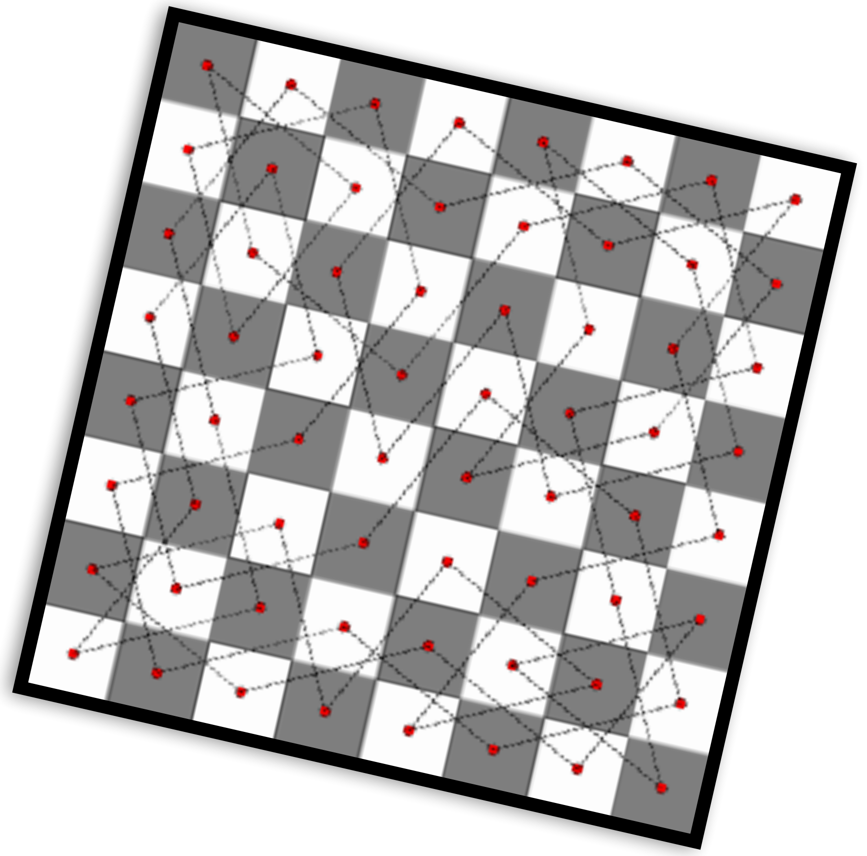
Advanced Dynamic Programming using informed search

LINFO2266

Pierre Schaus



Shortest Path Algorithm: Dijkstra

- Dijkstra solves the single source-shortest path problem.
- Starting from vertex v, it finds all the shortest path toward all the other vertices of a graph.

Dijkstra Algorithm (Dijkstra 1959)

Data: A graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}^+$, and a source vertex $s \in V$.

Result: The shortest distance $d[v]$ from s to every vertex $v \in V$.

for *each vertex* $v \in V$ **do**

  $d[v] \leftarrow \infty$; `// Shortest path distance from` s `to` v

  $\pi[v] \leftarrow \text{NIL}$; `// Predecessor in the shortest path`

$d[s] \leftarrow 0$;

$C \leftarrow \emptyset$; `// Closed Set`

$O \leftarrow \emptyset$; `// Open-Set: Min-priority queue`

$\text{Insert}(Q, s, d[s])$;

while $Q \neq \emptyset$ **do**

  $u \leftarrow \text{ExtractMin}(O)$;

  $C \leftarrow C \cup \{u\}$;

  for *each vertex* $v \in Adj[u]$ **do**

    if $d[v] > d[u] + w(u, v)$ **then**

      $d[v] \leftarrow d[u] + w(u, v)$;

      $\pi[v] \leftarrow u$;

      $\text{InsertOrUpdateKey}(O, v, d[v])$;

$O(E \log V)$ using binary heap for the priority queue

$O(E \log V)$ using binary heap for the priority queue

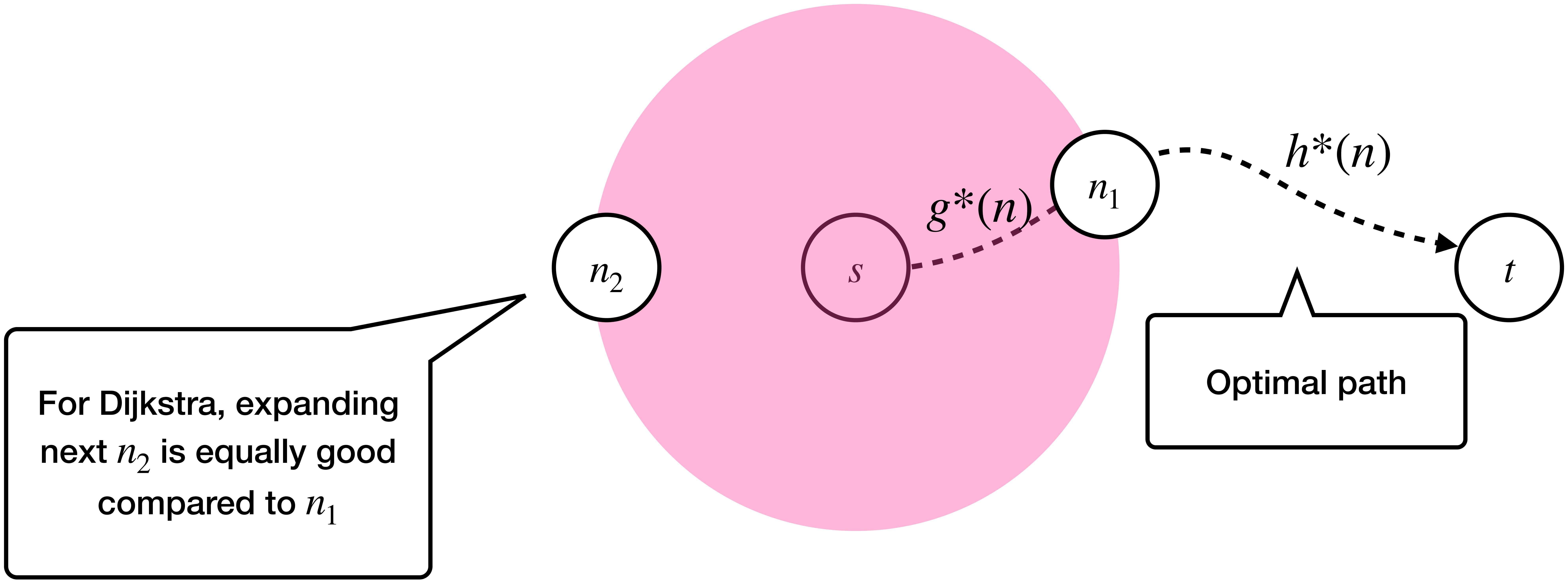
Dijkstra illustration

Subh83 wikimedia CC BY 3.0

- The empty circles represent the nodes in the *open set* O, i.e., those that remain to be explored, and
- The filled circles are in the closed set C.

Dijkstra is great but

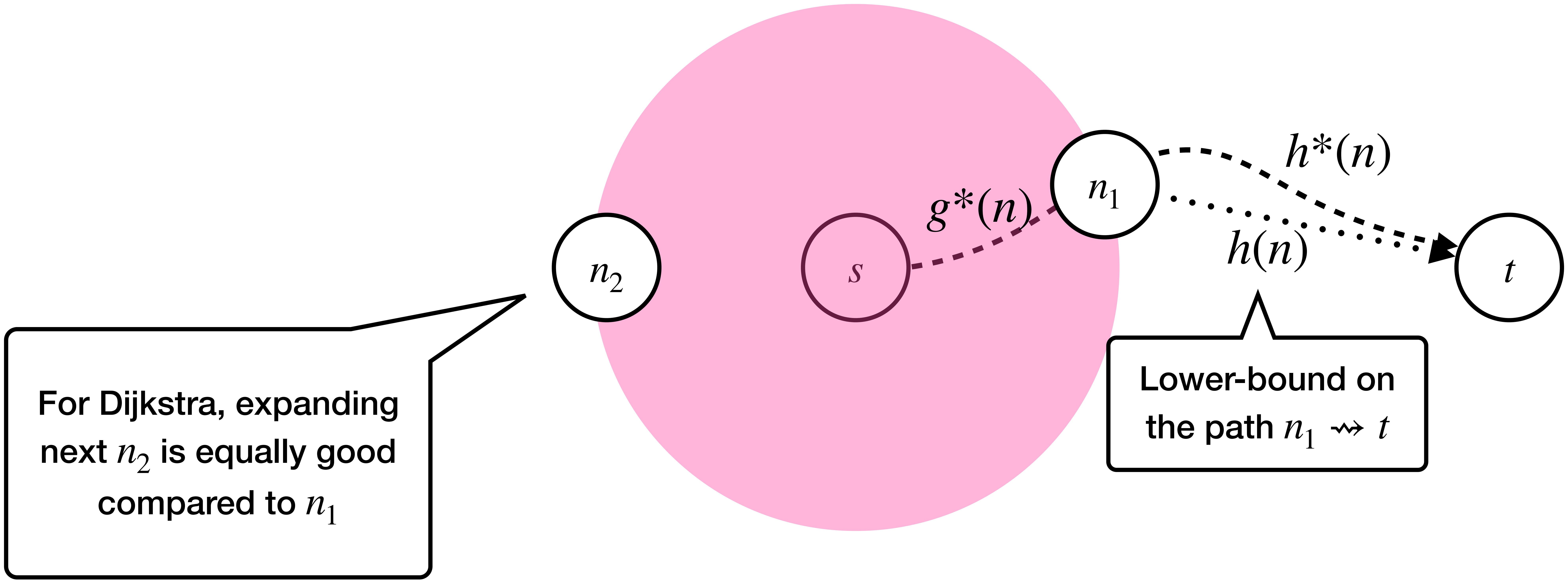
- It doesn't look to reach the target, for Dijkstra the target is a regular node of the graph, until you reach it (in this case you can stop Dijkstra earlier).



- Dijkstra will thus visit a lot of nodes! We can do better

💡 Use heuristics

- For Euclidian shortest path, you can for instance use the Euclidian distance (straight line distance) to estimate the distance left.



- Instead of expanding the node based on $g(n)$ we will expand it based on $g(n) + h(n)$

A* Algorithm (Hart, Nilsson, Raphael 1968)

Data: A graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}^+$, a source s, a goal t, and a heuristic $h(v)$.
Result: The shortest distance $g[t]$ from s to t.

```
for each vertex v ∈ V do
    g[v] ← ∞ ;                          // Actual cost from start s to v
    f[v] ← ∞ ;                          // Estimated total cost (g + h)
    π[v] ← NIL ;                        // Predecessor in the shortest path
g[s] ← 0;
f[s] ← h(s);
C ← ∅ ;                                 // Closed Set
O ← ∅ ;         // Open Set (Min-priority queue ordered by f[v])
Insert(O, s, f[s]) ;
while O ≠ ∅ do
    u ← ExtractMin(O);
    C ← C ∪ {u};
    if u = t then
        return g[t], π ;                // Goal reached
    for each vertex v ∈ Adj[u] do
        if g[u] + w(u, v) < g[v] then
            g[v] ← g[u] + w(u, v);
            f[v] ← g[v] + h(v) ;        // Score = actual + heuristic
            π[v] ← u;
            InsertOrUpdateKey(O, v, f[v]) ;
return No path to goal ;
```

