Advanced A* Improvements

Iterative Deepening A* (IDA*)

- Idea: Reduce memory requirement of A* by applying cutoff on values of f
- Consistent heuristic function h
- Algorithm IDA*:
  1. Initialize cutoff to f(initial-node)
  2. Repeat:
     a. Perform depth-first search by expanding all nodes N such that f(N) ≤ cutoff
     b. Reset cutoff to smallest value f of non-expanded (leaf) nodes

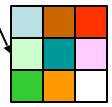
8-Puzzle

f(N) = g(N) + h(N)

with h(N) = number of misplaced tiles

4

Cutoff=4

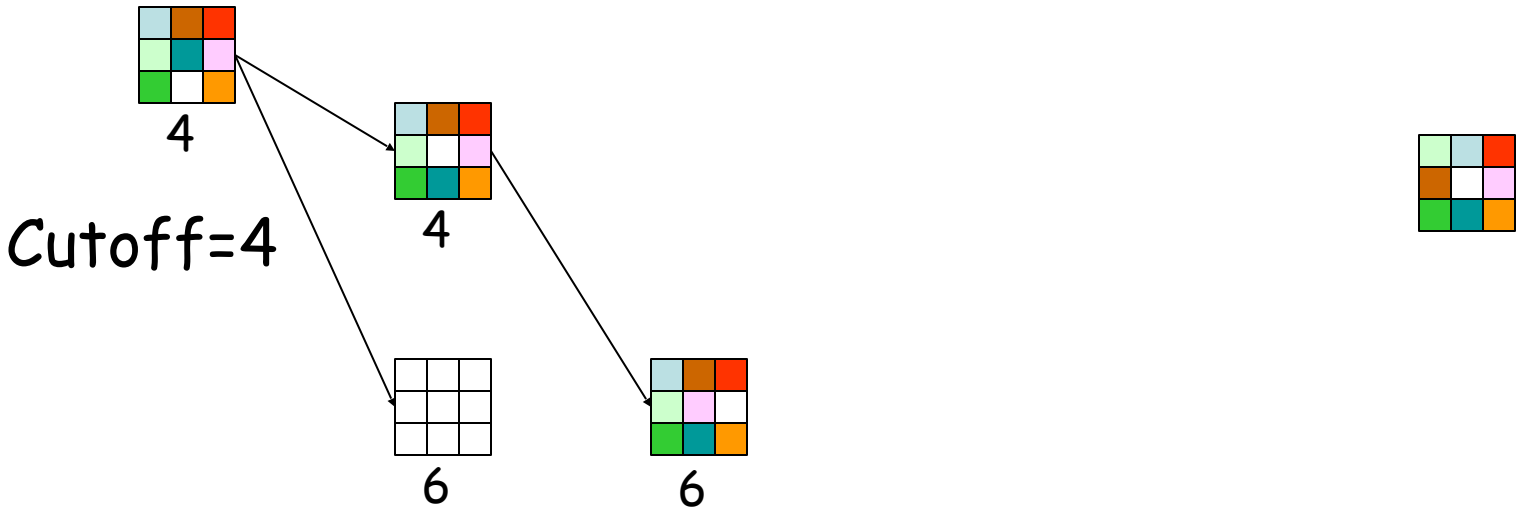


6

8-Puzzle

f(N) = g(N) + h(N)

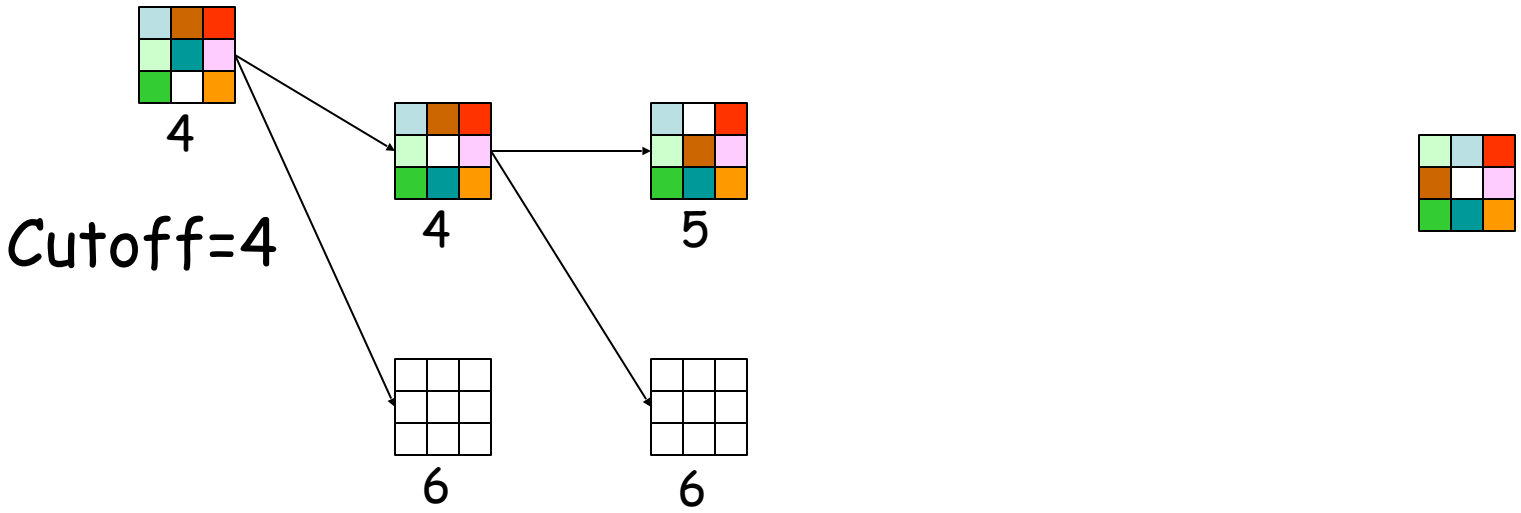
with h(N) = number of misplaced tiles



8-Puzzle

f(N) = g(N) + h(N)

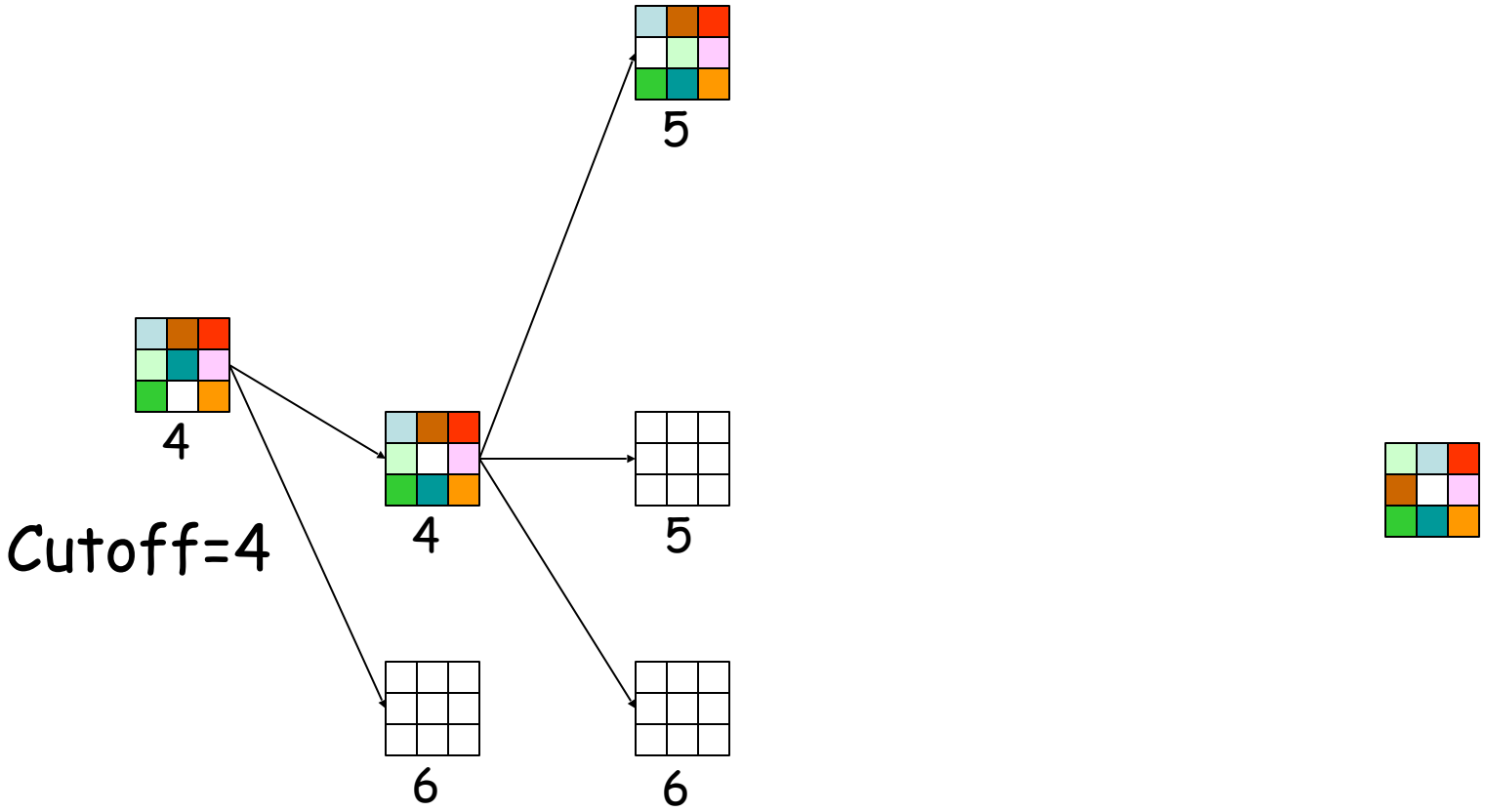
with h(N) = number of misplaced tiles



8-Puzzle

f(N) = g(N) + h(N)

