cities and swap their location in the tour
    - e.g. A – B – C – D – E
      swap(A, D) yields D – B – C – A – E
  - two-interchange
    - reverse the path between two cities
    - e.g. A – B – C – D – E
      interchange(A, D) yields D – C – B – A – E
  - both work since graph is fully connected

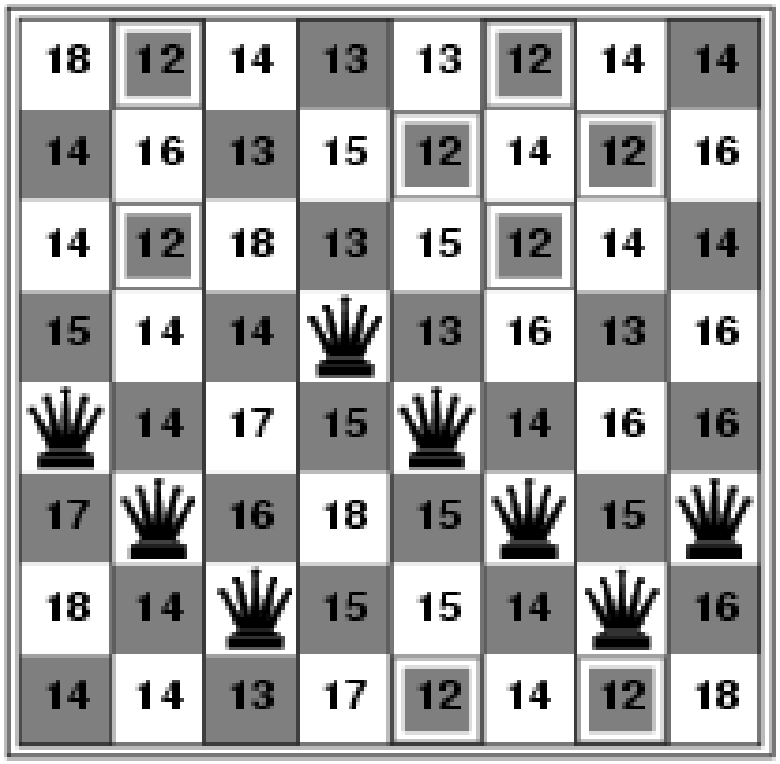
Example: n-queens

- Put n queens on an n × n board with no two queens on the same row, column, or diagonal
- Move a queen to reduce number of conflicts



- Almost always solves n-queens problems almost instantaneously for very large n, e.g., n = 1million.

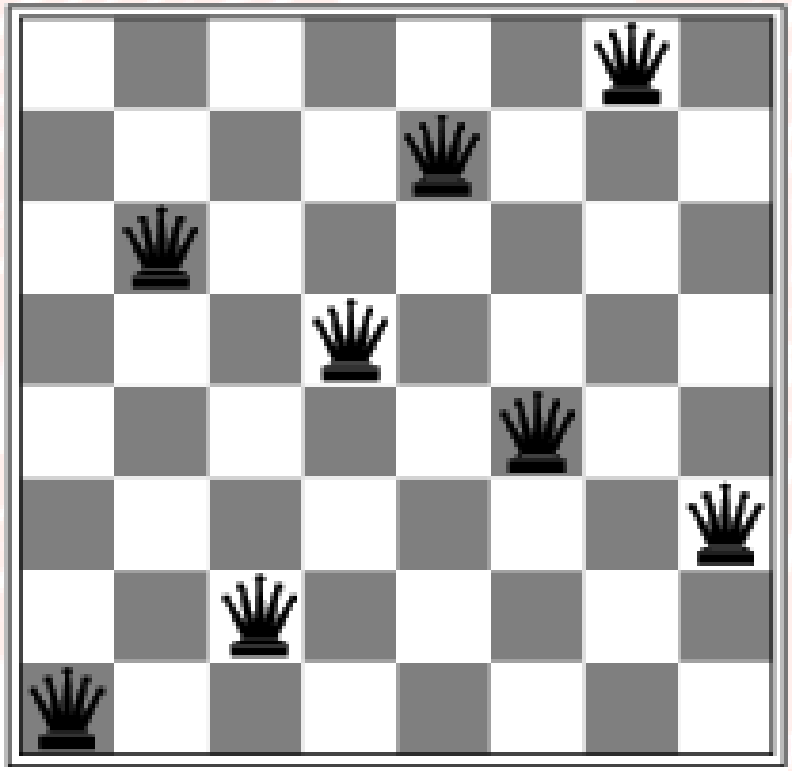
Hill-climbing search: 8-queens problem



h=17

- h = number of pairs of queens that are attacking each other, either directly or indirectly

