Informed Search Strategies and Local Search Algorithms

Contents

- Informed Search Strategies
  - Greedy Best-first Search
  - A^* Search
- Local Search Algorithms
  - Hill-climbing Search
  - Simulated Annealing Search
  - Genetic Algorithms
  - Continuous State Spaces
  - Constrained Optimization Problem

Informed Search Strategies

- Applies evaluation function to determine the order of the nodes in the queue
  - Evaluation function $f(n)$ = distance through node n to the goal
    - Comes from problem-specific knowledge
  - Priority queue maintains the order of the frontier nodes
- Components of evaluation function:

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

  - $g(n)$ = path cost from the start node to n
  - Heuristic function $h(n)$ = estimated cost of the cheapest path from node n to the goal

Greedy Best-first Search

- Expands the node closest to the goal
  - $f(n) = h(n)$
    - Evaluates nodes by using just the heuristic function
  - Takes the biggest bite possible $\rightarrow$ the name "greedy search"
  - Note that in uniform-cost search $f(n) = g(n)$
    - g does not direct search toward the goal

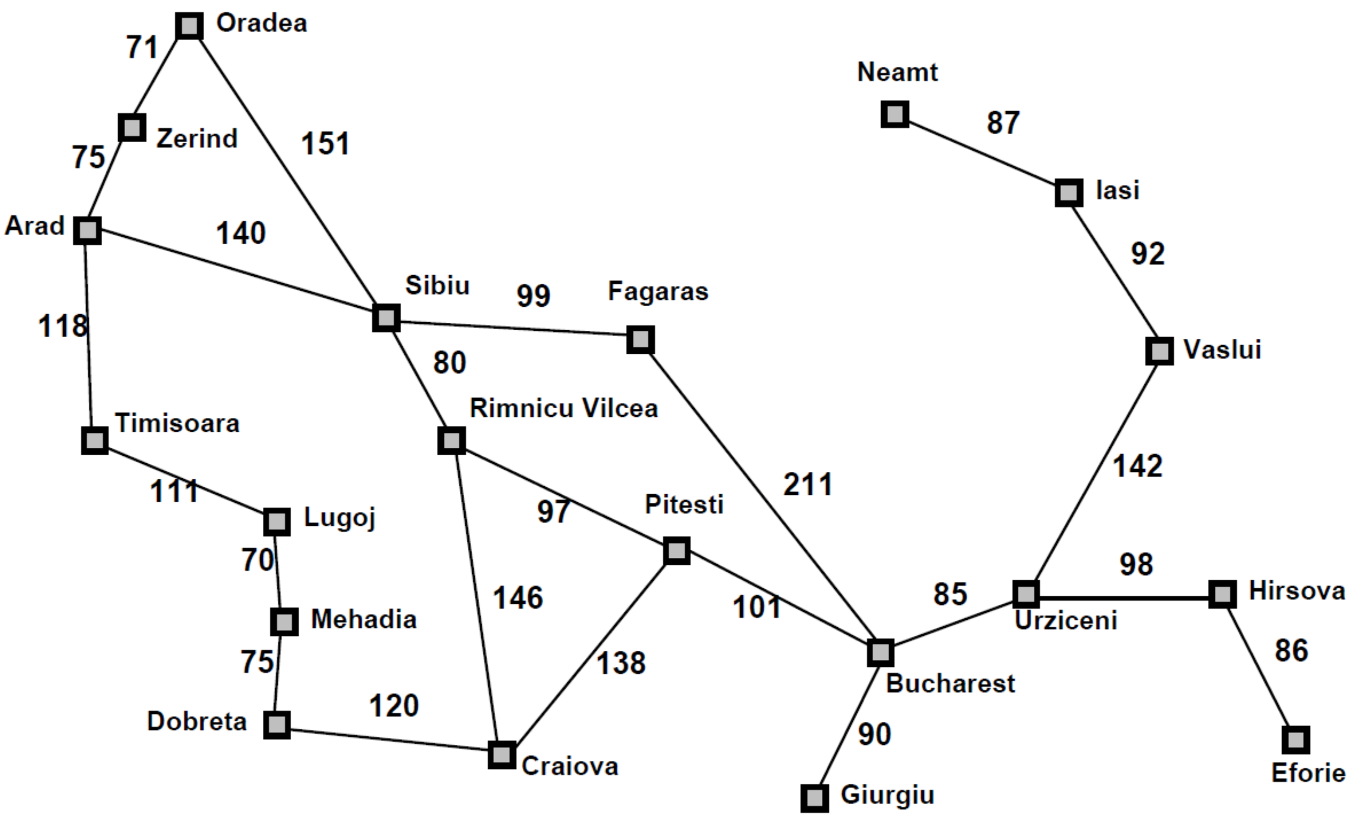
      (uniform-cost search is not an informed search)

<u>Example</u>:

A good heuristic function for route-finding problem

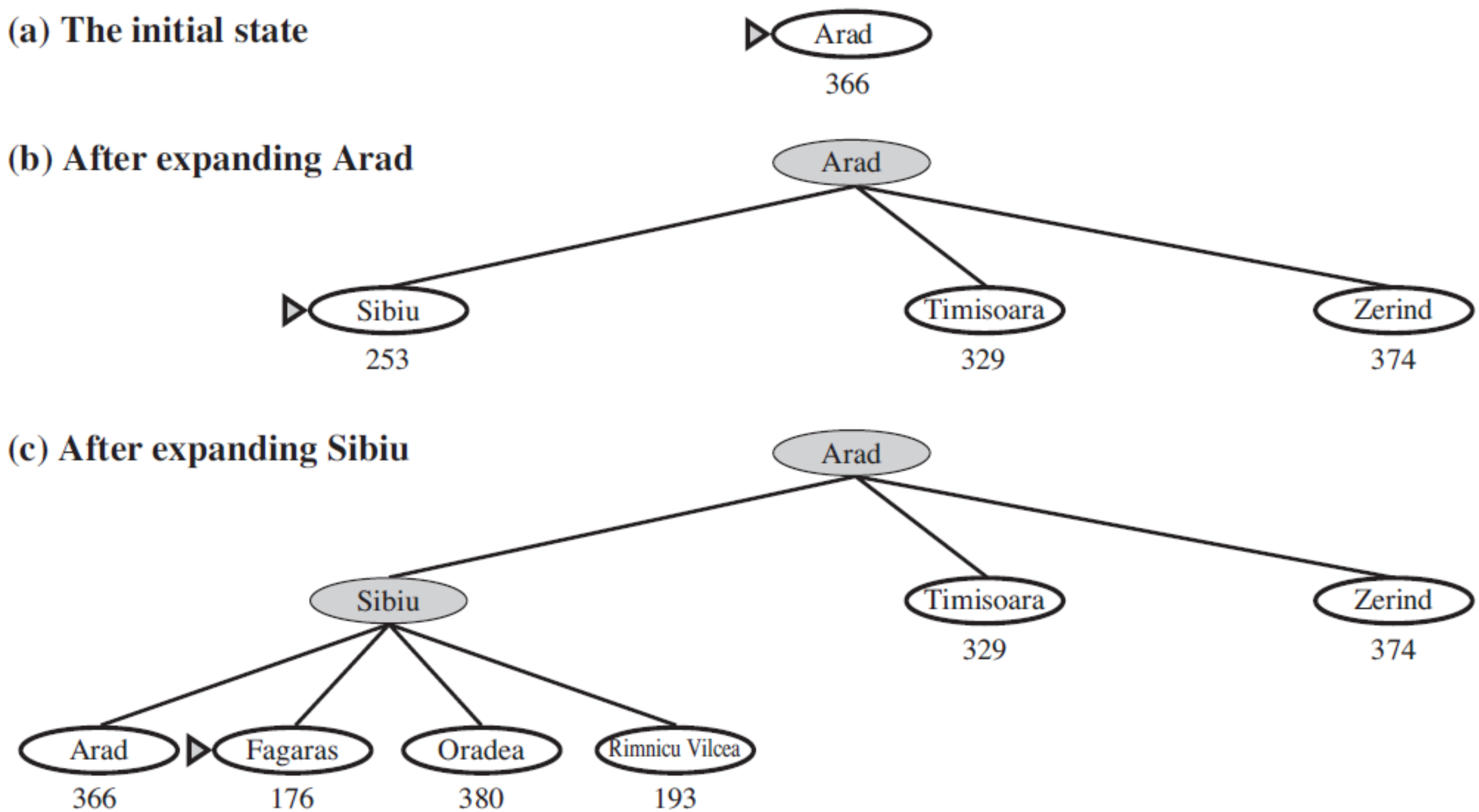
$h_{SLD}(n)$ = straight-line distance between n and the goal location

Greedy Best-first Search



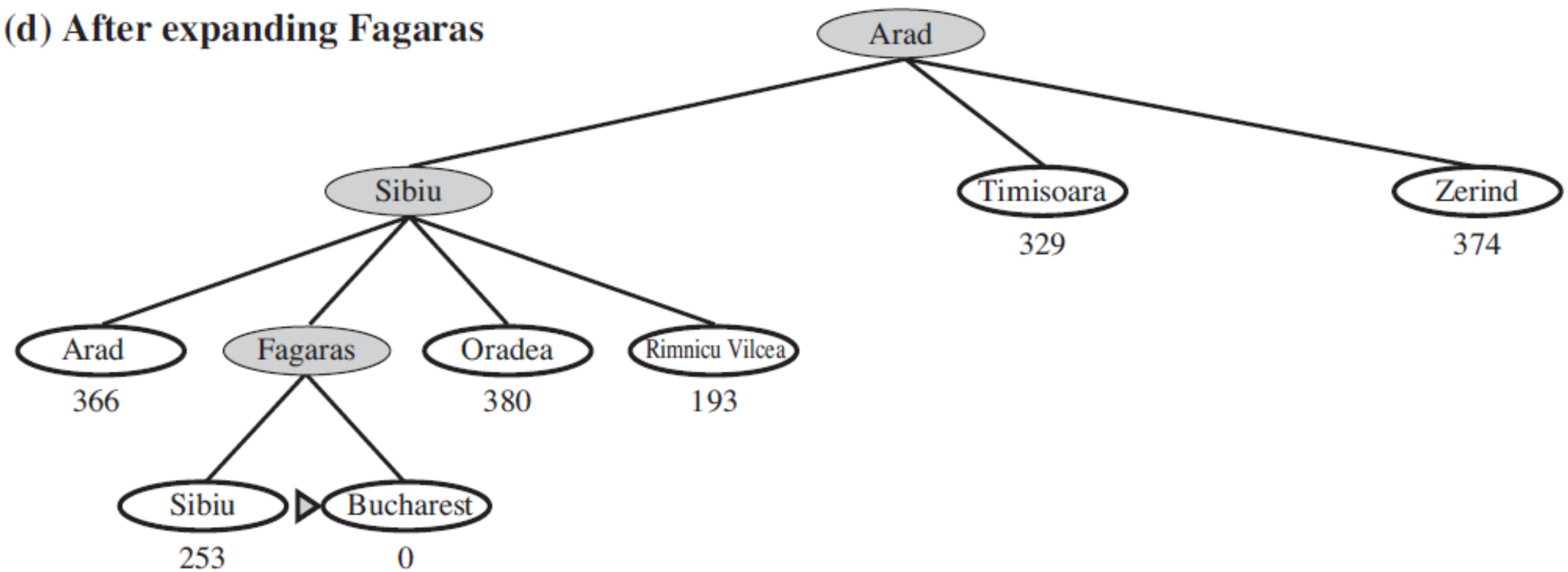
Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Best-first Search



Greedy best-first tree search

Greedy Best-first Search



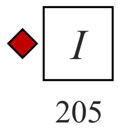
Greedy best-first tree search

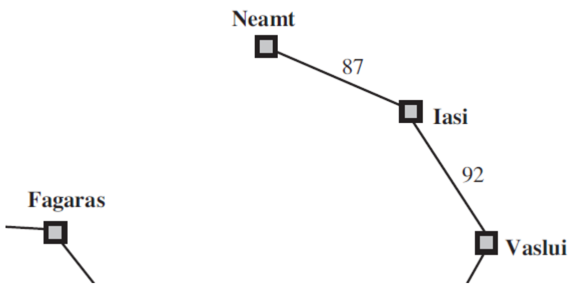
Greedy Best-first Search

- Evaluation (Greedy best-first *tree* search):
  - Neither optimal nor complete even in finite space
    - Infinite loop from Iasi to Fagaras because of Neamt ☞
  - Time and space complexity
    - Retains all leaf nodes in memory
    - $O(b^m)$, m : maximum depth of the search space
    - Good heuristic function can reduce the space and time complexity
- Greedy best-first *graph* search is complete in finite space
  - The implementation is identical to uniform-cost search except for the use of h instead of g ☞

Greedy Best-first Search

Greedy best-first tree search

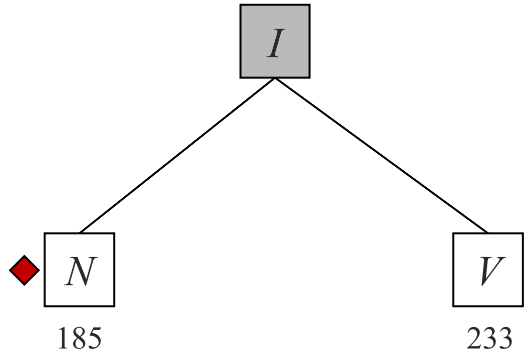


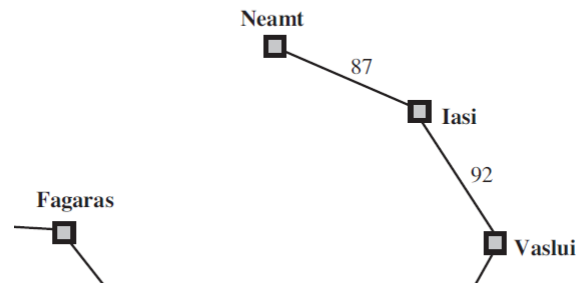


An infinite loop by greedy best-first tree search: From Iasi to Fagaras

Greedy Best-first Search

Greedy best-first tree search

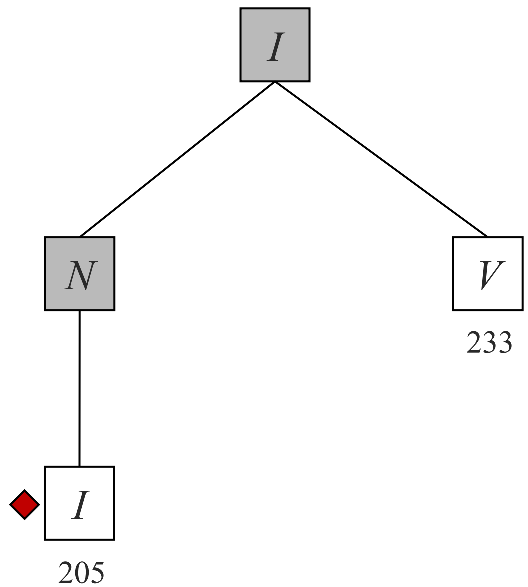


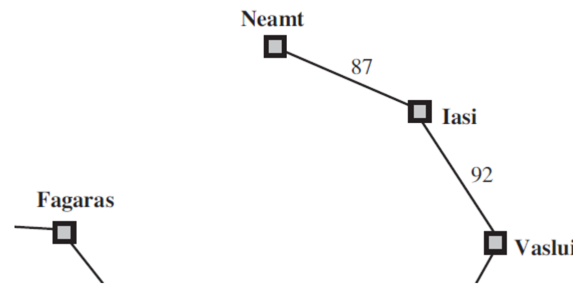


An infinite loop by greedy best-first tree search: From Iasi to Fagaras

Greedy Best-first Search

Greedy best-first tree search

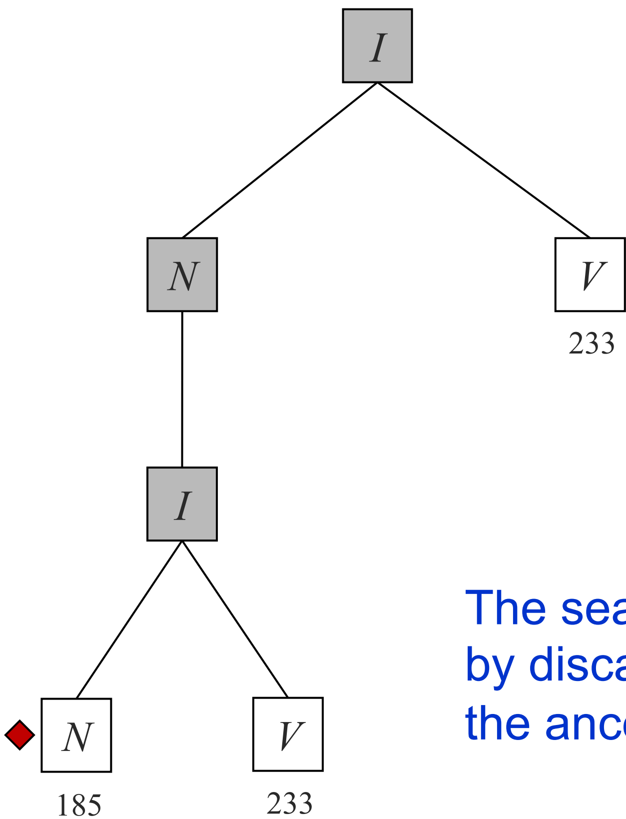


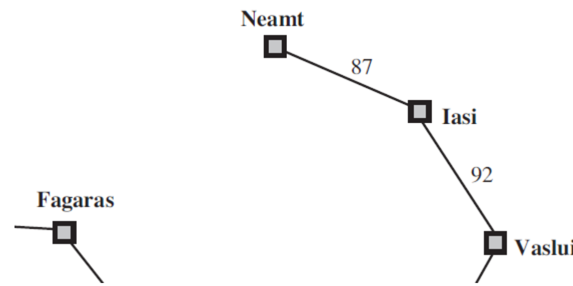


An infinite loop by greedy best-first tree search: From Iasi to Fagaras

Greedy Best-first Search

Greedy best-first tree search



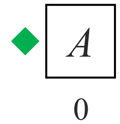


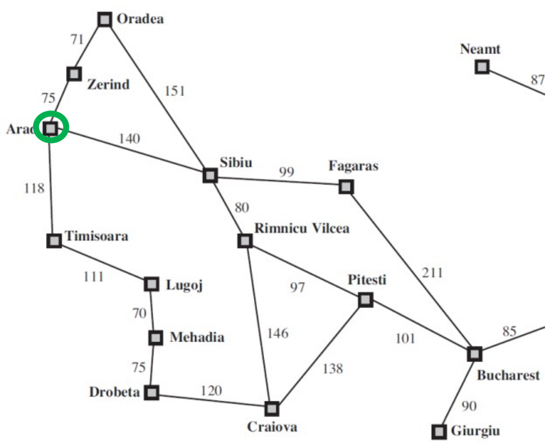
The search can be made complete in finite space by discarding the child that is the same as one of the ancestors