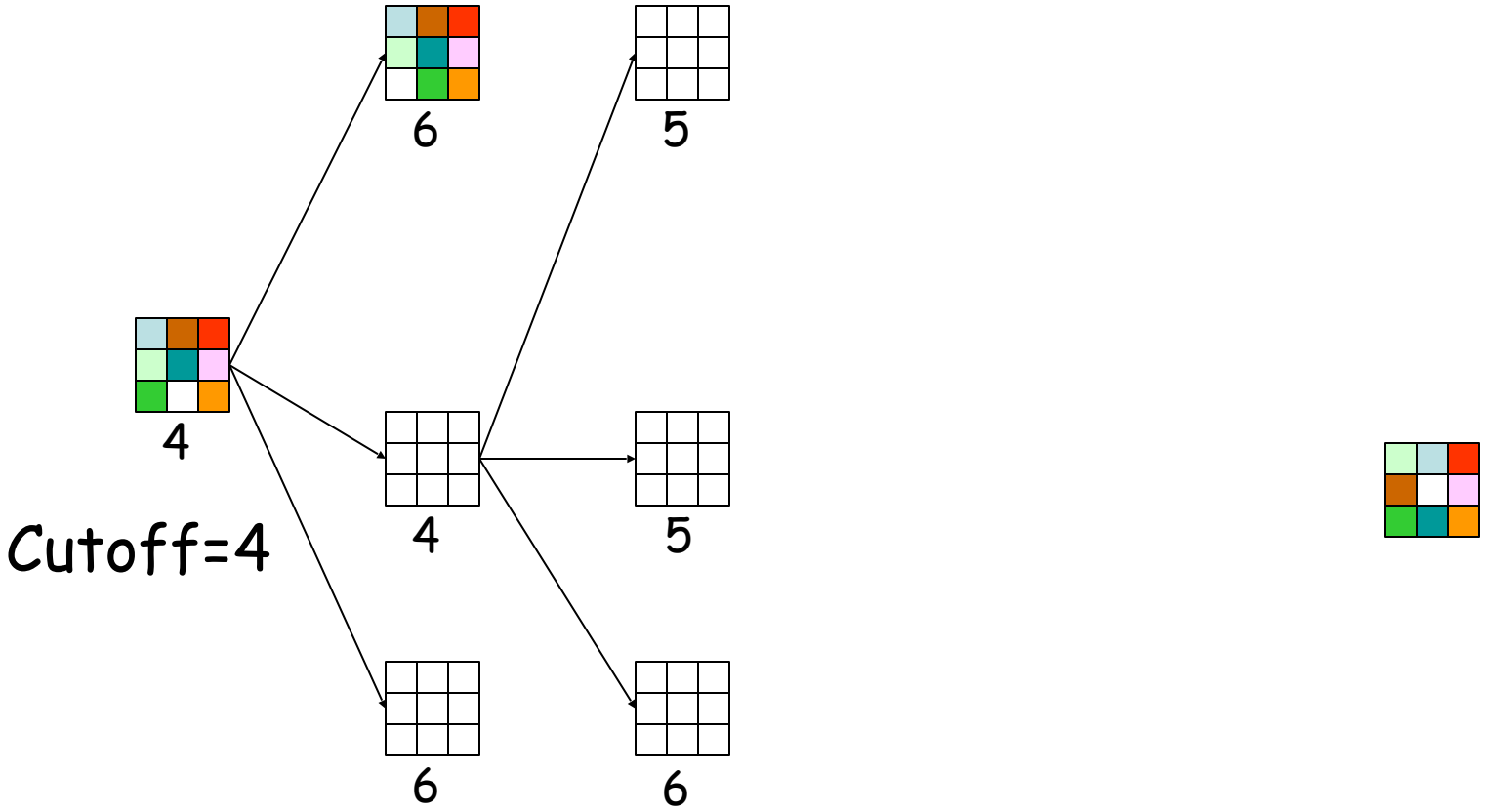
with h(N) = number of misplaced tiles



8-Puzzle

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

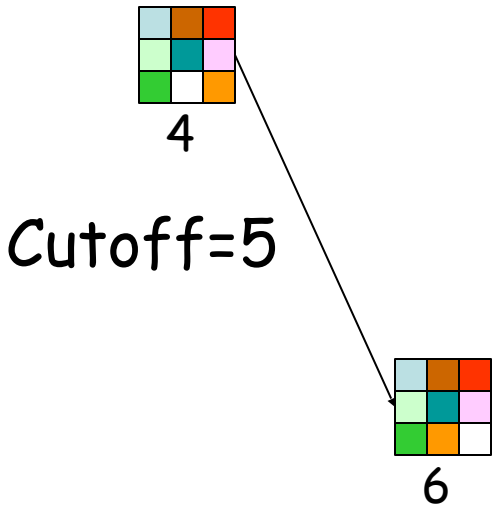
with $h(N)$ = number of misplaced tiles



8-Puzzle

f(N) = g(N) + h(N)

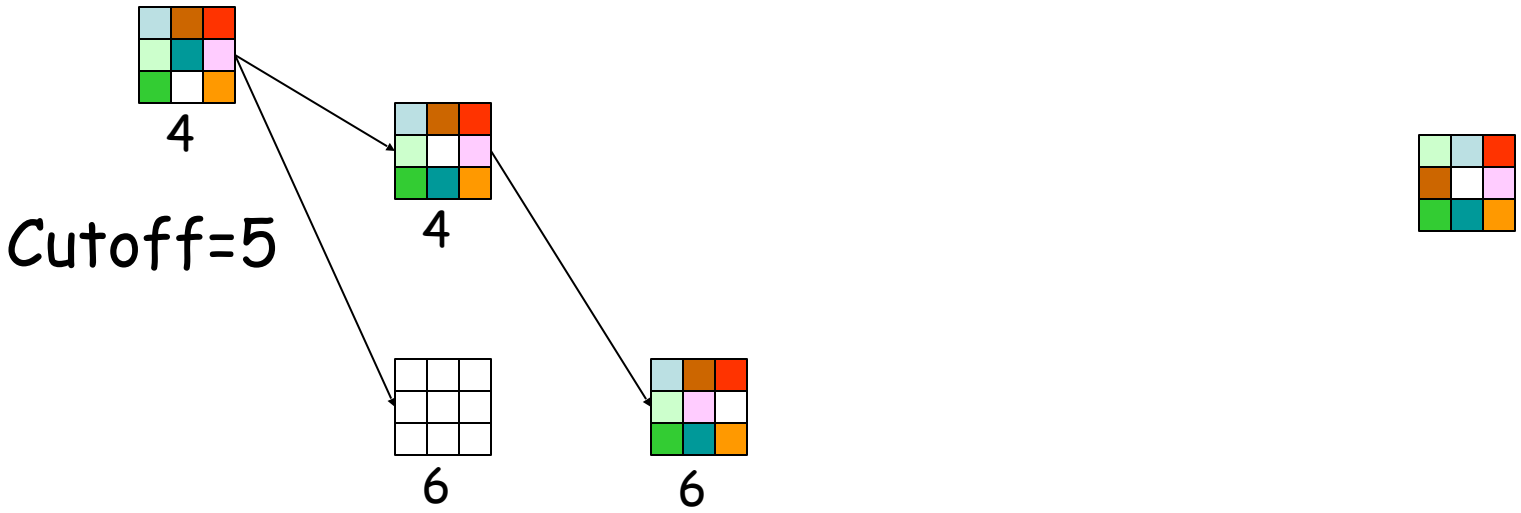
with h(N) = number of misplaced tiles



8-Puzzle

f(N) = g(N) + h(N)

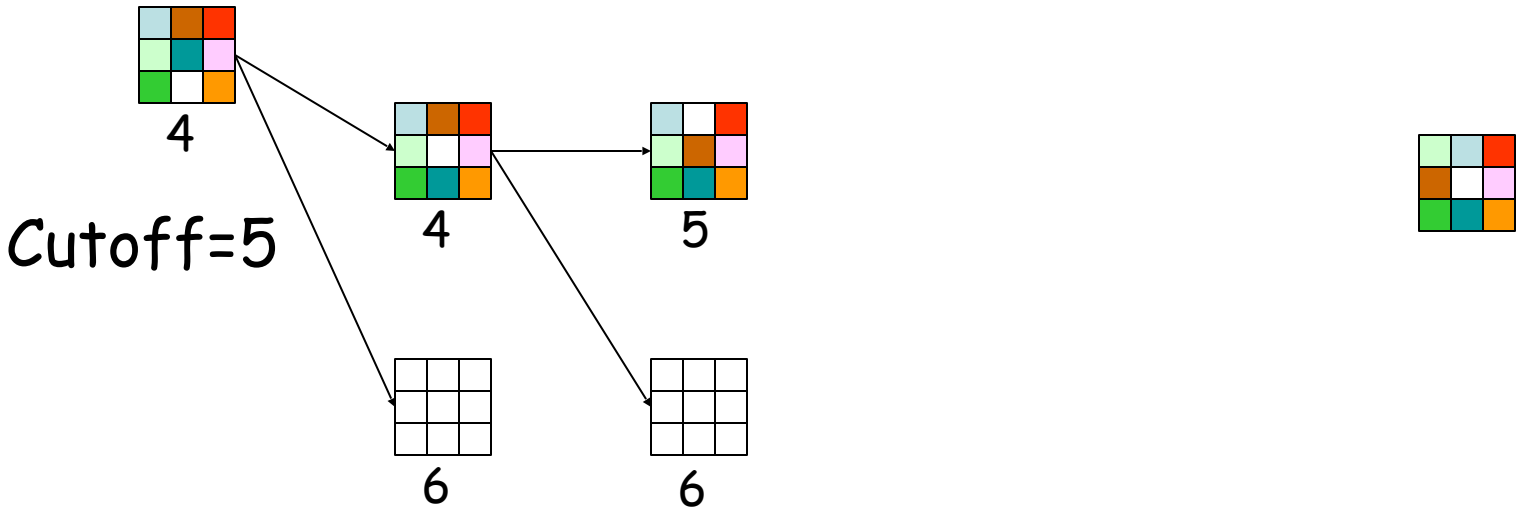
with h(N) = number of misplaced tiles



8-Puzzle

f(N) = g(N) + h(N)

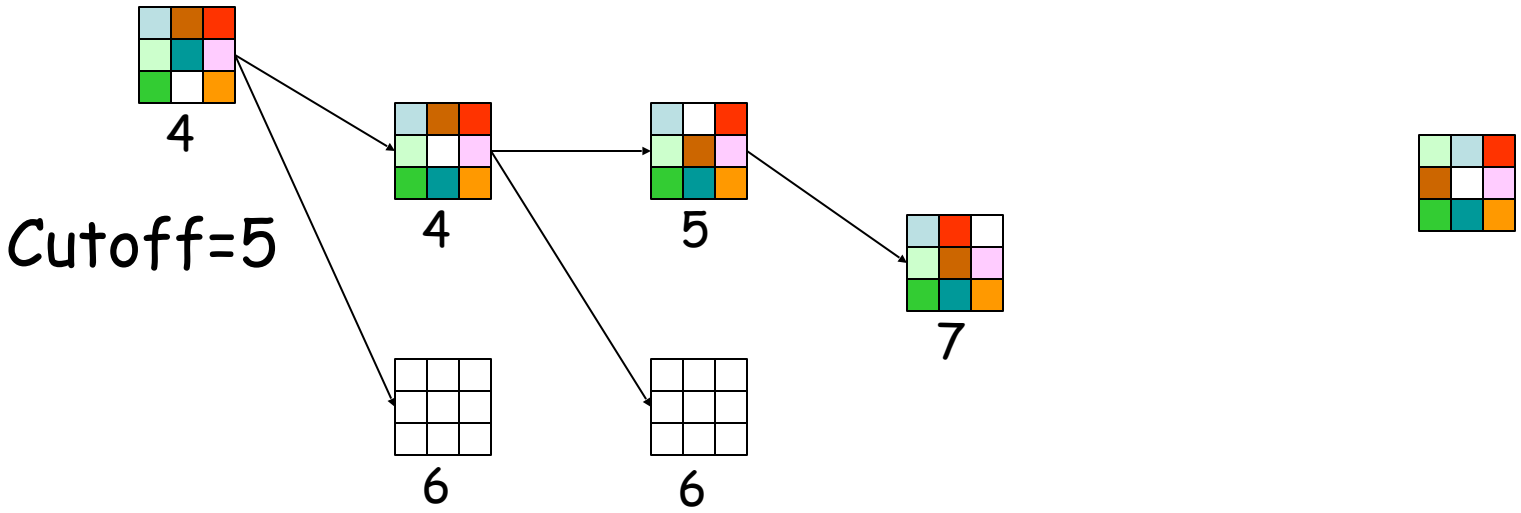
with h(N) = number of misplaced tiles



8-Puzzle

f(N) = g(N) + h(N)

