Recitation 2
Search 02

Informed Search

Today

Formal Search Problem

- Informed Search
- Greedy best-first
- A*
- Heuristics
  - Relaxation
  - Abstraction

Local Search

- Gradient Descent
- Simulated Annealing
- Genetic Algorithms

Reminder – Formal Search Problem

Formal Search Problem

$\langle S, s_0, G, A, F, C \rangle$

S is a set of **states**

$s_0 \in S$ is a **start state**

$G \subset S$ is a set of **goal states**

A is a set of **actions**

$F: S \times A \to S$ is a **transition function**

$C: S \times A \to \mathbb{R}_{\geq 0}$ is the **cost functions**

A solution is a pair

$\langle \{s_i\}_{i=0}^{n}, \{a_i\}_{i=0}^{n-1} \rangle$

solution cost:

$\sum_{i=0}^{n-1} C(s_i, a_i)$

where

s_0 is the start state,

$s_n \in G$, and

for $\forall 1 \leq i \leq n$:

$s_i = F(s_{i-1}, a_{i-1})$

Formal Search Problem and Guidance

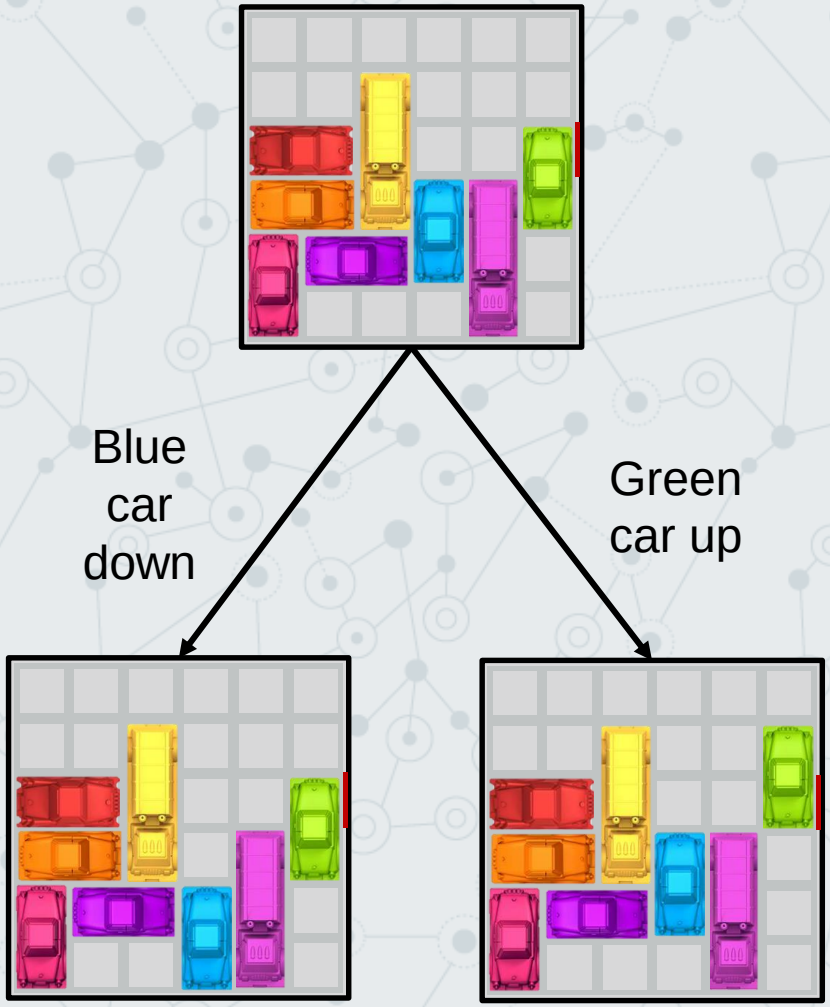
search strategy is defined by picking the order of node expansion from the $fringe$:

- Ordering of the $fringe$ - choose which node is more likely to lead to the goal
- Addition of new nodes to the $fringe$ - not all nodes are relevant to our search
- Reset of the $fringe$ - partial reset with heuristic update

Search Tree

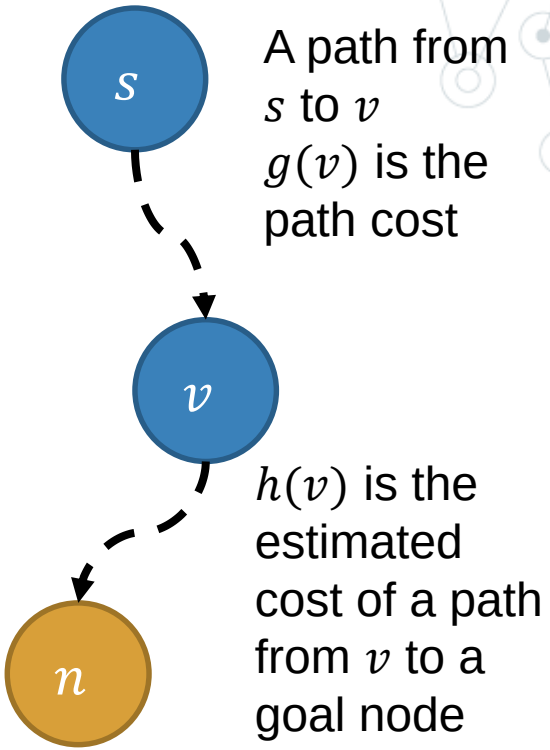
A **search tree**:

- This is a "what if" tree of plans and outcomes
- Start state at the root node
- Children correspond to successors
- Nodes contain states, correspond to PLANS to those states
- **For most problems, we never actually build the whole tree**



Evaluation Function

- $g(v)$: exact cost of reaching v from the root node
- $h(v)$: estimated cost of reaching a goal node from v. This is called a heuristic function
- $f\ (v) = g(v) + h(v)$: estimated cost of a solution (path from root to goal) that contains v

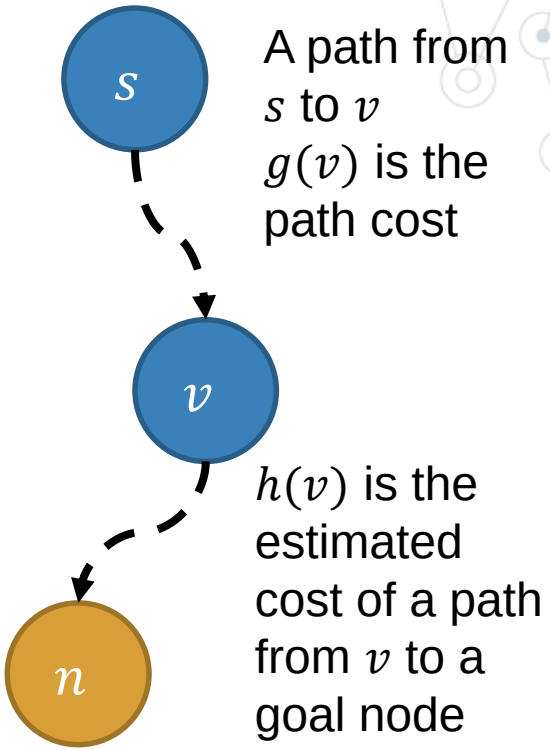


Evaluation Function More Formally

Given a search trace (V, E):

- Given step cost function
$$C: E \to \mathbb{R}_{\geq 0}$$
then for the path $(v_0, \ldots, v_n)$:
$g(v_n) = \sum_{i=1}^{n} C(e_i),\ e_i = (v_{i-1}, v_i)$
- $h: V \to \mathbb{R}_{\geq 0}$

- **Complete heuristic function**
$$f(v) = g(v) + h(v)$$



Best-first Search

Idea: use an evaluation function (there are many possibilities) for each node

- Estimate of "desirability", of various forms → Expand most desirable unexpanded node

Implementation: Order the nodes in the $fringe$ in decreasing order of desirability, e.g.:

- **Uniform-cost** search (uninformed) : uses $g(v)$
- **Greedy best-first** search : uses $h(v)$
- **A*** search : uses $f(v)$
- **Weighted-A*** search : uses $g(v) + w \cdot h(v)$,$w > 1$

Greedy Best-First Search

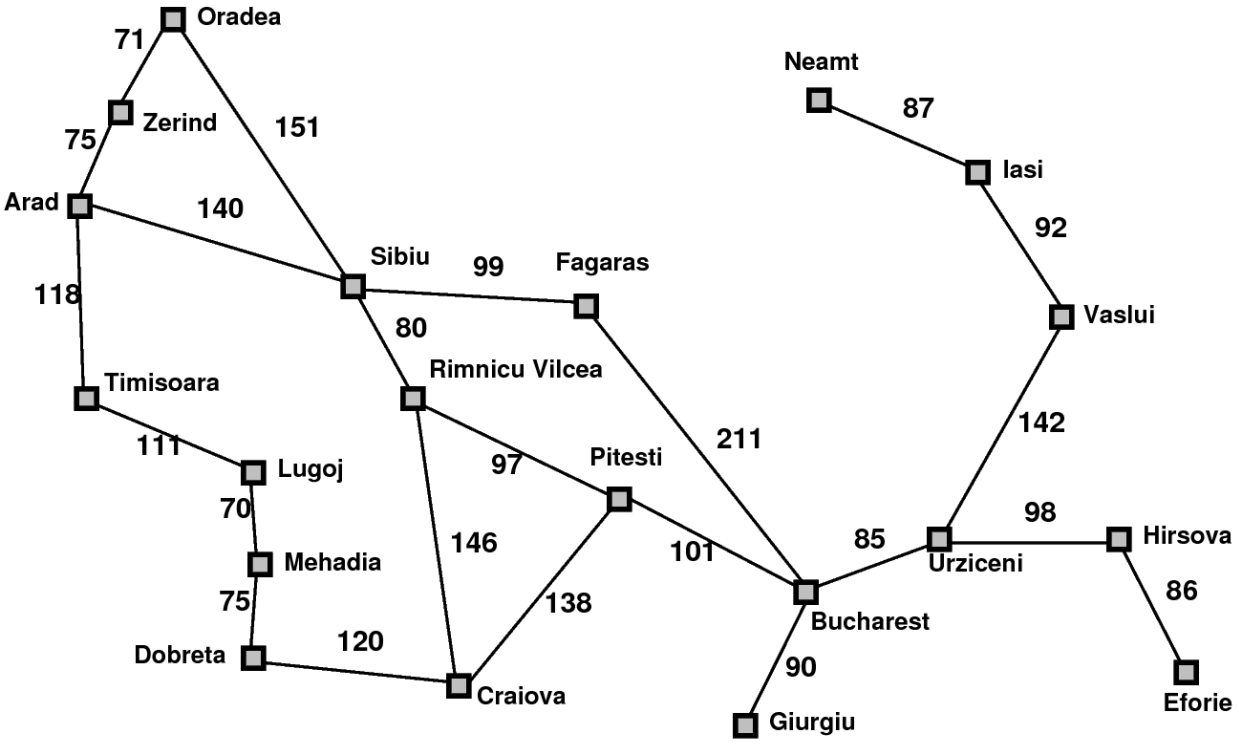
Evaluation function $h(v)$ (pure heuristic)

i.e. estimate of cost from current state v to $goal$

e.g., $h_{SLD}(v)$ = *StraightLineDistance* from v to $Bucharest$

Greedy best-first search expands the node that **appears** to be closest to goal

Example - Romania



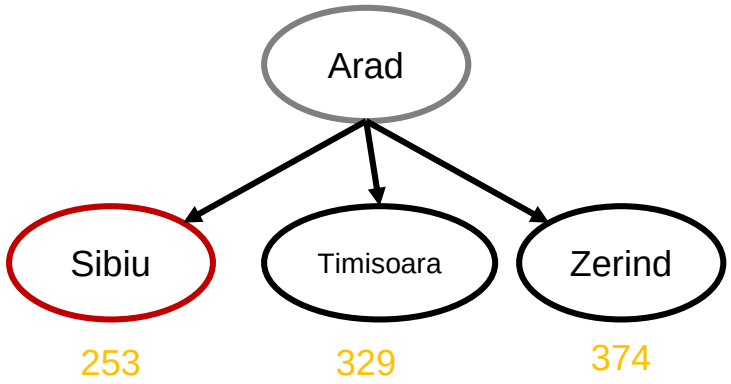
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Example - Romania

(a) The initial state

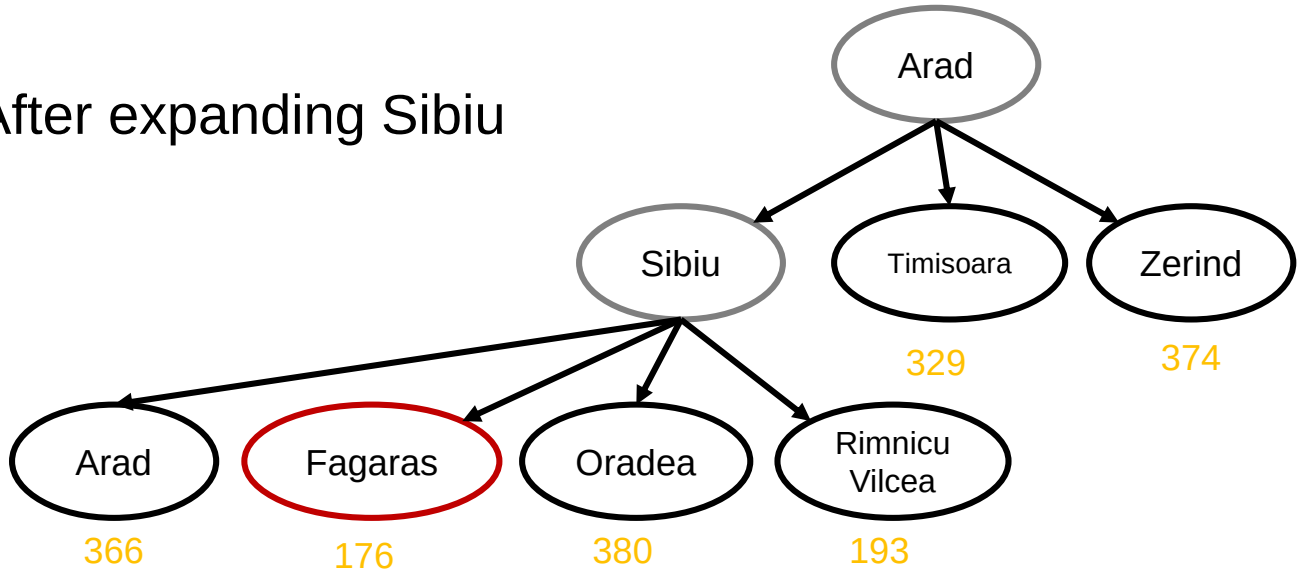
Example - Romania

(b) After expanding Arad



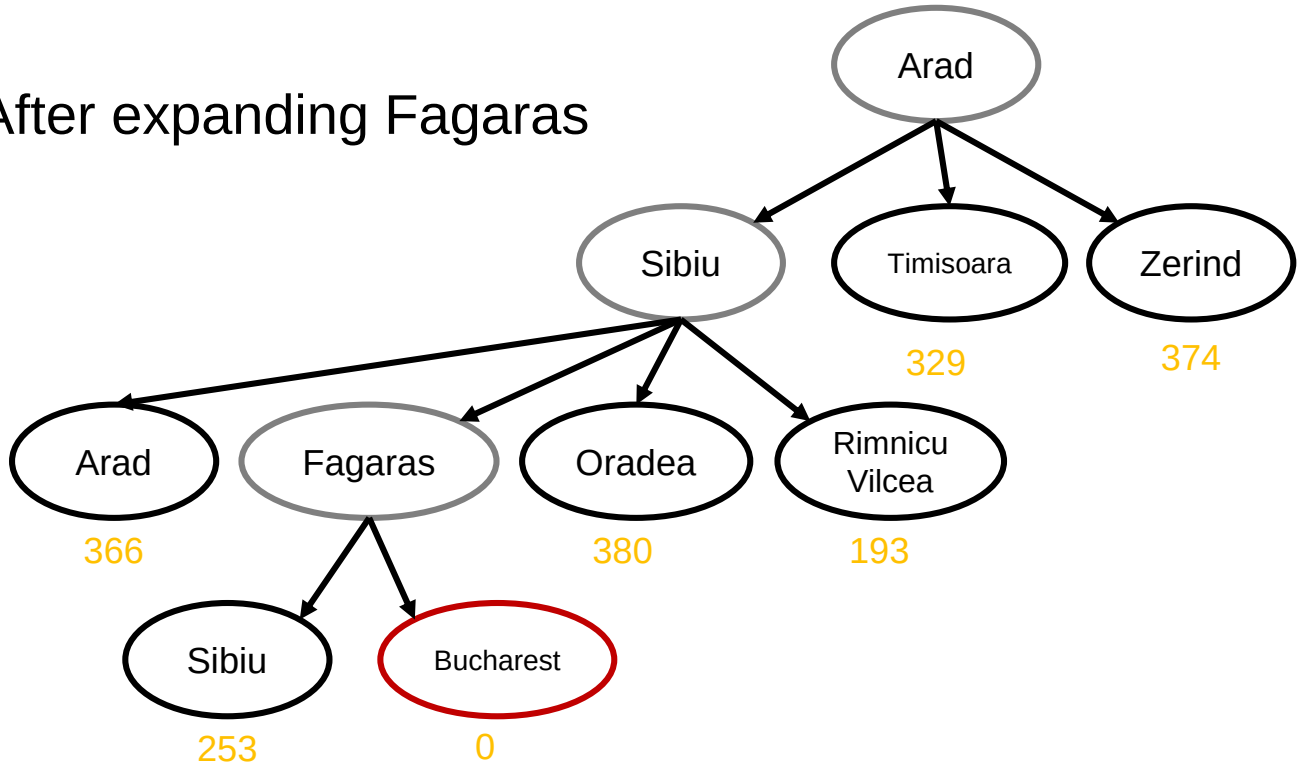
Example - Romania

(c) After expanding Sibiu

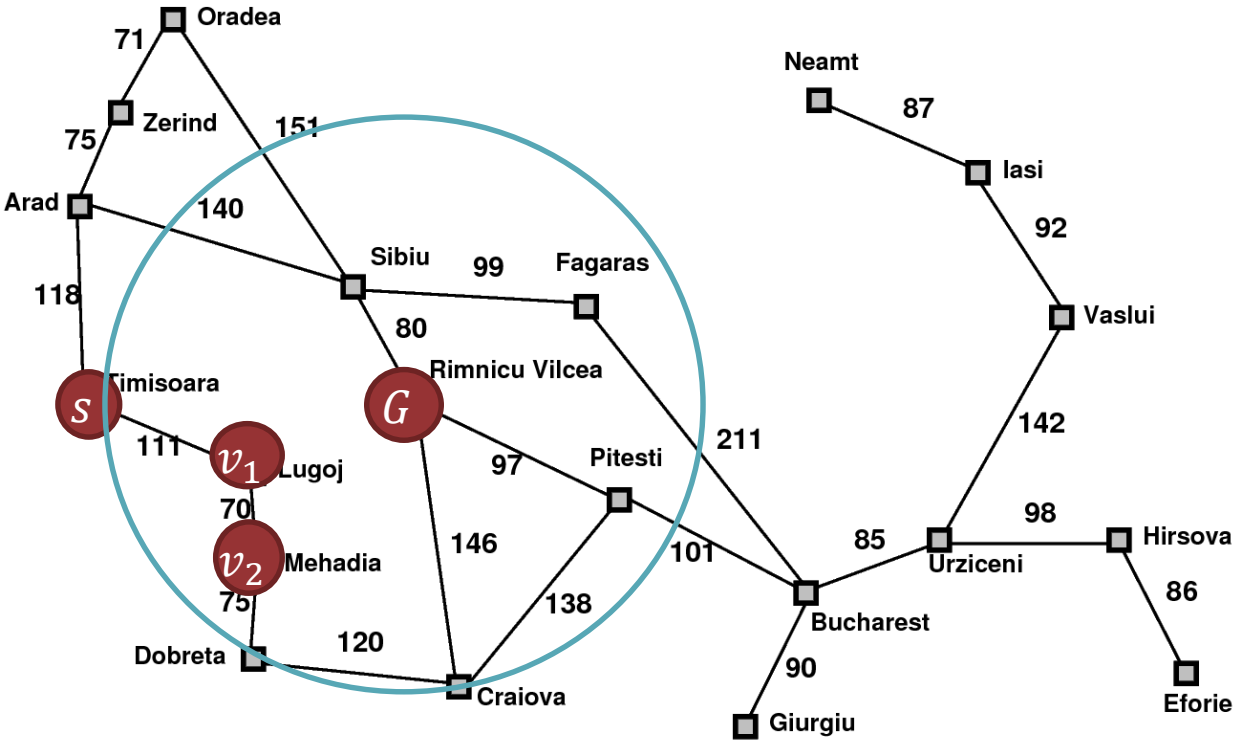


Example - Romania

(d) After expanding Fagaras



What could possibly go wrong here?