Hill-climbing search: 8-queens problem



- A local minimum with h = 1

Hill-climbing variations

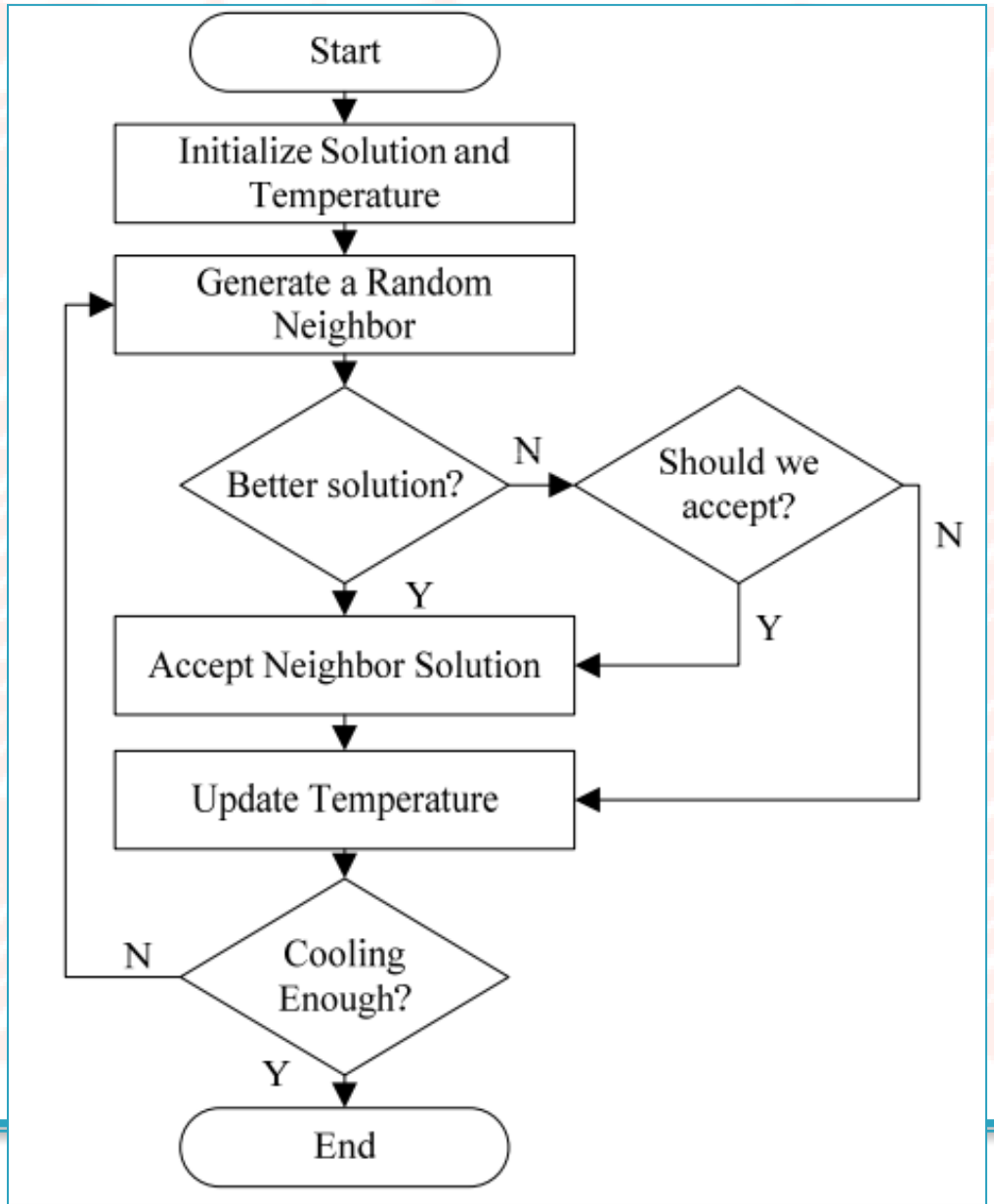
- **Stochastic hill-climbing**
  - Random selection among the uphill moves.
  - The selection probability can vary with the steepness of the uphill move.
- **First-choice hill-climbing**
  - implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.
- There are several ways we can try to avoid local optima and find more globally optimal solutions:
  - **Random-restart hill-climbing**, "If at first you don't succeed, try, try again."
  - **Simulated Annealing**
  - **Tabu Search**

Simulated Annealing

What is SA?

- SA algorithm is one of the most preferred heuristic methods for solving the optimization problems
  - inspiring the annealing procedure of the metal working
- SA a probabilistic technique for approximating the global optimum of a given function
- One of the advantages of SA over other methods is the ability to reduced being trapped in local minima

Simulated Annealing search



Simulated Annealing search (cont.)

- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their size and frequency

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] – VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability e^(Δ E/T)
```

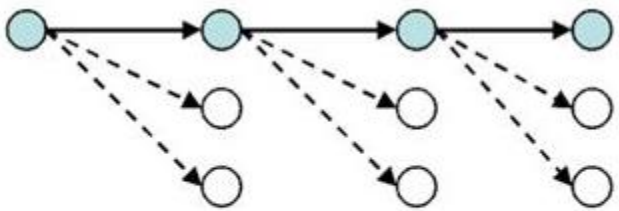
Properties of SA search

- One can prove:
  - If T decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1.

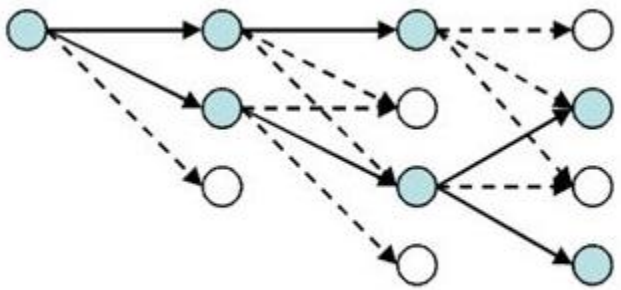
- Widely used in VLSI layout (Very-large-scale integration: is the process of creating an integrated circuit), airline scheduling, etc.

Local beam search

- Keep track of *k* states rather than just one
- Start with *k* randomly generated states
- At each iteration, all the successors of all *k* states are generated (k^d)
- If any one is a **goal state**, stop;
- Else select the *k* best successors from the complete list and repeat.



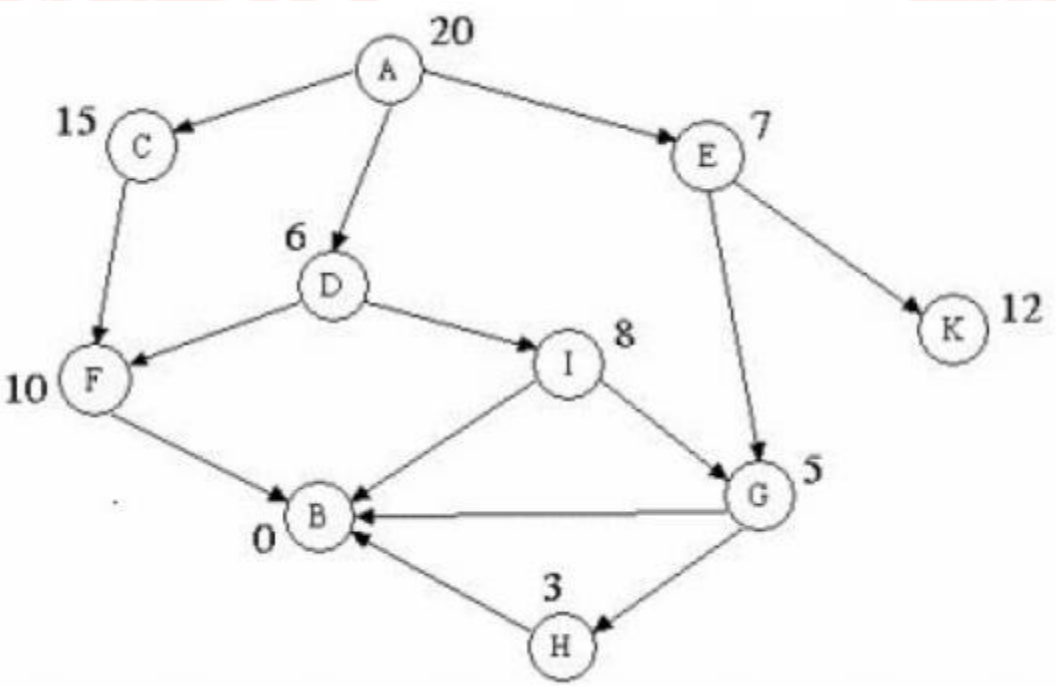
Greedy search



