ALGORITHMS IN AI & DATA SCIENCE 1 (AKIDS 1)

Heuristic Search

Prof. Dr. Goran Glavaš

8.1.2024

Content

- Heuristics
- Greedy Best-First & Hill-Climbing Search
- A* Algorithm
- Heuristics Revisited
- Example: Path Finding on Terrain Map

Based on the materials from Prof. Dr. Jan Šnajder:
https://www.fer.unizg.hr/_download/repository/AI-3-HeuristicSearch.pdf

Recap: State Space Search

- We will denote the set of all states (state space) with **S**
  - The state space is commonly **so large** that we can't iteratively list all states
  - All states in the space are not really „known" in advance
  - When in state s, we typically only then compute the set of possible next states

State space search

A state space search problem is defined with a triple (s_0, *succ*, goal) where $s_0 \in S$ is the **initial state**, *succ*: **S** → ℘(**S**) is the **successor function** that for some state s returns a set of states that we can **transition to** from s, and *goal*: **S** → {True, False} is a **predicate** (function that returns a boolean value) that for a given state s determines if s is a **goal state** or not (there can be multiple states that satisfy the goal predicate). A state space search (typically) ends as soon as any goal state is found.

Recap: State Space Search

- There are generally two types of search
- **Uninformed (blind) search**
  - No additional information about the problem, that could indicate whether one state is perhaps closer to the goal state than another state
- **Informed (directed, heuristic) search**
  - Additional information helps avoid some states and speed up the search
  - Problem-specific estimate of <u>state’s distance from the goal</u> is available

Heuristic Search: Motivation

- **Uninformed/blind search** relies only on exact information (initial state, operators, goal predicate)
  - Starting from an **initial state**, we try to reach a **goal state**
  - Always considering all possible transitions, without knowing which is more promising

- Blind search doesn't leverage additional information about the **nature of the problem** that might make the **search more efficient**

Heuristics

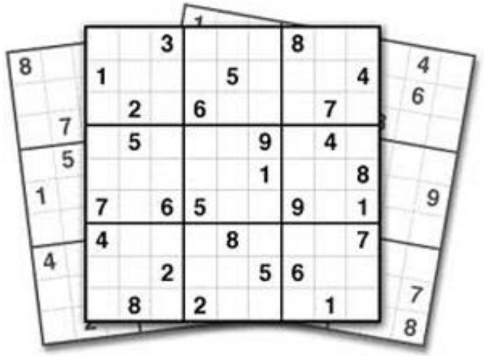
- If we have an **idea in which direction to look** for the solution, why not use this information to improve the search?

- **Heuristics = problem-specific rules about the nature of the problem**
  - **Purpose**: direct the search towards the goal so it becomes more efficient

Heuristic function

Heuristic function h: **S** $\rightarrow \mathbb{R}^{+}$ assigns to each state $s \in$ **S** an **estimate** of the **distance between that state and the goal state**

Typical state space search problems



Example: 8-Puzzle

- **Q:** think of some examples of heuristic functions
  - **Estimates** of **distances** between the state and goal state
  - h_1: number of displaced squares
  - h_2: sum of city-block (Manhattan) distances between the current and correct/final position of each square/number
  - Note that $h_1(s) \leq h_2(s)$
- If the **heuristic** is „good” then it can substantially reduce the number of states that are „opened” before finding the **goal**

initial state

8		7
6	5	4
3	2	1

goal state

1	2	3
4	5	6
7	8	

Heuristic Search

- **Heuristic search algorithms** decide on the order of „opening” nodes in the search tree based on nodes’ **values for a given heuristic** *h*

- Greedy algorithms
  - Greedy best-first search
  - Hill-climbing

- Optimal* algorithm
  - A-Star (A*)
    - *Assuming the heuristic function satisfies certain properties

Content

- Heuristics
- Greedy Best-First Search & Hill-Climbing
- A* Algorithm
- Heuristics Revisited
- Example: Path Finding on Terrain Map

Recap: General Search Algorithm

- We define a general search algorithm
  - Think of it as **abstract search algorithm**
- Contains functions, whose concrete implementation depends on the choice of the actual search algorithm
- (Dynamic) Set of **open nodes** – nodes in the search tree that we reached: a „frontier" of the search tree
- **Generic (abstract) functions**:
  - *take*(*l*) – gets the next node from the set of open nodes *l*
  - *expand(n, succ)* – expands node n using *succ*
  - *insert(n, l) – adds node n to the list of open nodes l*

```
search(s0, succ, goal)
  open = [init(s0)]
  while len(open) > 0
    n = take(open)
    if goal(state(n))
      return n
    for m in expand(n, succ)
      insert(m, open)
  return False
```

Greedy Best-First Search

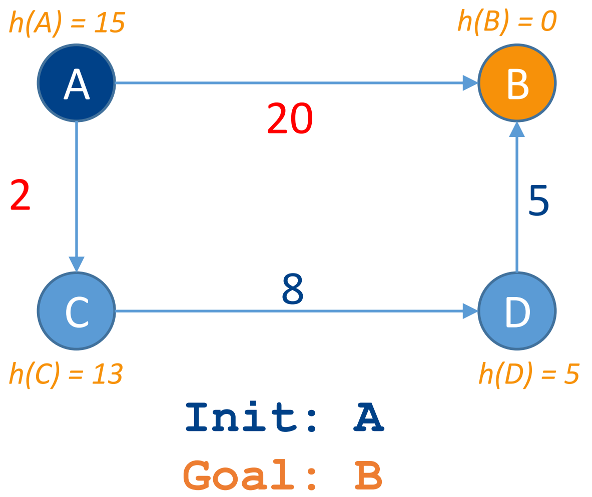
- If we could somehow know which of the (states of) open nodes is the **closest** to a **goal state,** we'd pick that state
- **Heuristics** estimate how close nodes (their states) are to the goal state
- **Greedy best-first**: in each step takes the node from *open* with **minimal** *h* score
  - Like in UCS, *open* is a **priority queue**
  - Only the priority is now given with *h*(s) and not the cost of reaching s, *cost*(s)
  - In pseudocode – s.heur is the *h*(s)

```
greedy-best-fs(s0, succ, goal, h)
  open = [init(s0)]
  while len(open) > 0
    n = extract-min(open) # min of h
    if goal(state(n))
      return n
    for m in expand(n, succ)
      m.heur = h(state(m))
      insert(m, open) # heap insertion
                      # according to m.heur
  return False
```