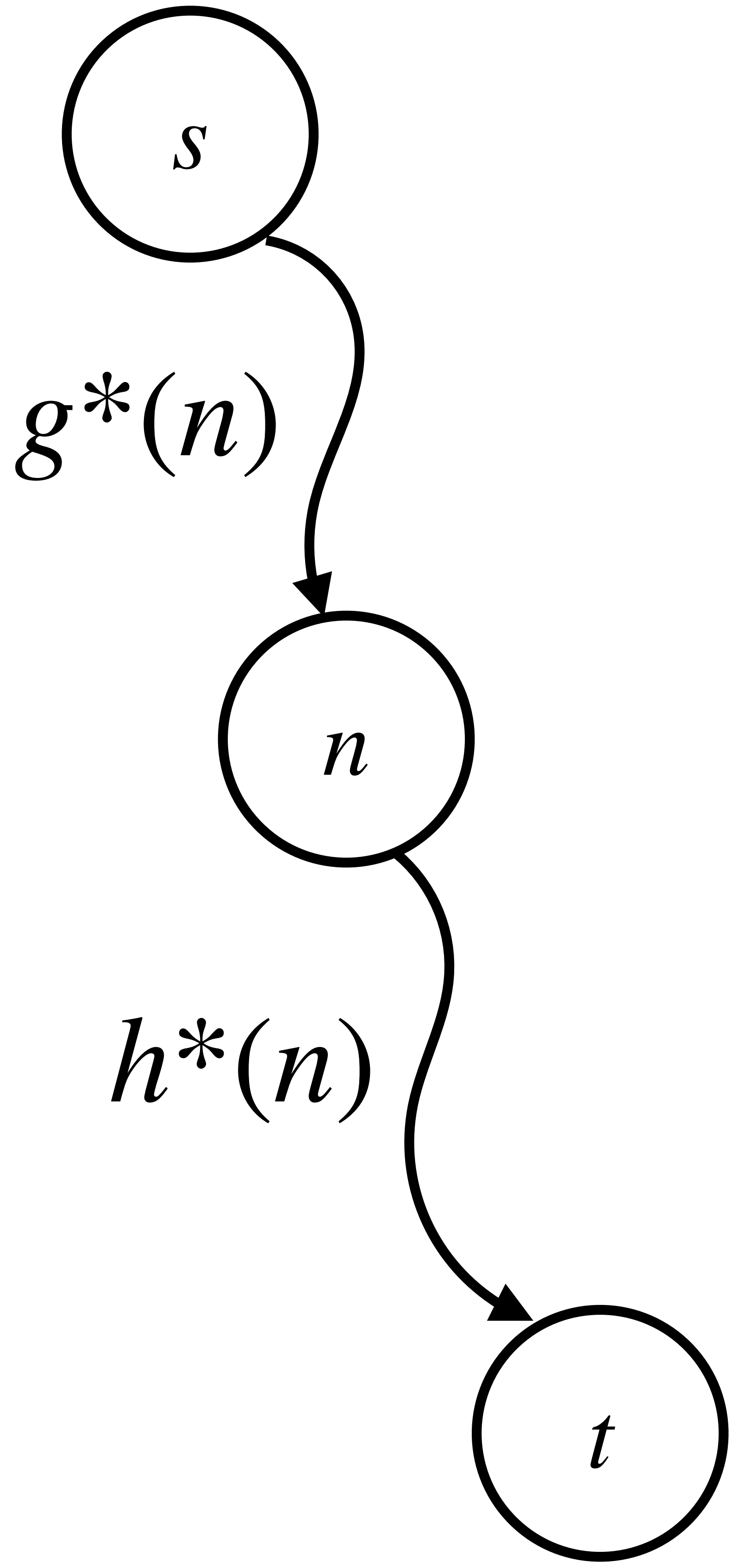
Properties Heuristics h and impact on A*

- A **search algorithm** is **admissible** if it is guaranteed to find an optimal solution
- A **heuristic h** is **admissible** if it never over-estimates the cost to the solution

A* using an admissible heuristic h is admissible

A* with admissible h is admissible (proof by contradiction)

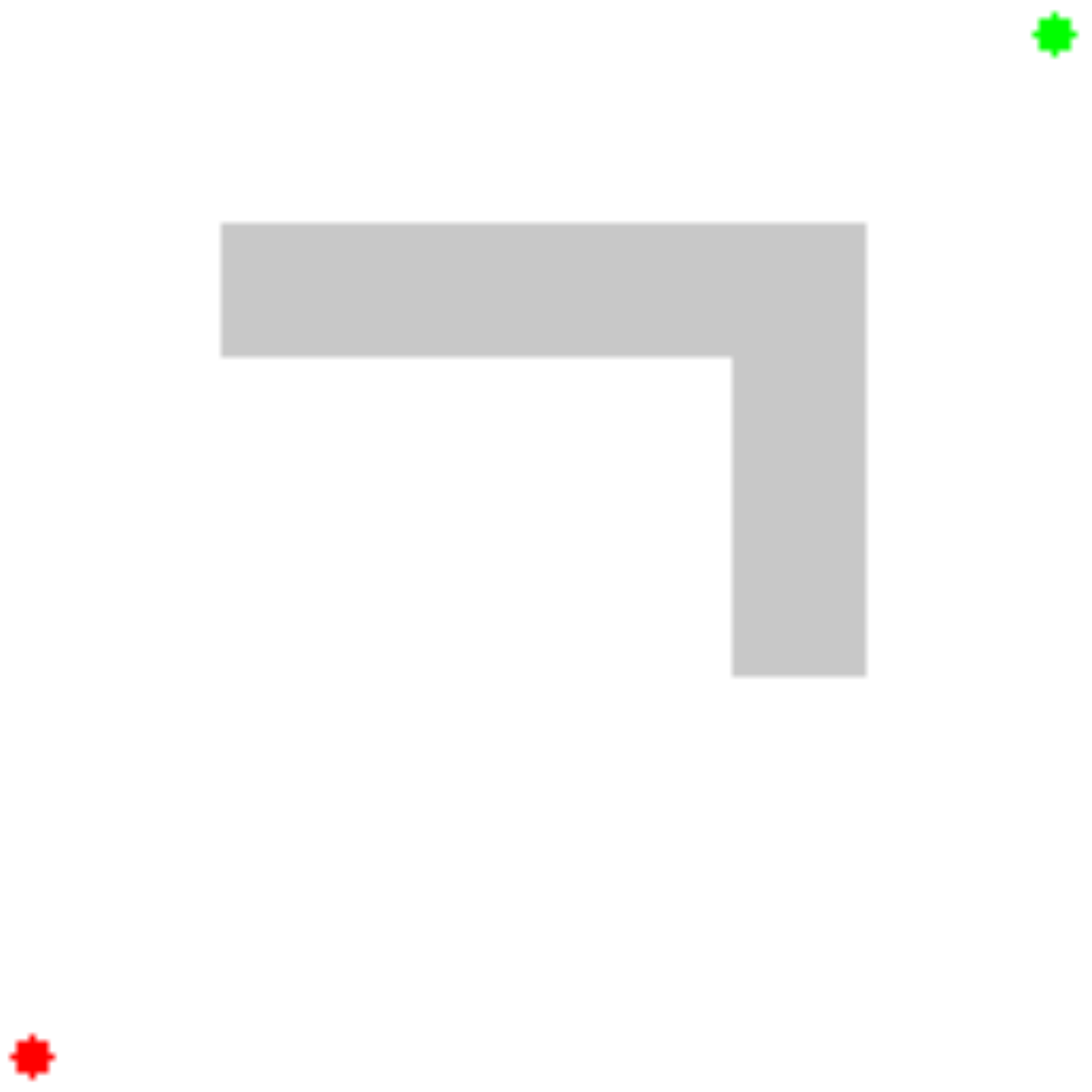
- C^* = optimal cost from s to t, let's assume A* returned a suboptimal one with cost $C > C^*$
- Then there must be some node n on the optimal path that was left unexpanded (i.e. not closed) otherwise we would have returned that solution. Let's denote $g^*(n)$ the cost of the optimal path from s to n, and $h^*(n)$ the cost of the optimal path from n to t. We have:
- $f(n) > C^*$ (otherwise n would have been expanded)
- $f(n) = g(n) + h(n)$ (by definition)
- $f(n) = g^*(n) + h(n)$ (because n is on an optimal path)
- $f(n) \leq g^*(n) + h^*(n)$ (because of admissibility of $h(n) \leq h^*(n)$)
- $f(n) \leq C^*$ (by definition $C^* = g^*(n) + h^*(n)$) contradiction!



Consistent heuristics

- A heuristic is consistent if for every node n and every successor node n' of n, we have that $h(n) \leq w(n, n') + h(n')$
- With consistent heuristics, the first time we reach a state, it will be on an optimal path, so we never have to re-add it to the queue or to change it's priority.
- In practice, it is quite difficult to design a heuristic that is admissible but not consistent.

A* illustration



[Subh83](#) wikimedia CC BY 3.0

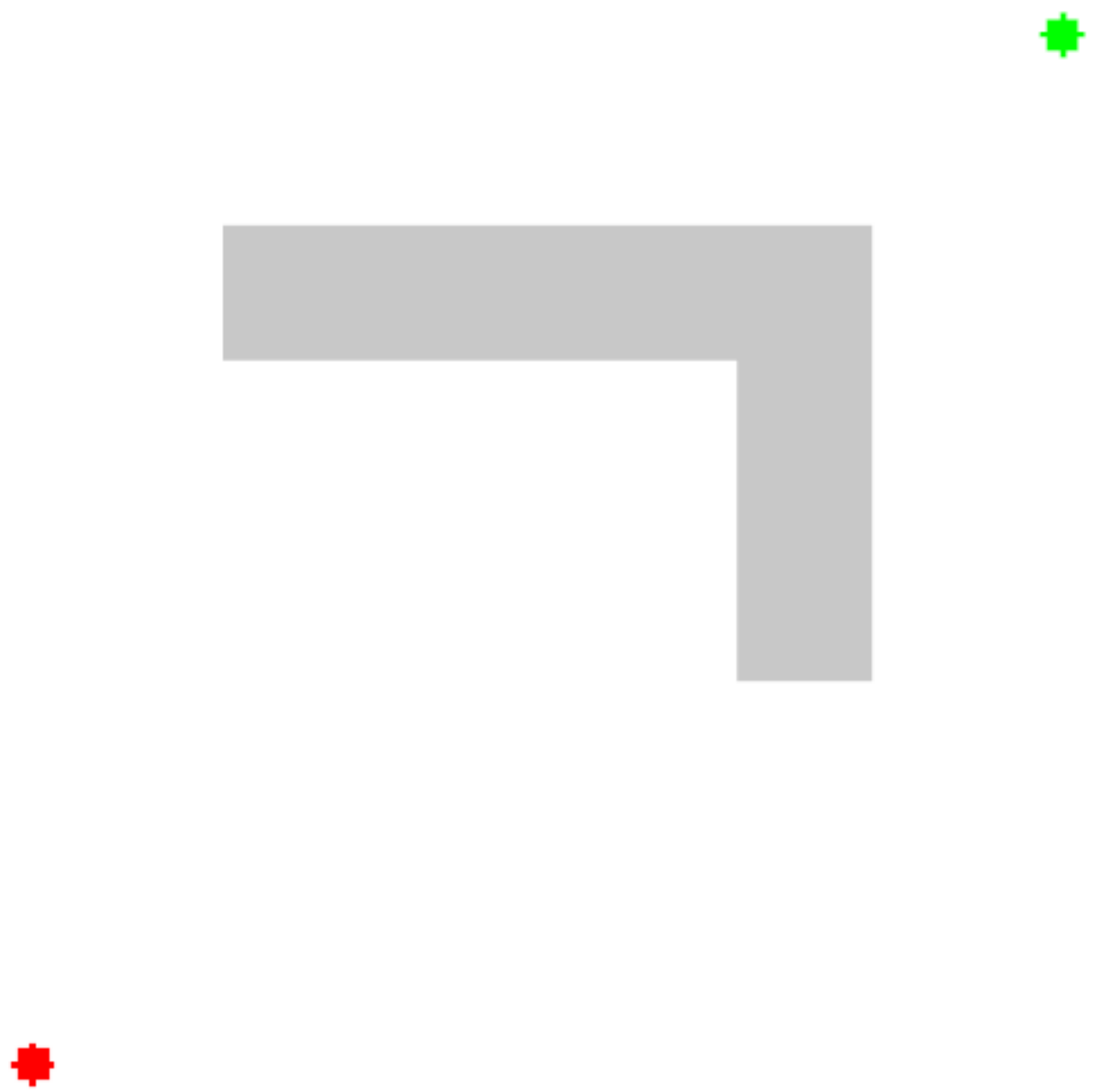
- The empty circles represent the nodes in the *open set* O, i.e., those that remain to be explored, and
- The filled circles are in the closed set C.
- Color on each closed node indicates the distance from the goal: the greener, the closer.

Weighted A* Algorithm

Data: A graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}^+$, a source s, a goal t, a heuristic $h(v)$, a weight parameter $w_h \geq 1$ (controls heuristic influence).
Result: The shortest distance $g[t]$ from s to t.

