Artificial Intelligence ENCS 434

Informed Search

Aziz M. Qaroush - Birzeit University

Overview

- Informed Search
  - best-first search
  - search with heuristics
  - memory-bounded search
  - iterative improvement search
  - local search and optimization

Improving Search Methods

- make algorithms more efficient
  - avoiding repeated states
  - utilizing memory efficiently
- use additional knowledge about the problem
  - properties (“shape”) of the search space
    - more interesting areas are investigated first
  - pruning of irrelevant areas
    - areas that are guaranteed not to contain a solution can be discarded

Informed Search

- relies on additional knowledge about the problem or domain
  - frequently expressed through heuristics ("rules of thumb")
  - A Heuristic is a function that, when applied to a state, returns a number that tells us approximately how far the state is from the goal state.
- used to distinguish more promising paths towards a goal
  - may be mislead, depending on the quality of the heuristic
- in general, performs much better than uninformed search
  - but frequently still exponential in time and space for realistic problems

Heuristic Functions

- A heuristic function is a function *f(n)* that gives an estimation on the "cost" of getting from node *n* to the goal state – so that the node with the least cost among all possible choices can be selected for expansion first.

  - A heuristic *h(n)* is admissible if for every node *n*,

    $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from *n*.

  - An admissible heuristic never overestimates the cost to reach the goal, i.e., will make the algorithm optimal.

- Three approaches to defining *f*:
  - *f* measures the value of the current state (its "goodness")
  - *f* measures the estimated cost of getting to the goal from the current state:
    - $f(n) = h(n)$ where $h(n)$ = an estimate of the cost to get from *n* to a goal
  - *f* measures the estimated cost of getting to the goal state from the *current state* and the cost of the existing path to it. Often, in this case, we decompose *f*:
    - $f(n) = g(n) + h(n)$ where $g(n)$ = the cost to get to *n* (from initial state)

Approach 1: *f* Measures the Value of the Current State

- Usually the case when solving optimization problems
  - Finding a state such that the value of the metric *f* is optimized

- Often, in these cases, *f* could be a weighted sum of a set of component values:

  - N-Queens
    - Example: the number of queens under attack …

Approach 2: f Measures the Cost to the Goal

A state X would be better than a state Y if the estimated cost of getting from X to the goal is lower than that of Y – because X would be closer to the goal than Y

- 8–Puzzle

h_1: The number of misplaced tiles (squares with number).

h_2: The sum of the distances of the tiles from their goal positions.

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

Approach 3: *f* measures the total cost of the solution path (Admissible Heuristic Functions)

- A heuristic function $f(n) = g(n) + h(n)$ is admissible if $h(n)$ **never** overestimates the cost to reach the goal.
  - Admissible heuristics are "optimistic": "the cost is not that much …"

- However, $g(n)$ is the exact cost to reach node n from the initial state.

- Therefore, $f(n)$ never over-estimate the true cost to reach the goal state through node n.

- Theorem: A search is optimal if $h(n)$ is admissible.
  - I.e. The search using $h(n)$ returns an optimal solution.

- Given $h_2(n) > h_1(n)$ for all n, it's always more <u>efficient</u> to use $h_2(n)$.
  - h_2 is more realistic than h_1 *(more informed)*, though both are optimistic.

Aziz M. Qaroush - Birzeit University

Best-First Search

- **General approach of informed search:** It is an improved combination of breadth-first and depth-first algorithms.
  - **Best-first search:** node is selected for expansion based on an *evaluation function f(n) that* measures distance to the goal.
- Chooses the best node from the queue to continue the search regardless of the node's actual position in the problem graph.

1. The main steps of this algorithm are as follow:

   1. Add the initial node (starting point) to the queue
   2. Compare the front node to the goal state. If they match then the solution is found.
   3. If they do not match then expand the front node by adding all the nodes from its links to a queue.
   4. If all nodes in the queue are expanded then the goal state is not found (e.g. there is no solution). Stop.
   5. Apply the heuristic function to evaluate and reorder the nodes in the queue.
   6. Go to step 2

- Special cases:
  - Greedy best first search
  - A* search

Traditional informed search strategies

- Greedy Best first search
  - "Always chooses the successor node with the best *f* value" where $f(n) = h(n)$
  - We choose the one that is nearest to the final state among all possible choices

- A* search
  - Best first search using an "admissible" heuristic function *f* that takes into account the current cost *g*
  - Always returns the optimal solution path

Greedy Best-First Search

- minimizes the estimated cost to a goal
  - expand the node that seems to be closest to a goal
  - utilizes a heuristic function as evaluation function
    - *f(n) = h(n)* = estimated cost from the current node to a goal
    - heuristic functions are problem-specific
    - often straight-line distance for route-finding and similar problems
  - often better than depth-first, although worst-time complexities are equal or worse (space)

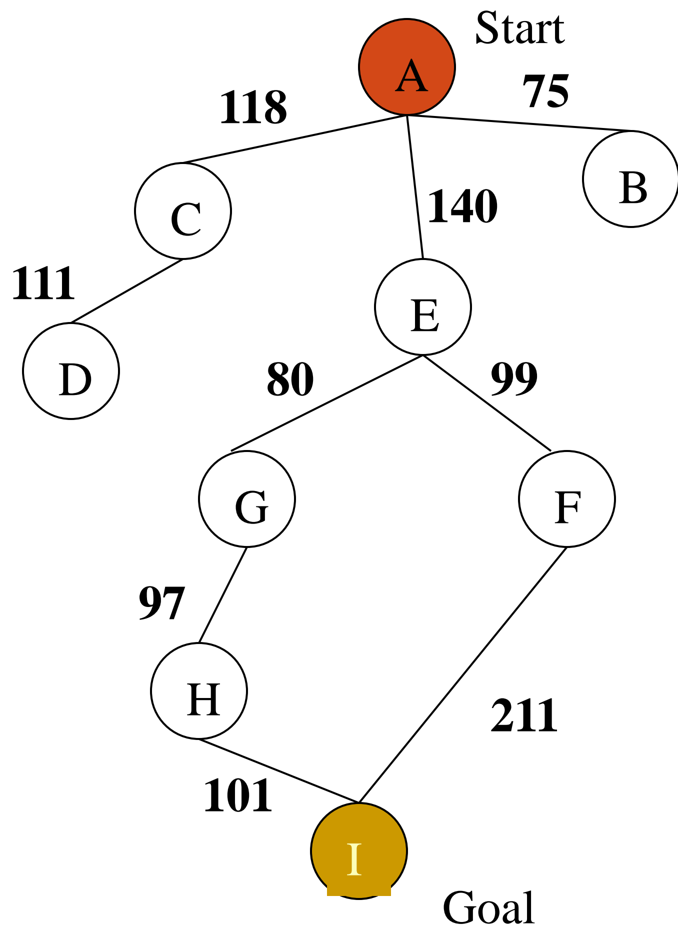
```
function GREEDY-SEARCH(problem) returns solution

return BEST-FIRST-SEARCH(problem, h)
```

Completeness	Time Complexity	Space Complexity	Optimality
no	b^m	b^m	no

b: branching factor, d: depth of the solution, m: maximum depth of the search tree, l: depth limit

Greedy Search

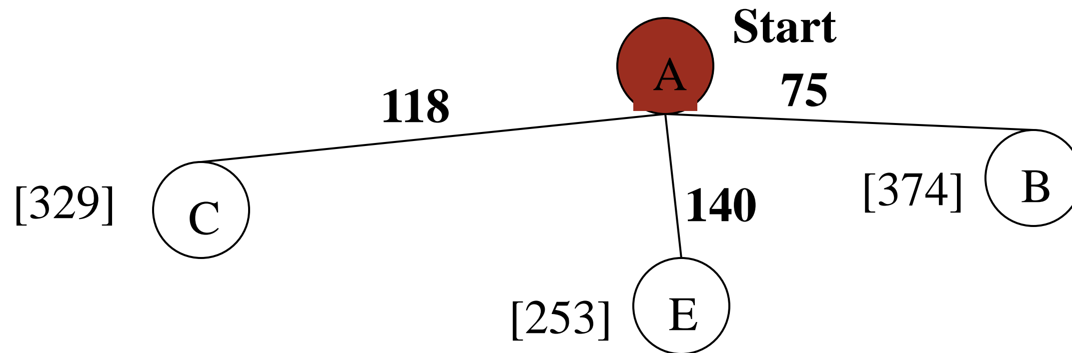


State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

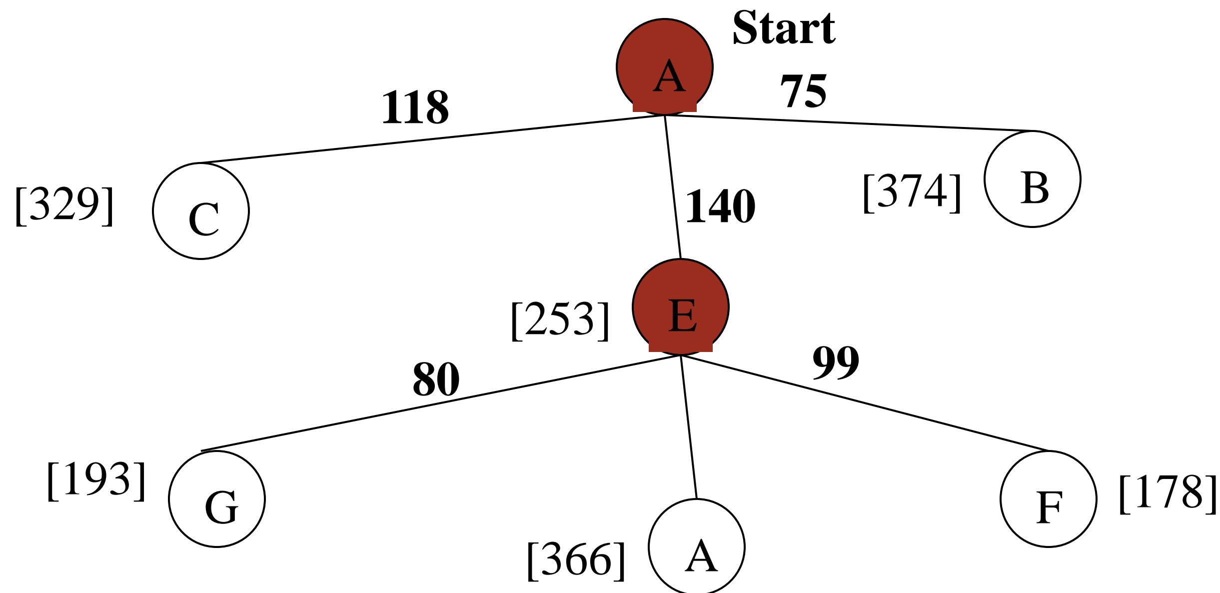
$f(n) = h(n)$ = straight-line distance heuristic

Greedy Search: Tree Search

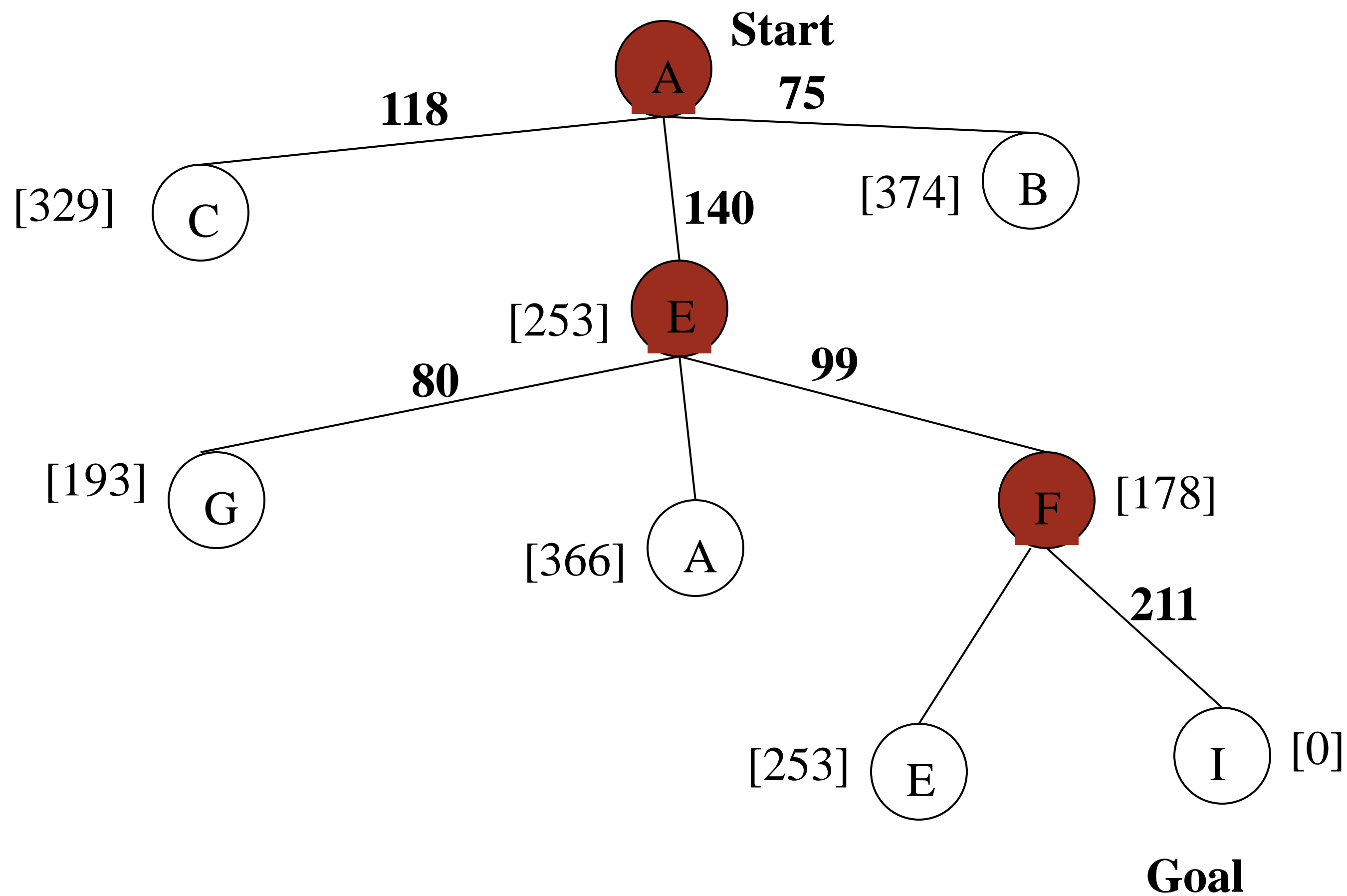
Greedy Search: Tree Search



Greedy Search: Tree Search

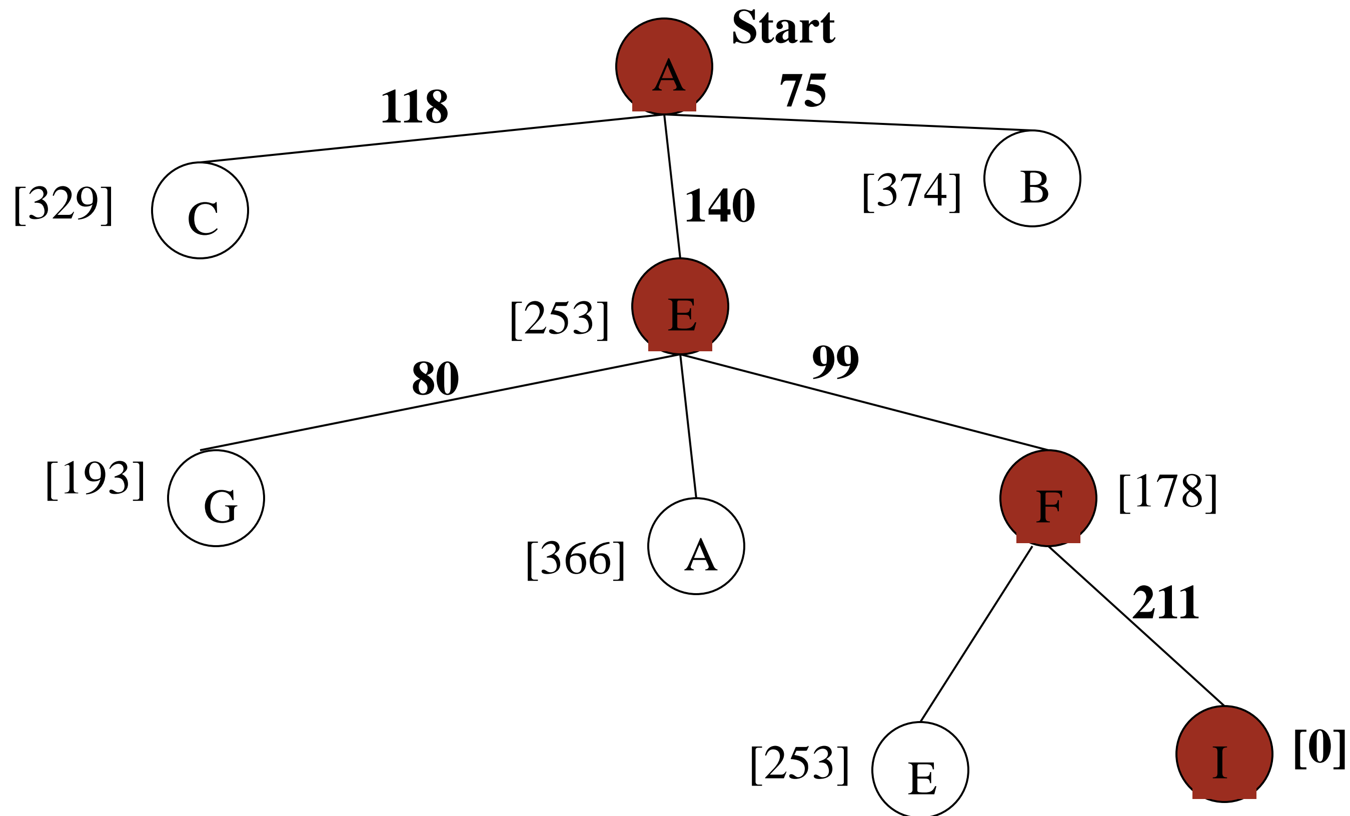


Greedy Search: Tree Search



Aziz M. Qaroush - Birzeit University

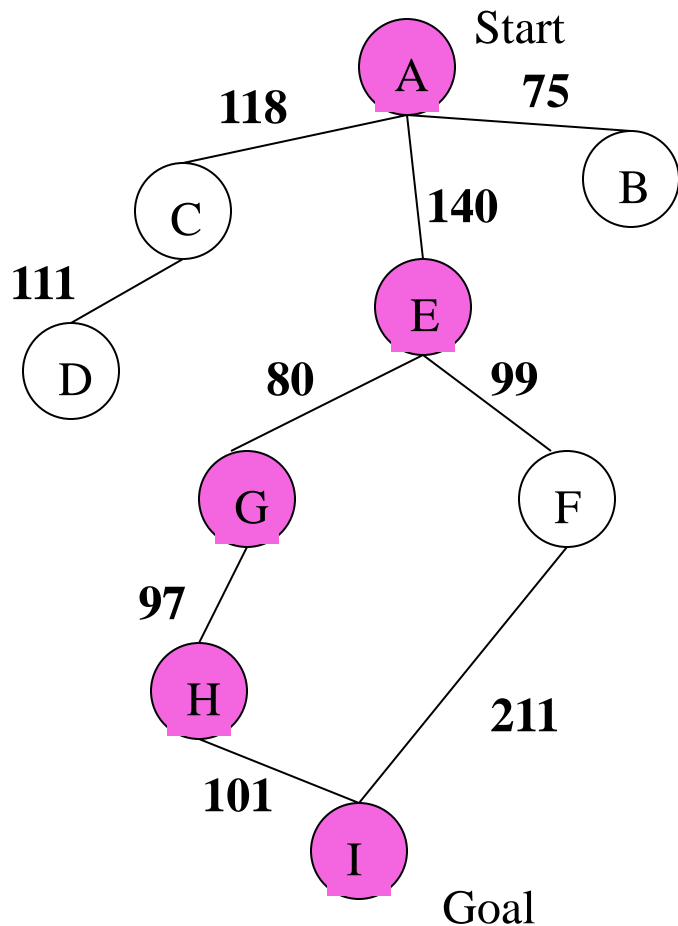
Greedy Search: Tree Search



Path cost(A-E-F-I) = 253 + 178 + 0 = 431

dist(A-E-F-I) = 140 + 99 + 211 = 450

Greedy Search: Optimal ?

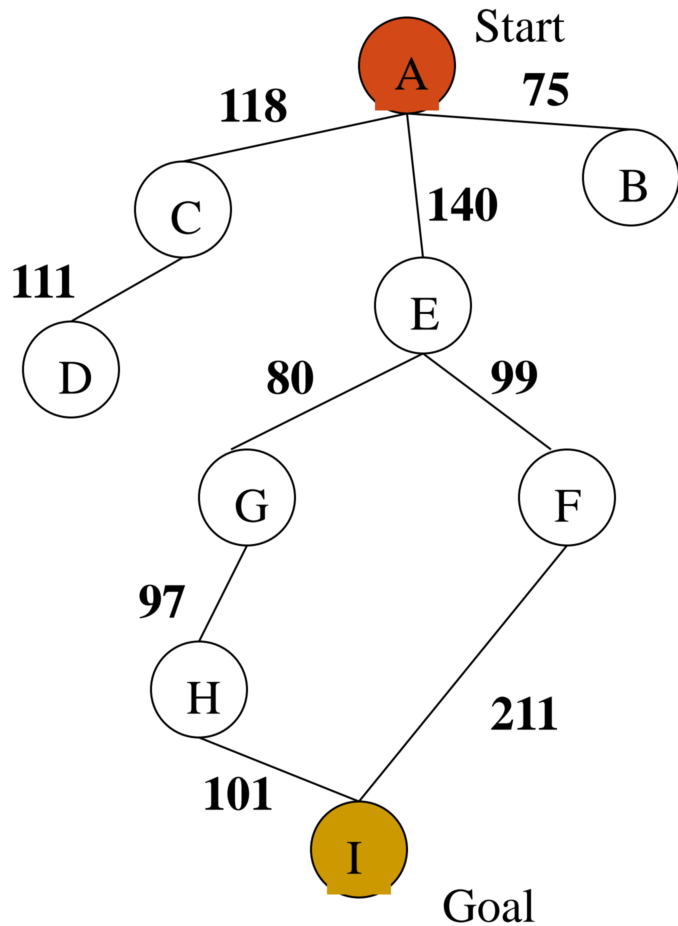


State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	**253**
F	178
G	**193**
H	**98**
I	0

$f(n) = h(n)$ = straight-line distance heuristic

dist(A-E-G-H-I) =140+80+97+101= 418

Greedy Search: Complete ?

