



Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

ENCS 434

Artificial Intelligence

Project 1

Single Vehicle Pickup & Delivery

Prepared By:

Name: Baraa Atta. ID: 1180445

Name: Ahmad Hamad. ID: 1180060

Instructor: Mr. Aziz Qaroush

Section: 1

Date: 2/12/2020

Table of Contents

List of Figures:	II
Introduction:	1
Constraint Satisfaction Problem (CSP):	1
The Single Vehicle Pickup and Delivery Problem with Time Windows (PDPTW):	1
Procedure:	3
Structs used in the code:	3
Task struct:	3
Node struct:	3
Files style:	4
The coordinates file:	4
The tasks file:	4
Reading from files:	4
MRV function:	6
Backtracking function:	7
Discussion ad simulation results:	9
Sample test 1:	9
Simulation results for test 1:	9
Sample test 2:	9
Simulation results for test 2:	10
Conclusion:	11

List of Figures:

Figure 1: Task Struct	3
Figure 2: Node Struct	4
Figure 3: Reading Coordinates File	4
Figure 4: First Part of Reading Tasks File	5
Figure 5: Second Part of Reading Tasks File	5
Figure 6: MRV Function Implementation	6
Figure 7: Backtracking Function Implementation	7
Figure 8: TEST 1 Simulation result	9
Figure 9: TEST 2 Simulation result	10

Introduction:

Constraint Satisfaction Problem (CSP):

Constraint satisfaction is the process of finding a solution to a set of constraints that impose conditions that the variables must satisfy. A solution is therefore a set of values for the variables that satisfies all constraints.

In the formalization process of a CSP, the problem is defined through a set of variables and a set of domains, variables can have possible values specified by the problem and constraints describe allowable combinations of values for a subset of the variables. States in a CSP defined by assignment of values to some or all variables. Solution in a CSP all variables have their own values in way that doesn't break any constraint.

The Single Vehicle Pickup and Delivery Problem with Time Windows (PDPTW):

The single vehicle PDPTW deals with a number of customer requests that must be satisfied by one vehicle with a known capacity. The route of the vehicle starts with a central depot which isn't a pickup or a delivery location. A request must be collected from a pickup location before being dropped off at a corresponding delivery location, and every pickup and delivery request is associated with a specific time window during which it must be served. If the vehicle arrives earlier than the beginning of the designated time window interval, it must wait until the service time begins. All requests must be served in a way that it doesn't violate capacity and time windows constraints.

The time window for a delivery location is the same as its pickup location and a simple location can be visited **only** once and it can be either a pickup or delivery or both observing that it can't be a pickup for more than one location and can't be a delivery for more than one location too, so in the case of being both pickup and delivery the location X is concerned with maximum of 2 other location the delivery that its pickup is at X and pickup that must deliver to X and in this case all three locations must have the same time window. In addition, for simplicity all pickups are assumed to contain the same demand which is one, when taking pickups from multiple locations without delivering previous ones the load of the car must not exceed its capacity.

Our formalization for solving this problem using CSP approach:

Variables: we assumed that variables are the locations which represented using nodes, locations are given (x, y) coordinates and the time to travel between location to other is simply the direct distance between them.

Domains: the domain for each variable (location) is the time that is available from arriving to the location and the end of its time window.

Constraints: all locations must be served without violating the capacity and the time window. For time window, travel time to the next location is taken into consideration so we need to take a path that doesn't exceed the end of the time window.

Heuristic: MRV (Minimum Remaining Values) is chosen as the heuristic which chooses the variable that might fail first due to time window & capacity constraints.

Goal State: deliver all tasks without violating time window & capacity constraints.

Procedure:

First of all, we will begin with defining two structs which will contain the details for each node and for each task.