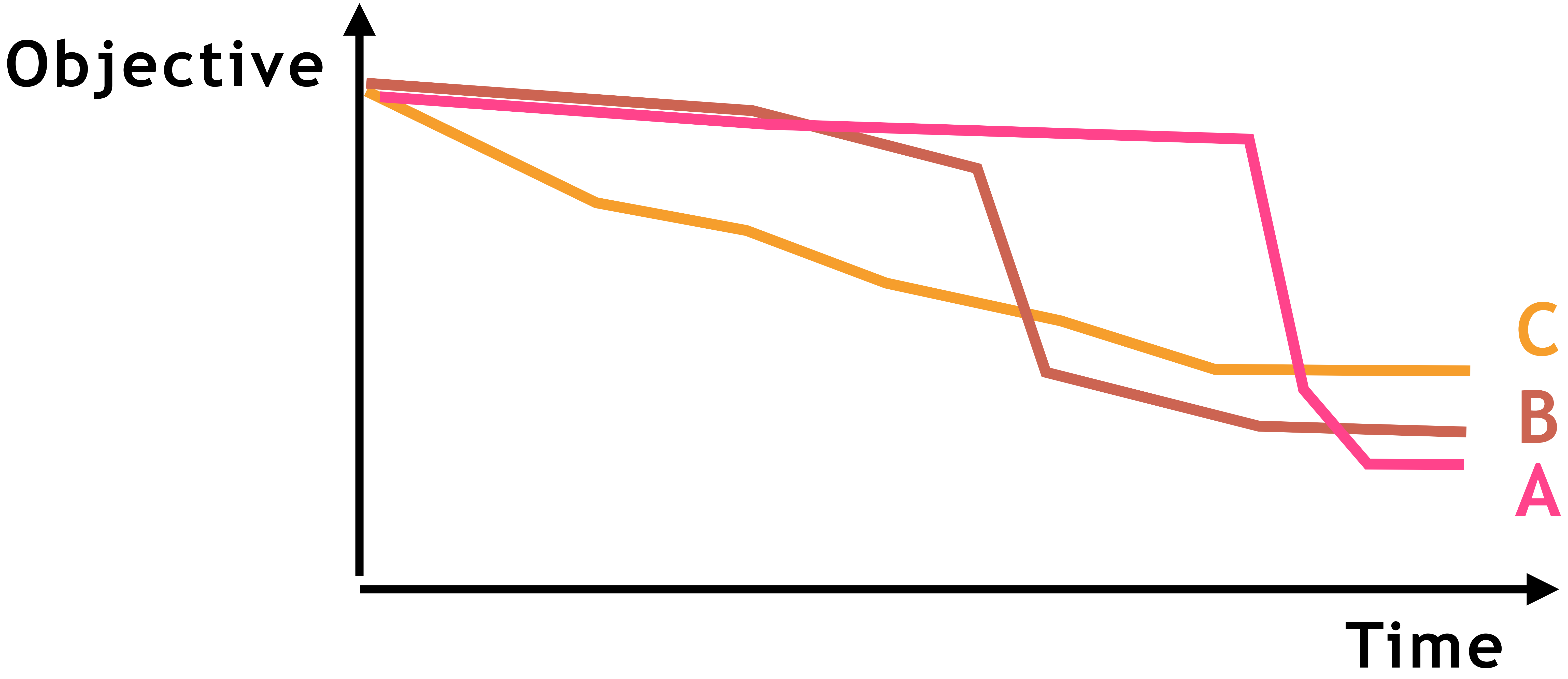
```
for each vertex v ∈ V do
    g[v] ← ∞ ;                          // Actual cost from start s to v
    f[v] ← ∞ ;                          // Estimated total cost (g + w_h · h)
    π[v] ← NIL ;                        // Predecessor in the shortest path
g[s] ← 0;
f[s] ← w_h · h(s);
C ← ∅ ;                                 // Closed Set
O ← ∅ ;          // Open Set (Min-priority queue ordered by f[v])
Insert(O, s, f[s]) ;
while O ≠ ∅ do
    u ← ExtractMin(O);
    C ← C ∪ {u};
    if u = t then
        return g[t] ;                   // Goal reached
    for each vertex v ∈ Adj[u] do
        if v ∉ C then
            if g[u] + w(u, v) < g[v] then
                g[v] ← g[u] + w(u, v);
                f[v] ← g[v] + w_h · h(v) ;   // Weighted score
                π[v] ← u;
                InsertOrUpdateKey(O, v, f[v]) ;
return g[t], π;
```

Weighted A* illustration: w = 5



Reminder: Anytime Algorithm

- An algorithm has good anytime behavior when it is able to find high-quality solutions, even when when the search is stopped before completion.



What algorithm do you prefer ?

Berthold, T. (2013). Measuring the impact of primal heuristics. *Operations Research Letters*, *41*(6), 611-614.

Is it possible to improve the anytime behavior of A* algorithms ?

- First idea: use weighted A*, and decrease gradually the weight w util reaching 1 (w=1 is standard A*)

- Better idea (Hansen & Zoo 2007), bring three changes to (weighted) A*:

1. Use an non-admissible evaluation function $f'(n) = g(n) + h'(n)$, where $h'(n)$ is not admissible (for instance $h'(n) = w \cdot h(n)$ with $w \geq 1$).

2. Continue the search after a solution (reaching node t) is found, but use the cost $g(t)$ as an upper-bound (branch and bound like).

3. Use an admissible $f(n) = g(n) + h(n)$ as lower-bound, and prune it when it is larger than the current upper-bound.

Anytime Weighted A*

Data: A graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}^+$, a source s, a goal t, a heuristic $h(v)$, a weight parameter $w_h \geq 1$.

Result: The cost of the best solution found, g_{inc}, and the incumbent path π_{inc}.

```
for each vertex v ∈ V do
    g[v] ← ∞ ;                          // Cost of best known path s ⇝ v
    f'[v] ← ∞ ;                         // Weighted cost: g + w_h · h
    π[v] ← NIL;                         // Best known predecessor
g[s] ← 0;
f'[s] ← w_h · h(s);
g_inc ← ∞ ;                             // Incumbent cost
C ← ∅ ;       // Closed Set (Contains nodes that were expanded)
O ← ∅ ;       // Open Set (Min-priority queue ordered by f'[v])
Insert(O, s, f'[s]);
while O ≠ ∅ and not user interrupted do
    u ← ExtractMin(O);
    C ← C ∪ {u};
    if g[u] + h(u) > g_inc then
        continue ;                      // Pruning by incumbent cost
    for each vertex v ∈ Adj[u] do
        if g[u] + w(u, v) + h(v) ≥ g_inc then
            continue ;                  // Pruning by incumbent cost
        else if g[u] + w(u, v) < g[v] then
            π[v] ← u ;
            g[v] ← g[u] + w(u, v) ;
            f'[v] ← g[v] + w_h · h(v) ;
            if v = t then
                g_inc ← g[t] ;          // Incumbent cost update
            else
                InsertOrUpdateKey(O, v, f'[v]);
                C ← C \ {v} ;
return g_inc, π ;                       // O = ∅ (optimal) or was interrupted
```

Solving Dynamic Programs with A* like algorithms

- Knapsack Problem:

The set of items: I

Variables $x_i \in \{0,1\}$, 1 iff item i selected

Objective: Maximize $\sum_{i \in I} v_i \cdot x_i$

Maximize value of selected items

Constraints: $\sum_{i \in I} w_i \cdot x_i \leq C$

Under capacity constraint

DP formulation of the Knapsack Problem

- Assume $I = \{1 \ldots n\}$
- Notation $O(k, i)$ = optimal objective with capacity k and items $\{1 \ldots j\}$

$$O(k, i) = \begin{cases} 0 & \text{if } i = 0 \\ \max(O(k, j-1), v_j + O(k - w_j, j-1)) & \text{if } w_j \leq k \\ O(k, j-1) & \text{otherwise} \end{cases}$$

The optimal solution is $O(C, n)$

Dynamic Programming = recursive method with caching of the values $O(k, i)$

Knapsack as a minimization problem

- Knapsack Problem:

The set of items: I

Variables $x_i \in \{0,1\}$, 1 iff item i selected

Objective: Minimize $\sum_{i \in I} - v_i \cdot x_i$

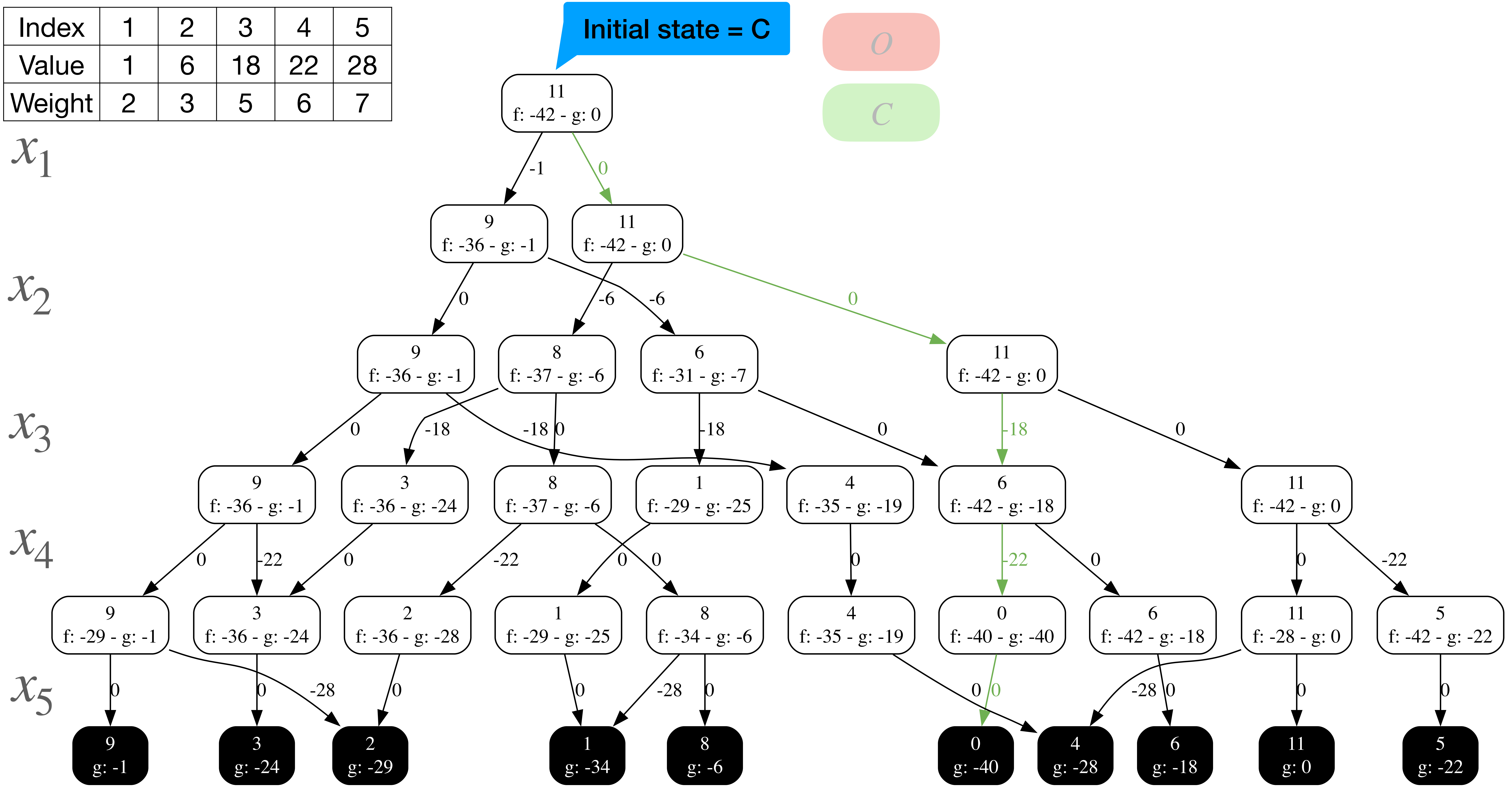
Minimize negative value of selected items