Greedy Best-First Search

- Always chooses the node that appears **the closest to the goal**
- The **chosen (whole) path** may not be optimal, but greedy best-first search doesn't backtrack
  - **Q:** even if the **heuristic is <u>perfect</u>** (h(s) = real minimal distance from s to the goal state), greedy search may not be optimal. **Why?**
  - Greedy doesn't consider the ***cost*(s)**, only **h(s)**
  - In reality, we don't have an oracle/perfect heuristic

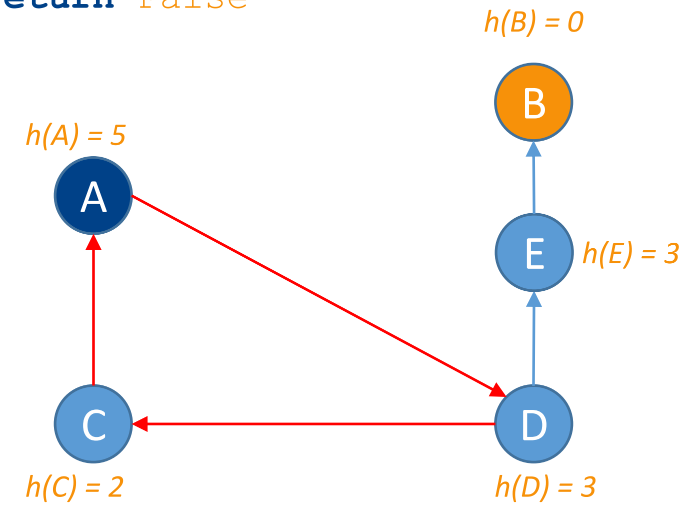
```
greedy-best-fs(s_0, succ, goal, h)
  open = [init(s_0)]
  while len(open) > 0
    n = extract-min(open) # min of h
    if goal(state(n))
      return n
    for m in expand(n, succ)
      m.heur = h(state(m))
      insert(m, open) # heap insertion
                      # according to m.heur
  return False
```



Greedy Best-First Search

- So, **greedy best-first search** is not optimal
- It is also not complete (unless we explicitly keep track of visited states)
  - There can be a cycle of states with locally minimal value of h
- **Time complexity: O(b^m)**
  - This is if we don't consider the maintenance of the priority queue, otherwise **O(b^m log b^m)**
- **Space complexity: O(b^m)**

```
greedy-best-fs(s0, succ, goal, h)
  open = [init(s0)]
  while len(open) > 0
    n = extract-min(open)  # min of h
    if goal(state(n))
      return n
    for m in expand(n, succ)
      m.heur = h(state(m))
      insert(m, open)  # heap insertion
                       # according to m.heur
  return False
```



Init: A
Goal: B

Hill-Climbing Search

- Let's ignore for a moment that greedy best-first search is not optimal
- Space complexity of greedy best-first search – **O(b^m)** – would be problematic, even if it was optimal

- **Hill-Climbing** is also „greedy" in principle, but does not keep track of all open nodes at all
  - Considers as next state only the ones reachable from current state
  - And out of those, picks the one with minimal *h*

  - **GFBS**: selects state with „globally" (from all known states so far) minimal *h*
  - **HC**: selects state with locally (only states reachable from current) minimal *h*

Hill-Climbing Search

- **Hill-Climbing** is also „greedy" in principle, but does not keep track of all open nodes at all
  - Considers as next state only the ones reachable from current state
  - And out of those, picks the one with minimal *h*

  - **HC**: selects state with locally (only states reachable from current) minimal *h*

```
hill-climbing(s0, succ, goal, h)
  n = init(s0)
  while True:
    if goal(state(n))
      return n

    M = expand(n, succ)
    if len(M) = 0
      return False

    m = min(M, h)
    if h(state(m)) > h(state(n))
      return False

    n = m
```

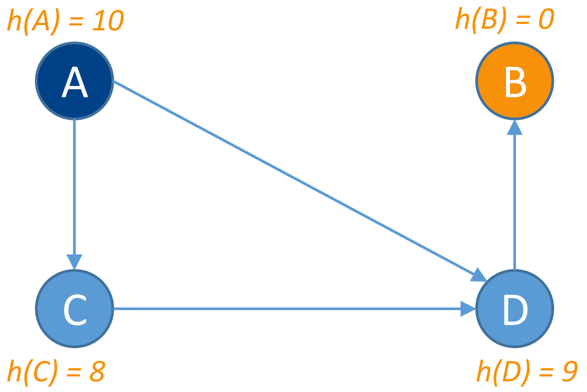
Hill-Climbing Search

- **Hill-Climbing** is easily trapped in the so-called local optima and therefore obviously
  - Not complete
  - Not optimal

- **Random restart**
  - Start many times from different initial states

- **Time complexity**: **O(m =|S|)**
- **Space complexity**: **O(1)**
  - No „open" set!

```
hill-climbing(s0, succ, goal, h)
  n = init(s0)
  while True:
    if goal(state(n))
      return n
    M = expand(n, succ)
    if len(M) = 0
      return False
    m = min(M, h)
    if h(state(m)) > h(state(n))
      return False
    n = m
```



Content

- Heuristics
- Greedy Best-First & Hill-Climbing Search
- A* Algorithm
- Heuristics Revisited
- Example: Path Finding on Terrain Map

A* Algorithm

- In principle similar to the greedy best-first algorithm, but makes the selection based on not just the heuristic but **cost + heuristic**

- As in **UCS**, we compute the **cost** of the node when we create it

- **A*** selects the node from „open" as the node n with minimal:

$$f(n) = \mathbf{cost}(n) + h(\mathbf{state}(n))$$

```
expand(n, succ)
  sstates = succ(state(n))
  nodes = []
  for (s, c) in sstates
    nodes = nodes ∪ (s, cost(n) + c)
  return nodes
```

A* Algorithm

- **A*** selects the node from „open" as the node n with minimal:

  $f(n) = \textbf{cost}(n) + h(\textbf{state}(n))$

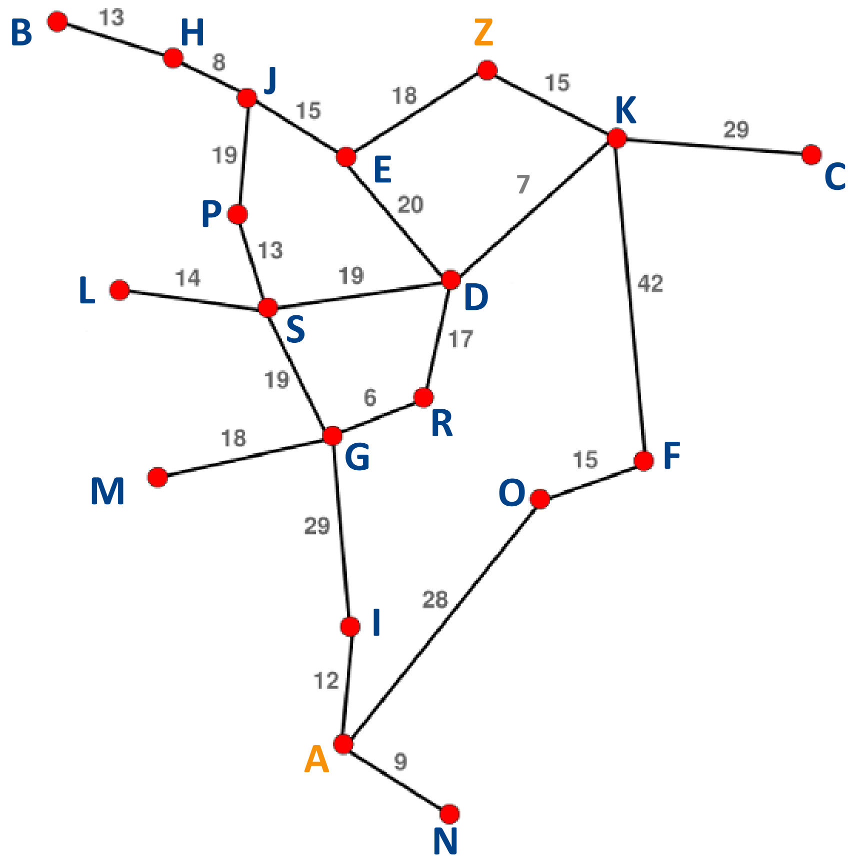
- *„open"* is (again) a **priority queue**

- It's possible to revisit the same state with **smaller total cost (c+h)**
  - Keep track of the smallest discovered cost for each **state**
  - Hashtable of minimal known cost for states (`visited`: key state, value is minimal known *cost* for the state)

```
astar-search(s0, succ, goal, h)
  visited = {}
  open = [(s0, h(s0))]
  visited[s0] = 0

  while len(open) > 0
    n = extract-min(open)
    if goal(state(n))
      return n
    for m in expand(n, succ)
      f = cost(m) + h(state(m))
      if state(m) not in visited
          visited[state[m]] = cost(m)
          insert(m, open, f) # heap insertion
      elif cost(m)< visited[state[m]]
        visited[state[m]] = cost(m)
        inop = False
         for l in open
            if state(l) == state(m)
                decrease-prio(open, l, f)
                inopen = True
                break
        if not inop
         insert(m, open, f)
```

A* Algorithm: Example



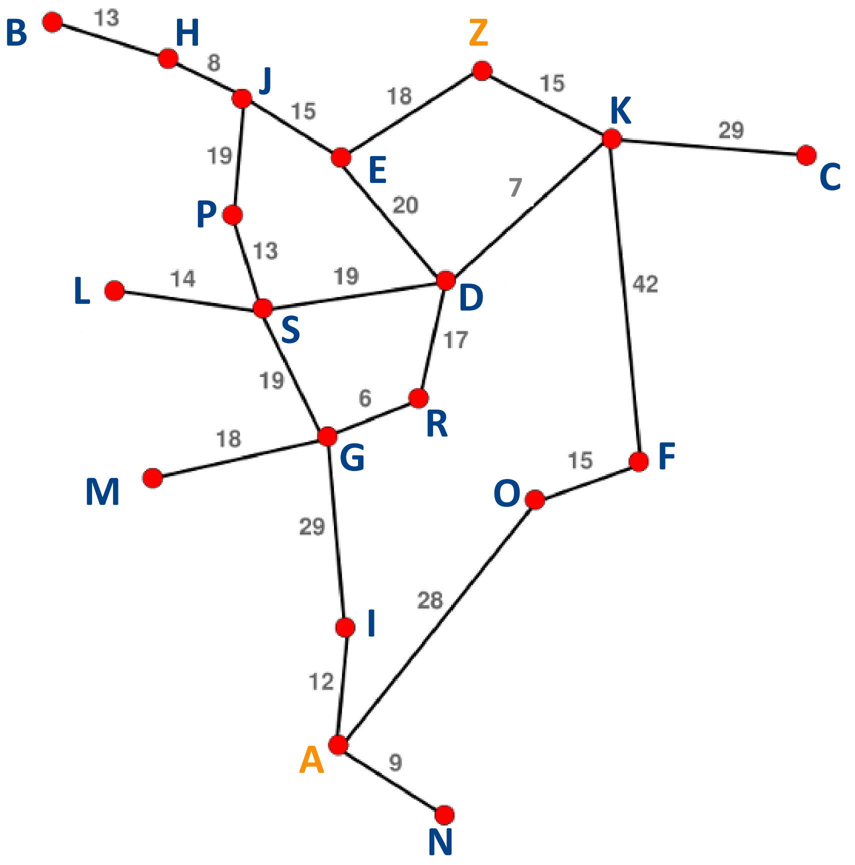
Task: shortest path from **A** to **Z**

Heuristic: h(**X**) is the **air distance** from **X** to **Z**

h(A) = 57	h(J) = 17
h(B) = 31	h(K) = 13
h(C) = 26	h(L) = 32
h(D) = 17	h(M) = 40
h(E) = 12	h(N) = 61
h(F) = 35	h(O) = 35
h(G) = 30	h(P) = 20
h(H) = 21	h(R) = 27
h(I) = 47	h(S) = 25

A* Algorithm: Example

Task: shortest path from **A** to **Z**
Heuristic: *h*(**X**) is the **air distance** from **X** to **Z**



h(A) = 57
h(B) = 31
h(C) = 26
h(D) = 17
h(E) = 12
h(F) = 35
h(G) = 30
h(H) = 21
h(I) = 47
h(J) = 17
h(K) = 13
h(L) = 32
h(M) = 40
h(N) = 61
h(O) = 35
h(P) = 20
h(R) = 27
h(S) = 25
h(Z) = 0

Initialization:
open = [(**A**, 0+57)]
visited = {(**A**: 0)}

1. Iteration (expand(**A**, 0+57))
open = [(**I**, 12+47), (**O**, 28+35), (**N**, 9+61)]
visited = {**A**: 0, **I**: 12, **N**: 9, **O**: 28}