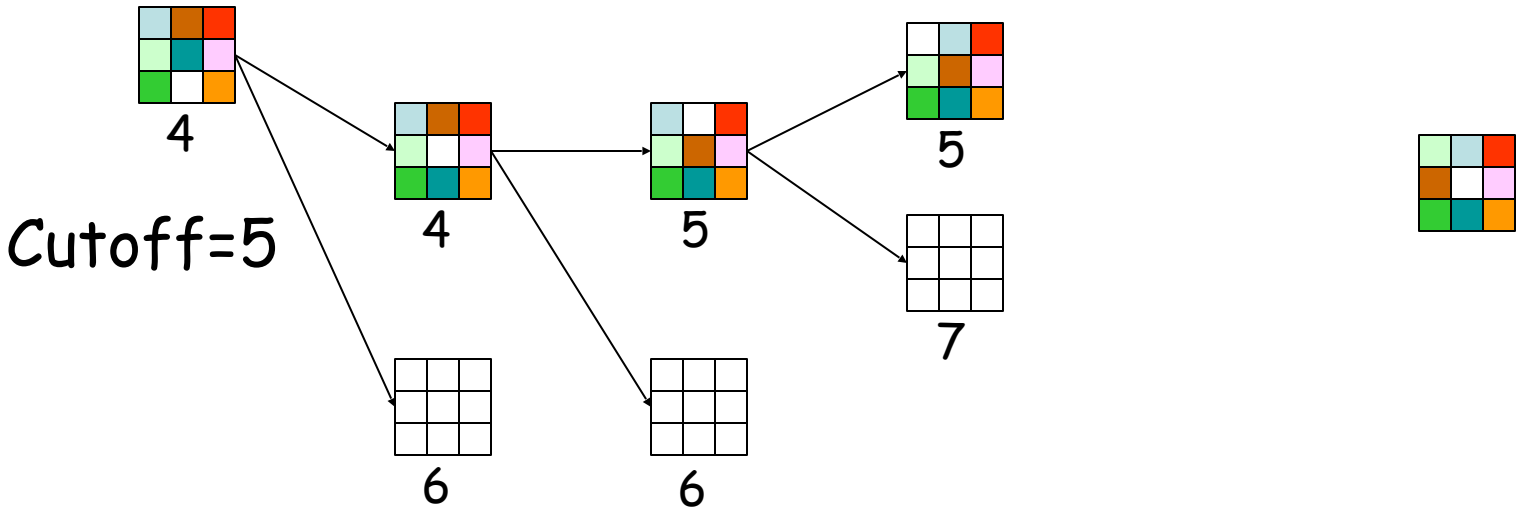
with h(N) = number of misplaced tiles



8-Puzzle

f(N) = g(N) + h(N)

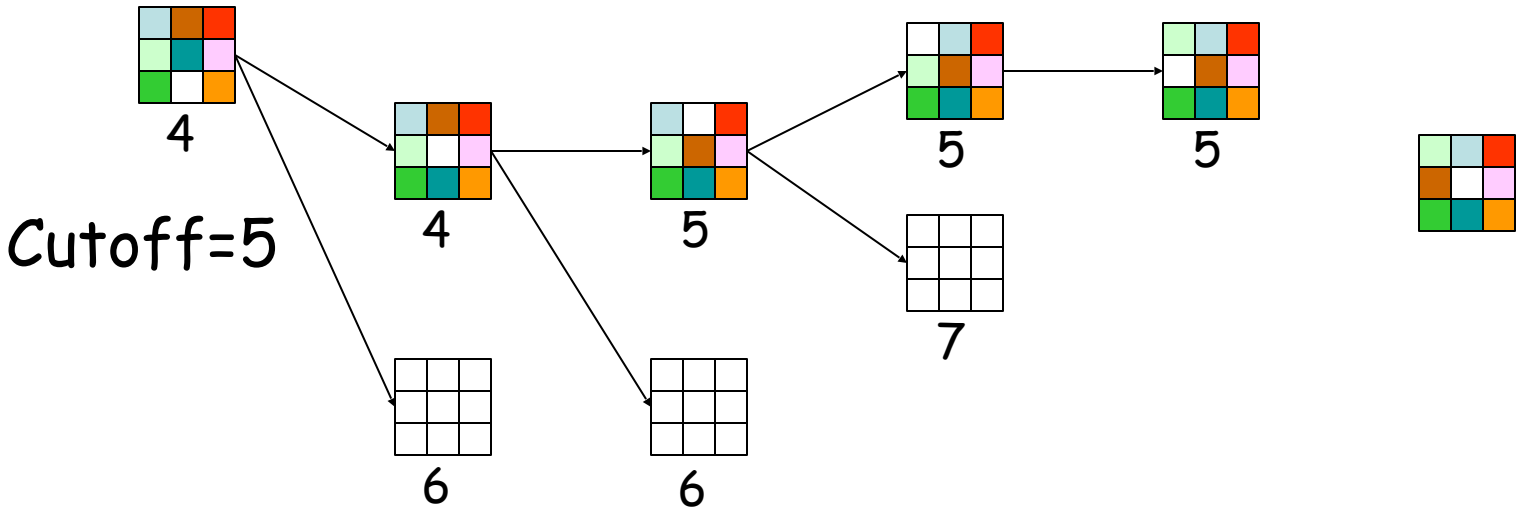
with h(N) = number of misplaced tiles



8-Puzzle

f(N) = g(N) + h(N)

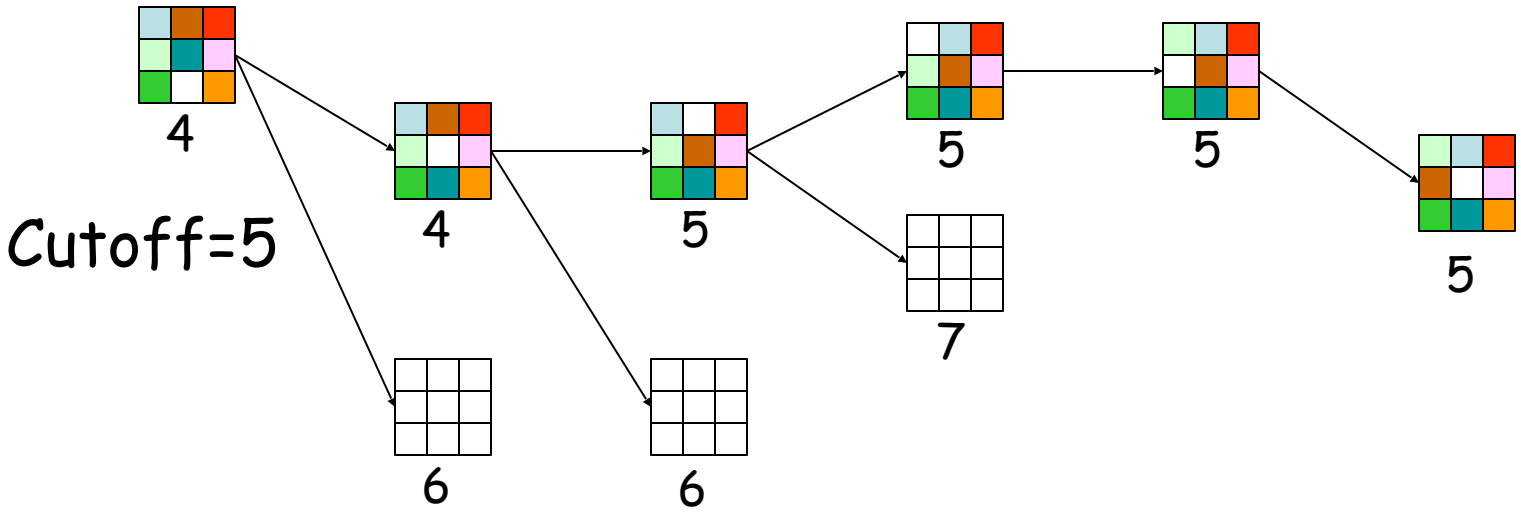
with h(N) = number of misplaced tiles



8-Puzzle

f(N) = g(N) + h(N)

with h(N) = number of misplaced tiles



Advantages/Drawbacks of IDA*

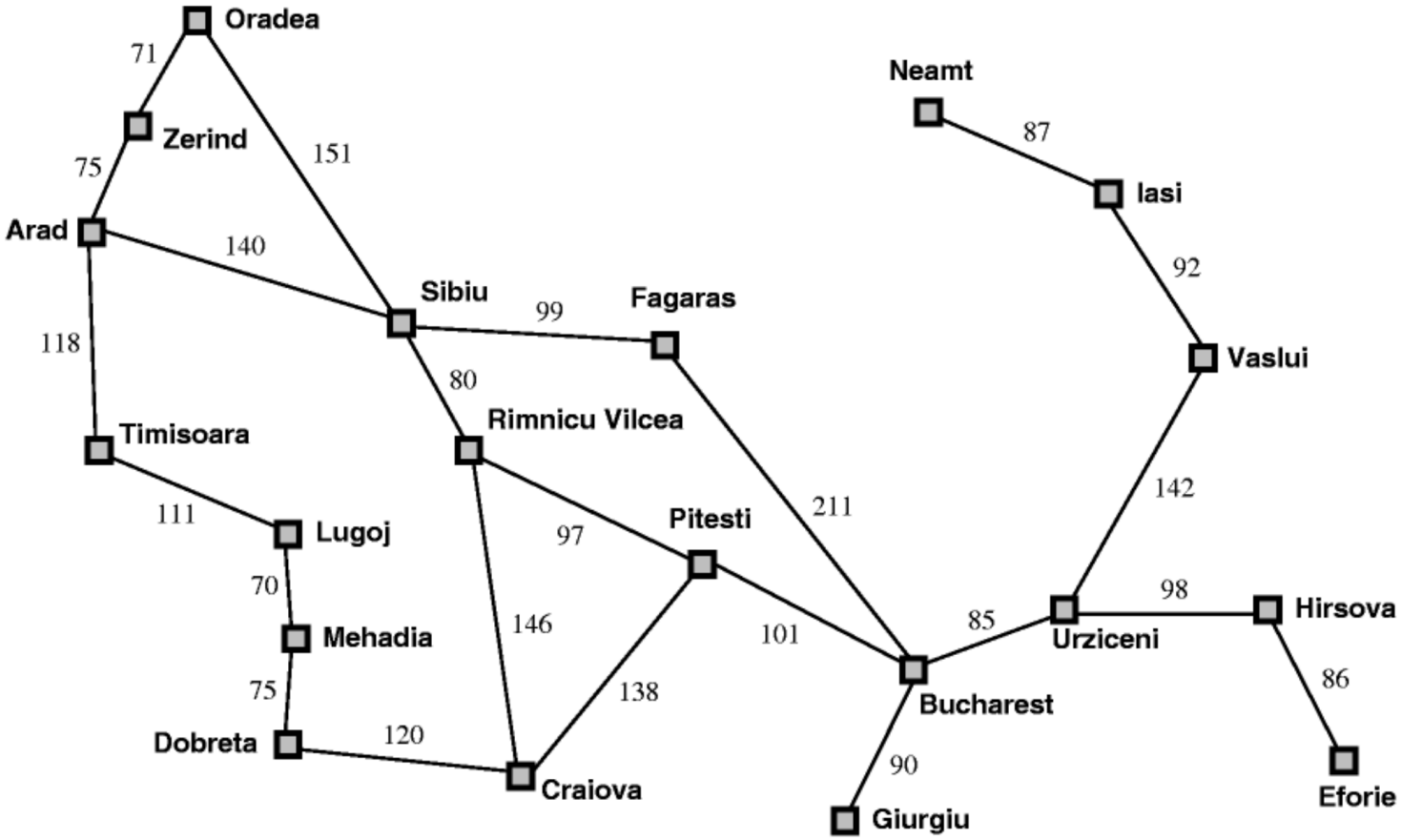
- Advantages:
  - Still complete and optimal
  - Requires less memory than A*
  - Avoid the overhead to sort the fringe
- Drawbacks:
  - Can't avoid revisiting states not on the current path
  - Available memory is poorly used