An infinite loop by greedy best-first tree search: From Iasi to Fagaras

Greedy Best-first Search

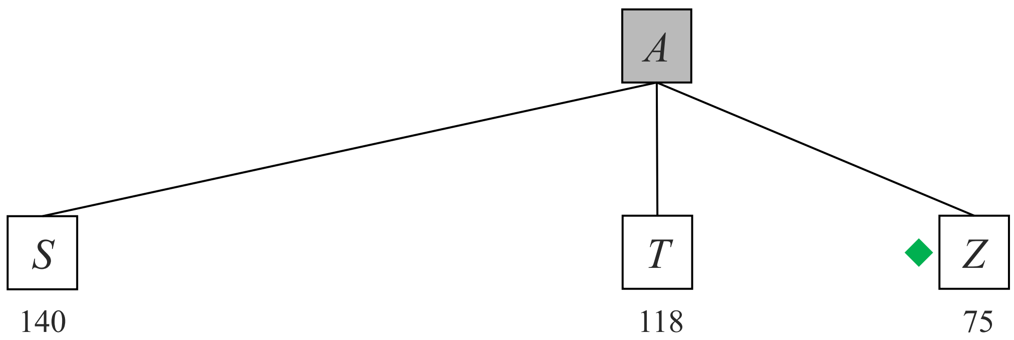
Uniform-cost search (review):

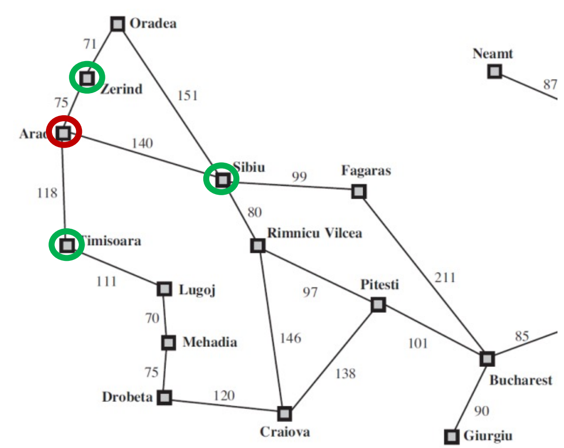




Greedy Best-first Search

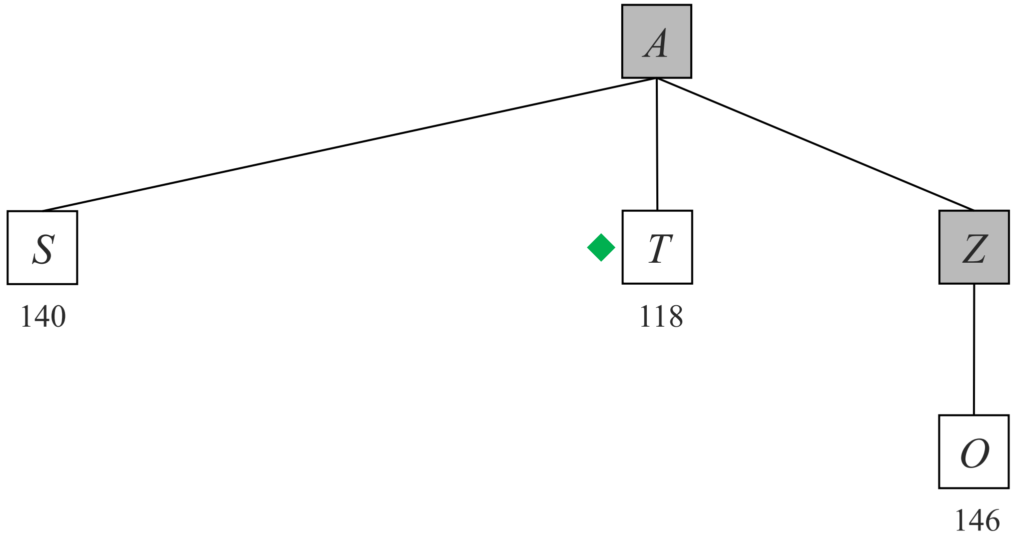
Uniform-cost search:

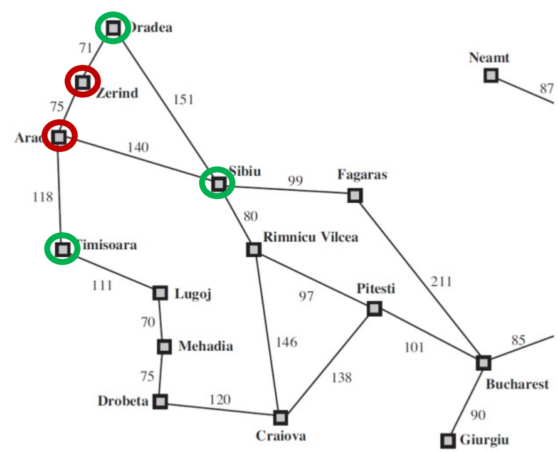




Greedy Best-first Search

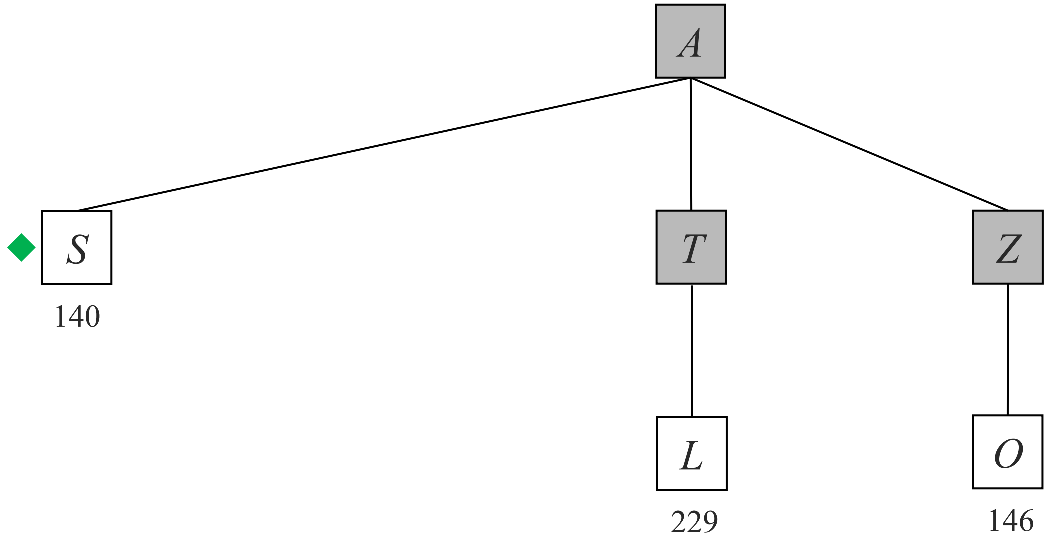
Uniform-cost search:

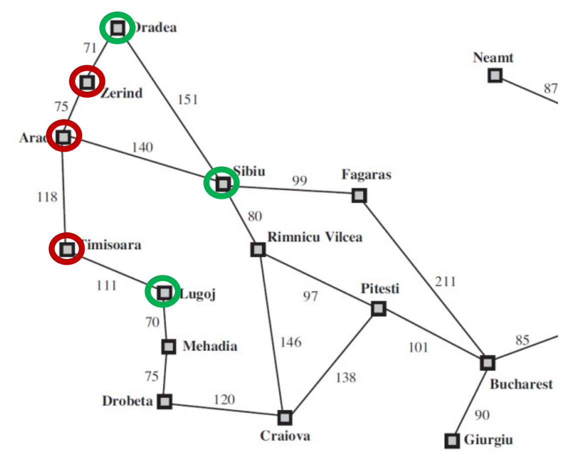




Greedy Best-first Search

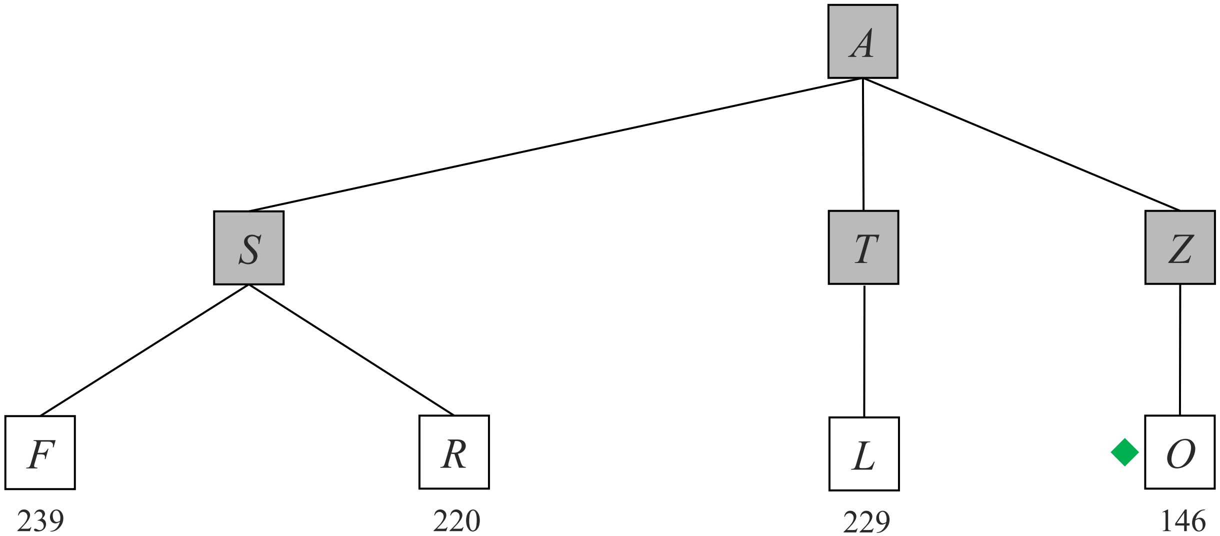
Uniform-cost search:

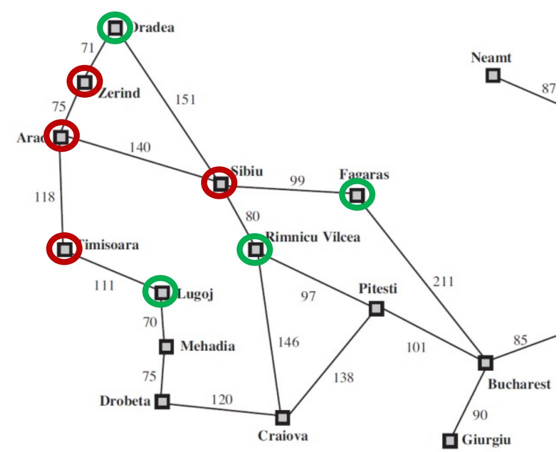




Greedy Best-first Search

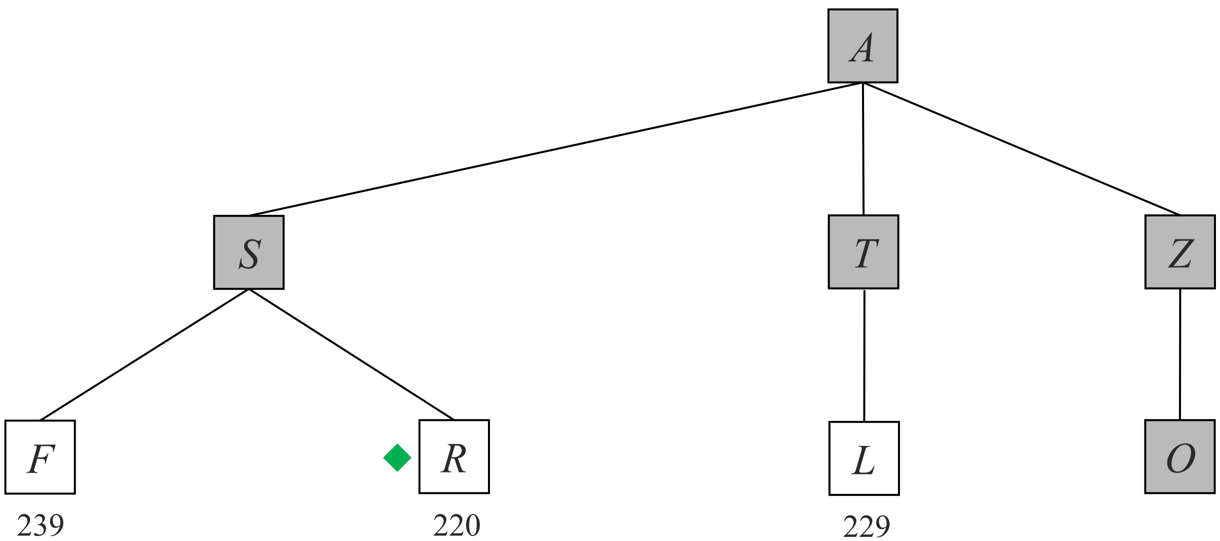
Uniform-cost search:

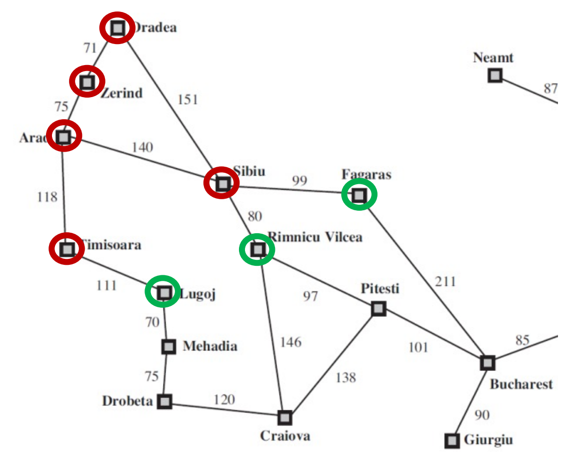




Greedy Best-first Search

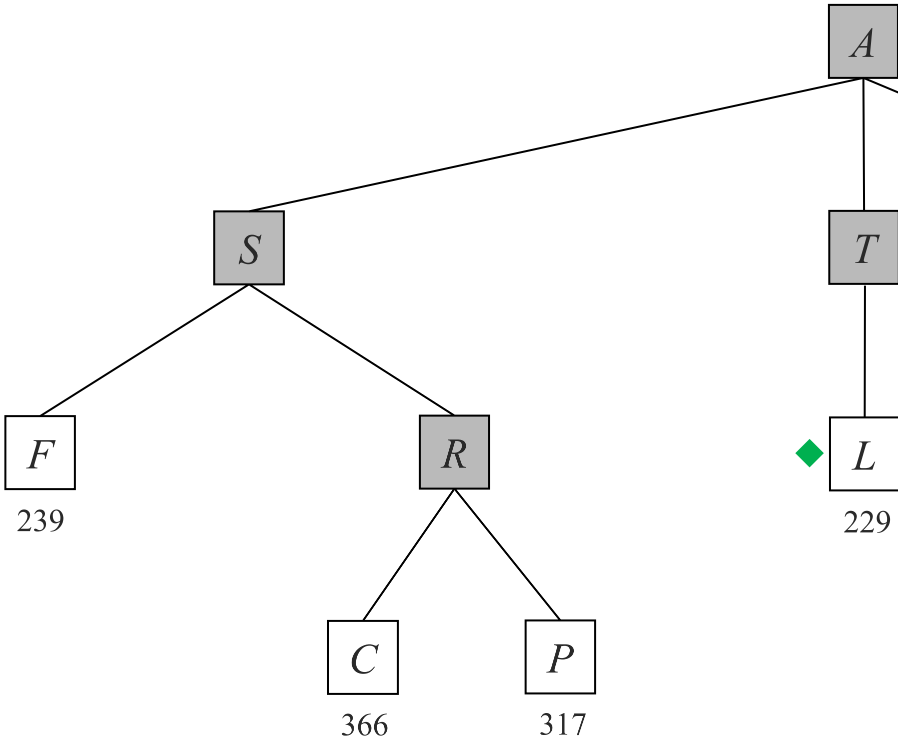
Uniform-cost search:

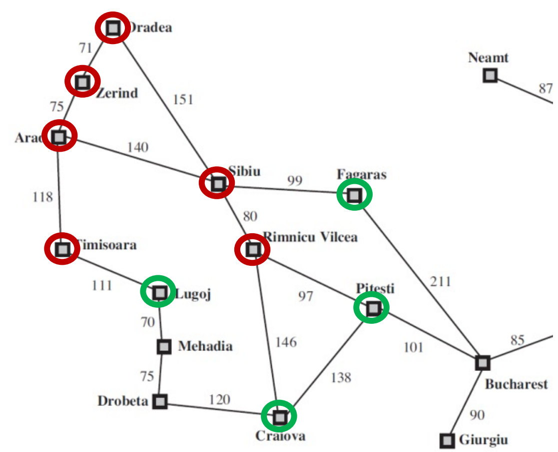




Greedy Best-first Search

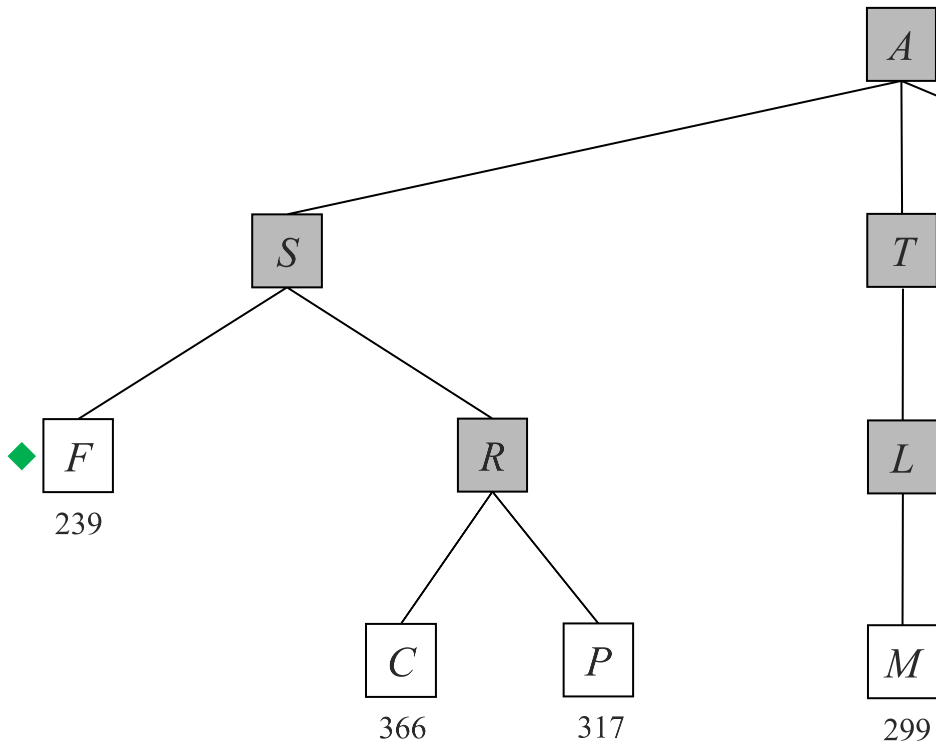
Uniform-cost search:

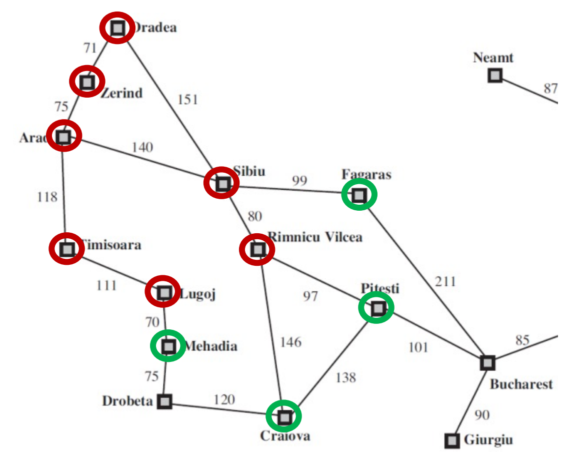




Greedy Best-first Search

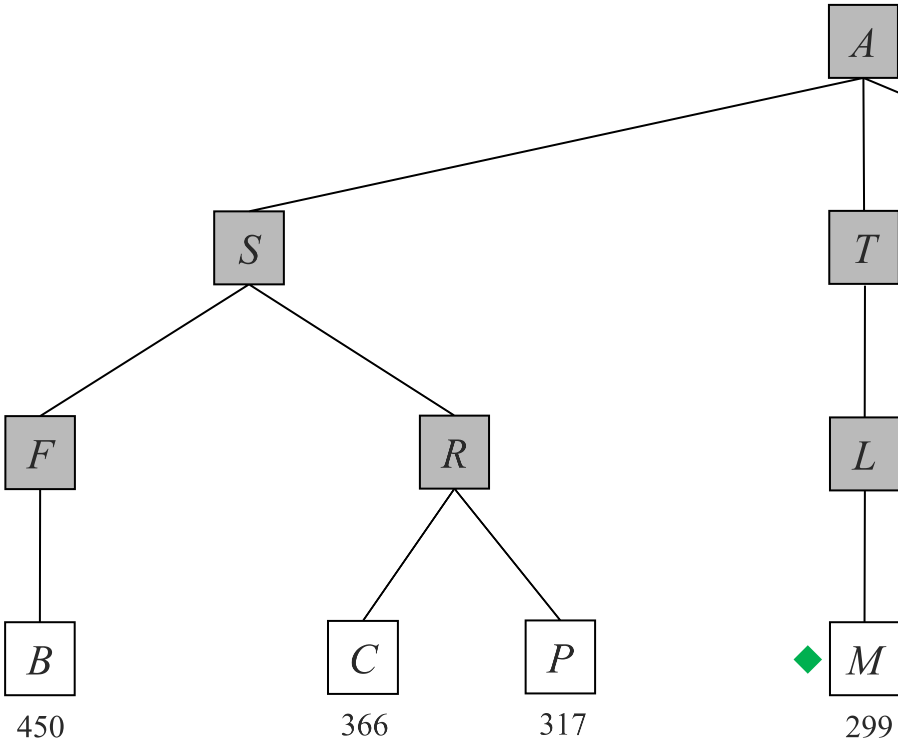
Uniform-cost search:

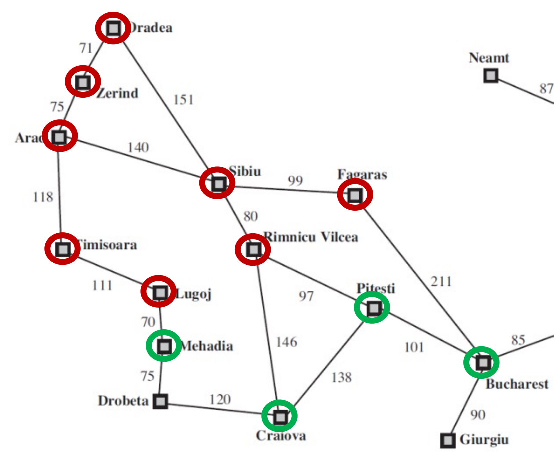




Greedy Best-first Search

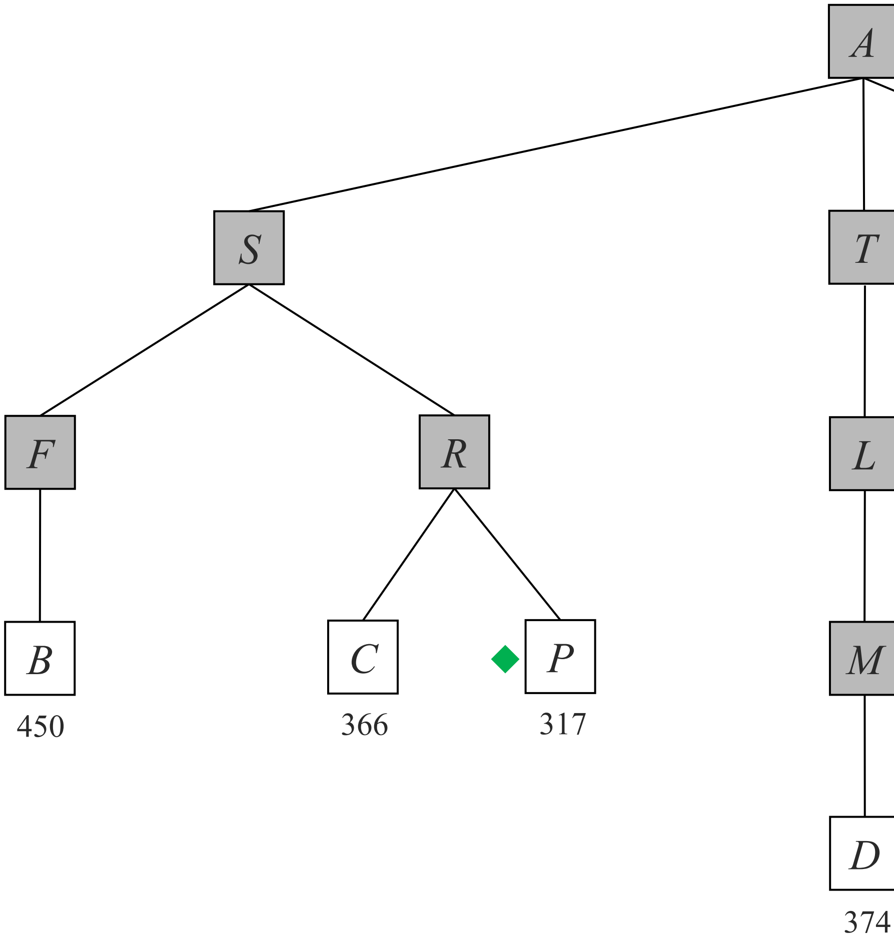
Uniform-cost search:

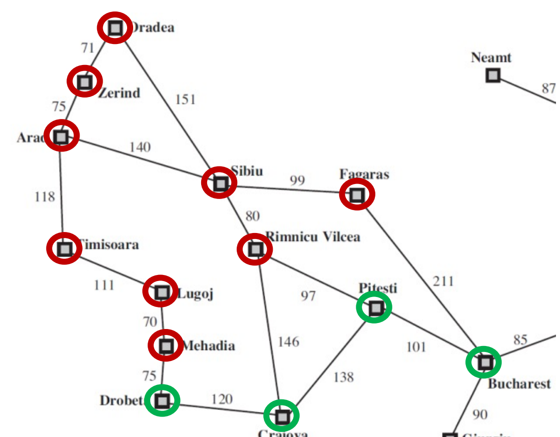




Greedy Best-first Search

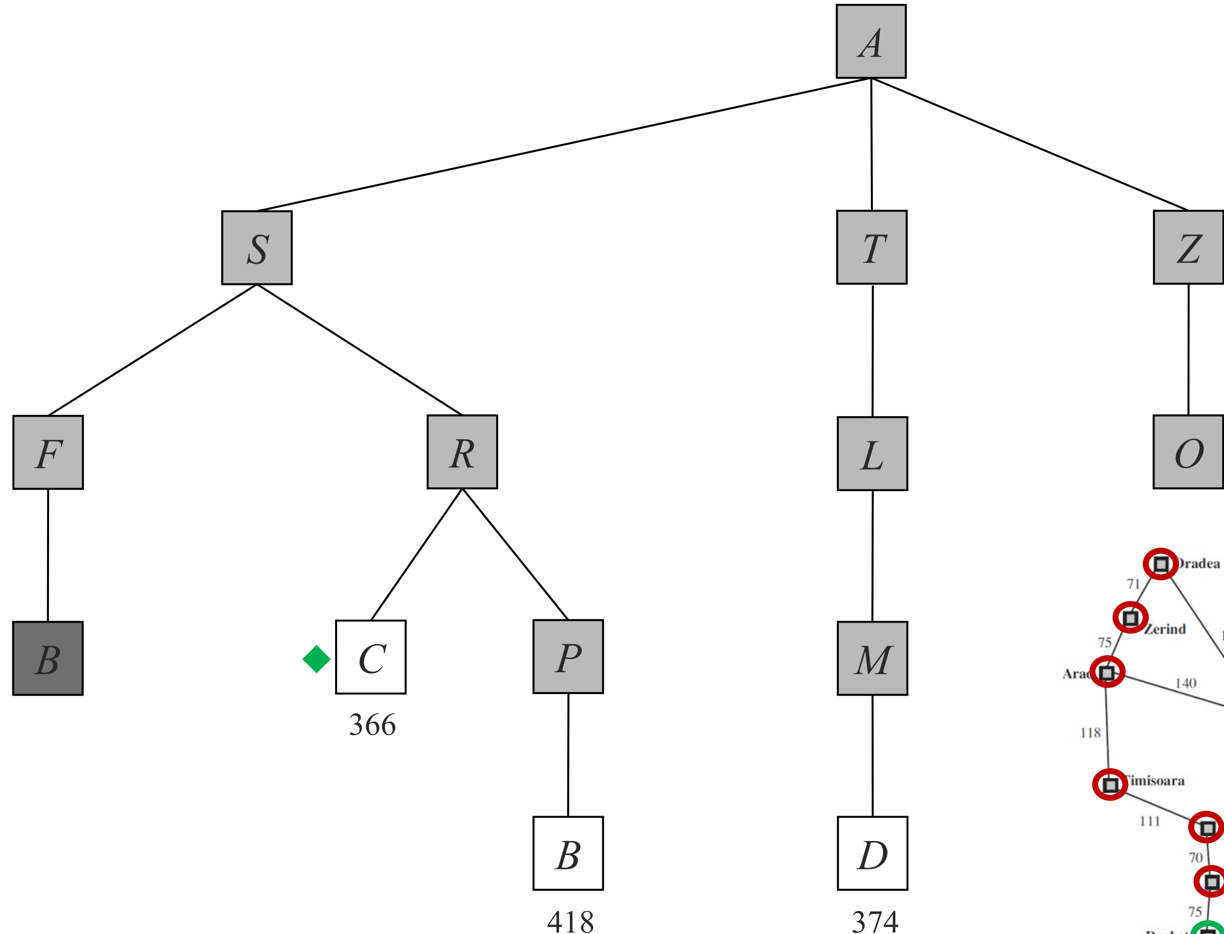
Uniform-cost search:

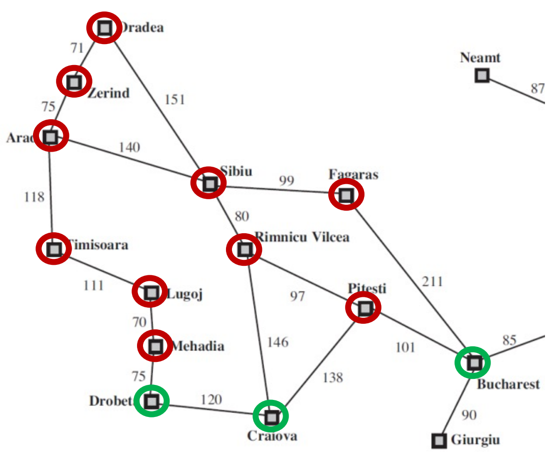




Greedy Best-first Search

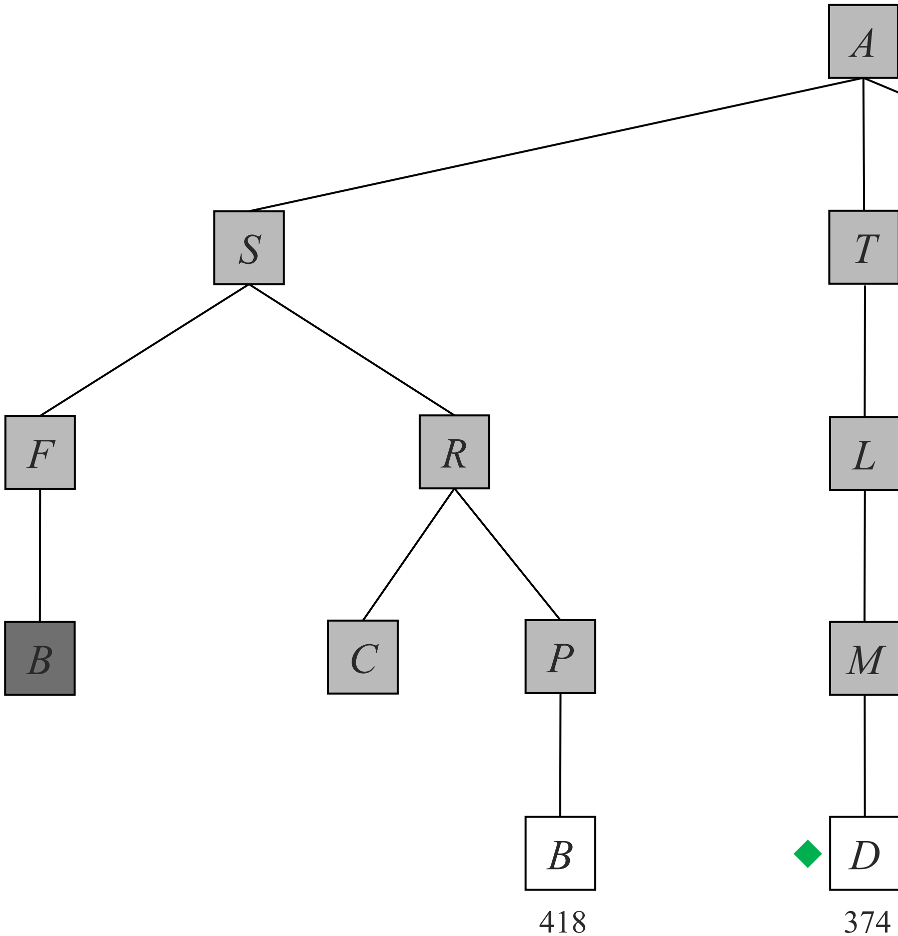
Uniform-cost search:

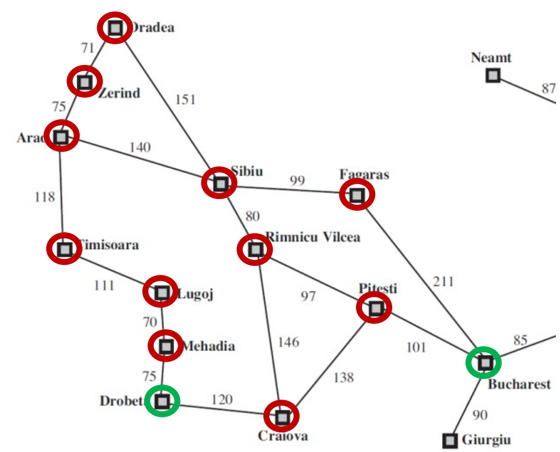




Greedy Best-first Search

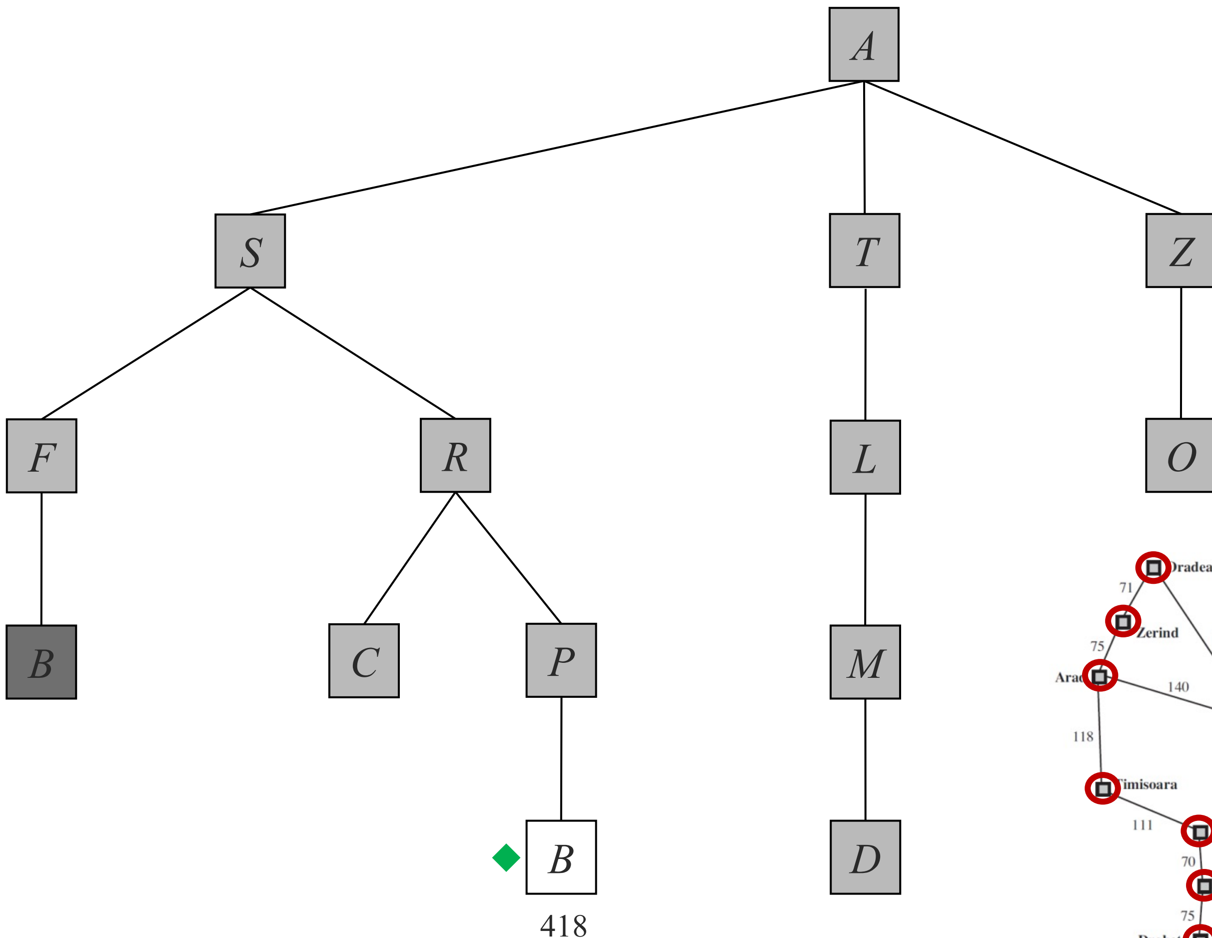
Uniform-cost search:

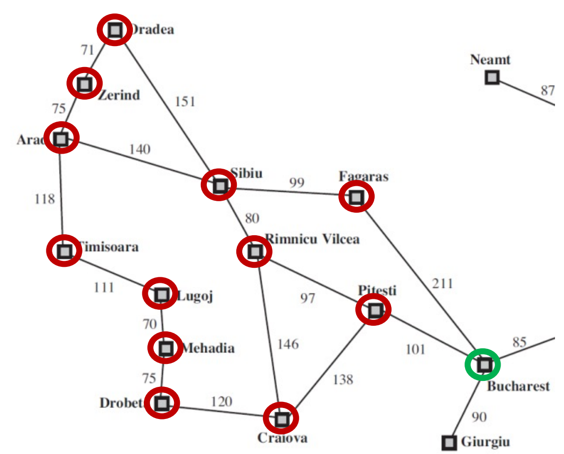




Greedy Best-first Search

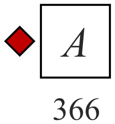
Uniform-cost search:

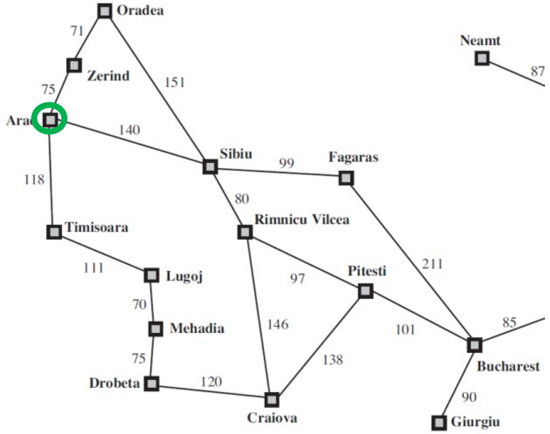




Greedy Best-first Search

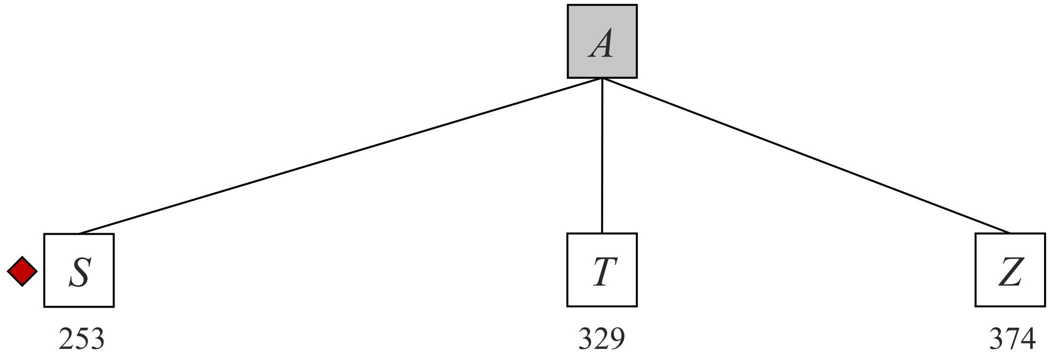
Greedy best-first graph search:

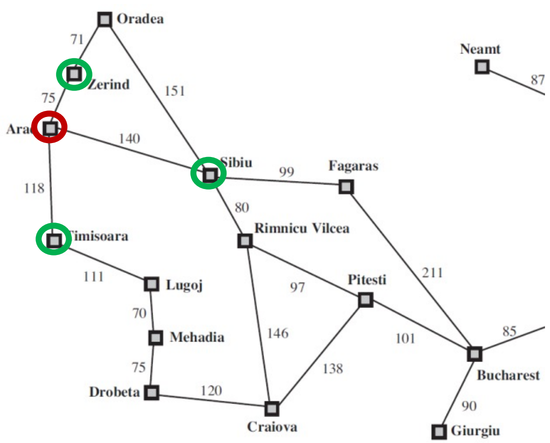




Greedy Best-first Search

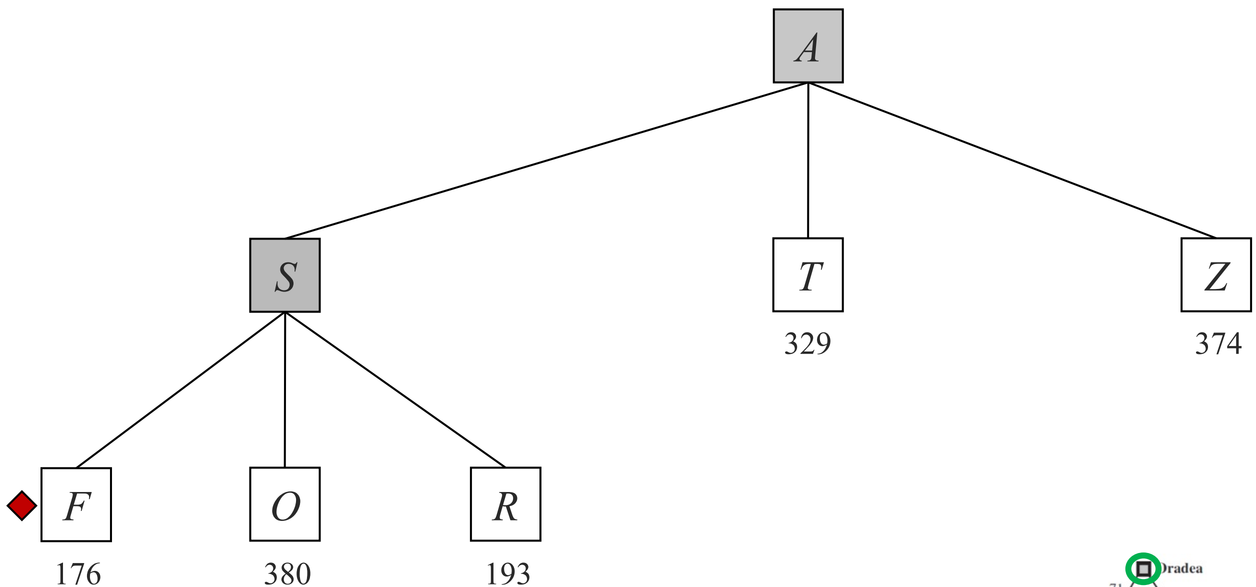
Greedy best-first graph search:

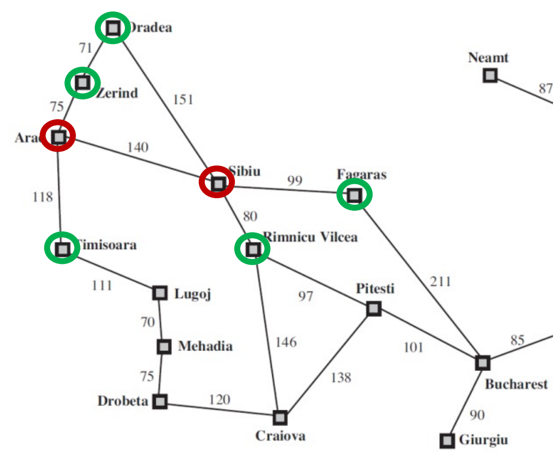




Greedy Best-first Search

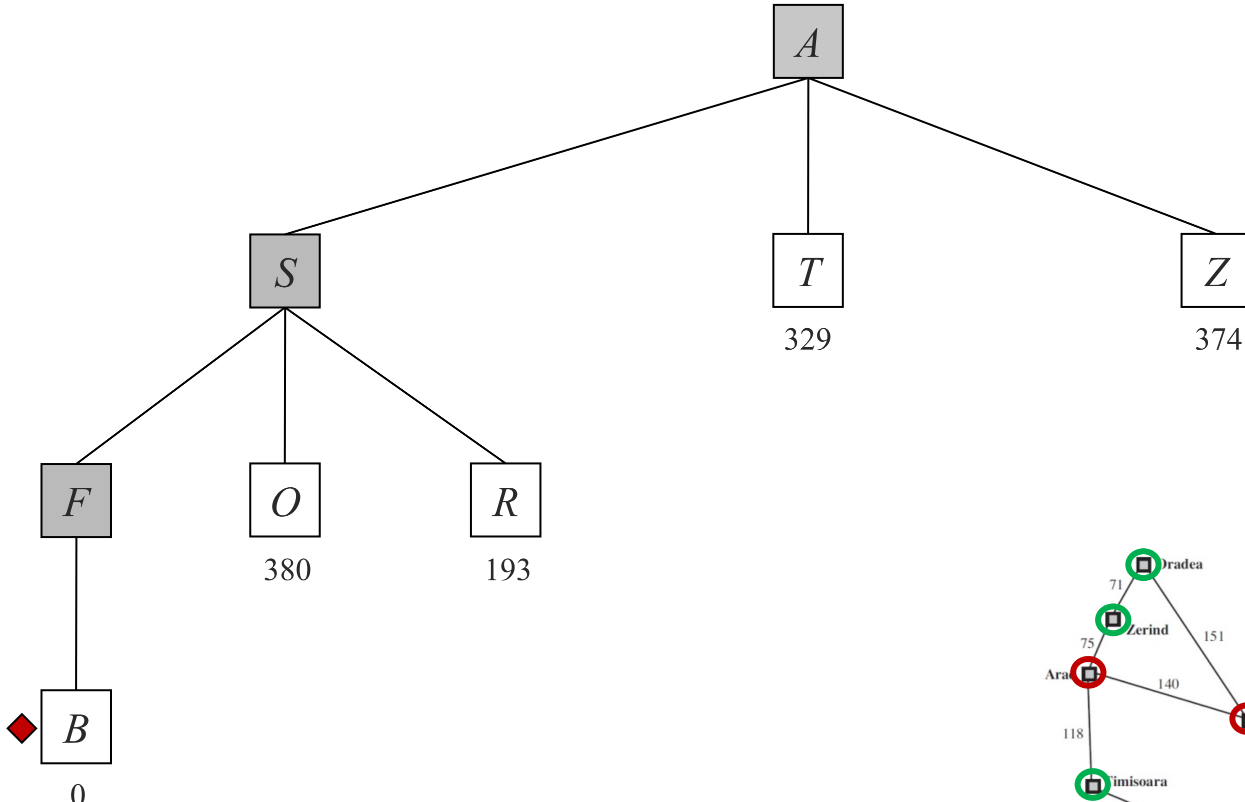
Greedy best-first graph search:

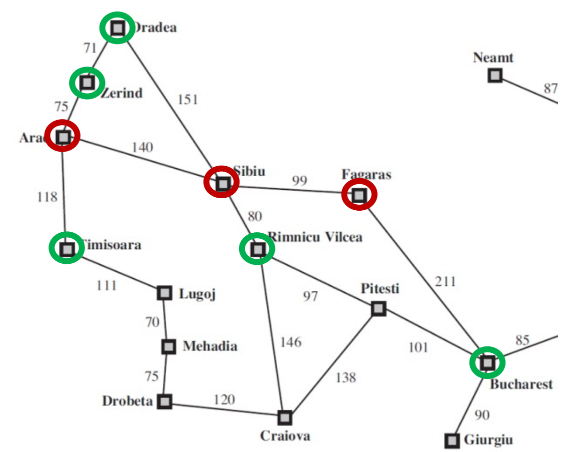




Greedy Best-first Search

Greedy best-first graph search:





A* Search

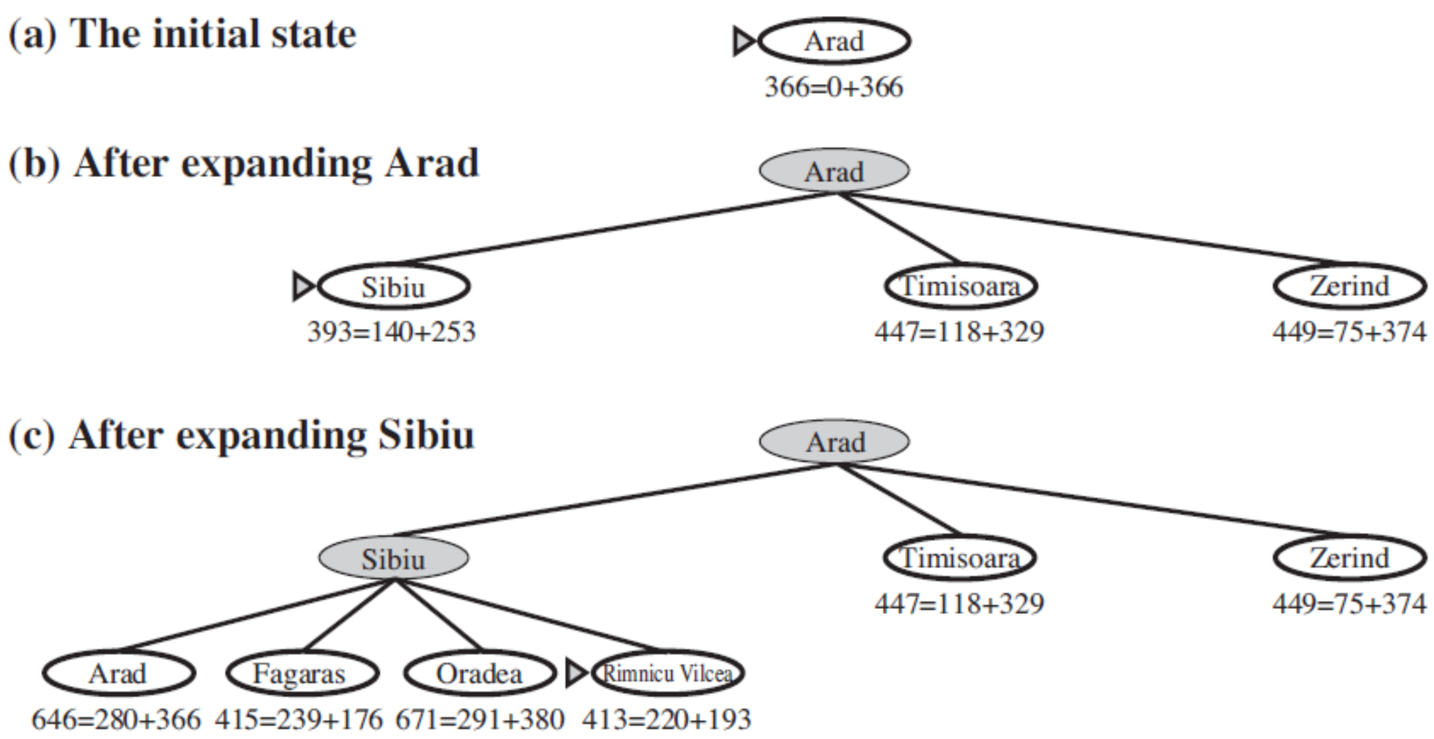
- Minimizes the total estimated solution cost
- Combination of greedy best-first search and uniform-cost search

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

  - $g(n)$: path cost from the start node to n
  - $h(n)$: estimated cost of the cheapest path from n to the goal
  - $f(n)$: estimated cost of the cheapest solution through n
- A* graph search is identical to uniform-cost search except that A* algorithm uses $g + h$ instead of g

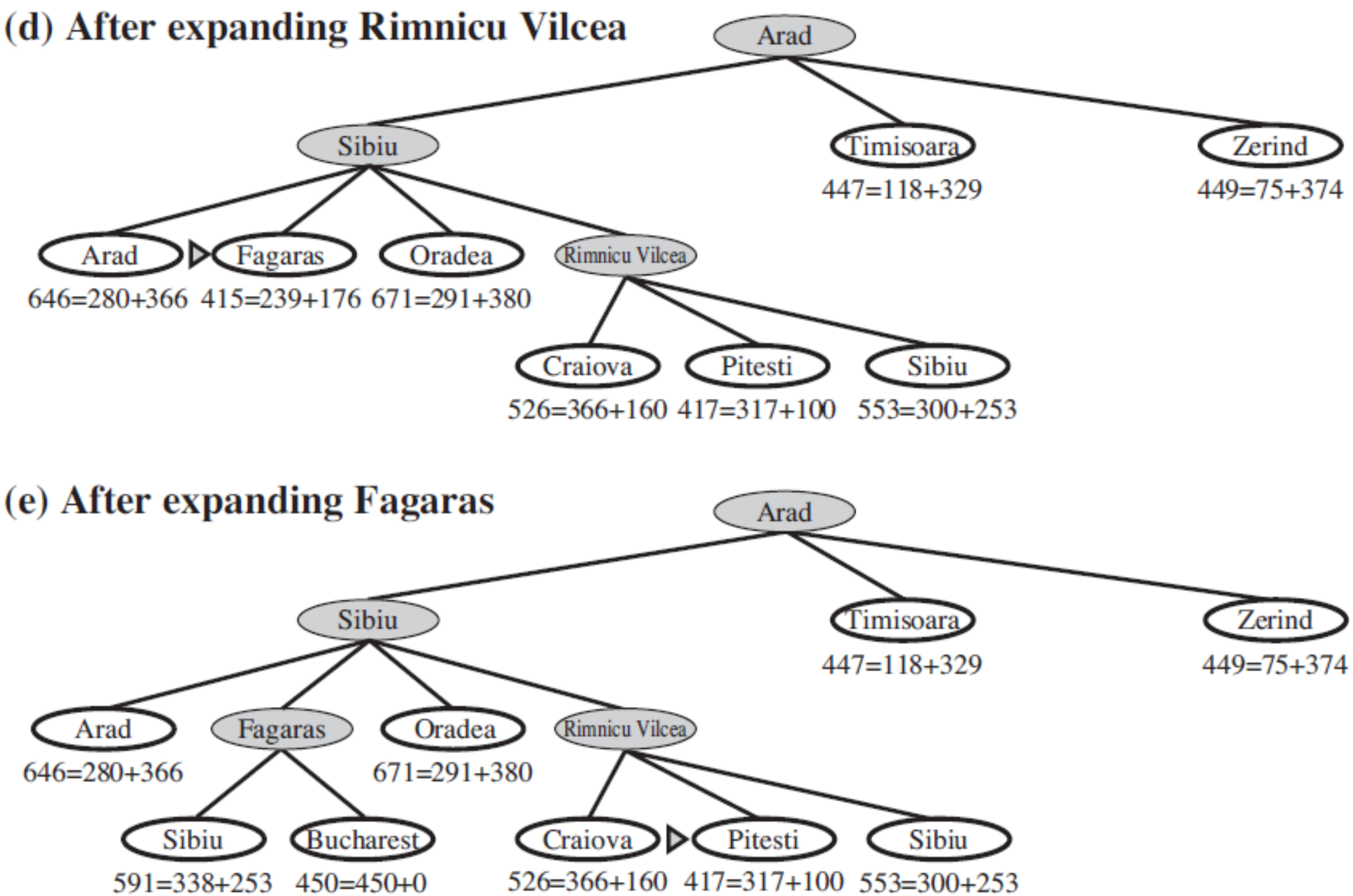
A* Search

A* tree search:



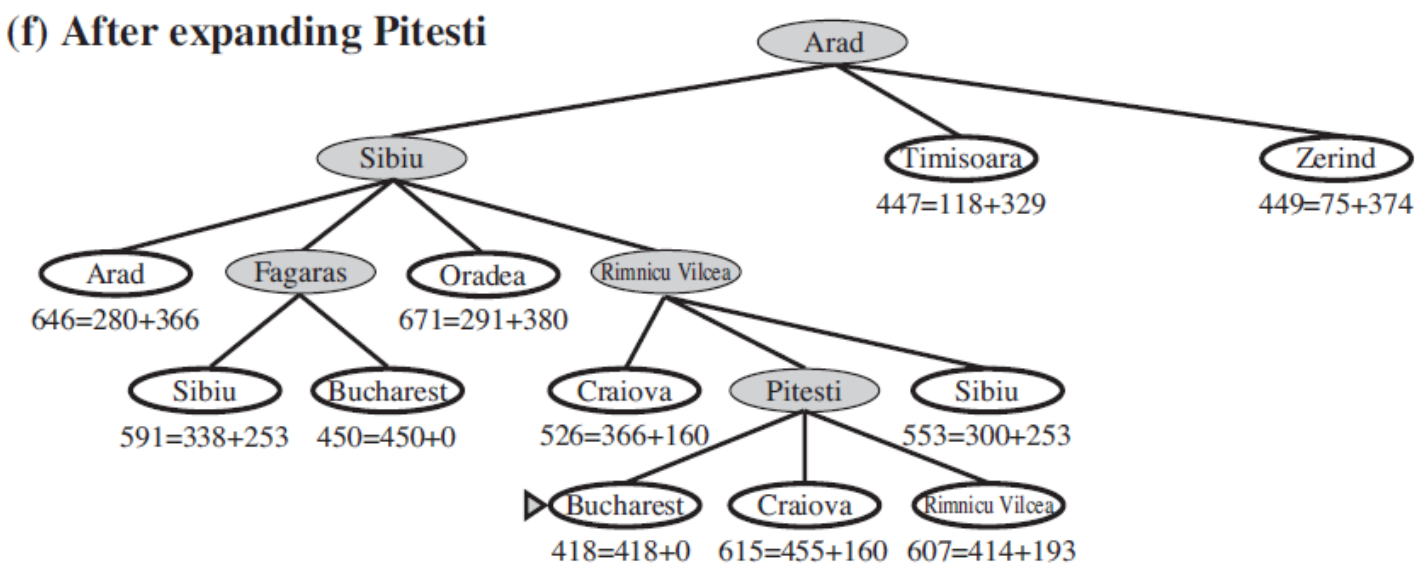
A* Search

A* tree search:



A* Search

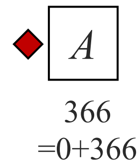
A* tree search:

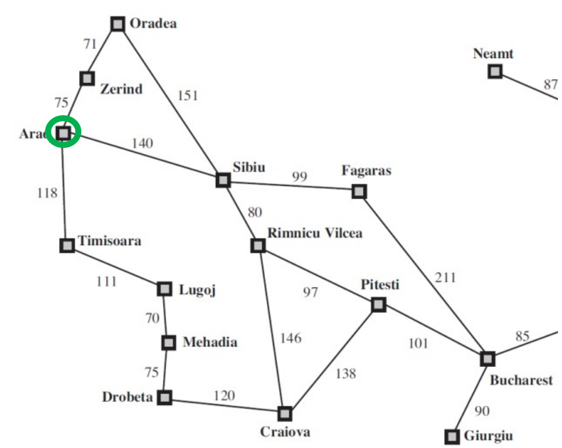


Number of nodes generated: 15

A* Search

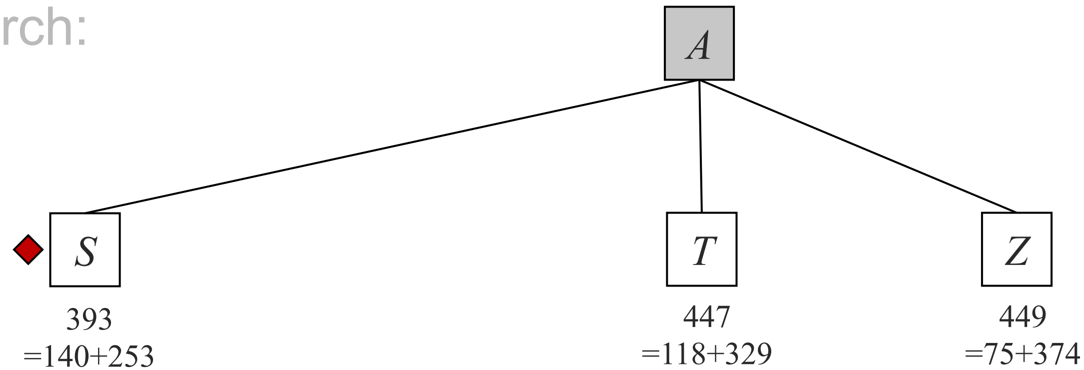
A* graph search:

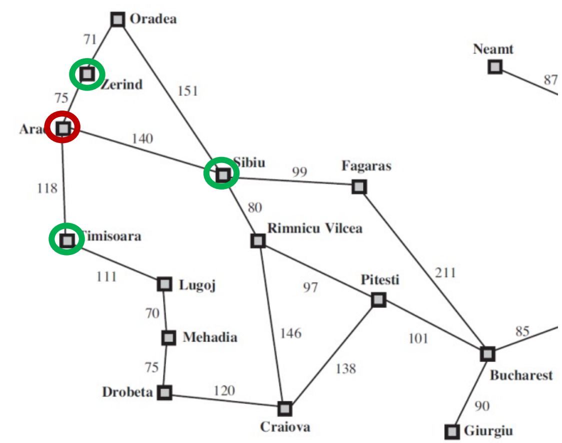




A* Search

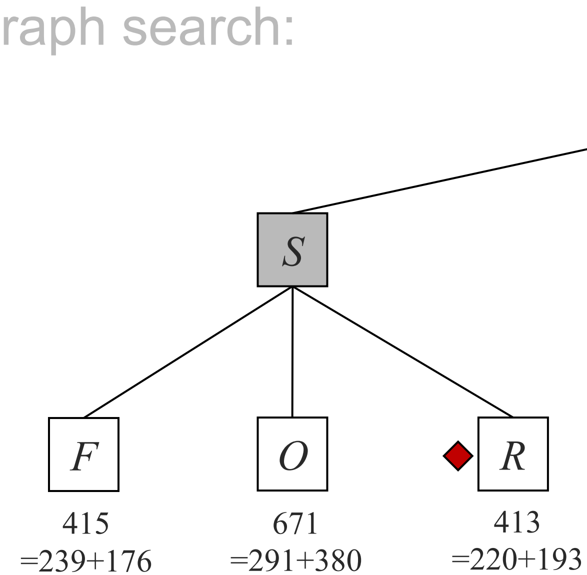
A* graph search:

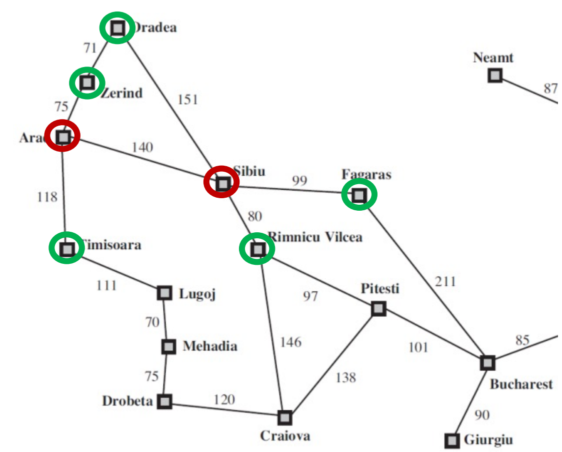




A* Search

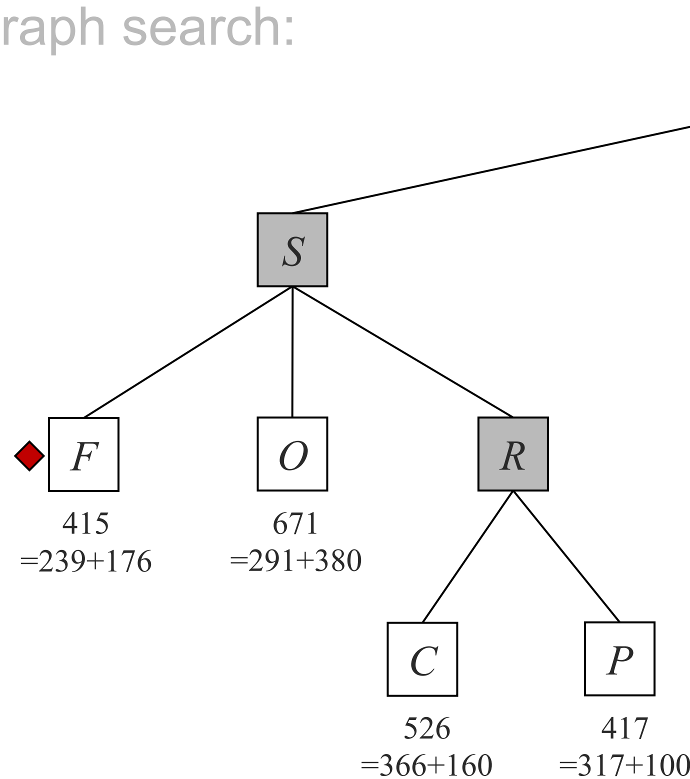
A* graph search:

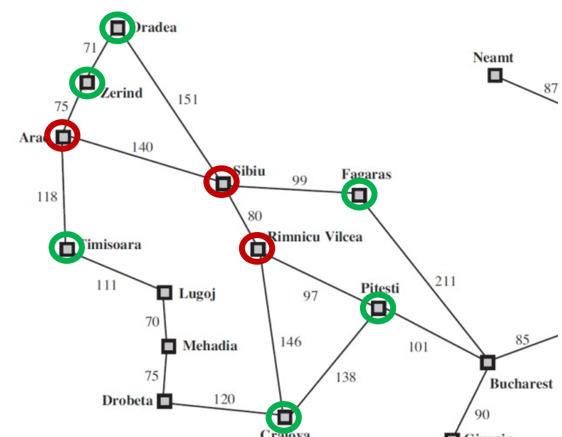




A* Search

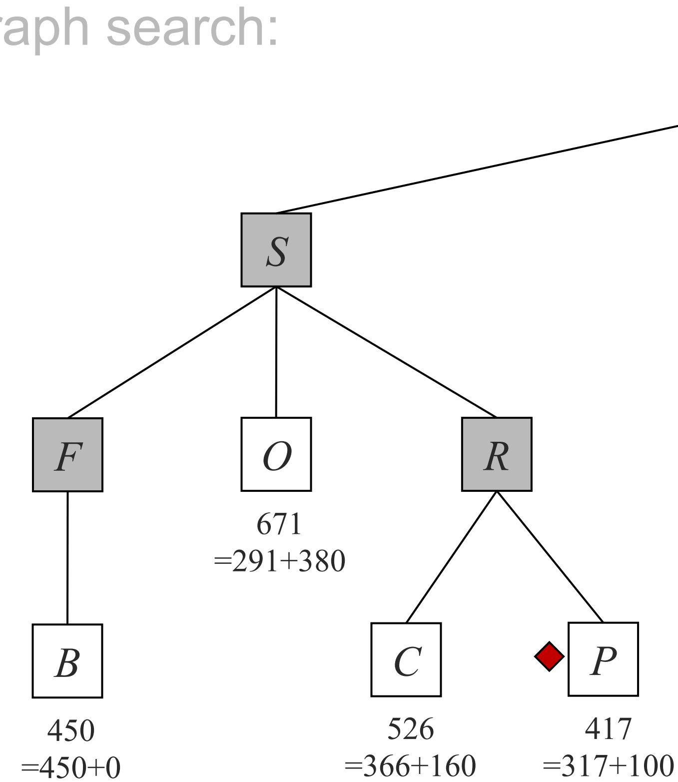
A* graph search:

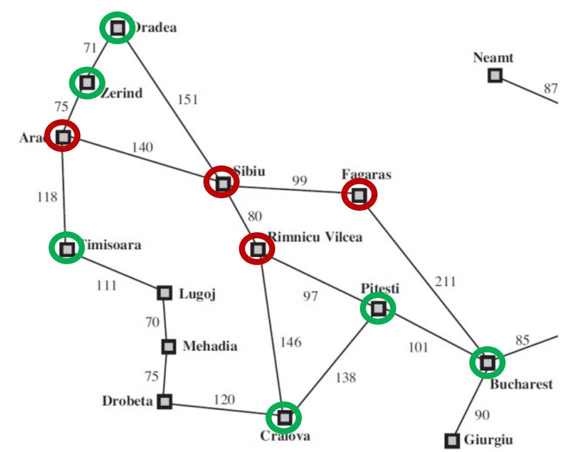




A* Search

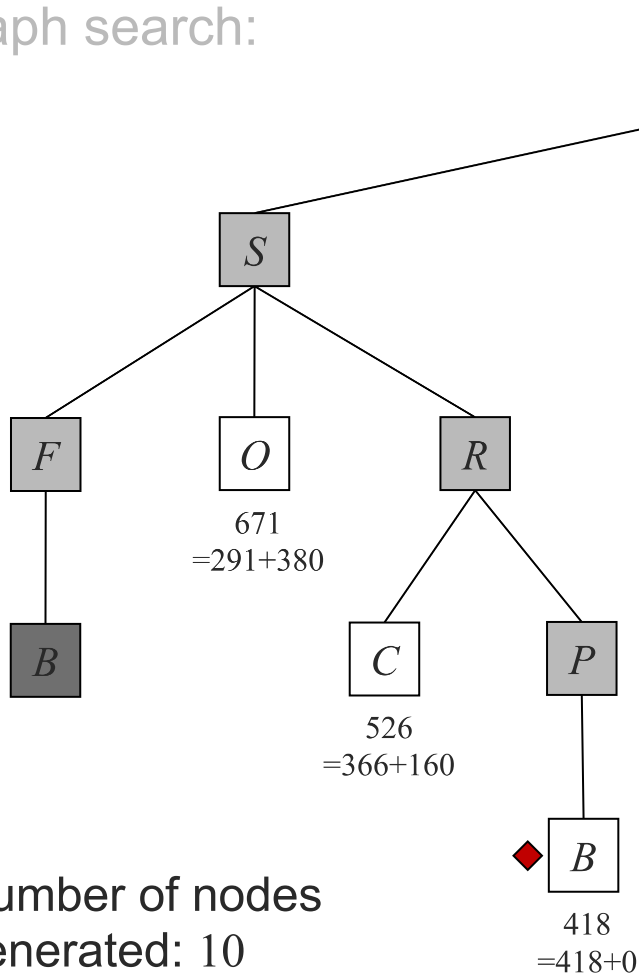
A* graph search:



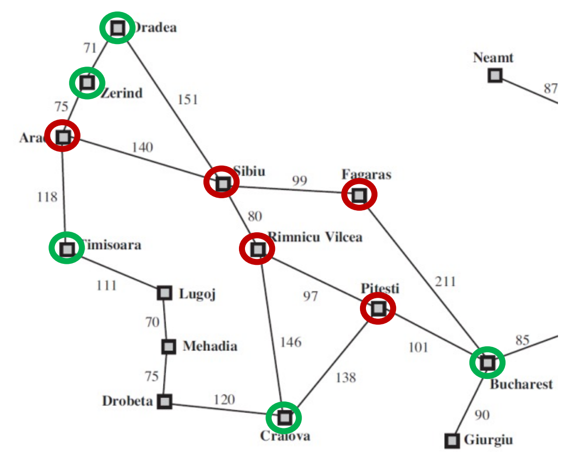


A* Search

A* graph search:



Number of nodes generated: 10



A* Search

Theorem:

The tree-search version of A* is optimal if $h(n)$ never overestimates the cost to reach the goal (i.e., $h(n)$ is admissible).

Proof:

Let C^* be the cost of the optimal solution.

Suppose a suboptimal goal node G_2 appears on the frontier. Then,

$$f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^* \quad (\because\ h(G_2) = 0)$$

But, for a frontier node n that is on an optimal solution path

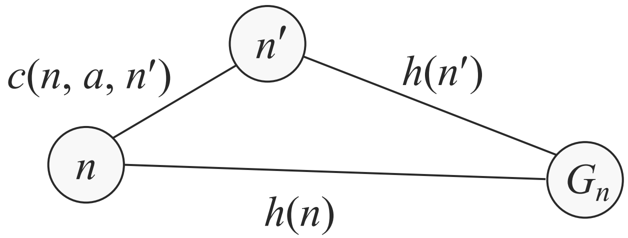
$$f(n) = g(n) + h(n) \leq C^*$$

Since $f(n) < g(G_2)$, G_2 will not be expanded. ■

- $h_{SLD}(n)$ is a good example of an admissible heuristic

A* Search

- A* graph-search is not guaranteed to be optimal even if $h(n)$ is admissible because the frontier may not contain any node on an optimal solution path due to the ignorance of redundant paths
- A stronger condition called consistency (or monotonicity) is required for the optimality of A* graph search
  - A heuristic $h(n)$ is consistent if for every node n and every successor n' generated by an action a, $h(n) \leq c(n, a, n') + h(n')$, where $c(n, a, n')$ is the step cost from n to n' by action a



If $h(n) \leq c(n, a, n') + h(n')$ then $f(n) \leq f(n')$

($f(n)$ along any path are nondecreasing)

  - A consistent heuristic is admissible

A* Search

Theorem:

The graph-search version of A* is optimal if $h(n)$ is consistent.

Proof:

Notice that the values of $f(n)$ along any path are nondecreasing, and that the sequence of nodes expanded by A* graph search is in nondecreasing order of $f(n)$.

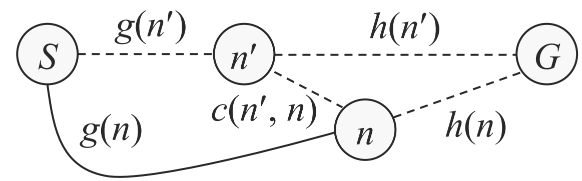
Suppose A* selects a node n for expansion, then the optimal path to that node has been found. Were this not the case, there must be another frontier node n' on the optimal path from the start node to n, i.e., $g(n) \geq g^*(n)$, where $g^*(n)$ is the path cost from the start node to n through n'.

A* Search

Proof:

Then, $f(n) = g(n) + h(n) \geq g^*(n) + h(n) = g(n') + c(n', n) + h(n)$,

where $c(n', n)$ is the step cost from n' to n.



But, $c(n', n) + h(n) \geq h(n')$ because h is consistent.

Therefore, $f(n) \geq g(n') + h(n') = f(n')$, which implies that n' should have been selected first for expansion.

Hence, the first goal node selected for expansion must be an optimal solution because f is the true cost for goal nodes ($h = 0$) and all later goal nodes will be at least as expensive. ■

A* Search

- Evaluation:
  - A* is optimal and complete
  - Time and memory complexity:
    - Exponential time complexity
      - Perfect $h \rightarrow$ no search (practically impossible)
    - Exponential memory complexity
  - A* is optimally efficient for any given heuristics
    - Search is done efficiently by pruning the subtrees below the nodes with $f(n) > C^*$
    - No other optimal algorithm is guaranteed to expand fewer nodes than A*

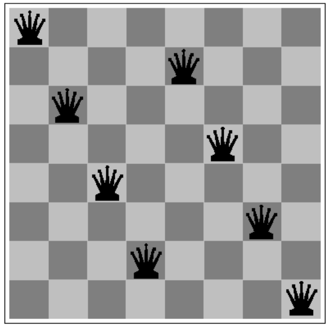
A* Search

- A* in reality:
  - For most real problems, however, the number of nodes expanded is exponential in the length of the solution
  - The use of a good heuristic provides enormous savings compared to the use of an uninformed search

- Recall that $f(n) = g(n) + h(n)$
  - $h = 0 \quad \rightarrow$ uniform cost search
  - $g = 1, h = 0 \rightarrow$ breadth-first search
  - $g = 0 \quad \rightarrow$ greedy best-first search

Local Search Algorithms

- Iterative improvement algorithms:
  - In many optimization problems, path is irrelevant;
    - The goal state itself is the solution (e.g., 8-queens problem)
  - State space = set of "complete" configurations
    - Find optimal configuration according to the objective function (e.g., TSP)
    - Find configuration satisfying constraints (e.g., time table)
  - Start with a complete configuration and make modifications to improve its quality



Local Search Algorithms

Example: TSP

- Current configuration

B	A	C	E	D

- Candidate neighborhood configurations:

A	B	C	E	D

B	C	A	E	D

B	A	E	C	D

B	A	C	D	E

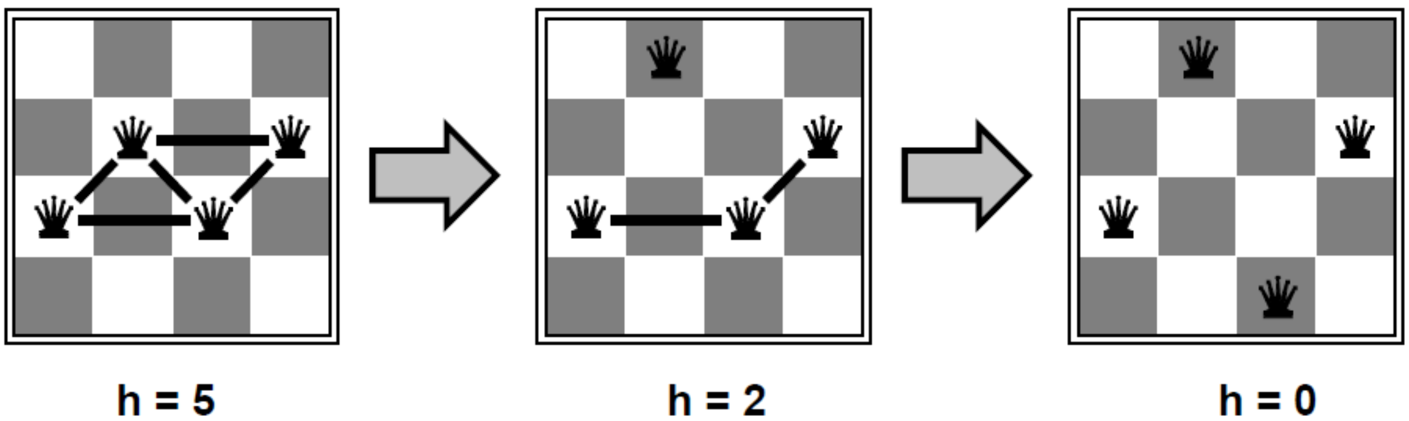
D	A	C	E	B

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

Local Search Algorithms

Example: n-queens

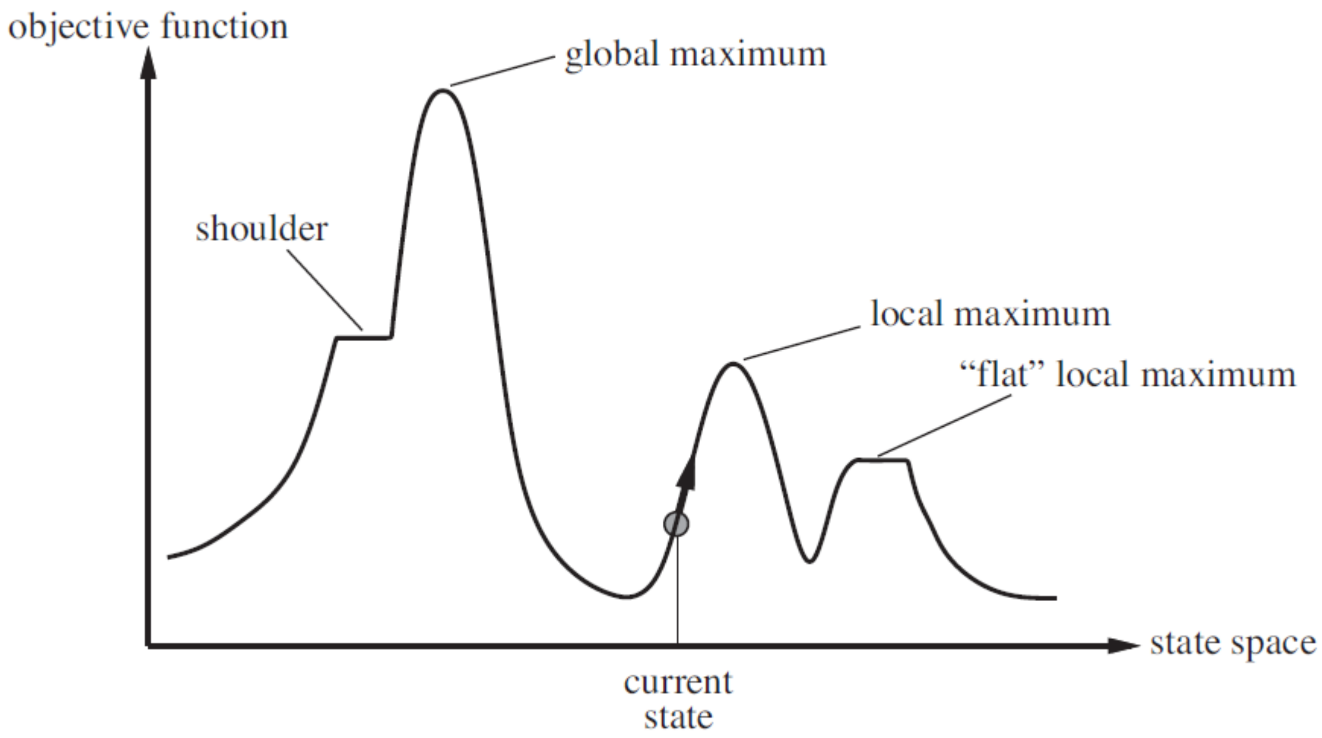
- Move a queen to reduce number of conflicts



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

Local Search Algorithms

- State space landscape
  - Location: state
  - Elevation: heuristic cost function or objective function



Hill-climbing Search

- "Like climbing Everest in thick fog with amnesia"
  - Continually moves in the direction of increasing value
  - Also called gradient ascent/descent search

[Steepest ascent version]

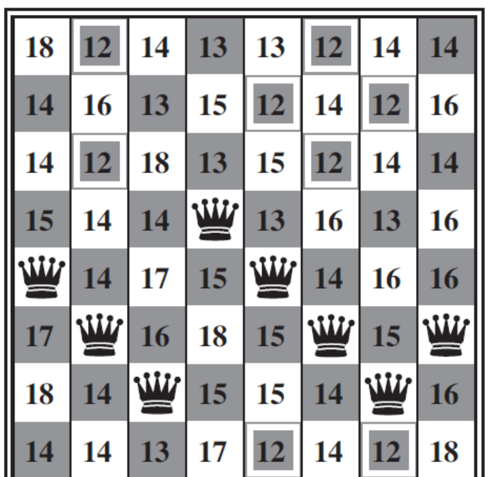
function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

current ← MAKE-NODE(*problem*.INITIAL-STATE)
loop do
  neighbor ← a highest-valued successor of *current*
  if *neighbor*.VALUE ≤ *current*.VALUE **then return** *current*.STATE
  current ← *neighbor*

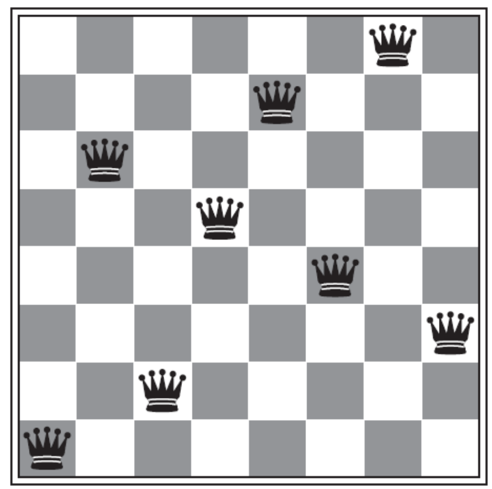
Hill-climbing Search

<u>Example</u>: 8-queens problem

- Each state has 8 queens on the board, one per column
- Successor function generates 56 states by moving a single queen to another square in the same column
- h is the # of pairs that are attacking each other



A state with $h = 17$



A local minimum

Hill-climbing Search

- Drawbacks: often gets stuck to local maxima due to greediness
- Possible solutions:
  - Stochastic hill climbing:
    - Chooses at random from among the uphill moves with probability proportional to steepness
  - First-choice (simple) hill climbing:
    - Generates successors randomly until one is found that is better than the current state
  - Random-restart hill climbing:
    - Conducts a series of hill-climbing searches from randomly generated initial states
    - Very effective for 8-queens

      Can find solutions for 3 million queens in under a minute

Hill-climbing Search

- Complexity:
  - The success of hill climbing depends on the shape of the state-space landscape
  - NP-hard problems typically have an exponential number of local maxima to get stuck on
  - A reasonably good local maximum can often be found after a small number of restarts

Simulated Annealing Search

- Idea:
  - Efficiency of valley-descending + completeness of random walk
  - Escape local minima by allowing some "bad" moves

    But gradually decrease their step size and frequency
- Analogy with annealing
  - At fixed temperature T, state occupation probability reaches Boltzman distribution $p(x) = \alpha e^{-E(x)/kT}$
  - T decreased slowly enough $\rightarrow$ always reach the best state
  - Devised by Metropolis *et al*., 1953, for physical process modeling
  - Widely used in VLSI layout, airline scheduling, etc.