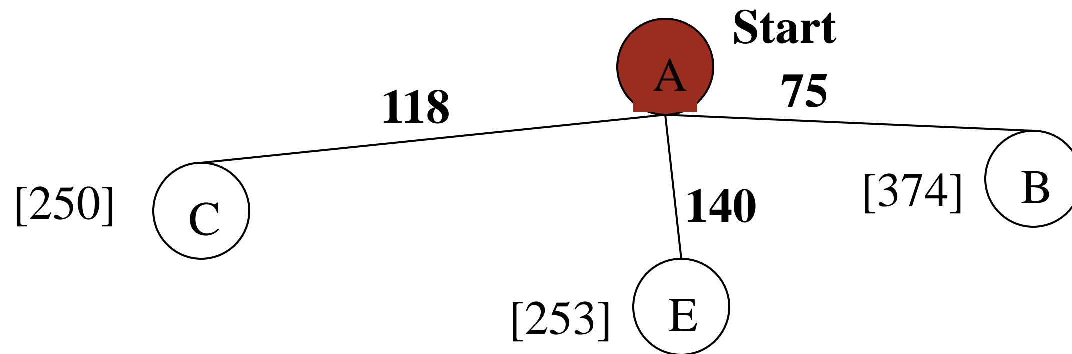


State	Heuristic: h(n)
A	366
B	374
** C	**250**
D	244
E	253
F	178
G	193
H	98
I	0

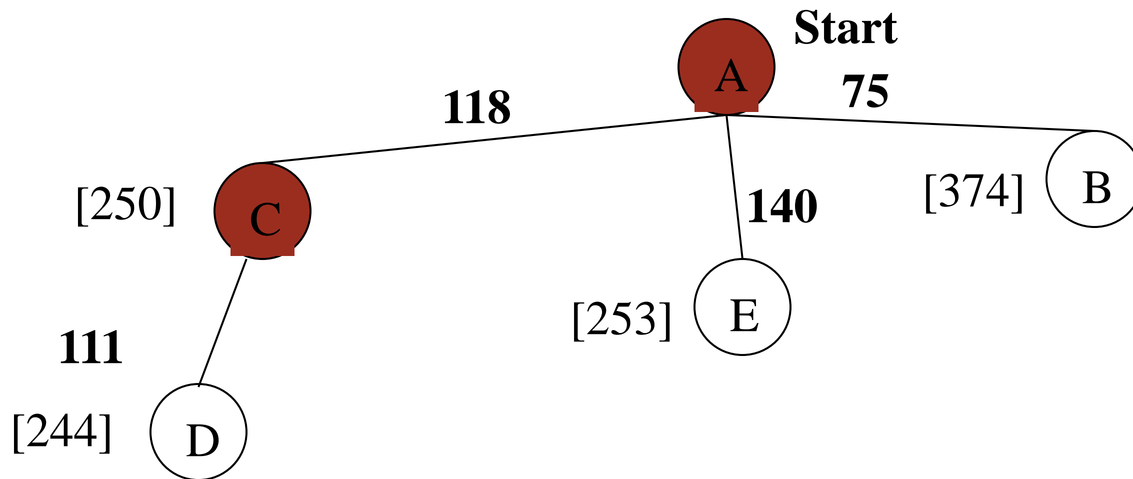
***f(n)* = *h* (*n*) = straight-line distance heuristic**

Greedy Search: Tree Search

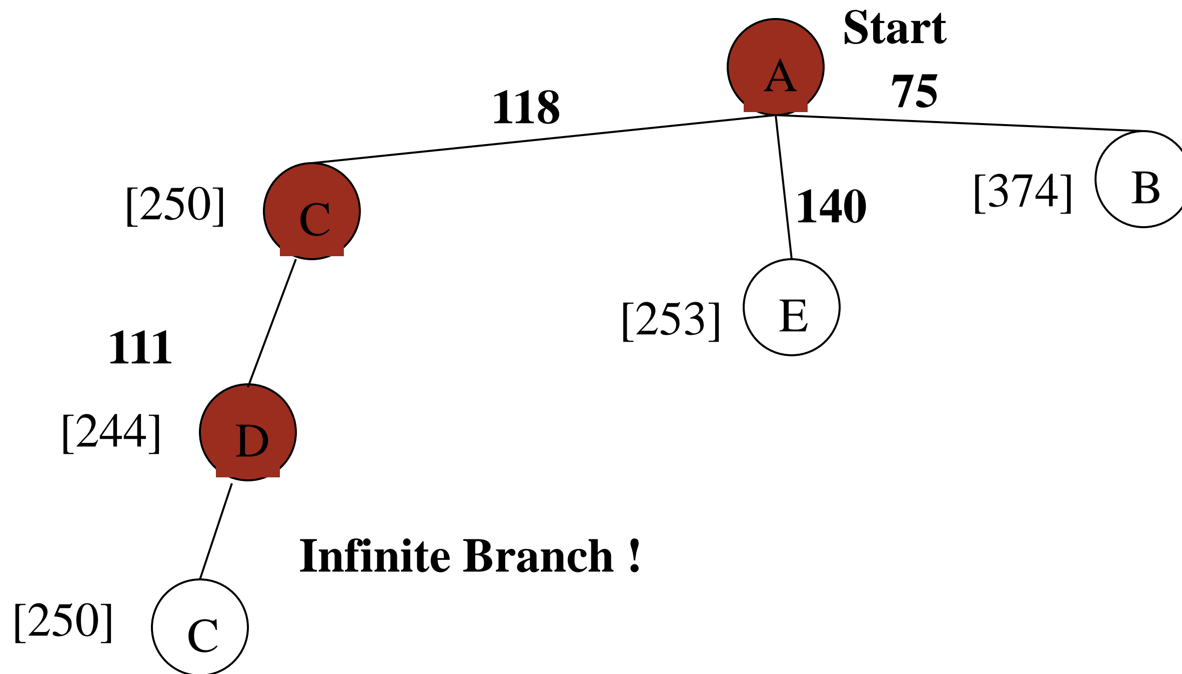
Greedy Search: Tree Search



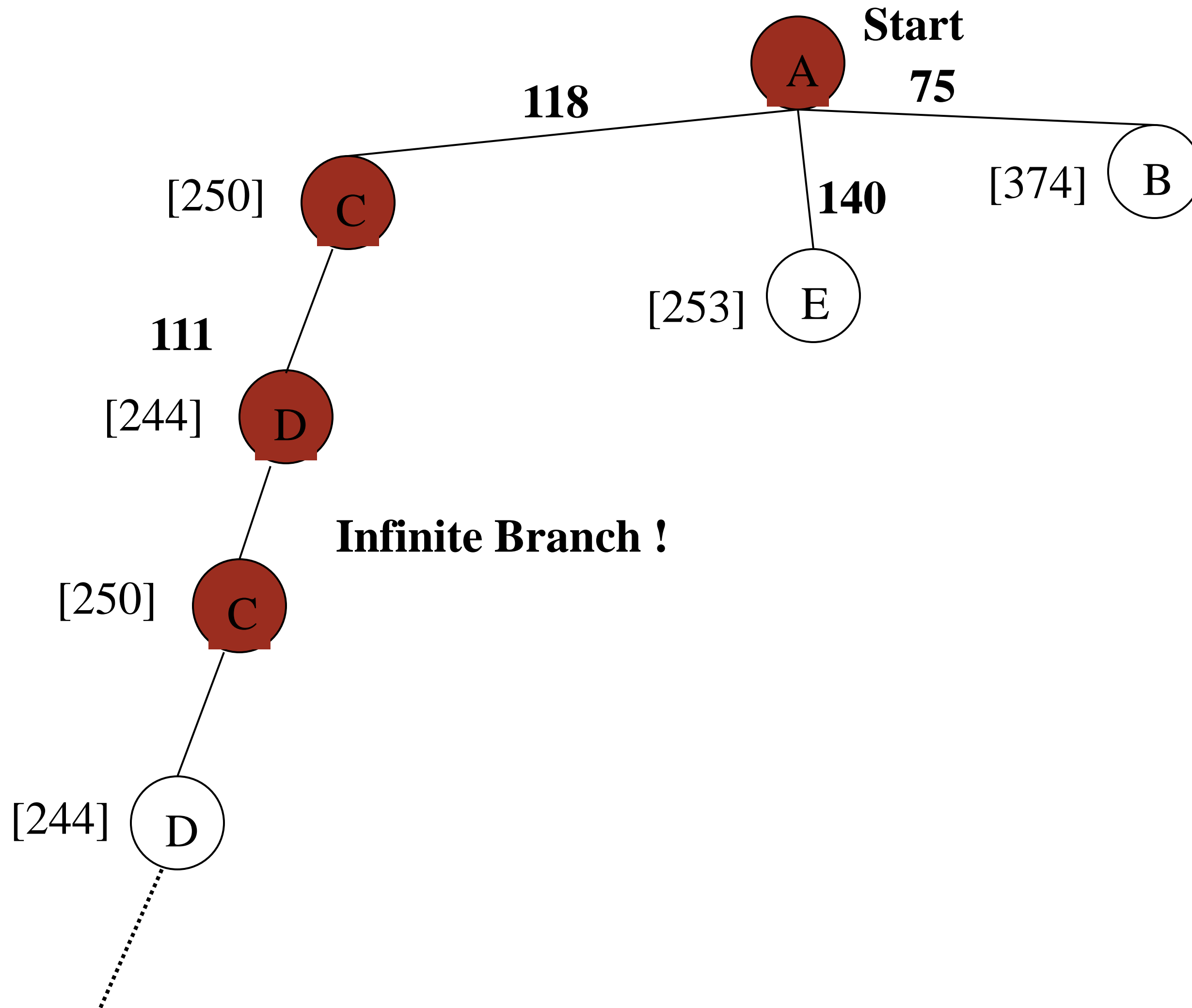
Greedy Search: Tree Search



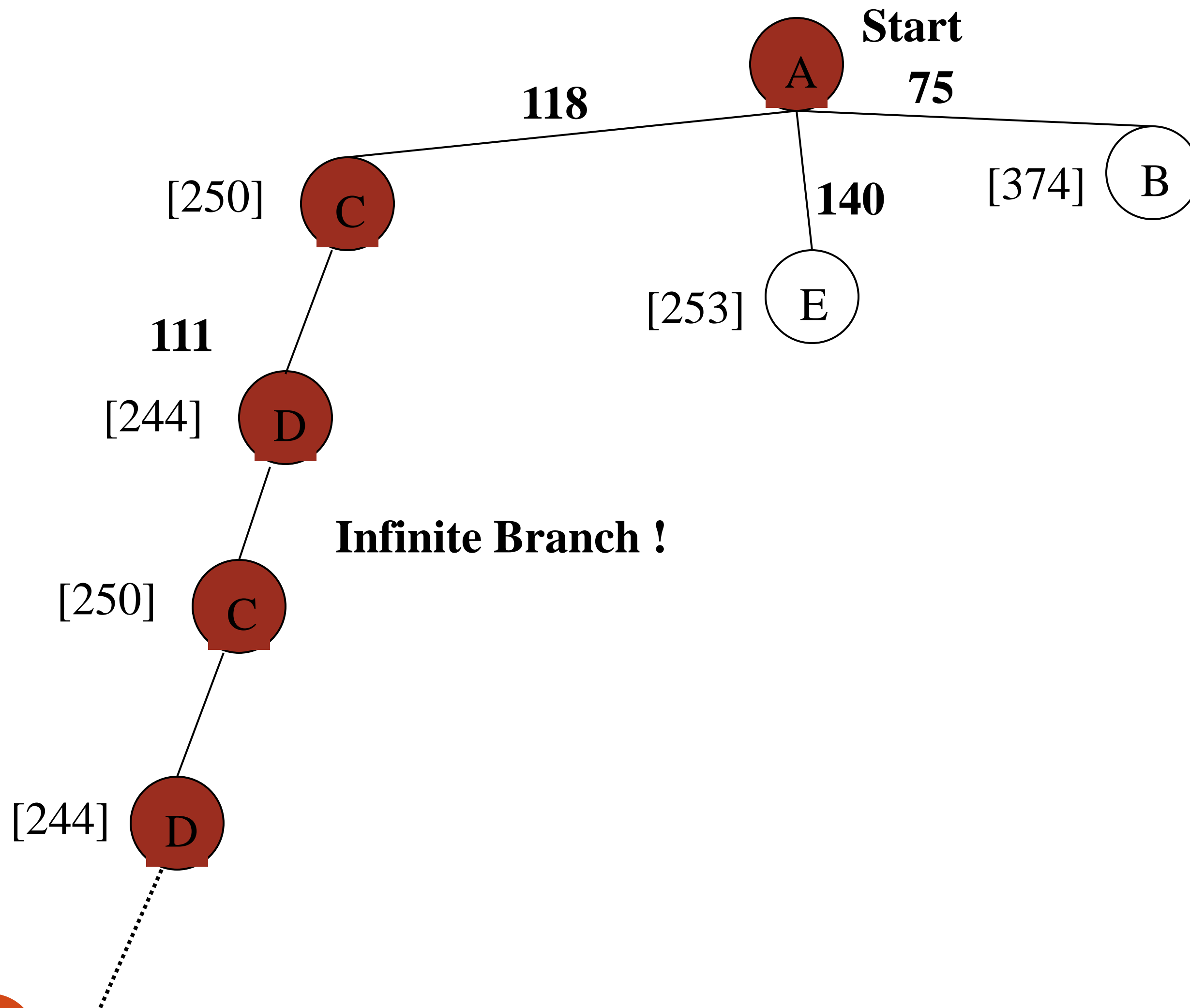
Greedy Search: Tree Search



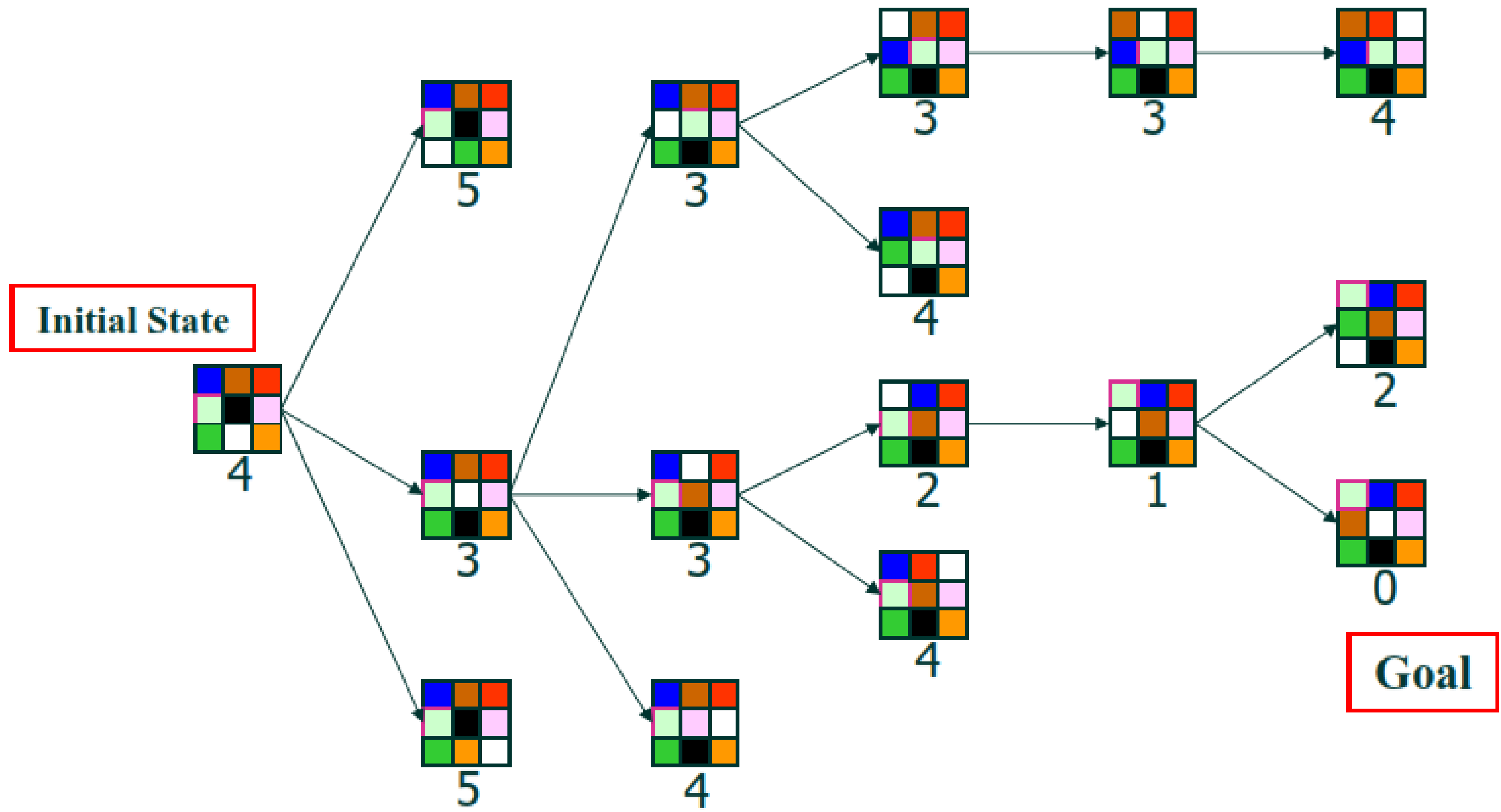
Greedy Search: Tree Search



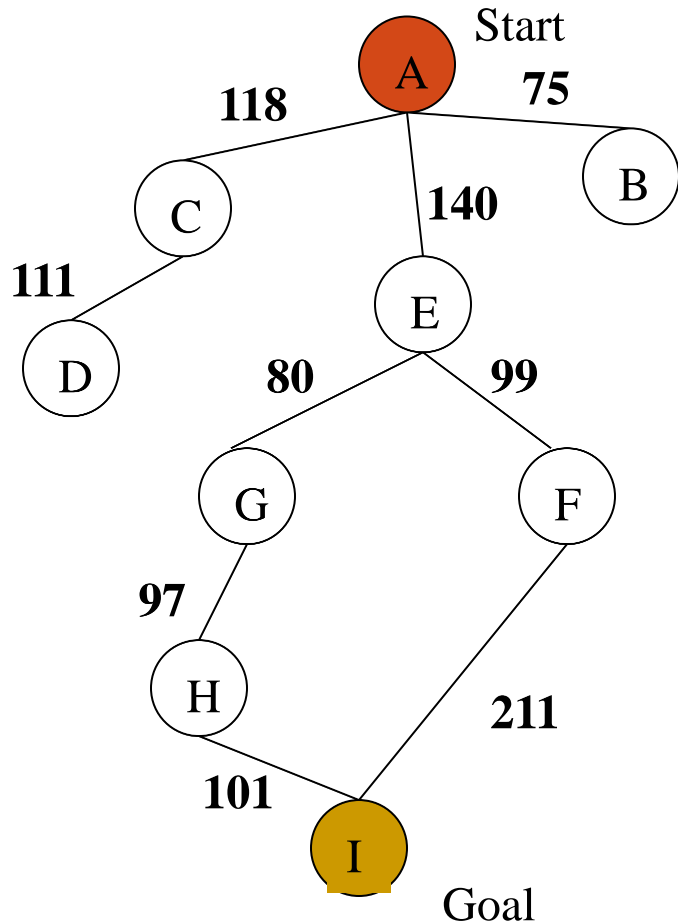
Greedy Search: Tree Search



Greedy Best-First Search: 8-Puzzle Example



Greedy Search: Time and Space Complexity ?

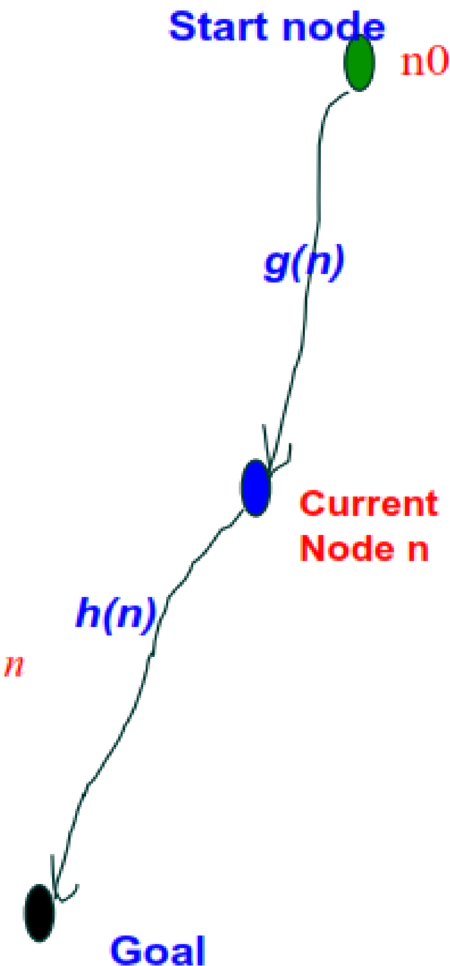


- Greedy search is not optimal.
- Greedy search is incomplete without systematic checking of repeated states.
- In the worst case, the Time and Space Complexity of Greedy Search are both $O(b^m)$

Where b is the branching factor and m the maximum path length

A* Search

- A* search is similar to Greedy best-first search with the difference that:
- A* search includes in its evaluation function the cost from the start node to the current node, in addition to the estimated cost from the current node to the goal.
- Evaluation function: $f(n) = g(n) + h(n)$, where:
  - $g(n)$: Cost so far to reach n (Uniform cost search minimizes it)
  - $h(n)$: Estimated cost to goal from n (best-first search minimizes it)
  - $f(n)$: Estimated total cost of path from the starting node n0 through n to the goal (best estimated cost for complete solution)
- Using current cost and estimated cost give this algorithm completeness (if the solution exists, it will be found) and optimality (it will find the best solution).
- All these advantages made A* algorithm the most popular search algorithm in AI applications.



A* Algorithm

1. Search queue Q is empty.
2. Place the start state s in Q with f value h(s).
3. If Q is empty, return failure.
4. Take node n from Q with lowest f value.

   (Keep Q sorted by f values and pick the first element).
5. If n is a goal node, stop and return solution.
6. Generate successors of node n.
7. For each successor n’ of n do:

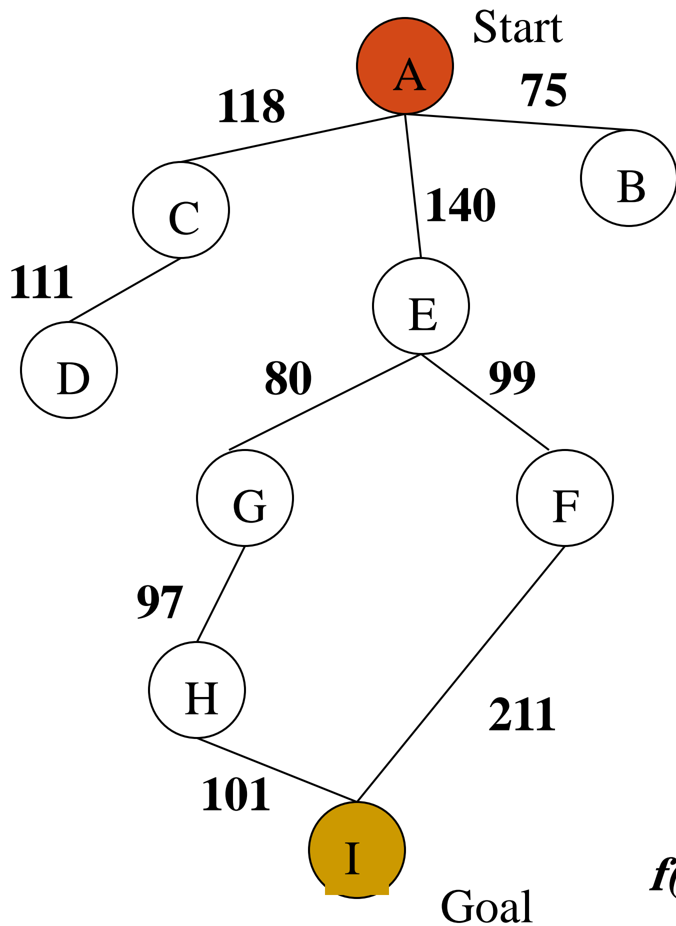
   a) Compute f(n’) = g(n) + cost(n,n’) + h(n’).

   b) If n’ is new (never generated before), add n’ to Q.

   c) If node n’ is already in Q with a higher f value, replace it with current f(n’) and place it in sorted order in Q.

   End for
8. Go back to step 3.

A* Search



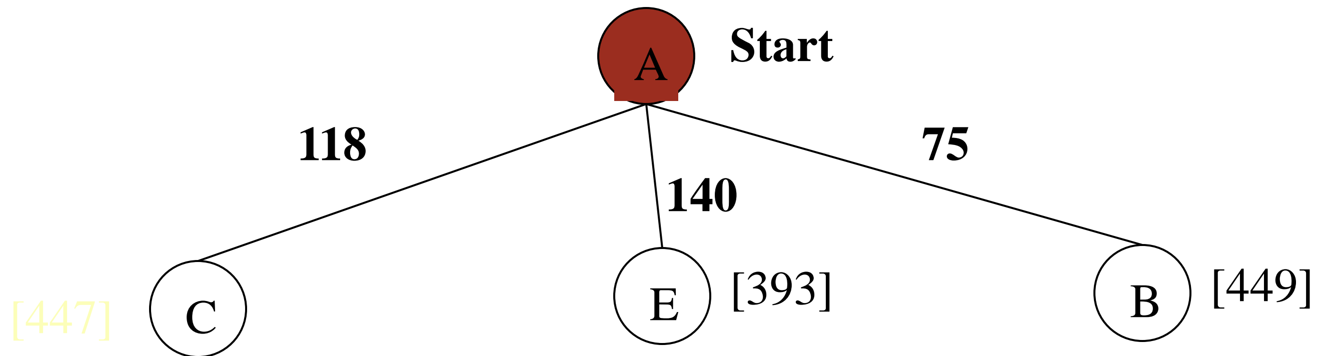
State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	98
I	0

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

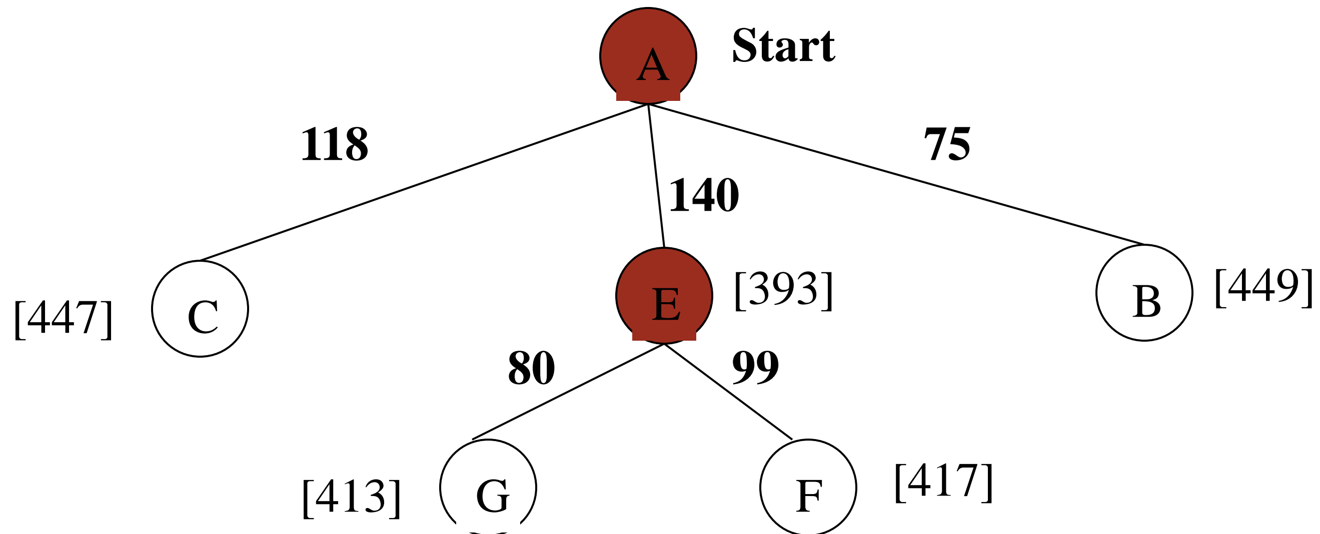
g(n): is the exact cost to reach node *n* from the initial state.