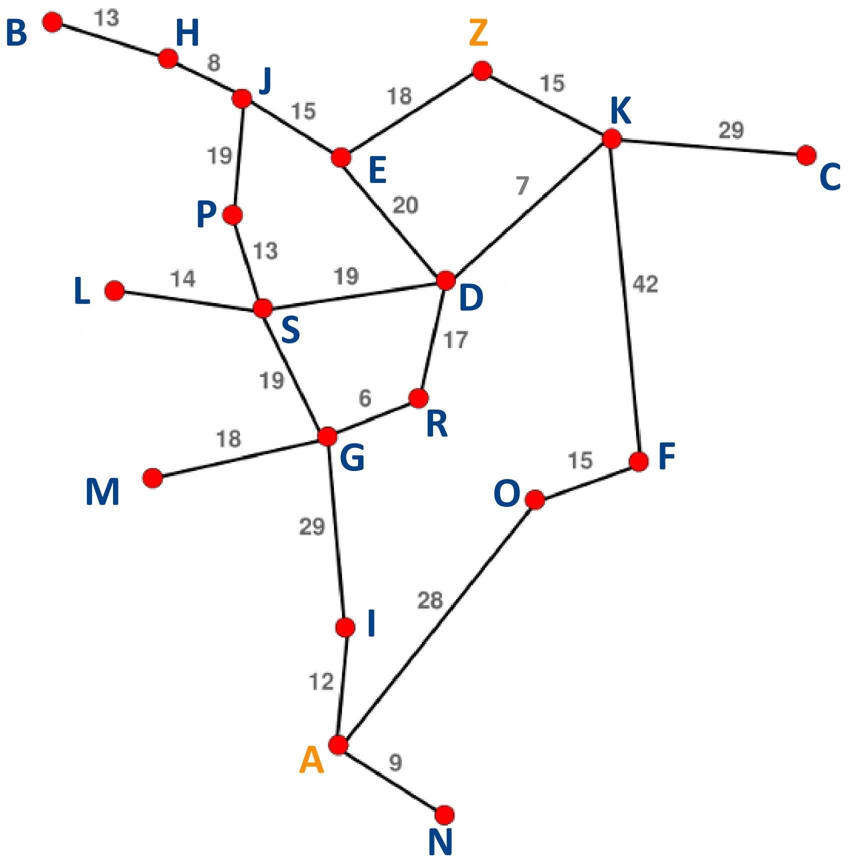
2. Iteration (expand(**I**, 12+47))
open = [(**O**, 28+35), (**N**, 9+61), (**G**, 41+30)]
visited = {**A**: 0, **I**: 12, **N**: 9, **O**: 28, **G**: 41}

3. Iteration (expand(**O**, 28+35))
open = [(**N**, 9+61), (**G**, 41+30), (**F**, 43+35)]
visited = {**A**: 0, **I**: 12, **N**: 9, **O**: 28, **G**: 41, **F**: 43}

4. Iteration (expand(**N**, 9+61))
open = [(**G**, 41+30), (**F**, 43+35)]
visited = {**A**: 0, **I**: 12, **N**: 9, **O**: 28, **G**: 41, **F**: 43}

A* Algorithm: Example

Task: shortest path from **A** to **Z**
Heuristic: *h*(**X**) is the **air distance** from **X** to **Z**



h(A) = 57
h(B) = 31
h(C) = 26
h(D) = 17
h(E) = 12
h(F) = 35
h(G) = 30
h(H) = 21
h(I) = 47
h(J) = 17
h(K) = 13
h(L) = 32
h(M) = 40
h(N) = 61
h(O) = 35
h(P) = 20
h(R) = 27
h(S) = 25
h(Z) = 0

4. Iteration (expand(**N**, 9+71))
open = [(**G**, 41+30), (**F**, 43+35)]
visited = {**A**: 0, **I**: 12, **N**: 9, **O**: 28, **G**: 41, **F**: 43}

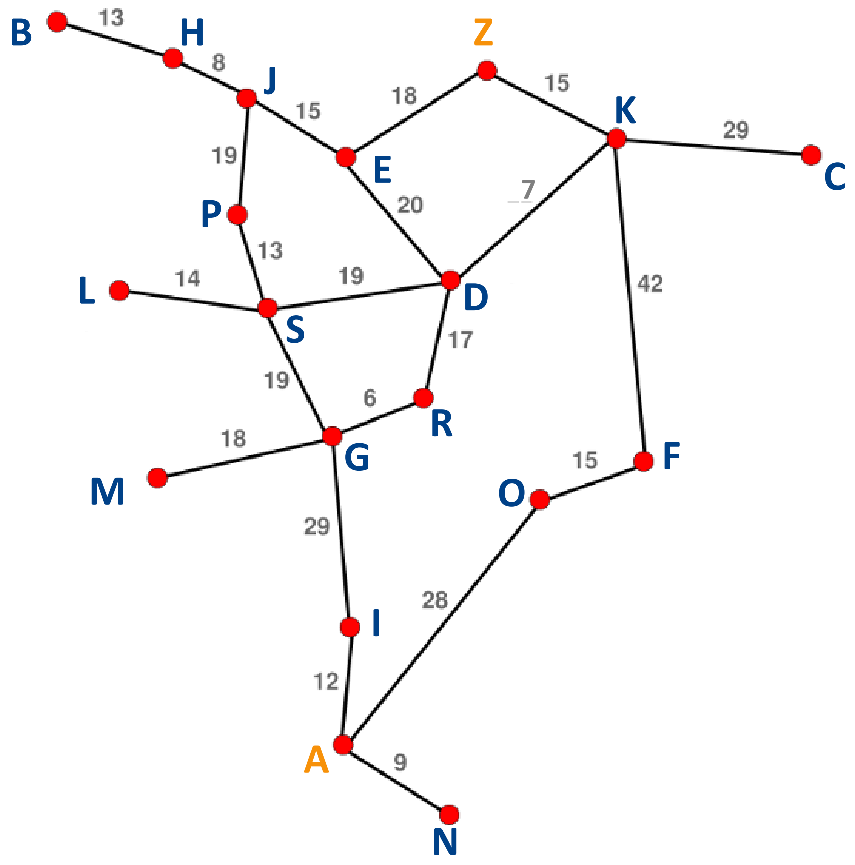
5. Iteration (expand(**G**, 41+30))
open = [(**R**, 47+27), (**F**, 43+35), (**S**, 60+25), (**M**, 59+40)]
visited = {**A**: 0, **I**: 12, **N**: 9, **O**: 28, **G**: 41, **F**: 43, **R**: 47, **S**: 60, **M**: 59}