Constraints: $\sum_{i \in I} w_i \cdot x_i \leq C$

Under capacity constraint

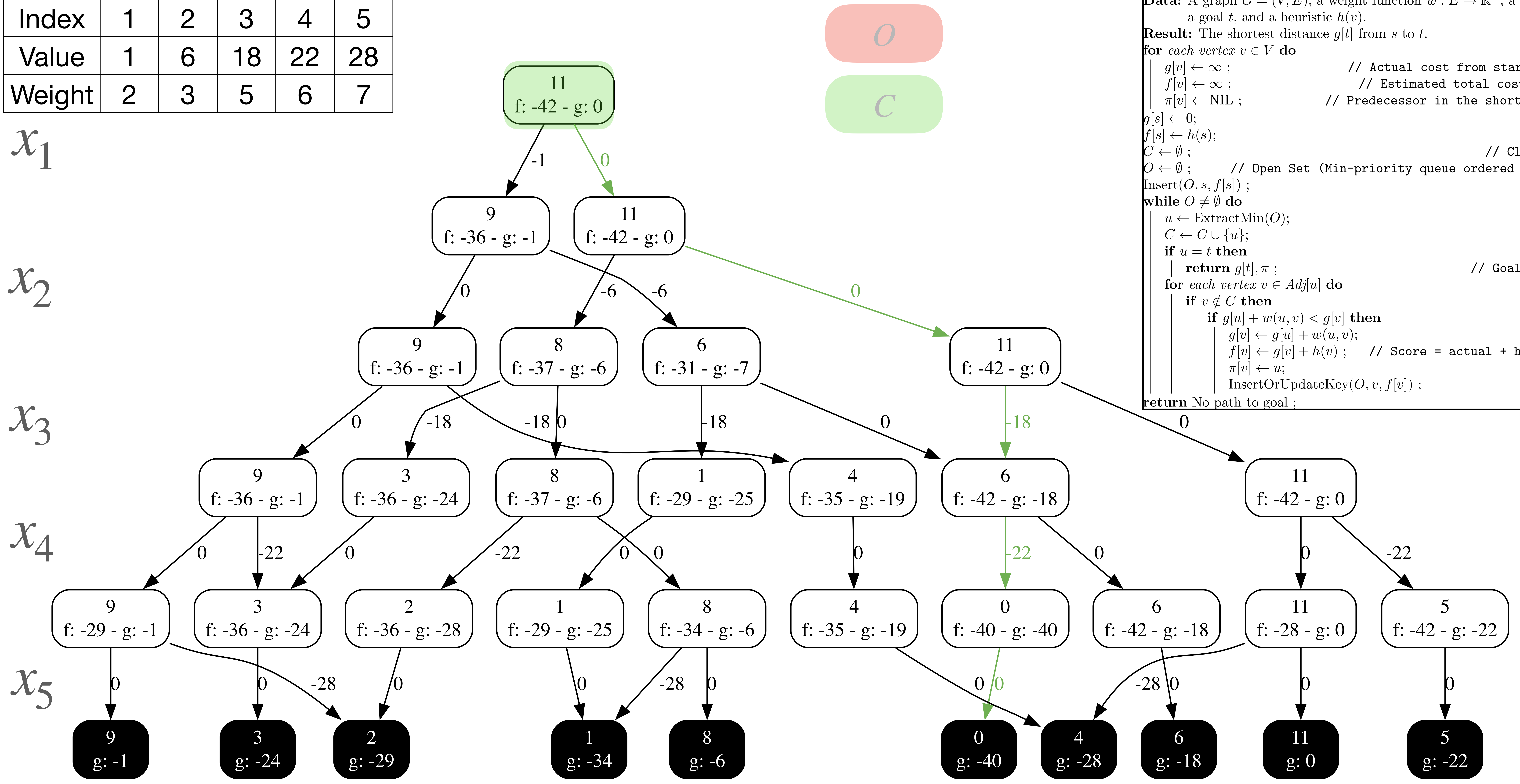
Dynamic Programming Resolution = Shortest Path on DAG

Index	1	2	3	4	5
Value	1	6	18	22	28
Weight	2	3	5	6	7



Resolution with A*, h = linear relaxation

Index	1	2	3	4	5
Value	1	6	18	22	28
Weight	2	3	5	6	7

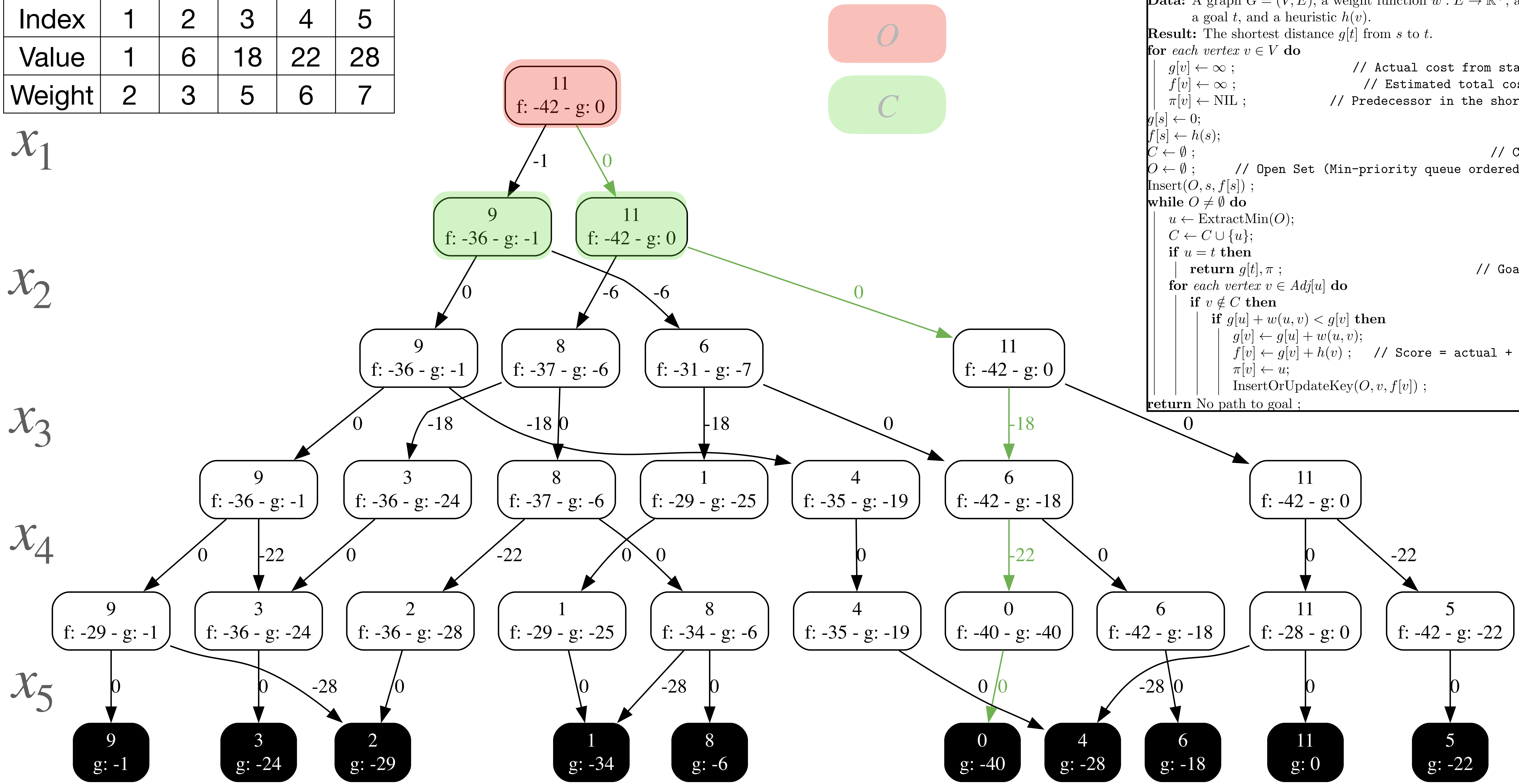


Data: A graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}^+$, a source s, a goal t, and a heuristic $h(v)$.
Result: The shortest distance $g[t]$ from s to t.

for *each vertex* $v \in V$ **do**
  $g[v] \leftarrow \infty$; `// Actual cost from start s to v`
  $f[v] \leftarrow \infty$; `// Estimated total cost (g + h)`
  $\pi[v] \leftarrow$ NIL ; `// Predecessor in the shortest path`
$g[s] \leftarrow 0$;
$f[s] \leftarrow h(s)$;
$C \leftarrow \emptyset$; `// Closed Set`
$O \leftarrow \emptyset$; `// Open Set (Min-priority queue ordered by f[v])`
Insert$(O, s, f[s])$;
while $O \neq \emptyset$ **do**
  $u \leftarrow$ ExtractMin(O);
  $C \leftarrow C \cup \{u\}$;
  if $u = t$ **then**
    return $g[t], \pi$; `// Goal reached`
  for *each vertex* $v \in Adj[u]$ **do**
    if $v \notin C$ **then**
      if $g[u] + w(u, v) < g[v]$ **then**
        $g[v] \leftarrow g[u] + w(u, v)$;
        $f[v] \leftarrow g[v] + h(v)$; `// Score = actual + heuristic`
        $\pi[v] \leftarrow u$;
        InsertOrUpdateKey$(O, v, f[v])$;
return No path to goal ;

Resolution with A*, h = linear relaxation

Index	1	2	3	4	5
Value	1	6	18	22	28
Weight	2	3	5	6	7



Data: A graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}^+$, a source s, a goal t, and a heuristic $h(v)$.
Result: The shortest distance $g[t]$ from s to t.

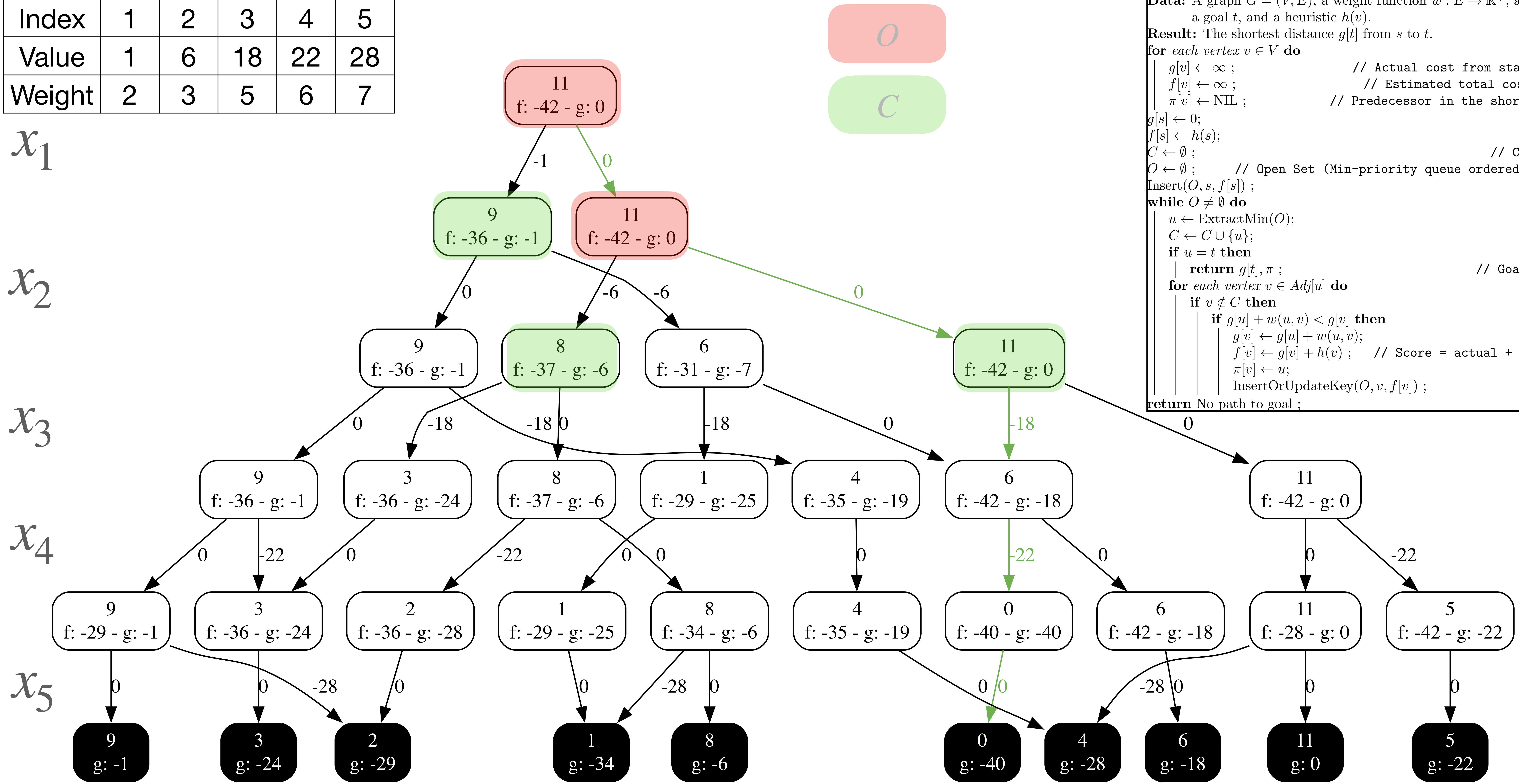
for *each vertex* $v \in V$ **do**
- $g[v] \leftarrow \infty$; // Actual cost from start s to v
- $f[v] \leftarrow \infty$; // Estimated total cost $(g + h)$
- $\pi[v] \leftarrow$ NIL ; // Predecessor in the shortest path

$g[s] \leftarrow 0$;
$f[s] \leftarrow h(s)$;
$C \leftarrow \emptyset$; // Closed Set
$O \leftarrow \emptyset$; // Open Set (Min-priority queue ordered by f[v])
Insert$(O, s, f[s])$;
while $O \neq \emptyset$ **do**
- $u \leftarrow$ ExtractMin(O);
- $C \leftarrow C \cup \{u\}$;
- **if** $u = t$ **then**
  - **return** $g[t], \pi$; // Goal reached
- **for** *each vertex* $v \in Adj[u]$ **do**
  - **if** $v \notin C$ **then**
    - **if** $g[u] + w(u, v) < g[v]$ **then**
      - $g[v] \leftarrow g[u] + w(u, v)$;
      - $f[v] \leftarrow g[v] + h(v)$; // Score = actual + heuristic
      - $\pi[v] \leftarrow u$;
      - InsertOrUpdateKey$(O, v, f[v])$;

return No path to goal ;

Resolution with A*, h = linear relaxation

Index	1	2	3	4	5
Value	1	6	18	22	28
Weight	2	3	5	6	7



Data: A graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}^+$, a source s, a goal t, and a heuristic $h(v)$.
Result: The shortest distance $g[t]$ from s to t.

