Algorithmen, KI und Data Science 1 (AKIDS 1): **State Space Search**

Herbst- / Wintersemester 2023/24

Prof. Dr. Goran Glavaš

Benedikt Ebing

Fabian David Schmidt

What is state space search (sss)?

Recap State Space Search

- ("Small") Set of start states
- Transitions between states resulting in large number of possible successor states
- Finding a goal state that meets a particular criterion "usually" with minimal cost/maximal gain
- Defined as:

$$sss = (s_0, succ, goal), \text{where}$$
$$initial\ state\ s_o \in \boldsymbol{S},$$
$$succ: \boldsymbol{S} \rightarrow f(\boldsymbol{S}),$$
$$goal: \boldsymbol{S} \rightarrow \{True, False\}$$

What is difference between a state and a (search) node?

State vs. Node

- Node is a data structure that stores the state and additional information like the depth/cost of the node (or the path to the current state)
- $n = (s, d)$ or $n = (s, c)$

What is the difference between uninformed state space search and informed state space search?

Uninformed vs. Informed

- Uninformed:
  - No additional information about the problem that tells whether a state is closer to the goal state than another state
- Informed:
  - An estimate of a state´s distance to the goal state is available

Recap: The General Search Algorithm

General Search Algorithm

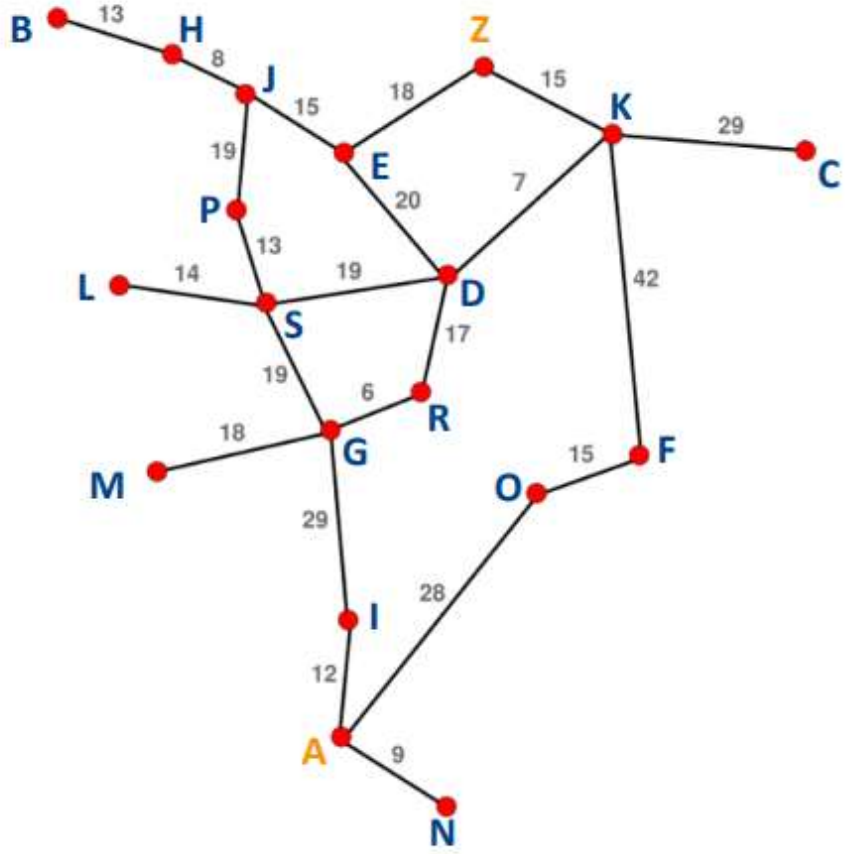
```
search(s0, succ, goal)
  open = [init(s0)]
  while len(open) > 0
    n = take(open)
    if goal(state(n))
      return n
    for m in expand(n, succ)
      insert(m, open)
  return False
```

```
expand(n, succ)
  sstates = succ(state(n))
  nodes = []
  for s in sstates
    nodes = nodes U (s, depth(n) + 1)
  return nodes
```

Two Friends Problem

Exercise 1.1

Suppose two friends live in different cities on a map (e.g., think of a map similar to the one shown in lecture 15, slide 21). On every turn, we must simultaneously move each friend to a neighboring city on the map. The amount of time needed to move from city i to neighbor j is equal to the road distance $d(i, j)$ between the cities, but on each turn the friend that arrives first must wait until the other one arrives before the next turn can begin. We want the two friends to meet as quickly as possible (smallest amount of time).