6. Iteration (expand(**R**, 47+27))
open = [(**F**, 43+35), (**D**, 66+17) (**S**, 60+25), (**M**, 59+40)]
visited = {**A**: 0, **I**: 12, **N**: 9, **O**: 28, **G**: 41, **F**: 43, **R**: 47, **S**: 60, **M**: 59, **D**: 66}

7. Iteration (expand(**F**, 43+35))
open = [(**D**, 66+17), (**S**, 60+25), (**M**, 59+40), (**K**, 85+13)]
visited = {**A**: 0, **I**: 12, **N**: 9, **O**: 28, **G**: 41, **F**: 43, **R**: 47, **S**: 60, **M**: 59, **D**: 66, **K**: 85}

A* Algorithm: Example

Task: shortest path from **A** to **Z**
Heuristic: *h*(**X**) is the **air distance** from **X** to **Z**



h(A) = 57
h(B) = 31
h(C) = 26
h(D) = 17
h(E) = 12
h(F) = 35
h(G) = 30
h(H) = 21
h(I) = 47
h(J) = 17
h(K) = 13
h(L) = 32
h(M) = 40
h(N) = 61
h(O) = 35
h(P) = 20
h(R) = 27
h(S) = 25
h(Z) = 0

7. Iteration (expand(**F**, 43+35))
open = [(**D**, 66+17), (**S**, 60+25), (**M**, 59+40), (**K**, 85+13)]
visited = {**A**: 0, **I**: 12, **N**: 9, **O**: 28, **G**: 41, **F**: 43, **R**: 47,
S: 60, **M**: 59, **D**: 66, **K**: 85}

8. Iteration (expand(**D**, 66+17))
→ (**K**, 66+7 = 73) (h = 13)
open = [(**D**, 66+17), (**S**, 60+25), (**M**, 59+40), (**K**, 85+13)]
visited = {**A**: 0, **I**: 12, **N**: 9, **O**: 28, **G**: 41, **F**: 43, **R**: 47,
S: 60, **M**: 59, **D**: 66, **K**: 85}

→ open = [(**D**, 66+17), (**S**, 60+25), (**K**, 73+13), (**M**, 59+40)]
visited = {**A**: 0, **I**: 12, **N**: 9, **O**: 28, **G**: 41, **F**: 43, **R**: 47,
S: 60, **M**: 59, **D**: 66, **K**: 73}

…

A* Algorithm: Properties

- A* is **complete**
  - Cannot end in infinite loop
  - Eventually reaches goal state (if reachable)
- If heuristic *h* is **optimistic**, **A*** is **optimal**

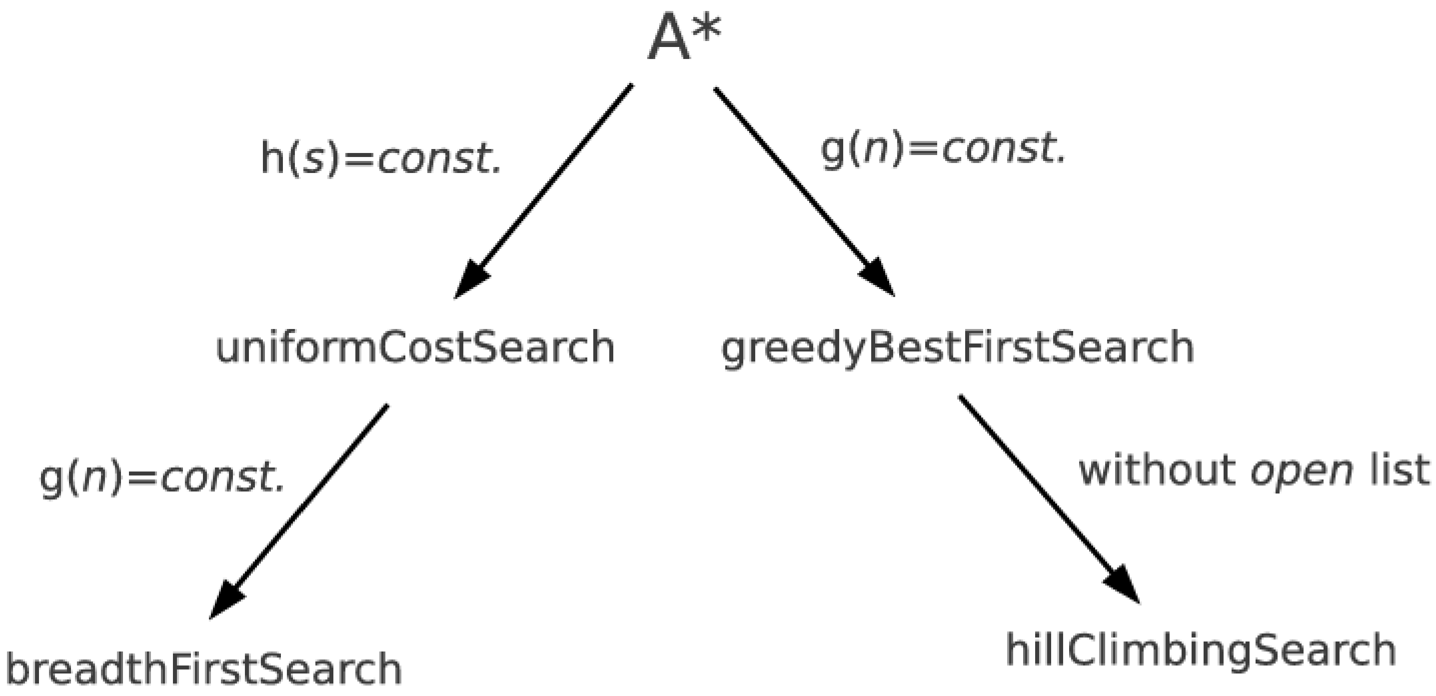
- **Time and space complexity?**
  - If *h* is optimistic, then no state will be expanded more than once
  - Thus, complexity $\mathbf{O(b^{d+1})}$ becomes $\mathbf{O(\min(b^{d+1}, b|S|))}$
    - In most problems, $\mathbf{b|S| < b^d}$

```
astar-search(s0, succ, goal, h)
  visited = {}
  open = [(s0, h(s0))]
  visited[s0] = 0

  while len(open) > 0
    n = extract-min(open)
    if goal(state(n))
      return n
    for m in expand(n, succ)
      f = cost(m) + h(state(m))
      if state(m) not in visited
        visited[state[m]] = cost(m)
        insert(m, open, f) # heap insertion
      elif cost(m)< visited[state[m]]
        visited[state[m]] = cost(m)
        inop = False
        for l in open
          if state(l) == state(m)
            decrease-prio(open, l, f)
            inopen = True
            break
        if not inop
          insert(m, open, f)
```

Putting search algorithms in perspective

$g(n) = \boldsymbol{cost}(n)$



Content

- Heuristics
- Greedy Best-First & Hill-Climbing Search
- A* Algorithm
- Heuristics Revisited
- Example: Path Finding on Terrain Map

Properties of heuristics

Optimistic heuristic

Heuristic function *h* is **optimistic** (or **admissible**) if and only if it **never** overestimates, that is, its value is never greater than the true cost needed to reach the goal:

$$\forall s \in \mathbf{S}.\ h(s) \leq h^*(s)$$

where $h^*(s)$ is the **true minimal cost** of reaching the goal state from state s.