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

Properties of Greedy Best-First Search

Complete?

No - can get stuck in loops.
Complete in finite space with checking for repeated states

Sound?

No

Time?

$O(b^m)$ but a good heuristic can give dramatic improvement

Space?

$O(b^m)$ – keeps all nodes in memory (up to good heuristic)

Optimal?

No

b	maximum branching factor of the search tree
d	depth of the shallowest solution
m	maximum depth of the state space (may be ∞)

	BFS	DFS
Time	$O(b^{d+1})$	$O(b^m)$
Space	$O(b^{d+1})$	$O(b \cdot m)$

A* Search

Idea: avoid expanding paths that are already expensive

- use complete evaluation function $f(v) = g(v) + h(v)$
  - $g(v)$: cost so far to reach v
  - $h(v)$: estimated cost from v to $goal$
  - $f(v)$: estimated total cost of path through v to $goal$

A* Search Strategy

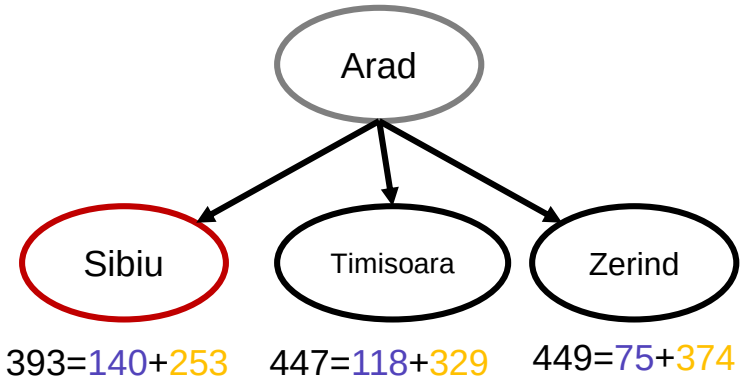
- Heuristic function: $f(v)$
- $fringe$ ordering: increasing by f
- New Nodes: all
- Reset: none

Example - Romania

(a) The initial state

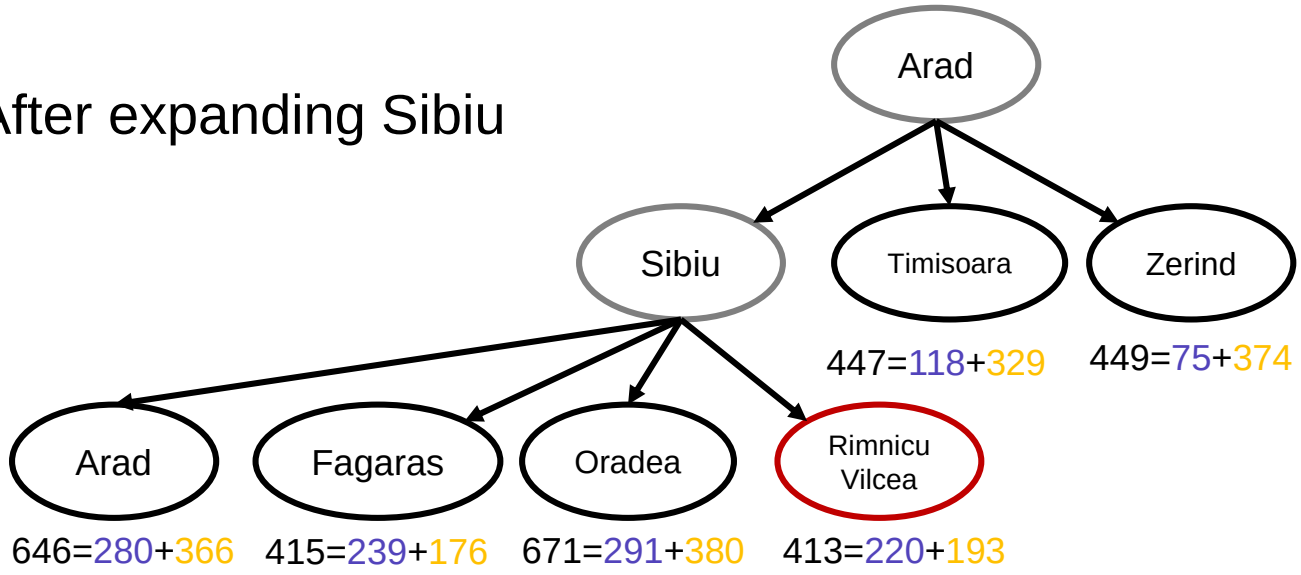
Example - Romania

(c) After expanding Arad



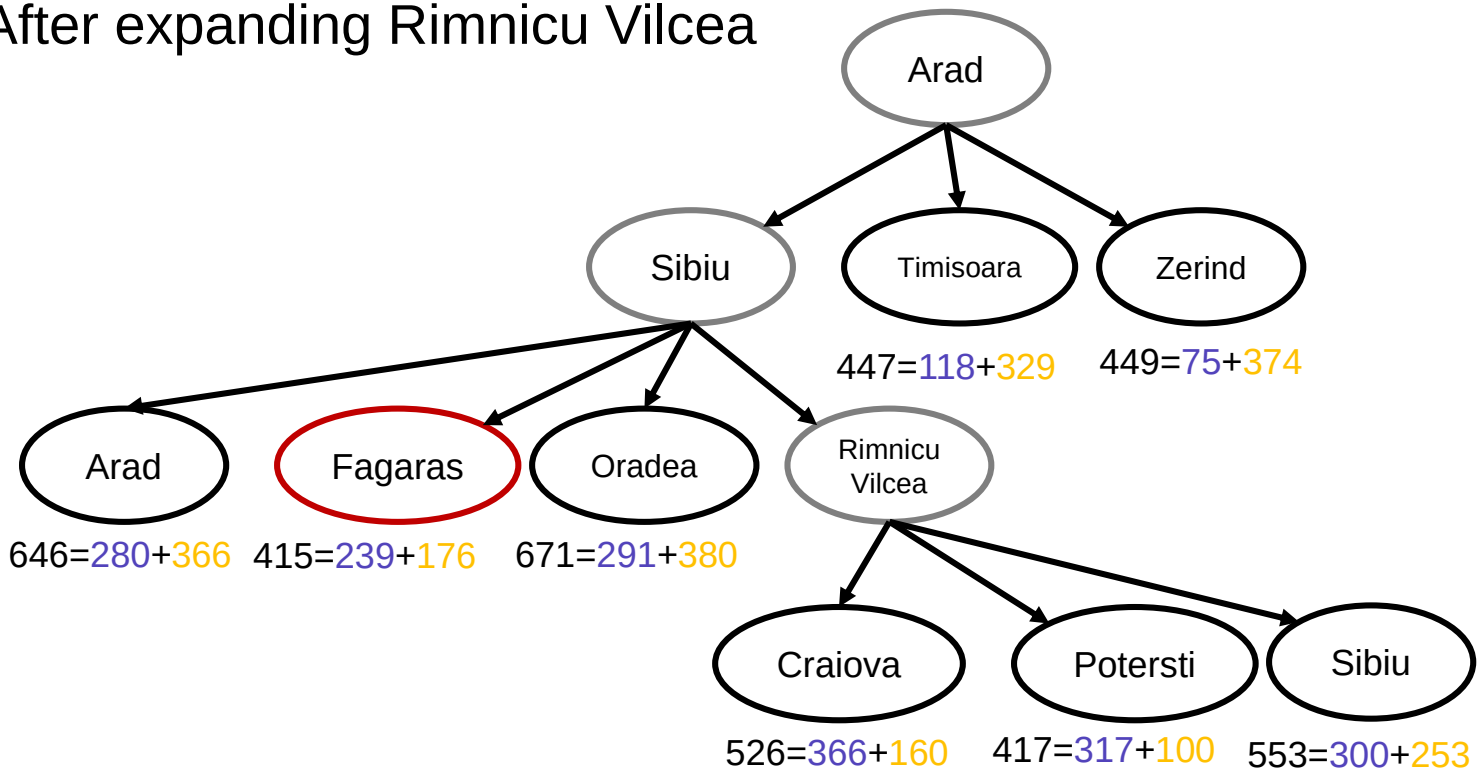
Example - Romania

(c) After expanding Sibiu



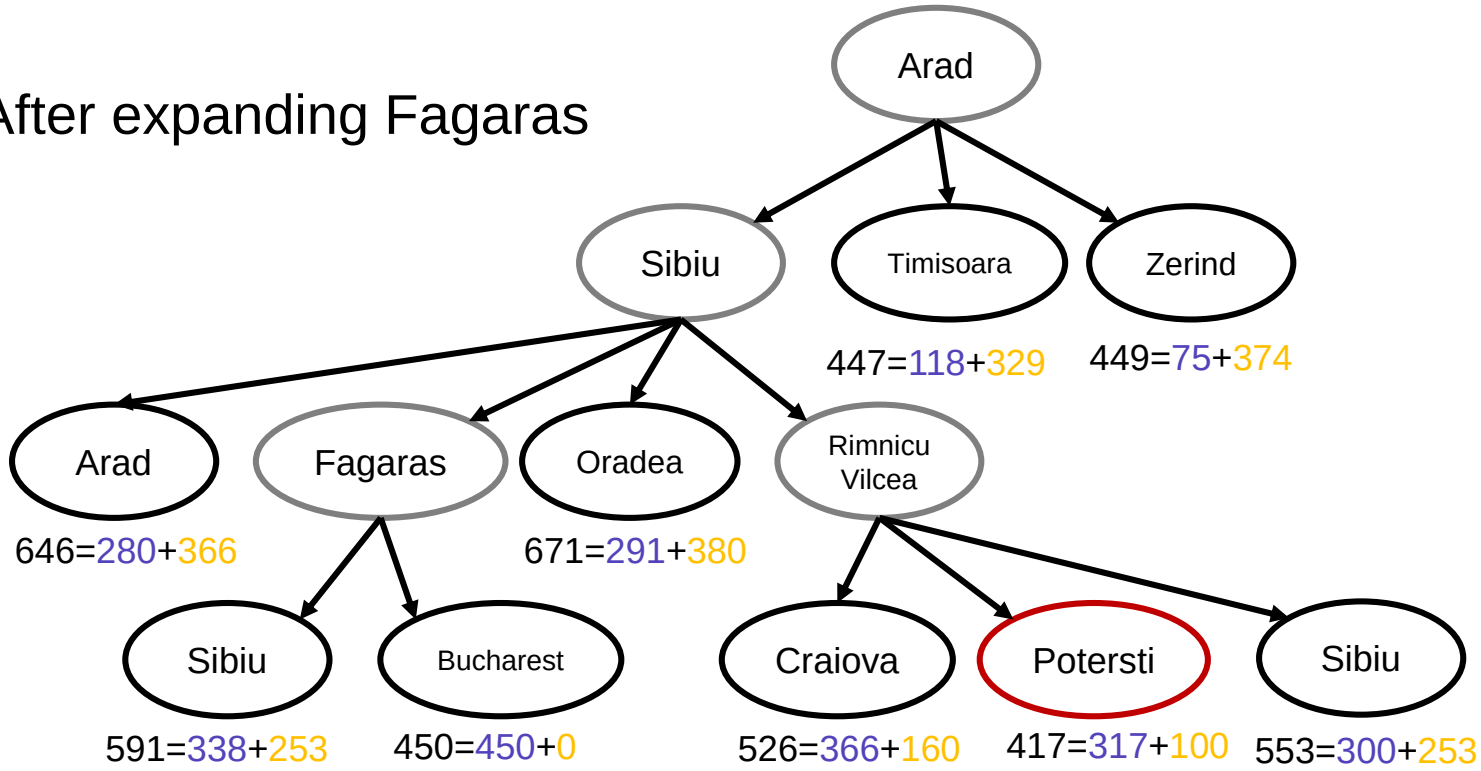
Example - Romania

(d) After expanding Rimnicu Vilcea



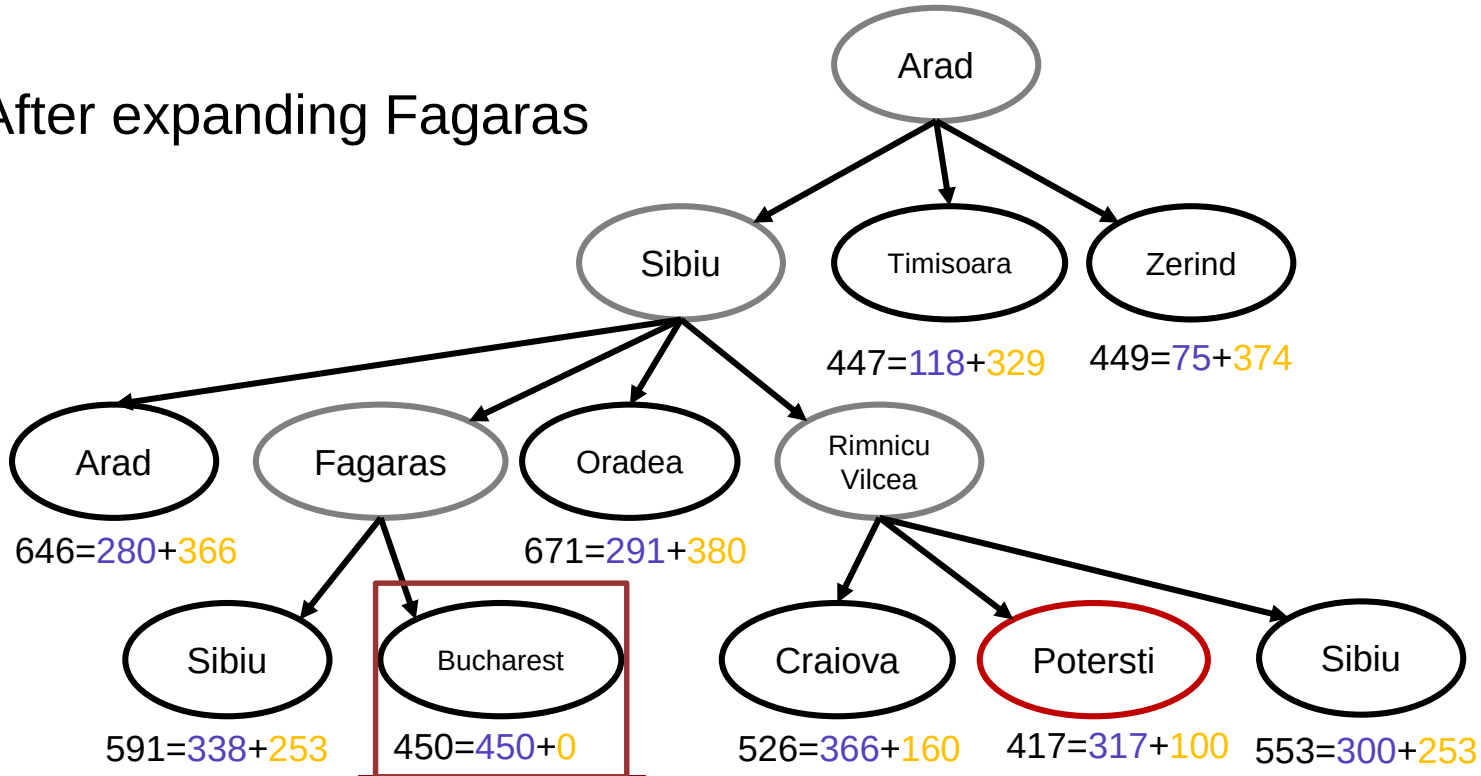
Example - Romania

(e) After expanding Fagaras



Example - Romania

(e) After expanding Fagaras



Heuristics Properties

h is **admissible**

$\forall\, v_0 \in V$

$\forall$ path from v_0 to a goal $v_0, \ldots, v_n$ $(v_n \in G)$

it holds that

$$h(v_0) \leq \sum_{i}^{n} c(e_i)$$

An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic

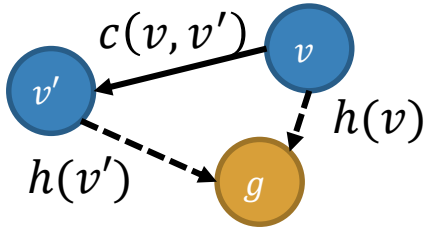
h is **consistent**

$\forall e = (v, v') \in E,$

$h(v) \leq c(e) + h(v')$

and $\forall\, v \in G,\ h(v) = 0$

A consistent heuristic becomes more precise as we get deeper in the search tree (because if h is consistent, then $f(n)$ cannot decrease along a path in the search graph)



h_2 **dominates** h_1

$\forall v \in V\ h_2(v) \geq h_1(v)$

A* Properties

Complete?

