25/40

In [ ]:

```
Solution submitted by:

Hsin-Yu Ku (3038591)
Chân Lê (3038545)
Lukas Schmid (3038594)
```

Thank you for pointing out the incomplete parts of your implementation. You have make an extensive explanation to let me understand your ideas. Furthermore, you should submit your codes as .ipynb.

Summary of the notation o and its translation into code

Sets ¶

$G = (V, A)$ is the graph, where $V = \{1, 2, \ldots, N\}$. Arcs are directional, where arc (i, j) leads from node i to node j (and vice versa). All arcs can be travelled in both directions.

The journey of one train, its path, is denoted as $P = \{1, i_1, i_2, \ldots, N\}$, so that 1 is the origin of the path, N its destination, and $i_1, i_2, \ldots$ are the nodes travelled. A feasible path is one where all movement constraints and time constraints are fulfilled.

Parameters and Variables

$f(i, j)$: according to the authors, it is the earliest feasible arrival time at node i, provided one can depart along arc (i, j). For now, a better interpretation seems to be that $f(i, j)$ is the arrival time at node i under the current solution for a given path. Written as `f[i][j]`

Constraints

Arc $a = (i, j)$ is closed for all other trains when a train travels along it from station i to statoin j, departing at time t and traveling for duration $d(a)$. This closing period is denoted as

$$[t, t + d(a)]$$

A station is closed for a security duration Δ after a train leaves a station.

A station has separate capacities for parking and passing through. Only one train can be parked at a station at each moment, but on other train can pass through the station while one other train is parked there.

A route along arc (i, j) is blocked if, for this specific route, the train can not depart from mode i to node j. This can have different reasons, which will be explained below.

Splitting the problem

Calculating arrival times

The problem can be solved by finding the earliest arrival time at each node, beginning from the start node. Hence, the main task is to calculate an arrival time for a current solution, then updating it if later restrictions make that necessary.

Once the arrival time $f(i,j)$ is known, there are three different possibilities for the arrival time $f(j,k)$:

1. The path is not blocked along arc (j,k). That means that the earliest arrival time $f(j,k)$ is the arrival time $f(i,j)$, plus waiting time at node i and travel time on arc (i,j), for the node i that minimises this sum.
$$f(j,k) = \min_{i,p_i}[f(i,j) + p_i + t_{ij}]$$
We have to account for the possibility that an arc may be reached via different routes.
2. The path is blocked along arc (j,k) and can not be unblocked. In that case, the route can not contain this arc. A route is irreversibly blocked if the train arrives after the closing of the departure time window.
3. A path is blocked along arc (i,j), but can be unblocked. In this case, the train might arrive within either the parking time window of departure time window by extending its parking time at other nodes or taking another route. The last option is not considered by the authors. The first option means that we have now two possible values for $f(i_1,i_2)$: to the already existing solution derived from earlier steps, we add the new one with increased parking times in some node preceding i_1.

If an arc is blocked for a current path solution, we write $f(i,j) = \infty$.

Blocked routes

A path is blocked on arc (i,j) if

1. Departure is allowed before parking (it is not necessary to wait for departure while parking), but arrival occurs before departure is allowed: $f(i,j) < \gamma_{ij} < \alpha_i$
2. Departure is allowed at some point in time during the parking period (it is possible to wait for departure while parking), but arrival occurs before parking is allowed: $f(i,j) < \alpha_i < \gamma_{ij} < \beta_i$
3. Departure is allowed after the parking period (it is not possible to wait for departure while parking), and arrival occurs before parking: $f(i,j) < \alpha_i < \beta_i < \gamma_{ij}$
4. Arrival occurs after departure is allowed: $\delta_{ij} < f(i,j)$

Reading in the data

In [2]:

```
import math
import numpy as np
import copy

# all fixed values:
alphaList = [0, 18, 35, 64, 35, 64, 0]
betaList  = [math.inf, 30, 45, 75, 50, 75, math.inf]
gammaList = [10, 10, 15, 28, 26, 32, 35, 43, 70, 30, 48, 72]
deltaList = [15, 25, 25, 35, 34, 60, 50, 50, 99, 80, 55, 80]
tTimeList = [10, 20, 25, 35, 35, 30, 15, 20, 30, 20, 25, 10]
nodes     = [1, 2, 3, 4, 5, 6, 7]
arcs      = {1:[2, 3, 5], 2:[4, 6, 7], 3:[2, 4], 4:[7], 5:[2, 6], 6:[7]}
feasible  = []

# to facilitate indexing, we zip the lists into dictionaries:
alpha = dict(zip(nodes, alphaList))
beta  = dict(zip(nodes, betaList ))
# gamma and delta have tuples as keys
arccount = 0
gamma = {}
delta = {}
tTime = {}
feasible = {}
for start in arcs.keys():
    for end in arcs[start]:
        gamma[(start, end)] = gammaList[arccount]
        delta[(start, end)] = deltaList[arccount]
        tTime[(start, end)] = tTimeList[arccount]
        feasible[(start, end)] = 0
        arccount += 1
```

Finding a solution

Explanation of the algorithm

We decided to use a pseudo-object-oriented programming approach. There is one dictionary structure used throughout the algorithm:

In [248]:

```
{'path': [1, 3],
 'parking': {1: [0, 10]},
 'arrival': {1: 0, 3: 20}}
```

Out[248]:

```
{'path': [1, 3], 'parking': {1: [0, 10]}, 'arrival': {1: 0, 3: 20}}
```

Each dictionary contains one possible path. In the key `'path'` , all visited nodes of that path are stored as a list. In `'parking` , the start and the end of the real parking times are stored as a list, whose key is the node at which the parking occurs. `'arrival'` stores the arrival times at each node under current conditions of the path (actual parking times), again in a dictionary. This allows for referencing of path information in the following style: `pathName['information'][node]` , where node is an integer.