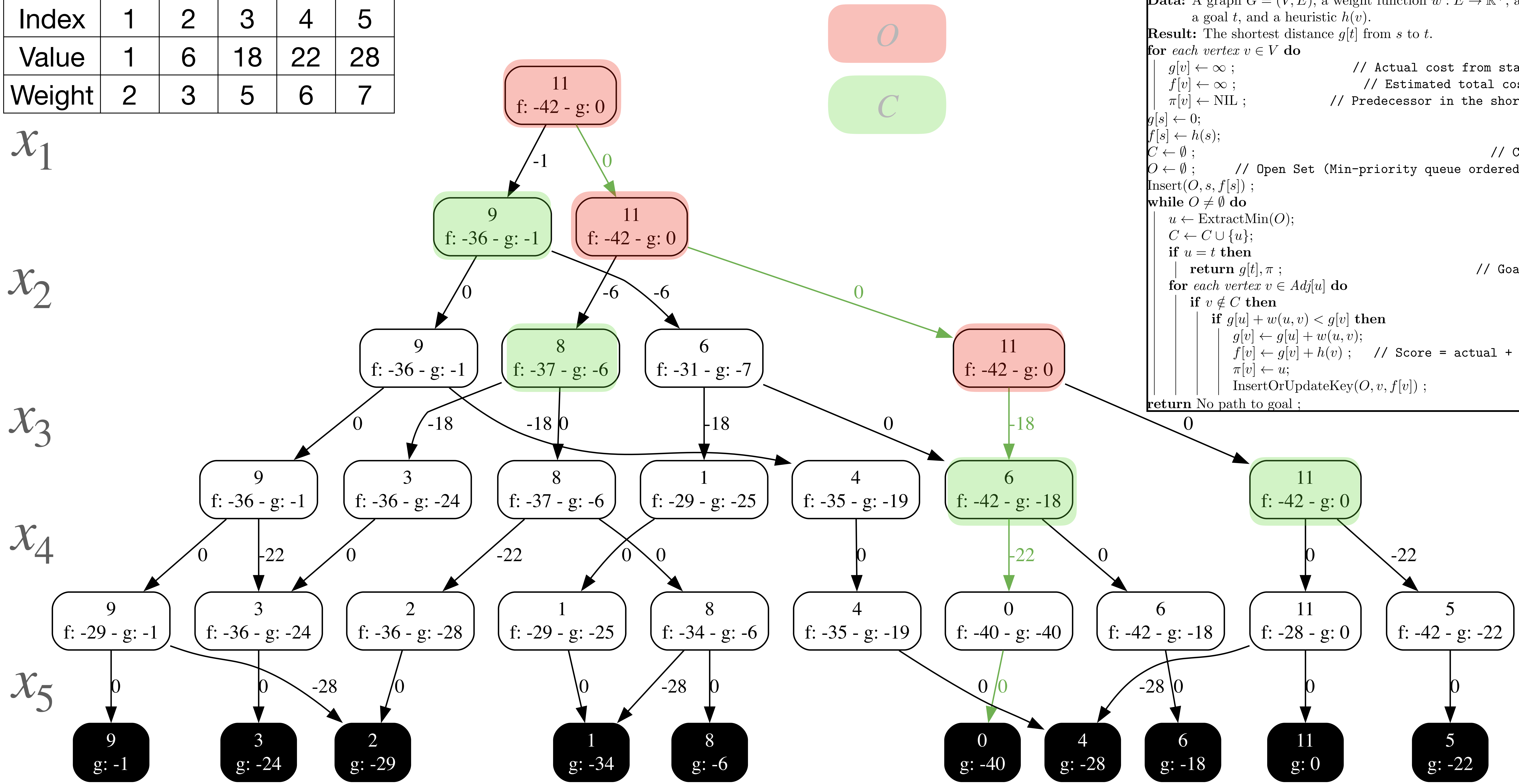
for *each vertex* $v \in V$ **do**
- $g[v] \leftarrow \infty$; `// Actual cost from start s to v`
- $f[v] \leftarrow \infty$; `// Estimated total cost (g + h)`
- $\pi[v] \leftarrow$ NIL ; `// Predecessor in the shortest path`

$g[s] \leftarrow 0$;
$f[s] \leftarrow h(s)$;
$C \leftarrow \emptyset$; `// Closed Set`
$O \leftarrow \emptyset$; `// Open Set (Min-priority queue ordered by f[v])`
Insert$(O, s, f[s])$;
while $O \neq \emptyset$ **do**
- $u \leftarrow$ ExtractMin(O);
- $C \leftarrow C \cup \{u\}$;
- **if** $u = t$ **then**
  - **return** $g[t], \pi$; `// Goal reached`
- **for** *each vertex* $v \in Adj[u]$ **do**
  - **if** $v \notin C$ **then**
    - **if** $g[u] + w(u, v) < g[v]$ **then**
      - $g[v] \leftarrow g[u] + w(u, v)$;
      - $f[v] \leftarrow g[v] + h(v)$; `// Score = actual + heuristic`
      - $\pi[v] \leftarrow u$;
      - InsertOrUpdateKey$(O, v, f[v])$;

return No path to goal ;

Resolution with A*, h = linear relaxation

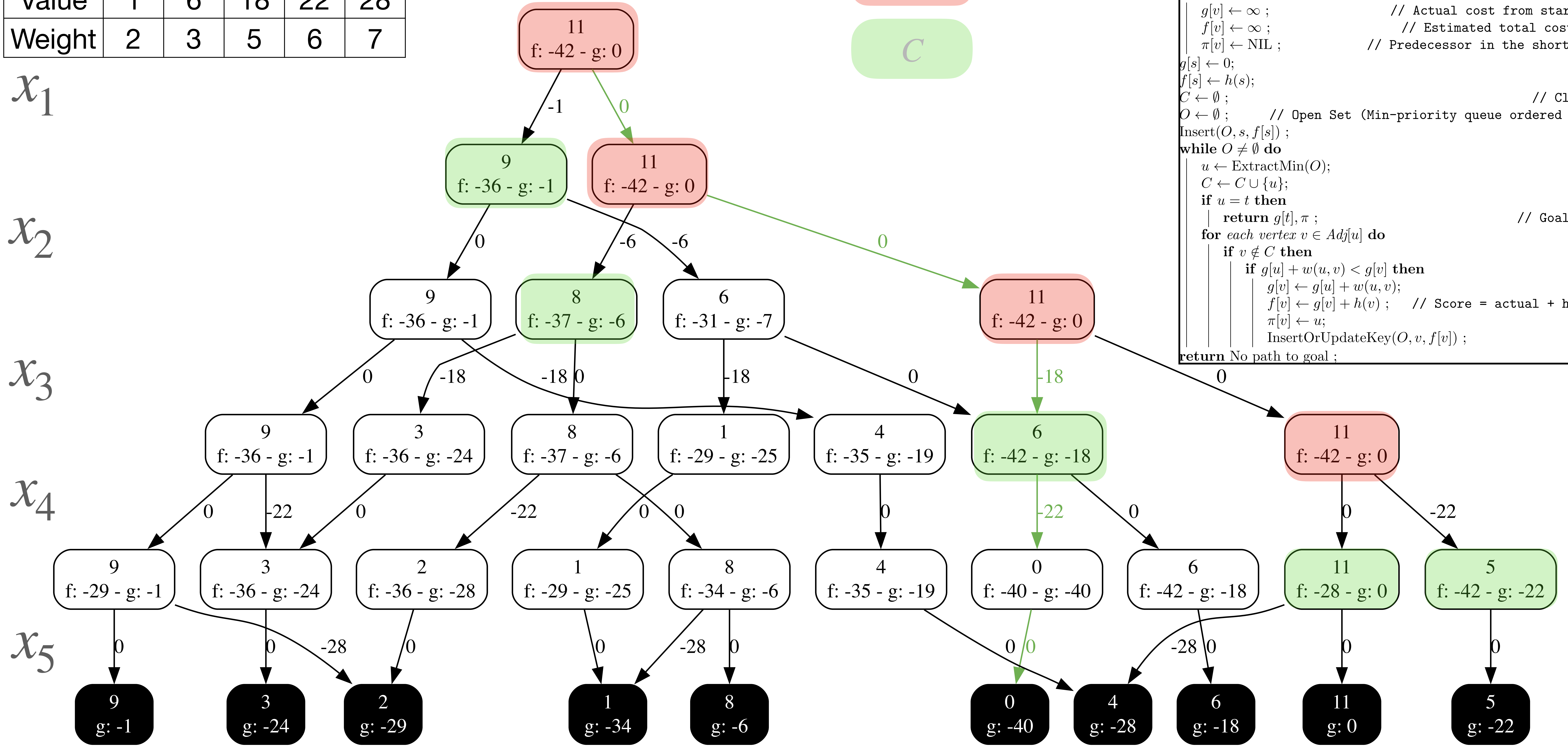
Index	1	2	3	4	5
Value	1	6	18	22	28
Weight	2	3	5	6	7




```
Data: A graph G = (V, E), a weight function w : E → ℝ+, a source s,
      a goal t, and a heuristic h(v).
Result: The shortest distance g[t] from s to t.
for each vertex v ∈ V do
    g[v] ← ∞ ;                         // Actual cost from start s to v
    f[v] ← ∞ ;                          // Estimated total cost (g + h)
    π[v] ← NIL ;                     // Predecessor in the shortest path
g[s] ← 0;
f[s] ← h(s);
C ← ∅ ;                                                  // Closed Set
O ← ∅ ;      // Open Set (Min-priority queue ordered by f[v])
Insert(O, s, f[s]) ;
while O ≠ ∅ do
    u ← ExtractMin(O);
    C ← C ∪ {u};
    if u = t then
        return g[t], π ;                                 // Goal reached
    for each vertex v ∈ Adj[u] do
        if v ∉ C then
            if g[u] + w(u, v) < g[v] then
                g[v] ← g[u] + w(u, v);
                f[v] ← g[v] + h(v) ;    // Score = actual + heuristic
                π[v] ← u;
                InsertOrUpdateKey(O, v, f[v]) ;
return No path to goal ;
```