  (→ memory-bounded search, see R&N p. 101-104)

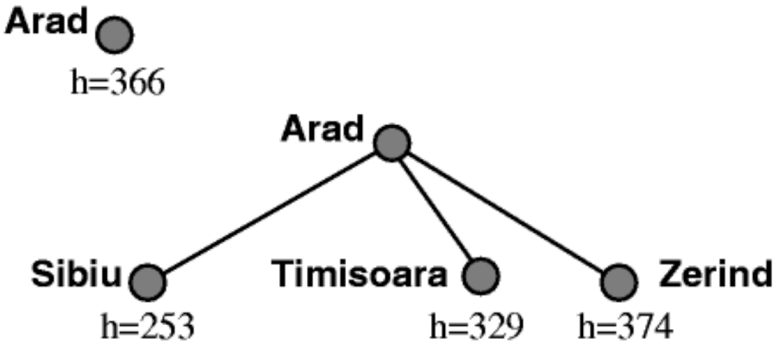
Recursive Best-First Search

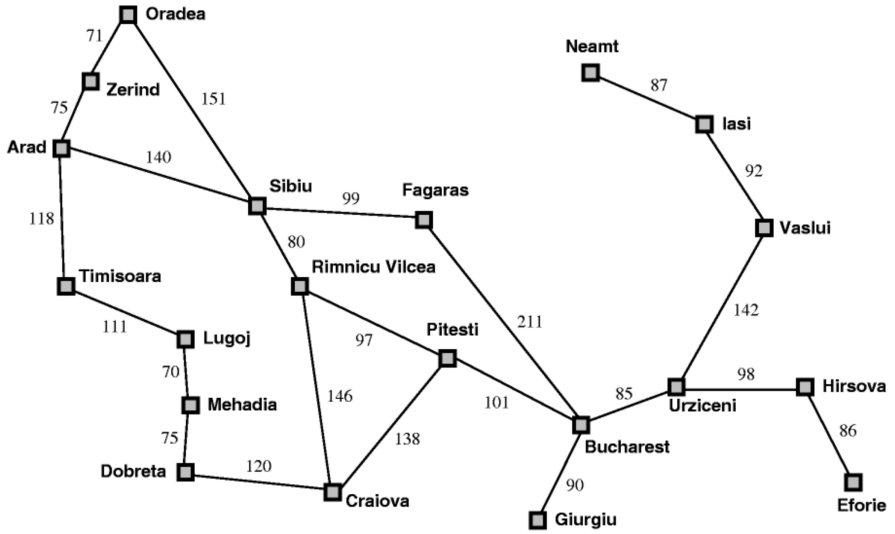
- Recursive algorithm that attempts to mimic standard best-first search with linear space.
  - Keeps track of the f-value of the best-alternative path available.
  - If current f-values exceeds this alternative f-value than backtrack to alternative path.
  - Upon backtracking change f-value to best f-value of its children.
  - Re-expansion of this result is thus still possible.