A* Search: Tree Search

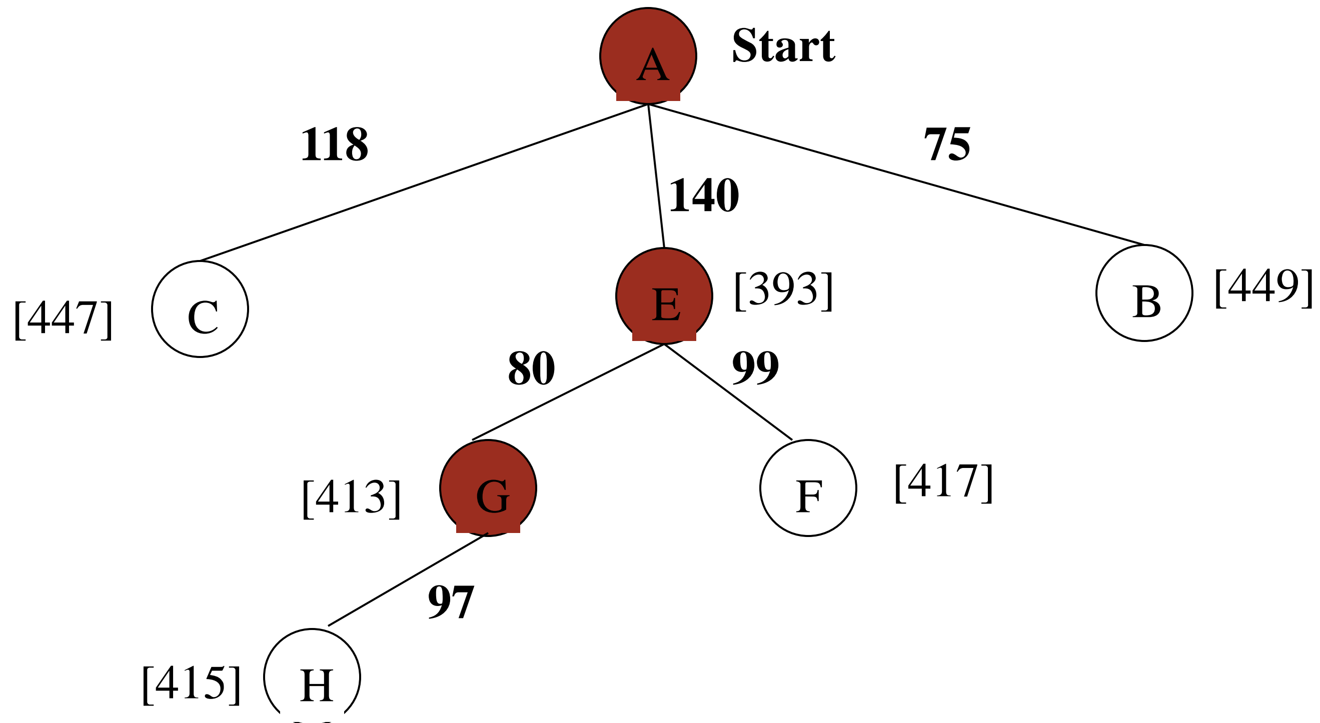
A* Search: Tree Search



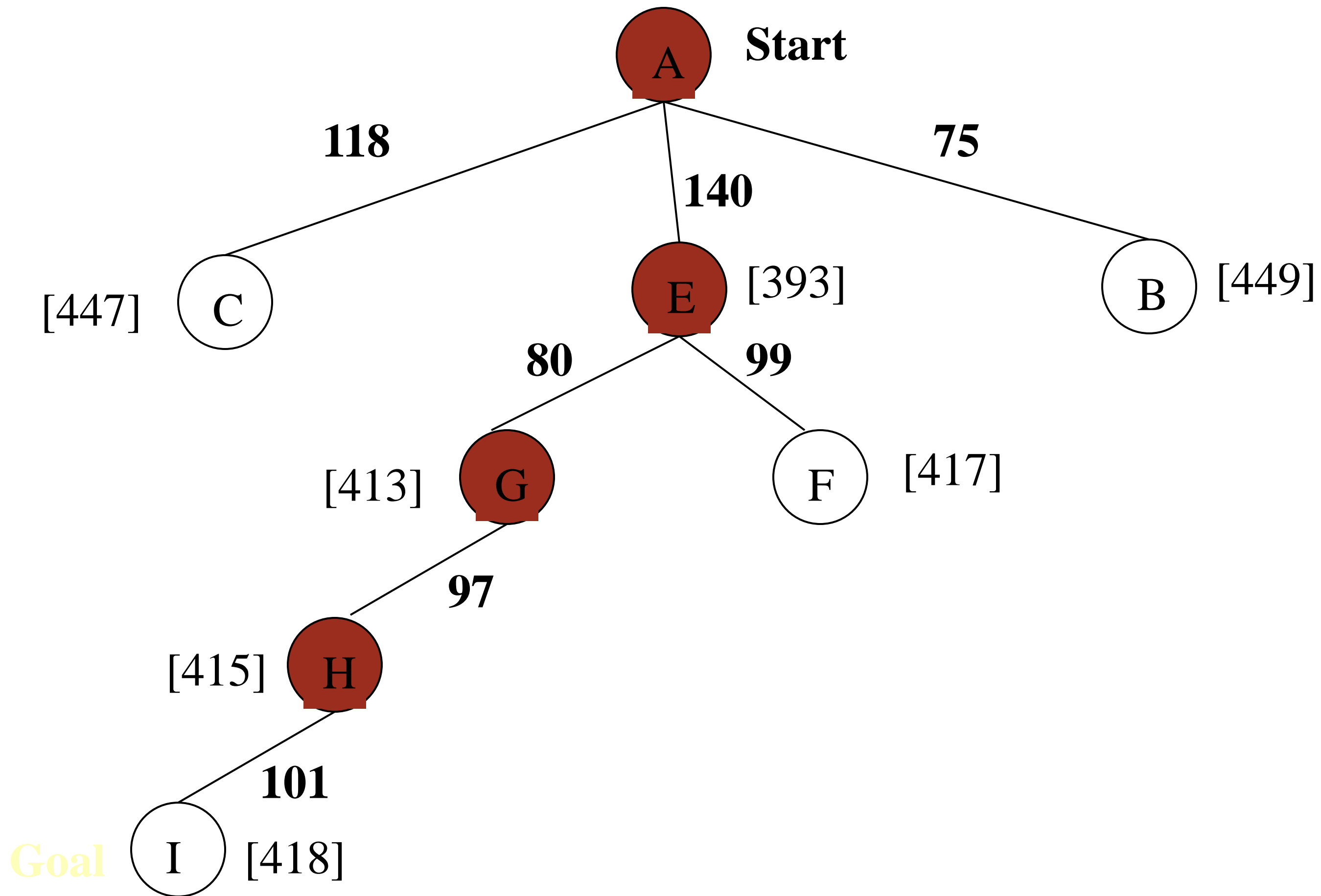
A* Search: Tree Search



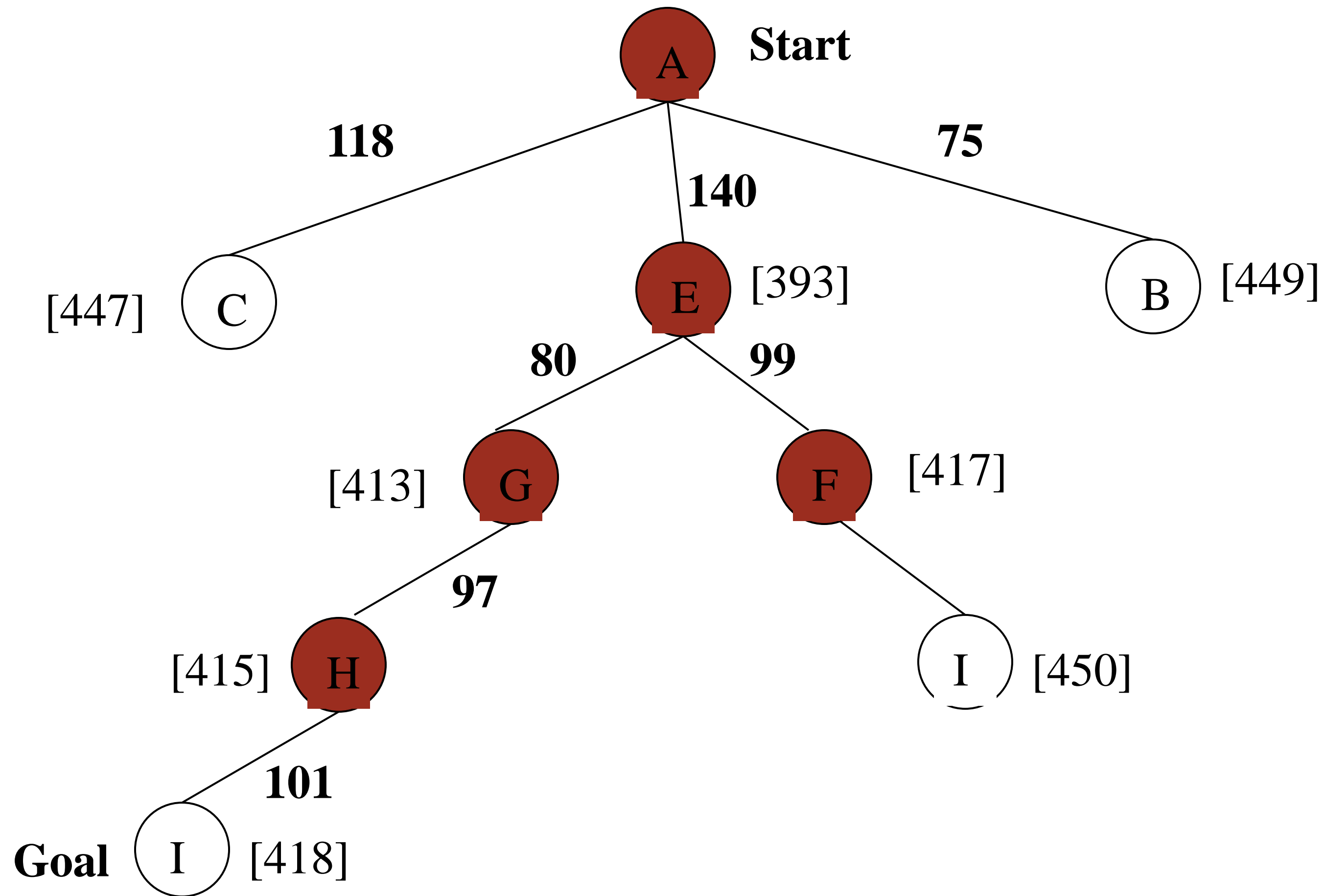
A* Search: Tree Search



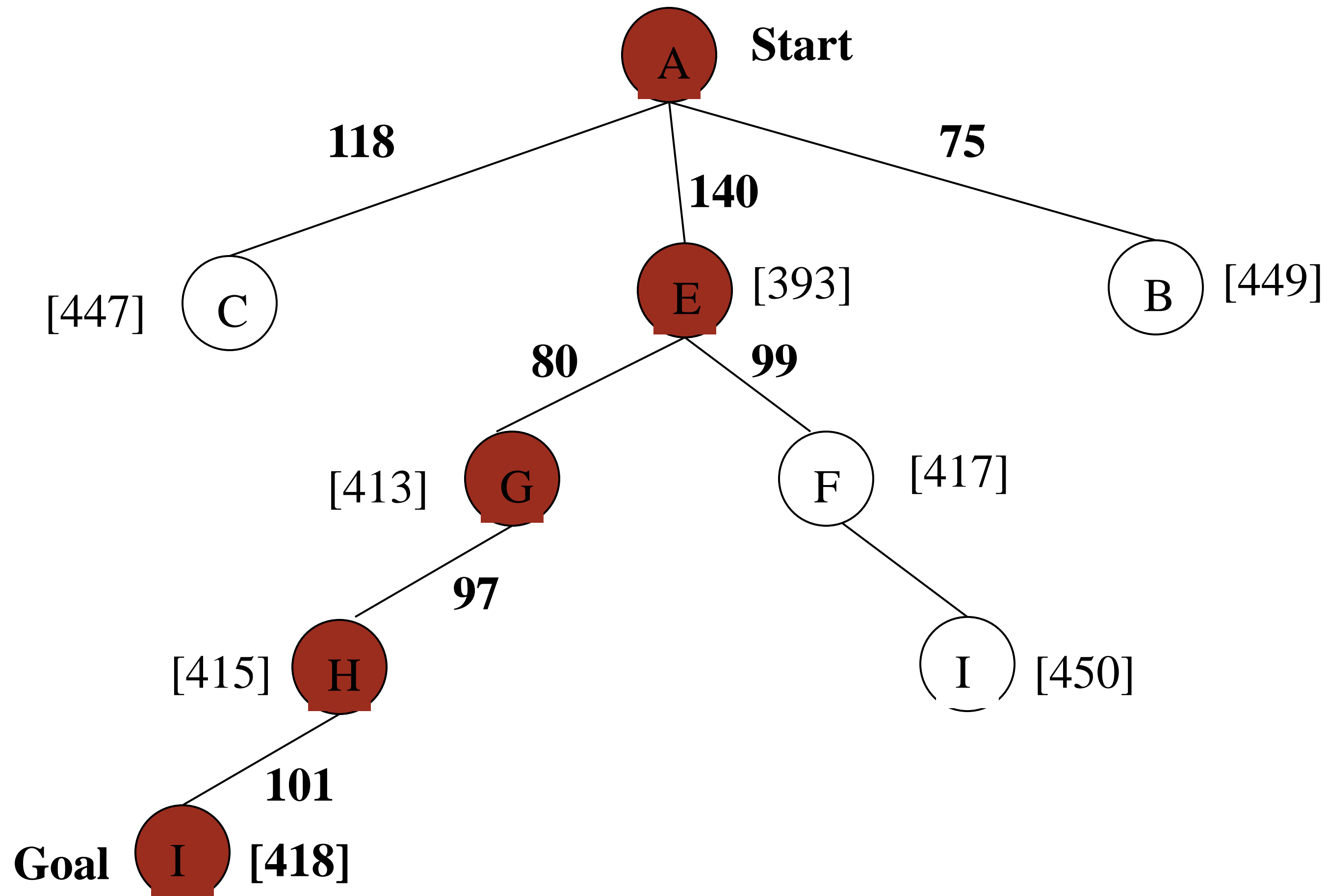
A* Search: Tree Search



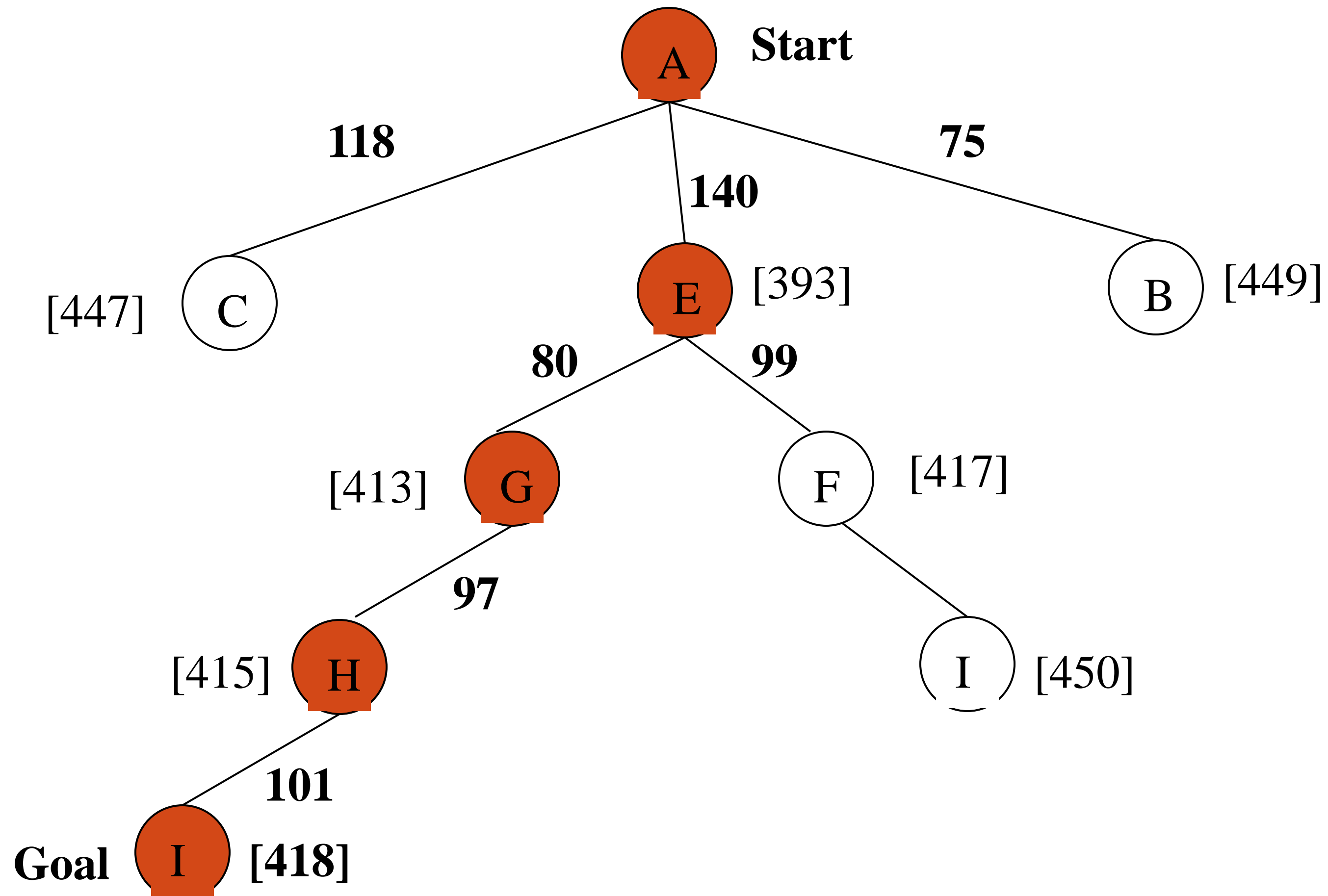
A* Search: Tree Search



A* Search: Tree Search

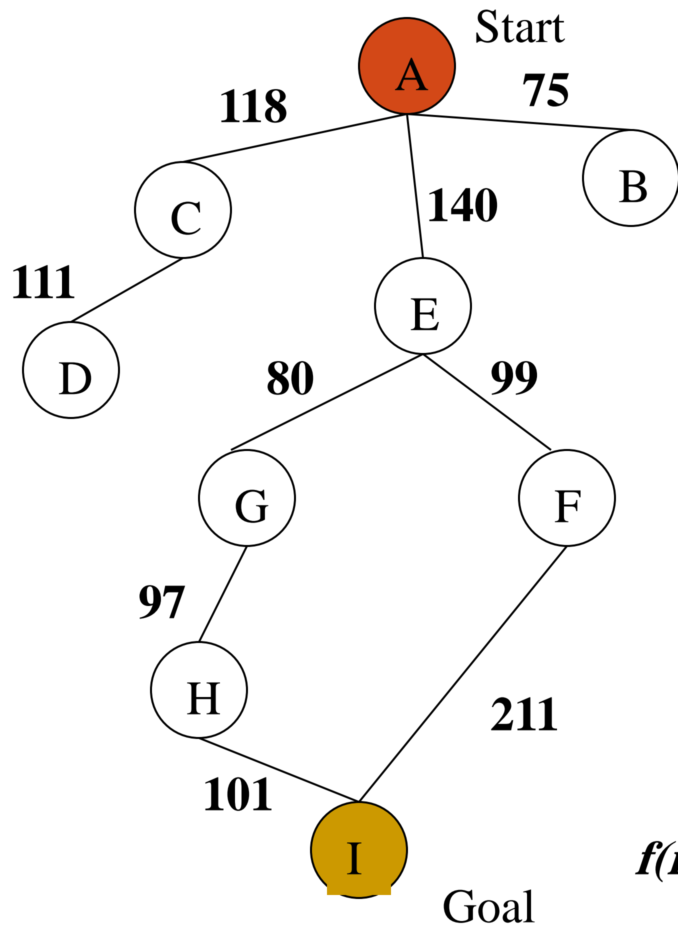


A* Search: Tree Search



A* Search: *h* not admissible !

h() overestimates the cost to reach the goal state



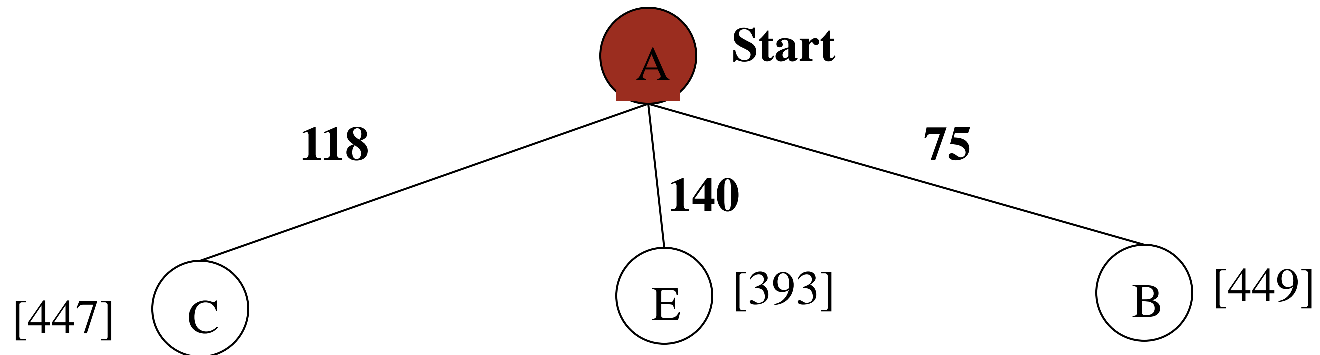
State	Heuristic: h(n)
A	366
B	374
C	329
D	244
E	253
F	178
G	193
H	138
I	0

f(n) = g(n) + h **(*n*) – (H-I) Overestimated**

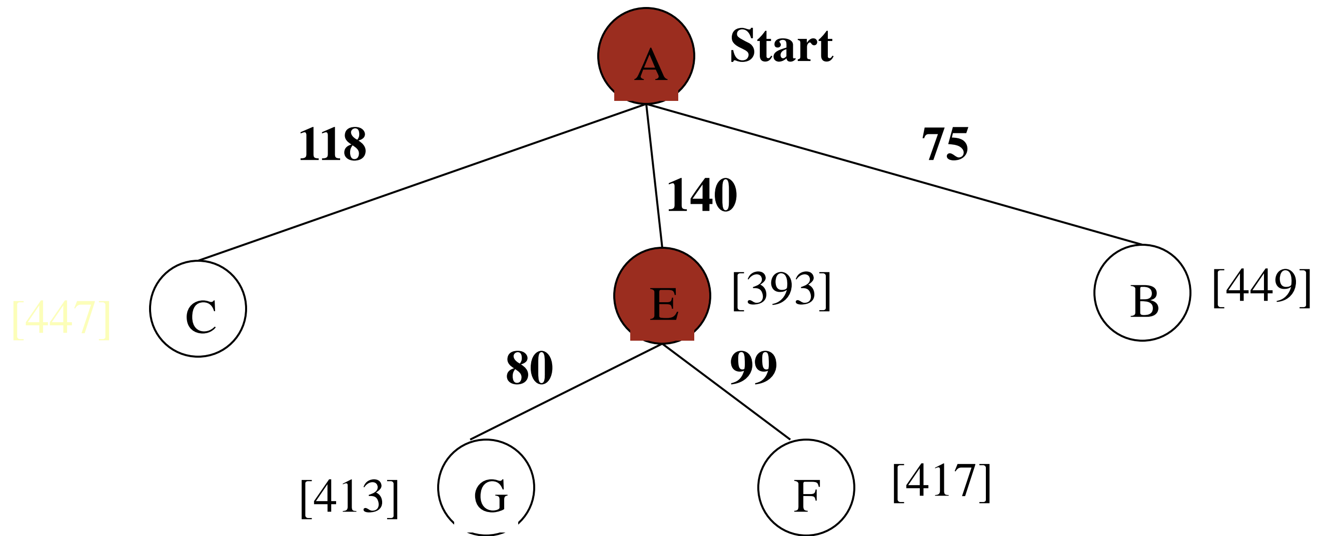
g(n): is the exact cost to reach node *n* from the initial state.