Note that `'parking'` could store integers rather than lists, because `pathName['parking'][node][0] == pathName['arrival'][node]` .

The core of the algorithm will be the same as described in the dynamic programming chapter of the slides:

1. **Initiate a set U that contains as only path the path [1]; initiate a set P that is empty.** Both U and P are dictionaries, whose keys are the complete paths, and whose items are path-dictionaries as described above.
2. **While U is not empty, draw and remove a path from U**. The drawn path is saved as `nextPath` . Selection happens in the function `selectNextPath()` , which returns the first key-value-pair from U. Deletion happens explicitly, outside the function.
3. **For all possible extensions to `nextPath` , check whether the extension is feasible; if it is, copy `nextPath` to P.** This is done inside the function `arrivalTime(pathInfo)` , which will be explained below in detail. Note that we could have written `arrivalTime()` in a way that, like `selectNextPath()` , it does not accept any arguments and instead works with the `nextPath` object. This was not done, mainly because we were rather used to a more functional style of programming, and debugging proved easier with the use of an argument.

Explanation of `arrivalTime()`

The feasibility check differs from the solving algorithm's explanation given in Sancho (1992):

1. There is never an explicit calculation of $f(i,j)$ for any pair of nodes.
2. Logical implementation of comparing arrival with departure and parking time windows is different.

In further explanation, we use the following terminology:

- The end node (in code: `endNode`) is the last node of the path whose possible extensions we currently check.
- The next node (in code: `nextNode`) is one potential extension node of the same path.
- The previous node (`previousNode`) is the node in the current path before the end node.
- Arrival time (accessed in code by `pathInfo["arrival"][node]`) is the actual arrival time at a node, *given its current parking times at all previous nodes*. Most importantly, this is *not* the *earliest feasible arrival time* as described in Sancho (1992).
- The current path (`pathInfo`) is always the paths whose extensions we currently evaluate.
- The departure and parking time windows of a node i (to node j) can be defined as $[\gamma_{ij}, \delta_{ij}]$ and $[\alpha_i, \beta_i]$.

`pathInfo` (see example above) always must contain a path as a list, actual parking times at each node other than the end node, and arrival times at each node (including the end node). The function first starts a for-loop over all possible extensions to the end node (if the current end node is 7, then the path is directly copied to P). For every next node, a copy is made of the current path to the object `currentPath` . Following this, the feasibility check is performed:

1. Check whether arrival at current end node occurs after the departure time window to the next node closes. This corresponds to equation (5) in Sancho (1992). In this case, the next iteration of the for-loop begins, meaning that the current extension is discarded.
2. Check whether arrival at end node is in departure time window to next node. This corresponds to equation (4) in Sancho (1992); note that we did not check for α_i, because regardles of its value, parking under this condition is 0 and the extension is feasible. The arrival time at the next node can thus be calculated by adding arrival time at the end node and travel time from there to the next node.
3. Check whether arrival at end node occurs in parking time window and parking time window overlaps with departure time window. This corresponds to equation (1) in Sancho (1992) (slightly altered). In this case, actual parking is set to end when the departure time window opens, and arrival at the next node is the end of the actual parking time at the end node, plus travel time to the next node.
4. Check whether the arrival time is too early. This should always be the case, except when arrival occurs in the parking time window, but parking time window and departure time window don't overlap. The algorithm calculates the necessary delay of arrival at the end node (`tooEarly`), and sets parking at the end node to 0. Then it jumps to the previous node. There it determines a the new parking period, which starts witht the arrival at the previous node, but ends at the earliest of end of departure time to the end node, end of waiting time window, or necessary delay of arrival time at end node. This way, the algorithm makes sure that parking time is within allowed boundaries and not larger than necessary. After having changed parking time at the previous node, the algorithm jumps back to the end node and calculates the new arrival time there. Now, the conditions 2. and 3. from above are implemented and the path is changed accordingly. Note that condition 1. does not need to be checked again; this is ensured by calculation of the end of the new parking period at the previous node.