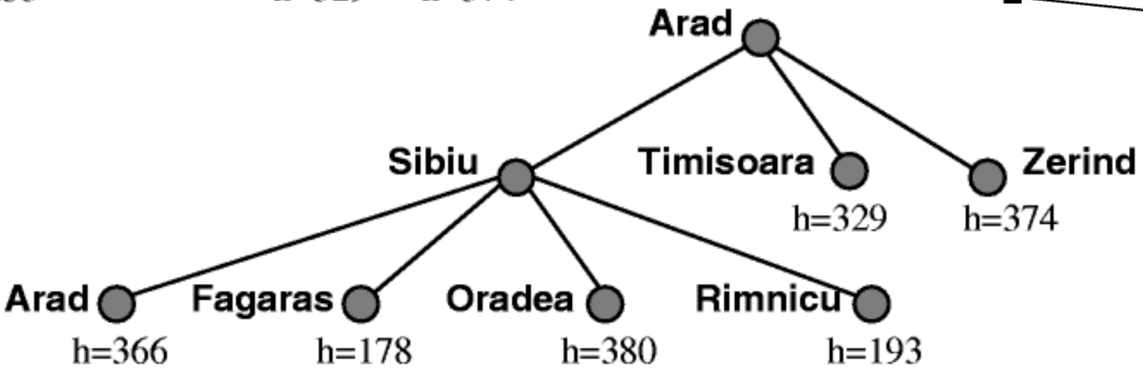
```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```

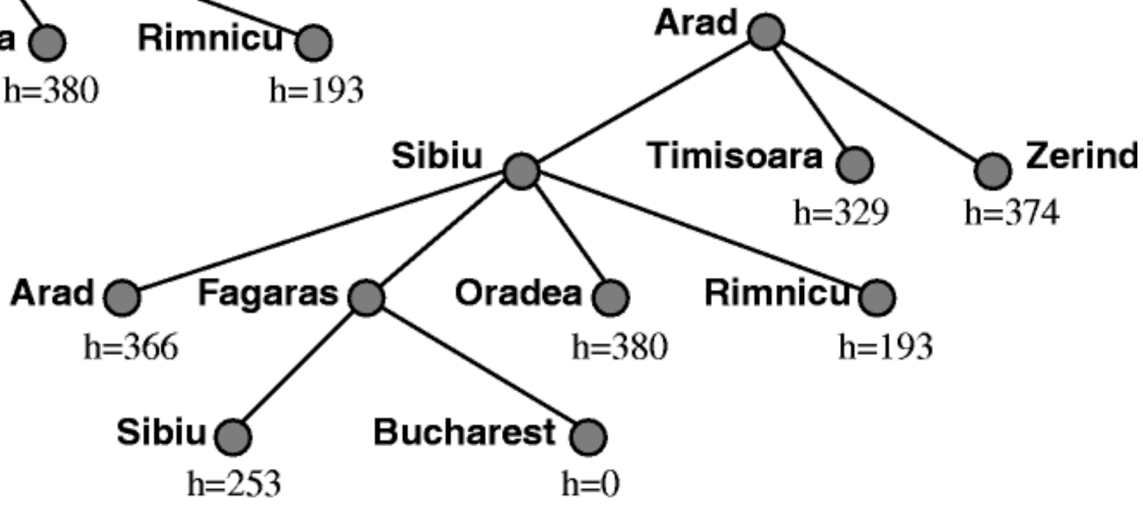


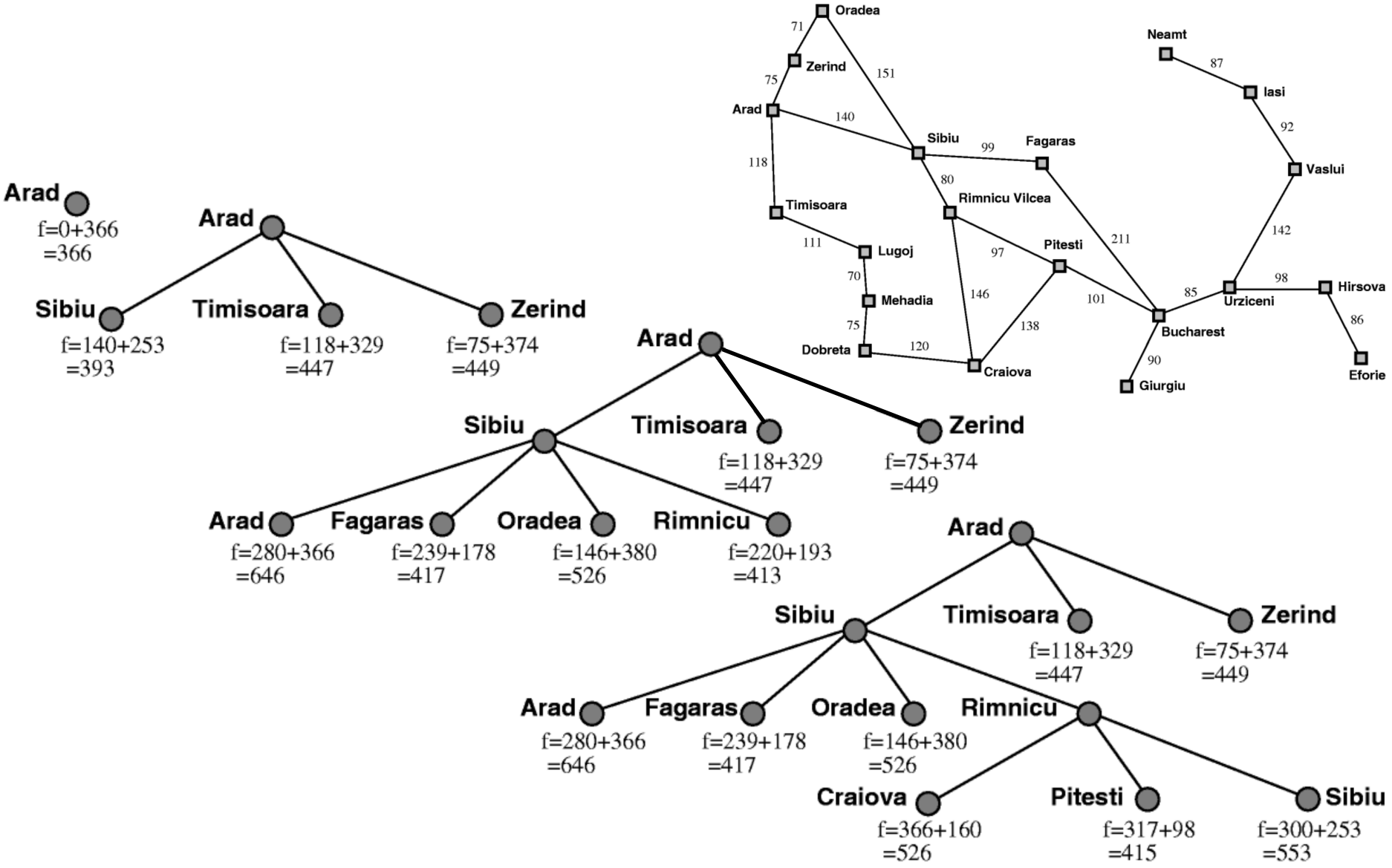
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

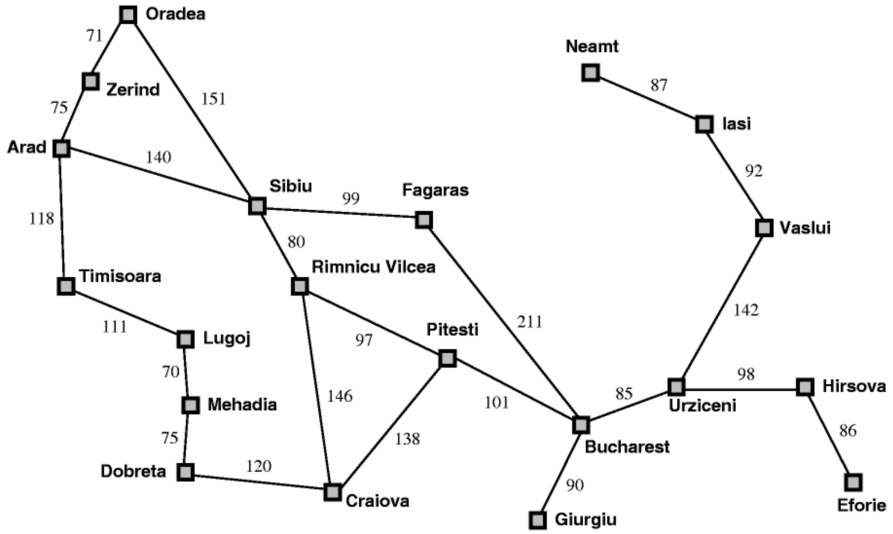

Arad
h=366
Arad
Sibiu
Timisoara
Zerind
h=253
h=329
h=374


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

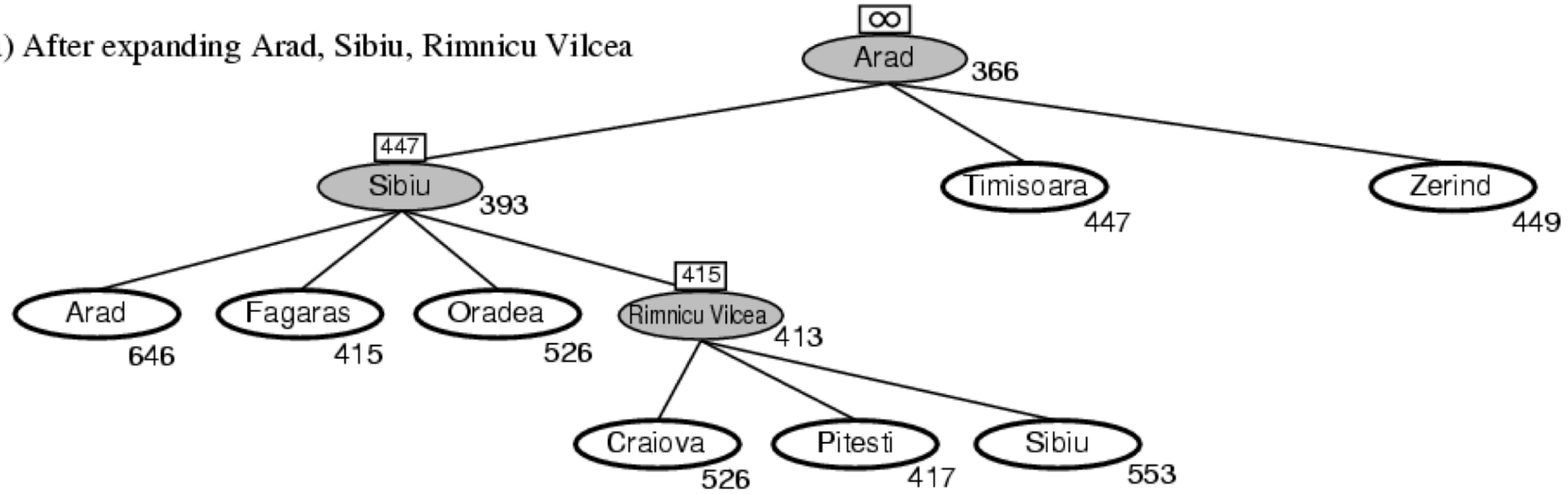

Arad
Sibiu
Timisoara
Zerind
h=329
h=374
Arad
Fagaras
Oradea
Rimnicu
h=366
h=178
h=380
h=193

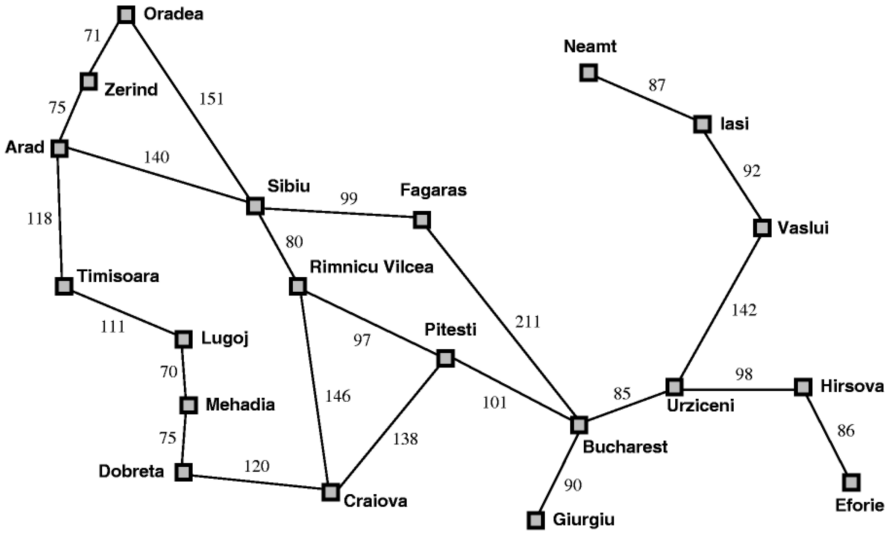

Arad
Sibiu
Timisoara
Zerind
h=329
h=374
Arad
Fagaras
Oradea
Rimnicu
h=366
h=380
h=193
Sibiu
Bucharest
h=253
h=0


Oradea
71
Neamt
87
Zerind
151
Iasi
75
Arad
140
Sibiu
99
Fagaras
92
118
Vaslui
80
Rimnicu Vilcea
Timisoara
142
111
Lugoj
97
Pitesti
211
70
98
Hirsova
Mehadia
146
85
Urziceni
101
75
86
Bucharest
138
Dobreta
120
Craiova
90
Giurgiu
Eforie
Arad
f=0+366
=366
Arad
Sibiu