Simulated Annealing Search

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

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

Simulated Annealing Search

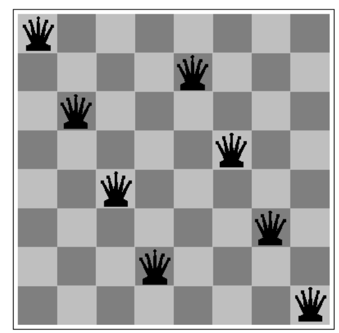
- A random move is picked instead of the best move
  - If the move improves the situation, it is always accepted
  - Otherwise, the move is accepted with probability $e^{-\Delta E/T}$
    - ΔE : the amount by which the evaluation is worsened
      - The acceptance probability decreases exponentially with the "badness" of the move
    - T : temperature, determined by the annealing schedule (controls the randomness)
      - Bad moves are more likely at the start when T is high
      - They become less likely as T decreases
  - $T \to 0$: simple hill-climbing (first-choice hill-climbing)
  - If the annealing schedule lowers T slowly enough, a global optimum will be found with probability approaching 1

Genetic Algorithms

- Starts with a population of individuals
  - Each individual (state) is represented as a string over a finite alphabet—most commonly, a string of 0s and 1s
- Each individual is rated by the fitness function
  - An individual is selected for reproduction by the probability proportional to the fitness score

1	3	5	7	2	4	6	8

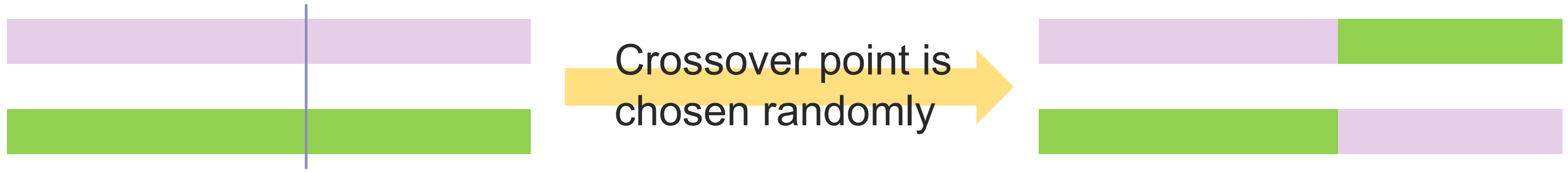
Column-by-column integer representation



Local search algorithms do not use any problem-specific heuristic
Simulated annealing and GA use higher-level heuristic → metaheuristic algorithms

Genetic Algorithms

- Selected pair are mated by a crossover
  - Crossover frequently takes large steps in the state space early in the search process when the population is quite diverse, and smaller steps later on when most individuals are quite similar



- Each locus is subject to random mutation with a small independent probability
- Advantage of GA comes from crossover:
  - Is able to combine large blocks of letters that have evolved independently to perform useful functions
  - ➡ Raises the level of granularity at which the search operates

Genetic Algorithms

1) **Chromosome design**
2) initialization
3) Fitness evaluation
4) Selection
5) Crossover
6) Mutation
7) Update generation
8) Go back to 3)

1) Chromosome design

0	1	1	1	0	0	1	0	0	1	0	0	1

Genetic Algorithms

1) Chromosome design
2) Initialization
3) Fitness evaluation
4) Selection
5) Crossover
6) Mutation
7) Update generation
8) Go back to 3)

2) Initialization

1	0	0	1	1	1	1	0	1	0	1	0	0	7
1	1	1	1	1	0	0	0	0	1	0	1	0	7
0	1	0	0	0	0	1	0	1	1	0	0	1	5
0	0	1	0	1	0	1	1	0	1	1	0	1	7
0	1	1	1	0	1	1	0	0	1	0	0	1	7
1	0	1	0	0	0	0	1	0	0	0	1	0	4
0	1	0	1	0	0	1	1	0	0	1	1	1	7
1	1	1	1	0	0	0	0	0	1	1	0	0	6

3) Fitness evalution

Genetic Algorithms

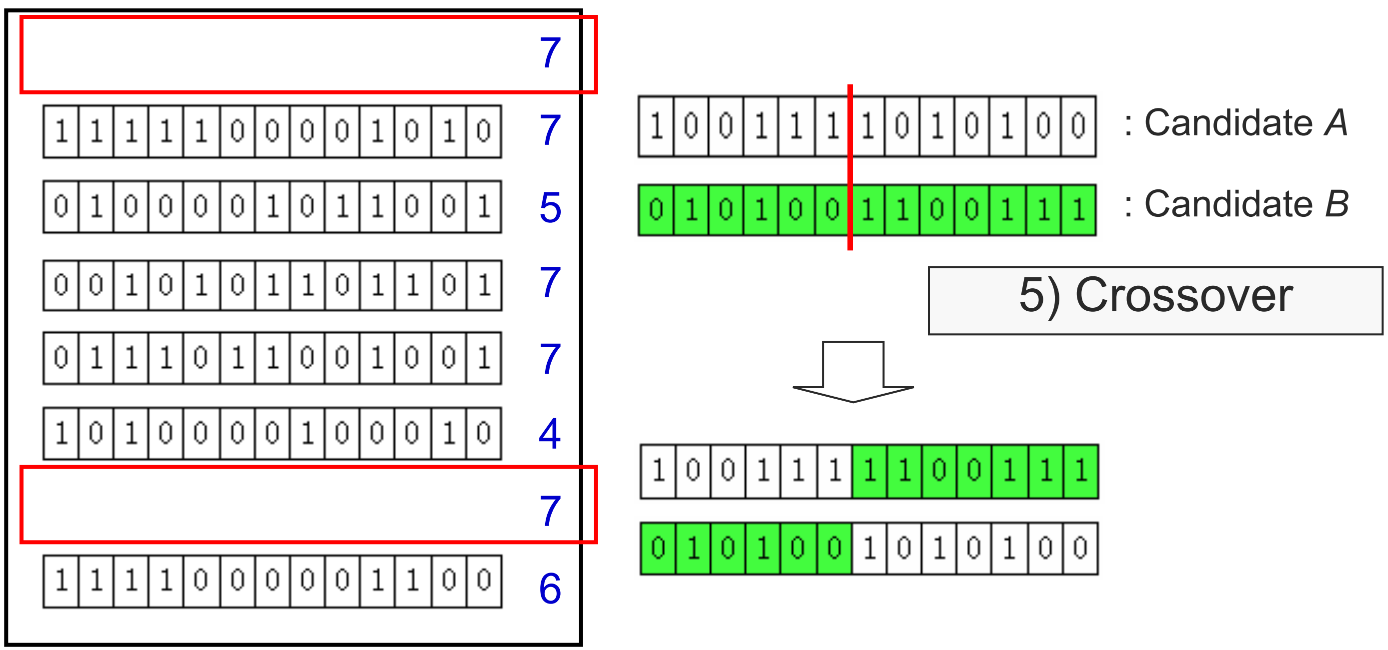
1) Chromosome design
2) Initialization
3) Fitness evaluation
4) **Selection**
5) Crossover
6) Mutation
7) Update generation
8) Go back to 3)

4) Selection

1	0	0	1	1	1	1	0	1	0	1	0	0	7
1	1	1	1	1	0	0	0	0	1	0	1	0	7
0	1	0	0	0	0	1	0	1	1	0	0	1	5
0	0	1	0	1	0	1	1	0	1	1	0	1	7
0	1	1	1	0	1	1	0	0	1	0	0	1	7
1	0	1	0	0	0	0	1	0	0	0	1	0	4
0	1	0	1	0	0	1	1	0	0	1	1	1	7
1	1	1	1	0	0	0	0	0	1	1	0	0	6

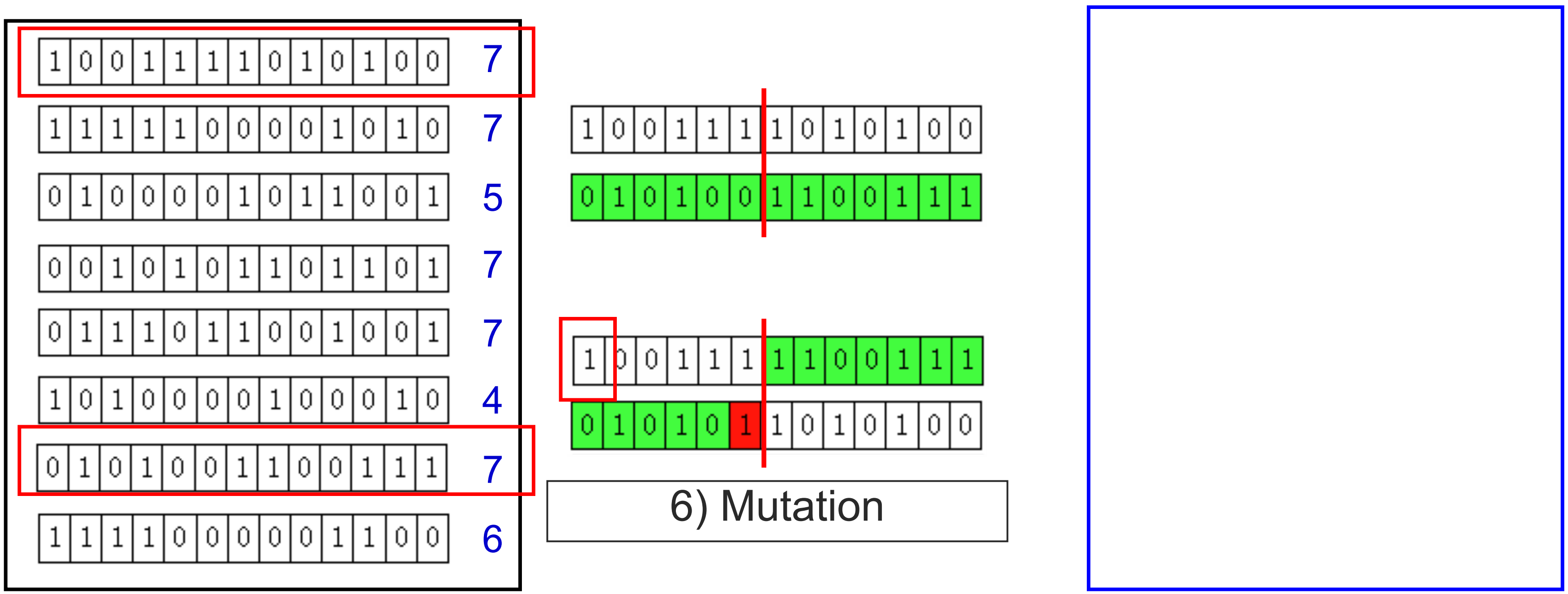
Genetic Algorithms

1) Chromosome design
2) Initialization
3) Fitness evaluation
4) Selection
5) **Crossover**
6) Mutation
7) Update generation
8) Go back to 3)



Genetic Algorithms

1) Chromosome design
2) Initialization
3) Fitness evaluation
4) Selection
5) Crossover
6) **Mutation**
7) Update generation
8) Go back to 3)



Genetic Algorithms

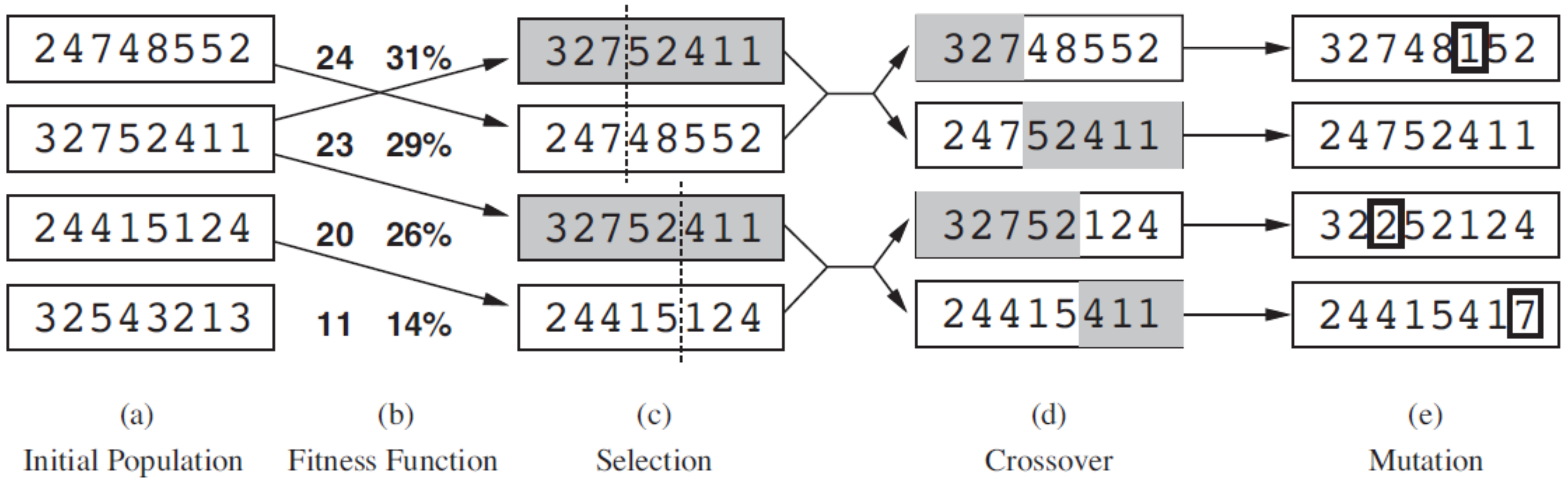
1) Chromosome design
2) Initialization
3) Fitness evaluation
4) Selection
5) Crossover
6) Mutation
7) **Update generation**
8) **Go back to 3)**

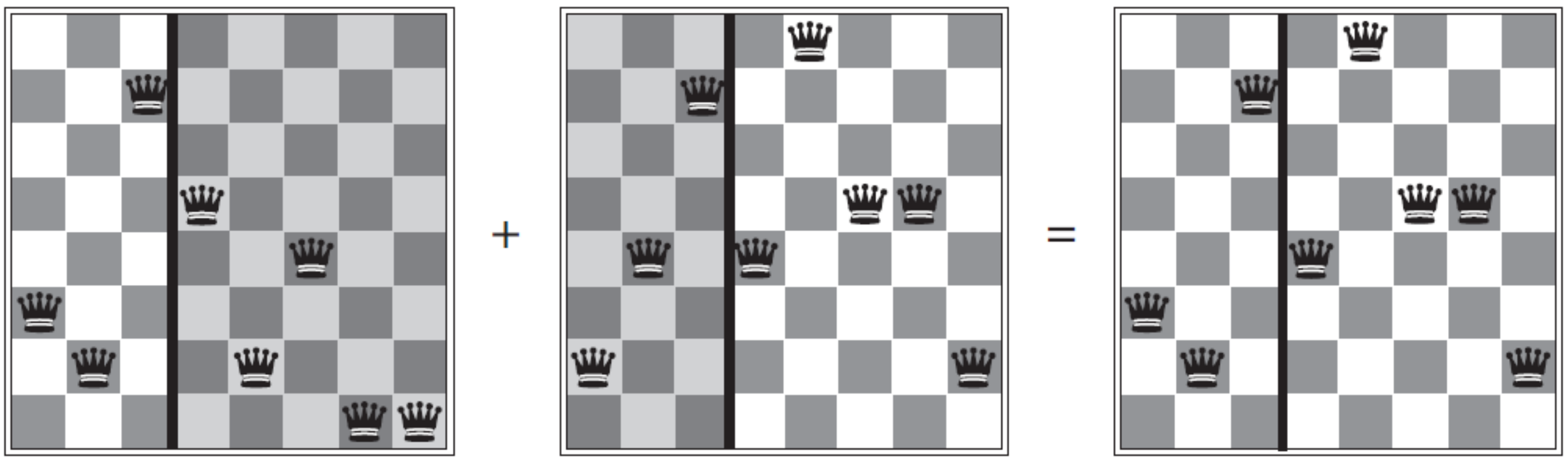
7) Update generation

1	0	0	1	1	1	1	0	1	0	1	0	0	7
1	1	1	1	1	0	0	0	0	1	0	1	0	7
0	1	0	0	0	0	1	0	1	1	0	0	1	5
0	0	1	0	1	0	1	1	0	1	1	0	1	7
0	1	1	1	0	1	1	0	0	1	0	0	1	7
1	0	1	0	0	0	0	1	0	0	0	1	0	4
0	1	0	1	0	0	1	1	0	0	1	1	1	7
1	1	1	1	0	0	0	0	0	1	1	0	0	6