Yes - (unless there are infinitely many nodes n with $f(n) < f(goal)$)

Sound? No

Time? Exponential in [relative error in h × length of solution]

Space? $O(b^m)$ - keeps all nodes in memory

Optimal? **A* is optimal** for admissible heuristic for tree trace (for graph trace we need also **consistency**)

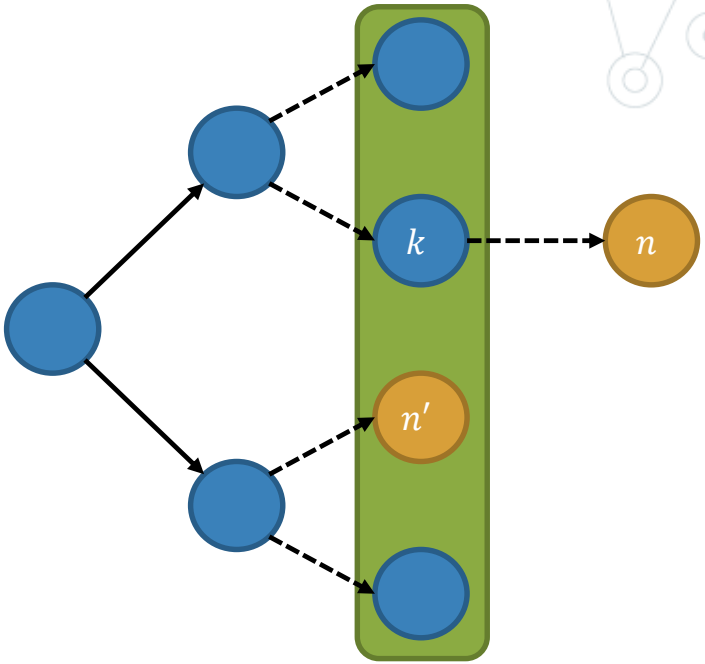
b	maximum branching factor of the search tree
d	depth of the shallowest solution
m	maximum depth of the state space (may be ∞)

A* Optimality (Trees)

Assume by way of contradiction that there exists $n, n' \in G$ such that $g(n') > g(n)$ but A* will return a path to n'

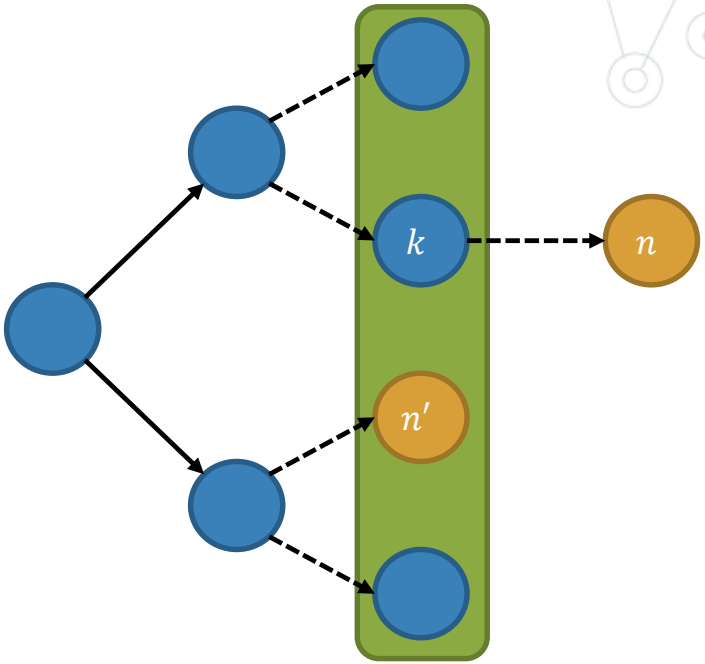
- Denote by n an optimal Goal node
- Denote by k a node on the path to n
- Denote by n' a suboptimal Goal node

consider the time just before A* removed n' form the $fringe$



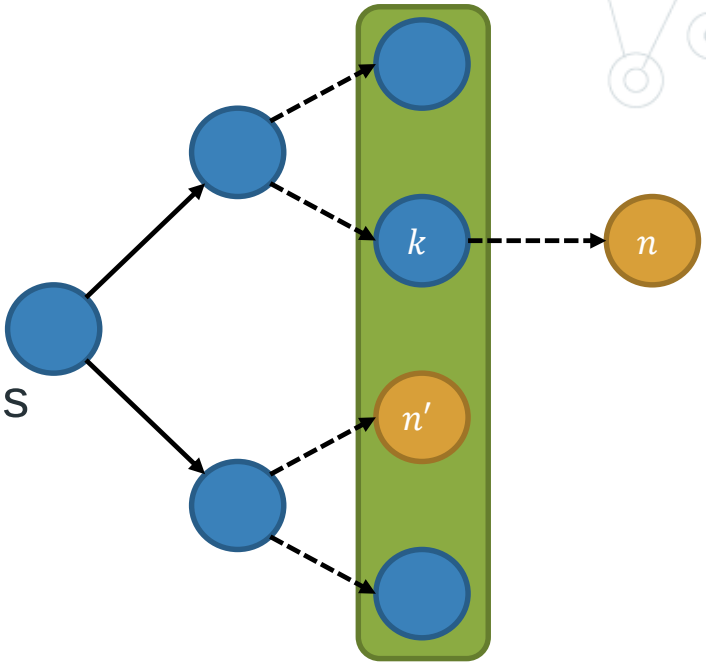
A* Optimality (Trees)

- Let C^* be the cost of the optimal solution
- Assume n' is in the $fringe$
- We know that $f(n') = g(n') + h(n')$
- Also $h(n') = 0$ (since n' is a Goal node)
- Thus, $f(n') = g(n')$
- Since g' is non-optimal, $g(n') > C^*$
- So, $\boldsymbol{f(n') > C^*}$



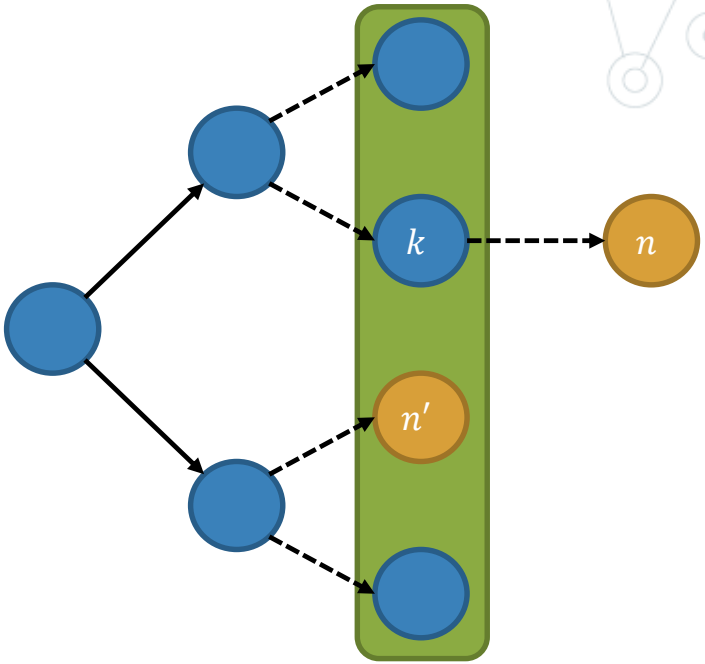
A* Optimality (Trees)

- There must be some node, k, which lies on the optimal solution path and $k \in fringe$
- We know that $f(k) = g(k) + h(k)$
- $g(k)$ is the true cost of getting to node k
- $h(k)$ must be less than the true cost of getting to the goal from node k (since h is admissible)
- Thus, since k lies on the solution path, $f(k) = g(k) + h(k) \leq C^*$
- So, $\boldsymbol{f(k) \leq C^*}$

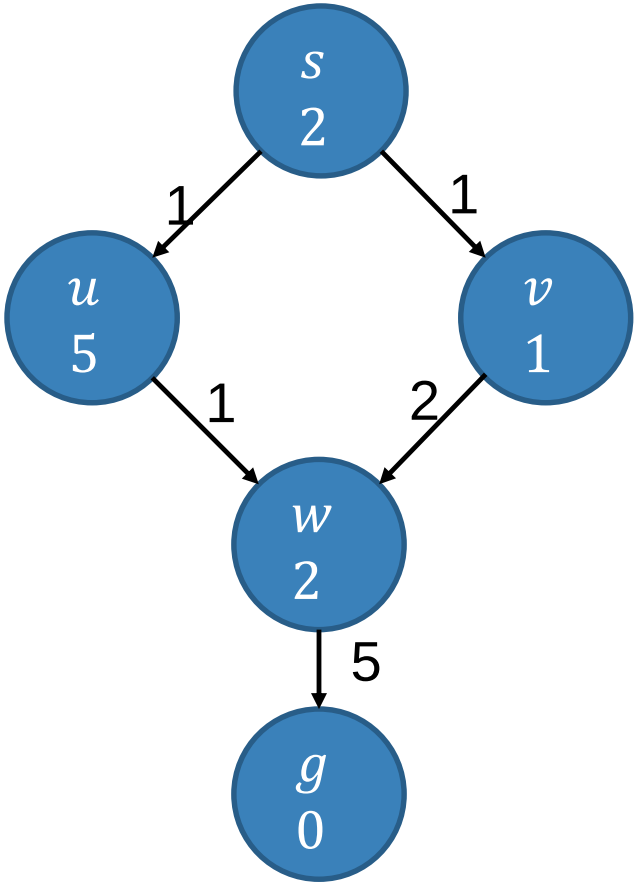


A* Optimality (Trees)

- From the last two slides we have $f(k) \leq C^*$ and $f(n') > C^*$
- Concluding $f(k) < f(n')$, and so n' will **not** be selected for expansion
- **And so, Whenever A* chooses to expand a goal node, the path to this node is optimal**



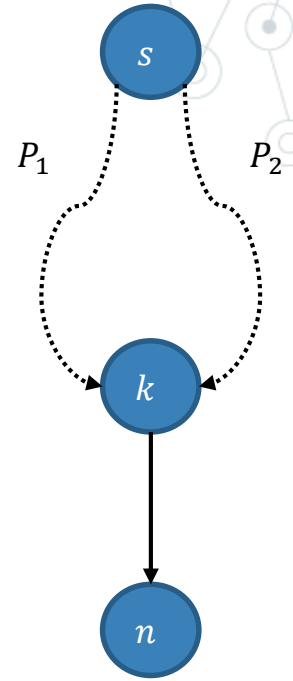
Suboptimal Solution – Graph-Search



A* Optimality (Graphs)

Theorem: If h is consistent, then whenever A* expands a node, it has already found an optimal path to this node's state

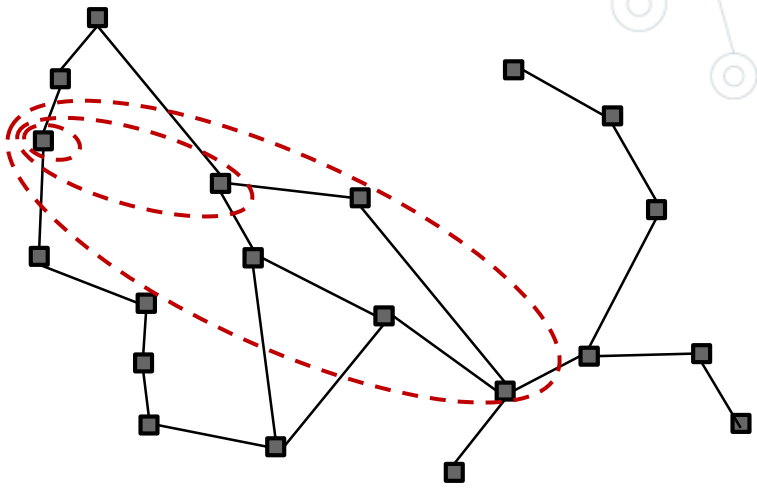
- Consider some node v and its child v'
- Since h is consistent, we know that $h(v) \leq c(v, v') + h(v')$
- $f(v) = g(v) + h(v) \leq g(v) + c(v, v') + h(v') = g(v') + h(v') = f(v')$
- So, $f(v) \leq f(v')$
- Thus, f is **non-decreasing** along any path
- When a node is selected for expansion, the optimal path to that node has been found
- The sequence of nodeт expanded is in non-decreasing order of $f(n)$
- The first goal node selected for expansion must be optimal



A* Optimality (Graphs)

A* expands nodes in order of increasing f value

- A* expands all nodes with $f(v) < C^*$
- A* expands some nodes with $f(v) = C^*$
- A* expands no nodes with $f(v) > C^*$



A* is optimal since it must expand all nodes with $f(v) < C^*$

Gradually adds "f-contours"

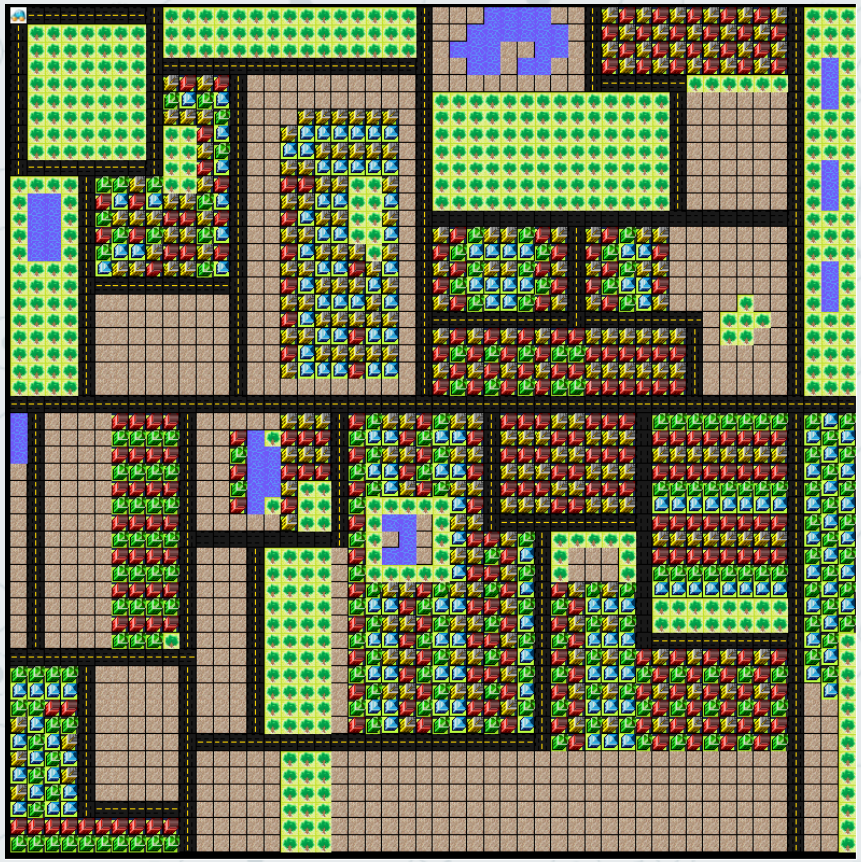
The need for good heuristic

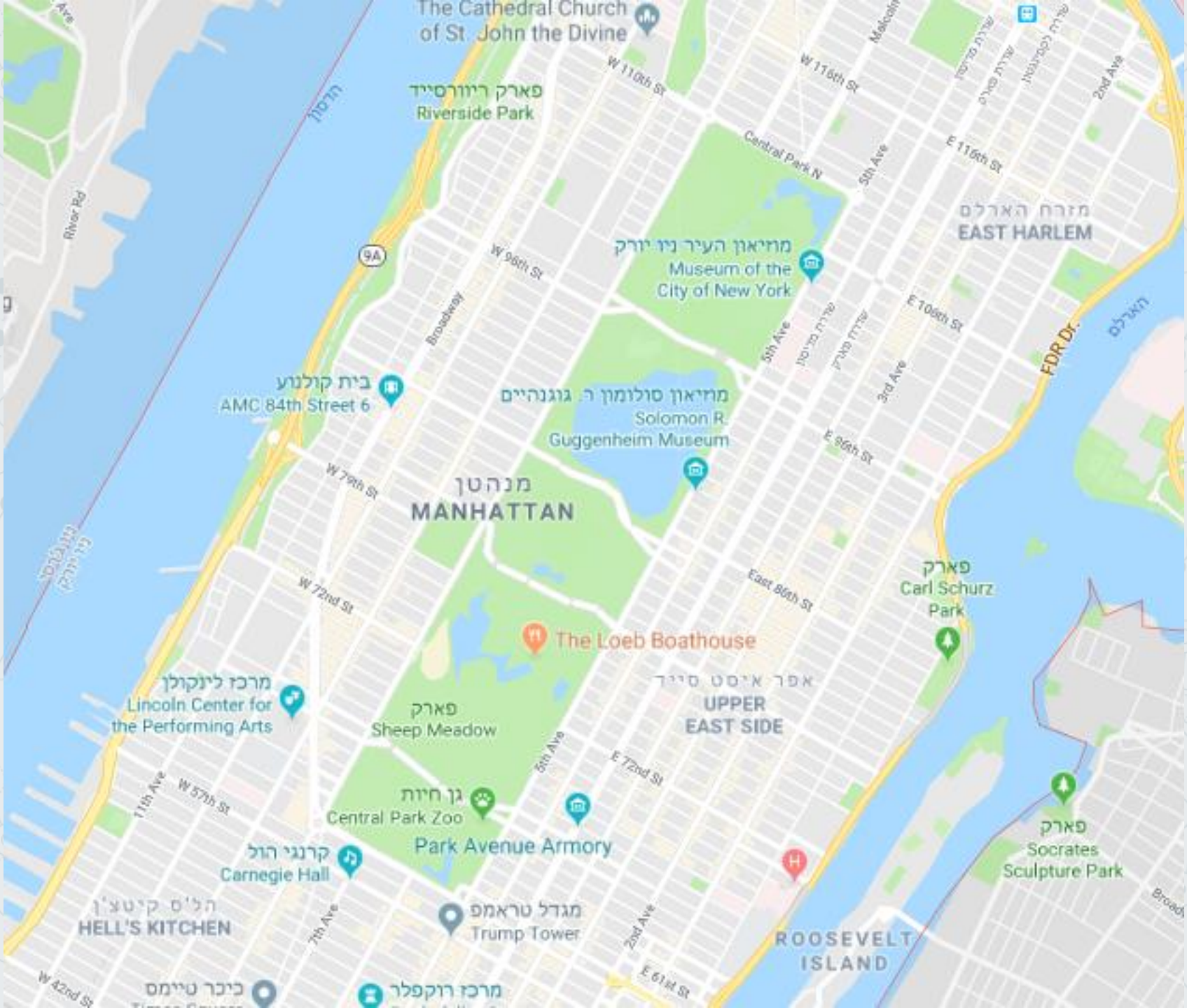
Where Does h Come From?