In [10]:

```
def arrivalTime(pathInfo):
    """Writing a function to check for feasibility of arrival time and
    calculate the arrival time for some suggested extension
    of a current path, taking into account the current parking and travel times
    at all nodes in this path (without the extension). Here we assume that the
    arrival times given in the path are all feasible.

    pathinfo must be a dictionary with the keys path, parking and arrival.
    """
    endNode = pathInfo["path"][-1]
    print("> Last node of current path:", endNode)
    if endNode == 7:
        movePath = tuple(pathInfo["path"])
        print("> Key of path to remove is:")
        print(movePath)
        P[movePath] = copy.deepcopy(pathInfo)
        return None
    # get all possible extension (arcs in A)
    nextNodes = arcs[endNode]
    print("> Current arrival time: ", pathInfo["arrival"][endNode])
    print("> Possible Extensions:", nextNodes)
    print(f"> Waiting time window at node {endNode}: [", alpha[endNode], ",",
        beta[endNode], "]", sep = "")
    # calculate new arrival times for all next nodes
    for nextNode in nextNodes:
        # getting the new arc
        currentArc = tuple([endNode, nextNode])
        # making a new entry in the dictionary, i.e. writing to the new path
        suggestedPath = copy.deepcopy(pathInfo)
        suggestedPath["path"].append(nextNode)
        currentPath = tuple(suggestedPath["path"])
        #print("  > pathInfo (this should not change):")
        #print("  ", pathInfo)
        print(f"  > Checking extension {nextNode}")
        #print("    Suggested Path is:")
        #print("  ", suggestedPath)
        print("  > Current path is:")
        print("   ", currentPath)
        print("    > Departure time window: [", gamma[currentArc], ",",
              delta[currentArc], "]", sep = "")
        # 1. Check whether arrival at current end node occurs after the departure time
        # window to the next node closes
        # If True, then arc to suggested next node is permanently blocked.
        if pathInfo["arrival"][endNode] > delta[currentArc]:
            print("    Arrival at current end node occurs after departure time window",
                  f"to \n      node {nextNode} closes.")
        # 2. Check whether arrival at end node is in departure time window to next nod
e.
        # arc to suggested next node is free, parking time not necessary
        elif gamma[currentArc] <= pathInfo["arrival"][endNode] <= delta[currentArc]:
            print(f"    > Arrival time at ({endNode}) is in the departure time window "
,
                  f"([{gamma[currentArc]}, {delta[currentArc]}]). No parking time \n",
                  f"     at node {endNode} necessary.")
            suggestedPath["parking"][endNode] = [suggestedPath["arrival"][endNode],
                                                 suggestedPath["arrival"][endNode]]
            suggestedPath["arrival"][nextNode] = suggestedPath["arrival"][endNode] + \
                                                 tTime[currentArc]
            U[(currentPath)] = copy.deepcopy(suggestedPath)
        # 3. Check whether arrival at end node occurs in parking time window and parkin
g
        # time window overlaps with departure time window.
        # arc to suggested node is not free, but parking on the end node allows us to
        # depart to the next node
        elif (alpha[endNode] <= pathInfo["arrival"][endNode] <= beta[endNode] and
                 gamma[currentArc] <= beta[endNode]):
            suggestedPath["parking"][endNode] = [suggestedPath["arrival"][endNode],
                                                gamma[currentArc]]
            suggestedPath["arrival"][nextNode] = suggestedPath["parking"][endNode][1] +
\
                                                 tTime[currentArc]
            U[(currentPath)] = copy.deepcopy(suggestedPath)
            print("    > Parking is possible and parking time is ",
                 suggestedPath["parking"], ".", sep = "")
        # 4. Check whether arrival time is too early (should always be the case).
        elif pathInfo["arrival"][endNode] < gamma[currentArc]:
            tooEarly = gamma[currentArc] - pathInfo["arrival"][endNode]
            print(f"    > Arriving {tooEarly} units too early.")
            print("    > Parking is not feasible (either because arrival is not in",
                 " waiting time window, or")
            print("      waiting time window and departure time window do not overlap.
 Setting parking at")
            print("      end node to 0.")
            suggestedPath["parking"][endNode] = [suggestedPath["arrival"][endNode],
                                                 suggestedPath["arrival"][endNode]]
            print("    > Path is:", suggestedPath["path"])
            print("    > Checking whether parking time at previous node can be prolonge
d.")
            previousNode = suggestedPath["path"][-3]
            print(f"    > Previous node is {previousNode}.")
            previousArc = (suggestedPath["path"][-3], endNode)
            print("    > Previous arc is", previousArc)
            print("    > Parking time window at previous node is [",
                  alpha[previousNode], ", ", beta[previousNode], "]", sep = "")
            print("    > Parking at previous node currently is",
                 suggestedPath["parking"][previousNode])
            print("    > Limitations on extending waiting time:")
            print("      > End of waiting time window at previous node:",
                  beta[previousNode])
            print("      > End of departure time window at previous node:",
                  delta[previousArc])
            newEndWaitingTime = tooEarly + suggestedPath["parking"][previousNode][1]
            print("      > Necessary end of waiting time:",
                  newEndWaitingTime)
            newEndWaitingTime = min([beta[previousNode], delta[previousArc], newEndWait
ingTime])
            print(f"    > Setting parking time at node {previousNode} to {newEndWaiting
Time}")
            suggestedPath["parking"][previousNode][1] = newEndWaitingTime
            print("    > Updating arrival and parking times at end node:")
            suggestedPath["arrival"][endNode] = suggestedPath["parking"][previousNode][
1] + \
                                                tTime[previousArc]
            suggestedPath["parking"][endNode] = [suggestedPath["arrival"][endNode],
                                                 suggestedPath["arrival"][endNode]]
            print("    > Path now looks like this:",
                  suggestedPath)
            if (gamma[currentArc] <= suggestedPath["arrival"][endNode] <= delta[current
Arc]):
                print("      > New arrival at end node is now in departure time windo
w.")
                suggestedPath["parking"][endNode] = [suggestedPath["arrival"][endNode],
                                                     suggestedPath["arrival"][endNode]]
                suggestedPath["arrival"][nextNode] = suggestedPath["arrival"][endNode]
+ \
                                                     tTime[currentArc]
                print("      > Adding path to U.")
                U[(currentPath)] = copy.deepcopy(suggestedPath)
            elif (alpha[endNode] <= suggestedPath["arrival"][endNode] <= beta[endNode]
and
                gamma[currentArc] <= beta[endNode]):
                print("      > New arrival at end node is now in waiting time window, a
nd")
                print("        waiting time window overlaps with departure time windo
w.")
                print("      > Adding waiting time at end node:")
                suggestedPath["parking"][endNode] = [suggestedPath["arrival"][endNode],
                                                gamma[currentArc]]
                suggestedPath["arrival"][nextNode] = suggestedPath["parking"][endNode][
1] + \
                                                     tTime[currentArc]
                print("      > Adding path to U.")
                U[(currentPath)] = copy.deepcopy(suggestedPath)
            else:
                print("      > New arrival at end node is still outside departure time
 window,")
                print("        or in parking time window, but parking time window does
 not overlap")
                print("        with wating time window. Discarding path.")
        else:
            print("    > Current condition not implemented.")
    # remove fully checked path from U, move it to P
    movePath = tuple(pathInfo["path"])
    print("> Key of path to remove is:")
    print(movePath)
    P[movePath] = copy.deepcopy(pathInfo)
```