A* Search: Tree Search

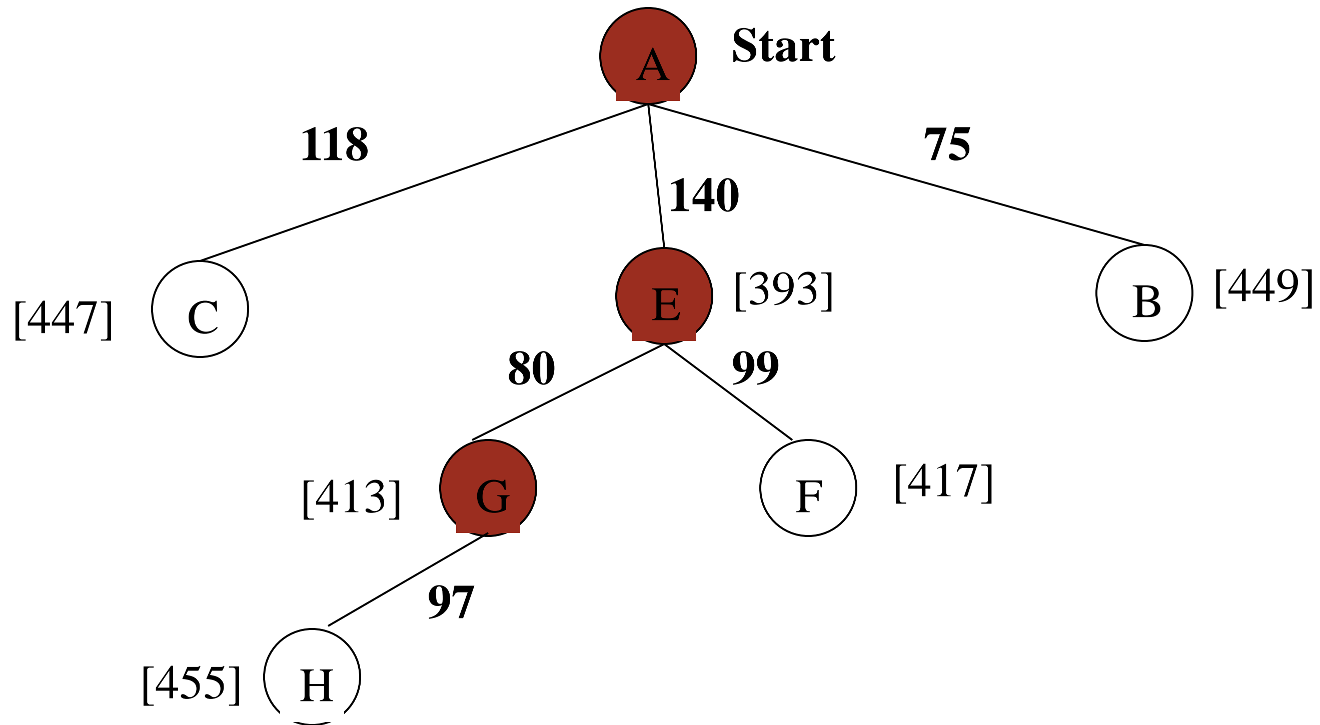
A* Search: Tree Search



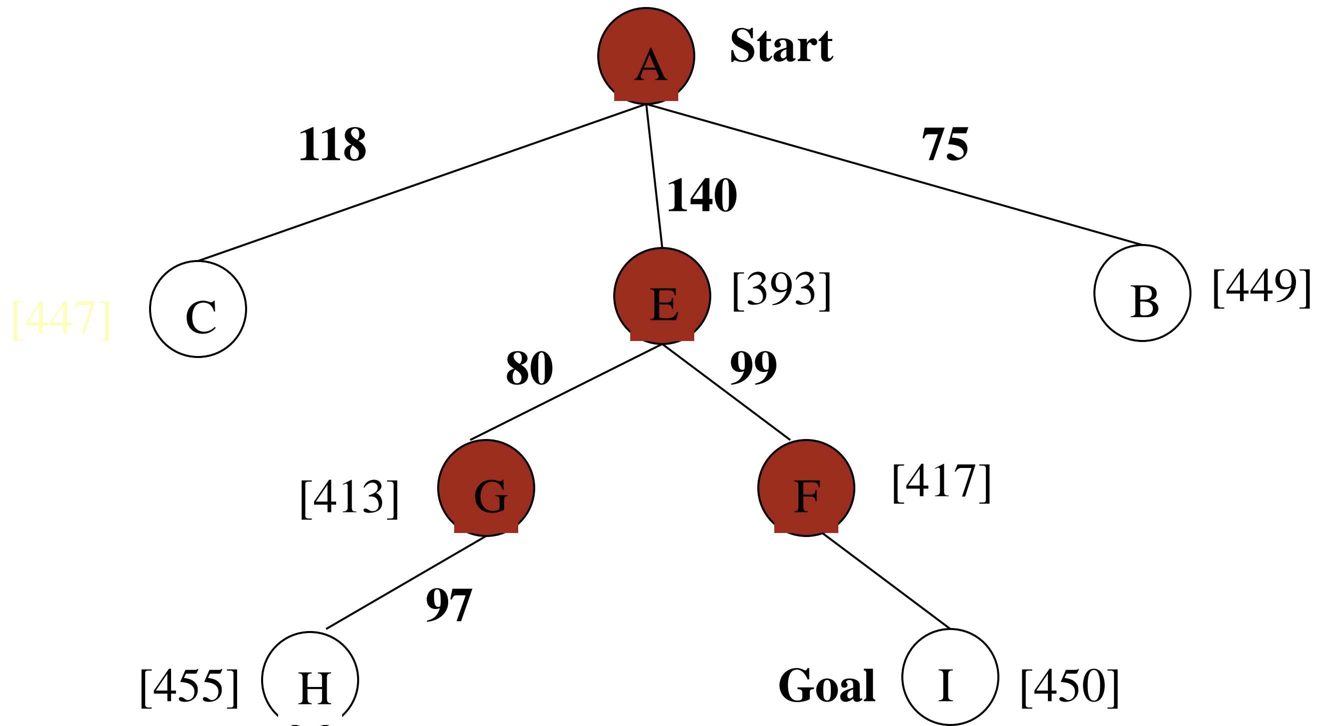
A* Search: Tree Search



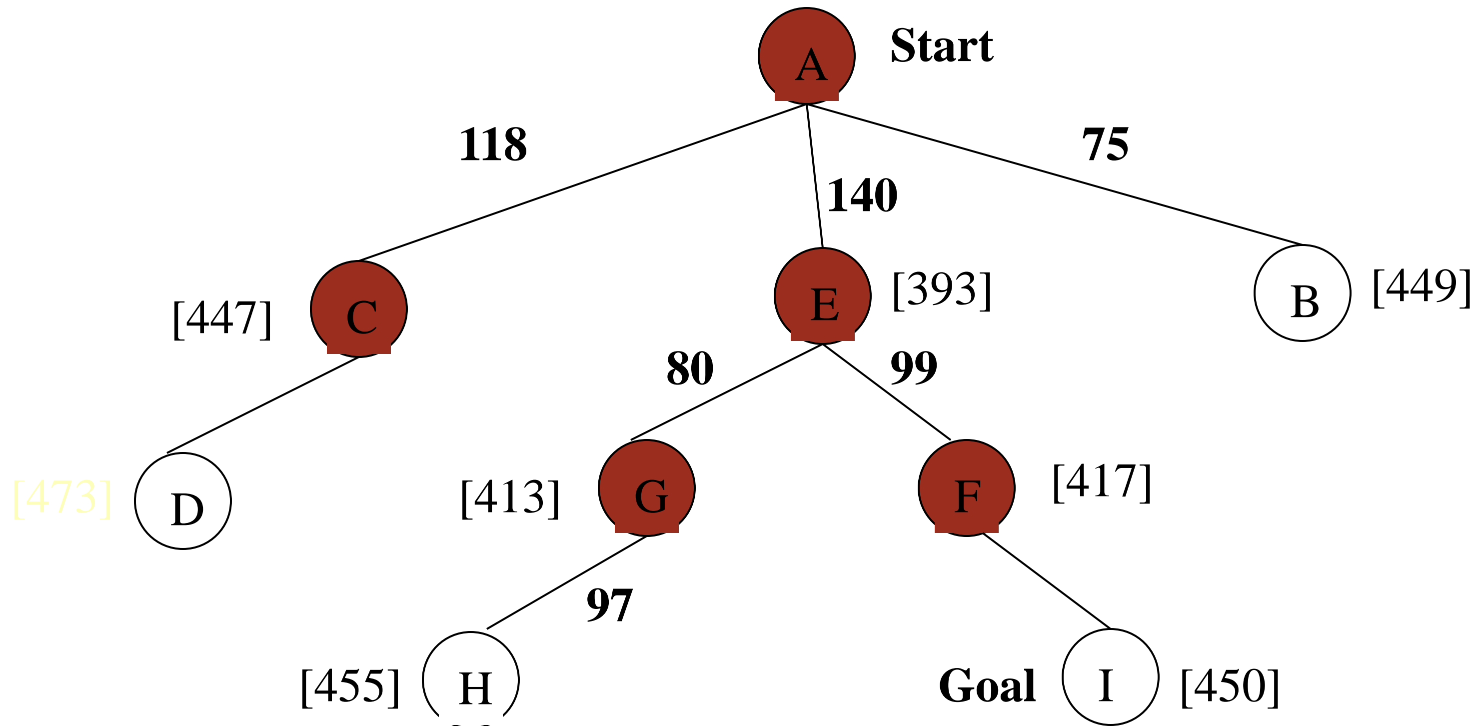
A* Search: Tree Search



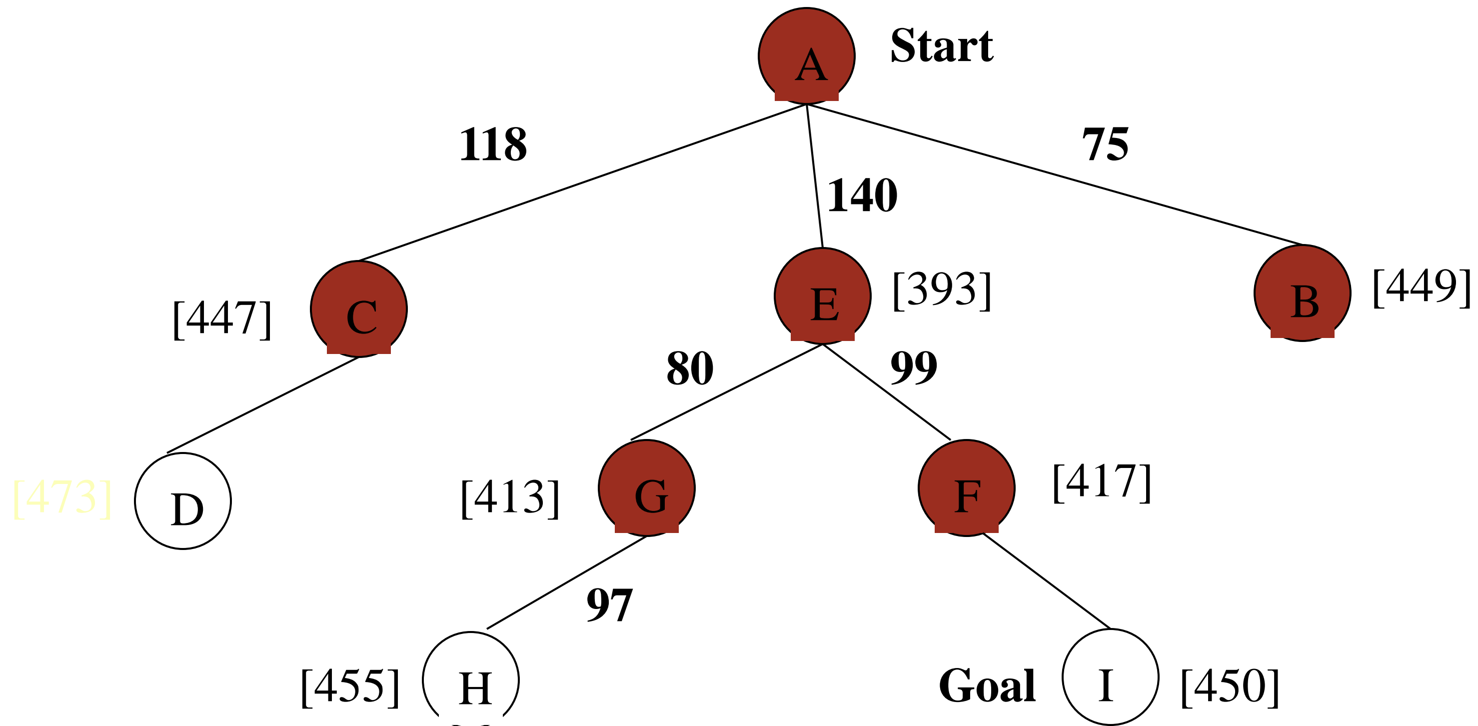
A* Search: Tree Search



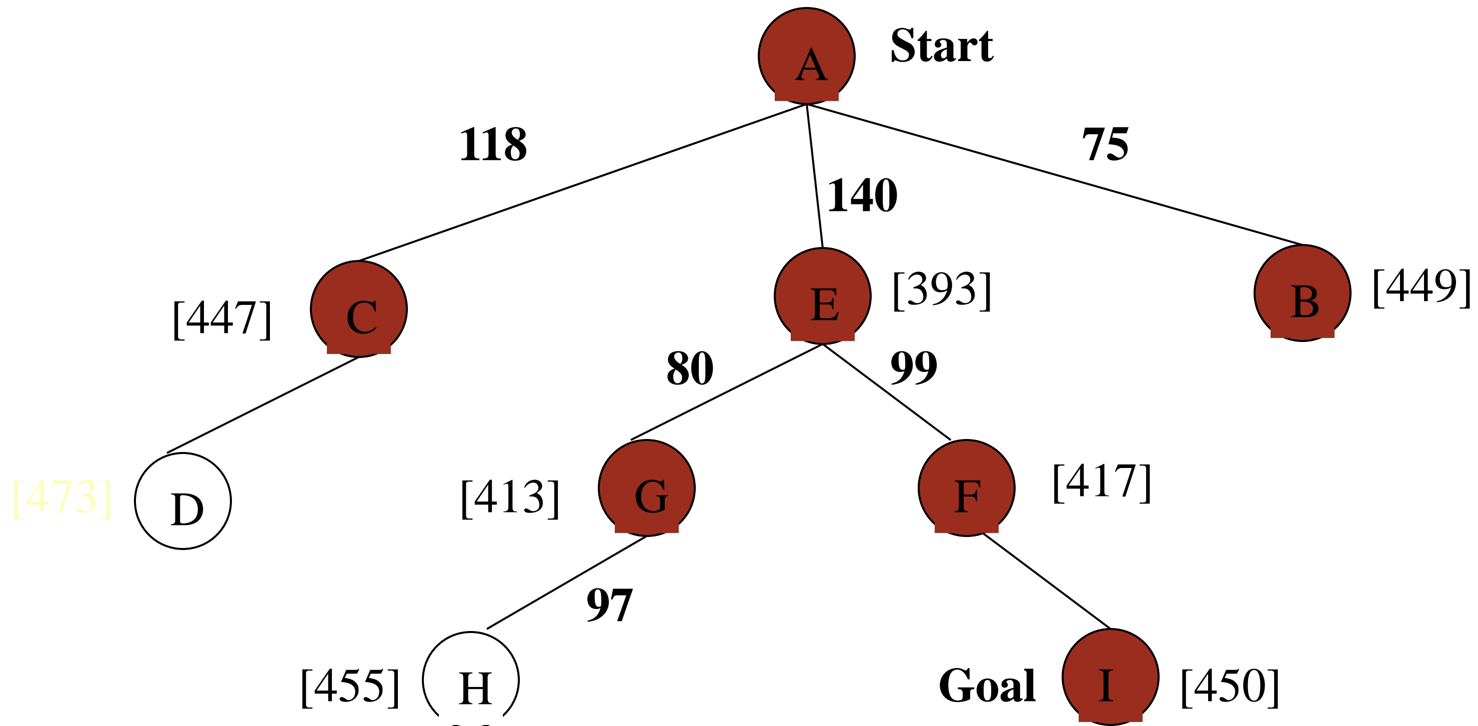
A* Search: Tree Search



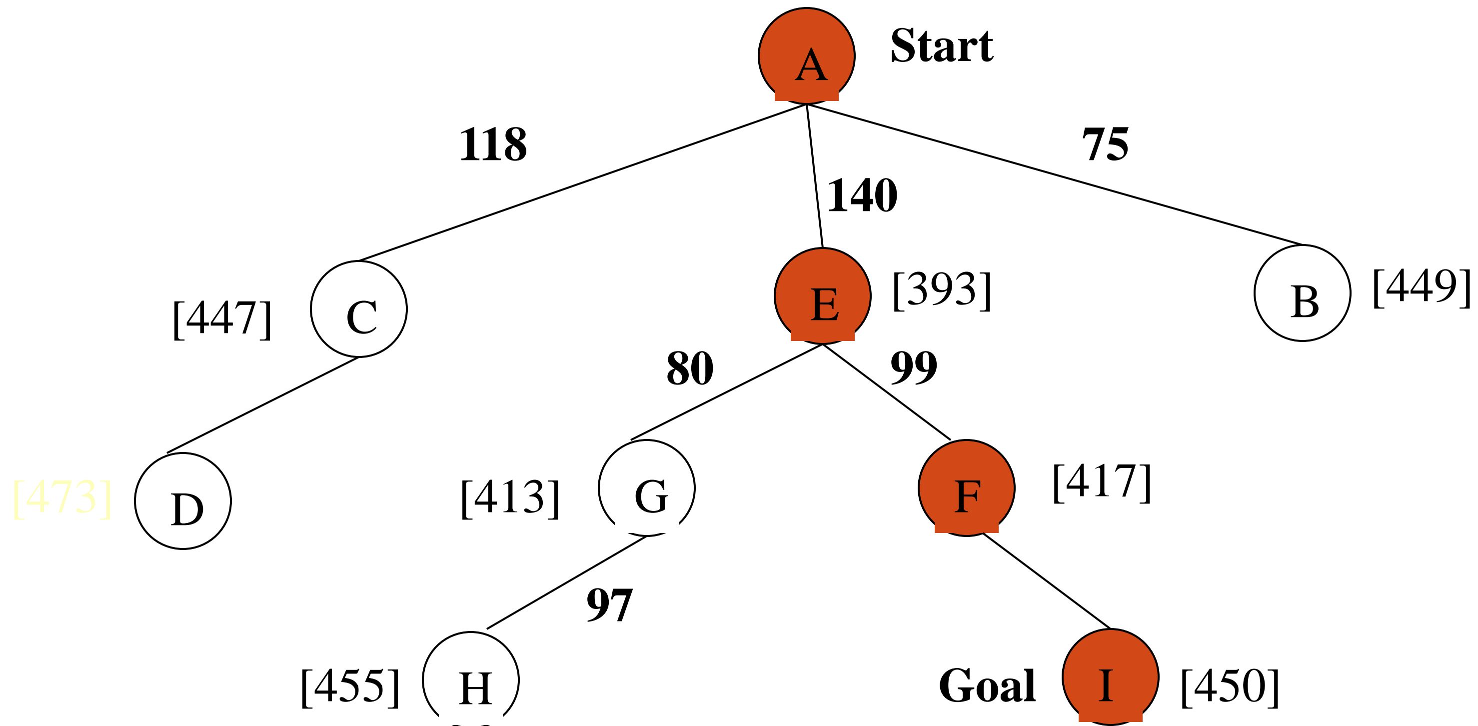
A* Search: Tree Search



A* Search: Tree Search



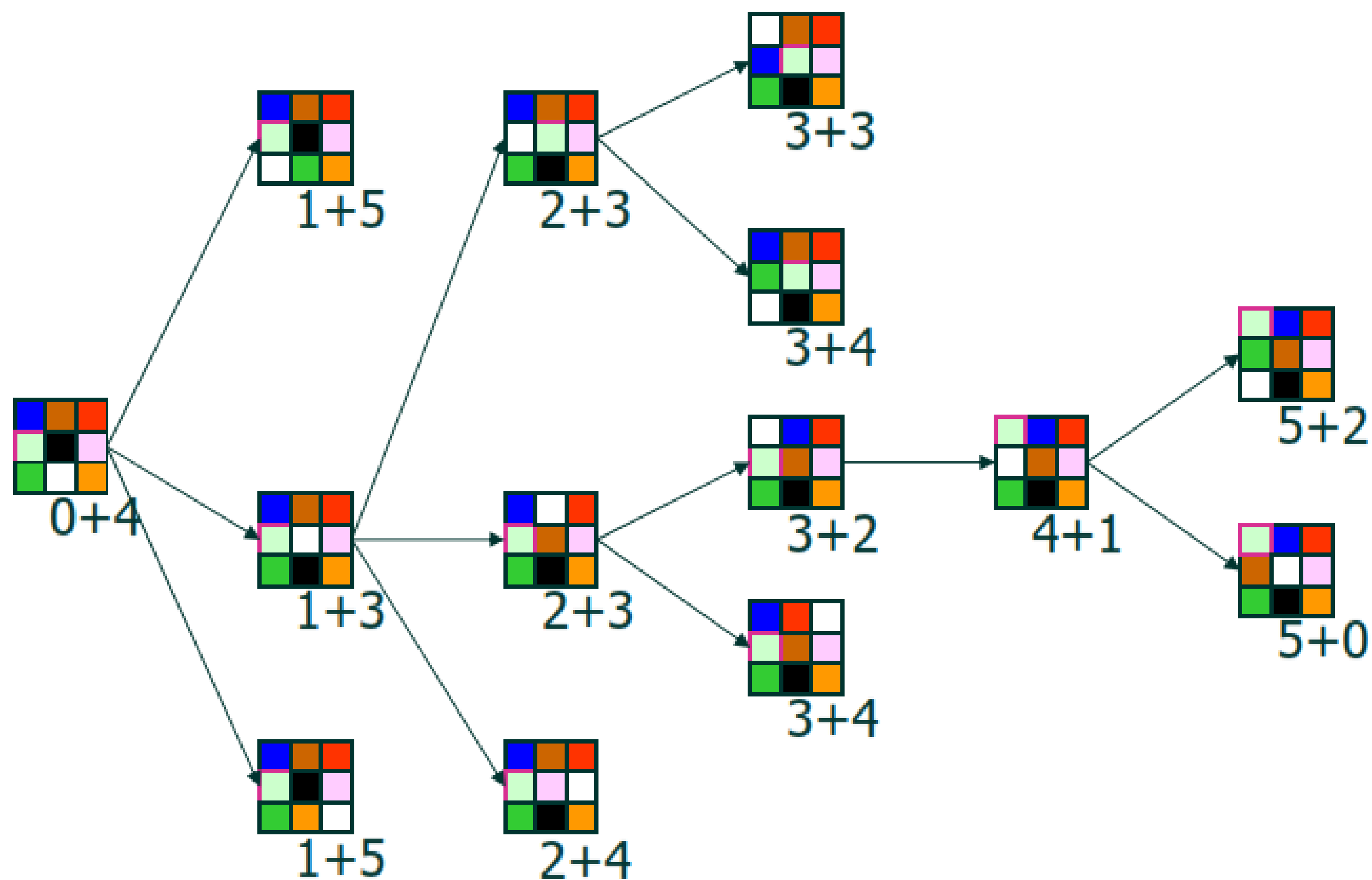
A* Search: Tree Search



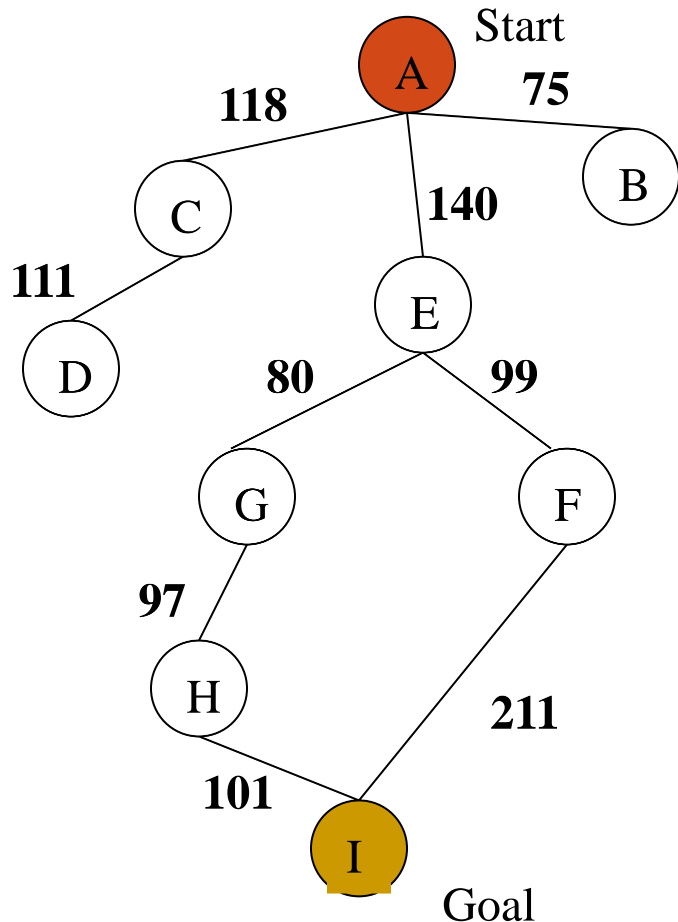
A* not optimal !!!

8-Puzzle Example: A* Search

f(n) = g(n) + h(n): with h(n) = number of misplaced tiles (not including the blank)



A* Search: Analysis



- A* is complete except if there is an infinity of nodes with $f < f(G)$.
- A* is optimal if heuristic *h* is admissible.
- Time complexity depends on the quality of heuristic but is still exponential.
- For space complexity, A* keeps all nodes in memory. A* has worst case $O(b^d)$ space complexity, but an iterative deepening version is possible (IDA*).