Beam search

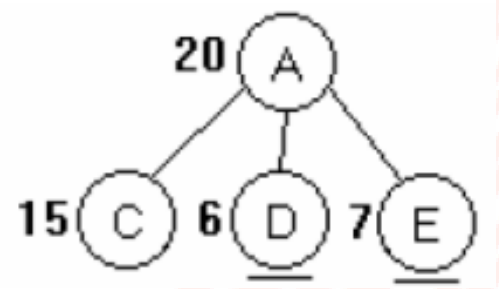
Local beam search (cont.)

- How to get B from A using Beam search (k=2)



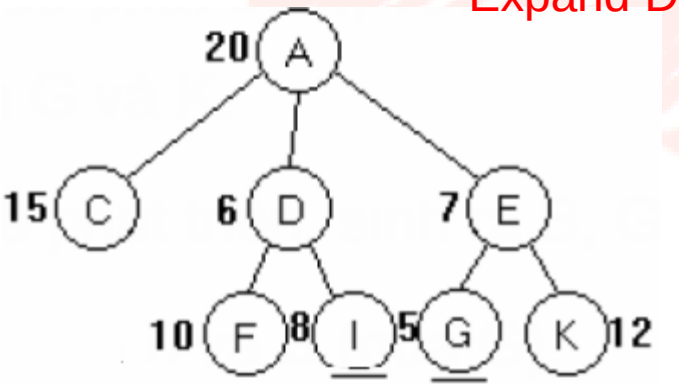
Local beam search (cont.)

- Start with A:

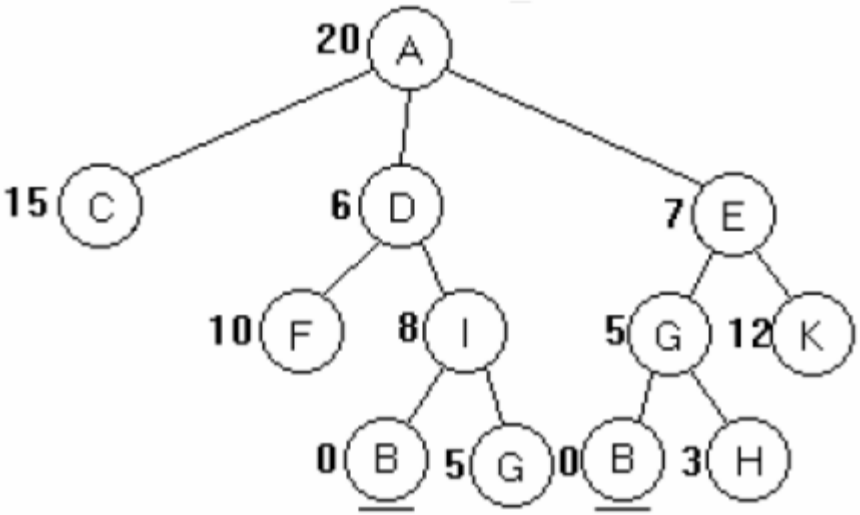


Select D, E

Expand D, E

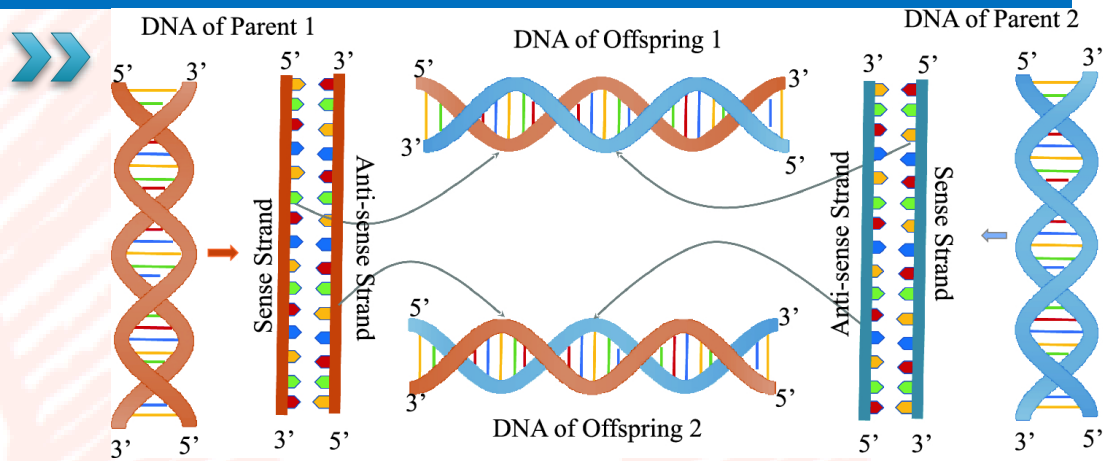


Select I, G



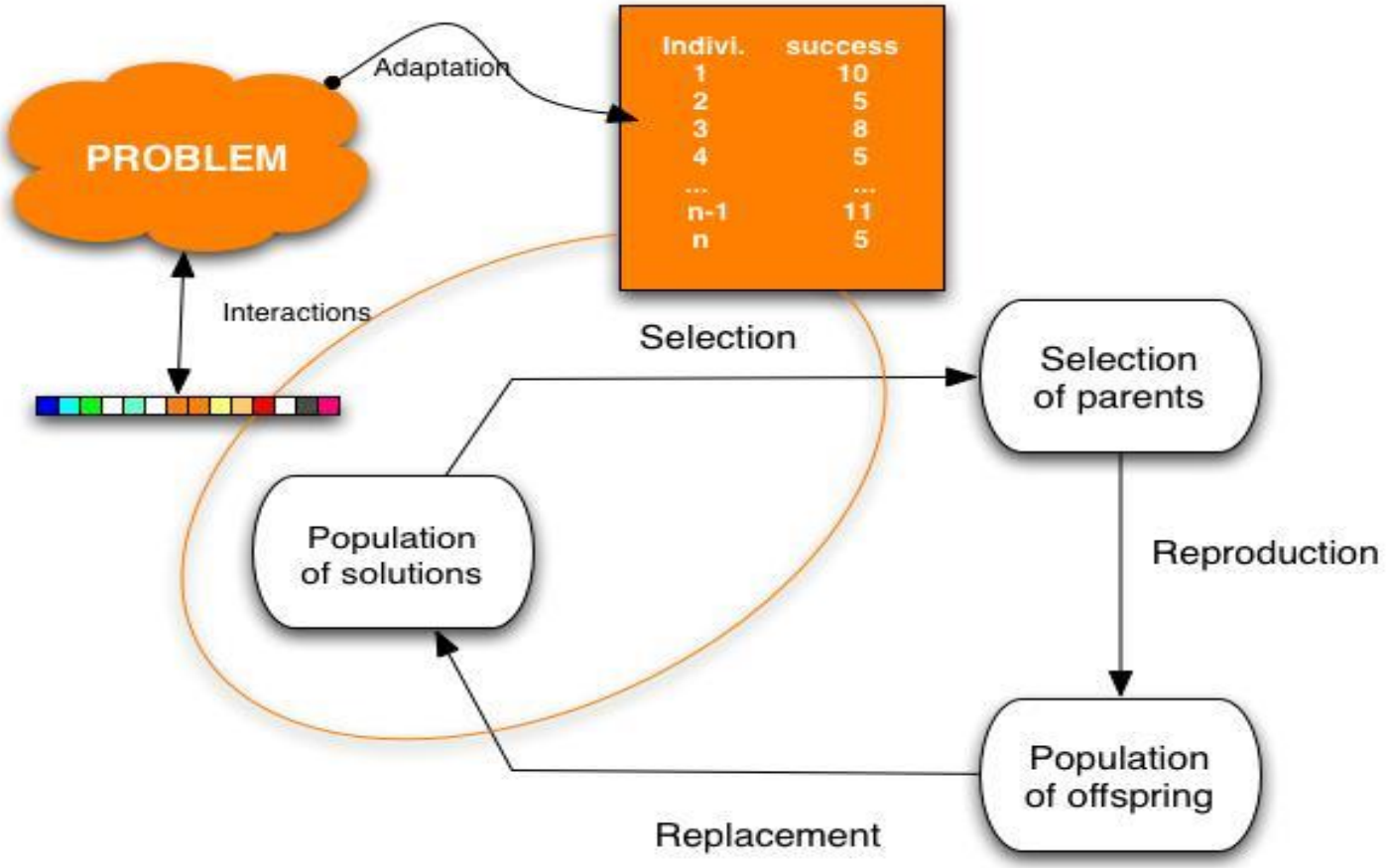
Expand I, G

Genetic algorithms



Genetic algorithms

- Variant of local beam search with *sexual recombination.*



Genetic algorithms (cont.)

- A successor state is generated by combining two parent states
- Start with *k* randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function
  - fitness function
  - Higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation

Genetic algorithms (cont.)

- Encoding methods:
  - Binary encoding
    - Example: Knapsack problem

Encoding: 0 = not exist, 1 = exist in the Knapsack