A very general strategy to compute h:

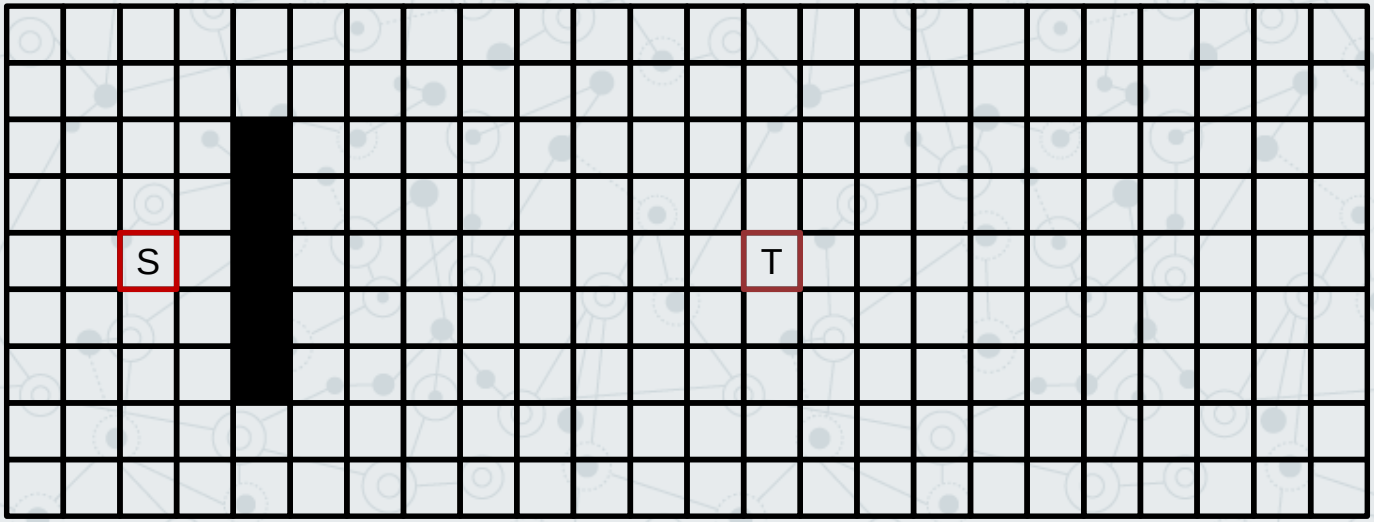
- Relax the original problem
- Solve the relaxed problem, i.e. find a path from v to a goal
- Use the relaxed solution as an estimate for the real solution, i.e. define $h(v)$ as the length of a relaxed path from v to a goal

Example - Grid World

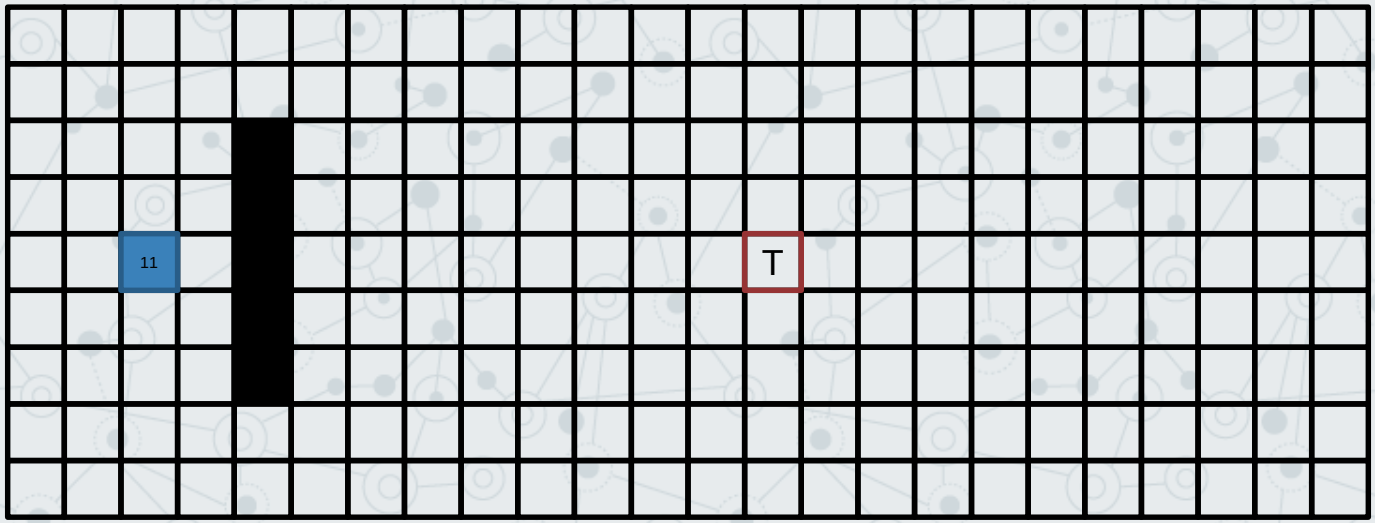



The Cathedral Church of St. John the Divine
פארק ריוורסייד
Riverside Park
W 110th St
W 116th St
Central Park N
5th Ave
E 116th St
2nd Ave
מזרח הארלם
EAST HARLEM
מוזיאון העיר ניו יורק
Museum of the City of New York
W 96th St
Broadway
E 106th St
FDR Dr
בית קולנוע
AMC 84th Street 6
מוזיאון סולומון ר. גוגנהיים
Solomon R. Guggenheim Museum
E 96th St
3rd Ave
W 79th St
מנהטן
MANHATTAN
פארק
Carl Schurz Park
East 86th St
W 72nd St
The Loeb Boathouse
מרכז לינקולן
Lincoln Center for the Performing Arts
פארק
Sheep Meadow
אפר איסט סייד
UPPER EAST SIDE
E 72nd St
11th Ave
W 57th St
גן חיות
Central Park Zoo
Park Avenue Armory
קרנגי הול
Carnegie Hall
פארק
Socrates Sculpture Park
הלס קיטצ'ן
HELL'S KITCHEN
מגדל טראמפ
Trump Tower
ROOSEVELT ISLAND
W 42nd St
E 61st St
מרכז רוקפלר