A* Properties

- the value of *f* never decreases along any path starting from the initial node
  - also known as monotonicity of the function
  - almost all admissible heuristics show monotonicity
    - those that don't can be modified through minor changes

- this property can be used to draw contours
  - regions where the f-cost is below a certain threshold
  - with uniform cost search ($h = 0$), the contours are circular
  - the better the heuristics *h*, the narrower the contour around the optimal path

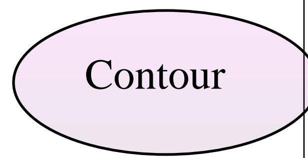
A* Snapshot with Contour f=11

Initial
Visited
Fringe
Current
Visible
Goal

Edge Cost 9

Heuristics 7

f-cost 10



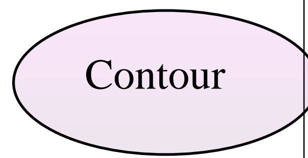
A* Snapshot with Contour f=13

Initial
Visited
Fringe
Current
Visible
Goal

Edge Cost 9

Heuristics 7

f-cost 10



Optimality of A*

- A* will find the optimal solution
  - the first solution found is the optimal one
- A* is optimally efficient
  - no other algorithm is guaranteed to expand fewer nodes than A*
- A* is not always "the best" algorithm
  - optimality refers to the expansion of nodes
    - other criteria might be more relevant
  - it generates and keeps all nodes in memory
    - improved in variations of A*

Complexity of A*

- the number of nodes within the goal contour search space is still exponential
  - with respect to the length of the solution
  - better than other algorithms, but still problematic

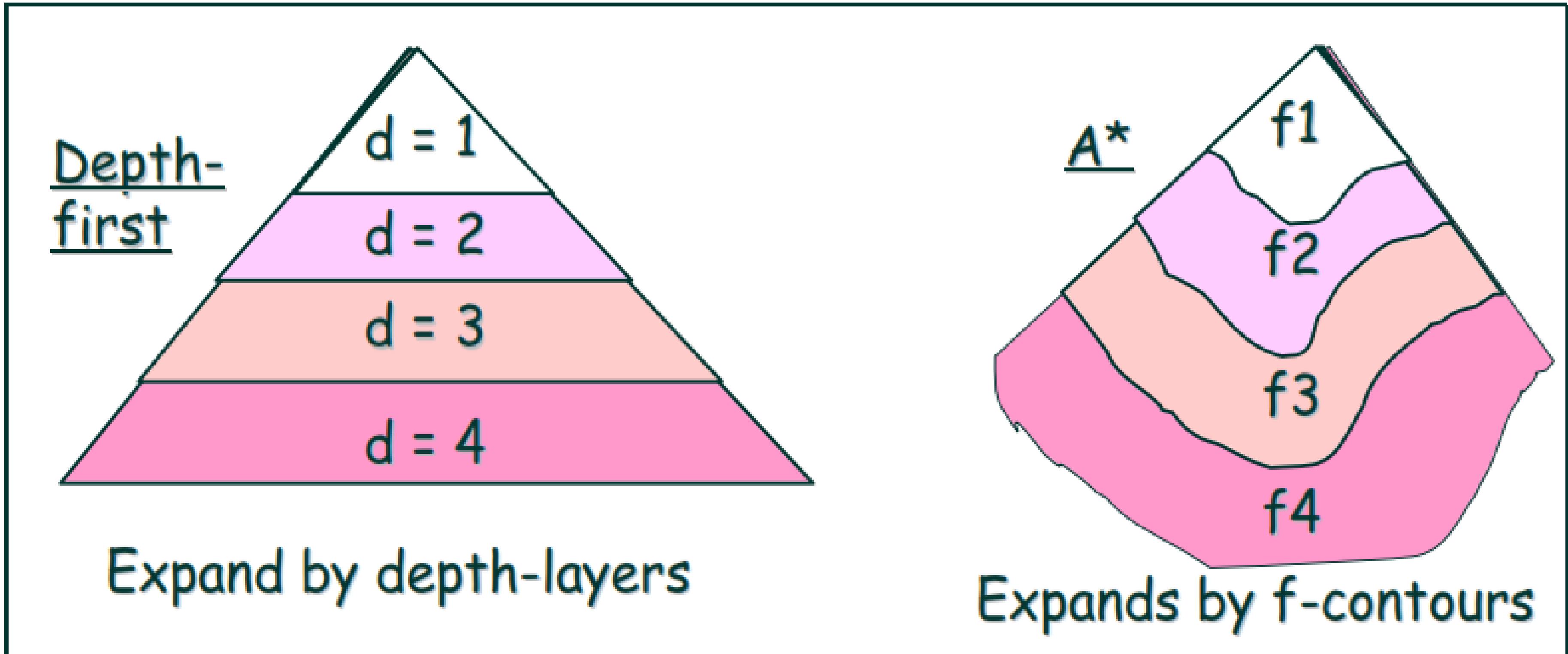
- frequently, space complexity is more severe than time complexity
  - A* keeps all generated nodes in memory

Improving A*: Memory-bounded Heuristic Search

- Iterative-Deepening A* (IDA*)
  - Using f(g+h) as a cut off rather than the depth for the iteration.
  - Cutoff value is the smallest f-cost of *any node* that exceeded the cutoff on the previous iteration; keep these nodes only.
  - Space complexity O(bd)
- Recursive Best-First Search (RBFS)
  - It replaces the f-value of each node along the path with the best f-value of its children.
  - Space complexity O(bd)
- Simplified Memory Bounded A* (SMA*)
  - Works like A* until memory is full
  - Then SMA* drops the node in the fringe with the largest f value and "backs up" this value to its parent.
  - When all children of a node n have been dropped, the smallest backed up value replaces f(n)

Iterative deepening A*

- IDA* is similar to Iterative depth-first:

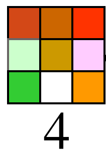


IDA* Algorithm

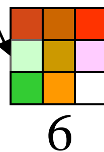
- In the first iteration, we determine a **"f-cost limit" – cut-off value**
  $f(n_0) = g(n_0) + h(n_0) = h(n_0)$, where n_0 is the start node.
- We expand nodes using the **depth-first algorithm** and backtrack whenever $f(n)$ for an expanded node n exceeds the cut-off value.
- If this search does not succeed, determine the **lowest f-value** among the nodes that were visited but not expanded.
- Use this f-value as the **new limit value – cut-off value** and do another depth-first search.
- Repeat this procedure until a goal node is found.

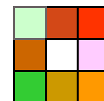
8-Puzzle

$f(N) = g(N) + h(N)$
with $h(N)$ = number of misplaced tiles



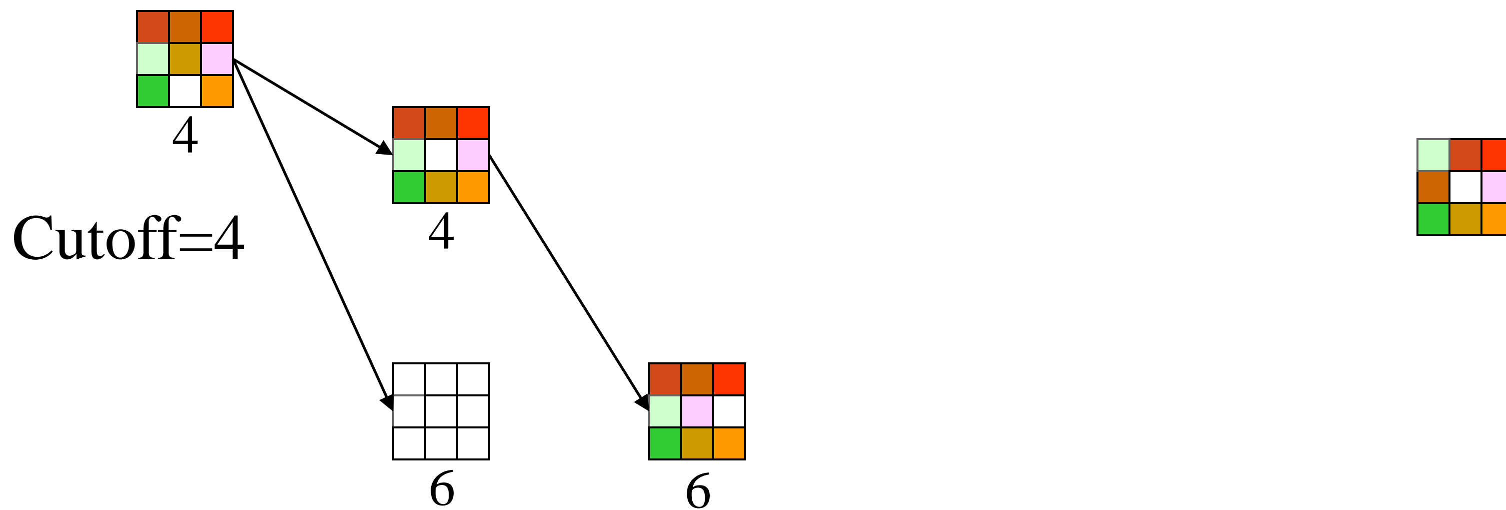
Cutoff=4





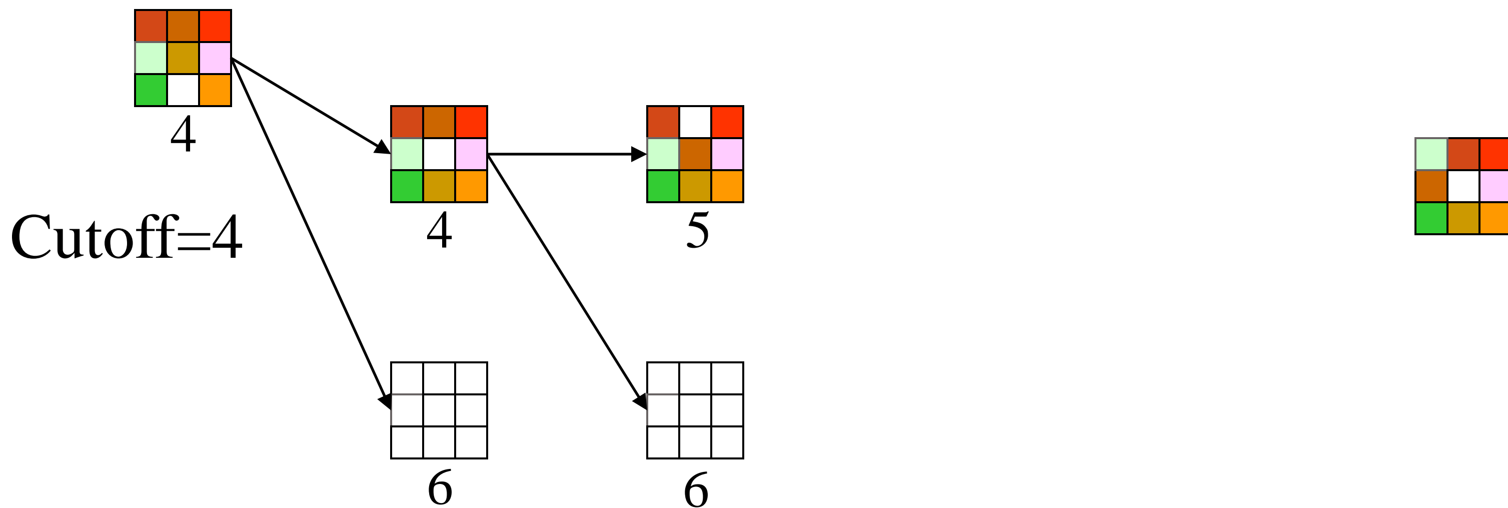
8-Puzzle

$f(N) = g(N) + h(N)$
with $h(N)$ = number of misplaced tiles



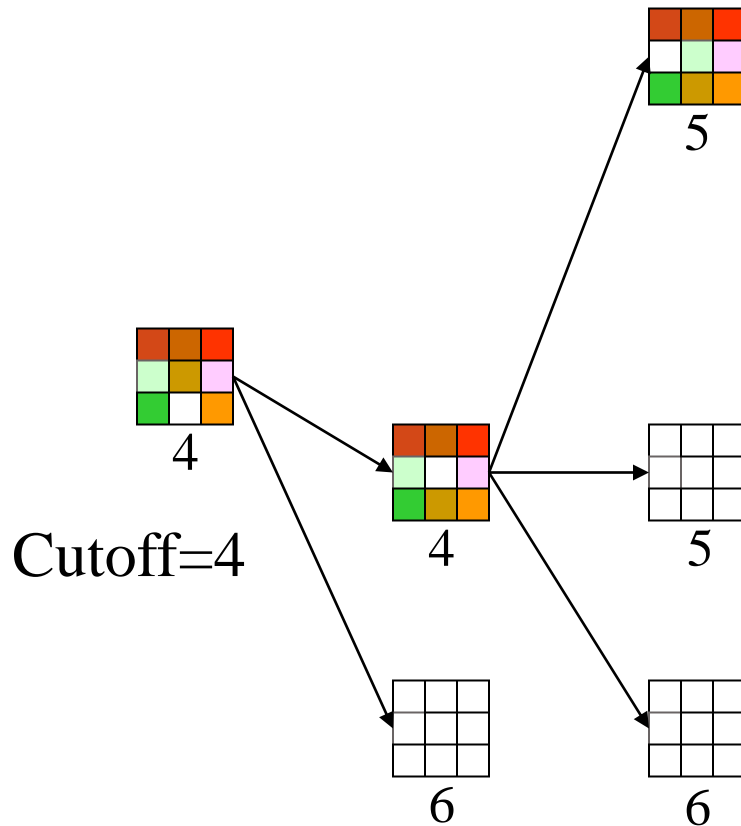
8-Puzzle

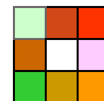
$f(N) = g(N) + h(N)$
with $h(N)$ = number of misplaced tiles



8-Puzzle

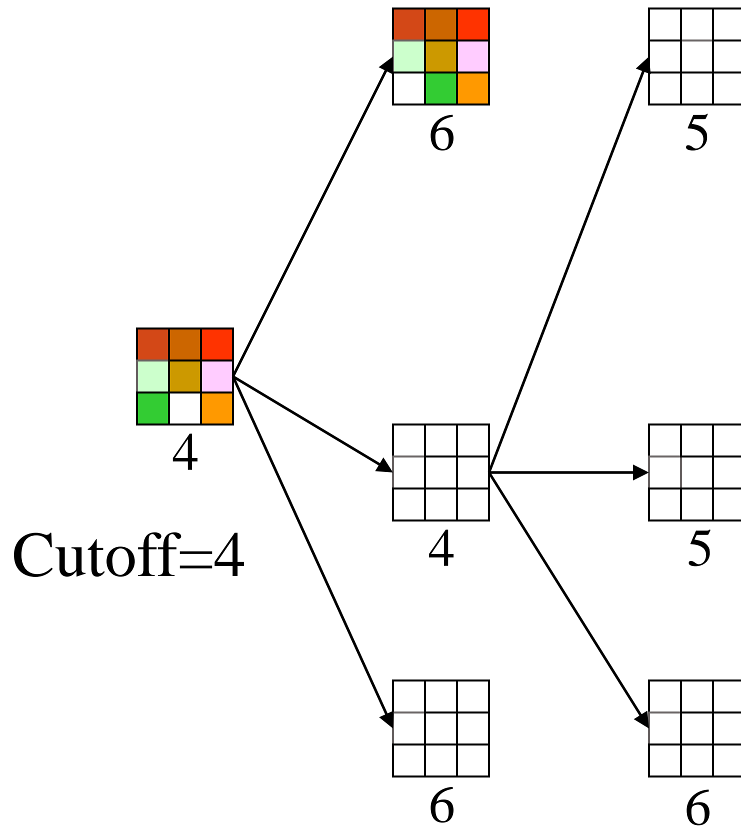
$f(N) = g(N) + h(N)$
with $h(N)$ = number of misplaced tiles

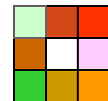




8-Puzzle

$f(N) = g(N) + h(N)$
with $h(N)$ = number of misplaced tiles

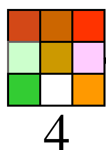




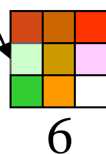
Aziz M. Qaroush - Birzeit University

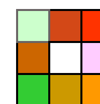
8-Puzzle

$f(N) = g(N) + h(N)$
with $h(N)$ = number of misplaced tiles


4

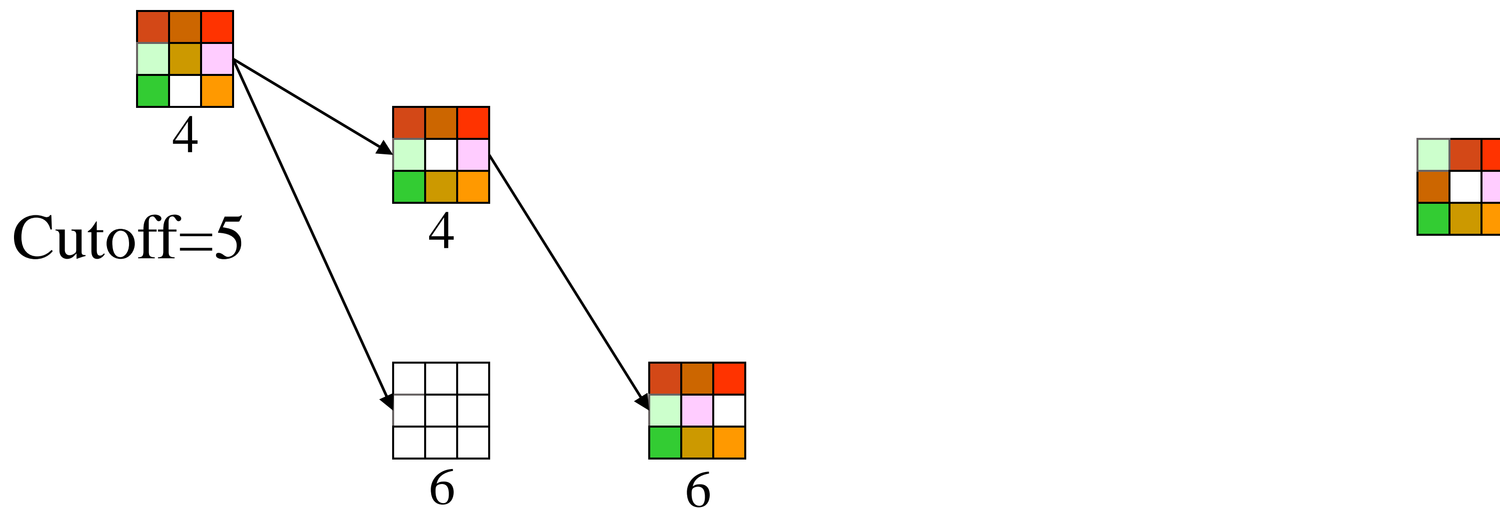
Cutoff=5


6



8-Puzzle

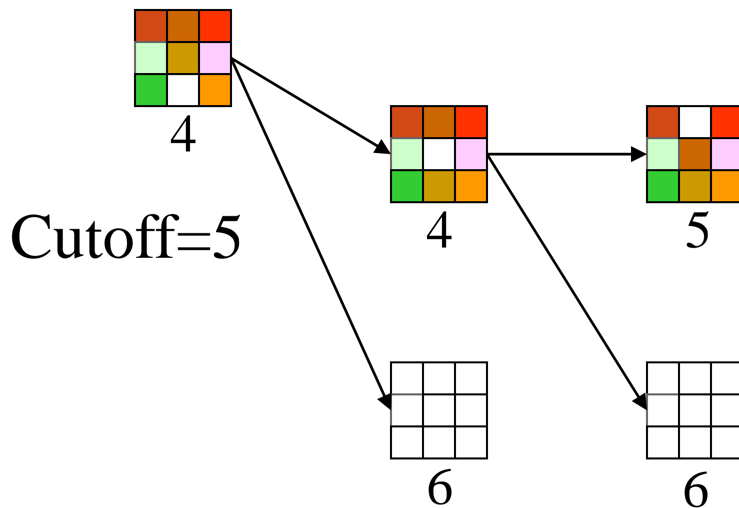
f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles



Aziz M. Qaroush - Birzeit University

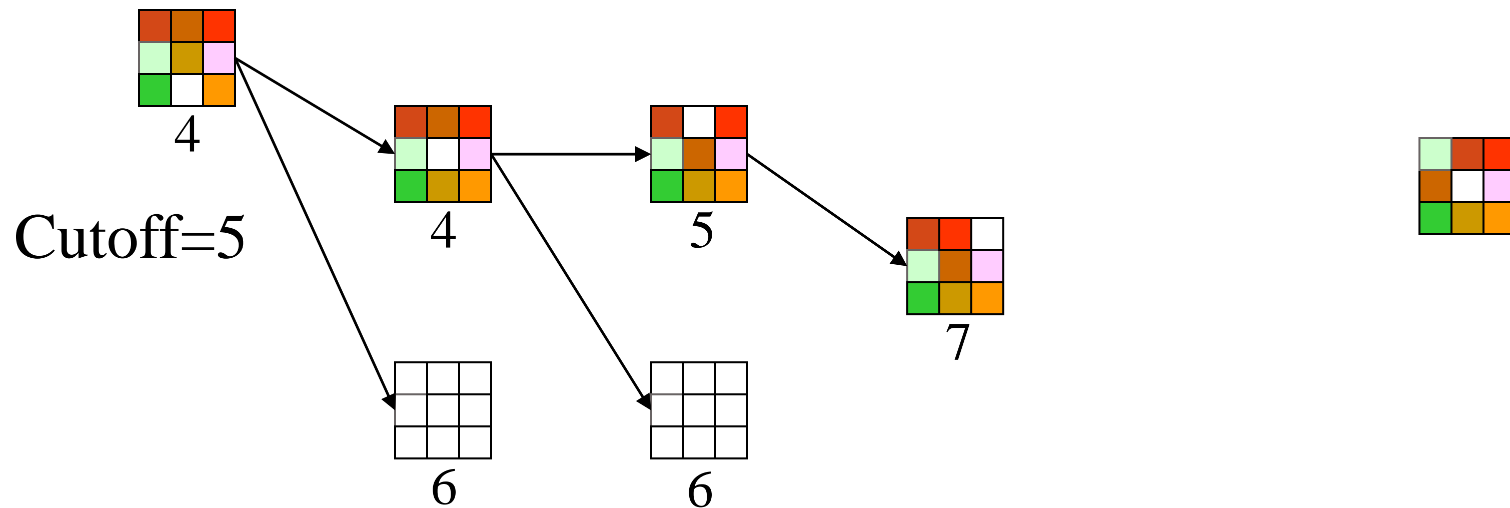
8-Puzzle

$f(N) = g(N) + h(N)$
with $h(N)$ = number of misplaced tiles



8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles

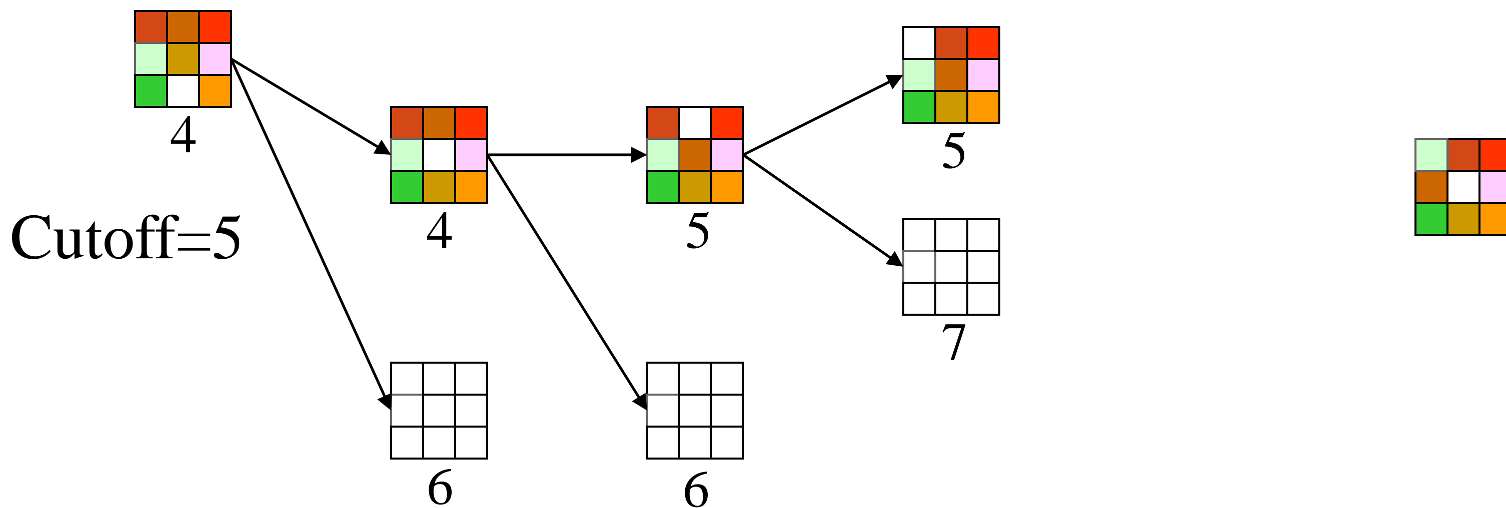


Aziz M. Qaroush - Birzeit University

8-Puzzle

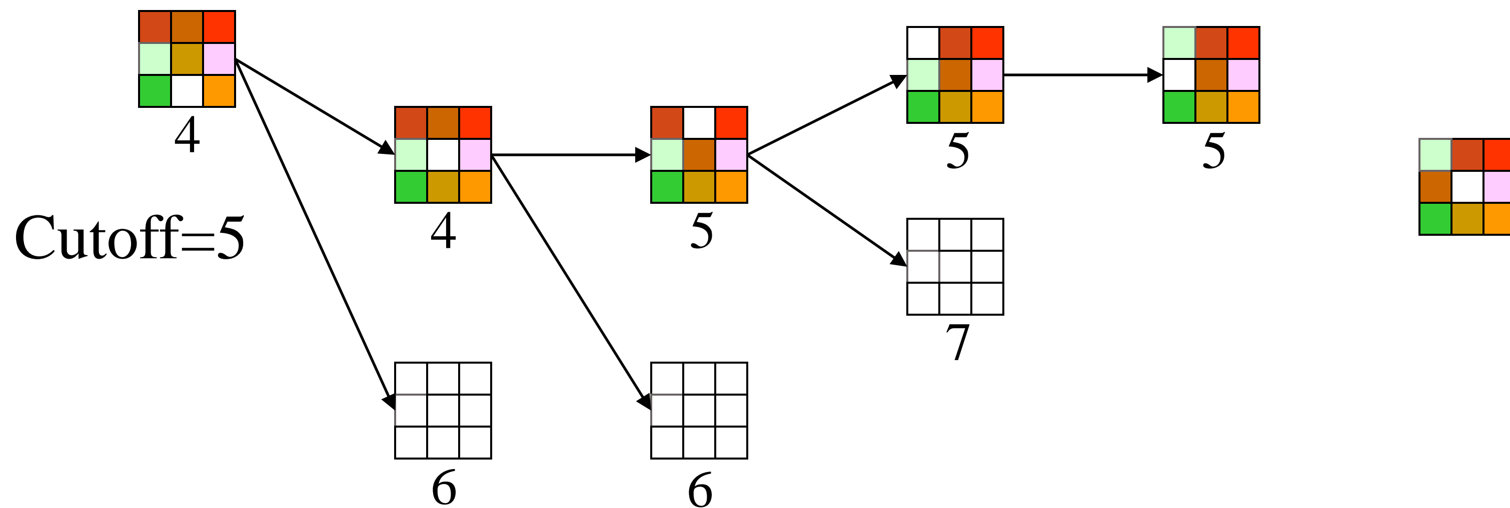
$f(N) = g(N) + h(N)$

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



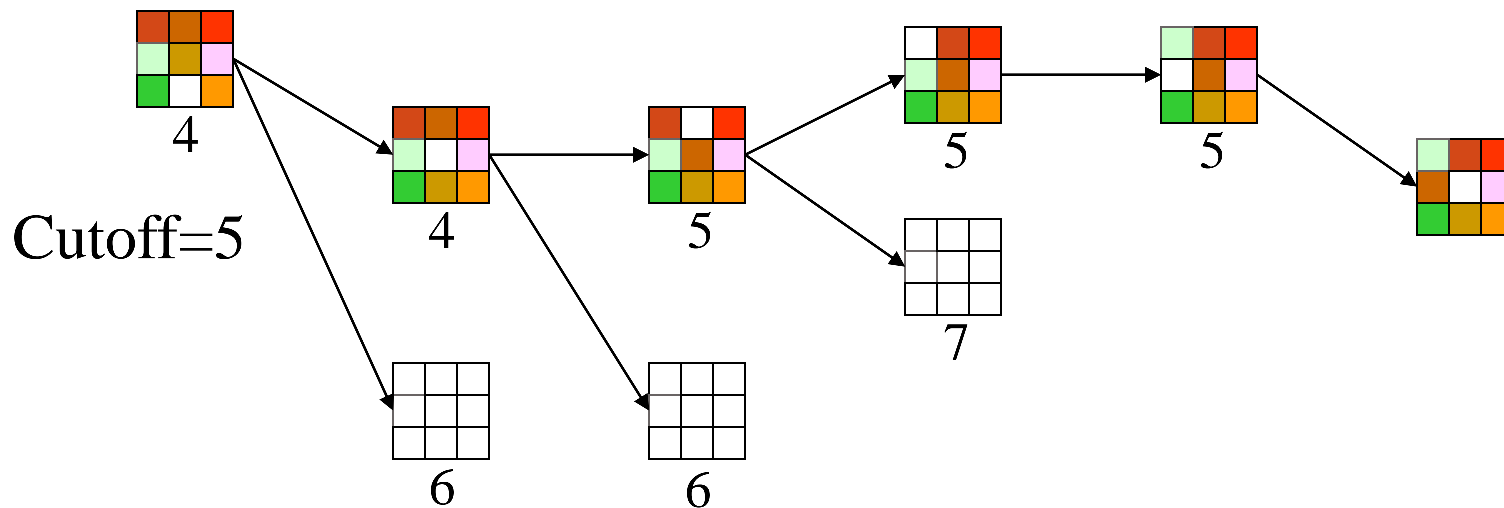
8-Puzzle

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



Aziz M. Qaroush - Birzeit University

Simple Recursive Best-First Search

- **Keeps track of the f-value of the best-alternative path available.**
  - If current f-values exceeds this alternative f-value then backtrack to alternative path.
  - Upon backtracking change f-value to best f-value of its children.
- It takes 2 arguments:
  - a node
  - an upper bound
  - Upper bound= min (upper bound on it's parent, current value of it's lowest cost brother).
- It explores the sub-tree below the node as long as it contains child nodes whose costs do not exceed the upper bound.
- If the current node exceeds this limit, the recursion unwinds back to the alternative path.
- As the recursion unwinds, RBFS replaces the *f*-value of each node along the path with the best *f*-value of its children.

SRBFS -The Algorithm

```
SRBFS ( node: N ,bound B)
    IF f( N) > B RETURN f(n)
    IF N is a goal, EXIT algorithm
    IF N has no children, RETURN infinity
    FOR each child Ni of N, F[i]  := f(Ni)
        sort Ni and F[i] in increasing order of F[i]
    IF only one child, F[2] = infinity
    WHILE (F[1] ≤ B and f[1] < infinity)
        F[1]  := SRBFS (N1, MIN(B, F[2]))
        insert N1 and F[1] in sorted order
RETURN F[1]
```

Simplified Memory-Bounded A* (SMA*)

- **Idea**
  - Expand the best leaf (just like A*) until memory is full
  - When memory is full, drop the worst leaf node (the one with the highest f-value) to accommodate new node.
    - If all leaf nodes have the same f-value, SMA* deletes the oldest worst leaf and expanding the newest best leaf.
  - Avoids re-computation of already explored area
    - Keeps information about the best path of a "forgotten" subtree in its ancestor.
  - Complete if there is enough memory for the shortest solution path
  - Often better than A* and IDA*
    - Trade-off between time and space requirements

Partial Searching

☞ The searches covered so far are characterized as partial searches. Why?

❑ **Partial Searching:**

- Means, it looks through a set of nodes for shortest path to the goal state using a heuristic function.
- The heuristic function is an estimate, based on domain-specific information, of how close we are to a goal.
- **Nodes**: state descriptions, partial solutions
- **Edges**: action that changes state for some cost
- **Solution**: sequence of actions that change from the start to the goal state
- BFS, IDS, UCS, Greedy, A^*, etc.
- **Ok** for small search spaces that are often "toy world" problems
- **Not ok** for Hard problems requiring exponential time to find the optimal solution, i.e. Traveling Salesperson Problem (TSP)

Local Search and Optimization

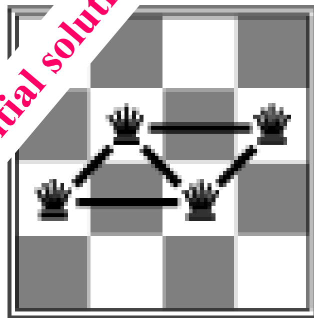
- **Previous searches:**
  - keep paths in memory, and remember alternatives so search can backtrack. Solution is a path to a goal.
  - Path may be irrelevant, if the final configuration only is needed (8-queens, IC design, network optimization, …)
- **Local Search:**
  - Use a single current state and move only to neighbors.
  - Use little space
  - Can find reasonable solutions in large or infinite (continuous) state spaces for which the other algorithms are not suitable
- **Optimization:**
  - Local search is often suitable for optimization problems.
  - Search for best state by optimizing an objective function.

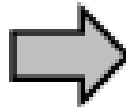
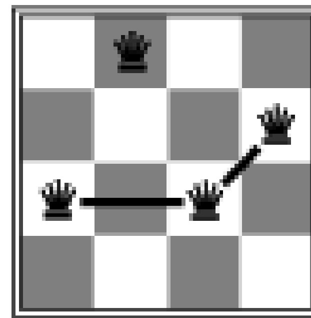
Local Search Methods

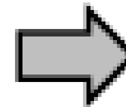
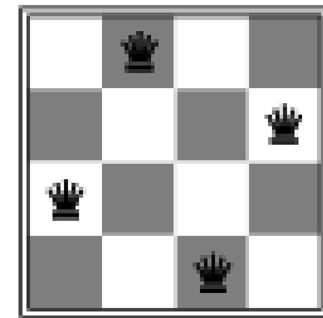
- Applicable when seeking Goal State & don't care how to get there. E.g.,
  - N-queens,
  - map coloring,
  - finding shortest/cheapest round trips **(TSP, VRP)**
  - finding models of propositional formulae **(SAT)**
  - *VLSI layout, scheduling, time-tabling, . . .*
  - *resource allocation*
  - *protein structure prediction*
  - *…………..*

Example: 4 Queen

- States: 4 queens in 4 columns (256 states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: h(n) = number of attacks

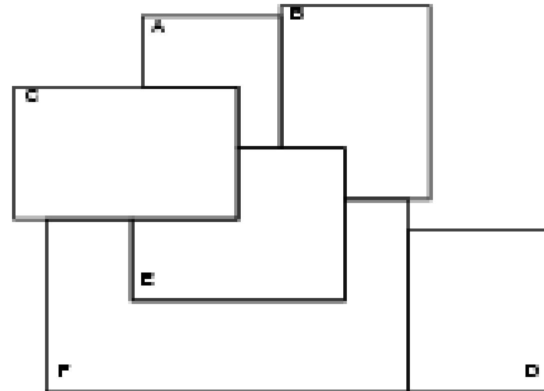


Example: Graph Coloring

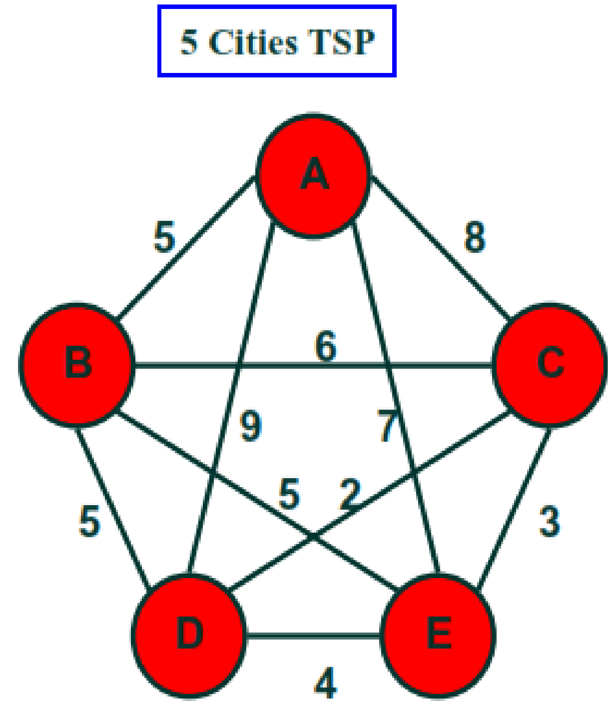
1. Start with random coloring of nodes
2. Change color of one node to reduce # of conflicts
3. Repeat 2



iteration	A	B	C	D	E	F	# conflicts	
1	b	g	g	r	b	r	2	{AE, DF}
2	b	g	g	B	b	r	1	{AE}
3	R	g	g	b	b	r	0	{}

Example: Traveling Salesperson Problem (TSP)

- **A salesman wants to visit a list of cities**
  - Stopping in each city only once
  - Returning to the first city
  - Traveling the shortest distance
- **Nodes are cities**
- **Arcs are labeled with distances between cities**

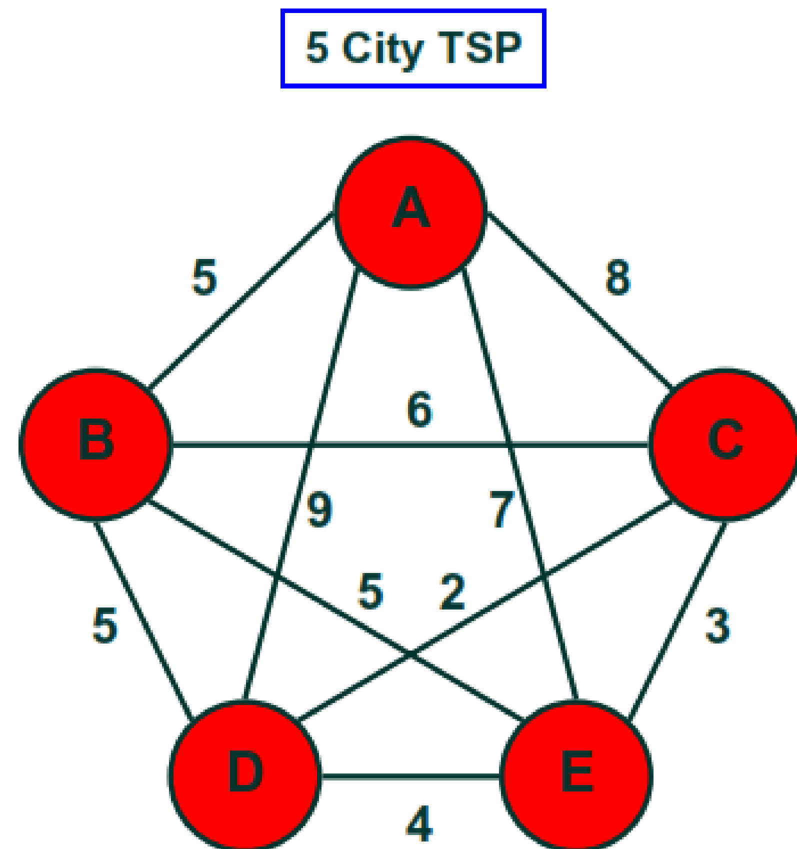


	A	B	C	D	E
A	0	5	8	9	7
B	5	0	6	5	5
C	8	6	0	2	3
D	9	5	2	0	4
E	7	5	3	4	0

Example: Traveling Salesperson Problem (TSP)

- **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**

	A	B	C	D	E
A	0	5	8	9	7
B	5	0	6	5	5
C	8	6	0	2	3
D	9	5	2	0	4
E	7	5	3	4	0



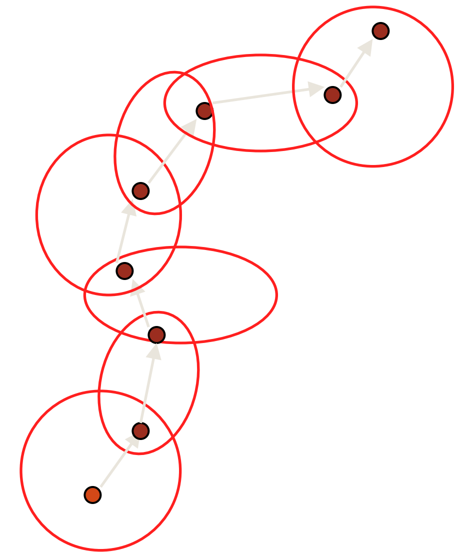
Local search

- Key idea (surprisingly simple):

1. Select (random) initial state (generate an initial guess)

2. Make local modification to improve current state (evaluate current state and move to other states)

3. Repeat Step 2 until goal state found (or out of time)



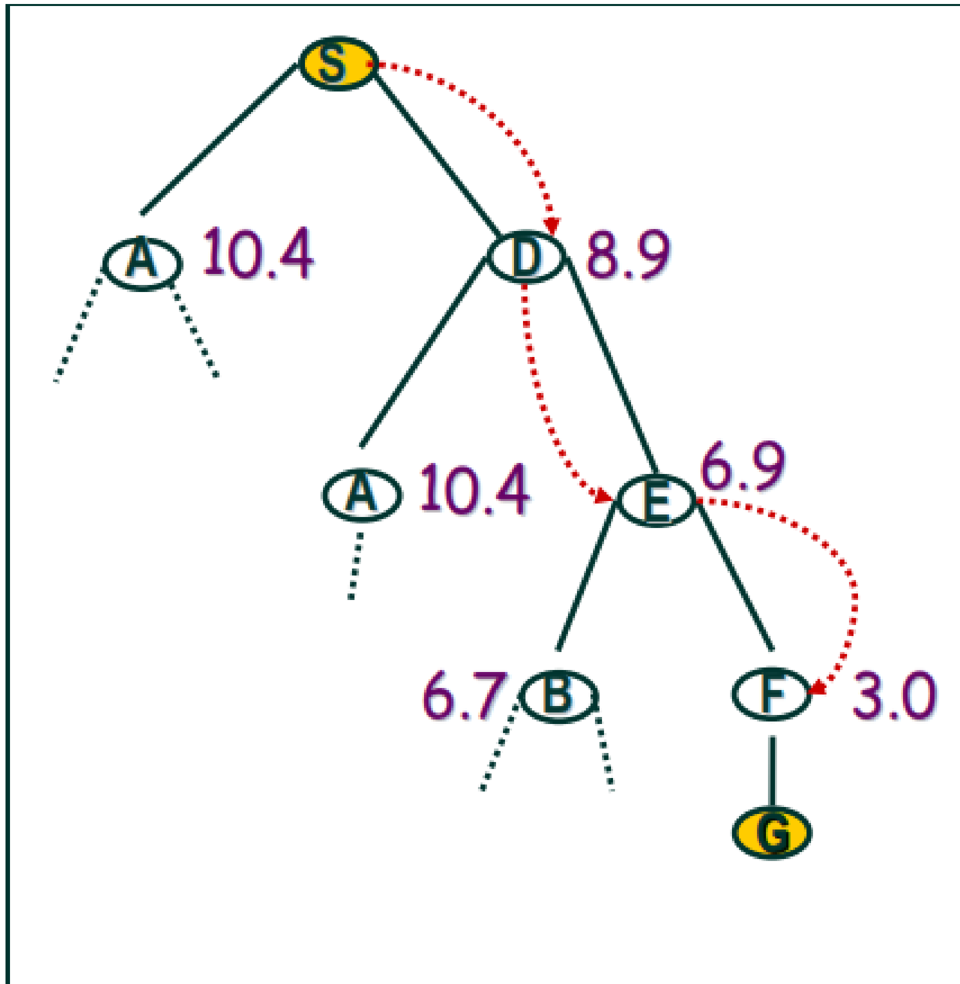
Local Search Algorithms

- **Basic idea:** Local search algorithms operate on a *single* state – current state – and move to one of its neighboring states.
- **The principle**: keep a single "current" state, try to improve it
- **Therefore**: Solution path needs not be maintained. Hence, the search is "local".
- **Two advantages**
  - Use little memory.
  - More applicable in searching large/infinite search space. They find reasonable solutions in this case.
- **Algorithms**
  - Hill Climbing
  - Local Beam Search
  - Genetics algorithms

Hill-Climbing Search

- **"Continuously moves in the direction of increasing value"**
  - It terminates when a peak is reached.
- Looks one step ahead to determine if any successor is better than the current state; if there is, move to the best successor.
- If there exists a successor s for the current state n such that
  - $h(s) < h(n)$
  - $h(s) <= h(t)$ for all the successors t of n,
- Then move from n to s. Otherwise, halt at n.
- Similar to Greedy search in that it uses h, but does not allow backtracking or jumping to an alternative path since it doesn't "remember" where it has been.
- Not complete since the search will terminate at "local minima," "plateaus," and "ridges."

Hill-Climbing Search



- Perform depth-first, BUT:
  instead of left-to-right selection,
- FIRST select the child with the best heuristic value

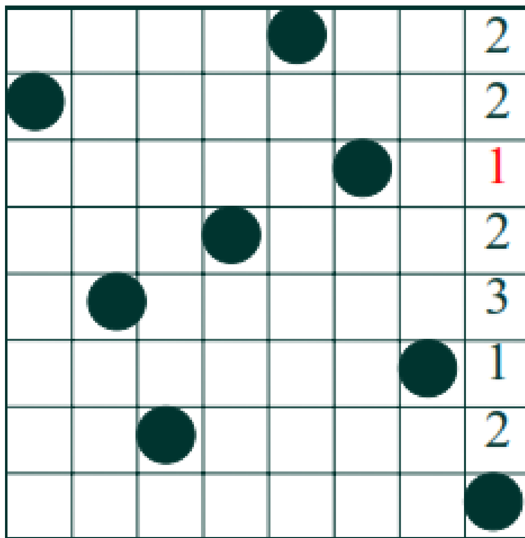
1. Pick a random point in the search space
2. Consider all the neighbors of the current state
3. Choose the neighbor with the best quality and move to that state
4. Repeat 2 thru 4 until all the neighboring states are of lower quality
5. Return the current state as the solution state

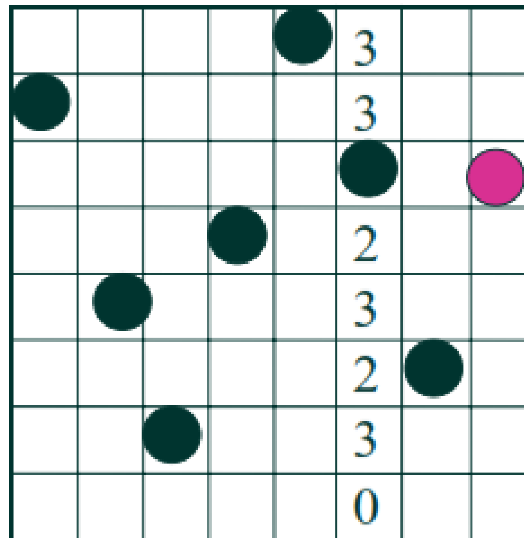
8 Queens Example

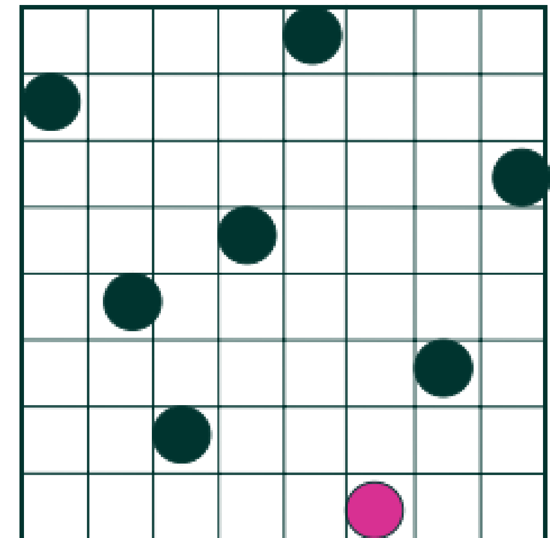
Move the queen in this column

Move the queen in this column

Done!

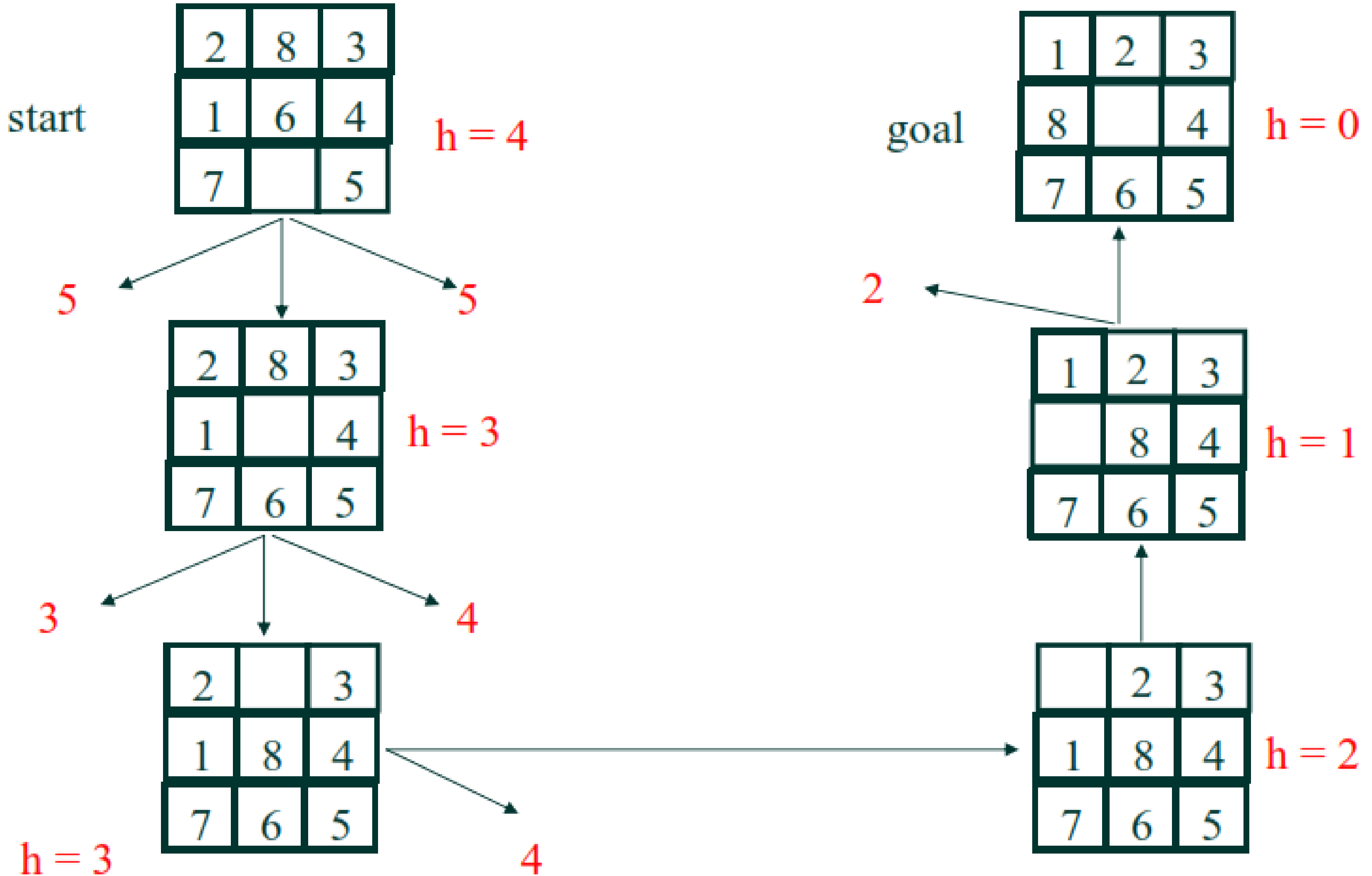






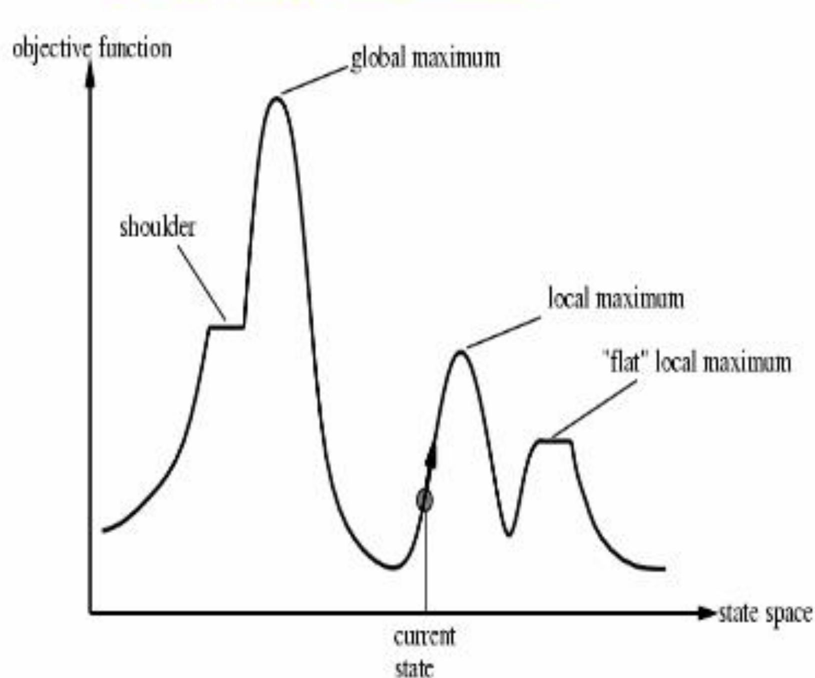
The numbers give the number of conflicts. Choose a move with the lowest number of conflicts. Randomly break ties. Hill descending.