Chromosome: 1010110

Item.	1	2	3	4	5	6	7
Chro	1	0	1	0	1	1	0
Exist?	y	n	y	n	y	y	n

Genetic algorithms (cont.)

- Encoding methods:
  - Permutaion encoding
    - Example: TSP with 9 cities

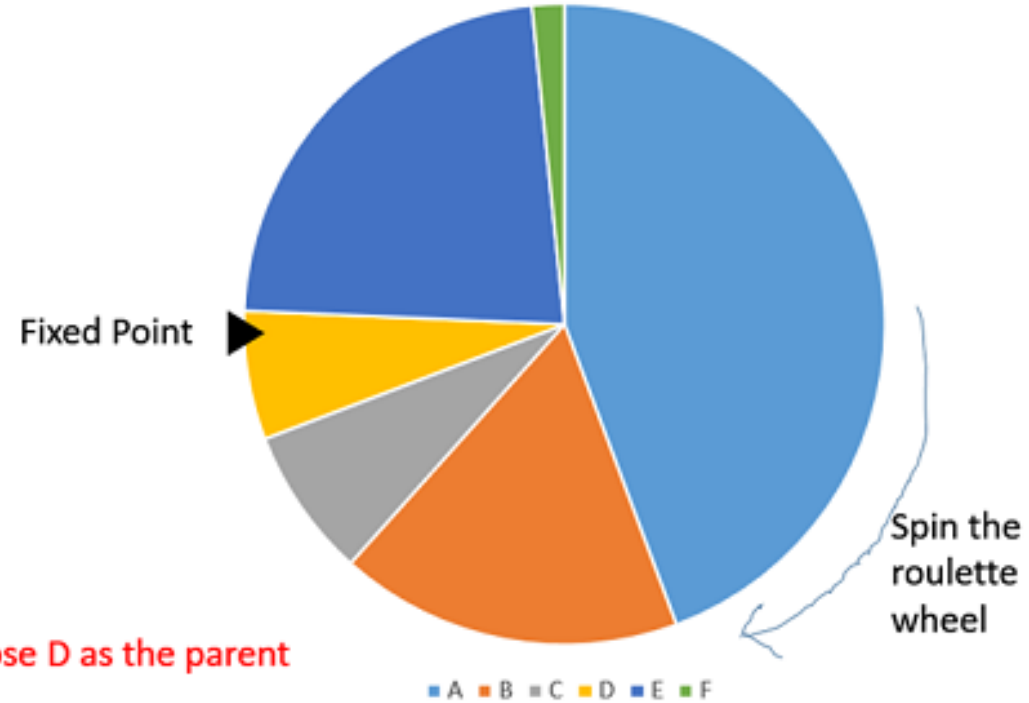
A	1	5	3	2	6	4	7	9	8
B	8	5	6	7	2	3	1	4	9

Genetic algorithms (cont.)

- Selection, use a fitness function to rank the individuals of the population
- Reproduction, define a crossover operator which takes state descriptions of individuals and combines them to create new ones
- Mutation, merely choose individuals in the population and alter part of its state

Roulette Wheel Selection

- A fixed point is chosen on the wheel circumference as shown and the wheel is rotated.
- The region of the wheel which comes in front of the fixed point is chosen as the parent.

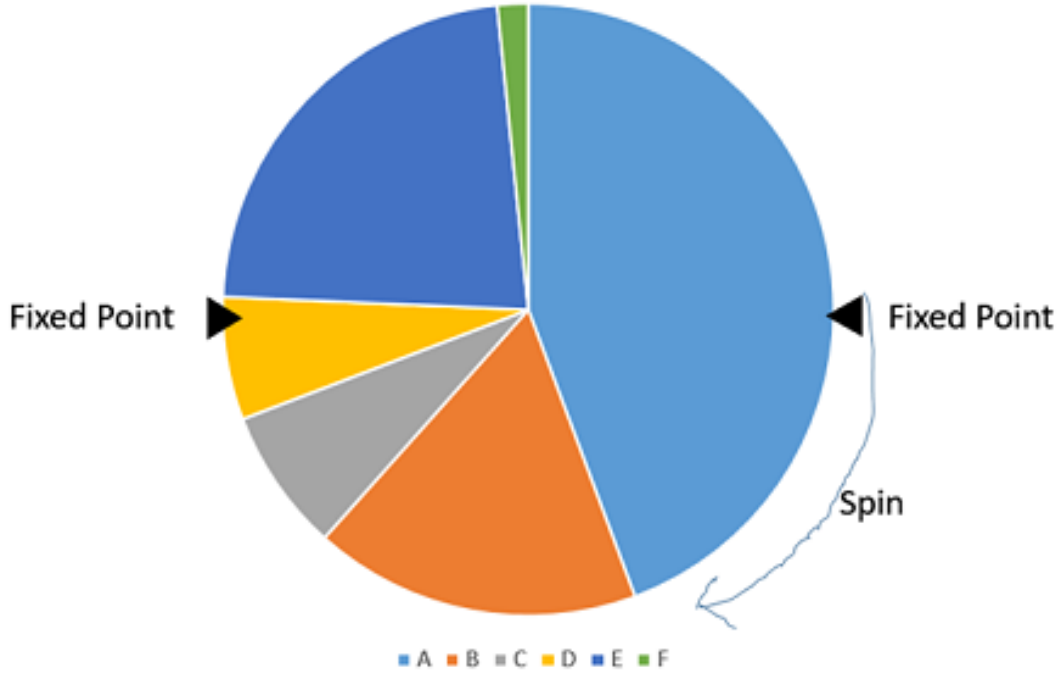


Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Stochastic Universal Sampling (SUS)