- If the heuristic is not optimistic, the search may bypass the optimal path because it seems more expensive than it really is

Example: 8-Puzzle

- **Q:** Are these two heuristics optimistic?
  - **Cost**: number of moves
  - $h_1(s)$: number of displaced squares
  - $h_2(s)$: sum of city-block (Manhattan) distances between the current and correct/final position of each square/number
- **What about**:
  - $h_3(s) = 0$?
  - $h_4(s) = 1$?
  - $h_5(s) = h^*(s)$?
  - $h_6(s) = \min(2, h^*(s))$?
  - $h_7(s) = \max(3, h^*(s))$?

initial state

8		7
6	5	4
3	2	1

goal state

1	2	3
4	5	6
7	8	

Consistent heuristics

- For an optimistic heuristic *h*, there exists an **upper bound** for f = cost + h (across all states)

$$f(n) = \mathbf{cost}(n) + h(\mathbf{state}(n)) \leq C$$

- C = max. value of *f* than any node during **A* search** with *h* would have
- As we search, the value *f*(n) for the states we expand may generally increase and decrease
- If *f*(n) would only **monotonically increase** as we execute A*
  - Guarantee that once expanded the first time (extract-min), a state cannot be reached with smaller *f*
  - No need to check and decrease priority in *open*!
    - **Faster execution!**

```
astar-search(s0, succ, goal, h)
 visited = {}
 open = [(s0, h(s0))]
 visited[s0] = 0

 while len(open) > 0
   n = extract-min(open)
   if goal(state(n))
     return n
   for m in expand(n, succ)
     f = cost(m) + h(state(m))
     if state(m) not in visited
          visited[state[m]] = cost(m)
          insert(m, open, f)  # heap insertion
     elif cost(m)< visited[state[m]]
        visited[state[m]] = cost(m)
        inop = False
        for l in open
          if state(l) == state(m)
              decrease-prio(open, l, f)
              inopen = True
              break
        if not inop
          insert(m, open, f)
```

Consistent heuristics

- When is a **heuristic** *h* **consistent**?
  - f = **cost**(n) + *h*(**state**(n)) cannot drop, this means that **drop** in *h* for **neighboring states** s_1, s_2 cannot be larger than cost of the transition $c(s_1, s_2)$
  - $\forall n_2 \in \textit{expand}(n_1) \rightarrow$

$$f(n_2) \geq f(n_1)$$

$$\mathbf{cost}(n_2) + h(\mathbf{state}(n_2)) \geq \mathbf{cost}(n_1) + h(\mathbf{state}(n_1))$$

$$(\mathbf{cost}(n_1) + c(s_1, s_2)) + h(s_2) \geq \mathbf{cost}(n_1) + h(s_1)$$

$$c(s_1, s_2) + h(s_2) \geq h(s_1)$$

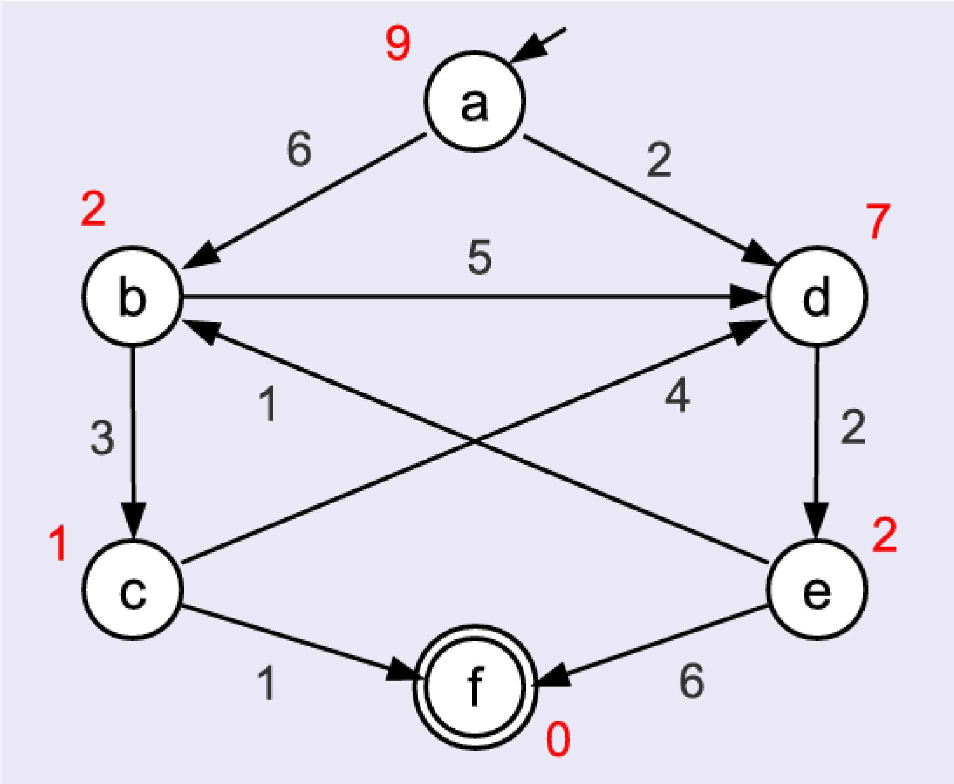
$$c(s_1, s_2) \geq h(s_1) - h(s_2)$$

- A **consistent heuristic** is necessarily **optimistic,** but not vice-versa
  - Still, most optimistic heuristics used in practical problems are also consistent

Example: heuristics properties

$h(n)$
$c(s_1, s_2)$

Q1: Is the heuristic **optimistic**?
Q2: Is it **consistent**?