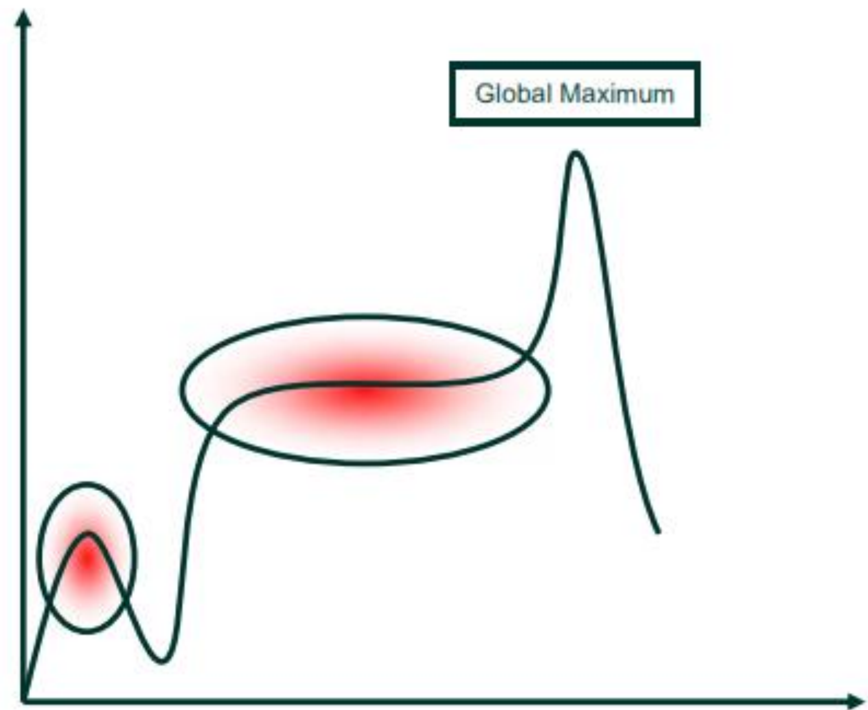
8 Puzzle Example



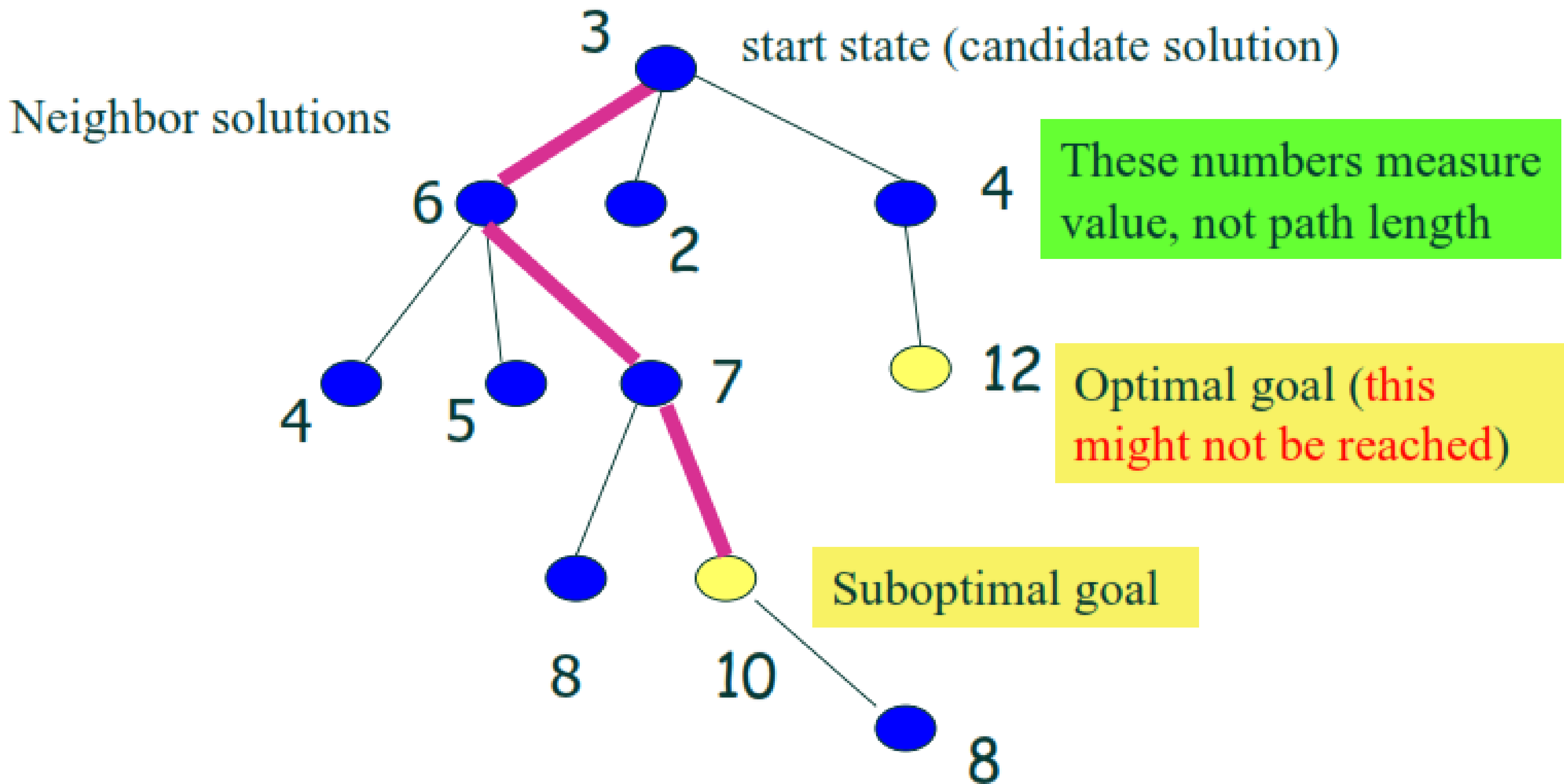
Drawbacks of Hill Climbing

- **Problems:**
  - **Local Maxima**: peaks that aren't the highest point in the space: No progress
  - **Plateaus:** the space has a broad flat region that gives the search algorithm no direction (random selection to solve)
  - **Ridges:** flat like a plateau, but with drop-offs to the sides; Search may oscillate from side to side, making little progress
- **Remedy:**
  - Introduce randomness





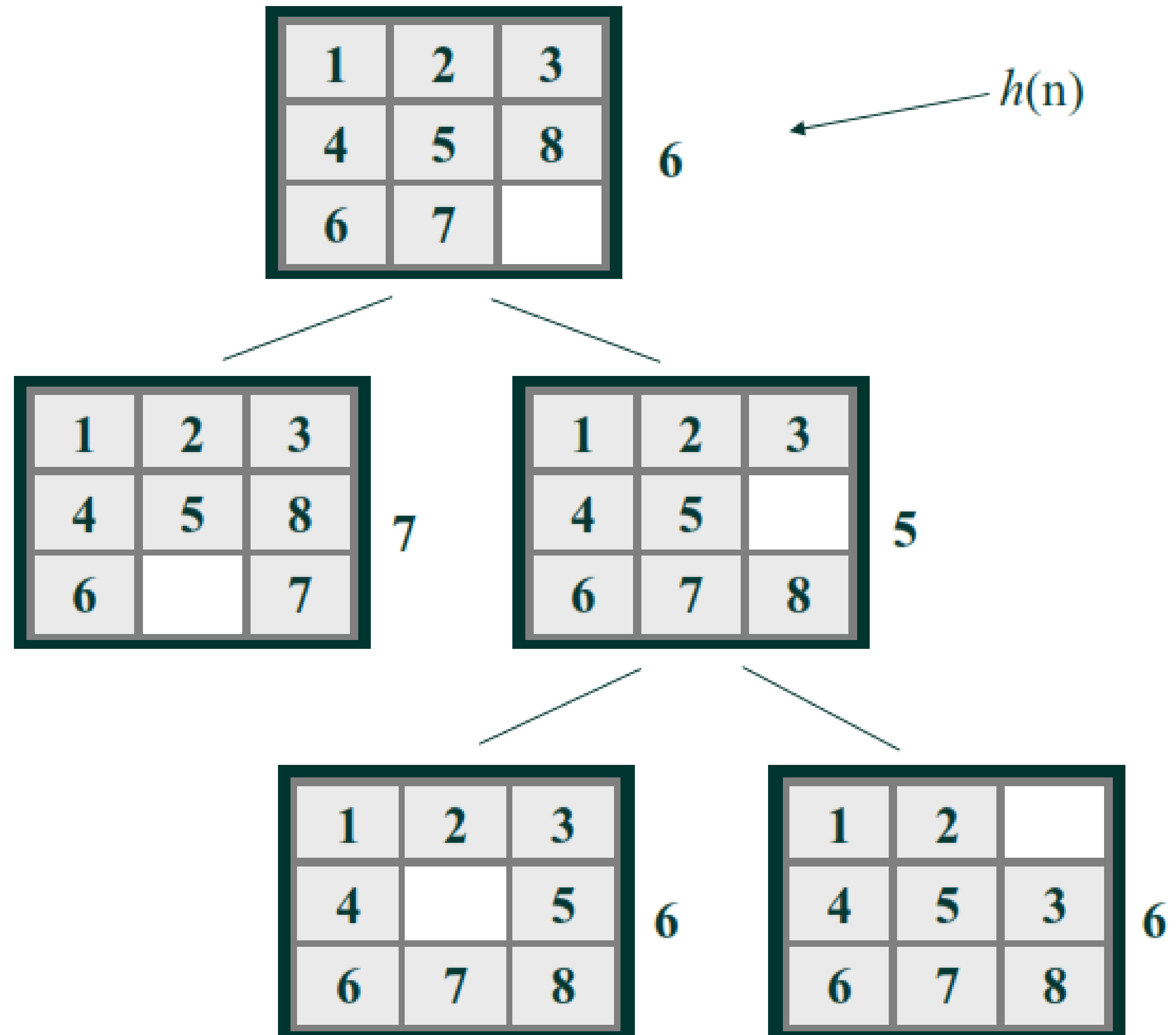
Drawbacks of Hill Climbing



Drawbacks of Hill Climbing

In this example, hill climbing does not work!

All the nodes on the fringe are taking a step "backwards" (local minima)

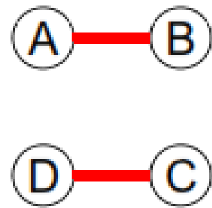


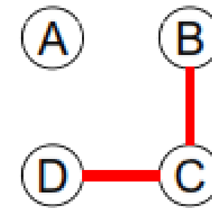
Local Beam Search

- The idea is that you just keep around those states that are relatively good, and just forget the rest.
- **Local beam search: somewhat similar to Hill Climbing:**
  - Start from **N** initial states.
  - Expand all **N** states and keep **k** best successors.
- This can avoid some local optima, but not always. It is most useful when the search space is big and the local optima aren't too common.
- **Local Beam Search Algorithm**
- Keep track of *k* states instead of one
  - Initially: *k* random states
  - Next: determine all successors of *k* states
    - Extend **all paths** one step
    - Reject all paths with loops
    - Sort all paths in queue by **estimated** distance to goal
  - If any of successors is goal → finished
  - Else select *k* best from successors and repeat.

Local Beam Search: Example

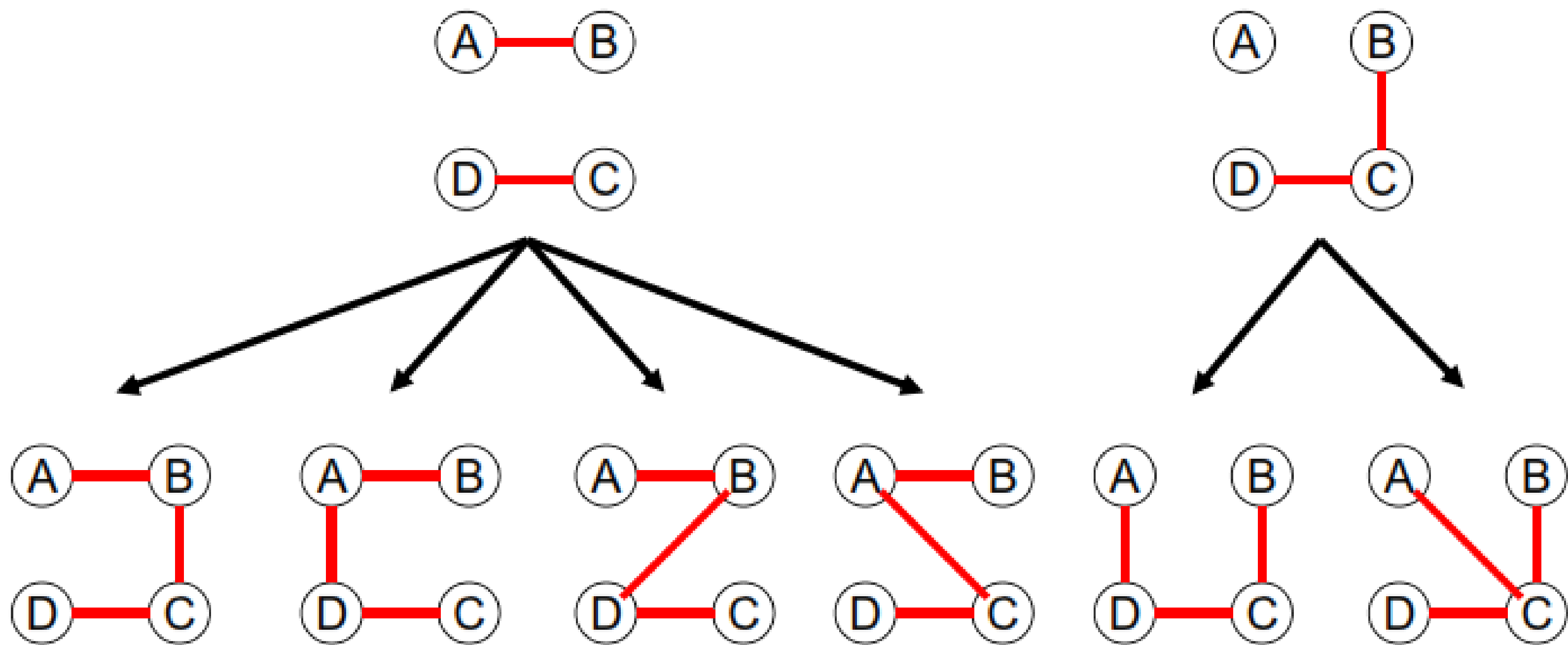
Travelling Salesman Problem



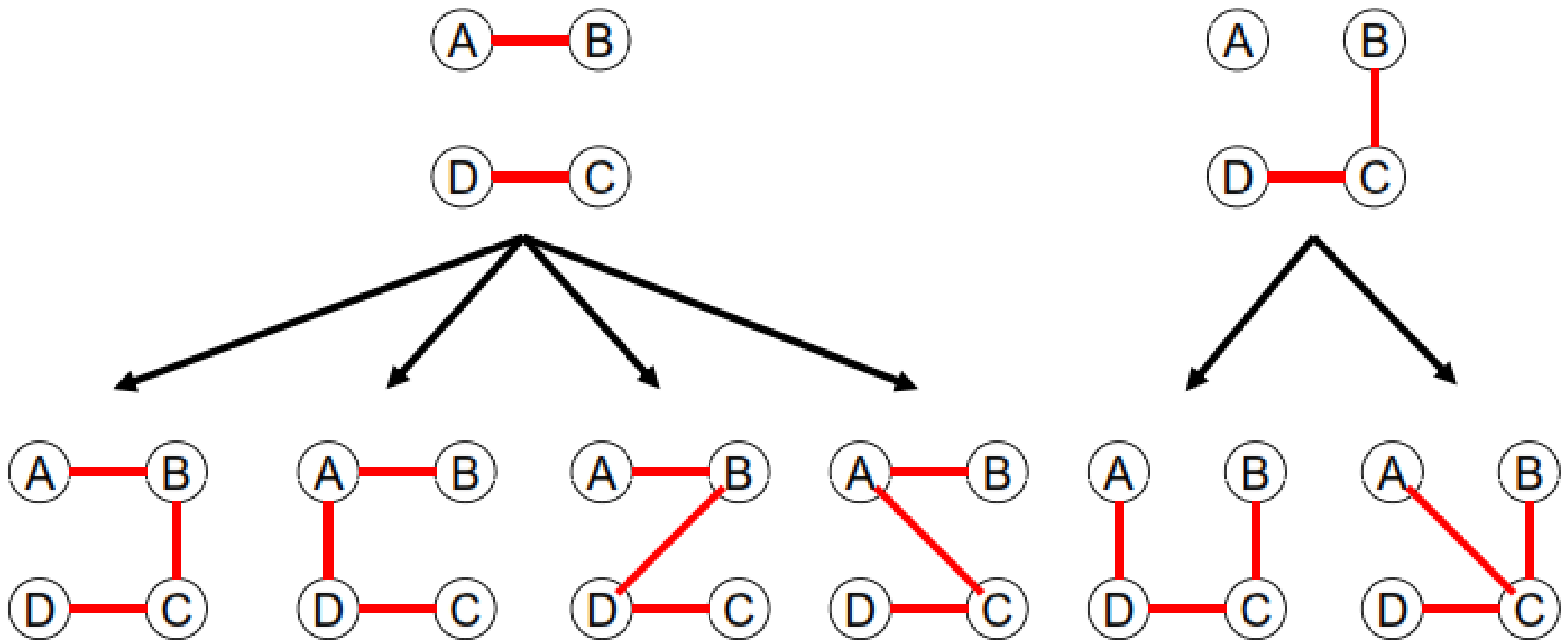


Keeps track of k states rather than just 1. k=2 in this example. Start with k randomly generated states.

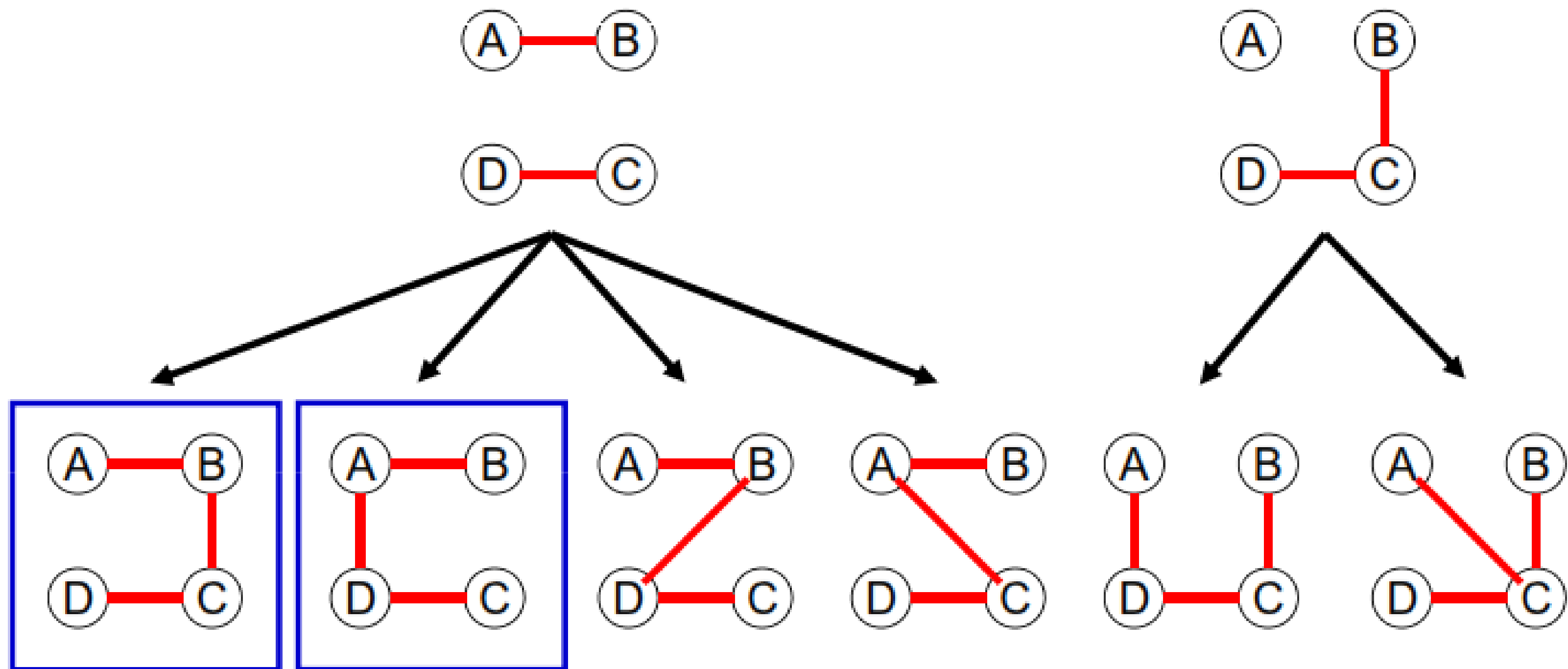
Local Beam Search: Example



Local Beam Search: Example



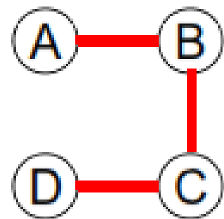
Local Beam Search: Example

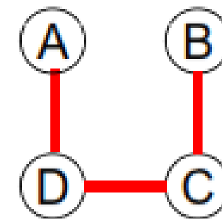


Select the best k successors from the **complete** list

Local Beam Search: Example

Travelling Salesman Problem (k=2)





Repeat the process until goal found

Genetic Algorithms

- An *algorithm* is a set of instructions that is repeated to solve a problem.
- A *genetic algorithm* conceptually follows steps inspired by the biological processes of evolution.
- Genetic Algorithms follow the idea of **SURVIVAL OF THE FITTEST**- Better and better solutions evolve from previous generations until a near optimal solution is obtained.
- A genetic algorithm is an iterative procedure that represents its candidate solutions as strings of genes called chromosomes.
- Genetic Algorithms are often used to improve the performance of other AI methods such as expert systems or neural networks.

Genetic Algorithms: Basic Terminology

1. Chromosomes: Chromosome means a candidate solution to a problem and is encoded as a string of bits.
2. Genes: a chromosome can be divided into functional blocks of DNA, genes, which encode traits, such as eye color. A different settings for a trait (blue, green, brown, etc.) are called alleles. Each gene is located at a particular position, called a locus, on the chromosome. Genes are single bits or short blocks of adjacent bits.
3. Genome: the complete collection of chromosomes is called the organism's genome.

- Population: A set of Chromosomes (Collection of Solutions)

1. Genotype: a set of genes contained in a genome.
2. Crossover (or recombination): occurs when two chromosomes bump into one another exchanging chunks of genetic information, resulting in an offspring.
3. Mutation: offspring is subject to mutation, in which elementary bits of DNA are changed from parent to offspring. In GAs, crossover and mutation are the two most widely used operators.
4. Fitness/Evaluation Function: the probability that the sates will live to reproduce.

Genetic Algorithms: Basic Terminology

- Before we can apply Genetic Algorithm to a problem, we need to answer:

  - How is an individual represented?

  - What is the fitness function?

  - How are individuals selected?

  - How do individuals reproduce?

Representing an Individual

- An individual is data structure representing the "genetic structure" of a possible solution.
- Genetic structure consists of an alphabet (usually 0,1)

- **Binary Encoding**
  - Most Common – string of bits, 0 or 1.

    Chrom: A = 1 0 1 1 0 0 1 0 1 1

    Chrom: B = 1 1 1 1 1 1 0 0 0 0
  - Gives you many possibilities
  - Example Problem: Knapsack problem
  - The problem: there are things with given value and size. The knapsack has given capacity. Select things to maximize the values.
  - Encoding: Each bit says, if the corresponding thing is in the knapsack

Representing an Individual

- **Permutation Encoding**
  - Used in “ordering problems”
  - Every chromosome is a string of numbers, which represents number is a sequence.

