In [1]:
```
# -*- coding: utf-8 -*-
print("""
Created on Sun Oct  7 12:32:15 2018

@author: Andrew

"In an ISRP, inventory slack is defined as the duration between reliefs
arriving time and estimated inventory stock-out time."
https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0198443
""")
```

```
Created on Sun Oct  7 12:32:15 2018

@author: Andrew

"In an ISRP, inventory slack is defined as the duration between reliefs
arriving time and estimated inventory stock-out time."
https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0198443
```

In [2]:
```
import numpy as np
import pandas as pd
from ortools.constraint_solver import pywrapcp
from ortools.constraint_solver import routing_enums_pb2
import matplotlib.pyplot as plt
import matplotlib as mpl
import matplotlib.patheffects
%matplotlib inline
```

In [3]:
```
# user-defined parameters
np.random.seed(44444)
num_locations = 15
depot = 0
num_trucks = 3
```

In [4]:
```
def create_params(n):
    from numpy.random import randint
    locations = [[randint(0, n*2), randint(0, n*2)] for i in range(n - 1)]
    demands = randint(1, 30, n - 1).tolist()
    demands.insert(0, 0)
    start_times = randint(4800, 80000, n).tolist()
    return locations, demands, start_times
def find_center_location(locations):
    x, y = [list(i) for i in zip(*locations)]
    center_location = [int(round((max(x) + min(x)) / 2, 0)),
                       int(round((max(y) + min(y)) / 2, 0))]
    return center_location
```

In [5]:
```
locations, demands, start_times = create_params(num_locations)
center_location = find_center_location(locations)
locations.insert(0, center_location)
routing = pywrapcp.RoutingModel(num_locations, num_trucks, depot)
search_parameters = pywrapcp.RoutingModel.DefaultSearchParameters()
```

In [6]:
```
def plot_locations(locations, center_location):
    data = locations
    data = pd.DataFrame(data = data, columns = ['X', 'Y'])
    data['X'] = data['X'].astype(int)
    data['Y'] = data['Y'].astype(int)
    fig, ax = plt.subplots()
    fig.set_size_inches(15, 10)
    # Center the graph at center_location
    ax.set_title('Locations', fontsize = 16)
    ax.scatter(data.X, data.Y, s = 80)
    ax.spines['left'].set_position('center')
    ax.spines['right'].set_color('none')
    ax.spines['bottom'].set_position('center')
    ax.spines['top'].set_color('none')
    ax.spines['left'].set_smart_bounds(True)
    ax.spines['bottom'].set_smart_bounds(True)
    ax.xaxis.set_ticks_position('bottom')
    ax.yaxis.set_ticks_position('left')
    for axis, center in zip([ax.xaxis, ax.yaxis],
                            [center_location[0], center_location[1]]):
        # Turn on minor and major gridlines and ticks
        axis.set_ticks_position('both')
        axis.grid(True, 'major', ls='solid', lw=0.5, color='gray')
        axis.grid(True, 'minor', ls='solid', lw=0.1, color='gray')
    for xy in zip(data.X, data.Y):
        x, y = xy
        ax.annotate('(%s, %s)' % (str(x), str(y)),
                    xy = xy, fontsize = 12,
                    xytext=(-20, 0),
                    textcoords = 'offset points',
                    ha = 'right', va = 'bottom',
                    bbox = dict(boxstyle = 'round,pad=0.5',
                              fc = 'yellow', alpha = 0.5),
                    arrowprops=dict(arrowstyle = '->',
                                    connectionstyle='arc3,rad=0')
                   )
    ax.set_yticklabels([])
    ax.set_xticklabels([])
    plt.tight_layout()
    plt.show()
```