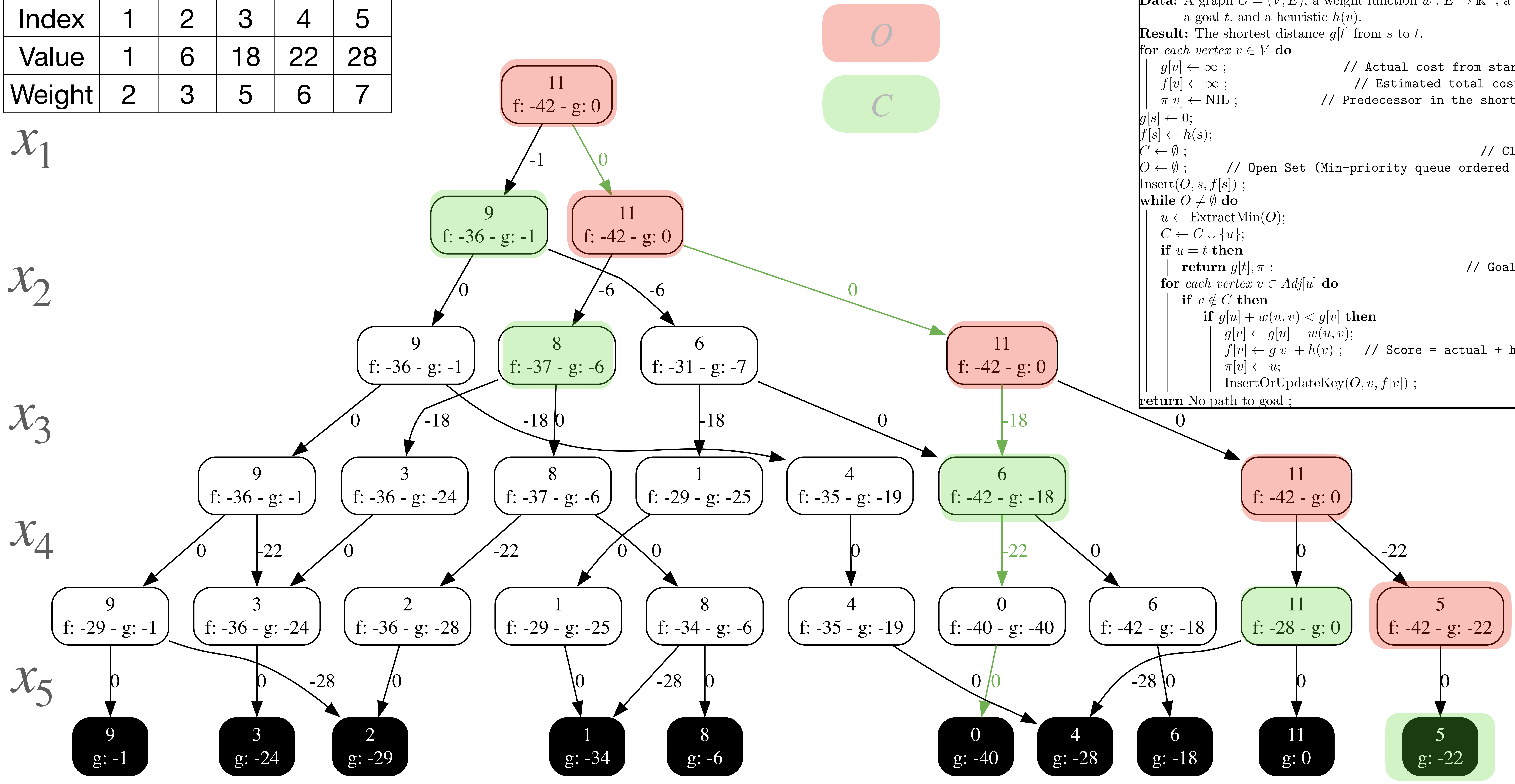
Resolution with A*, h = linear relaxation

Index	1	2	3	4	5
Value	1	6	18	22	28
Weight	2	3	5	6	7



Data: A graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}^+$, a source s, a goal t, and a heuristic $h(v)$.
Result: The shortest distance $g[t]$ from s to t.
for *each vertex* $v \in V$ **do**
  $g[v] \leftarrow \infty$; `// Actual cost from start s to v`
  $f[v] \leftarrow \infty$; `// Estimated total cost (g + h)`
  $\pi[v] \leftarrow$ NIL ; `// Predecessor in the shortest path`
$g[s] \leftarrow 0$;
$f[s] \leftarrow h(s)$;
$C \leftarrow \emptyset$; `// Closed Set`
$O \leftarrow \emptyset$; `// Open Set (Min-priority queue ordered by f[v])`
Insert$(O, s, f[s])$;
while $O \neq \emptyset$ **do**
  $u \leftarrow$ ExtractMin(O);
  $C \leftarrow C \cup \{u\}$;
  if $u = t$ **then**
    return $g[t], \pi$; `// Goal reached`
  for *each vertex* $v \in Adj[u]$ **do**
    if $v \notin C$ **then**
      if $g[u] + w(u, v) < g[v]$ **then**
        $g[v] \leftarrow g[u] + w(u, v)$;
        $f[v] \leftarrow g[v] + h(v)$; `// Score = actual + heuristic`
        $\pi[v] \leftarrow u$;
        InsertOrUpdateKey$(O, v, f[v])$;
return No path to goal ;

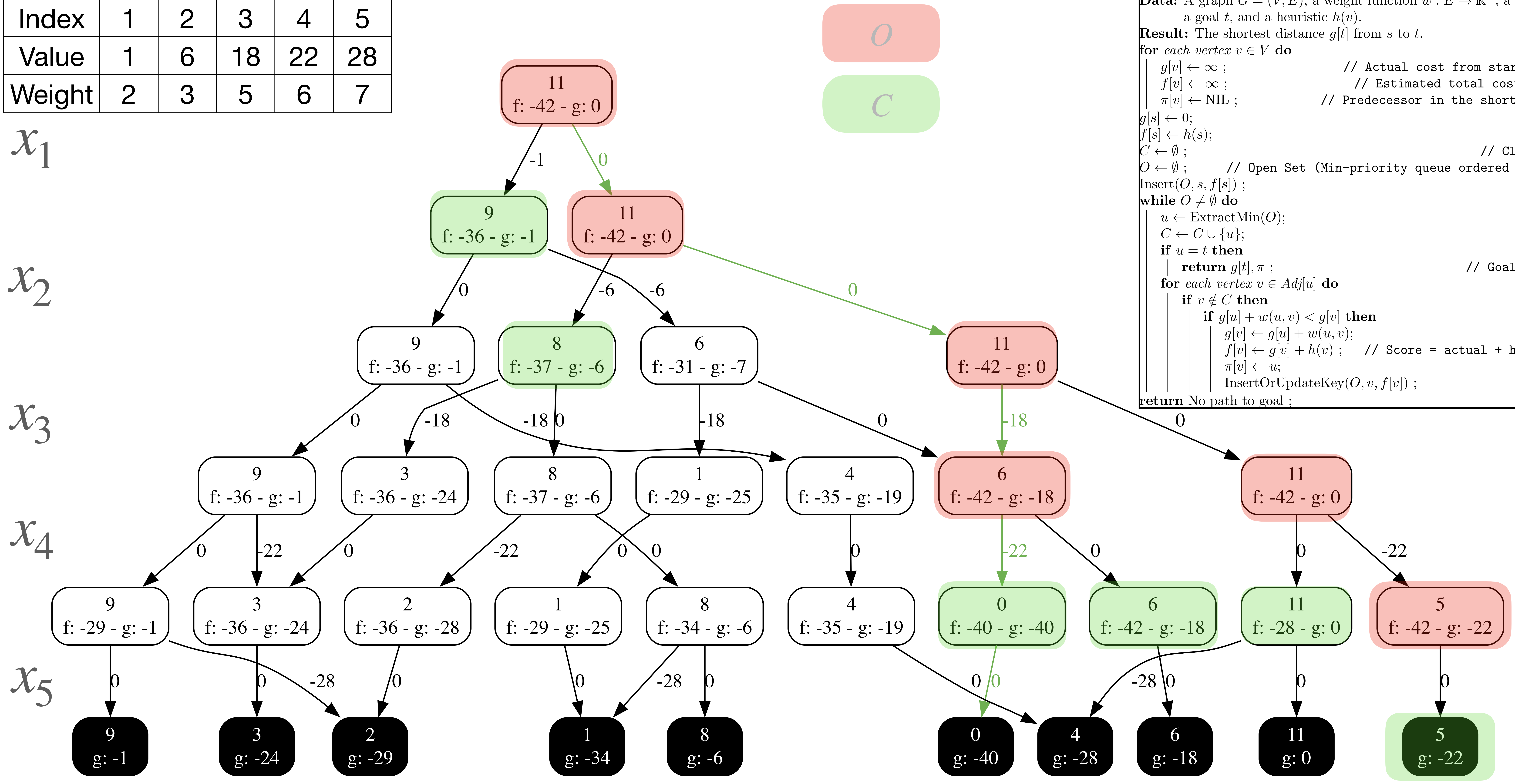
Index	1	2	3	4	5
Value	1	6	18	22	28
Weight	2	3	5	6	7



Data: A graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}^+$, a source s, a goal t, and a heuristic $h(v)$.
Result: The shortest distance $g[t]$ from s to t.

```
for each vertex v ∈ V do
    g[v] ← ∞ ;                        // Actual cost from start s to v
    f[v] ← ∞ ;                        // Estimated total cost (g + h)
    π[v] ← NIL ;                      // Predecessor in the shortest path
g[s] ← 0;
f[s] ← h(s);
C ← ∅ ;                               // Closed Set
O ← ∅ ;      // Open Set (Min-priority queue ordered by f[v])
Insert(O, s, f[s]) ;
while O ≠ ∅ do
    u ← ExtractMin(O);
    C ← C ∪ {u};
    if u = t then
        return g[t], π ;              // Goal reached
    for each vertex v ∈ Adj[u] do
        if v ∉ C then
            if g[u] + w(u, v) < g[v] then
                g[v] ← g[u] + w(u, v);
                f[v] ← g[v] + h(v) ;  // Score = actual + heuristic
                π[v] ← u;
                InsertOrUpdateKey(O, v, f[v]) ;
return No path to goal ;
```

Index	1	2	3	4	5
Value	1	6	18	22	28
Weight	2	3	5	6	7



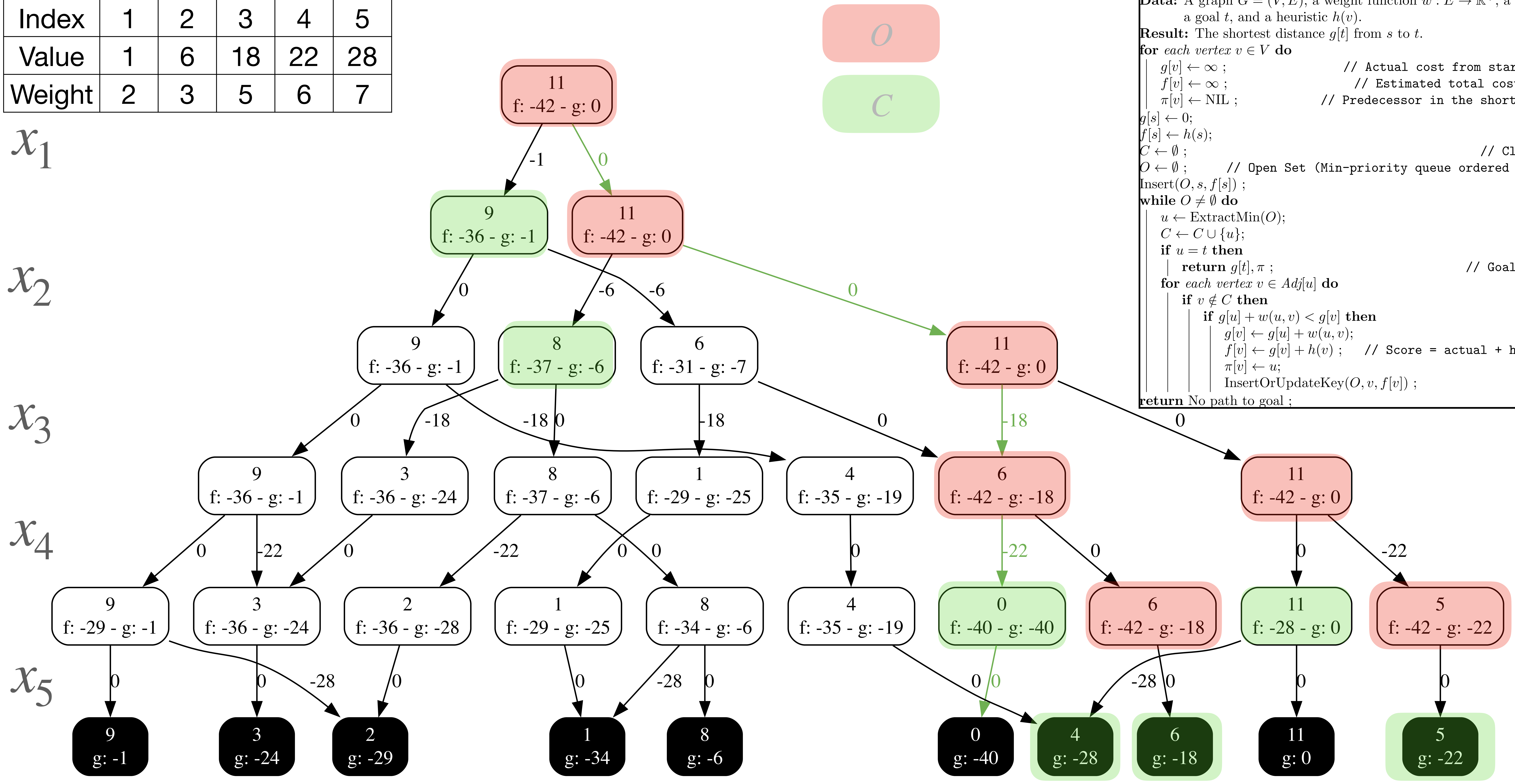
Data: A graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}^+$, a source s, a goal t, and a heuristic $h(v)$.
Result: The shortest distance $g[t]$ from s to t.