Structs used in the code:

Task struct:

Attributes:

- 1) Picked at: which identify the time in which the task is being picked up.
- 2) Delivered at: which identify the time in which the task is being delivered.
- 3) Destination: which identify the value of the nod where the task is to be delivered.
- 4) Source: which identify the value of the nod where the task is to be picked up.
- 5) Start: which identify the starting time of the time window.
- 6) End: which identify the ending time of the time window.
- 7) Id: a unique id for each task.

```
typedef struct task {
    double pickedAt;
    double deliveredAt;
    int destination;
    int source;
    int start;
    int end;
    int id;

    task(int id, int source, int destination, int start, int end) {
        this->id = id;
        this->source = source;
        this->destination = destination;
        this->start = start;
        this->end = end;
        this->pickedAt = -1;
        this->deliveredAt = -1;
    }
} task;
```

Figure 1: Task Struct

Node struct:

Attributes:

- 1) Pickup and delivery: two Boolean values indicates if the current node is either delivery or pick up or the both.
- 2) X and Y: two double values which identify the location of the node with x and y coordinates.
- 3) Id: a unique id for each node.

- 4) Start: which identify the starting time of the time window.
- 5) End: which identify the ending time of the time window
- 6) taskD: contains the Id of the task in which the current node is the delivery node in it.
- 7) taskP: contains the Id of the task in which the current node is the pickup node in it.

```
typedef struct node {
    bool pickup, delivery;
    double x, y;
    int id;
    int start, end;
    int taskD;
    int taskP;
}
```

Figure 2: Node Struct

Files style:

The coordinates file:

The file is designed as follows:

The first line contains an integer n (the number of locations), each line of the next n lines contains x and y the x and y coordinates of the location.

The tasks file:

The file is designed as follows:

The first line contains an integer m (the maximum capacity of the vehicle), each line of the next lines contains the source and the delivery and the time window for each task.

Reading from files:

First, reading the coordinates file which is shown in **Figure 3**.

```
139 // Reading Coordinates File.
140 fscanf(coordinates, format: "%d", &locations); // First read number of locations from coordinates file.
141 vector<node> loc;
142 double x, y;
143 int id = 0; // To use as an ID for each location, after adding a location it will be incremented.
144 while (fscanf(coordinates, format: "%lf%lf", &x, &y) != EOF) {
145     loc.push_back(node(x, y, id)); // Add a node contains information (Its ID and X, Y coordinated) about read location to
146     id++; // Increment the ID for next location.
147 }
```

Figure 3: Reading Coordinates File

As shown in **Figure 3**, first we read the number of locations and store it in locations variable. Then, we create an ID variable that stores the generated id for each location starting from 0 and get into a loop that reads remaining lines from the file. Each line contains X & Y coordinates so we read them and create a node with these X, Y and current id then add it to the locations vector (loc).

Next the tasks file is read as shown in **Figure 4** which shows first part of reading this file.

```

150     fscanf(tasksFile, format: "%d", &capacity); // First read the capacity of the car from tasks file.
151     int start, end, source, destination;
152     id = 0; // Now its used to add ID's for each read task.
153     while (fscanf(tasksFile, format: "%d%d%d", &source, &destination, &start, &end) != EOF) {
154         tasks.push_back(task(id, source, destination, start, end)); // Add each task to a vector of tasks.
155         loc[source].pickup = true; // Mark the pickup location in loc vector of the task as pickup.
156         loc[destination].delivery = true; // Mark the delivery location in loc vector of the task as delivery.
157         // -----
158         // Specify the start and end times for the source and delivery locations of the read task
159         loc[source].setStart(start);
160         loc[destination].setStart(start);
161         loc[source].setEnd(end);
162         loc[destination].setEnd(end);
163         //-----

```

Figure 4: First Part of Reading Tasks File

At first, the capacity of the car is read then the ID is reset to zero to generate ID for each task. Next, each line of the line is read which contains the source and destination locations and the start and end times of the time windows for each task and each created task is added to tasks vector.