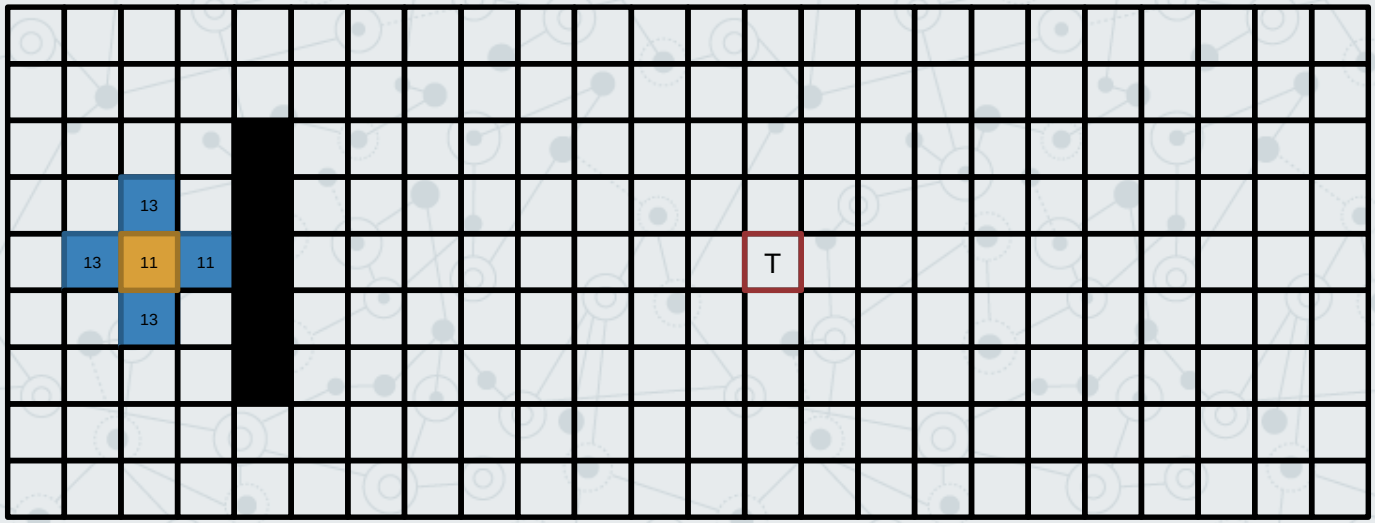
Example - Simple Grid



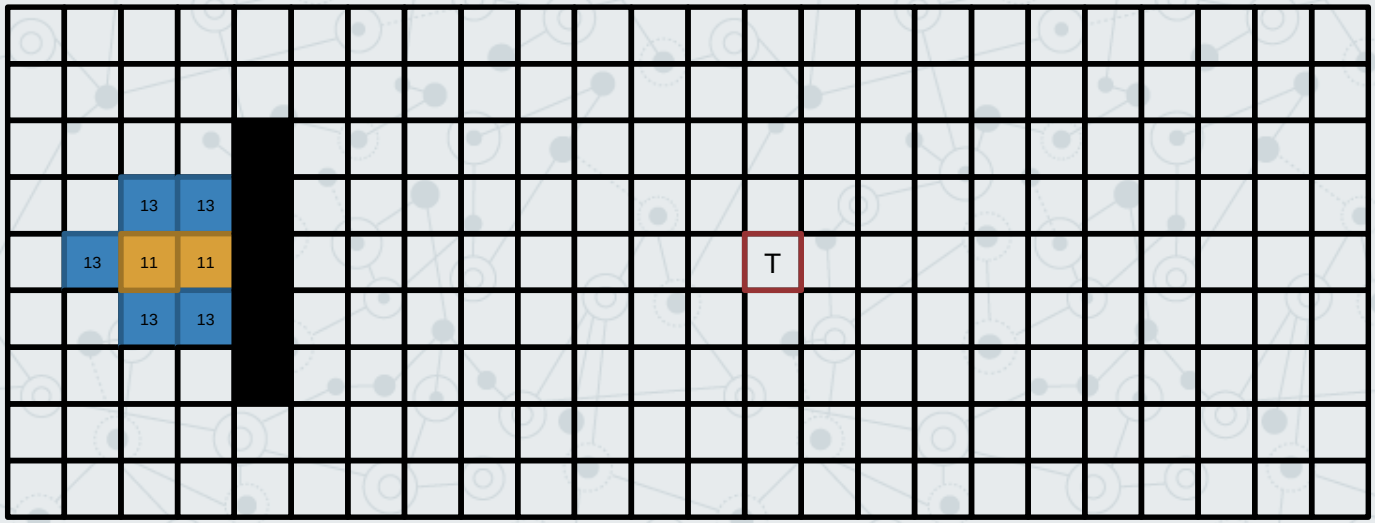
Example - Simple Grid



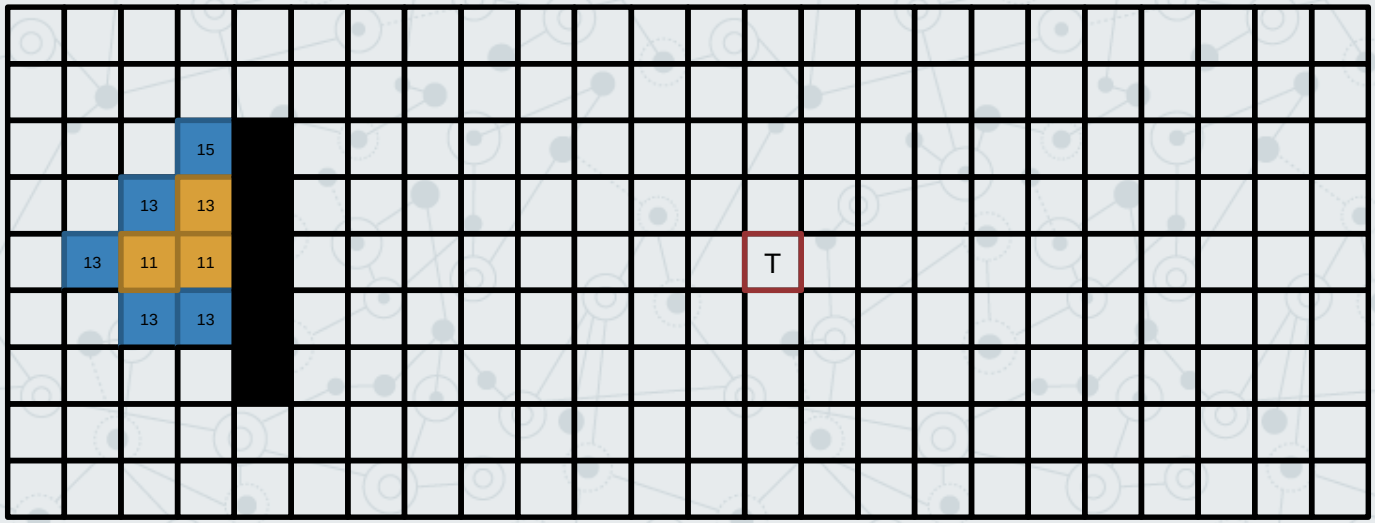
Example - Simple Grid



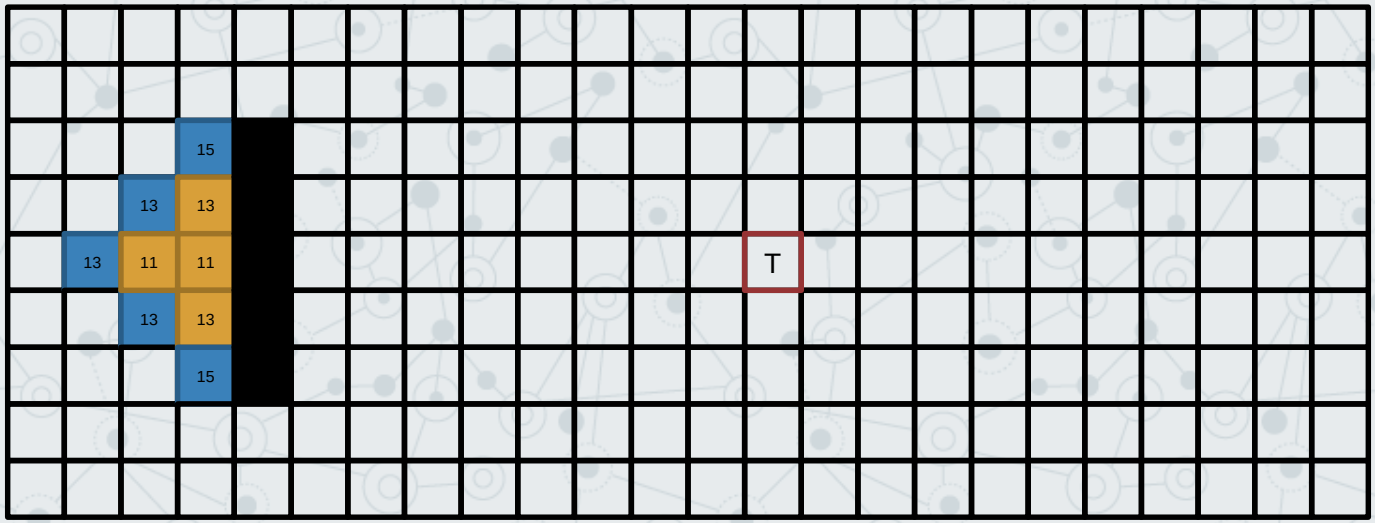
Example - Simple Grid



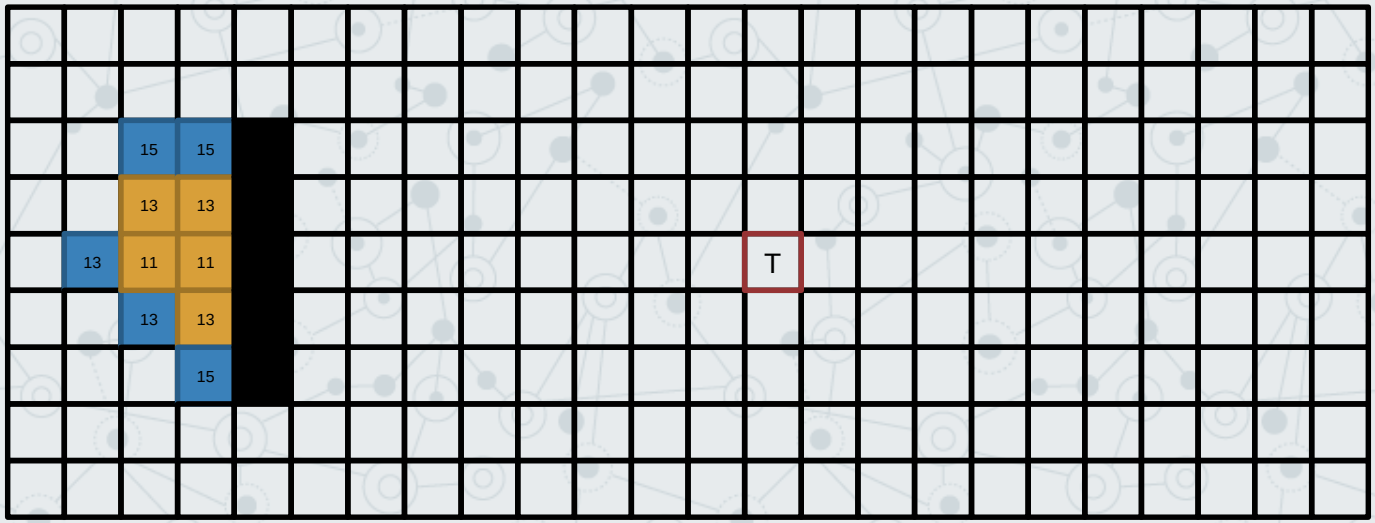
Example - Simple Grid



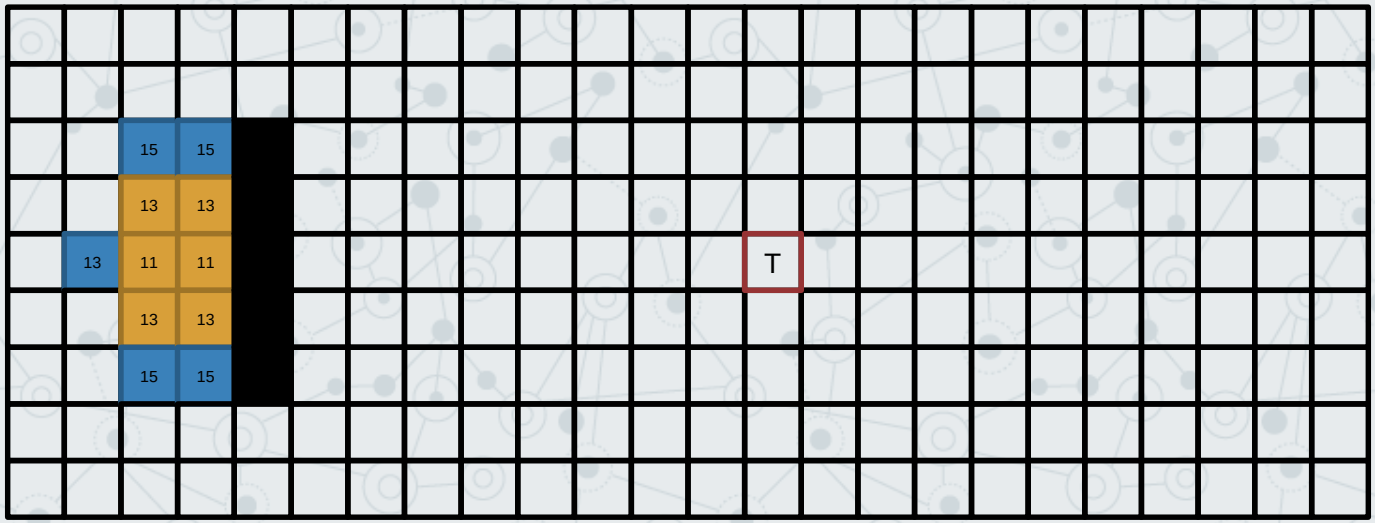
Example - Simple Grid



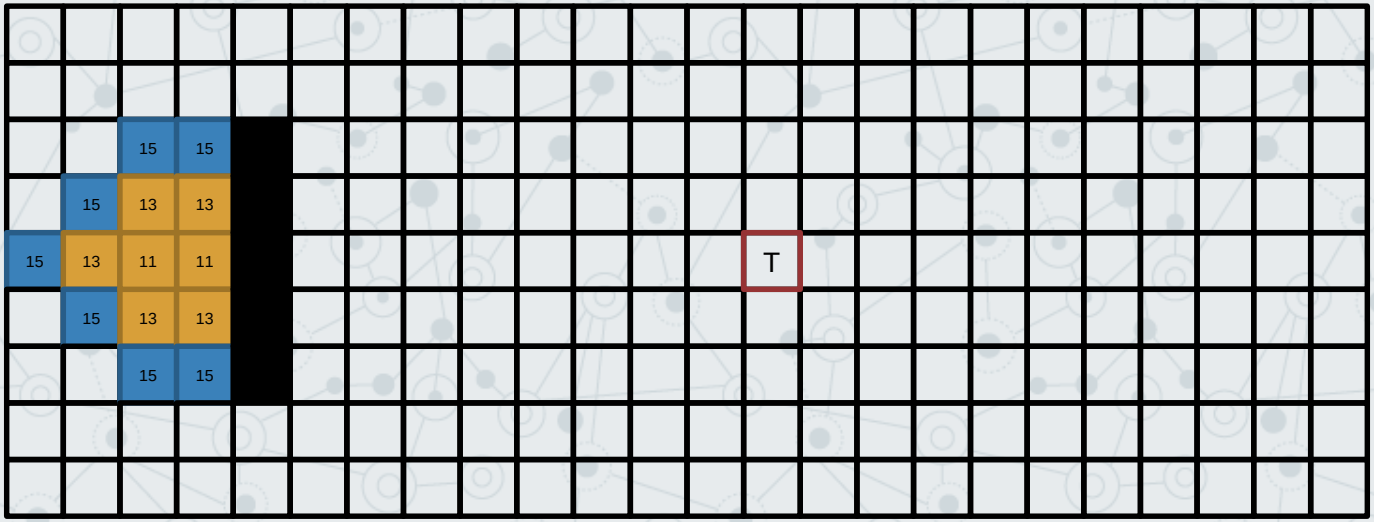
Example - Simple Grid



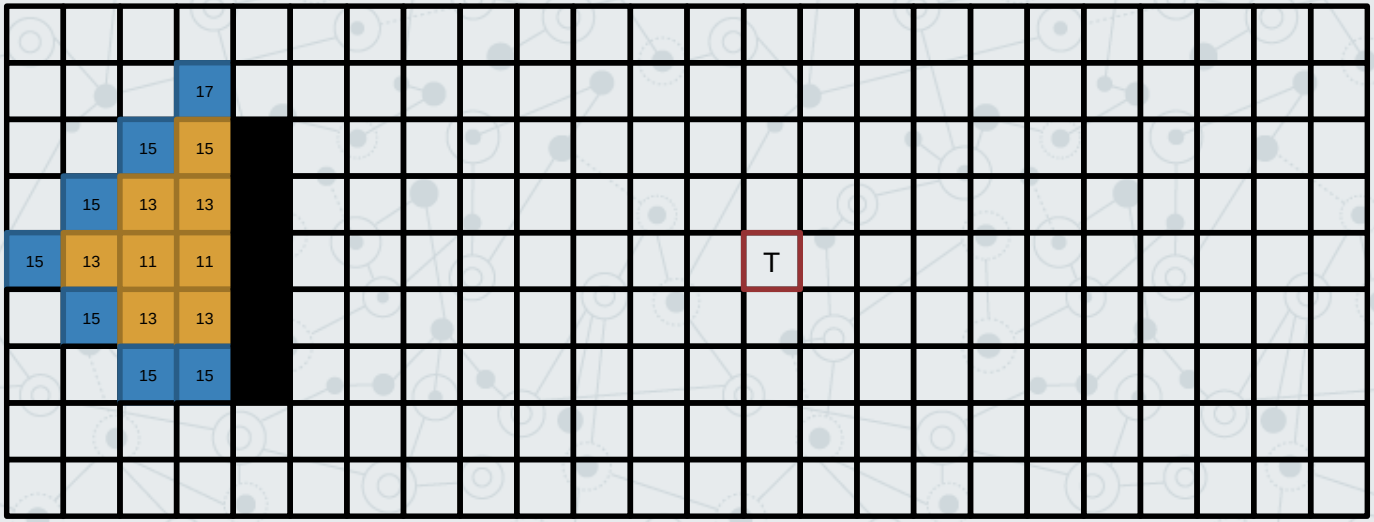
Example - Simple Grid



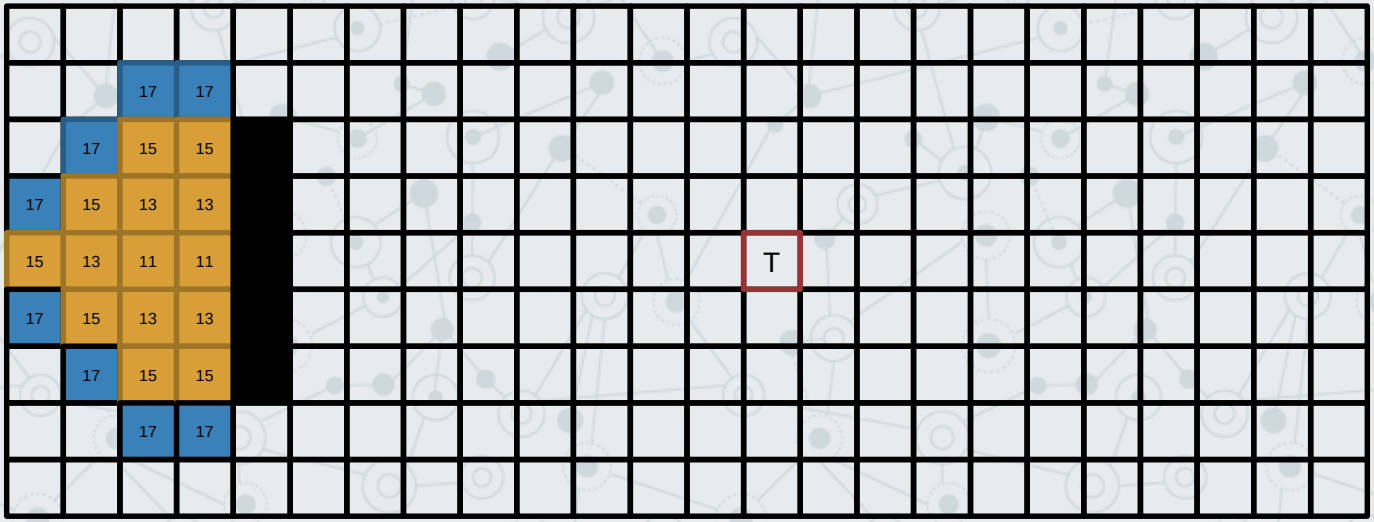
Example - Simple Grid



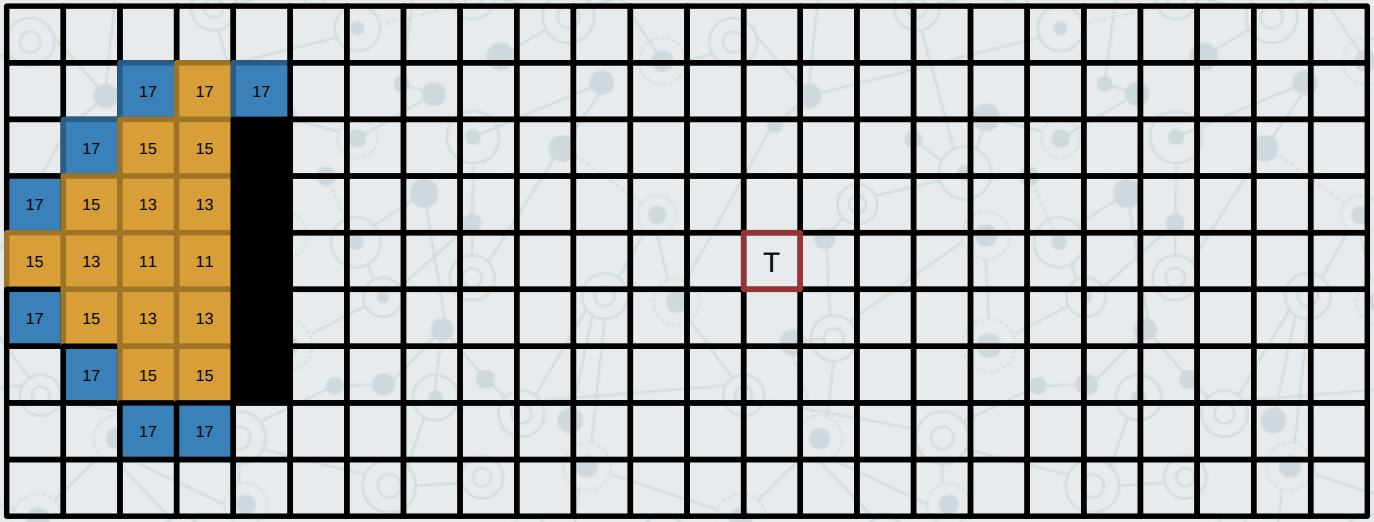
Example - Simple Grid



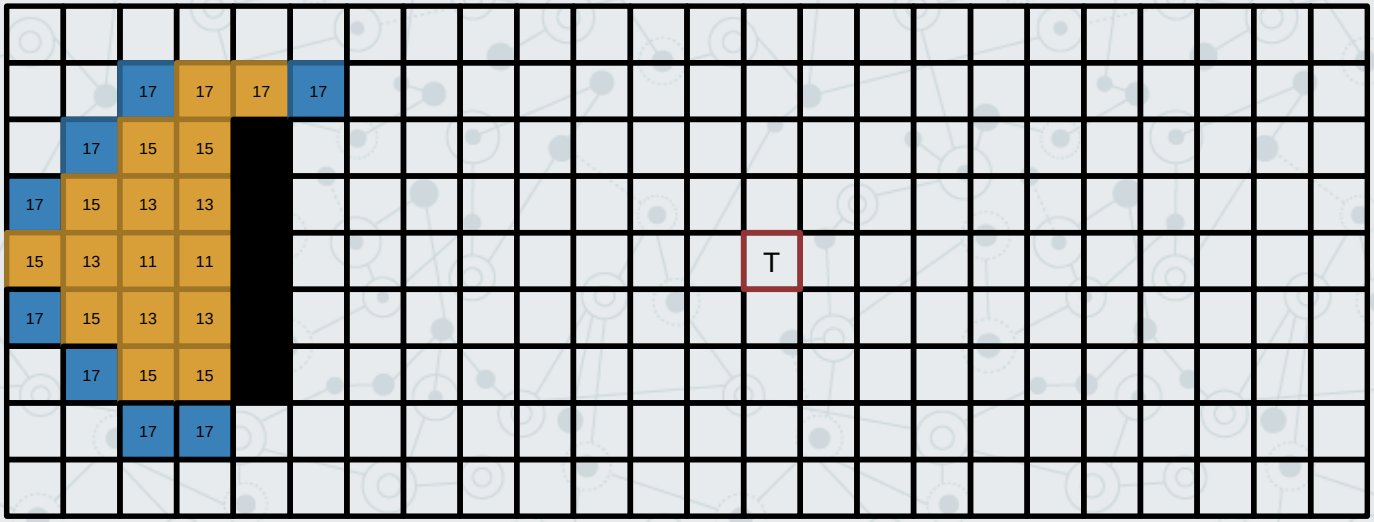
Example - Simple Grid



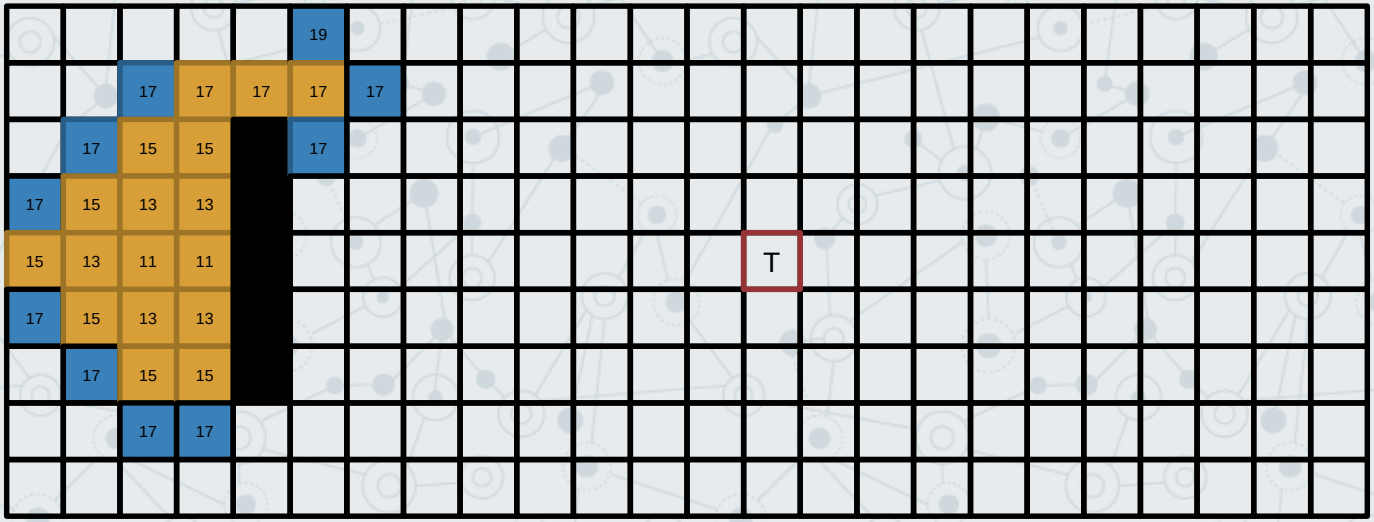
Example - Simple Grid



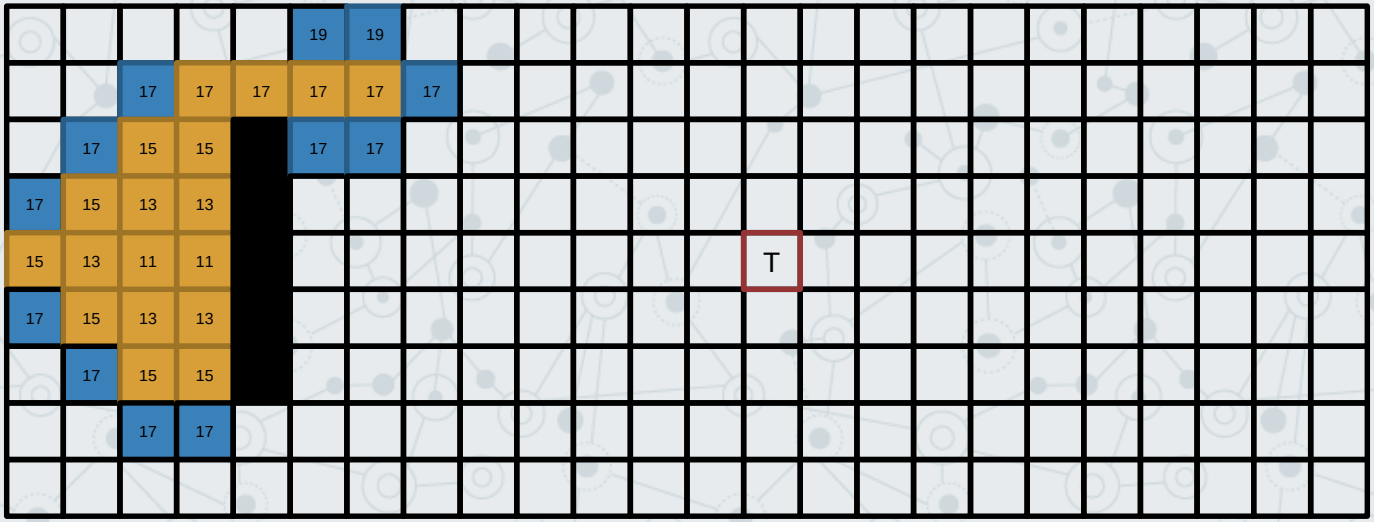
Example - Simple Grid



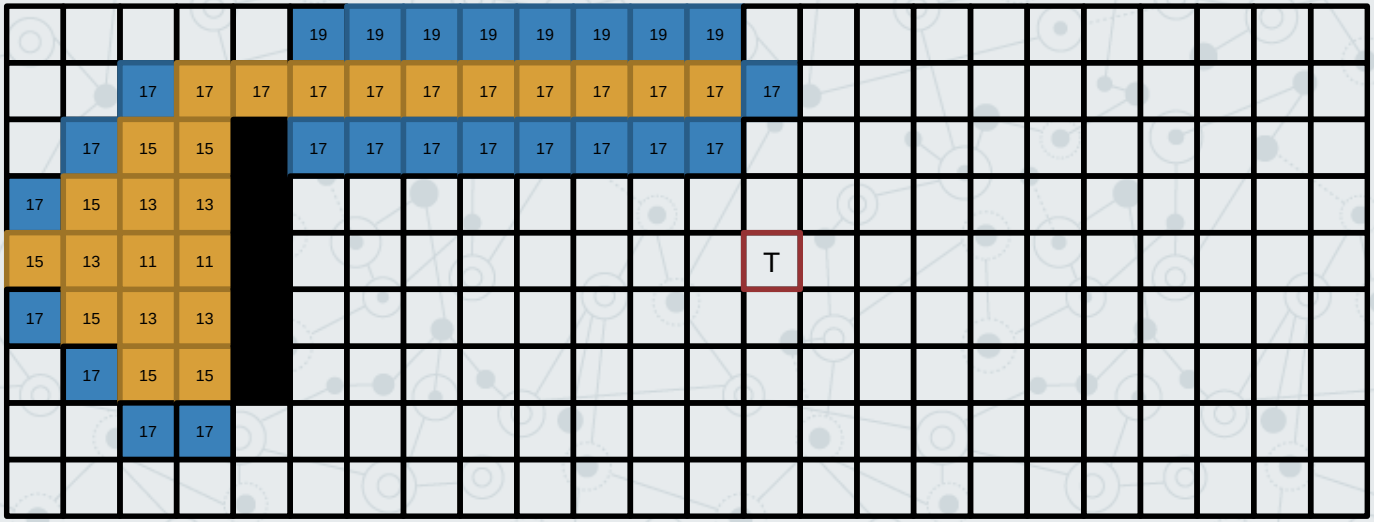
Example - Simple Grid