for *each vertex* $v \in V$ **do**
- $g[v] \leftarrow \infty$; `// Actual cost from start s to v`
- $f[v] \leftarrow \infty$; `// Estimated total cost (g+h)`
- $\pi[v] \leftarrow$ NIL ; `// Predecessor in the shortest path`

$g[s] \leftarrow 0$;
$f[s] \leftarrow h(s)$;
$C \leftarrow \emptyset$; `// Closed Set`
$O \leftarrow \emptyset$; `// Open Set (Min-priority queue ordered by f[v])`
Insert$(O, s, f[s])$;
while $O \neq \emptyset$ **do**
- $u \leftarrow$ ExtractMin(O);
- $C \leftarrow C \cup \{u\}$;
- **if** $u = t$ **then**
  - **return** $g[t], \pi$; `// Goal reached`
- **for** *each vertex* $v \in Adj[u]$ **do**
  - **if** $v \notin C$ **then**
    - **if** $g[u] + w(u, v) < g[v]$ **then**
      - $g[v] \leftarrow g[u] + w(u, v)$;
      - $f[v] \leftarrow g[v] + h(v)$; `// Score = actual + heuristic`
      - $\pi[v] \leftarrow u$;
      - InsertOrUpdateKey$(O, v, f[v])$;

return No path to goal ;

Index	1	2	3	4	5
Value	1	6	18	22	28
Weight	2	3	5	6	7



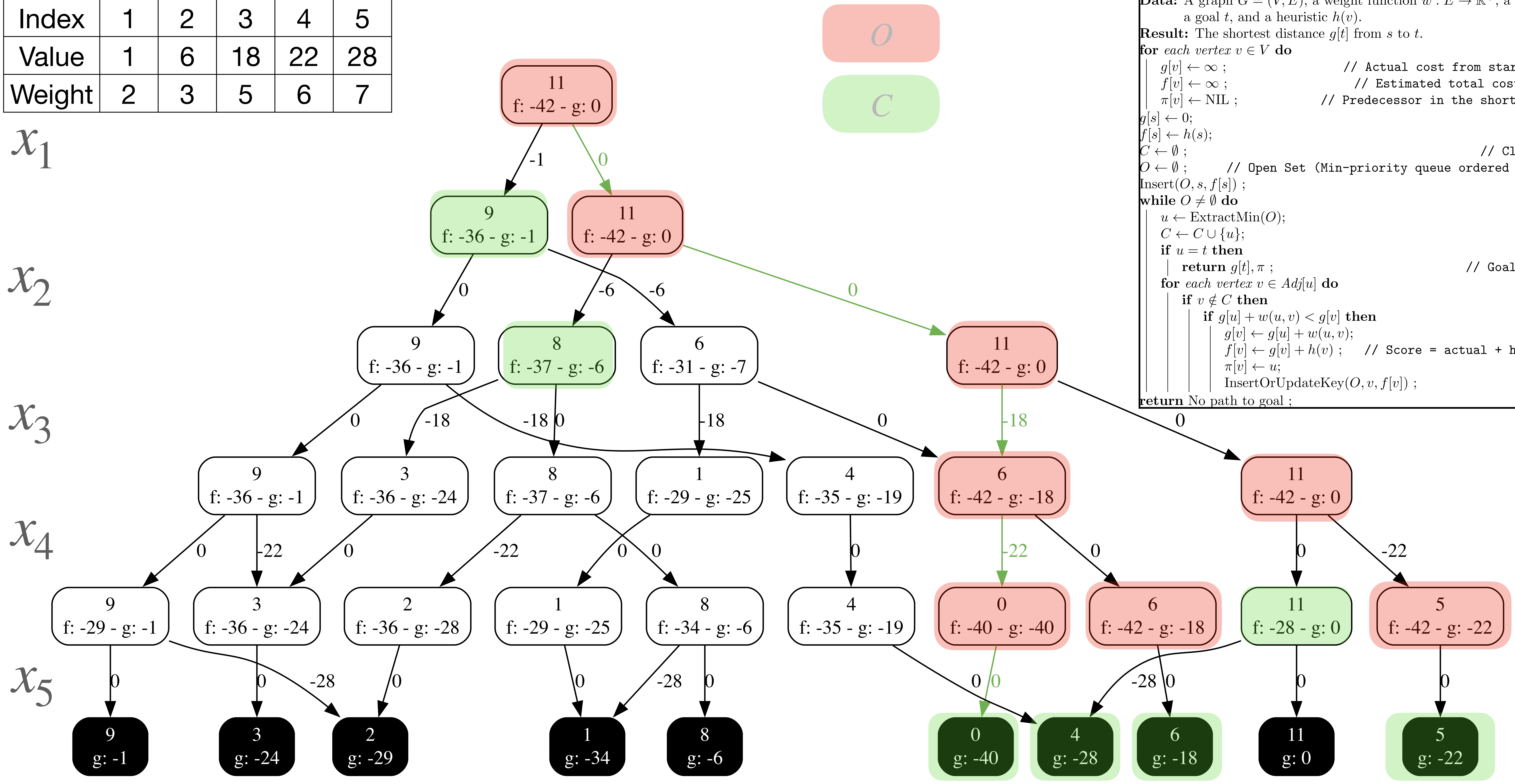
Data: A graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}^+$, a source s, a goal t, and a heuristic $h(v)$.
Result: The shortest distance $g[t]$ from s to t.

for *each vertex* $v \in V$ **do**
- $g[v] \leftarrow \infty$; `// Actual cost from start s to v`
- $f[v] \leftarrow \infty$; `// Estimated total cost (g + h)`
- $\pi[v] \leftarrow$ NIL ; `// Predecessor in the shortest path`

$g[s] \leftarrow 0$;
$f[s] \leftarrow h(s)$;
$C \leftarrow \emptyset$; `// Closed Set`
$O \leftarrow \emptyset$; `// Open Set (Min-priority queue ordered by f[v])`
Insert$(O, s, f[s])$;
while $O \neq \emptyset$ **do**
- $u \leftarrow$ ExtractMin(O);
- $C \leftarrow C \cup \{u\}$;
- **if** $u = t$ **then**
  - **return** $g[t], \pi$; `// Goal reached`
- **for** *each vertex* $v \in Adj[u]$ **do**
  - **if** $v \notin C$ **then**
    - **if** $g[u] + w(u, v) < g[v]$ **then**
      - $g[v] \leftarrow g[u] + w(u, v)$;
      - $f[v] \leftarrow g[v] + h(v)$; `// Score = actual + heuristic`
      - $\pi[v] \leftarrow u$;
      - InsertOrUpdateKey$(O, v, f[v])$;

return No path to goal ;

Index	1	2	3	4	5
Value	1	6	18	22	28
Weight	2	3	5	6	7



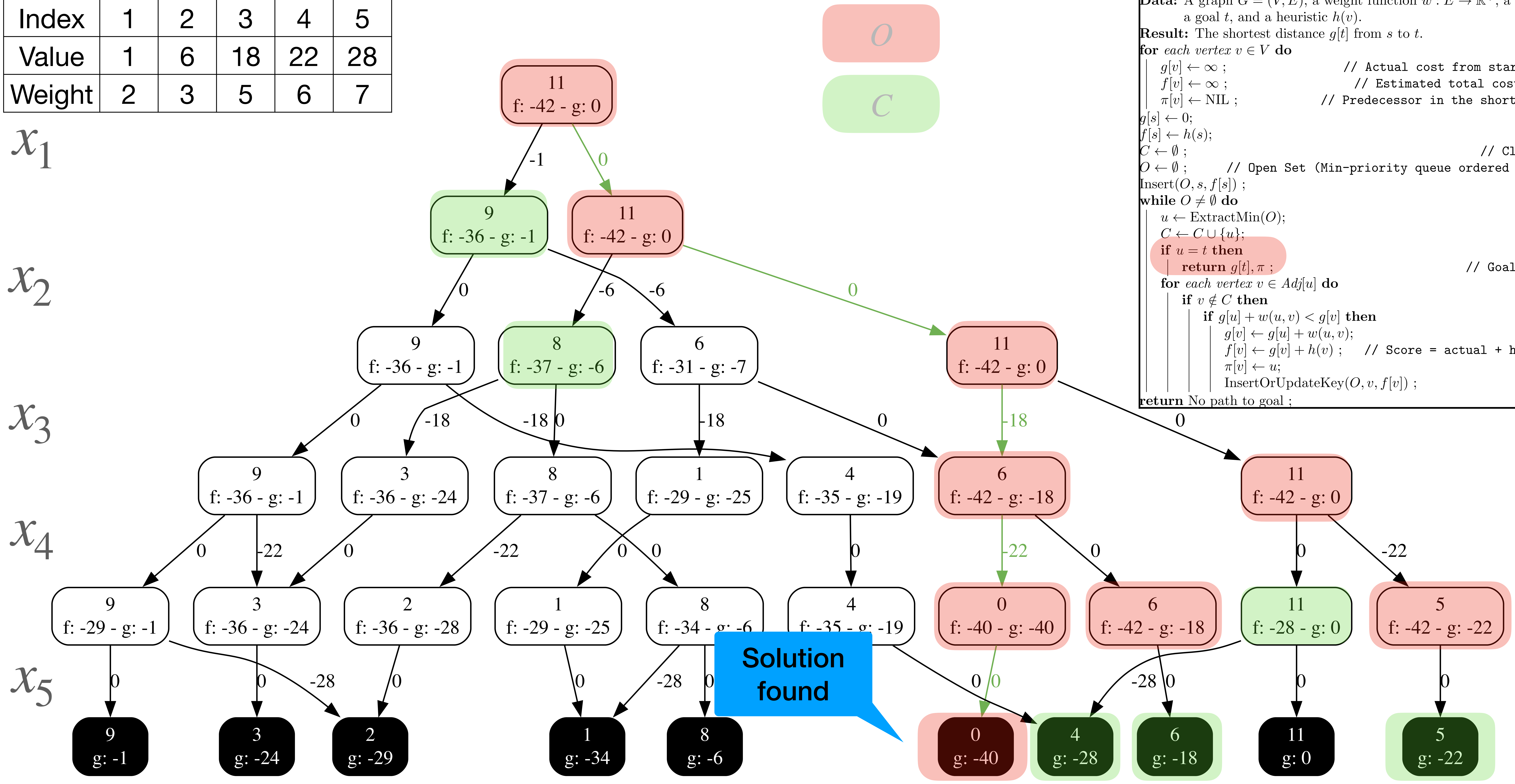
Data: A graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}^+$, a source s, a goal t, and a heuristic $h(v)$.
Result: The shortest distance $g[t]$ from s to t.

```
for each vertex v ∈ V do
    g[v] ← ∞ ;                           // Actual cost from start s to v
    f[v] ← ∞ ;                           // Estimated total cost (g + h)
    π[v] ← NIL ;                         // Predecessor in the shortest path
g[s] ← 0;
f[s] ← h(s);
C ← ∅ ;                                  // Closed Set
O ← ∅ ;      // Open Set (Min-priority queue ordered by f[v])
Insert(O, s, f[s]) ;
while O ≠ ∅ do
    u ← ExtractMin(O);
    C ← C ∪ {u};
    if u = t then
        return g[t], π ;                 // Goal reached
    for each vertex v ∈ Adj[u] do
        if v ∉ C then
            if g[u] + w(u, v) < g[v] then
                g[v] ← g[u] + w(u, v);
                f[v] ← g[v] + h(v) ;     // Score = actual + heuristic
                π[v] ← u;
                InsertOrUpdateKey(O, v, f[v]) ;
return No path to goal ;
```