- Quite similar to Roulette wheel selection, however we have multiple fixed points.

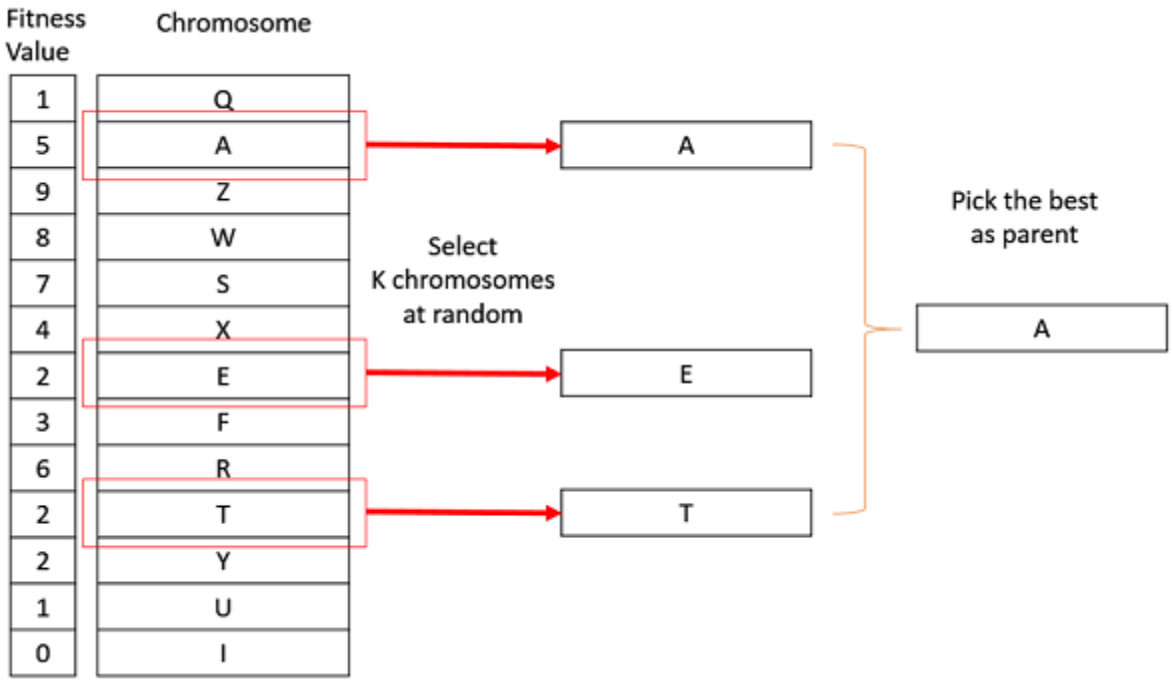
- Therefore, all the parents are chosen in just one spin of the wheel.



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Tournament Selection

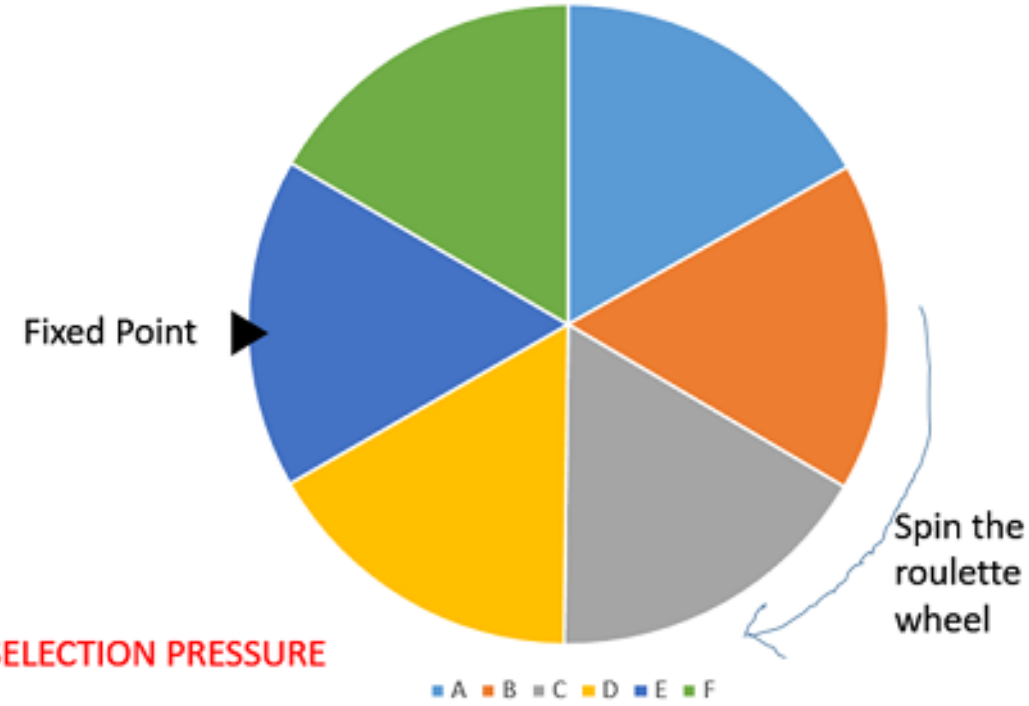
- Randomly select K individuals from the population ➔ select the best out of these to become a parent.
- Similar for selecting the next parent.



Rank Selection

- Also works with negative fitness values

- mostly used when the individuals in the population have very close fitness values



Chromosome	Fitness Value	
A	8.1	1
B	8.0	4
C	8.05	2
D	7.95	6
E	8.02	3
F	7.99	5

NO SELECTION PRESSURE

Random Selection

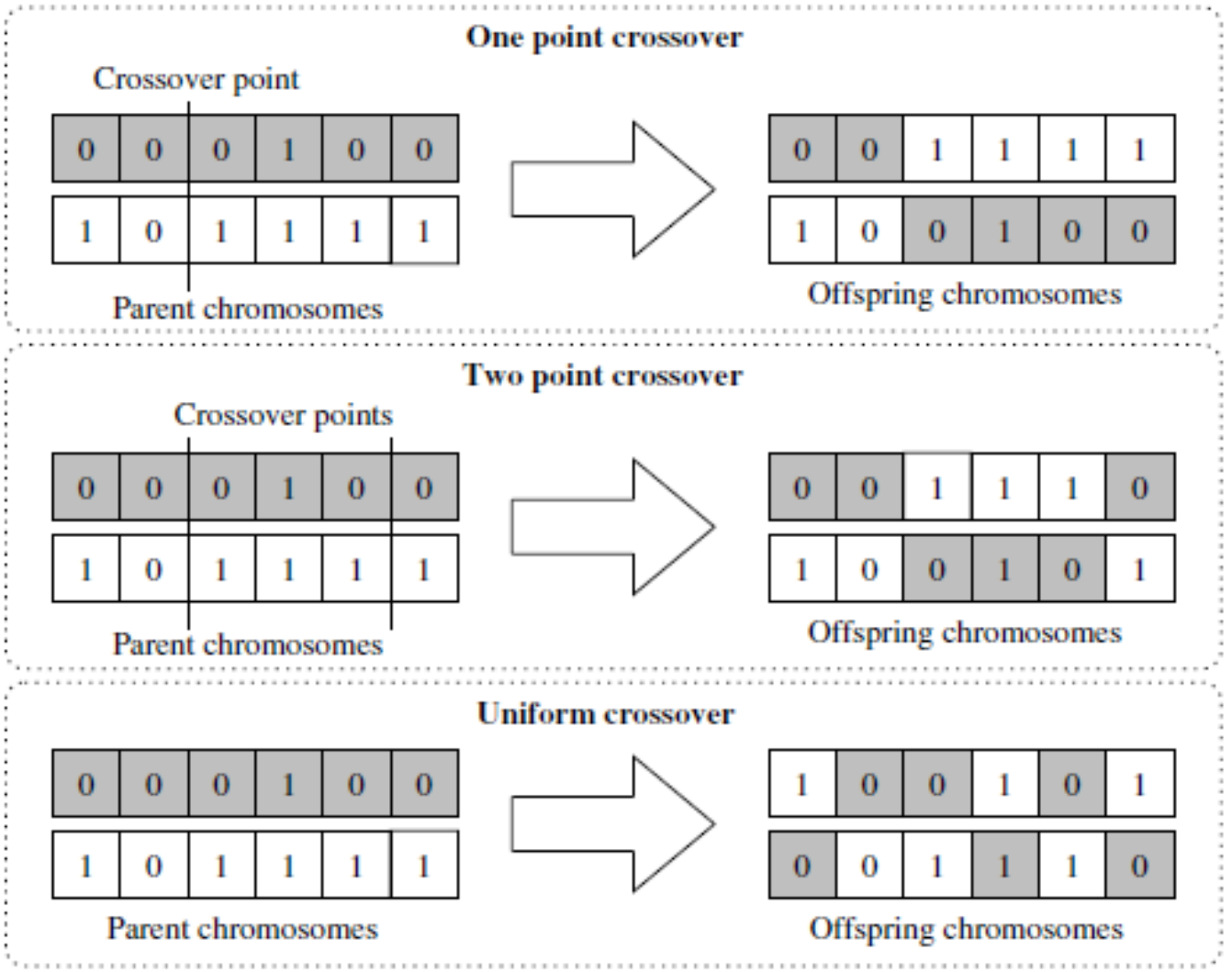
- Randomly select parents from the existing population.
- No selection pressure towards fitter individuals and therefore this strategy is usually avoided.