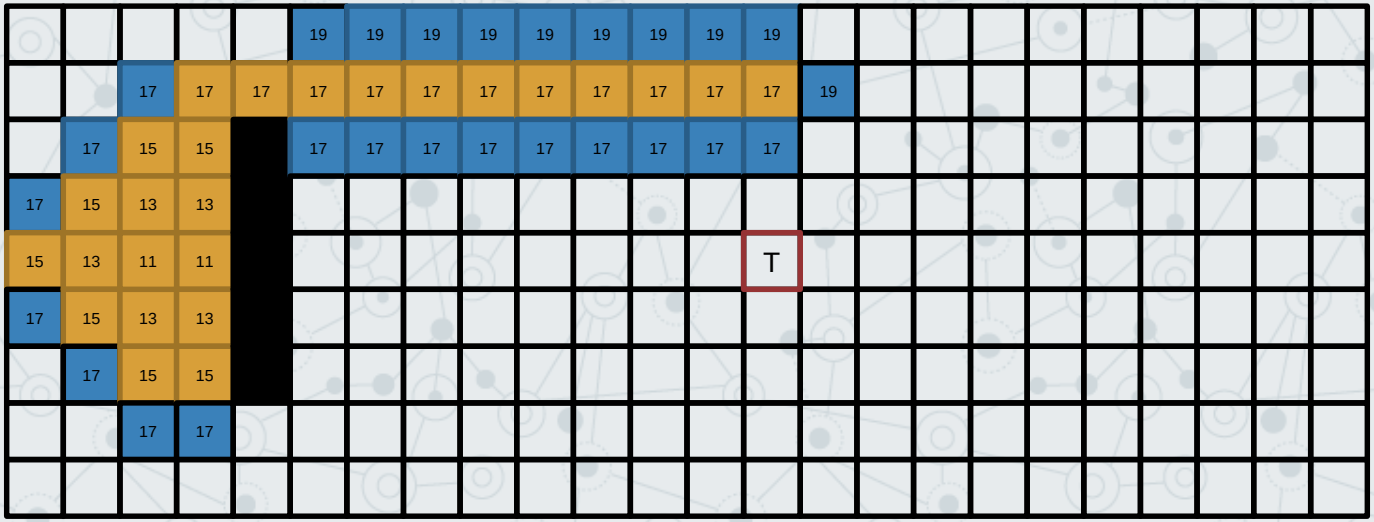
Example - Simple Grid



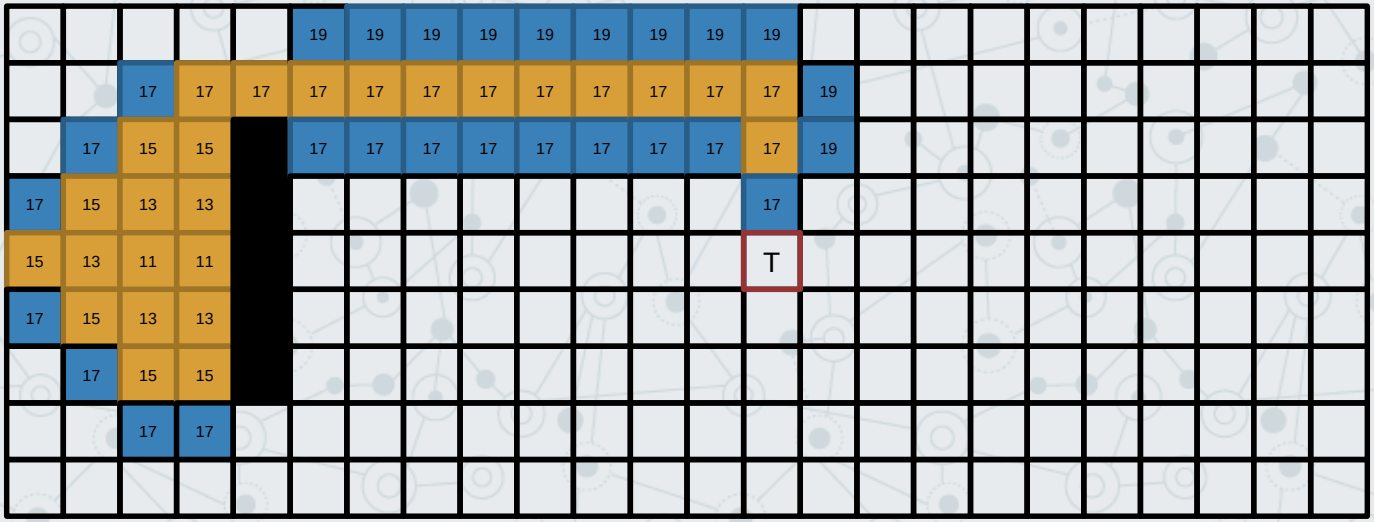
Example - Simple Grid



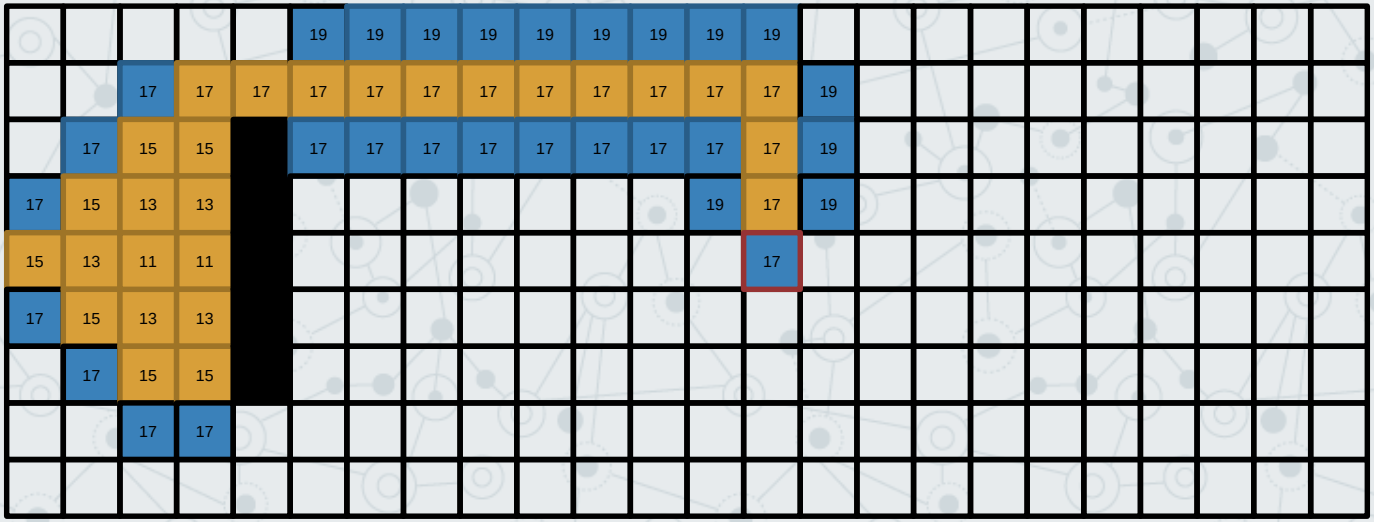
Example - Simple Grid



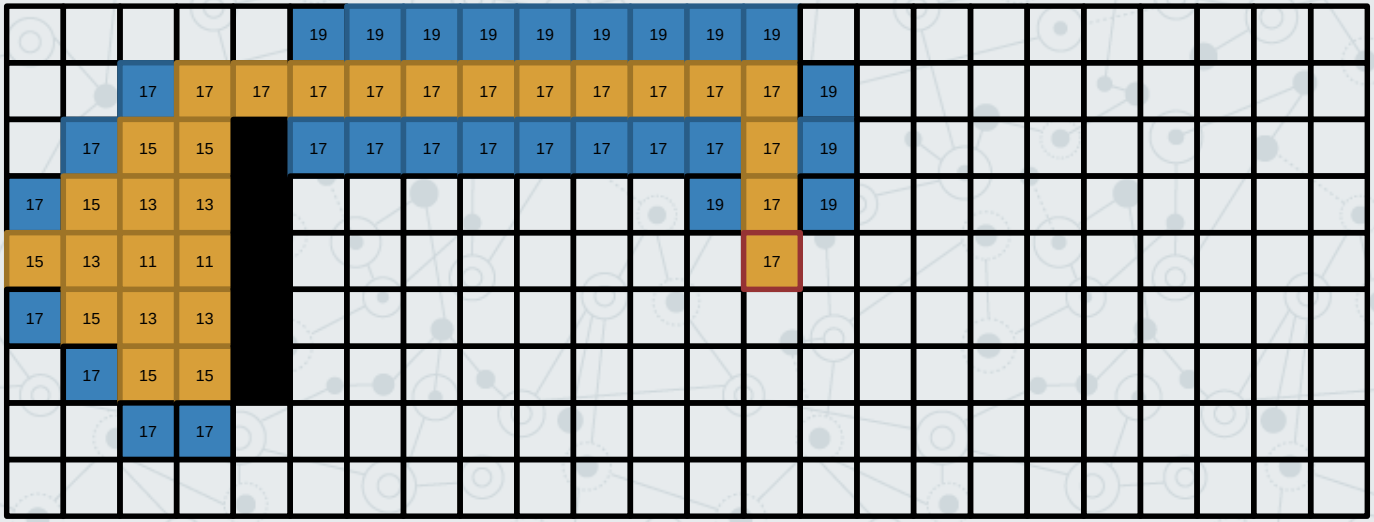
Example - Simple Grid



Example - Simple Grid

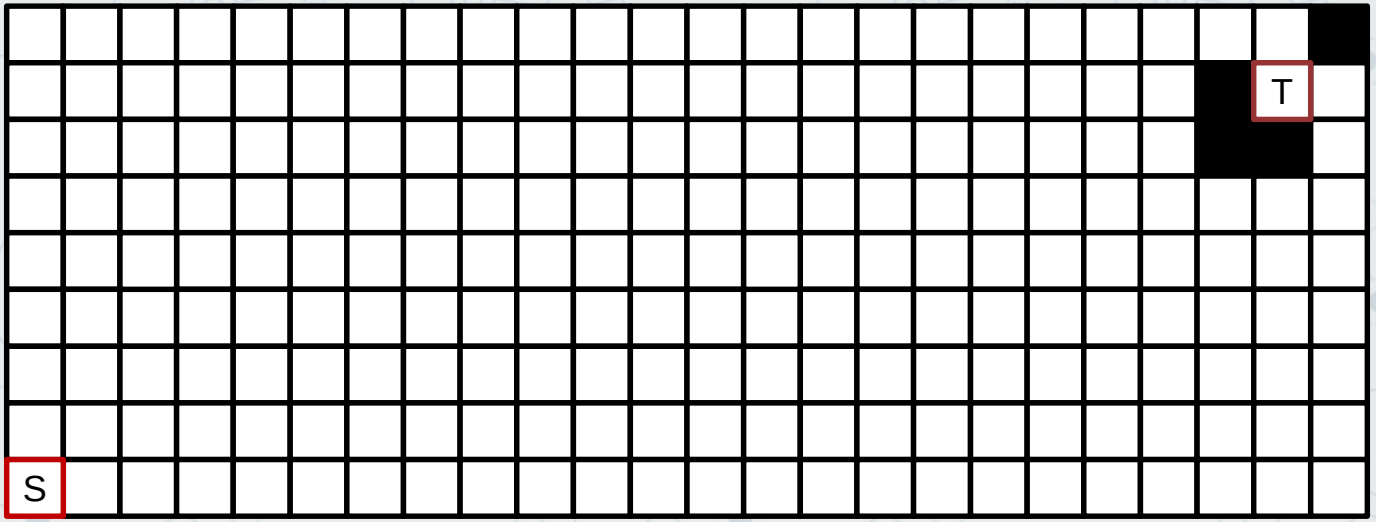


Example - Simple Grid



How Good is Almost Perfect ?

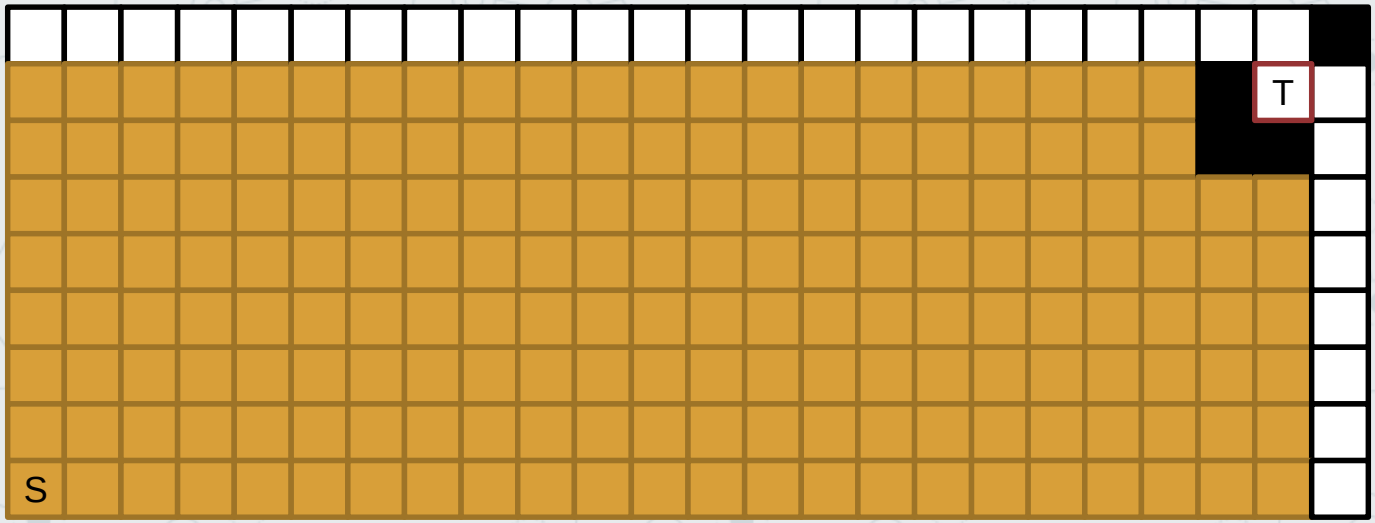
In this example $\forall v, |h^*(v) - h(v)| \leq 2$ i.e. h is almost perfect



How good is almost perfect?

How Good is Almost Perfect ?

In this example $\forall v, |h^*(v) - h(v)| \leq 2$ i.e. h is almost perfect



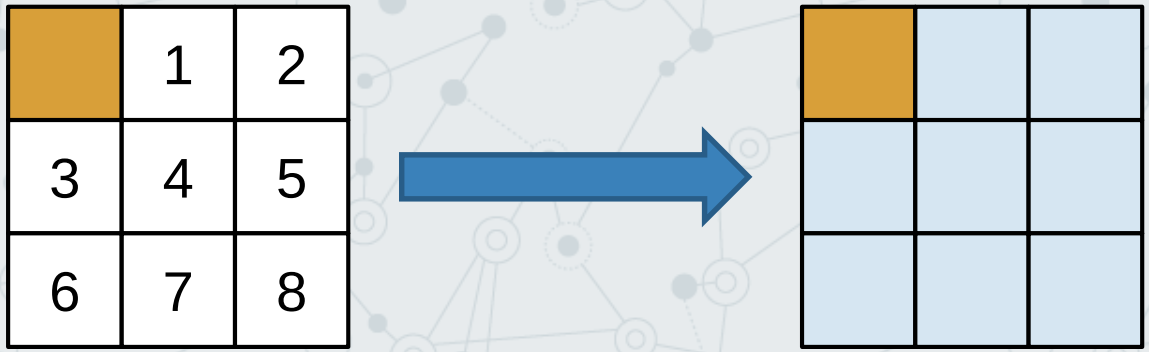
A* will expand almost all states

Heuristic Defined by Abstraction

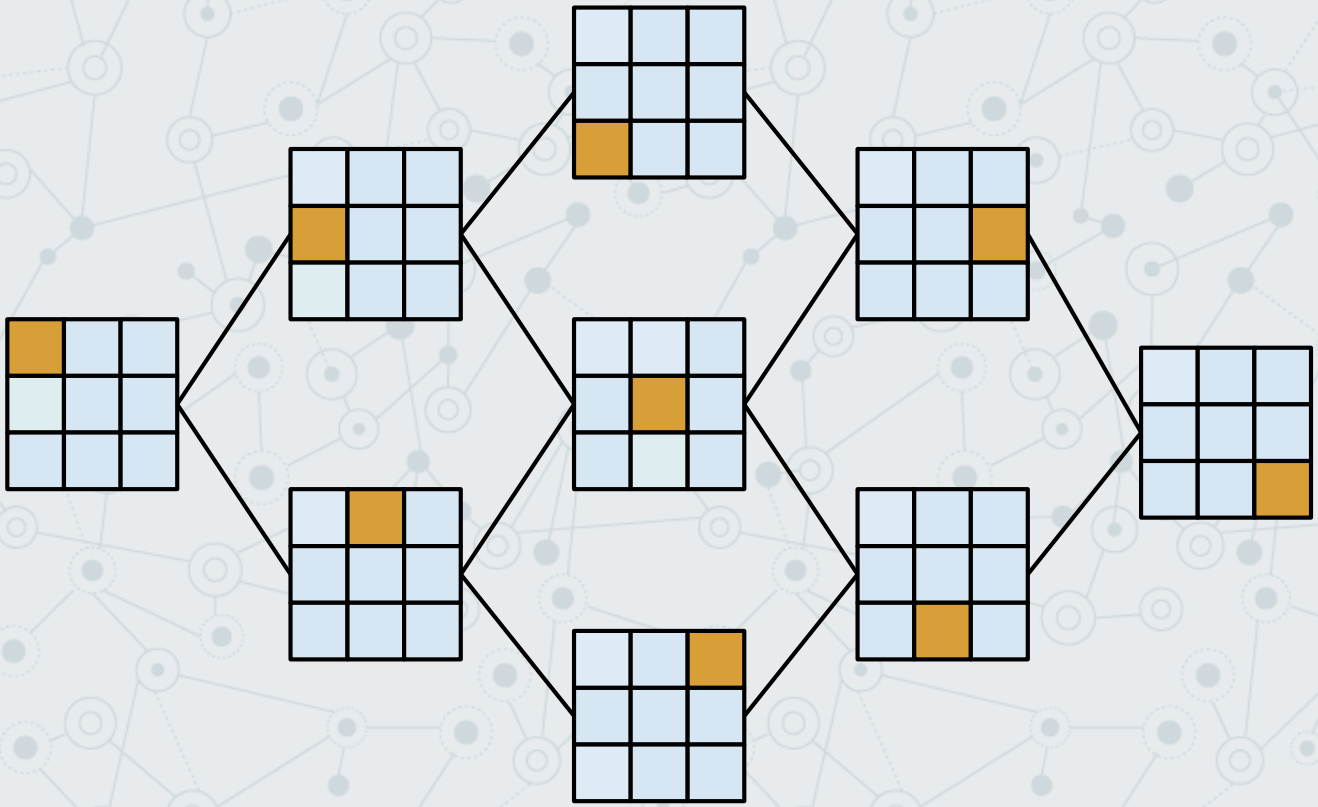
Create a simplified version of your problem and then use the exact distances in the simplified version as heuristic estimates in the original problem.