    Chrom A: 1 5 3 2 6 4 7 9 8

    Chrom B: 8 5 7 7 2 3 1 4 9
  - Example: Travelling salesman problem
  - The problem: cities that must be visited.
  - Encoding says order of cities in which salesman willl visit.

How offspring are produced

- *Reproduction*- Through **reproduction**, genetic algorithms produce new generations of improved solutions by selecting parents with higher fitness ratings or by giving such parents a greater probability of being contributors and by using random selection
- *Crossover*- Many genetic algorithms use strings of binary symbols for chromosomes, as in our Knapsack example, to represent solutions. **Crossover** means choosing a random position in the string (say, after 2 digits) and exchanging the segments either to the right or to the left of this point with another string partitioned similarly to produce two new off spring.
- *Mutation*- **Mutation** is an arbitrary change in a situation. Sometimes it is used to prevent the algorithm from getting stuck. The procedure changes a 1 to a 0 to a 1 instead of duplicating them. This change occurs with a very low probability (say 1 in 1000)

Crossover Example 1

- **Parent A 011011**
- **Parent B 101100**
- "Mate the parents by splitting each number as shown between the second and third digits (position is randomly selected)

  01*1011 **10*1100**

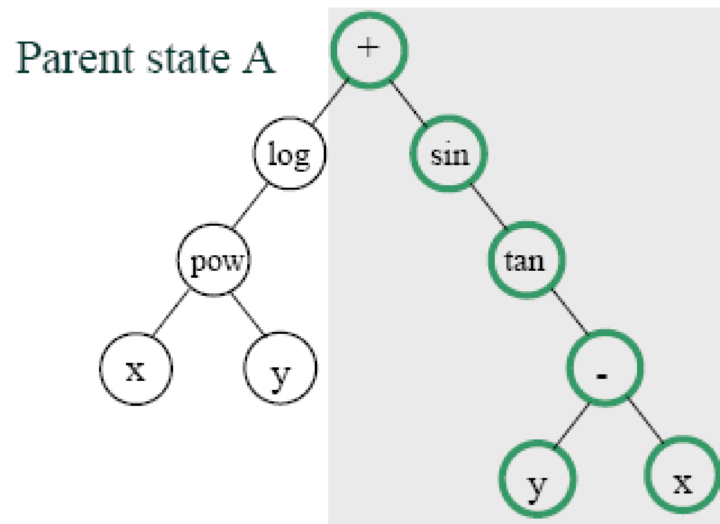
- Now combine the first digits of A with the last digits of B, and the first digits of B with the last digits of A
- This gives you two new offspring

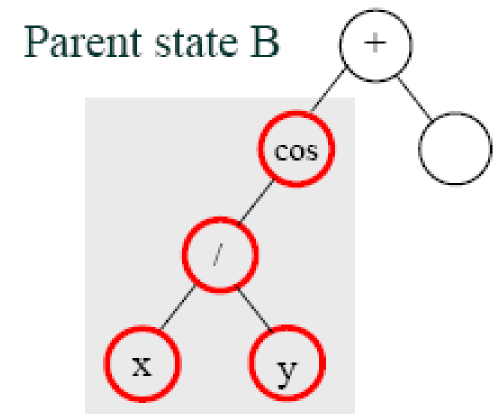
  011100 **101011**

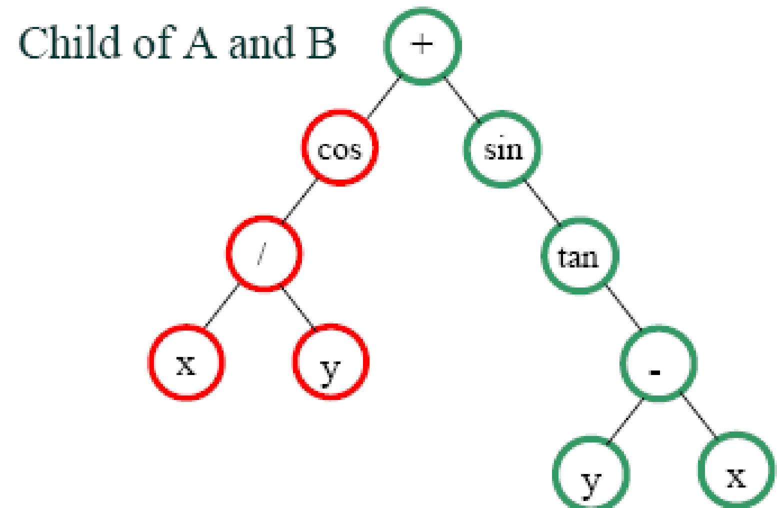
- If these new solutions, or offspring, are better solutions than the parent solutions, the system will keep these as more optimal solutions and they will become parents. This is repeated until some condition (for example number of populations or improvement of the best solution) is satisfied.

Crossover Example 2

Reproduction (Crossover)







Parent state A

10011101

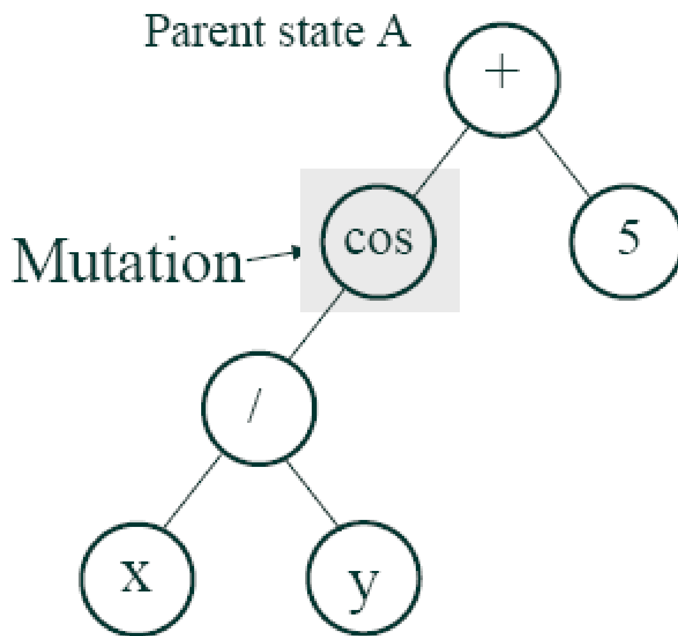
11101000

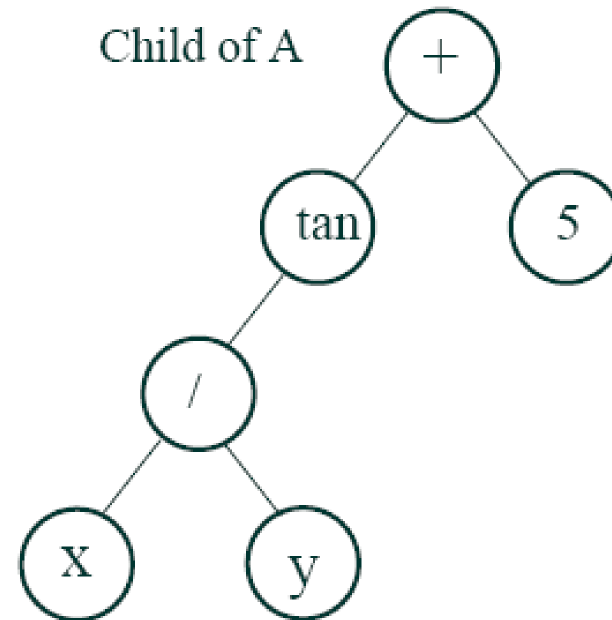
Parent state B

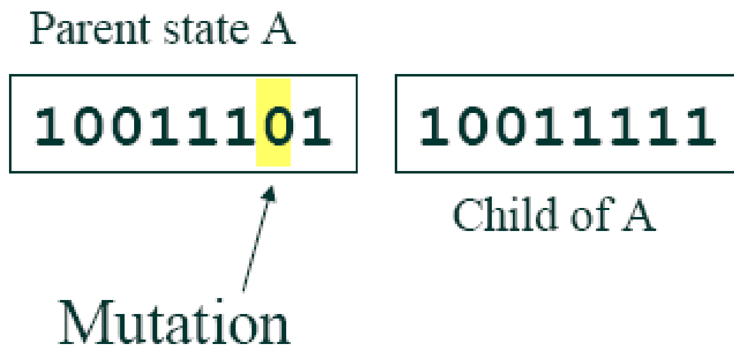
10011000

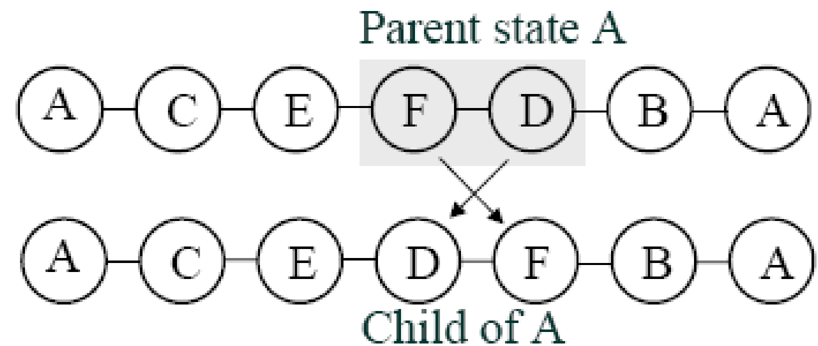
Child of A and B

Mutation Example



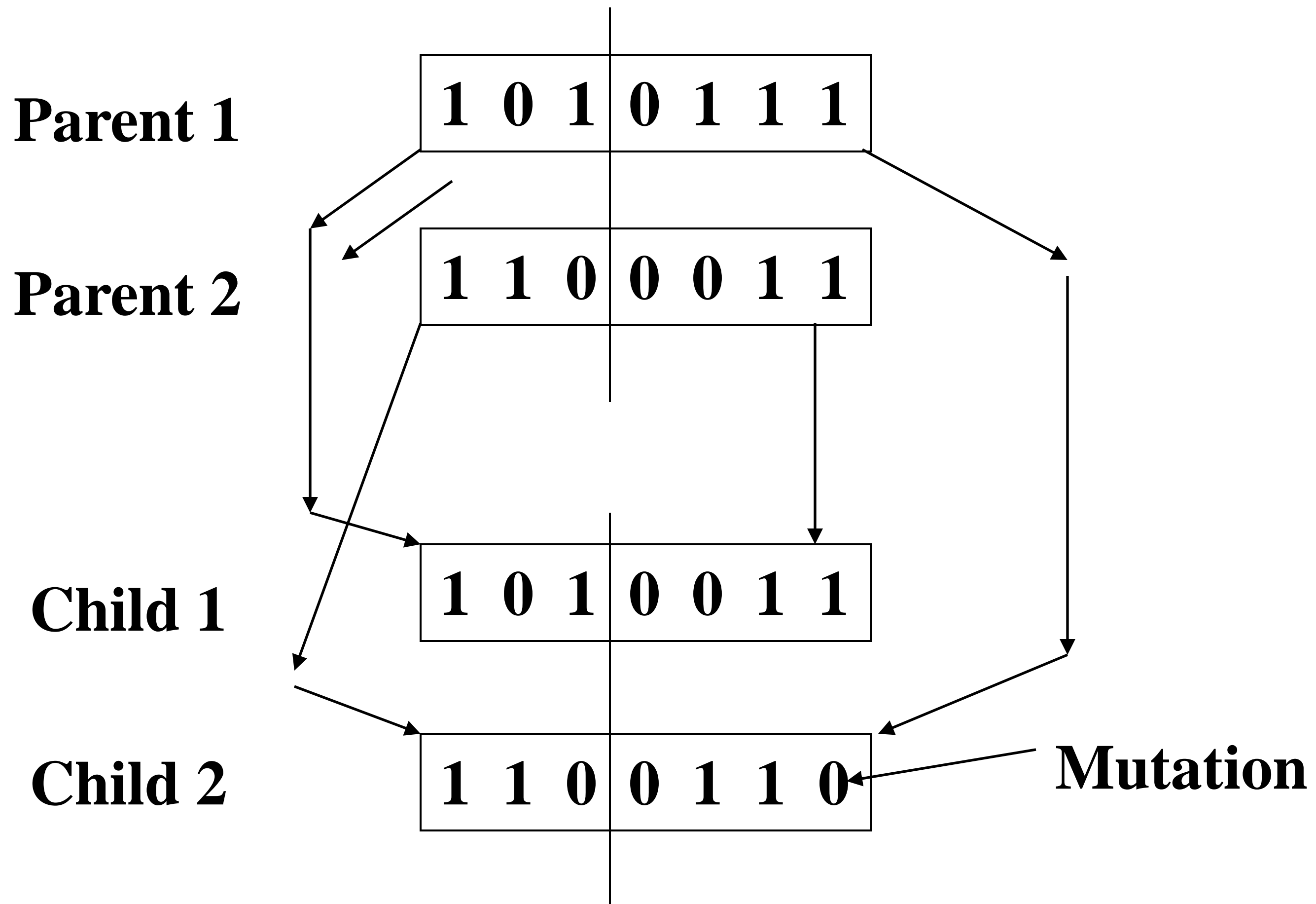






Aziz M. Qaroush - Birzeit University

Genetic Algorithm Operators
Mutation and Crossover



Selection Criteria

- Fitness proportionate selection, rank selection methods.
  - Fitness proportionate – each individual, *I,* has the *probability* *fitness(I)/sum_over_all_individual_j Fitness(j)*, where *Fitness(I)* is the fitness function value for individual *I.*
    - Represents a rank of the "representation"
    - It is usually a real number.
    - E.g. the length of the route in the traveling salesperson problem is a good measure, because the shorter the route, the better the solution
  - Rank selection – sorts individual by fitness and the probability that an individual will be selected is proportional to its rank in this sorted list.

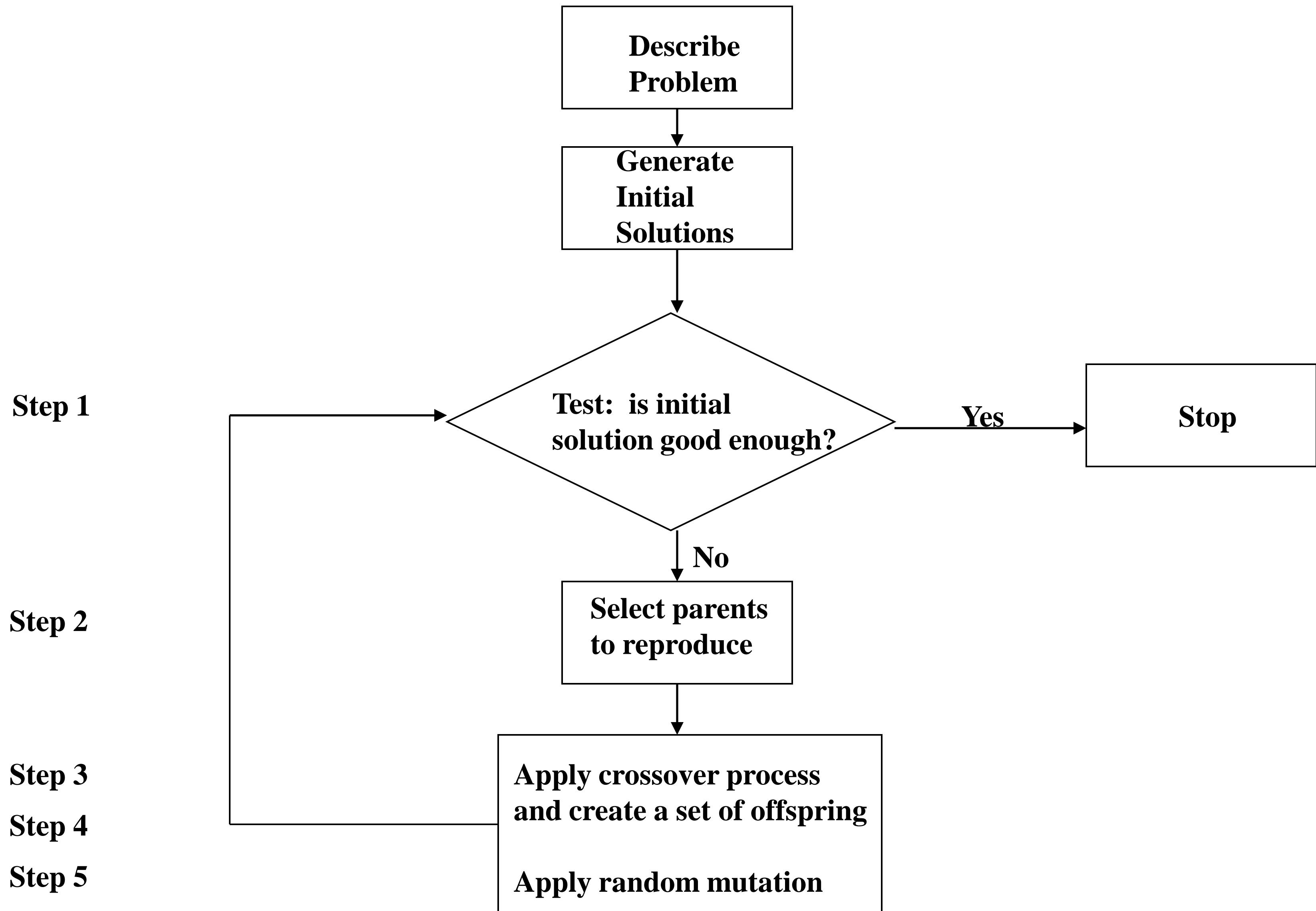
Outline of the Basic Genetic Algorithm

1. **[Start]** Generate random population of *n* chromosomes (suitable solutions for the problem)
2. **[Fitness]** Evaluate the fitness $f(x)$ of each chromosome x in the population
3. **[New population]** Create a new population by repeating following steps until the new population is complete
4. **[Selection]** Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected) The idea is to choose the better parents.

Outline of the Basic Genetic Algorithm

5. **[Crossover]** With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
6. **[Mutation]** With a mutation probability mutate new offspring at each locus (position in chromosome).
7. **[Accepting]** Place new offspring in a new population
8. **[Replace]** Use new generated population for a further run of algorithm
9. **[Test]** If the end condition is satisfied, **stop**, and return the best solution in current population
10. **[Loop]** Go to step 2

Flow Diagram of the Genetic Algorithm Process



Example: The Knapsack Problem

- You are going on an overnight hike and have a number of items that you could take along.
- Each item has a weight (in pounds) and a benefit or value to you on the hike(for measurements sake let's say, in US dollars), and you can take one of each item at most.
- There is a capacity limit on the weight you can carry (constraint).
- This problem only illustrates one constraint, but in reality there could be many constraints including volume, time, etc.

GA Example: The Knapsack Problem

Item:	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Benefit:	**5**	**8**	**3**	**2**	**7**	**9**	**4**
Weight:	**7**	**8**	**4**	**10**	**4**	**6**	**4**

- **Knapsack holds a maximum of 22 pounds**
- **Fill it to get the maximum benefit**
- **Solutions take the form of a string of 1's and 0's. Also known as strings of genes called Chromosomes**
  - **0101010**
  - **1101100**
  - **0100111**

Example: The Knapsack Problem

- We represent a solution as a string of seven 1s and 0s and the fitness function as the total benefit, which is the sum of the gene values in a string solution times their representative benefit coefficient.
- The method generates a set of random solutions (initial parents), uses total benefit as the fitness function and selects the parents randomly to create generations of offspring by crossover and mutation.
- Possible solutions generated by the system using *Reproduction, Crossover*, or *Mutations*
  - 0101010
  - 1101100
  - 0100110

Knapsack Example
Solution 1

Item	1	2	3	4	5	6	7
Solution	**0**	**1**	**0**	**1**	**0**	**1**	**0**
Benefit	5	8	3	2	7	9	4
Weight	7	8	4	10	4	6	4

- Benefit $8 + 2 + 9 = 19$
- Weight $8 + 10 + 6 = 24$

Knapsack Example Solution 2

Item	1	2	3	4	5	6	7
Solution	**1**	**1**	**0**	**0**	**1**	**0**	**0**
Benefit	5	8	3	2	7	9	4
Weight	7	8	4	10	4	6	4

- Benefit 5 + 8 + 7 = 20
- Weight 7 + 8 + 4 = 19

Knapsack Example Solution 3

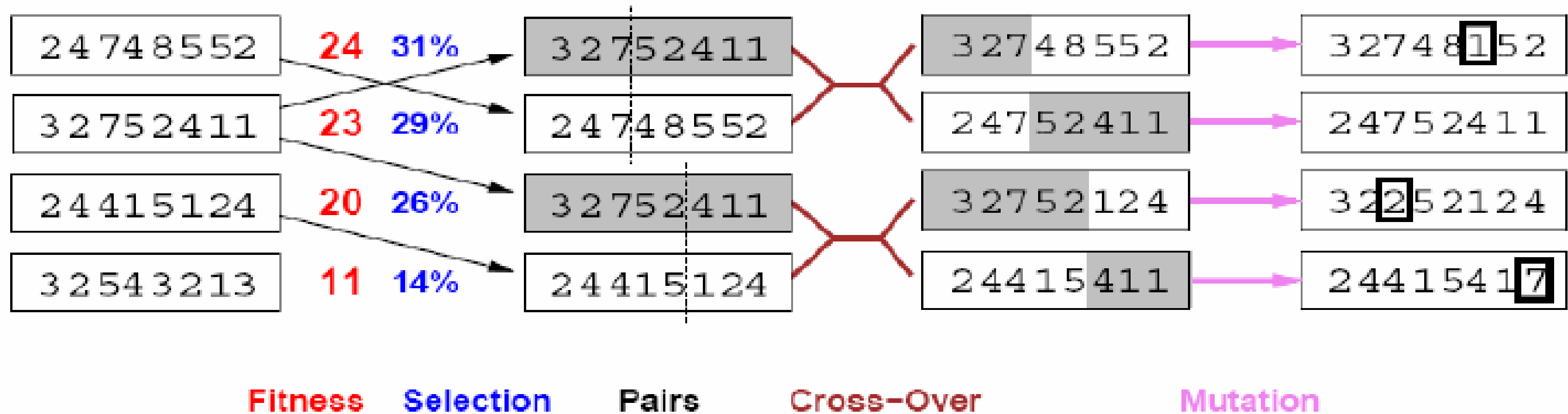
Item	1	2	3	4	5	6	7
Solution	**0**	**1**	**0**	**0**	**1**	**1**	**1**
Benefit	5	8	3	2	7	9	4
Weight	7	8	4	10	4	6	4

- Benefit 8 + 7 + 9 + 4 = 28
- Weight 8 + 4 + 6 + 4 = 22

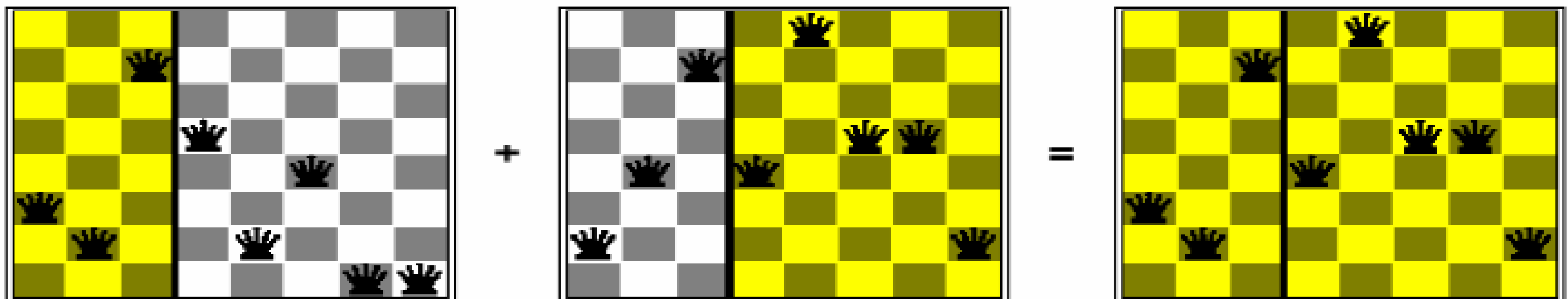
Knapsack Example

- Solution 3 is clearly the best solution and has met our conditions, therefore, item number 2, 5, 6, and 7 will be taken on the hiking trip. We will be able to get the most benefit out of these items while still having weight equal to 22 pounds.
- This is a simple example illustrating a genetic algorithm approach.

8 Queen Example



- Fitness function: number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)
- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29% etc



Summery

- Best-first search = general search, where the minimum-cost nodes (according to some measure) are expanded first.
- Greedy search = best-first with the estimated cost to reach the goal as a heuristic measure.
  - Generally faster than uninformed search
  - Not optimal, Not complete.
- A* search = best-first with measure = path cost so far + estimated path cost to goal.
  - Combines advantages of uniform-cost and greedy searches
  - Complete, optimal and optimally efficient
  - Space complexity still exponential
- Time complexity of heuristic algorithms depend on quality of heuristic function. Good heuristics can sometimes be constructed by examining the problem definition or by generalizing from experience with the problem class.
- Iterative improvement algorithms keep only a single state in memory.
- Can get stuck in local extreme; simulated annealing provides a way to escape local extreme, and is complete and optimal given a slow enough cooling schedule.

Summery

- **Hill Climbing**
  1. Moves in the direction of increasing value.
  2. Terminates when it reaches a “peak”.
  3. Does not look beyond immediate neighbors
- **Local beam search:**
  1. Start with k randomly generated nodes.
  2. Generate k successors of each.
  3. Keep the best k out of the them.
- **Genetic algorithms:**
  1. Start with k randomly generated nodes called the population.
  2. Generate successors by combining pairs of nodes.
  3. Allow mutations.