1	0	0	1	1	1	1	1	0	0	1	1	1
0	1	0	1	0	1	1	0	1	0	1	0	0
1	1	1	1	0	0	0	0	0	1	0	0	1
0	1	0	0	0	0	1	0	1	1	1	0	0
1	0	0	1	0	0	1	1	0	0	1	1	1
0	1	0	1	1	1	1	0	1	0	1	0	0
1	1	1	1	1	0	0	1	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	1	0

Genetic Algorithms





Genetic Algorithms

```
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
```

Genetic Algorithms

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

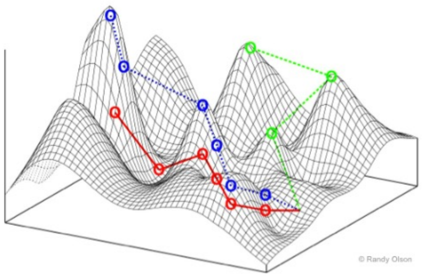
  $n \leftarrow$ LENGTH(x)
  $c \leftarrow$ random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c+1, n))

Continuous State Spaces

- Gradient methods attempt to use the gradient of the landscape to maximize/minimize f by

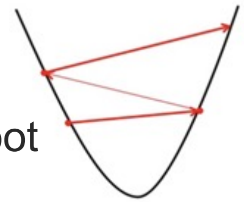
  $$x \leftarrow x +/- \alpha \nabla f(x) \quad (\alpha: \text{step size})$$

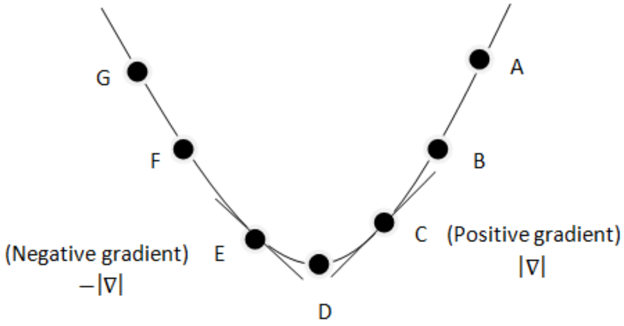
  where $\nabla f(x)$ is the gradient vector (containing all of the partial derivatives) of f that gives the magnitude and direction of the steepest slope



  - Too small α: too many steps are needed
  - Too large α: the search could overshoot the target
  - Points where $\nabla f(x) = 0$ are known as critical points

Overshoot





Continuous State Spaces

Example: Gradient descent

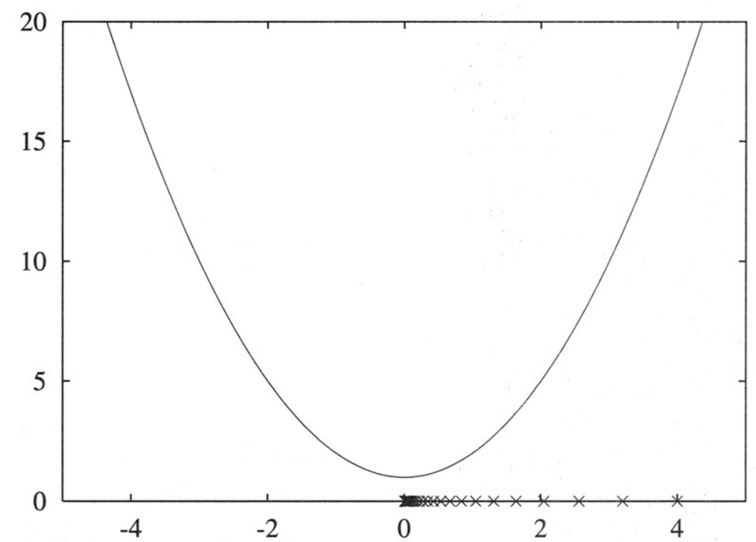
$$w \leftarrow w - \alpha f'(w)$$

- If $f(w) = w^2 + 1$, then $f'(w) = 2w$

  Starting from an initial value $w = 4$, with the step size of 0.1:
  - $4 - (0.1 \times 2 \times 4) = 3.2$
  - $3.2 - (0.1 \times 2 \times 3.2) = 2.56$
  - $2.56 - (0.1 \times 2 \times 2.56) = 2.048$

    . . . .

- Stops when the change in parameter value becomes too small



Continuous State Spaces

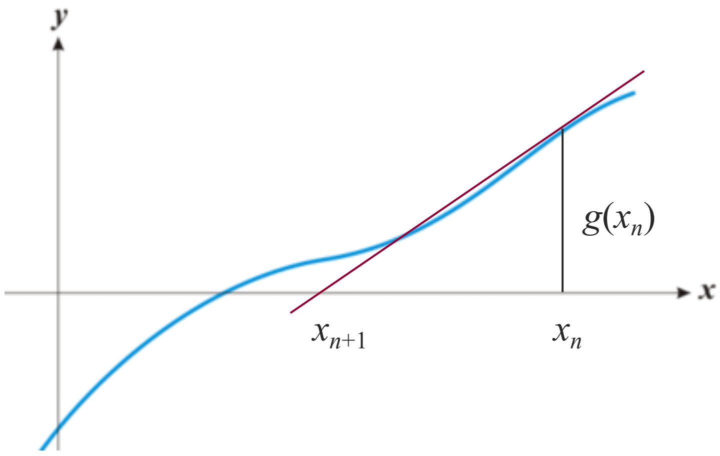
- Line search (for minimization): adapts the size of α
  - Evaluate $f(\boldsymbol{x} - \alpha \nabla f(\boldsymbol{x}))$ for several values of α and choose the one that results in the smallest objective function value
  - New direction of search should be chosen from that point

- In some cases, we may be able to jump directly to the critical point by solving the equation $\nabla f(\boldsymbol{x}) = 0$ for $\boldsymbol{x}$

(In many cases, $\nabla f(\boldsymbol{x}) = 0$ cannot be solved in closed form)

Continuous State Spaces

- **Newton-Raphson method** (1664, 1690) for finding roots of $g(x)$:
  - Finds successively better approximations to the roots
  - From some current approximation x_n, we can find a better approximation x_{n+1} by computing the x-intercept of the tangent line at $(x_n, g(x_n))$



  - Observe that $g'(x_n) = \dfrac{g(x_n)}{x_n - x_{n+1}}$

    Solving for x_{n+1} gives $x_{n+1} = x_n - \dfrac{g(x_n)}{g'(x_n)}$

Continuous State Spaces

- Newton-Raphson method:
  - To find a minimum of f we need to solve $\nabla f(\boldsymbol{x}) = 0$
  - $g(x)$ and $g'(x)$ become $\nabla f(\boldsymbol{x})$ and $\mathbf{H}_f(\boldsymbol{x})$, respectively, where $\mathbf{H}_f(\boldsymbol{x})$ is the Hessian matrix of second derivatives with elements $H_{ij} = \partial^2 f / \partial x_i \partial x_j$
  - The update equation to solve $\nabla f(\boldsymbol{x}) = 0$ can be written in matrix-vector form as

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \mathbf{H}_f^{-1}(\boldsymbol{x}) \nabla f(\boldsymbol{x})$$

$$x_{n+1} = x_n - \frac{g(x_n)}{g'(x_n)}$$

  - Newton-Raphson becomes expensive in high-dimensional spaces (need approximations)
    - Hessian has n^2 entries, needs to be inverted
    - Approximate versions of Newton-Raphson method can be used

Constrained Optimization Problem

- A constrained optimization problem is formulated as

$$\min f(\boldsymbol{x}), \ \boldsymbol{x} \in \mathbf{R}^n$$

subject to
$$g_i(\boldsymbol{x}) = 0, \ i = 1, \ldots, q$$
$$h_j(\boldsymbol{x}) \le 0, \ j = q+1, \ldots, k$$
$$L_l \le x_l \le U_l, \ l = 1, \ldots, n$$

where

- L_l and U_l are the the lower and upper bounds of x_l, respectively, which define the search space S
- The q equality constraints and $k - q$ inequality constraints define the feasible region $F \subseteq S$

Penalty Method

- The extent of the violation of constraint j can be measured as

$$v_j(\boldsymbol{x}) = \begin{cases} |g_j(\boldsymbol{x})| & 1 \le j \le q \\ \max\{0, h_j(\boldsymbol{x})\} & q+1 \le j \le k \end{cases}$$

- By assigning weights, or penalty coefficients w_j, to each constraint violation to represent the importance or to adjust the scaling, the penalty function is formulated as

$$penalty(\boldsymbol{x}) = \sum_{j=1}^{k} w_j v_j(\boldsymbol{x})$$

- Then the objective value of a solution $\boldsymbol{x}$ can be represented by

$$f'(\boldsymbol{x}) = f(\boldsymbol{x}) + penalty(\boldsymbol{x})$$

  - This formulation converts a COP into an unconstrained optimization problem
    (but, does not guarantee feasibility)

Karush-Kuhn-Tucker (KKT) Approach

- A constrained optimization problem $\min f(\boldsymbol{x})$ subject to $g(\boldsymbol{x}) = 0$ and $h(\boldsymbol{x}) \leq 0$ can be converted to an unconstrained optimization problem by introducing a function called generalized Lagrange function (or generalized Lagrangian*):

$$L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\boldsymbol{x}) + \sum_i \lambda_i g_i(\boldsymbol{x}) + \sum_j \alpha_j h_j(\boldsymbol{x})$$

where λ_i and α_j are the Lagrange multipliers (or KKT multipliers)

* Allowing inequality constraints, the KKT approach generalizes the method of Lagrange multipliers, which allows only equality constraints

- The generalized Lagrangian should be minimized with respect to $\boldsymbol{x}$ and maximized with respect to $\boldsymbol{\lambda}$ and $\boldsymbol{\alpha} \geq 0$ (to maximally penalize constraint violations), i.e.,

$$\min_{\boldsymbol{x} \in F} f(\boldsymbol{x}) = \min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}, \boldsymbol{\alpha} \geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$$

While LHS has concern about F, RHS does not

where F is the feasible region

- Note that $\min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda},\, \boldsymbol{\alpha} \geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$ has the same optimal objective function value and set of optimal points $\boldsymbol{x}^*$ as $\min_{\boldsymbol{x} \in F} f(\boldsymbol{x})$ because whenever the constraints are satisfied,

$$\max_{\boldsymbol{\lambda},\, \boldsymbol{\alpha} \geq 0} L(\boldsymbol{x}^*, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\boldsymbol{x}^*)$$

$\because$ (1) $g_i(\boldsymbol{x}^*) = 0$
(2) $h_j(\boldsymbol{x}^*) = 0$ or $\alpha_j = 0$ maximizes $\alpha_j h_j(\boldsymbol{x}^*)$ when $h_j(\boldsymbol{x}^*) < 0$

while any time a constraint is violated,

$$\max_{\boldsymbol{\lambda},\, \boldsymbol{\alpha} \geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = \infty$$

  - No infeasible point can be optimal
  - The optimum within the feasible region is unchanged

- Since $\min \max E \geq \max \min E$ for any E, the following inequality holds

$$
\begin{aligned}
&\forall \boldsymbol{x}\ \forall \boldsymbol{y}\ \varphi(\boldsymbol{x}, \boldsymbol{y}) \geq \min_{\boldsymbol{x}} \varphi(\boldsymbol{x}, \boldsymbol{y}) \\
&\Rightarrow \forall \boldsymbol{x} \max_{\boldsymbol{y}} \varphi(\boldsymbol{x}, \boldsymbol{y}) \geq \max_{\boldsymbol{y}} \min_{\boldsymbol{x}} \varphi(\boldsymbol{x}, \boldsymbol{y}) \\
&\Rightarrow \min_{\boldsymbol{x}} \max_{\boldsymbol{y}} \varphi(\boldsymbol{x}, \boldsymbol{y}) \geq \max_{\boldsymbol{y}} \min_{\boldsymbol{x}} \varphi(\boldsymbol{x}, \boldsymbol{y})
\end{aligned}
$$

$$
\begin{aligned}
\min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}, \boldsymbol{\alpha} \geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) &\geq \max_{\boldsymbol{\lambda}, \boldsymbol{\alpha} \geq 0} \min_{\boldsymbol{x}} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) \\
&= \max_{\boldsymbol{\lambda}, \boldsymbol{\alpha} \geq 0} L_d(\boldsymbol{\lambda}, \boldsymbol{\alpha})
\end{aligned}
$$

  where $L_d(\boldsymbol{\lambda}, \boldsymbol{\alpha}) = \min_{\boldsymbol{x}} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$ is called the dual function

- What we have now is therefore,

$$
\min_{\boldsymbol{x} \in F} f(\boldsymbol{x}) \geq \max_{\boldsymbol{\lambda}, \boldsymbol{\alpha} \geq 0} L_d(\boldsymbol{\lambda}, \boldsymbol{\alpha})
$$

  - Instead of directly finding $\boldsymbol{x}$ that satisfies $\min_{\boldsymbol{x} \in F} f(\boldsymbol{x})$, it is often easier to find $\boldsymbol{\lambda}$ and $\boldsymbol{\alpha}$ satisfying $\max_{\boldsymbol{\lambda}, \boldsymbol{\alpha} \geq 0} L_d(\boldsymbol{\lambda}, \boldsymbol{\alpha})$ and then use them to find the values for $\boldsymbol{x}$

- The inequality constraint $h_j(\boldsymbol{x})$ is said to be active if $h_j(\boldsymbol{x}^*) = 0$, where $\boldsymbol{x}^*$ is an optimal solution

$$L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\boldsymbol{x}) + \sum_i \lambda_i g_i(\boldsymbol{x}) + \sum_j \alpha_j h_j(\boldsymbol{x})$$

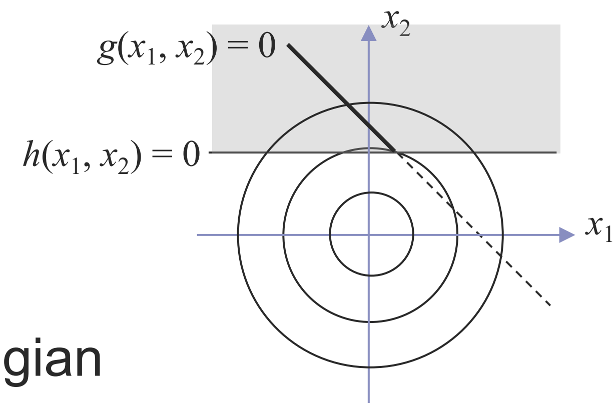
  - The solution is on the boundary imposed by the inequality
  - α_j for an active constraint h_j has a positive value
  - We must use $\boldsymbol{\alpha}$ to maximize $L_d(\boldsymbol{\lambda}, \boldsymbol{\alpha})$ (i.e, solve $\max_{\boldsymbol{\lambda},\, \boldsymbol{\alpha} \geq 0} L_d(\boldsymbol{\lambda}, \boldsymbol{\alpha})$) when $h_j(\boldsymbol{x})$ is active

Example: Generalized Lagrangian with an active constraint

$$\min f(x_1,x_2)=\frac{1}{2}(x_1^2+x_2^2)$$

subject to $g(x_1, x_2) = 1 - x_1 - x_2 = 0$

$$h(x_1,x_2)=\frac{3}{4}-x_2 \le 0$$



- We can solve the generalized Lagrangian

$$L(x_1,x_2,\lambda,\alpha)=\frac{1}{2}(x_1^2+x_2^2)+\lambda(1-x_1-x_2)+\alpha(\frac{3}{4}-x_2)$$

whose dual function is

$$L_d(\lambda,\alpha)=\min_{x_1,x_2} L(x_1,x_2,\lambda,\alpha)$$

- From $\dfrac{\partial L(x_1,x_2,\lambda,\alpha)}{\partial x_1}=x_1-\lambda=0$ we get $x_1=\lambda$

Example: Generalized Lagrangian with an active constraint

- From $\frac{\partial L(x_1, x_2, \lambda, \alpha)}{\partial x_2} = x_2 - \lambda - \alpha = 0$ we get $x_2 = \lambda + \alpha$
- Substituting back to $L(x_1, x_2, \lambda, \alpha)$, we obtain the dual

$$L_d(\lambda, \alpha) = -\lambda^2 - \frac{1}{2}\alpha^2 - \lambda\alpha + \lambda + \frac{3}{4}\alpha$$

- We now solve the dual problem $\max_{\lambda,\alpha} L_d(\lambda, \alpha)$

$$\begin{aligned} \frac{\partial L_d(\lambda, \alpha)}{\partial \lambda} &= -2\lambda - \alpha + 1 = 0 \\ \frac{\partial L_d(\lambda, \alpha)}{\partial \alpha} &= -\alpha - \lambda + \frac{3}{4} = 0 \end{aligned} \quad \Rightarrow \quad \lambda = \frac{1}{4}, \quad \alpha = \frac{1}{2}$$

- Therefore, $x_1 = \lambda = \frac{1}{4}$ and $x_2 = \lambda + \alpha = \frac{3}{4}$
- Gradient ascent search needed when not solvable analytically