Reproduction

- Ways to choose crossover point(s) for reproduction:
  - **Single-point**: choose some "optimal" point in the state description, take the first half of one parent, the second half of the other.
  - **Random**: choose the split point randomly (or proportional to the parents' fitness scores)
  - **n-point**: make not 1 split point, but n different ones
  - **Uniform**: choose each element of the state description independently, at random (or proportional to fitness)

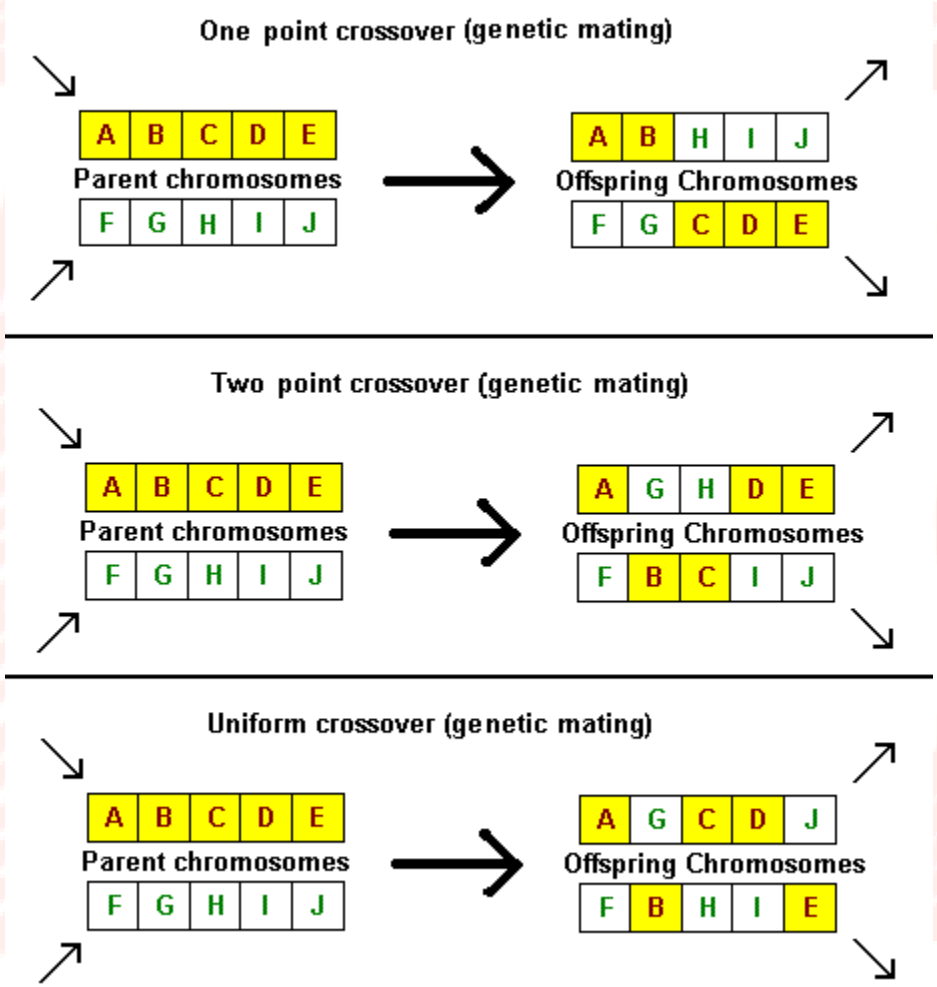
Reproduction (cont.)

- Crossover example (binary encoding):



Reproduction (cont.)

- Crossover example (permutation encoding):



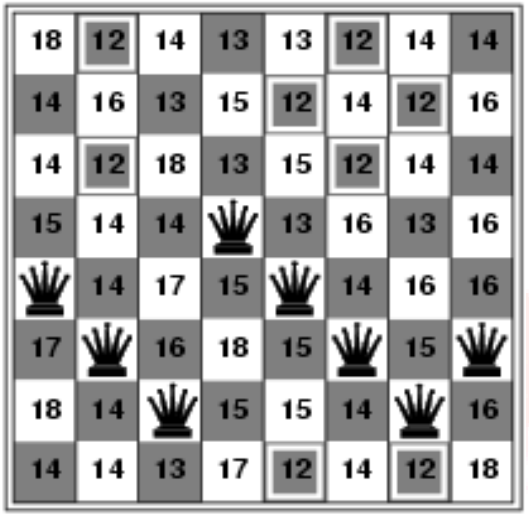
Pseudocode

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

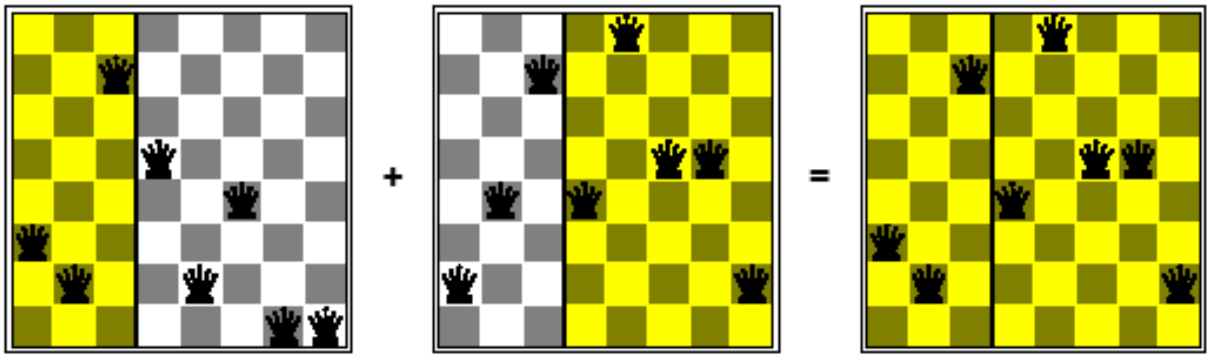
```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