B
H
Z
J
K
C
E
P
L
S
D
R
G
M
O
F
I
A
N
13
8
15
18
15
29
19
20
7
13
14
19
42
17
19
6
18
15
29
28
12
9

Formulate this problem giving (a) initial state, (b) goal test, (c) successor function, and (d) cost function.

Exercise 1.1 – Solution

- Start state: any pair of nodes $(i, j) \in V \times V$, where V represents the set of cities
- Goal state: $i == j$
- Successors: all pairs (x, y), where x is an adjacent node of i and y is an adjacent node of j ➔ $\{(x, y) | \forall x \in G.Adj[i], \forall y \in G.Adj[j], \}$
- Cost function: $\max\big(d(i, x), d(j, y)\big)$

Exercise 1.2

Recap:
What is a heuristic function in the context of sss? What is an admissible/optimistic heuristic?

Recap admissable/optimistic heuristic?

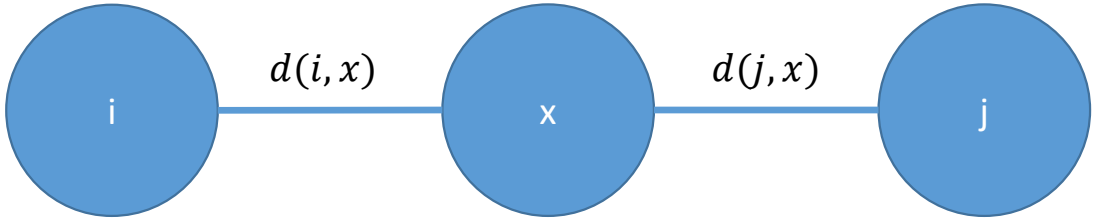
- Heuristic function $h(s)$: Gives an estimate of a state s distance to the goal state
- Admissible/optimistic:
  - For all states s in $\boldsymbol{S}$, $h(s)$ never overestimates the distance of s to the goal state:

$$\forall s \in \boldsymbol{S}, it\ holds\ h(s) \leq h^*(s),$$
$$where\ h^*(s)\ gives\ the\ true\ costs\ from\ s\ to\ the\ goal\ state$$

Let $D(i,j)$ be the straight-line distance between cities i and j. Which of the following heuristic functions are admissible/optimistic?
(a) $D(i,j)$; (b) $2 \times D(i,j)$; (c) $D(i,j)/2$.
Either give a counter example or show that the heuristic is admissible/optimistic.

What is the scenario for the lowest possible costs?

Exercise 1.2 – Solution 1



Best case:

- $d(i,x) = d(j,x) = D(i,x) = D(i,y)$

➔ No one has to wait, and we travel straight line distance (there cannot be a shorter way on a map)

Exercise 1.2 – Solution 2

Given our example:

a) $D(i,j)$ not optimistic/admissible:

- $D(i,j) = 2 \times D(i,x) = 2 \times d(i,x) > \max(d(i,x), d(i,y))$

b) $2 \times D(i,j)$ not optimistic/admissible:

- $2 \times D(i,j) = 4 \times D(i,x) = 4 \times d(i,x) > \max(d(i,x), d(i,y))$

c) $\frac{D(i,j)}{2}$ is admissible:

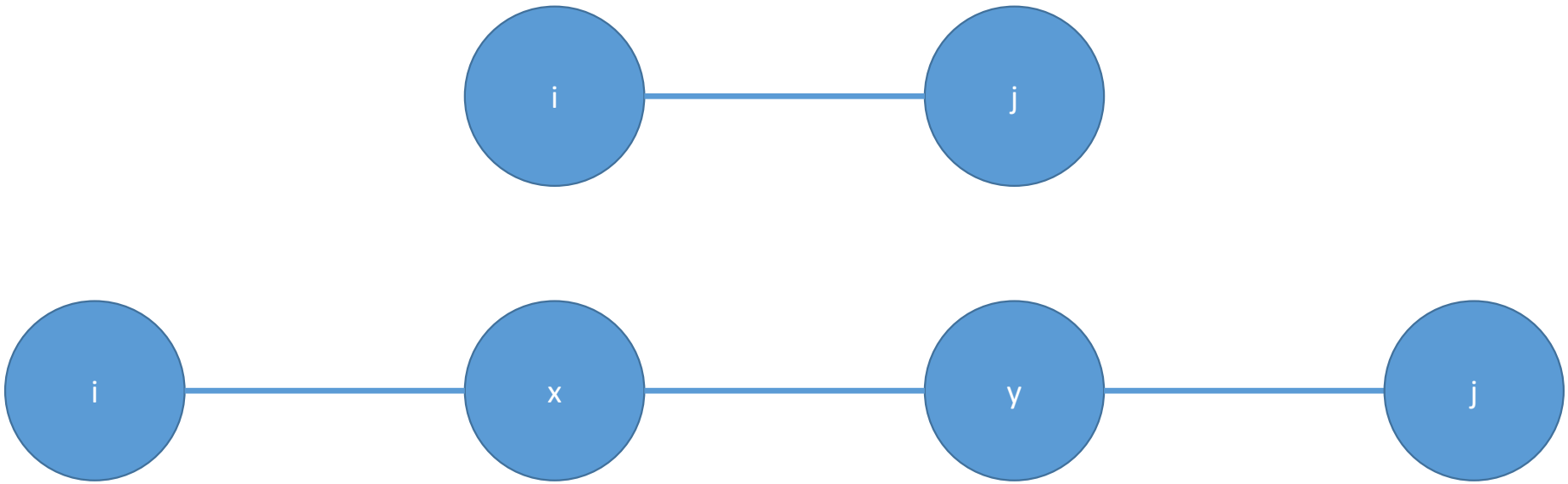
- $\frac{D(i,j)}{2} = \frac{2 \times D(i,x)}{2} = d(i,x)$
- Heuristic always assumes the best-case scenario ➔ Cannot overestimate the true costs

Exercise 1.3

Are there completely connected maps for which no solution exists?

Exercise 1.3 – Solution

Yes:

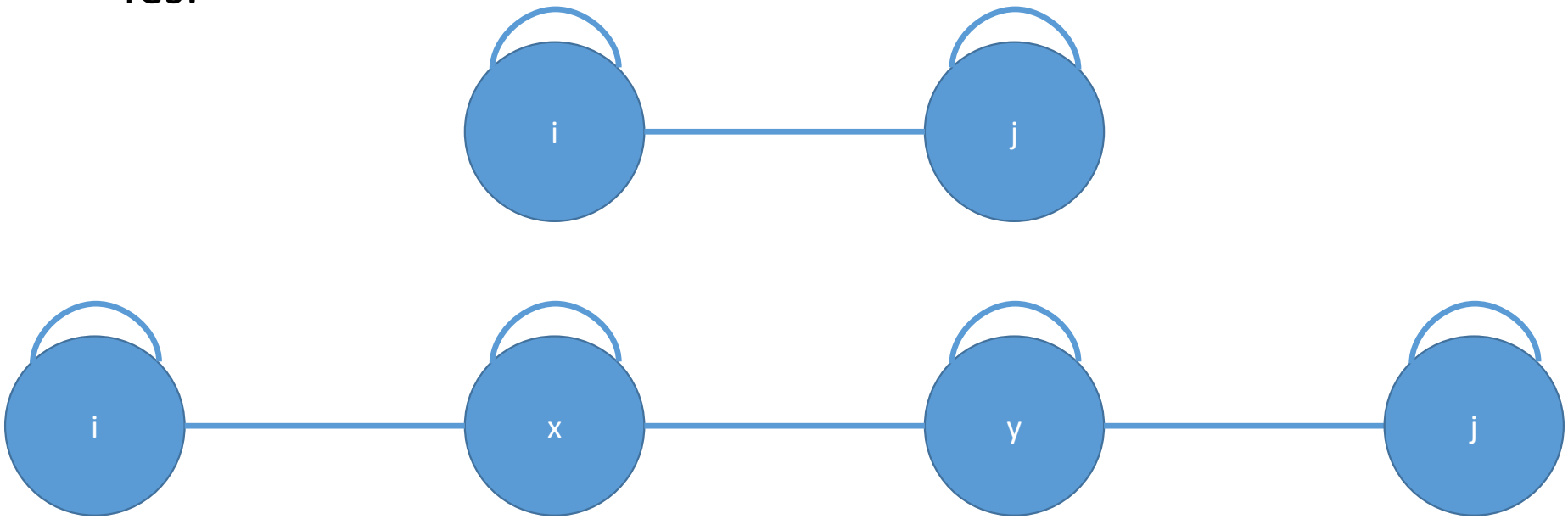


Exercise 1.4

Are there maps in which all solutions require one friend to visit the same city twice?