Timisoara
Zerind
f=140+253
=393
f=118+329
=447
f=75+374
=449
Arad
Sibiu
Timisoara
Zerind
f=118+329
=447
f=75+374
=449
Arad
Fagaras
Oradea
Rimnicu
f=280+366
=646
f=239+178
=417
f=146+380
=526
f=220+193
=413
Arad
Sibiu
Timisoara
Zerind
f=118+329
=447
f=75+374
=449
Arad
Fagaras
Oradea
Rimnicu
f=280+366
=646
f=239+178
=417
f=146+380
=526
Craiova
Pitesti
Sibiu
f=366+160
=526
f=317+98
=415
f=300+253
=553

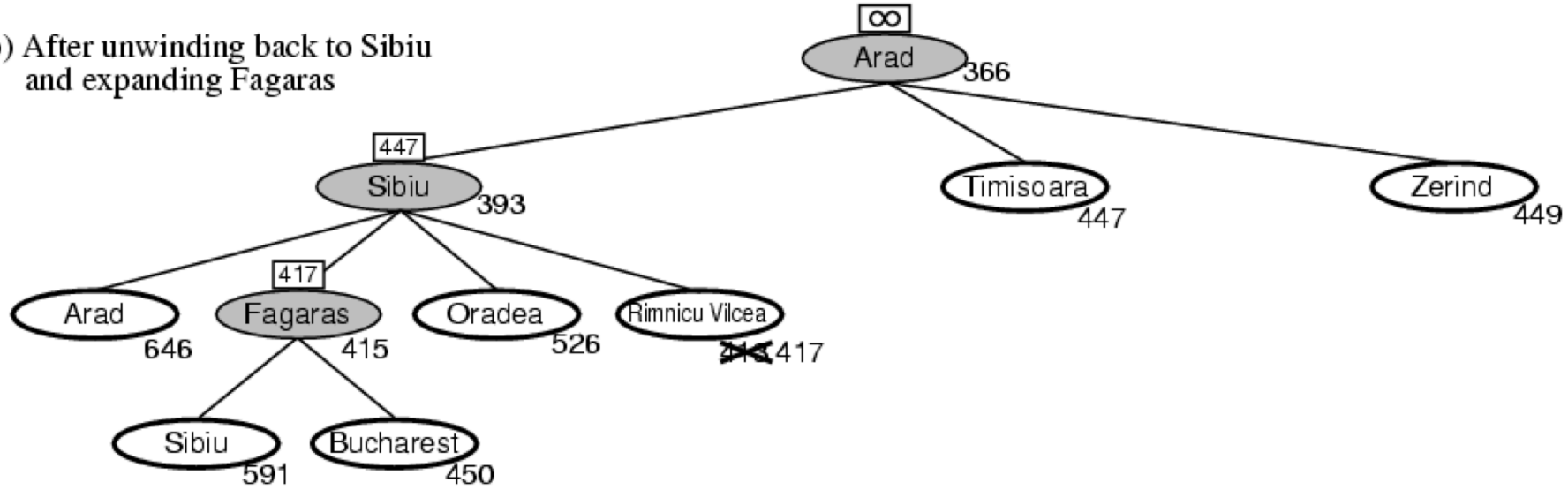


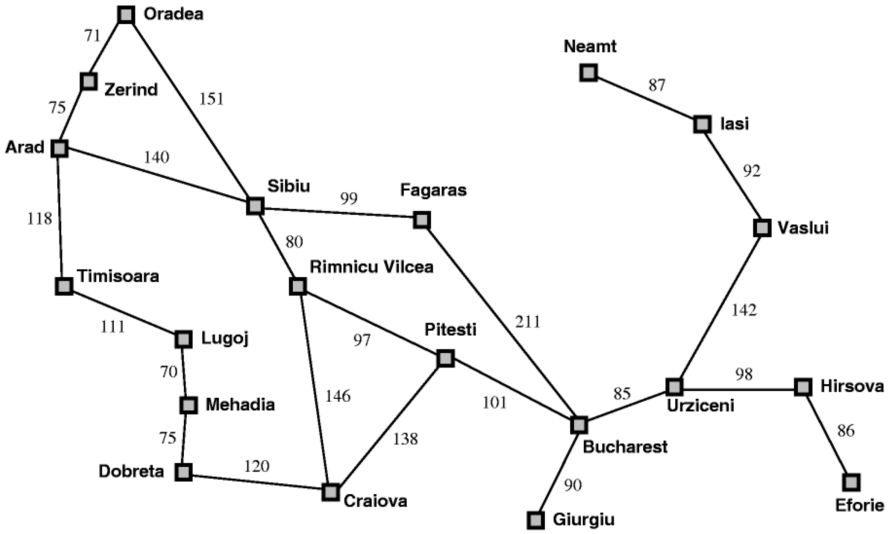
(a) After expanding Arad, Sibiu, Rimnicu Vilcea



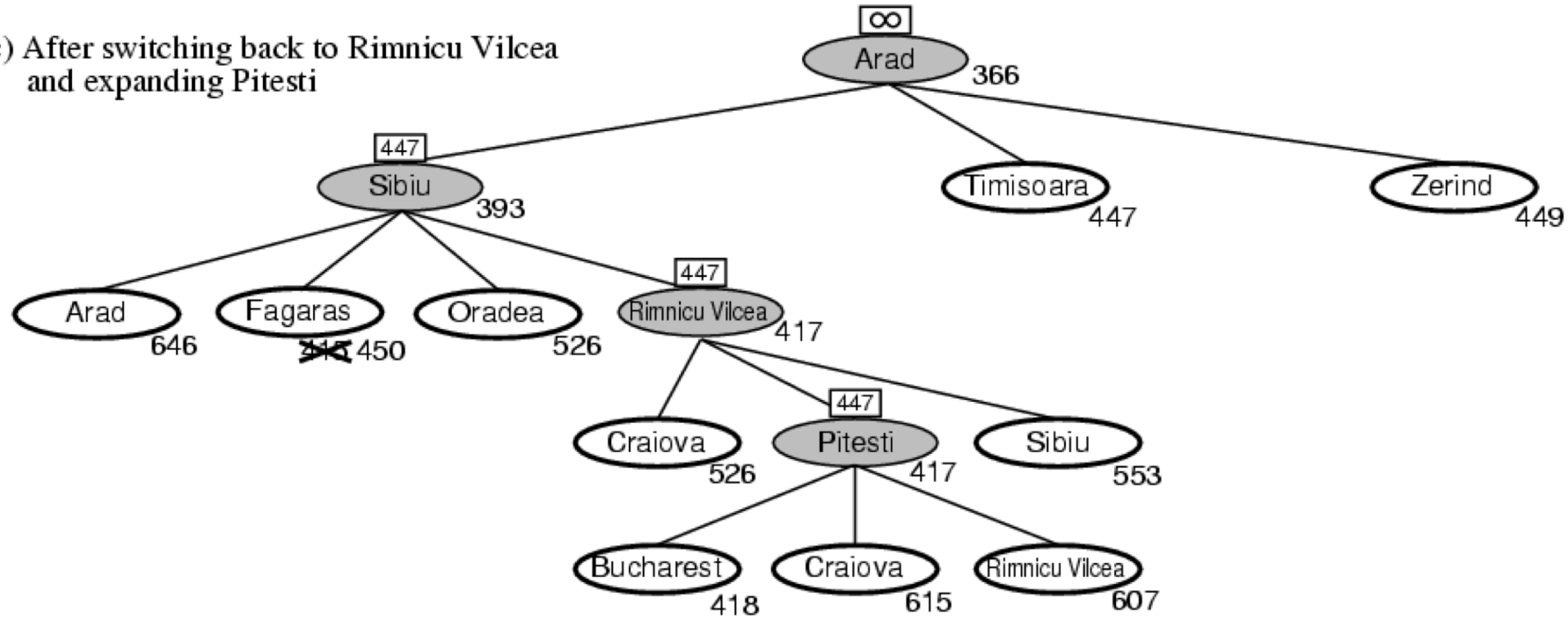


(b) After unwinding back to Sibiu and expanding Fagaras





(c) After switching back to Rimnicu Vilcea and expanding Pitesti

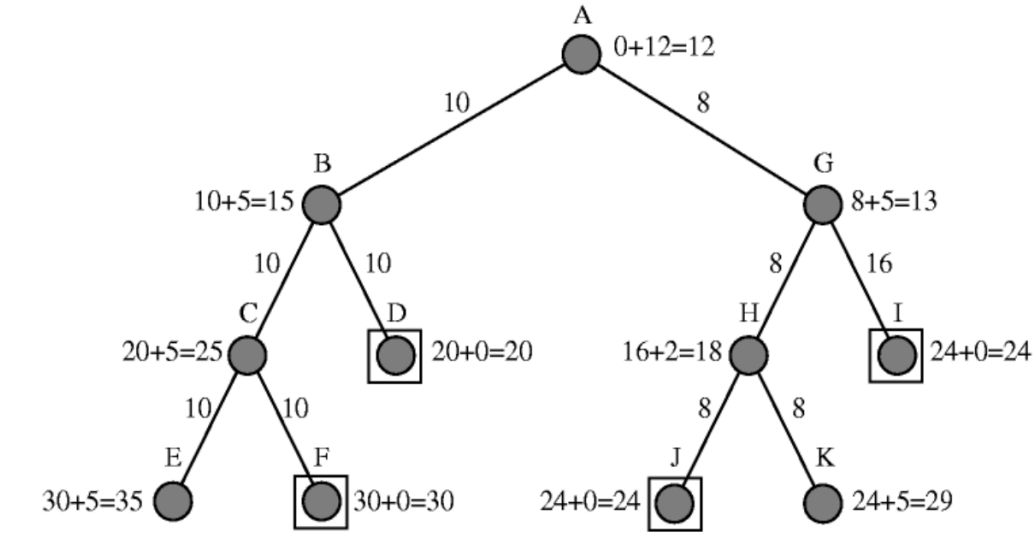


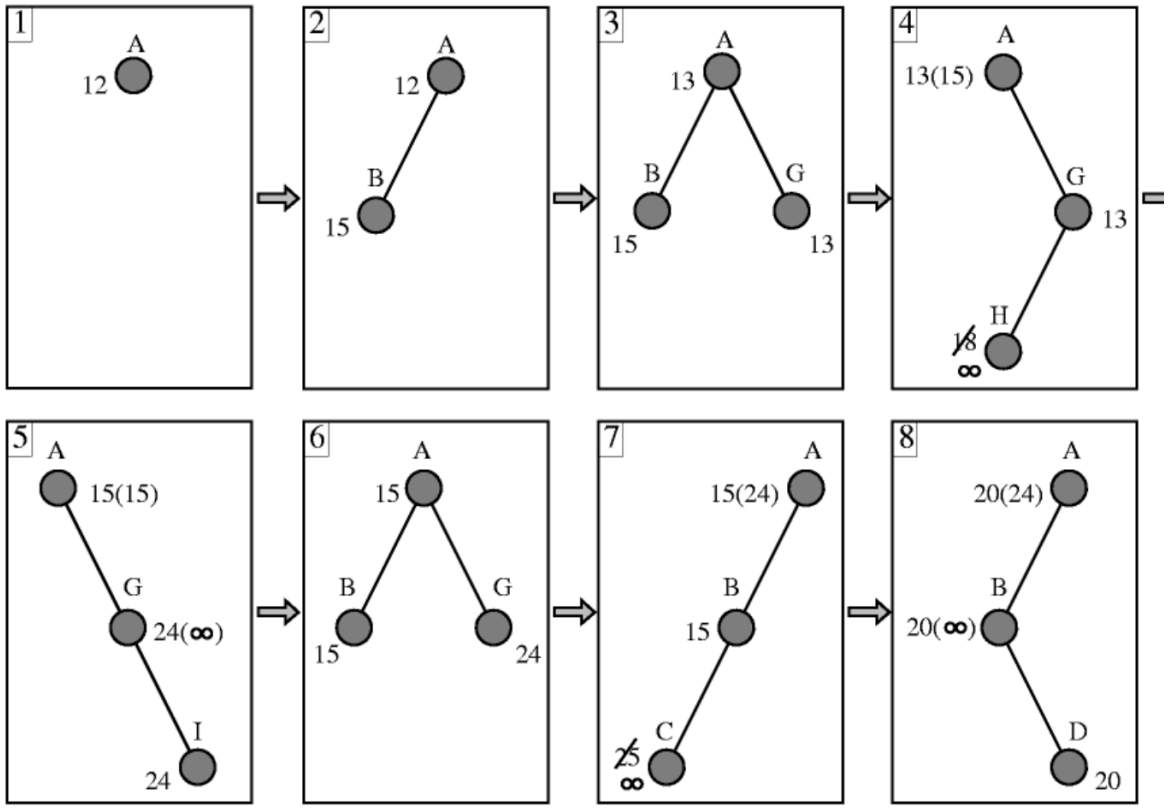
(simplified) Memory-Bounded A*

- 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

- What if all leafs 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.