In [4]:

```
def selectNextPath(printing = True):
    paths = [path for path in U.keys()]
    selectedPath = paths[0]
    pathInfo     = copy.deepcopy(U[selectedPath])
    return(selectedPath)
    if(printing):
        print( "> Possible paths in U to choose from:")
        print(f"  {paths}")
        print( "> Selected Path:     ")
        print(f"  {selectedPath}")
        print("> Selected Path Data (pathInfo):")
        print(f"  {pathInfo}")
```

Shortcomings

Currently, there are two main problems with the algorithm provided.

Missing domination criteria

We have not implemented a domination criterium because of our understanding of the criteria derived by Sancho (1992). Using equation (6), the algorithm should set $f(j,k)$ to is minimal value for all nodes i preceding j. This leads to a potential pr ases where later actual arrival times at k fall in the departure time window to some n actual arrival times might fall outside both the departure time window to l and waiting t ath via l may eventually be faster than another route. This is the reason that we think mately not suitable to find fastest paths. For the same reasons, we discarded equation iterium.

For your concern, you can use the dominance test for the paths with the same ending arc (i,j).

Both criteria could still be implemented. To do this, we could add a new object that stores minimal feasible arrival times for a pair of nodes (i,j). These can then be compared to actual arrival time at every step of the while-loop, and paths with subopotimal arrival times can be discarded from U and P.

Shallow "backpropagation" for too early arrival times

If the algorithm encounters too early arrival time and condition 4. from above is fulfilled, it checks whether parking at the previous node can be delayed. In this example, this "shallow backpropagation" is sufficient to find the optimal path. In other cases, it might be necessary to go back further along the path to determine whether parking can be increased in any of the other preceding nodes. This could be implemented recursively; however, this in turn requires some of the conditions to be implemented as functions.

The actual algorithm

Currently, there are extensive printouts to facilitate debugging and also understand the working of the algorithm.