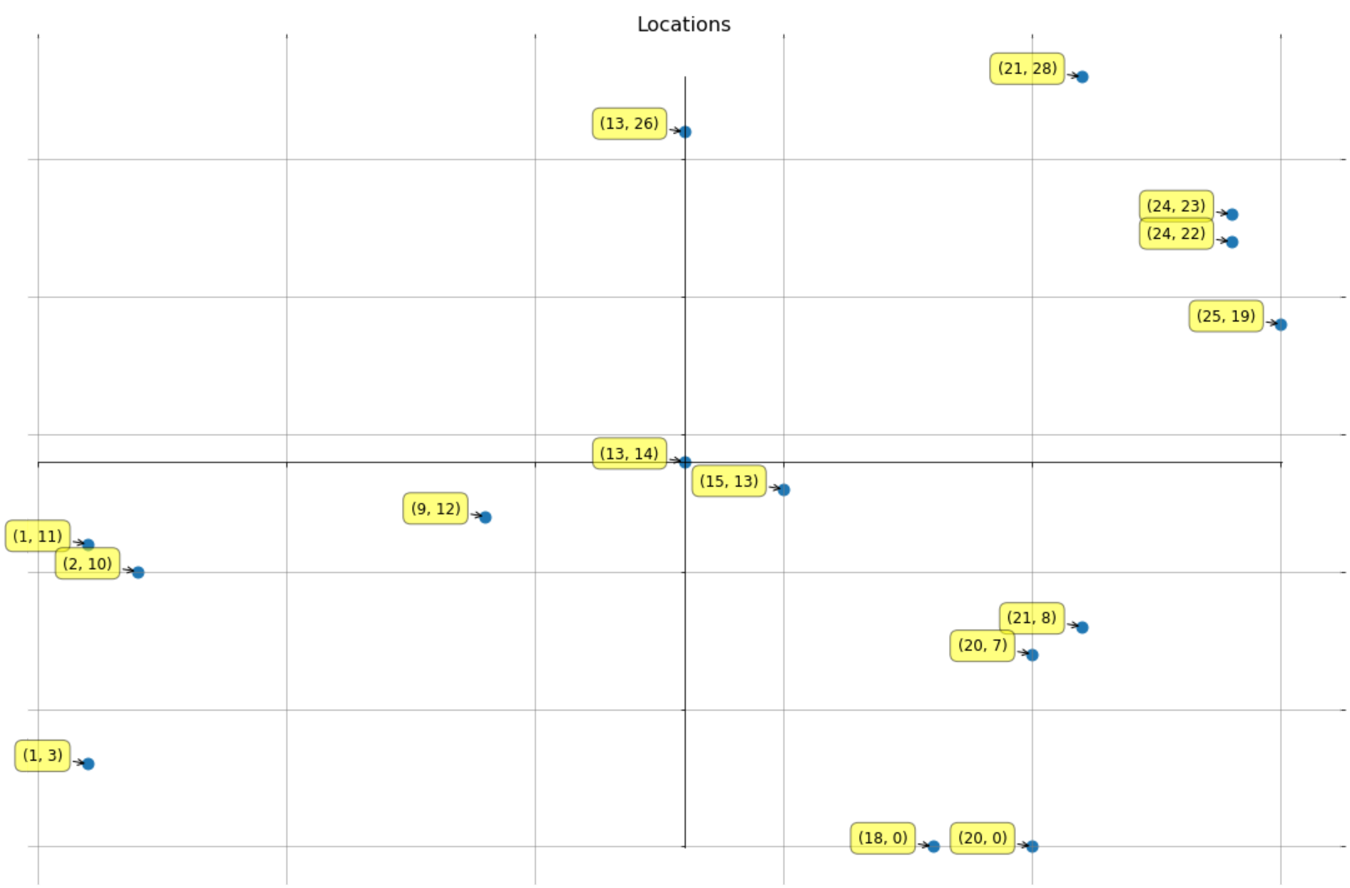
In [7]:

```
nodes = ['Node ' + str(i) for i in range(len(locations))]
locations_arr = np.array(locations)
x = locations_arr[:, 0].tolist()
y = locations_arr[:, 1].tolist()
params_df = pd.DataFrame({'Nodes': nodes, 'Locations_X': x,
                          'Locations_Y': y, 'Demands': demands,
                          'Start Times': [str(int(round(t / 60, 0))) +\
                                          ' min' for t in start_times]})
print(params_df.set_index('Nodes'))
```

```
         Locations_X  Locations_Y  Demands Start Times
Nodes
Node 0            13           14        0     452 min
Node 1            13           26       24     617 min
Node 2             1           11        8     849 min
Node 3             9           12       10     649 min
Node 4            24           23       26     547 min
Node 5            21            8        2     619 min
Node 6             2           10       28     780 min
Node 7            18            0       21    1218 min
Node 8            20            7       23     542 min
Node 9            21           28       10     660 min
Node 10           20            0       19     587 min
Node 11           15           13       23     231 min
Node 12           25           19        2     231 min
Node 13           24           22       15     665 min
Node 14            1            3        5     294 min
```