```
function SMA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: Queue, a queue of nodes ordered by f-cost

  Queue ← MAKE-QUEUE({MAKE-NODE(INITIAL-STATE[problem])})
  loop do
      if Queue is empty then return failure
      n ← deepest least-f-cost node in Queue
      if GOAL-TEST(n) then return success
      s ← NEXT-SUCCESSOR(n)
      if s is not a goal and is at maximum depth then
          f(s) ← ∞
      else
          f(s) ← MAX(f(n), g(s)+h(s))
      if all of n's successors have been generated then
          update n's f-cost and those of its ancestors if necessary
      if SUCCESSORS(n) all in memory then remove n from Queue
      if memory is full then
          delete shallowest, highest-f-cost node in Queue
          remove it from its parent's successor list
          insert its parent on Queue if necessary
      insert s on Queue
  end
```

Further A* Variants

- Anytime A*
- Dynamic Weighting Anytime A*
- Dynamic A*
- Theta*
- Accelerated A*

Anytime A*

Three changes make A* an anytime algorithm:

1) Use a non-admissible heuristic so that sub-optimal solutions are found quickly.
2) Continue the search after the first solution is found using it to prune the open list
3) When the open list is empty, the best solution generated is optimal.

How to choose a non-admissible heuristic?

Anytime A*

How to choose a non-admissible heuristic?

Weighted evaluation functions:

$f'(N) = (1-w)*g(N) + w*h(N)$

Higher weight on h(n) tends to search deeper.

- Admissible if h(n) is admissible and $w \leq 0.5$
- Otherwise, the search is non-admissible, but it normally finds solutions much faster.

An appropriate w makes possible a tradeoff between the solution quality and the computation time.

Dynamic Weighting Anytime A*

With dynamic weighting, you assume that at the beginning of your search, it's more important to get (anywhere) quickly; at the end of the search, it's more important to get to the goal.

There is a weight (w >= 1) associated with the heuristic. As you get closer to the goal, you decrease the weight; this decreases the importance of the heuristic, and increases the relative importance of the actual cost of the path.

Dynamic A*

- There are variants of A* that allow for changes to the world after the initial path is computed. D* is intended for use when you don't have complete information. If you don't have all the information, A* can make mistakes; D*'s contribution is that it can correct those mistakes without taking much time.
- However, D* require a lot of space to keep around its internal information (OPEN/CLOSED sets, path tree, gvalues), and then when the map changes, D* will tell you if you need to adjust your path to take into account the map changes.
- For a game with lots of moving units, you usually don't want to keep all that information around, so D* isn't applicable. D* is designed for robotics, where there is only one robot—you don't need to reuse the memory for some other robot's path.

Theta*

- Sometimes grids are used for pathfinding because the map is made on a grid, not because you actually want movement on a grid. A* would run faster and produce better paths if given a graph of key points (such as corners) instead of the grid. However if you don't want to precompute the graph of corners, you can use Theta*, a variant of A* that runs on square grids, to find paths that don't strictly follow the grid. When building parent pointers, Theta* will point directly to an ancestor if there's a line of sight to that node, and skips the nodes in between.

Accelerated A*

- Trajectory planning in the 4D space (3D+time)
  - represented by OctanTrees
  - dynamic search step, function of the proximity to an obstacle or a waypoint

Configuration	Shortest Paths A*	Θ*	AA*	RRT PS	dynamic bi-RRT PS
a wall	357.106 (2 249.5)	357.125 (0.504)	357.106 (0.881)	490.124 (0.0004)	525.953 (0.0001)
a half circle	422.154 (2 827.2)	424.876 (0.652)	422.154 (0.264)	685.465 (0.0021)	695.129 (0.0001)
a single gap	395.991 (3 985.4)	399.072 (0.736)	395.991 (0.207)	505.632 (0.1962)	737.162 (0.0006)
a double gap	485.213 (6 131.4)	490.056 (1.744)	485.213 (0.735)	581.479 (0.2949)	614.254 (0.2173)
a maze	4 121.478 (10 989.3)	4 133.491 (3.148)	4 121.478 (7.202)	4 542.958 (0.4502)	4 559.624 (0.0188)
multi-obstacles	662.550 (6 750.2)	696.697 (250.208)	662.550 (20.586)	2 971.787 (40.3726)	2 768.783 (17.7803)