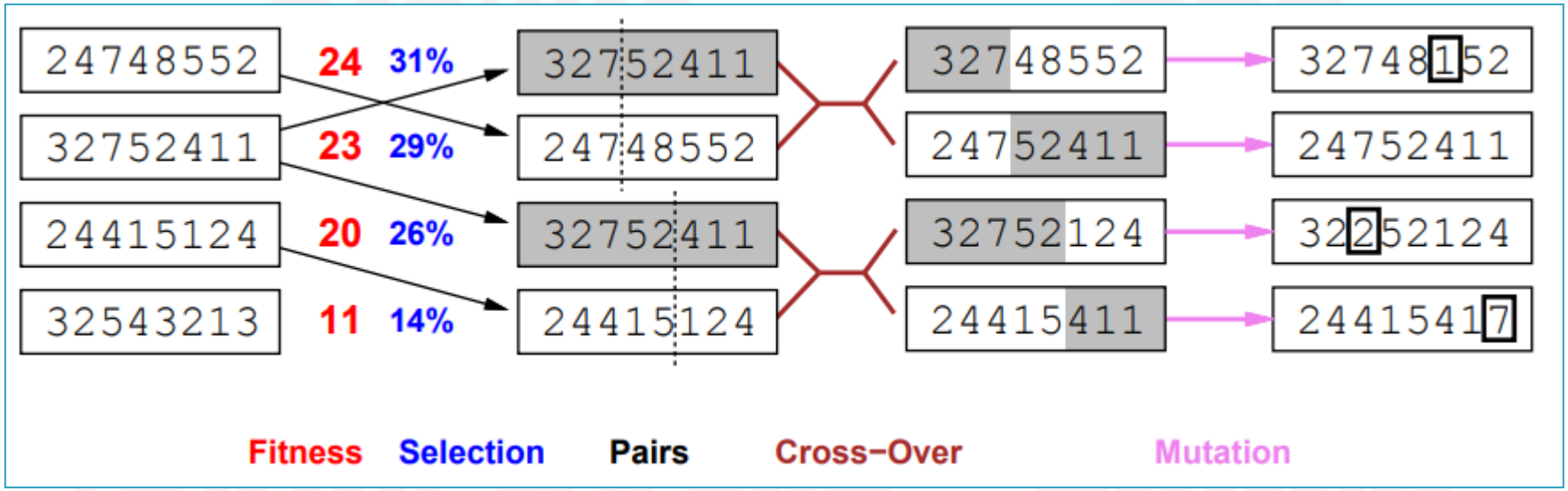
GA: 8 Queens problem

- Each state (individual) specify the positions of 8 queens, each in a column of 8 squares, each in the range from 1 to 8
- Fitness function: number of non-attacking pairs of queens (min = 0, max = $8 \times 7/2 = 28$)
- State: 4 3 2 5 4 3 2 3
- Fitness value: ???



GA: 8 Queens problem (cont.)

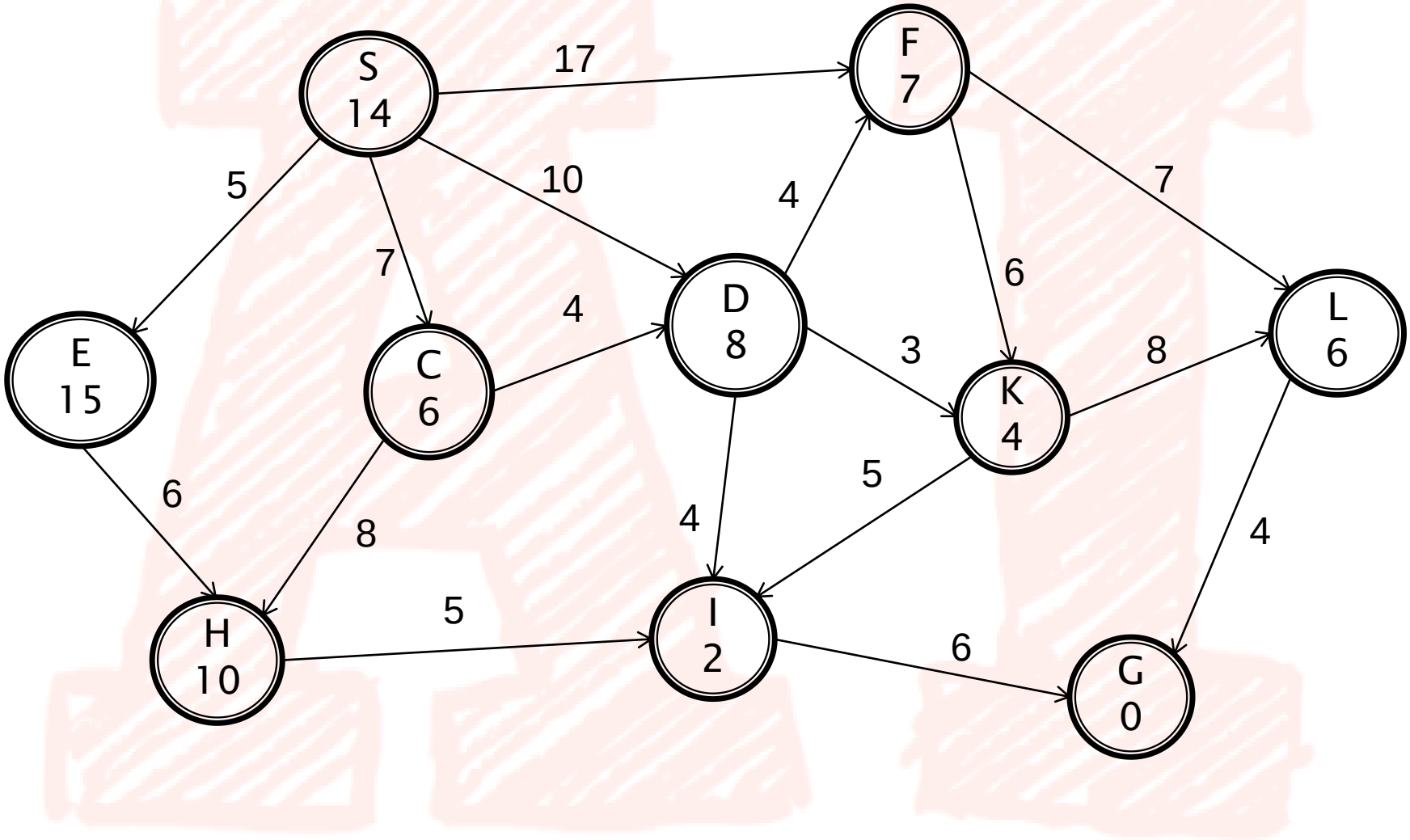




- Fitness function:
  24/(24+23+20+11) = 31%
  23/(24+23+20+11) = 29%, etc.

FACULTY OF INFORMATION TECHNOLOGY

Exercise



Exercise (cont.)

- **Find a path from S to G**
  - Depth-First Search
  - Breadth First Search
  - Uniform Cost Search
  - Greedy Best First Search
  - Hill-climbing (using heuristic)
  - A*