Then, for each task the source is marked up as a pickup and the destination is marked up as a delivery location in loc vector and the time window (Start & End times) is assigned to the source and destination locations. The rest of the code for reading this file is shown **Figure 5**.

```

164         // Specify in which task each location is involved so we can easily know the location is pickup or delivery for which
165         loc[source].setTaskP(id);
166         loc[destination].setTaskD(id);
167         //-----
168         haveDemand[destination] = source; // Map the source for each destination in each task so we can know from where does t
169         //-----
170         // Mark the source and destination for each task as must be visited.
171         toVisit.insert(source);
172         toVisit.insert(destination);
173         //-----
174         id++; // Increment ID for next task.
175     }

```

Figure 5: Second Part of Reading Tasks File

Now, to easily know each location is a pickup or a delivery in which task the ID of the task is set for both the source and destination. Then, the destination is mapped to its source in the **haveDemand** map so it's used to know the pickup for each destination. next, the source and destination are marked up as must be visited in the **toVisit** set. Finally, ID is incremented to be used for next task.

MRV function:

As the MRV algorithm states that we should choose the next variable depending on the having the least number of values.

In our problem the variable that has the least number of the values is the one that the travel time to it plus the current time is the nearest to the end time of its time window.

So now we need to sort all the unvisited children of the location depending on the difference between its end time and the time after reaching it.

```
void MRV(int current, vector<bool> vis, vector<node> nodes, queue<pair<int, int>> &q) {
    multiset<pair<int, int>> ml;
    for (int i = 0; i < nodes.size(); i++) {
        if (vis[i] || i == current)
            continue;
        double distance = sqrt( (nodes[i].x - nodes[current].x) * (nodes[i].x - nodes[current].x) + (nodes[i].y - nodes[current].y) * (nodes[i].y - nodes[current].y));
        double timeConstraint = nodes[i].end - (distance + currentTime);
        if (timeConstraint >= 0) {
            if ((nodes[i].pickup && !nodes[i].delivery && content == capacity) ||
                (nodes[i].delivery && !vis[haveDemand[i]]))
                continue;
            ml.insert({timeConstraint, i});
        }
    }
    for (auto i : ml) {
        q.push({ sqrt( (nodes[i.second].x - nodes[current].x) * (nodes[i.second].x - nodes[current].x) + (nodes[i.second].y - nodes[current].y) * (nodes[i.second].y - nodes[current].y)),
                i.second});
    }
}
```

Figure 6: MRV Function Implementation

As shown in the previous code, first of all a multiset is created because it keeps it self – sorted after each insertion. Secondly, we will loop through all the children of the current node with skipping the locations which are already visited or if the location is a delivery location and its demand has not taken yet. Now the remaining locations are the ones we can move to, then all the remaining locations will be inserted into the multi set as a pair with the difference between their end time and the theoretical time when reaching it, after looping through them all the multi set now has all the locations with their time difference sorted ascending, but the problem now that we do not need this difference but we need the actual time needed to travel. To bypass this problem, we will iterate over those pairs of locations again from the multiset and insert them again to a queue as a pair containing the id of the location and the required time to reach it from the current node.

Notes:

- We cannot depend on the required time to reach a node to determine which node we should visit next because the travel time might be very large but also the time window is very large hence this is not a good estimate.
- A multiset containing pairs is always sorted ascending depending on the first value of the pair, for that we put the time difference in the beginning of the pair.
- We used a queue due to its property FIFO, since we are inserting the locations having the highest priority first and then popping them one by one from the front, so this is the best data structure that fits with this application.

Backtracking function:

The concept behind the back-tracking function is very simple, the only thing we do is to try to move to every available location from the one we are now in the order found by the MRV as was explained before.

```
99
100 bool backTrack(int v, vector<bool> &vis, vector<node> nodes, vector<task> &tasks, double prevTime) {
101     vis[v] = true;
102     queue<pair<int, int>> q;
103     MRV(v, vis, nodes, &q);
104     while (!q.empty()) {
105         vis[(q.front()).second] = true;
106         double prevTime = currentTime;
107         currentTime + q.front().first >= (double) nodes[q.front().second].start ? currentTime += q.front().first
108                                                     : currentTime = (double) nodes[q.front().second].start;
109         if (nodes[(q.front()).second].pickup) {
110             content++;
111             tasks[nodes[(q.front()).second].taskP].pickedAt = currentTime;
112         }
113         if (nodes[(q.front()).second].delivery) {
114             content--;
115             tasks[nodes[(q.front()).second].taskD].deliveredAt = currentTime;
116         }
117         backTrack((q.front()).second, &vis, nodes, &tasks, prevTime);
118         if (done(vis))
119             return true;
120         q.pop();
121     }
122     if (done(vis)) {
123         cout<<"V: "<<v<<endl;
124         return true;
125     }
126     currentTime = prevTime;
127     if (nodes[v].pickup)
128         content--;
129     else if (nodes[v].delivery)
130         content++;
131     vis[v] = false;
132     return false;
}
```

Figure 7: Backtracking Function Implementation

As shown in the previous code, first of all the MRV function is invoked and the results got from it is saved into a queue called q, after this we will start looping through the locations in the queue (which are sorted already by the MRV function), for every location we will call the

back track function again by marking the node as visited and taking its delivery if it is a pick up location or delivering its delivery if it is a delivery location, like this the function will keep calling itself for every location in the queue and whenever we reach a dead end we will return the time to the previous time before calling the function at this level , also the location will be remarked as not visited more than that if the location is a pick up one the demand is returned to it including if it is a delivery location the demand is returned from it and then we will try the other locations on the previous level.

Every call for the function we call a function called “done” that will check if every location in the tasks is visited, if so then all the tasks are completed successfully, otherwise the function will keep trying by moving to the next location in every level. At the end if it returns to the initial location after expanding all its children it will return false indicating that the problem has no solution under this constraints.

Discussion ad simulation results:

Sample test 1:

We will begin with a simple test that does not require any back tracking.

Coordinates file content:

```
4
0 0
0 10
0 20
0 30
```

The tasks file is as follows:

```
1
1 2 0 20
2 3 0 30
```

Simulation results for test 1:

```
Task : 0, Delivered from 1 to 2, Picked up at : 10, Delivered at : 20
```

```
Task : 1, Delivered from 2 to 3, Picked up at : 20, Delivered at : 30
```

```
Process finished with exit code 0
```

Figure 8: TEST 1 Simulation result

As we can see the two tasks were done successfully during their available time window and without any conflict between them.

Sample test 2:

This test has no solution so we try it to see if the program will backtrack to try all paths correctly and tell us that there is no solution.

Coordinates file content:

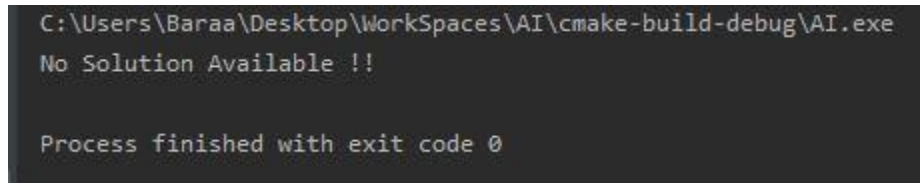
```
8
0 0
0 10
0 20
0 30
0 40
0 50
15 40
0 100
```

The tasks file is as follows:

```
4
1 2 0 20
3 4 10 50
6 7 10 66
4 5 10 50
```

In this test the third task which is delivering from 6 to 7 is impossible during the given time window so the program must print that there is no solution.

Simulation results for test 2:



```
C:\Users\Baraa\Desktop\WorkSpaces\AI\cmake-build-debug\AI.exe
No Solution Available !!

Process finished with exit code 0
```

Figure 9: TEST 2 Simulation result

As we shown from the simulation the program couldn't find any solution for this test.

Conclusion:

At the end we can say that the project was a very interesting one because it takes us from the theoretical part of the algorithms to a real-life application requiring algorithms like this to arrange the tasks required from it.

Also, we recognized how the algorithms used like the MRV reduces a lot of backtrackings that could happen without it; however, those algorithms are not sufficient to solve like this problem in an optimal way when the state space is very large.