In [5]:

```
# Trying to put the thing into a loop
# initiate set of unprocessed paths with only the source:
U = {(1,):{"path":[1], "parking":{}, "arrival":{1:0}}}
P = {}
print("> Current unprocessed paths:")
print(U)
print("> Set of processed paths:")
print(P)
print("")

i=0
while len(U) > 0:
    i += 1
    print("iteration", i)
    print("-----------")
    # select a path q from U and delete it from U
    nextPath = U[selectNextPath()]
    print("> Selecting a path:")
    print(" ", nextPath["path"])
    print("> Path looks like this:")
    print(" ", nextPath)
    print("> Removing this path from U.")
    #Checking next nodes
    del (U[selectNextPath()])
    arrivalTime(nextPath)
    #print("> printing P")
    #for key, value in P.items():
    #    print("  Path:", key)
    #    print("    Parking times:", P[key]["parking"])
    #    print("    Arrival times:", P[key]["arrival"])
    #print("> printing U")
    #for key, value in U.items():
    #    print("  Path:", key)
    #    print("    Parking times:", U[key]["parking"])
    #    print("    Arrival times:", U[key]["arrival"])
    #print("> applying dominance criteria:")
    print("")
```

```
> Current unprocessed paths:
{(1,): {'path': [1], 'parking': {}, 'arrival': {1: 0}}}
> Set of processed paths:
{}

iteration 1
-----------
> Selecting a path:
  [1]
> Path looks like this:
  {'path': [1], 'parking': {}, 'arrival': {1: 0}}
> Removing this path from U.
> Last node of current path: 1
> Current arrival time:  0
> Possible Extensions: [2, 3, 5]
> Waiting time window at node 1: [0,inf]
  > Checking extension 2
  > Current path is:
    (1, 2)
    > Departure time window: [10,15]
    > Parking is possible and parking time is {1: [0, 10]}.
  > Checking extension 3
  > Current path is:
    (1, 3)
    > Departure time window: [10,25]
    > Parking is possible and parking time is {1: [0, 10]}.
  > Checking extension 5
  > Current path is:
    (1, 5)
    > Departure time window: [15,25]
    > Parking is possible and parking time is {1: [0, 15]}.
> Key of path to remove is:
(1,)

iteration 2
-----------
> Selecting a path:
  [1, 2]
> Path looks like this:
  {'path': [1, 2], 'parking': {1: [0, 10]}, 'arrival': {1: 0, 2: 20}}
> Removing this path from U.
> Last node of current path: 2
> Current arrival time:  20
> Possible Extensions: [4, 6, 7]
> Waiting time window at node 2: [18,30]
  > Checking extension 4
  > Current path is:
    (1, 2, 4)
    > Departure time window: [28,35]
    > Parking is possible and parking time is {1: [0, 10], 2: [20, 28]}.
  > Checking extension 6
  > Current path is:
    (1, 2, 6)
    > Departure time window: [26,34]
    > Parking is possible and parking time is {1: [0, 10], 2: [20, 26]}.
  > Checking extension 7
  > Current path is:
    (1, 2, 7)
    > Departure time window: [32,60]
    > Arriving 12 units too early.
    > Parking is not feasible (either because arrival is not in  waiting t
ime window, or
      waiting time window and departure time window do not overlap. Settin
g parking at
      end node to 0.
    > Path is: [1, 2, 7]
    > Checking whether parking time at previous node can be prolonged.
    > Previous node is 1.
    > Previous arc is (1, 2)
    > Parking time window at previous node is [0, inf]
    > Parking at previous node currently is [0, 10]
    > Limitations on extending waiting time:
      > End of waiting time window at previous node: inf
      > End of departure time window at previous node: 15
      > Necessary end of waiting time: 22
    > Setting parking time at node 1 to 15
    > Updating arrival and parking times at end node:
    > Path now looks like this: {'path': [1, 2, 7], 'parking': {1: [0, 1
5], 2: [25, 25]}, 'arrival': {1: 0, 2: 25}}
      > New arrival at end node is still outside departure time window,
        or in parking time window, but parking time window does not overla
p
        with wating time window. Discarding path.
> Key of path to remove is:
(1, 2)

iteration 3
-----------
> Selecting a path:
  [1, 3]
> Path looks like this:
  {'path': [1, 3], 'parking': {1: [0, 10]}, 'arrival': {1: 0, 3: 30}}
> Removing this path from U.
> Last node of current path: 3
> Current arrival time:  30
> Possible Extensions: [2, 4]
> Waiting time window at node 3: [35,45]
  > Checking extension 2
  > Current path is:
    (1, 3, 2)
    > Departure time window: [35,50]
    > Arriving 5 units too early.
    > Parking is not feasible (either because arrival is not in  waiting t
ime window, or
      waiting time window and departure time window do not overlap. Settin
g parking at
      end node to 0.
    > Path is: [1, 3, 2]
    > Checking whether parking time at previous node can be prolonged.
    > Previous node is 1.
    > Previous arc is (1, 3)
    > Parking time window at previous node is [0, inf]
    > Parking at previous node currently is [0, 10]
    > Limitations on extending waiting time:
      > End of waiting time window at previous node: inf
      > End of departure time window at previous node: 25
      > Necessary end of waiting time: 15
    > Setting parking time at node 1 to 15
    > Updating arrival and parking times at end node:
    > Path now looks like this: {'path': [1, 3, 2], 'parking': {1: [0, 1
5], 3: [35, 35]}, 'arrival': {1: 0, 3: 35}}
      > New arrival at end node is now in departure time window.
```

```
      > Adding path to U.
  > Checking extension 4
  > Current path is:
    (1, 3, 4)
    > Departure time window: [43,50]
    > Arriving 13 units too early.
    > Parking is not feasible (either because arrival is not in  waiting t
ime window, or
      waiting time window and departure time window do not overlap. Settin
g parking at
      end node to 0.
    > Path is: [1, 3, 4]
    > Checking whether parking time at previous node can be prolonged.
    > Previous node is 1.
    > Previous arc is (1, 3)
    > Parking time window at previous node is [0, inf]
    > Parking at previous node currently is [0, 10]
    > Limitations on extending waiting time:
      > End of waiting time window at previous node: inf
      > End of departure time window at previous node: 25
      > Necessary end of waiting time: 23
    > Setting parking time at node 1 to 23
    > Updating arrival and parking times at end node:
    > Path now looks like this: {'path': [1, 3, 4], 'parking': {1: [0, 2
3], 3: [43, 43]}, 'arrival': {1: 0, 3: 43}}
      > New arrival at end node is now in departure time window.
      > Adding path to U.
> Key of path to remove is:
(1, 3)

iteration 4
-----------
> Selecting a path:
  [1, 5]
> Path looks like this:
  {'path': [1, 5], 'parking': {1: [0, 15]}, 'arrival': {1: 0, 5: 40}}
> Removing this path from U.
> Last node of current path: 5
> Current arrival time:  40
> Possible Extensions: [2, 6]
> Waiting time window at node 5: [35,50]
  > Checking extension 2
  > Current path is:
    (1, 5, 2)
    > Departure time window: [30,80]
    > Arrival time at (5) is in the departure time window  ([30, 80]). No
parking time
      at node 5 necessary.
  > Checking extension 6
  > Current path is:
    (1, 5, 6)
    > Departure time window: [48,55]
    > Parking is possible and parking time is {1: [0, 15], 5: [40, 48]}.
> Key of path to remove is:
(1, 5)

iteration 5
-----------
> Selecting a path:
  [1, 2, 4]
> Path looks like this:
 {'path': [1, 2, 4], 'parking': {1: [0, 10], 2: [20, 28]}, 'arrival': {1:
0, 2: 20, 4: 63}}
> Removing this path from U.
> Last node of current path: 4
> Current arrival time:  63
> Possible Extensions: [7]
> Waiting time window at node 4: [64,75]
 > Checking extension 7
 > Current path is:
  (1, 2, 4, 7)
  > Departure time window: [70,99]
  > Arriving 7 units too early.
  > Parking is not feasible (either because arrival is not in  waiting t
ime window, or
   waiting time window and departure time window do not overlap. Settin
g parking at
   end node to 0.
  > Path is: [1, 2, 4, 7]
  > Checking whether parking time at previous node can be prolonged.
  > Previous node is 2.
  > Previous arc is (2, 4)
  > Parking time window at previous node is [18, 30]
  > Parking at previous node currently is [20, 28]
  > Limitations on extending waiting time:
   > End of waiting time window at previous node: 30
   > End of departure time window at previous node: 35
   > Necessary end of waiting time: 35
  > Setting parking time at node 2 to 30
  > Updating arrival and parking times at end node:
  > Path now looks like this: {'path': [1, 2, 4, 7], 'parking': {1: [0,
10], 2: [20, 30], 4: [65, 65]}, 'arrival': {1: 0, 2: 20, 4: 65}}
   > New arrival at end node is now in waiting time window, and
    waiting time window overlaps with departure time window.
   > Adding waiting time at end node:
   > Adding path to U.
> Key of path to remove is:
(1, 2, 4)

iteration 6
-----------
> Selecting a path:
 [1, 2, 6]
> Path looks like this:
 {'path': [1, 2, 6], 'parking': {1: [0, 10], 2: [20, 26]}, 'arrival': {1:
0, 2: 20, 6: 61}}
> Removing this path from U.
> Last node of current path: 6
> Current arrival time:  61
> Possible Extensions: [7]
> Waiting time window at node 6: [64,75]
 > Checking extension 7
 > Current path is:
  (1, 2, 6, 7)
  > Departure time window: [72,80]
  > Arriving 11 units too early.
  > Parking is not feasible (either because arrival is not in  waiting t
ime window, or
   waiting time window and departure time window do not overlap. Settin
g parking at
   end node to 0.
  > Path is: [1, 2, 6, 7]
  > Checking whether parking time at previous node can be prolonged.
  > Previous node is 2.
  > Previous arc is (2, 6)
  > Parking time window at previous node is [18, 30]
  > Parking at previous node currently is [20, 26]
  > Limitations on extending waiting time:
   > End of waiting time window at previous node: 30
   > End of departure time window at previous node: 34
   > Necessary end of waiting time: 37
  > Setting parking time at node 2 to 30
  > Updating arrival and parking times at end node:
  > Path now looks like this: {'path': [1, 2, 6, 7], 'parking': {1: [0, 
10], 2: [20, 30], 6: [65, 65]}, 'arrival': {1: 0, 2: 20, 6: 65}}
   > New arrival at end node is now in waiting time window, and
    waiting time window overlaps with departure time window.
   > Adding waiting time at end node:
   > Adding path to U.
> Key of path to remove is:
(1, 2, 6)

iteration 7
-----------
> Selecting a path:
 [1, 3, 2]
> Path looks like this:
 {'path': [1, 3, 2], 'parking': {1: [0, 15], 3: [35, 35]}, 'arrival': {1: 
0, 3: 35, 2: 50}}
> Removing this path from U.
> Last node of current path: 2
> Current arrival time:  50
> Possible Extensions: [4, 6, 7]
> Waiting time window at node 2: [18,30]
 > Checking extension 4
 > Current path is:
  (1, 3, 2, 4)
  > Departure time window: [28,35]
  Arrival at current end node occurs after departure time window to
   node 4 closes.