Comparing heuristics

Optimistic heuristic

Let A_1^* and A_2^* be two optimal **A*** search algorithms (for the same problem) with corresponding heuristics h_1 and h_2. We say that A_1^* **dominates** (or is **more informed** than) A_2^* if and only if: $\forall s \in \mathbf{S}.\ h_1(s) \geq h_2(s)$

- You can say also that h_2 is more optimistic than h_1
- A **more informed algorithm** (a less optimistic heuristic) will generally search through a **smaller** state space

- **Caveat**: cost of the **heuristic computation** $h(s)$ also must be considered
  - **More informed heuristics** typically have **larger computation runtimes**

Good heuristics

- A good heuristic is:
  - **(1) optimistic**,
  - **(2) well informed**
    - We try to find the **least optimistic** of all optimistic heuristics
  - **(3) simple to compute**
    - Ideal heuristic is the **oracle one**, but to have it we need to solve the original problem ☹
    - Heuristic computation cannot be as expensive than solving the original problem!
- What happens if the heuristic is **pessimistic**?
  - We may not get an optimal solution, but perhaps one that is good enough
  - A pessimistic heuristic would additionally reduce the number of nodes/states
  - **Trading off** solution quality for computational efficiency!

Coming up with heuristics

Q: How do we come up with a good heuristic for a problem?

(1) problem relaxation

- True cost of a relaxed (easier) problem
- Example: 8-puzzle
  - h = sum of Manhattan distances of current position to correct position for blocks
  - **Relaxed problem**: we are allowed to move blocks as if other blocks are not there

(2) Combining heuristics

- If we have **optimistic** heuristics $h_1, h_2, \ldots, h_n$ than $h(s) = \mathbf{max}(h_1(s), \ldots, h_n(s))$ is also going to be optimistic and **more informed** than each of the individual h_i

Coming up with heuristics

Q: How do we come up with a good heuristic for a problem?

(3) Using sub-problem costs

- Memorization approach, applicable if across different problems common subproblems occur – quite common in game playing
- **Database** of stored solutions (actual costs) for subproblems – use them as „oracle" heuristics when known subproblems are recognized

(4) Learning heuristics

- Use of machine learning to derive useful heuristics. We design ***features*** that describe each state: $x_1(s)$, $x_2(s)$, ..., $x_m(s)$ and **learn** weights w_1, ..., w_m such that

$$h(s) = w_1 x_1(s) + w_2 x_2(s)$$

Content

- Heuristics
- Greedy Best-First & Hill-Climbing Search
- A* Algorithm
- Heuristics Revisited
- Example: Path Finding on Terrain Map

Problem: Path Finding on Terrain Map

- You're given a **terrain map**: positions (x, y) assigned an altitude $a(x,y)$
- You can directly move between two positions (x_1,y_1) to (x_2,y_2) if
  - $|x_1 - x_2| \leq 1$
  - $|y_1 - y_2| \leq 1$
  - $\Delta a = a(x_2,y_2) - a(x_1,y_1) \leq m$ (you can climb at most m meters for 1 meter of direction change)

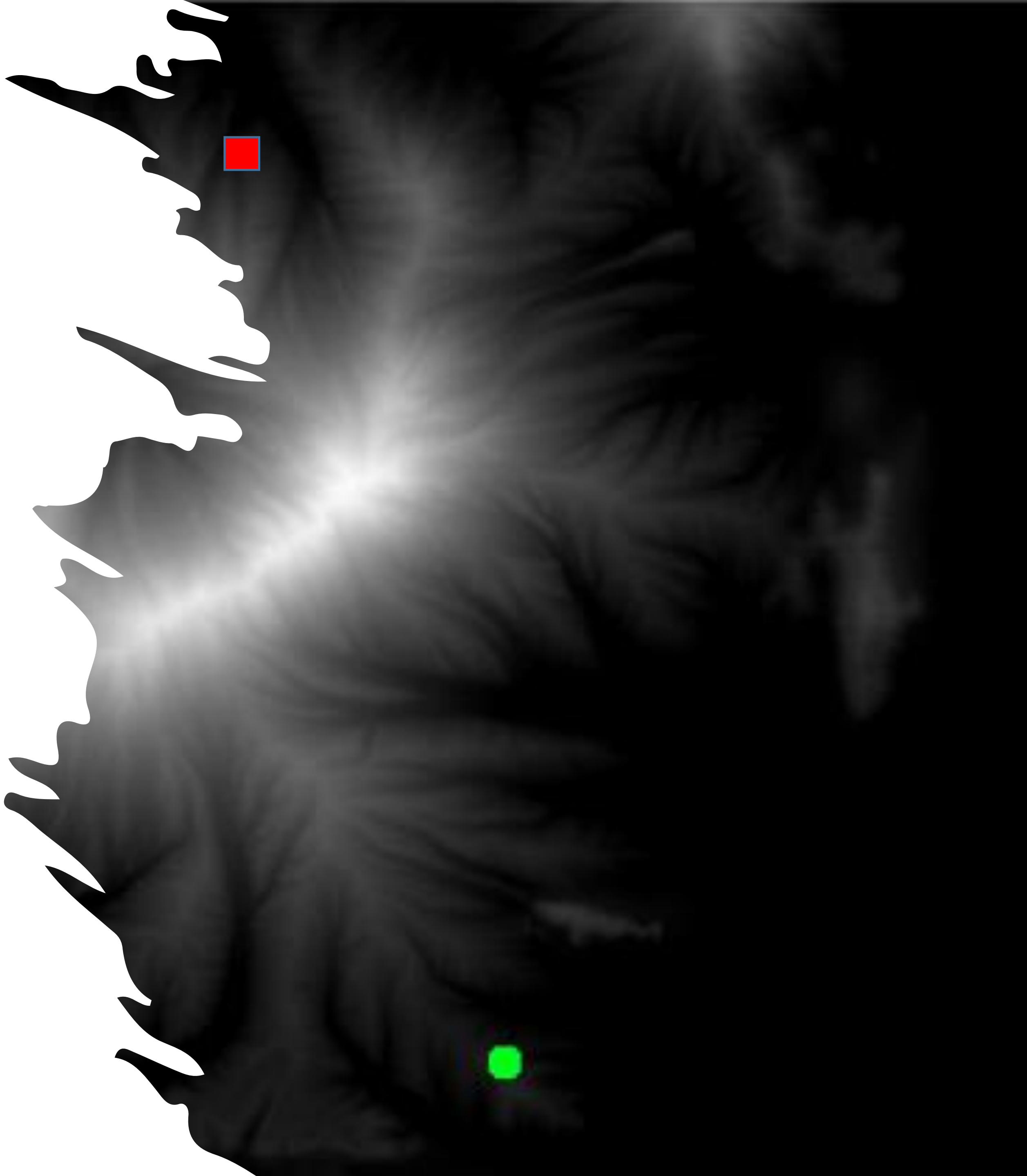
  - This rule defines the **allowed state transitions**