Exercise 1.4 – Solution

Yes:



Decantation Problem

UNI
WÜ

You are given an 8-liter jar full of water and two empty jars of 5- and 3-liter capacity. You have to get exactly 4 liters of water in one of the jars. You can completely empty a jar into another jar with space or completely fill up a jar from another jar.

Exercise 2.1

Formulate this problem giving (a) initial state, (b) goal test, and (c) successor function. Represent the states by a 3-tuple where each element refers to one of the jars (e.g., (8, 0, 0))

Exercise 2.1 – Solution

- Initial state: The water is in the 8-liter jar, the other jars are empty ➔ (8,0,0)
- Goal test: One jar contains exactly 4 liters (any element of the tuple is equal to 4)
- Successor function:
  - Fill up a jar completely from another jar
  - Empty a jar completely into another jar

Exercise 2.2

Implement the functions $succ(s)$ and $goal(s)$ in the provided .ipynb. The function $succ(s)$ takes a state s as input (e.g., $(8,0,0)$). The function returns a list of possible next states of the same form. One of the possible next states would be $(3, 5, 0)$. The function $goal$ takes a state s as input and returns true if the goal test is successful and false otherwise.

Exercise 2.3

Implement Breadth-First-Search algorithm ($bfs(search\_problem)$ in the provided .ipynb) to search the state space graph for a goal state. Keep track of the expanded states and expand each state only once. Return the goal state, number of pourings required and solution path if a goal state is found and false otherwise.

Exercise 2.3 – Recap BFS

```
bfs-search(s_0, succ, goal)
  open = [init(s_0)]
  while len(open) > 0
    n = dequeue(open)
    if goal(state(n))
      return n
    for m in expand(n, succ)
      enqueue(m, open)
  return False
```

Exercise 2.4

Implement Iterative-Deepening-Search algorithm ($ids(search\_problem)$ and $limited - dfs(search\_problem, k)$ to search the state space graph for a goal state. In limited−dfs, keep track of the expanded states and expand each state only once. Both methods return the goal state, the number of pourings required and the solution path if a goal state is found and false otherwise.

Exercise 2.3 – Recap limited-DFS and IDS

```
limited-dfs-search(s0, succ, goal, k)
  open = [init(s0)]
  while len(open) > 0
    n = pop(open)
    if goal(state(n))
      return n
    if depth(n) < k
      for m in expand(n, succ)
        push(m, open)
  return False
```

```
iter-deep-search(s0, succ, goal)
   for k = 0 to inf
     n = limited-dfs-search
                (s0, succ, goal, k)
     if n ≠ False
       return res
```

Shortest Path Problem

Exercise 3.1

Apply the A$*$ algorithm (see lecture 15, slides 21-24) to the graph below to find the shortest path and the total cost from s to x. You are given the following heuristics:

$$h(s) = 9, h(t) = 1, h(y) = 4, h(z) = 13, h(x) = 0$$

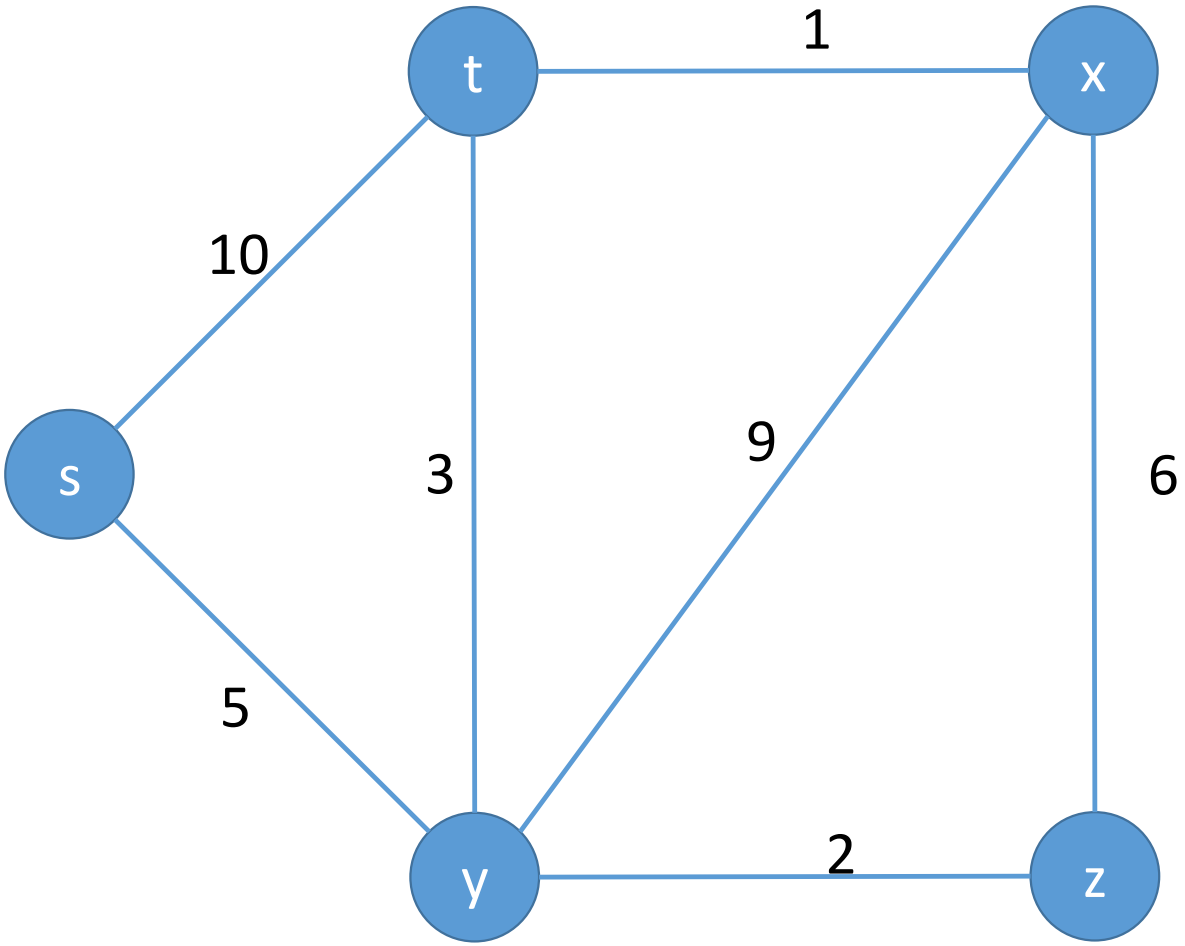
Recap: How does A^* work?

Exercise 3.1 – Recap A^*

```
astar-search(s_0, succ, goal, h)
  visited = {}
  open = [(s_0, h(s_0))]
  visited[s_0] = 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)
```

Exercise 3.1 - Solution



Exercise 3.1 - Solution

- Init:
  - $open = [(s, 0 + 9)]$
  - $visited = \{s : 0\}$
- 1. Iteration:
  - extract-min: $(s, 0 + 9)$
  - $open = [(y, 5 + 4), (t, 10 + 1)]$
  - $visited = \{s : 0, t : 10, y : 5\}$
- 2. Iteration :
  - extract–min: $(y, 5 + 4)$
  - $open = [(t, 8 + 1), (x, 14 + 0), (z, 7 + 13)$
  - $visited = \{s : 0, t : 8, y : 5, x : 14, z : 7\}$

```
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)
```

Exercise 3.1 - Solution

- 3. Iteration:
  - extract–min: $(t, 8 + 1)$
  - $open = [(x, 9 + 0), (z, 7 + 13)]$
  - $visited = \{s: 0, t: 8, y: 5, x: 9, z: 7\}$
- 4. Iteration:
  - $goal(x) == True$

Total cost: 9
shortest-path: s → y → t → x

```
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)
```

Exercise 3.2

Apply the GreedyBestFirstSearch algorithm to the graph below to find the shortest path and the total cost from s to x. You are given the following heuristics: $h(s) = 9, h(t) = 1, h(y) = 4, h(z) = 13, h(x) = 0$

Recap: How does GreedyBestFirstSearch work?

Exercise 3.2 – Recap GreedyBestFirstSearch

```
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
```

Exercise 3.2 – Solution

- Init:
  - $open = [(s, 9)]$
- 1. Iteration:
  - extract-min: $(s, 9)$
  - $open = [(t, 1), (y, 4)]$
- 2. Iteration:
  - extract−min: $(t, 1)$
  - $open = [(x, 0), (y, 4), (s, 9)]$

```
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
```

Exercise 3.2 – Solution

- 3. Iteration:
  - $goal(x) == True$

Total cost: 11
“shortest-path”: s → t → x

```
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
```

Exercise 3.3

Implement A^* algorithm (function $a\_star(search\_problem)$ in the provided .ipynb) for shortest path problems (similar to the previous exercise). Return the goal state and the total costs if a goal state is found and false otherwise.