 > Checking extension 6
 > Current path is:
  (1, 3, 2, 6)
  > Departure time window: [26,34]
  Arrival at current end node occurs after departure time window to
   node 6 closes.
 > Checking extension 7
 > Current path is:
  (1, 3, 2, 7)
  > Departure time window: [32,60]
  > Arrival time at (2) is in the departure time window  ([32, 60]). No 
parking time
   at node 2 necessary.
> Key of path to remove is:
(1, 3, 2)

iteration 8
-----------
> Selecting a path:
 [1, 3, 4]
> Path looks like this:
 {'path': [1, 3, 4], 'parking': {1: [0, 23], 3: [43, 43]}, 'arrival': {1: 
0, 3: 43, 4: 63}}
```

```
> Removing this path from U.
> Last node of current path: 4
> Current arrival time:  63
> Possible Extensions: [7]
> Waiting time window at node 4: [64,75]
  > Checking extension 7
  > Current path is:
    (1, 3, 4, 7)
    > Departure time window: [70,99]
    > Arriving 7 units too early.
    > Parking is not feasible (either because arrival is not in  waiting t
ime window, or
      waiting time window and departure time window do not overlap. Settin
g parking at
      end node to 0.
    > Path is: [1, 3, 4, 7]
    > Checking whether parking time at previous node can be prolonged.
    > Previous node is 3.
    > Previous arc is (3, 4)
    > Parking time window at previous node is [35, 45]
    > Parking at previous node currently is [43, 43]
    > Limitations on extending waiting time:
      > End of waiting time window at previous node: 45
      > End of departure time window at previous node: 50
      > Necessary end of waiting time: 50
    > Setting parking time at node 3 to 45
    > Updating arrival and parking times at end node:
    > Path now looks like this: {'path': [1, 3, 4, 7], 'parking': {1: [0,
23], 3: [43, 45], 4: [65, 65]}, 'arrival': {1: 0, 3: 43, 4: 65}}
      > New arrival at end node is now in waiting time window, and
        waiting time window overlaps with departure time window.
      > Adding waiting time at end node:
      > Adding path to U.
> Key of path to remove is:
(1, 3, 4)

iteration 9
-----------
> Selecting a path:
  [1, 5, 2]
> Path looks like this:
  {'path': [1, 5, 2], 'parking': {1: [0, 15], 5: [40, 40]}, 'arrival': {1:
0, 5: 40, 2: 60}}
> Removing this path from U.
> Last node of current path: 2
> Current arrival time:  60
> Possible Extensions: [4, 6, 7]
> Waiting time window at node 2: [18,30]
  > Checking extension 4
  > Current path is:
    (1, 5, 2, 4)
    > Departure time window: [28,35]
    Arrival at current end node occurs after departure time window to
      node 4 closes.
  > Checking extension 6
  > Current path is:
    (1, 5, 2, 6)
    > Departure time window: [26,34]
    Arrival at current end node occurs after departure time window to
      node 6 closes.
  > Checking extension 7
  > Current path is:
    (1, 5, 2, 7)
    > Departure time window: [32,60]
    > Arrival time at (2) is in the departure time window  ([32, 60]). No
parking time
   at node 2 necessary.
> Key of path to remove is:
(1, 5, 2)

iteration 10
-----------
> Selecting a path:
  [1, 5, 6]
> Path looks like this:
  {'path': [1, 5, 6], 'parking': {1: [0, 15], 5: [40, 48]}, 'arrival': {1:
0, 5: 40, 6: 73}}
> Removing this path from U.
> Last node of current path: 6
> Current arrival time:  73
> Possible Extensions: [7]
> Waiting time window at node 6: [64,75]
  > Checking extension 7
  > Current path is:
    (1, 5, 6, 7)
    > Departure time window: [72,80]
    > Arrival time at (6) is in the departure time window  ([72, 80]). No
parking time
   at node 6 necessary.
> Key of path to remove is:
(1, 5, 6)

iteration 11
-----------
> Selecting a path:
  [1, 2, 4, 7]
> Path looks like this:
  {'path': [1, 2, 4, 7], 'parking': {1: [0, 10], 2: [20, 30], 4: [65, 7
0]}, 'arrival': {1: 0, 2: 20, 4: 65, 7: 100}}
> Removing this path from U.
> Last node of current path: 7
> Key of path to remove is:
(1, 2, 4, 7)

iteration 12
-----------
> Selecting a path:
  [1, 2, 6, 7]
> Path looks like this:
  {'path': [1, 2, 6, 7], 'parking': {1: [0, 10], 2: [20, 30], 6: [65, 7
2]}, 'arrival': {1: 0, 2: 20, 6: 65, 7: 82}}
> Removing this path from U.
> Last node of current path: 7
> Key of path to remove is:
(1, 2, 6, 7)

iteration 13
-----------
> Selecting a path:
  [1, 3, 2, 7]
> Path looks like this:
  {'path': [1, 3, 2, 7], 'parking': {1: [0, 15], 3: [35, 35], 2: [50, 5
0]}, 'arrival': {1: 0, 3: 35, 2: 50, 7: 80}}
> Removing this path from U.
> Last node of current path: 7
> Key of path to remove is:
(1, 3, 2, 7)

iteration 14
-----------
> Selecting a path:
  [1, 3, 4, 7]
> Path looks like this:
  {'path': [1, 3, 4, 7], 'parking': {1: [0, 23], 3: [43, 45], 4: [65, 7
0]}, 'arrival': {1: 0, 3: 43, 4: 65, 7: 100}}
> Removing this path from U.
> Last node of current path: 7
> Key of path to remove is:
(1, 3, 4, 7)

iteration 15
-----------
> Selecting a path:
  [1, 5, 2, 7]
> Path looks like this:
  {'path': [1, 5, 2, 7], 'parking': {1: [0, 15], 5: [40, 40], 2: [60, 6
0]}, 'arrival': {1: 0, 5: 40, 2: 60, 7: 90}}
> Removing this path from U.
> Last node of current path: 7
> Key of path to remove is:
(1, 5, 2, 7)

iteration 16
-----------
> Selecting a path:
  [1, 5, 6, 7]
> Path looks like this:
  {'path': [1, 5, 6, 7], 'parking': {1: [0, 15], 5: [40, 48], 6: [73, 7
3]}, 'arrival': {1: 0, 5: 40, 6: 73, 7: 83}}
> Removing this path from U.
> Last node of current path: 7
> Key of path to remove is:
(1, 5, 6, 7)
```