Configuration	Shortest Paths A*	Θ*	AA*	RRT PS	dynamic bi-RRT PS
a wall	436 (876)	24 020 (25 216)	687 (715)	24	74
a half circle	605 (2 132)	25 756 (26 912)	811 (855)	48	131
a single gap	562 (1 871)	24 308 (25 524)	786 (836)	24 462	463
a double gap	1 092 (2 694)	44 380 (45 302)	1 940 (2 003)	25 266	28 228
a maze	3 324 (5 347)	145 284 (147 940)	13 769 (13 851)	46 272	81 843
multi-obstacles	17 634 (43 796)	166 091 (168 355)	28 149 (28 532)	781 558	638 117

Local Search

Local Search

- Light-memory search method
- No search tree; only the current state is represented!
- Only applicable to problems where the path is irrelevant (e.g., 8-queen), unless the path is encoded in the state
- Many similarities with optimisation techniques

Gradient Descent

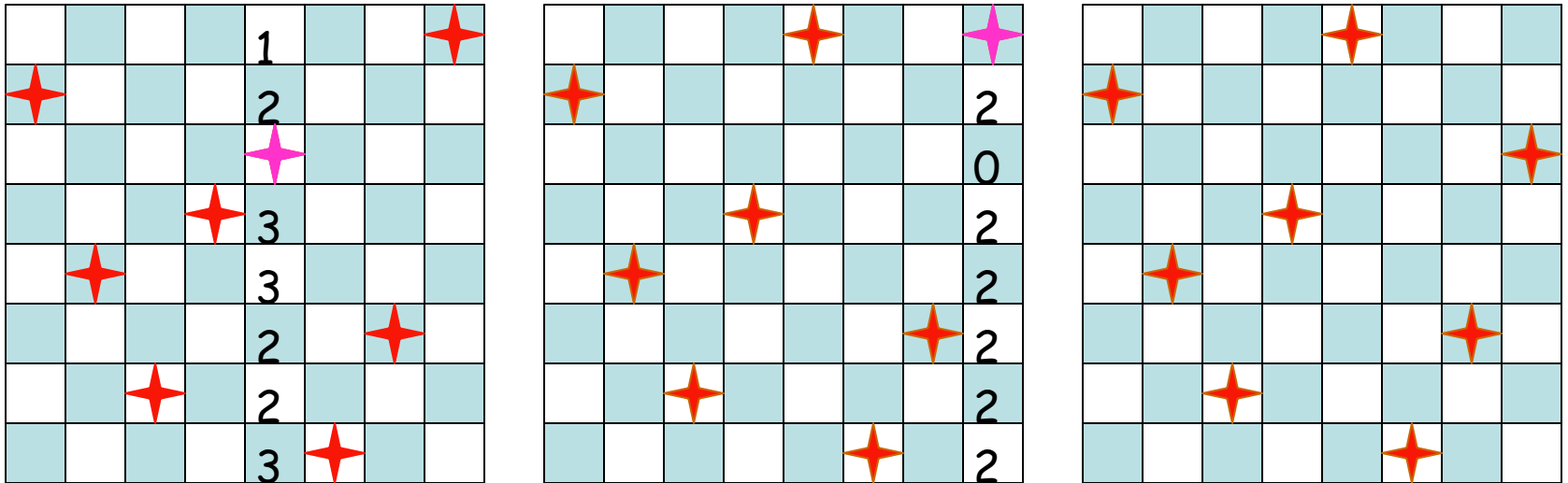
1) S ← initial state
2) Repeat:
   a) S' ← arg $\min_{S' \in SUCCESSORS(S)}\{h(S')\}$
   b) if GOAL?(S') return S'
   c) if h(S') < h(S) then S ← S' else return failure

Similar to:

- hill climbing with –h
- gradient descent over continuous space

Application: 8-Queen

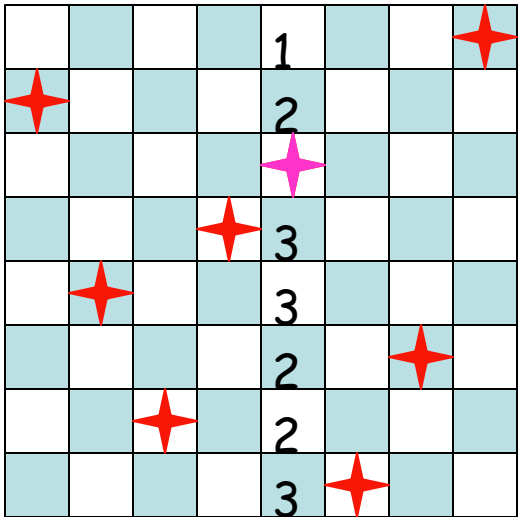
1) Pick an initial state S at random with one queen in each column
2) Repeat k times:
   a) If GOAL?(S) then return S
   b) Pick an attacked queen Q at random
   c) Move Q in its column to minimize the number of attacking queens → new S [min-conflicts heuristic]
3) Return failure

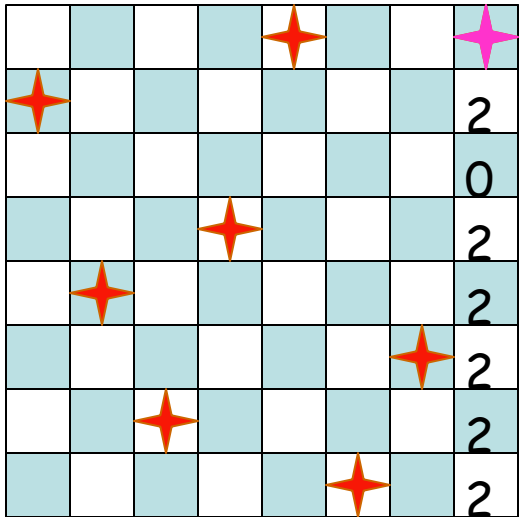


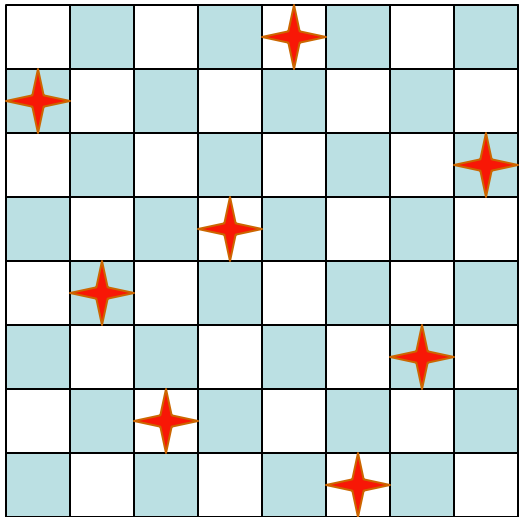
Application: 8-Queen

Repeat n times:

1) Pick an initial state S at random with one queen in each column
2) Repeat k times:
   a) If GOAL?(S) then return S
   b) Pick an attacked queen Q at random
   c) Move Q in its column to minimize the number of attacking queens → new S [min-conflicts heuristic]
3) Return failure







Application: 8-Queen

Re

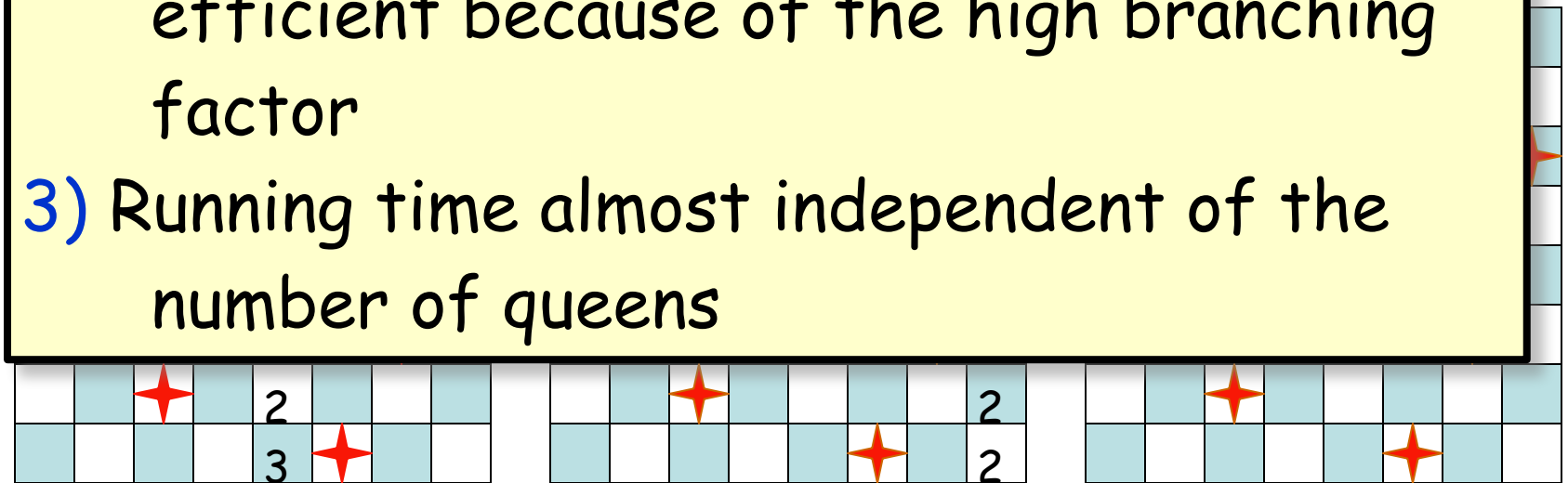
1)

2)

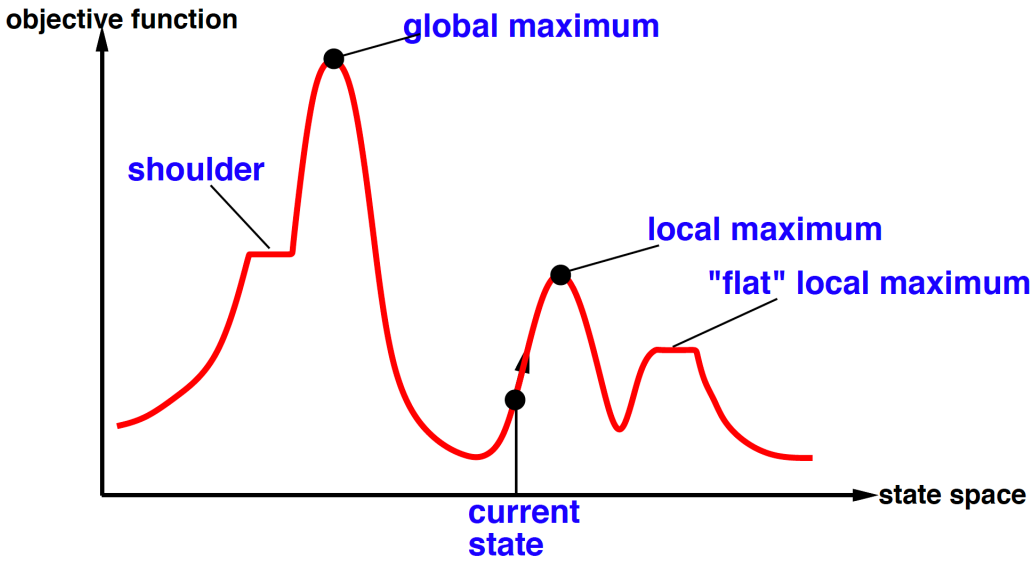
3)

Why does it work ???

1) There are **many** goal states that are well-distributed over the state space
2) If no solution has been found after a few steps, it's better to start it all over again. Building a search tree would be much less efficient because of the high branching factor
3) Running time almost independent of the number of queens



Gradient Descent



may easily get stuck in local minima

→ Random restart (as in n-queen example)

→ Monte Carlo descent

Monte Carlo Descent

1) S ← initial state
2) Repeat k times:
   a) If *GOAL?*(S) then return S
   b) S′ ← successor of S picked at random
   c) if h(S′) ≤ h(S) then S ← S′
   d) else
      - $\Delta h = h(S')-h(S)$
      - with probability ~ $\exp(-\Delta h/T)$, where T is called the "temperature", do: S ← S′ [Metropolis criterion]
3) Return failure

Simulated annealing lowers T over the k iterations.

It starts with a large T and slowly decreases T

Simulated Annealing

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 $\leftarrow$ MAKE-NODE(INITIAL-STATE[*problem*])
for $t \leftarrow$ 1 **to** ∞ **do**
$T \leftarrow$ *schedule*[t]
if $T = 0$ **then return** *current*
next $\leftarrow$ a randomly selected successor of *current*
$\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]
if $\Delta E > 0$ **then** *current* $\leftarrow$ *next*
else *current* $\leftarrow$ *next* only with probability $e^{\Delta\ E/T}$

"Parallel" Local Search Techniques

They perform several local searches concurrently, but not independently:

- Beam search
- Genetic algorithms

See R&N, pages 115-119

Local Beam Search

Idea: keep k states instead of 1; choose top k of all their successors

> Not the same as k searches run in parallel. Searches that find good states recruit other searches to join them

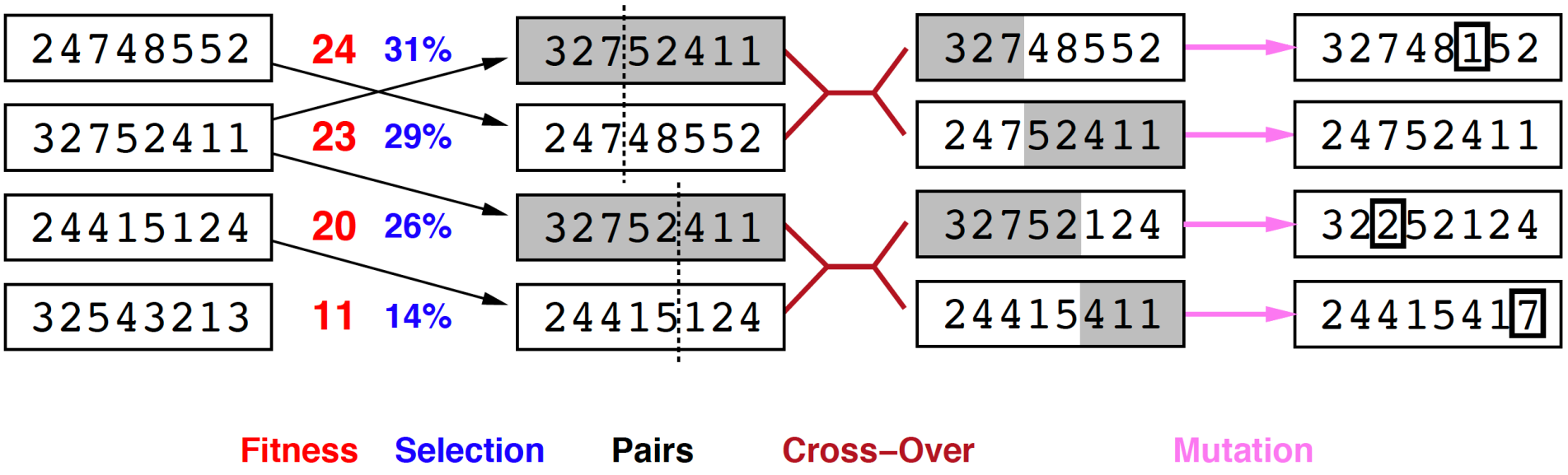
Problem: quite often, all k states end up on same local hill

Idea: choose k successors randomly, biased towards good ones

Observe the close analogy to natural selection!

Genetic Algorithms

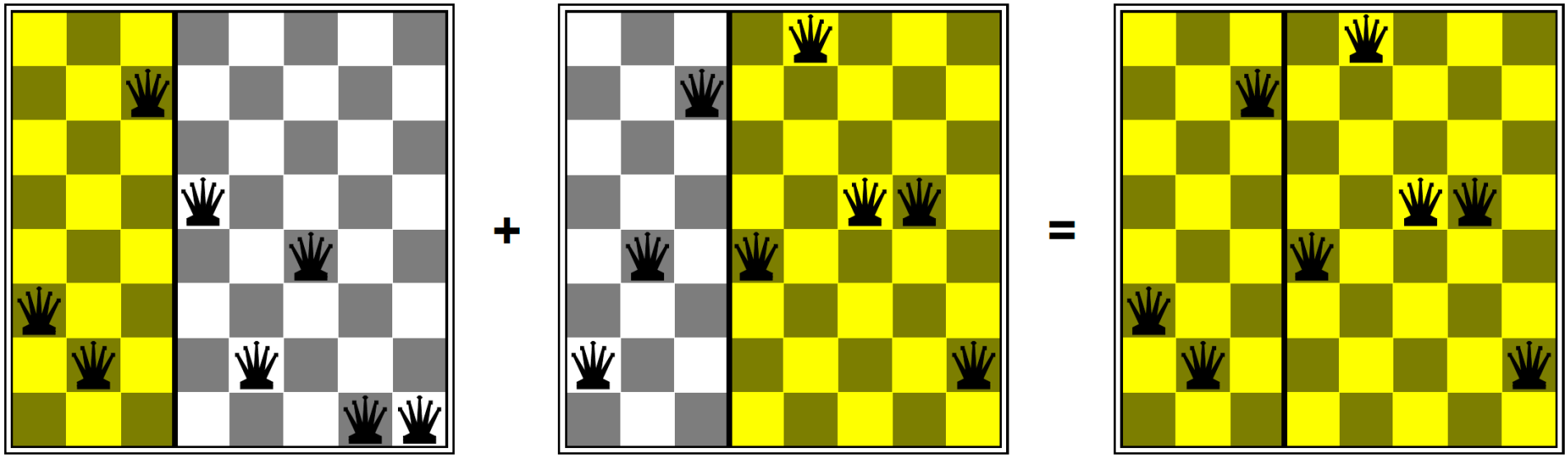
Idea: stochastic local beam search + generate successors from pairs of states

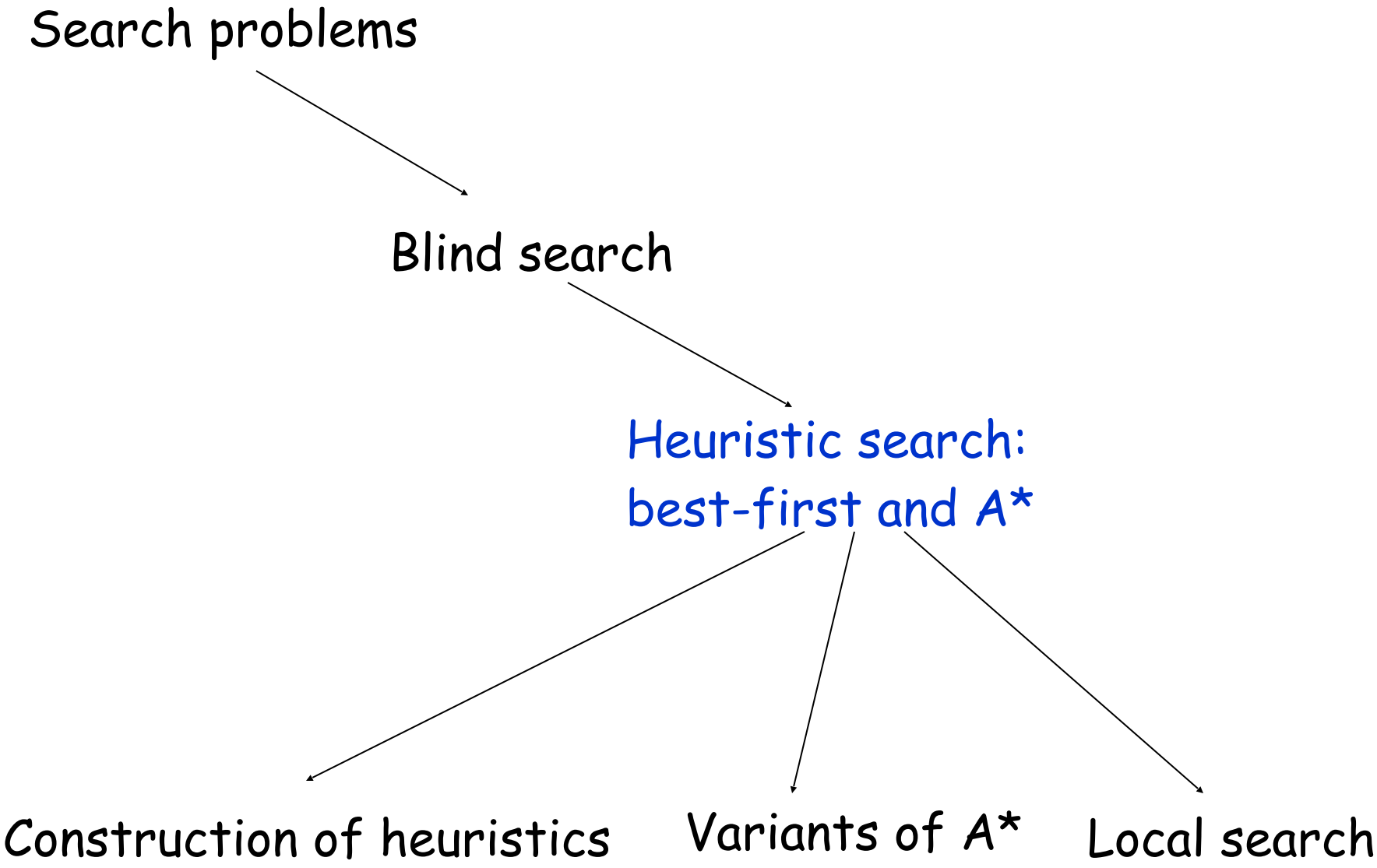


Genetic Algorithms

GAs require states encoded as strings (GPs use programs)

Crossover helps iff substrings are meaningful components




Search problems
Blind search
Heuristic search:
best-first and A*
Construction of heuristics
Variants of A*
Local search