- The **cost** of moving from (x_1,y_1) to (x_2,y_2) is

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} + (½ * \mathrm{sgn}(\Delta a) + 1) \cdot |\Delta a|$$

Problem: Path Finding on Terrain Map

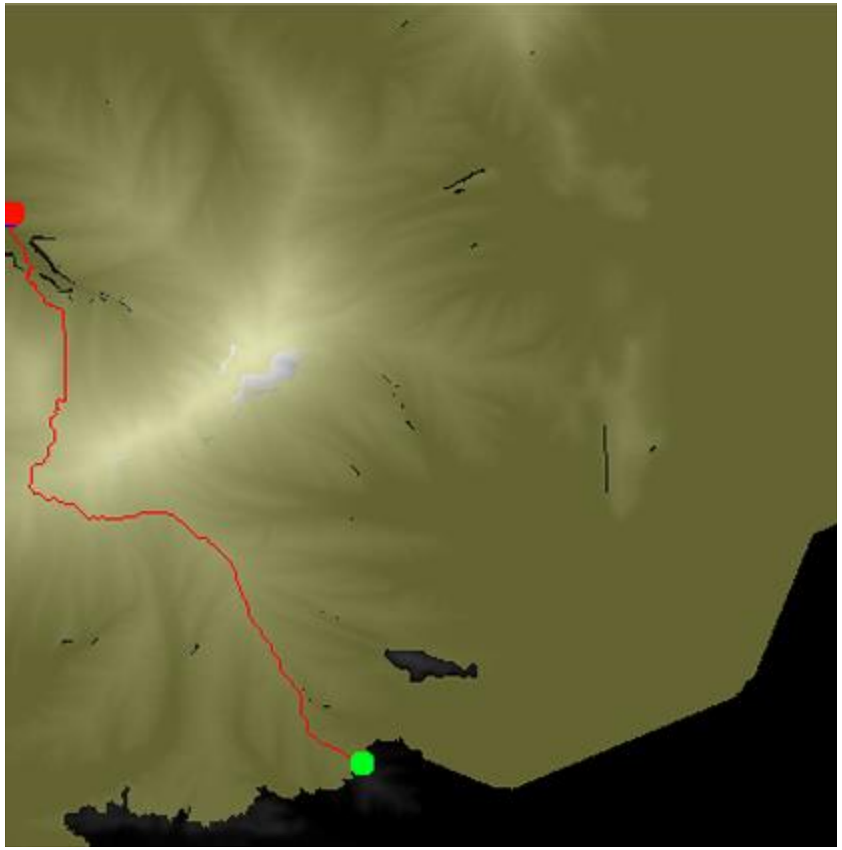
- You read the configuration of the terrain from a file
  - A list of positions (x, y, a)
  - Start and goal positions (x_s, y_s) and (x_g, y_g) given
  - You can plot the terrain in **2D**: **altitude** can be indicated with a color
  - Find optimal path from the red to green dot



Problem: Path Finding on Terrain Map

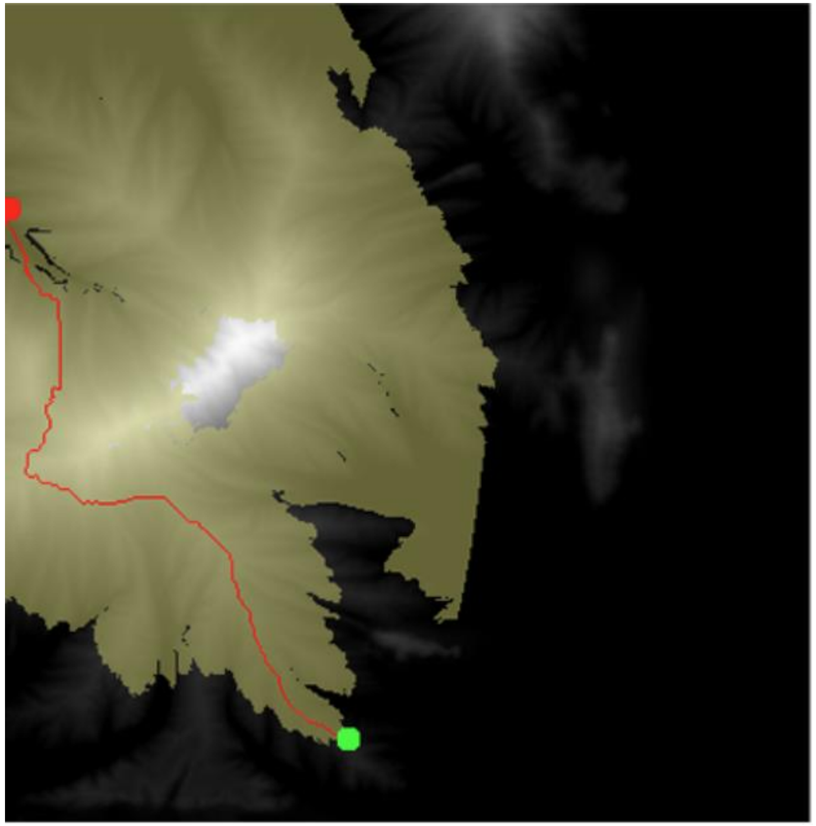
- **Uniform cost search (uninformed**, no heuristics)
  - Yellow/green –visited „states" (positions)

- **States visited**: 140K+



Problem: Path Finding on Terrain Map

- **A* search**
- **Heuristic:** air distance
  - $\sqrt{(x - x_g)^2 + (y - y_g)^2}$
  - Yellow/green – visited „states” (positions)
  - **States visited**: 64K+



Questions?


Awọn ibeere?
ਸਵਾਲ?
Küsimusi?
Turite klausimų?
Pitanja?
Sorusu olan?
Dúvidas?
質問は？
有问题吗？
Fragen?
¿Preguntas?
Questions?
Frågor?
Pytania?
Domande?
Питання?
Vragen?
Ερωτήσεις;
Porandukuéra?
คำถาม?
أسئلة؟