- Abstraction of a state space $\mathcal{S}$ is any state space $\varphi(\mathcal{S})$ s.t.
  - for every state $s \in \mathcal{S}$ there is a corresponding state $\varphi(s) \in \varphi(\mathcal{S})$
  - $distance(\varphi(s_1), \varphi(s_2)) \leq distance(s_1, s_2)$
- The distances in $\varphi(\mathcal{S})$ are admissible and consistent heuristics for searching $\mathcal{S}$

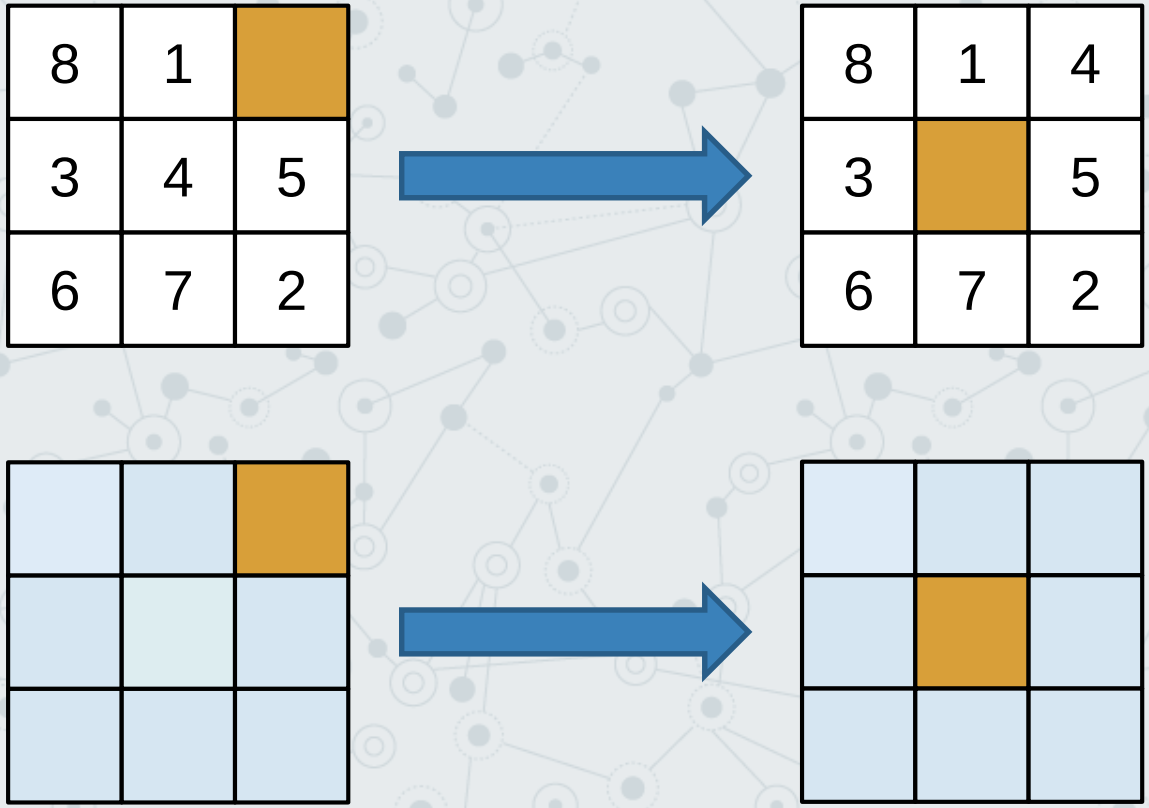
Example – Sliding-Tile Puzzle (Abstraction)



Example – Sliding-Tile Puzzle (Abstraction)



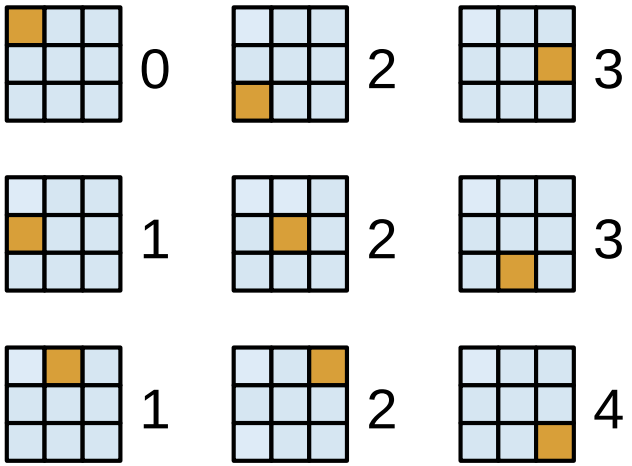
Example – Sliding-Tile Puzzle (Abstraction)



$$h(s) = 2$$

Pattern Databases (PDBs)

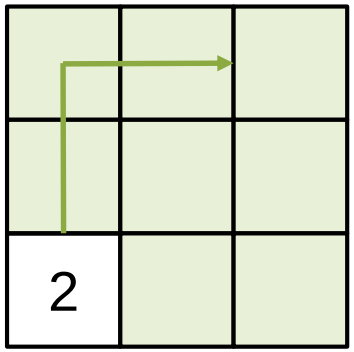
- Enumerate the entire abstract space as a preprocessing step (e.g. by BFS backwards from $\varphi(goal)$)
- Store distance-to-goal for every abstract state in a **lookup table**
- During search in the original state space, $h(s)$ is computed by a lookup in the table



Example – Sliding-Tile Puzzle (Relaxation)

In order to move a tile from x to y:

- x and y must be adjacent
- position y must be empty



By ignoring the second condition, the cost of such a solution is exactly the Manhattan distance

Note - that a constraint has been removed, the cost of an optimal solution to the simplified problem is a lower bound on the cost of an optimal solution to the original problem. This ensures the heuristic derived this way it is admissible.

Heuristics Design

Relaxation/Abstraction

- Relax/Abstract original problem
- Obtain (exact) solution to the relaxed/abstracted problem
- Based on relaxed/abstracted solution estimate original optimum

Weighting

- Create a set of heuristics $\{h\}_{i=0}^{n-1}$
- Obtain relative importance $(w_0, \ldots, w_{n-1})$ s.t. $\sum w_i = 1$
- Let $h(v) = \sum_{i=0}^{n-1} w_i \cdot h_i(v)$

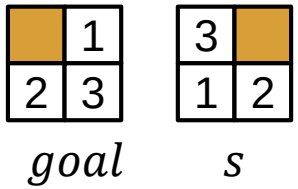
Composition

- Create a set of heuristics $\{h\}_{i=0}^{n-1}$
- Create a composition function $F: \mathbb{R}^n \rightarrow \mathbb{R}$ e.g. $F = \max, sum$, ...
- Let $h(v) = F\left(h_0(v), \ldots, h_{n-1}(v)\right)$

Example – Sliding-Tile Puzzle (Relaxation)

Manhattan distance is also an example of a set of additive PDB :

- Each group contains only a single tile
- For each tile we compute the minimum number of moves that it needs from his current position to his goal position (either directly from formula or from in a pre-calculated table)
- Finally, we sum the resulting values for all tiles



$goal$ s

$$MD(s) = 2 + 1 + 2 = 5$$

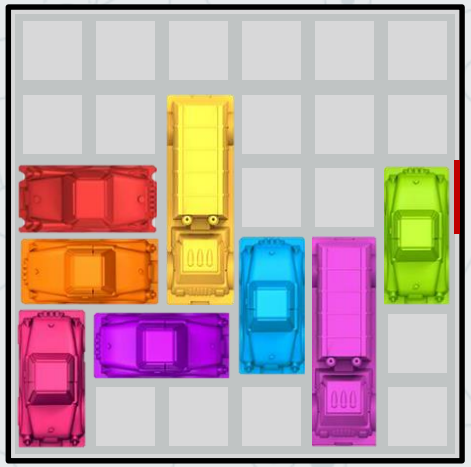
Example – Sliding-Tile Puzzle (Relaxation)

- Consider a situation of two tiles are in their goal row (column) but reversed in order
- Manhattan distance does not account for the fact that one of these tiles must move out of the row (column) and allow the other to pass by, and then move back to this row (column)
- **Full linear conflict** heuristic finds all tiles that are in their goal row (column), but reversed in order, computes the number of moves needed to resolve these conflicts, and adds this to the Manhattan distance heuristic
- Note that full linear conflict does not violates admissibility

Example – Rush Hour

Admissible Heuristics:

- Distance Heuristic – The distance between the red car to the exit
- Blocking Cars Heuristic – The number of cars between the red car to the exit (admissible)
- Blocking Cars Heuristic + Distance Heuristic
- Blocked Blocking Cars Heuristic + Distance Heuristic



Local Search

- Local search is a heuristic method for solving a general class of **computational hard optimization problems**.
- In many optimization problems, **the path to the goal is irrelevant**. The goal state itself is the solution.
- The search space is the **space of candidate solutions** and local search algorithms move from solution to solution in that space until an optimal solution found.
- The exploration of the solution space is done in sequential fashion, moving from a current solution to a "nearby" one (we need some **neighbor relation**).

Local Search

Local search = use single current state, try to improve it by moving to neighboring states

Advantages:

- Use very little memory (so we actually use constant space)
- Find often reasonable solutions in large or infinite state spaces

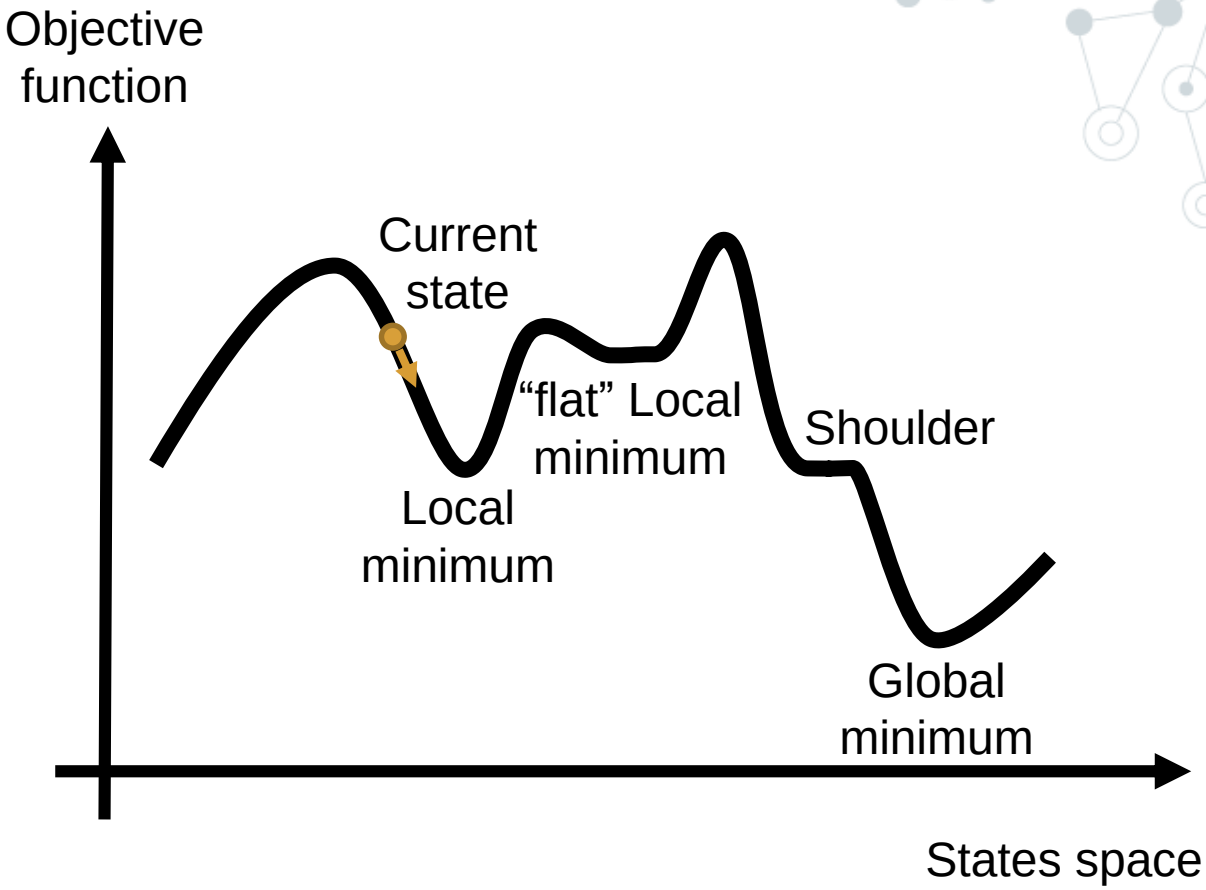
Are also useful for pure optimization problems

Gradient Descent

Let $\mathcal{S}$ denote current solution.

If there is a neighbor $\mathcal{S}'$ of $\mathcal{S}$ with strictly lower cost replace $\mathcal{S}$ with the neighbor whose cost is as small as possible.

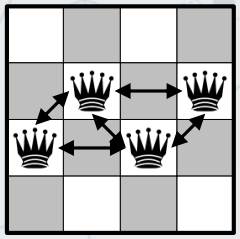
Otherwise terminate



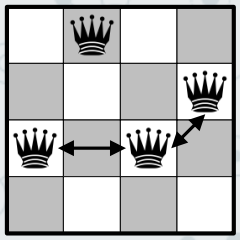
Example - n-Queens Problem

Complete State Formulation

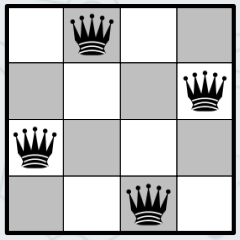
- **Successor function**: move a single queen to another square in the same column (so as to reduce the number of conflicts)
- **Heuristic function $h(n)$**: the number of pairs of queens that are attacking each other



$h = 5$



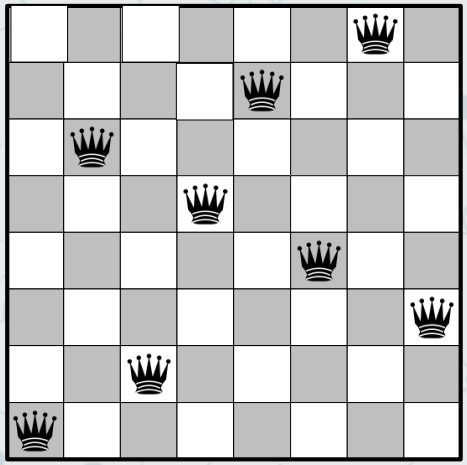
$h = 2$



$h = 0$

Example - n-Queens Problem

A state with $h = 17$ and the h-value for each possible successor



18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	14	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18

A local minimum in the 8-queens state space

Gradient descent is likely to get stuck in ~86% of the time in the 8-qeunns problem

Hill Climbing (Gradient Ascent) - Formal Definition

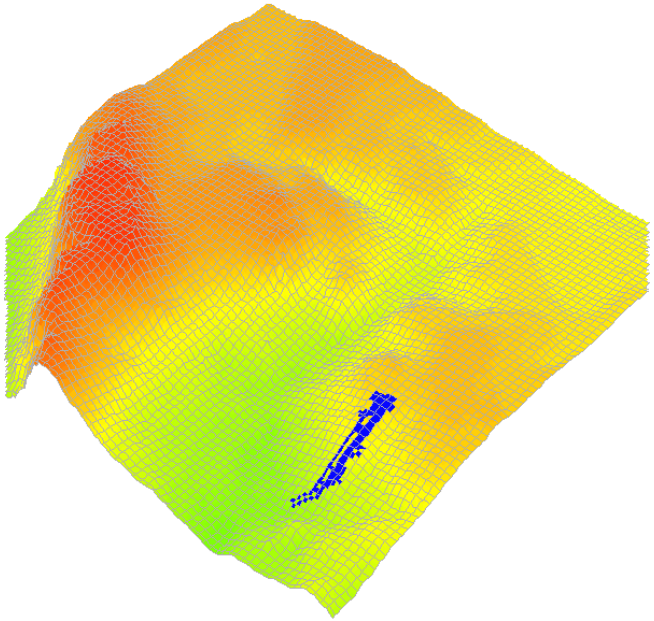
Also called Greedy Local Search since it grabs a good neighbor state without thinking ahead about where to go next.
Formally:

- $current \leftarrow MAKE-NODE(INITIAL-STATE)$
- loop:
  - $neighbor \leftarrow$ highest valued successor of current
  - if $f(neighbor) < f(current)$ return $STATE(current)$
  - else $current \leftarrow neighbor$

This is an iterative improvement algorithm but in this basic version it has problems with local maxima.

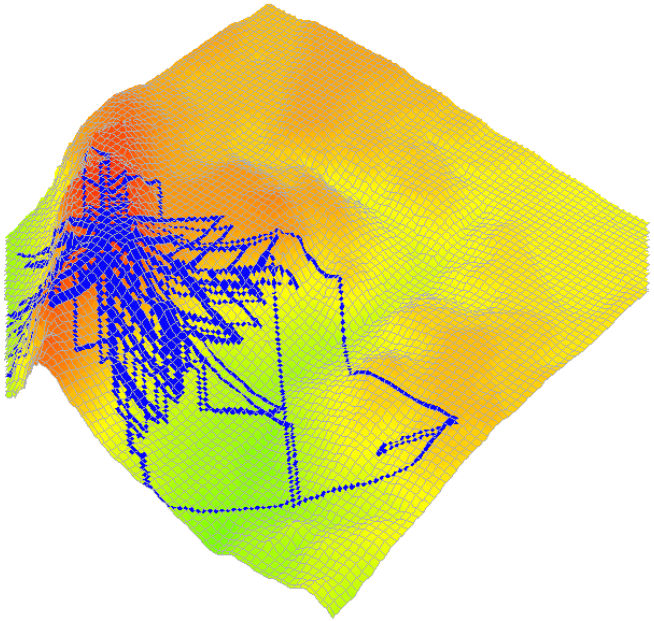
Hill Climbing - Heuristic Illustrated

A goal directed hill-climbing agent has failed to locate the highest point. This is the **local optimum problem**



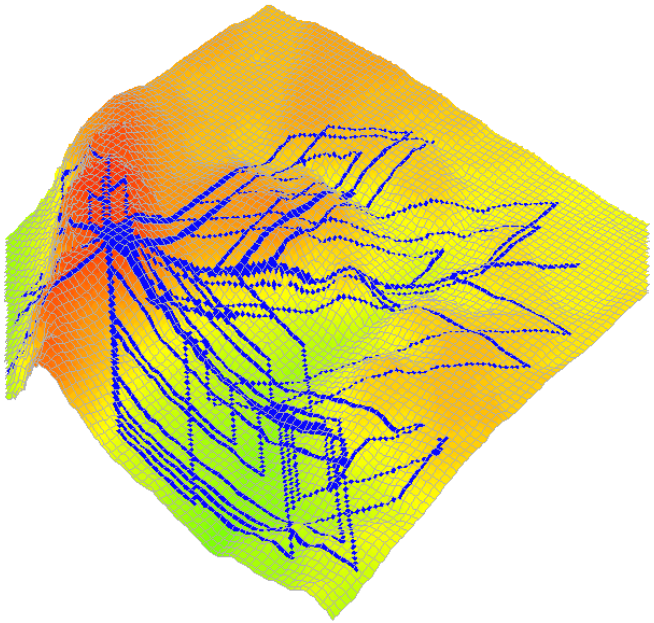
Hill Climbing - Heuristic Illustrated

This agent overcame the local optimum problem by accepting a moderate risk in the hope to find somewhere higher

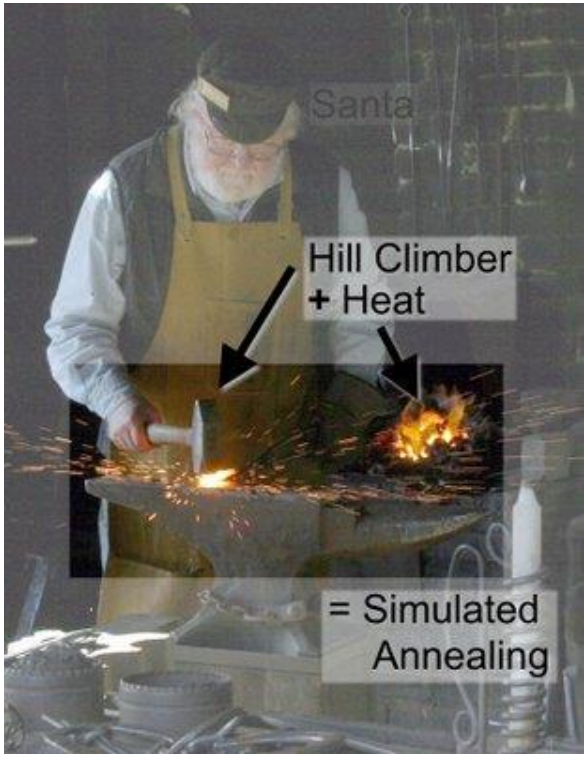


Hill Climbing - Heuristic Illustrated

There is a **trade-off** – an agent that takes very great risks finds the **highest location**, but also spends **more time** searching

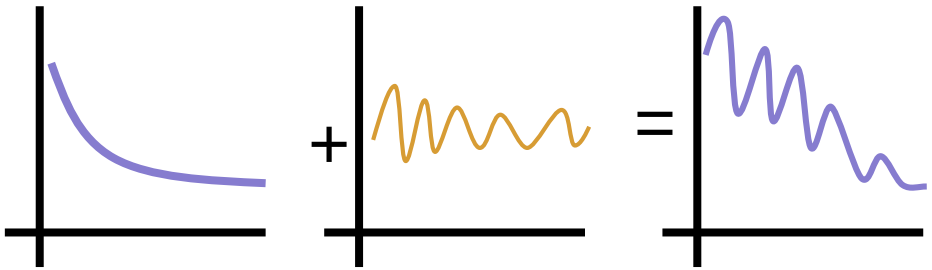


Simulated Annealing



Motivation for Simulated Annealing

- Hill-climbing is incomplete (might stuck in local maxima)
- On the other hand, choosing successor uniformly at random is complete but extremely inefficient.
- So why not combine the two ?