Checking for solutions

We can take a look at P:

In [6]:

```
P
```

Out[6]:

```
{(1,): {'path': [1], 'parking': {}, 'arrival': {1: 0}},
 (1, 2): {'path': [1, 2], 'parking': {1: [0, 10]}, 'arrival': {1: 0, 2: 2
0}},
 (1, 3): {'path': [1, 3], 'parking': {1: [0, 10]}, 'arrival': {1: 0, 3: 3
0}},
 (1, 5): {'path': [1, 5], 'parking': {1: [0, 15]}, 'arrival': {1: 0, 5: 4
0}},
 (1, 2, 4): {'path': [1, 2, 4],
  'parking': {1: [0, 10], 2: [20, 28]},
  'arrival': {1: 0, 2: 20, 4: 63}},
 (1, 2, 6): {'path': [1, 2, 6],
  'parking': {1: [0, 10], 2: [20, 26]},
  'arrival': {1: 0, 2: 20, 6: 61}},
 (1, 3, 2): {'path': [1, 3, 2],
  'parking': {1: [0, 15], 3: [35, 35]},
  'arrival': {1: 0, 3: 35, 2: 50}},
 (1, 3, 4): {'path': [1, 3, 4],
  'parking': {1: [0, 23], 3: [43, 43]},
  'arrival': {1: 0, 3: 43, 4: 63}},
 (1, 5, 2): {'path': [1, 5, 2],
  'parking': {1: [0, 15], 5: [40, 40]},
  'arrival': {1: 0, 5: 40, 2: 60}},
 (1, 5, 6): {'path': [1, 5, 6],
  'parking': {1: [0, 15], 5: [40, 48]},
  'arrival': {1: 0, 5: 40, 6: 73}},
 (1, 2, 4, 7): {'path': [1, 2, 4, 7],
  'parking': {1: [0, 10], 2: [20, 30], 4: [65, 70]},
  'arrival': {1: 0, 2: 20, 4: 65, 7: 100}},
 (1, 2, 6, 7): {'path': [1, 2, 6, 7],
  'parking': {1: [0, 10], 2: [20, 30], 6: [65, 72]},
  'arrival': {1: 0, 2: 20, 6: 65, 7: 82}},
 (1, 3, 2, 7): {'path': [1, 3, 2, 7],
  'parking': {1: [0, 15], 3: [35, 35], 2: [50, 50]},
  'arrival': {1: 0, 3: 35, 2: 50, 7: 80}},
 (1, 3, 4, 7): {'path': [1, 3, 4, 7],
  'parking': {1: [0, 23], 3: [43, 45], 4: [65, 70]},
  'arrival': {1: 0, 3: 43, 4: 65, 7: 100}},
 (1, 5, 2, 7): {'path': [1, 5, 2, 7],
  'parking': {1: [0, 15], 5: [40, 40], 2: [60, 60]},
  'arrival': {1: 0, 5: 40, 2: 60, 7: 90}},
 (1, 5, 6, 7): {'path': [1, 5, 6, 7],
  'parking': {1: [0, 15], 5: [40, 48], 6: [73, 73]},
  'arrival': {1: 0, 5: 40, 6: 73, 7: 83}}}
```

We can look at only the arrival times at the end node to determine the optimal path.

In [12]:

```
finalPaths = {key:P[key]["arrival"][7] for key, value in P.items() if key[-1] == 7}
optimalPath = [value for key, value in finalPaths.items()]
optimalTime = min(optimalPath)
optimalPath = [key for key, value in finalPaths.items() if value == optimalTime]
```

In [13]:

```
print(f"The optimal path is {optimalPath}, and its arrival time at node 7 is {optimalTime}.")
```

```
The optimal path is [(1, 3, 2, 7)], and its arrival time at node 7 is 80.
```

This solution corresponds with the solution the authors have found.