Index	1	2	3	4	5
Value	1	6	18	22	28
Weight	2	3	5	6	7

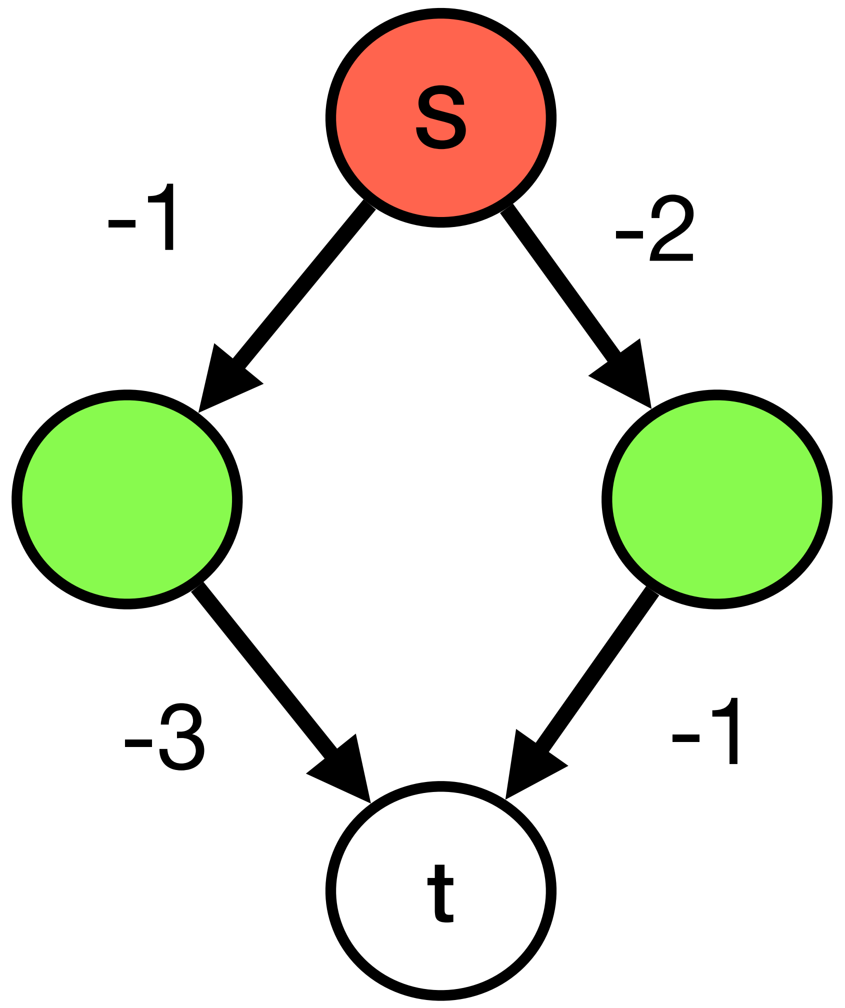
Data: A graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}^+$, a source s, a goal t, and a heuristic $h(v)$.
Result: The shortest distance $g[t]$ from s to t.

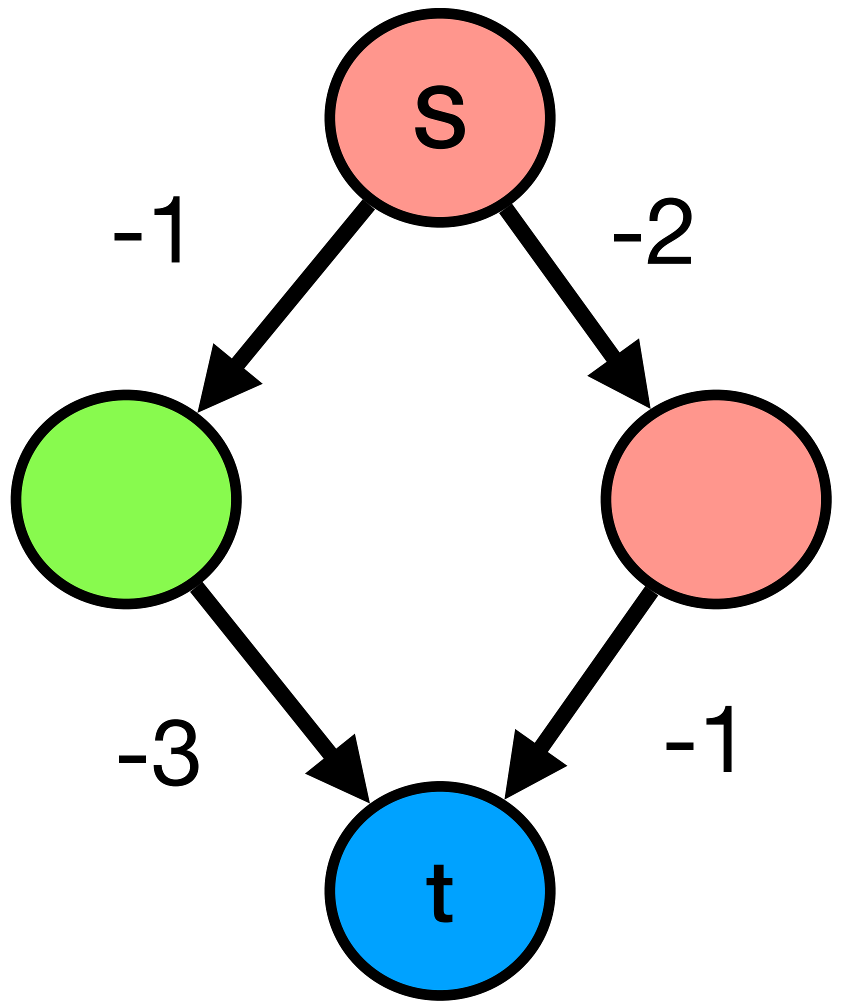
for *each vertex* $v \in V$ **do**
  $g[v] \leftarrow \infty$; `// Actual cost from start s to v`
  $f[v] \leftarrow \infty$; `// Estimated total cost (g+h)`
  $\pi[v] \leftarrow$ NIL ; `// Predecessor in the shortest path`
$g[s] \leftarrow 0$;
$f[s] \leftarrow h(s)$;
$C \leftarrow \emptyset$; `// Closed Set`
$O \leftarrow \emptyset$; `// Open Set (Min-priority queue ordered by f[v])`
Insert$(O, s, f[s])$;
while $O \neq \emptyset$ **do**
  $u \leftarrow$ ExtractMin(O);
  $C \leftarrow C \cup \{u\}$;
  if $u = t$ **then**
    return $g[t], \pi$; `// Goal reached`
  for *each vertex* $v \in Adj[u]$ **do**
    if $v \notin C$ **then**
      if $g[u] + w(u, v) < g[v]$ **then**
        $g[v] \leftarrow g[u] + w(u, v)$;
        $f[v] \leftarrow g[v] + h(v)$; `// Score = actual + heuristic`
        $\pi[v] \leftarrow u$;
        InsertOrUpdateKey$(O, v, f[v])$;
return No path to goal ;



Shortest path problems with negative weights are difficult

- Dijkstra does not work with negative weights, even in a DAG





Stern, R., Kiesel, S., Puzis, R., Felner, A., & Ruml, W. (2014). Max is more than min: Solving maximization problems with heuristic search. In *Proceedings of the International Symposium on Combinatorial Search.*

- Dijkstra algorithm always picks the node with the smallest current distance to process next, it assumes that adding edges to a path can only increase (or keep equal) the total distance. Negative edges violate this assumption because reaching a "farther" node first might allow you to cross a negative edge that reduces the total cost to a "closer" node.

- Thus A* may not work neither! We need to fix it to work with negative weights

Let's fix A* (we call it Robust A*) to work with negative cycles in a DAG

Data: A graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}^+$, a source s, a goal t, a heuristic $h(v)$, a weight parameter $w_h \geq 1$.
Result: The cost of the best solution found, g_{inc}, and the incumbent path π_{inc}.

```
for each vertex v ∈ V do
    g[v] ← ∞ ;                          // Cost of best known path s ⇝ v
    f[v] ← ∞ ;                          // Estimated cost: g + h
    π[v] ← NIL;                         // Best known predecessor
g[s] ← 0;
f[s] ← h(s);
g_inc ← ∞ ;                             // Incumbent cost
C ← ∅ ;       // Closed Set (Contains nodes that were expanded)
O ← ∅ ;       // Open Set (Min-priority queue ordered by f'[v])
Insert(O, s, f[s]);
while O ≠ ∅ do
    u ← ExtractMin(O);
    C ← C ∪ {u};
    if f[u] > g_inc then
        return g_inc, π ;               // Cannot do better than incument
    for each vertex v ∈ Adj[u] do
        if g[u] + w(u, v) + h(v) ≥ g_inc then
            continue ;                  // Pruning by incumbent cost
        else if g[u] + w(u, v) < g[v] then
            π[v] ← u ;
            g[v] ← g[u] + w(u, v) ;
            f[v] ← g[v] + h(v) ;
            if v = t then
                g_inc ← g[t] ;          // Incumbent cost update
            else
                InsertOrUpdateKey(O, v, f[v]);
                C ← C \ {v} ;
return g_inc, π ;                       // O = ∅ (optimal)
```

A* cannot stop on first solution. It can be stopped only when the set of open-nodes is empty or when the minimum f value is worse than the incumbent solution

Project

- You are given an implementation of A* and a model for the Knapsack
- You have to implement the Anytime Weighted A*
- You have to implement the TSP and solve it using A* and Anytime Weighted A*
  - For that you need to implement a heuristic h for the TSP

KnapsackState

```java
public class KnapsackState extends State {

    int item, capacity;

    public KnapsackState(int index, int capacity) {
        this.item = index;
        this.capacity = capacity;
    }

    @Override
    public int hash() {
        return Objects.hash(item, capacity);
    }

    @Override
    public boolean isEqual(State s) {
        if (s instanceof KnapsackState) {
            KnapsackState state = (KnapsackState) s;
            return item == state.item && capacity == state.capacity;
        }
        return false;
    }
}
```

Model

```java
/**
 * Interface for describing an A* based model
 */
public abstract class Model<S extends State> {

    /**
     * @return true if the state is a base case of the A* model
     */
    public abstract boolean isTerminalState(S state);

    /**
     * @return the value of the base case
     */
    public abstract double getTerminalStateValue(S state);

    /**
     * @return the root state of the A* model
     */
    public abstract S getRootState();

    /**
     * @return the list of transitions from the given state
     */
    public abstract List<Transition<S>> getTransitions(S state);


    public abstract double h(S state);
}
```

Knapsack (Model)

```java
public class Knapsack extends Model<KnapsackState> {

    KnapsackInstance instance;
    KnapsackState root;

    public Knapsack(KnapsackInstance instance) {
        this.instance = instance;
        this.root = new KnapsackState(0, instance.capacity);
    }

    @Override
    public boolean isTerminalState(KnapsackState state) {
        return state.item == instance.n || state.capacity == 0;
    }

    @Override
    public double getTerminalStateValue(KnapsackState state) {
        return 0;
    }

    @Override
    public KnapsackState getRootState() {
        return root;
    }
```

Knapsack (Model)

```java
@Override
public List<Transition<KnapsackState>> getTransitions(KnapsackState state) {
    List<Transition<KnapsackState>> transitions = new LinkedList<>();

    // do not take the item
    transitions.add(new Transition<KnapsackState>(
            new KnapsackState(state.item + 1, state.capacity),
            0,
            0
    ));

    // take the item if remaining capacity allows
    if (instance.weight[state.item] <= state.capacity) {
        transitions.add(new Transition<KnapsackState>(
                new KnapsackState(state.item + 1, state.capacity - instance.weight[state.item]),
                1,
                -instance.value[state.item]
        ));
    }

    return transitions;
}
```

Heuristic h for Knapsack = Linear Programming

```java
@Override
public double h(KnapsackState state) {
    double[] ratio = new double[instance.n];
    int capacity = state.capacity;
    for (int i = state.item; i < instance.n; i++) {
        ratio[i] = ((double) instance.value[i] / instance.weight[i]);
    }
    class RatioComparator implements Comparator<Integer> {
        @Override
        public int compare(Integer o1, Integer o2) {
            return Double.compare(ratio[o1], ratio[o2]);
        }
    }
    Integer[] sortedVariables = new Integer[instance.n - state.item];
    for (int i = state.item; i < instance.n; i++) {
        sortedVariables[i - state.item] = i;
    }
    Arrays.sort(sortedVariables, new RatioComparator().reversed());
    int maxProfit = 0;
    Iterator<Integer> itemIterator = Arrays.stream(sortedVariables).iterator();
    while (capacity > 0 && itemIterator.hasNext()) {
        int item = itemIterator.next();
        if (capacity >= instance.weight[item]) {
            maxProfit += instance.value[item];
            capacity -= instance.weight[item];
        } else {
            double itemProfit = ratio[item] * capacity;
            maxProfit += (int) Math.floor(itemProfit);
            capacity = 0;
        }
    }
    return -maxProfit;
}
```

Putting it all together and start the computation

```
KnapsackInstance instance =
   new KnapsackInstance(instance.capacity, instance.value, instance.weight);
Knapsack model = new Knapsack(instance);
Astar<KnapsackState> solver = new Astar<>(model);
Solution solution = solver.getSolution();
for (int decision : solution.getDecisions()) {
    if (decision == 1) {
        checkValue += instance.value[item];
        checkWeight += instance.weight[item];
    }

    item++;
}
```