- Escape local maxima by allowing "bad" moves – but gradually decrease their size and frequency

Simulated Annealing Formally

- **Heuristic**: Objective function $F: \mathcal{S} \rightarrow \mathbb{R}$
- $\boldsymbol{fringe}$: Complete ordering by F
- **New nodes**: One successor of Current randomly selected
- **Reset**: The new node is accepted if
  - $F(new) > F(current)$ or
  - with some exponentially decaying probability proportional, a “bad” move

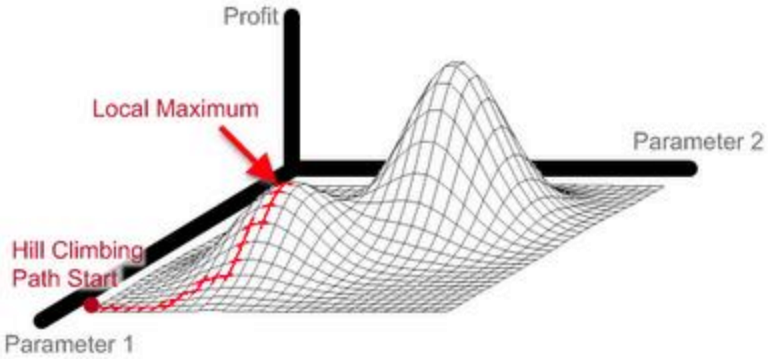
Otherwise, the active set is restored to previous (i.e. Current)

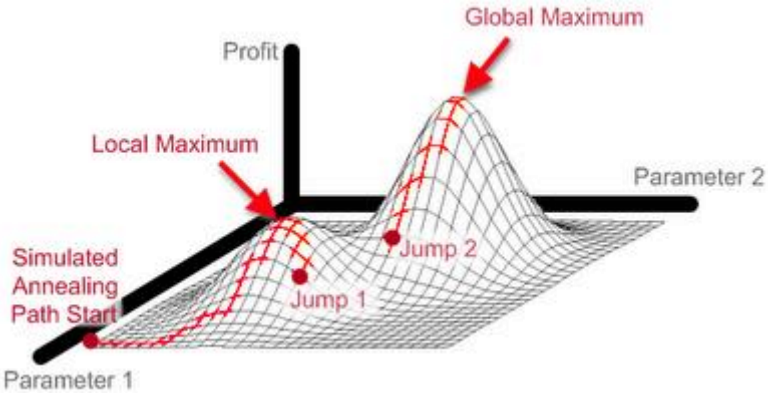
Simulated Annealing Algorithm

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

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

Simulated Annealing Graphically





Local beam search

Keep track of k states instead of one:

- Initially: k random states
- Next: determine all successors of k states
- If any of successors is goal → finished
- Else
  - select k best from successors and repeat

Local beam search

Major difference with random-restart search – **Information is shared** among k search threads

Can suffer from lack of diversity (concentrated in small region of state space)

Stochastic variant: choose k successors with probability of choosing a given successor an increasing function of its value **– Stochastic Beam Search**

Genetic Algorithms

Genetic Algorithms – Motivation

- In Simulated Annealing we maintained just one solution and successors generated only by modifying a single state ("mutation")
- Why not maintaining a pool of solutions?
- Why not generating successors states by combining two parent states and mutating?
- We are talking about breaking the $fringe$ management perspective - requires extension of search formalism

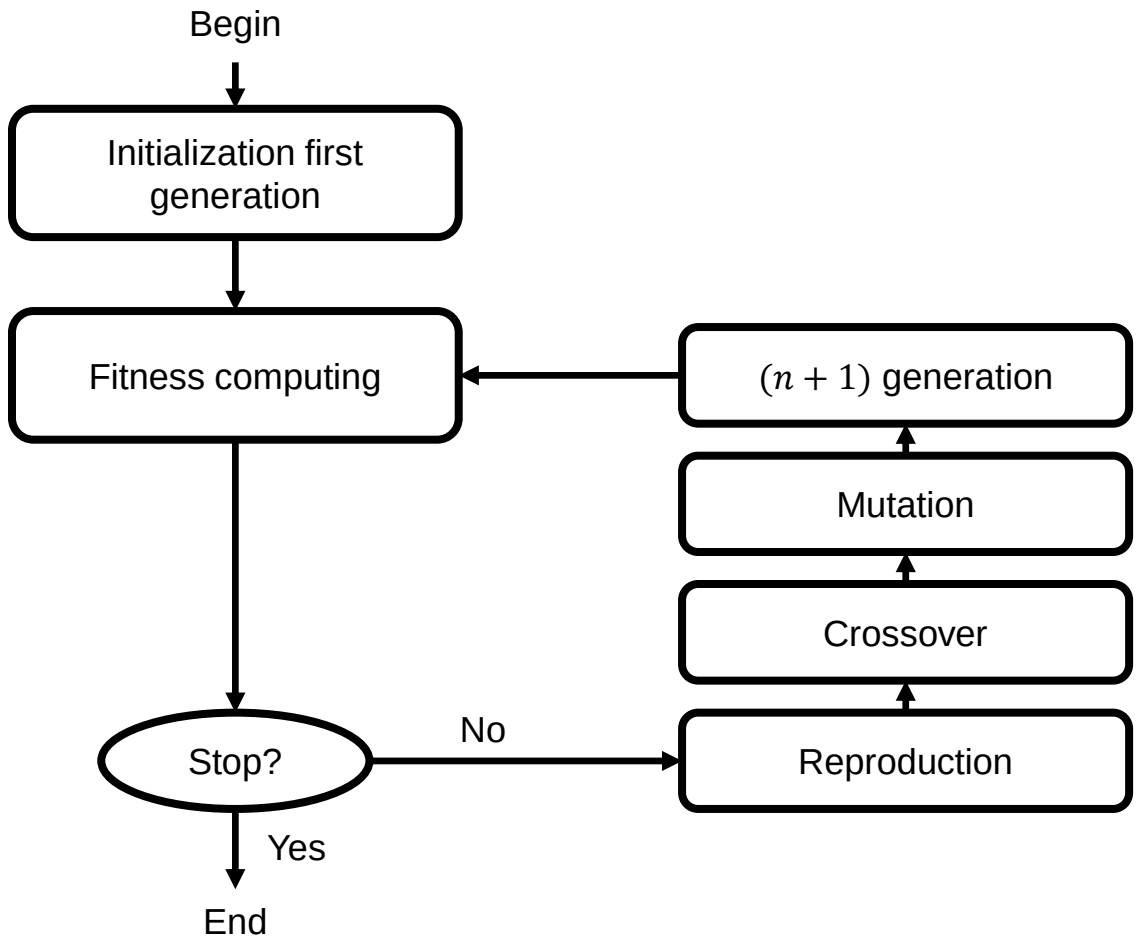
Genetic Algorithms Formally

The new formalism:

- Each state is called **individual**, represented by vector $v \in \mathbb{R}^n$
- Start with a set of random (or seeded in areas where optimal solution is likely) individuals, called **population**

- **Actions:**
  - **Reproduction**
  - **Selection**: The probability to be chosen for reproducing is related to some fitness function that rates the individuals (higher values for better states)
  - **Crossover**: Point of merge between to vectors (might be chosen randomly)
  - **Mutation**: Each location is subject to random mutation with some small independent probability (to reduce search locality)

Genetic Algorithms Flow Chart



Genetic Algorithms Formally

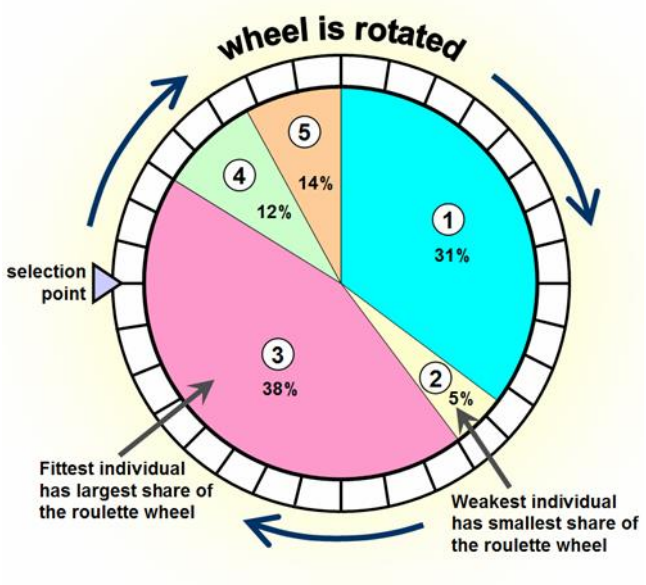
- **Heuristic**: Population performance (**fitness function**)
- $fringe$: Complete ordering by fitness function
- **New nodes**: by reproduction of the population
- **Reset**: $fringe$ modifications (for example, remove old generations – extinction of weak individuals)

Fitness Function

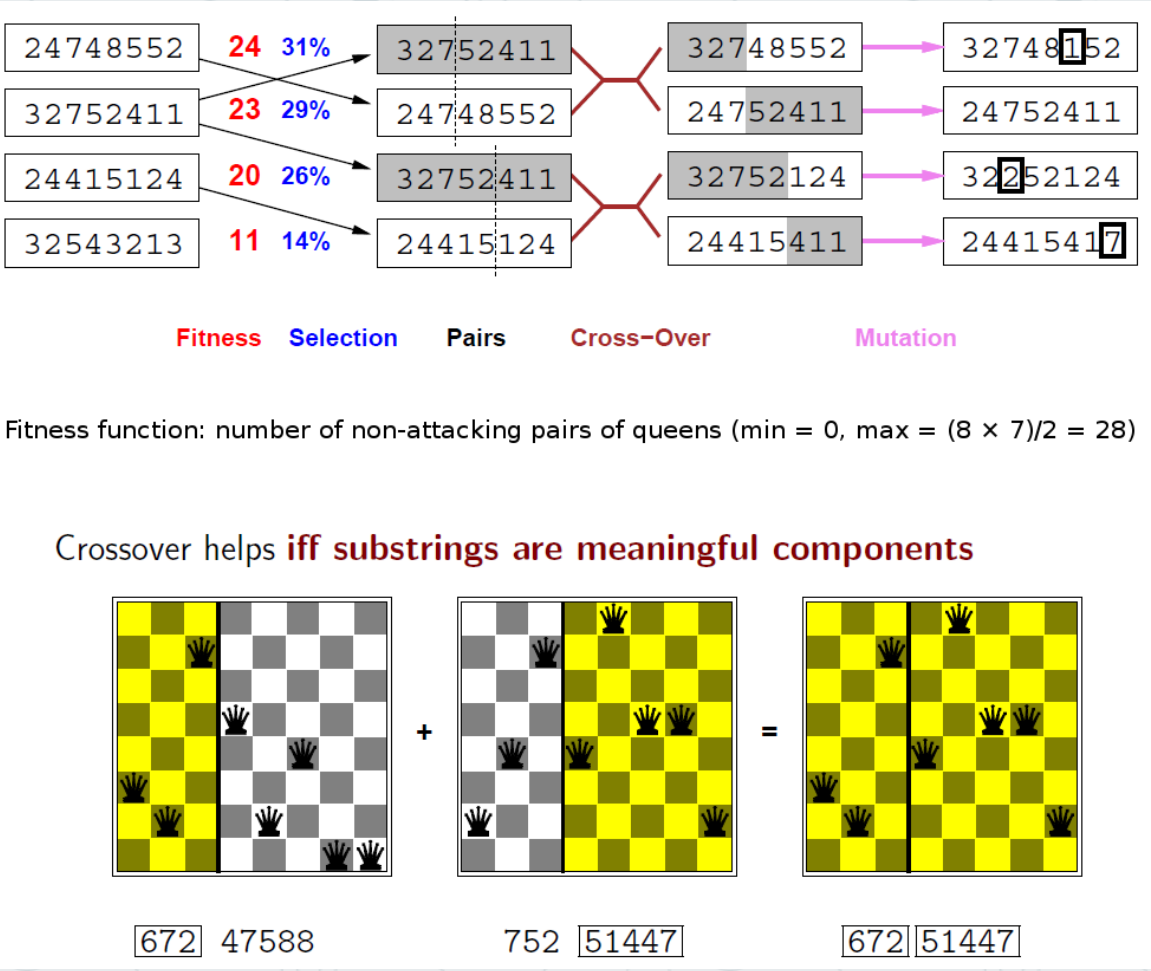
Most functions are stochastic and designed so that a small proportion of less fit solutions are selected.

This helps keep the diversity of the population large, preventing premature convergence on poor solutions.

For example, a population of size 5 roulette wheel selection:



Example - n-Queens Problem

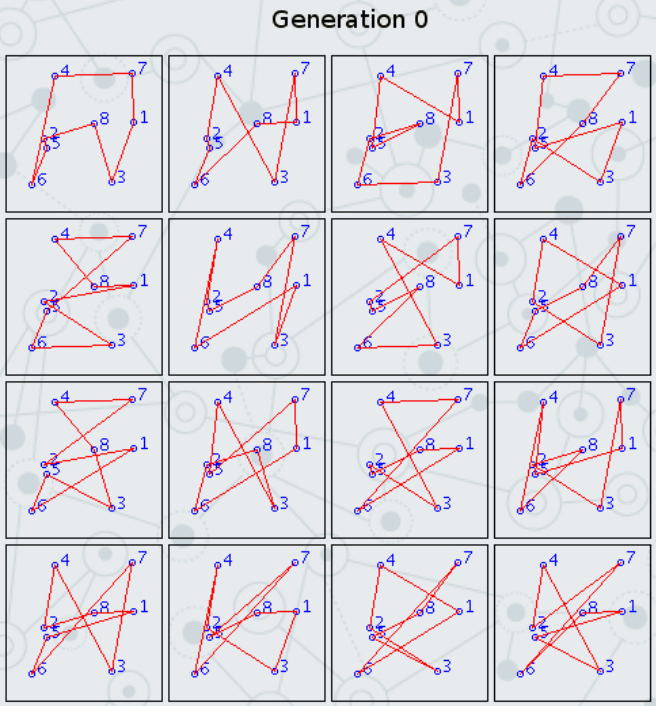


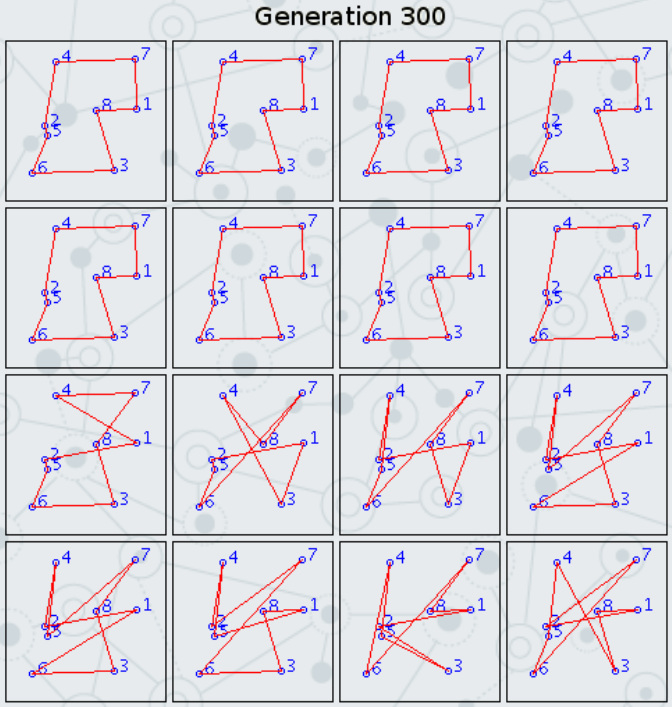
Example - Traveling Salesman Problem

- **Encoding**: each individual represents the order of cities in which the salesman will visit them
- **Fitness function**: the route length
- **Selection function**: stochastic based on the fitness function

Example - Traveling Salesman Problem

For example, if the population size is 16 (sorted by route length)



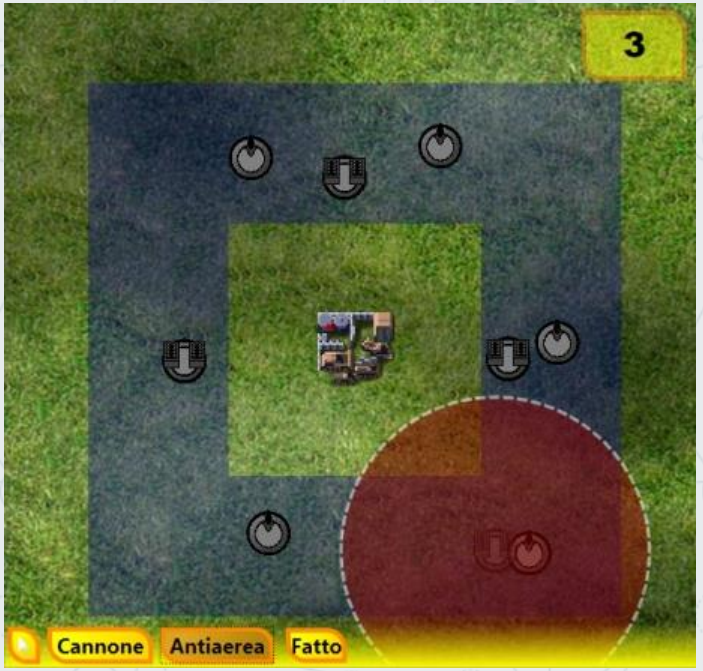


Example - Defense Grid Genome

Player's base at its center

At start, the player can place defensive structures around the base (in the blue area) according to budget:

- Canon tower can shoots quickly but cannot target airplanes
- Rocket tower is slower but can target airplanes
- Each tower has limited range and firepower but never misses



Example - Defense Grid Genome

The objective of the computer is to destroy the player's base by touching it (the base has 20 health points)

When the base reach 0 health the player loses

The player goal is to place his defenses to maximize resistance time

Example - Defense Grid Genome

The computer generate units (tank or airplane) that will start on the edge of the map and move towards the base

Each unit chooses "toughness" (number of hits required to destroy it) which come on the expanse of unit speed

Genome: (start position, unit type, speed, toughness)

Example - Defense Grid Genome

4 bits for starting position (16 grids)

1 bit for unit type (tank / airplane)

3 bits for toughness/speed (on scale [1-5])

Example - Defense Grid Genome

The closer the unit go to base, the more points it earns.

If the unit hit the base, we will double its fitness.

Example - Defense Grid Genome

Run GA procedure:

- The best individuals will be picked and their genome used to create the next generation of attackers.
- The next enemy wave will be unleashed against the base, and so on - until the weaknesses in the player's defenses are discovered and overcome through natural selection.
- The adaptation to the strategy used by the player is usually very quick, about 5 rounds is enough for the agent to find a good strategy (due to simple genome).

Example - Defense Grid Genome

Like Stochastic Beam Search, genetic algorithms combine an uphill tendency with random exploration and exchange of information among parallel search threads.

Crossover is one potential advantage of genetic algorithms.

Intuitively, it combines large blocks that have evolved independently, raising the level of granularity at which the search operates.

Summary

Heuristic function as estimation of cost to goal.

Good heuristics dramatically improve search performance.

Heuristics with special properties (admissible, consistent) can be used to find optimal solution.

Deterministic and probabilistic meta-heuristics (Local Search, Simulated Annealing) for solving optimization problems.

Extended search formalism for Genetic Algorithms.