In [8]:

```
plot_locations(locations, center_location)
```



In [9]:
```
def manhattan_distance(x1, y1, x2, y2):
    dist = abs(x1 - x2) + abs(y1 - y2)
    return dist

class DistancesBetweenLocations(object):

    def __init__(self, locations):

        num_locations = len(locations)
        self.distances = {}

        for from_node in range(num_locations):
            self.distances[from_node] = {}
            for to_node in range(num_locations):
                x1 = locations[from_node][0]
                y1 = locations[from_node][1]
                x2 = locations[to_node][0]
                y2 = locations[to_node][1]
                self.distances[from_node][to_node] = manhattan_distance(x1, y1, x2, y2)

    def distances_between_locations(self, from_location, to_location):
        return int(self.distances[from_location][to_location])
```

In [10]:
```
dbl = DistancesBetweenLocations(locations)
distances_between_locations = dbl.distances_between_locations
routing.SetArcCostEvaluatorOfAllVehicles(distances_between_locations)
```

In [11]:
```
class DemandsAtLocations(object):

    def __init__(self, demands):
        self.demands = demands

    def demands_at_locations(self, from_location, to_location):
        del(to_location)
        return self.demands[from_location]
```

In [12]:
```
# capacity dimension constraints
truck_capacity = 100
capacity_slack = 0
dal = DemandsAtLocations(demands)
demands_at_locations = dal.demands_at_locations
routing.AddDimension(demands_at_locations, capacity_slack,
                     truck_capacity, True, 'Capacity')
```

Out[12]:
```
True
```

In [13]:
```
class ServiceTimePerUnit(object):
    """service time — how long it takes to make a delivery
                  or provide a service at each location"""
    def __init__(self, demands, time_per_unit):
        self.demands = demands
        self.time_per_unit = time_per_unit

    def service_times(self, from_location, to_location):
        return self.demands[from_location] * self.time_per_unit
```

In [14]:
```
# time dimension constraints
time_per_unit = 300 # 5 min / unit
upper_bound = 24 * 3600 # convert 24 hours to seconds
time_window = 5 * 3600 # convert 5 hours to seconds
speed = 10 # 10 meters / second
stpu = ServiceTimePerUnit(demands, time_per_unit)
service_times = stpu.service_times
```

In [15]:
```
class TotalTripTime(object):
    def __init__(self, service_times, distances_between_locations, speed):
        self.service_times = service_times
        self.distances_between_locations = distances_between_locations
        self.speed = speed

    def total_times(self, from_location, to_location):
        total = self.service_times(from_location, to_location) +\
                self.distances_between_locations(from_location,
                                                 to_location) / self.speed
        return total
```

In [16]:
```
ttt = TotalTripTime(service_times, distances_between_locations, speed)
total_times = ttt.total_times
routing.AddDimension(total_times, upper_bound, upper_bound, False, 'Time')
time_dimension = routing.GetDimensionOrDie('Time')
for i in range(1, num_locations):
    start_time = start_times[i]
    time_dimension.CumulVar(routing.NodeToIndex(i)).SetRange(start_time, start_time + ti
me_window)
```

In [17]:
```
truck_load_time = 300 # 5 min / unit
truck_unload_time = 300 # 5 min / unit
solver = routing.solver()
intervals = []
for num in range(num_trucks):
    start_interval = solver.FixedDurationIntervalVar(
        routing.CumulVar(routing.Start(num),'Time'),
        truck_load_time, 'depot_interval')
    end_interval = solver.FixedDurationIntervalVar(
        routing.CumulVar(routing.End(num),'Time'),
        truck_unload_time, 'depot_interval')
    intervals.append(start_interval)
    intervals.append(end_interval)
```

In [18]:
```
depot_capacity = 2 # max loading capacity at the depot
depot_usage = [1 for i in range(num_trucks * 2)]
solver.AddConstraint(solver.Cumulative(intervals, depot_usage, depot_capacity, 'depot'))
for num in range(num_trucks):
    routing.AddVariableMinimizedByFinalizer(routing.CumulVar(routing.End(num), 'Time'))
    routing.AddVariableMinimizedByFinalizer(routing.CumulVar(routing.Start(num), 'Time'
))
assignment = routing.SolveWithParameters(search_parameters)
```

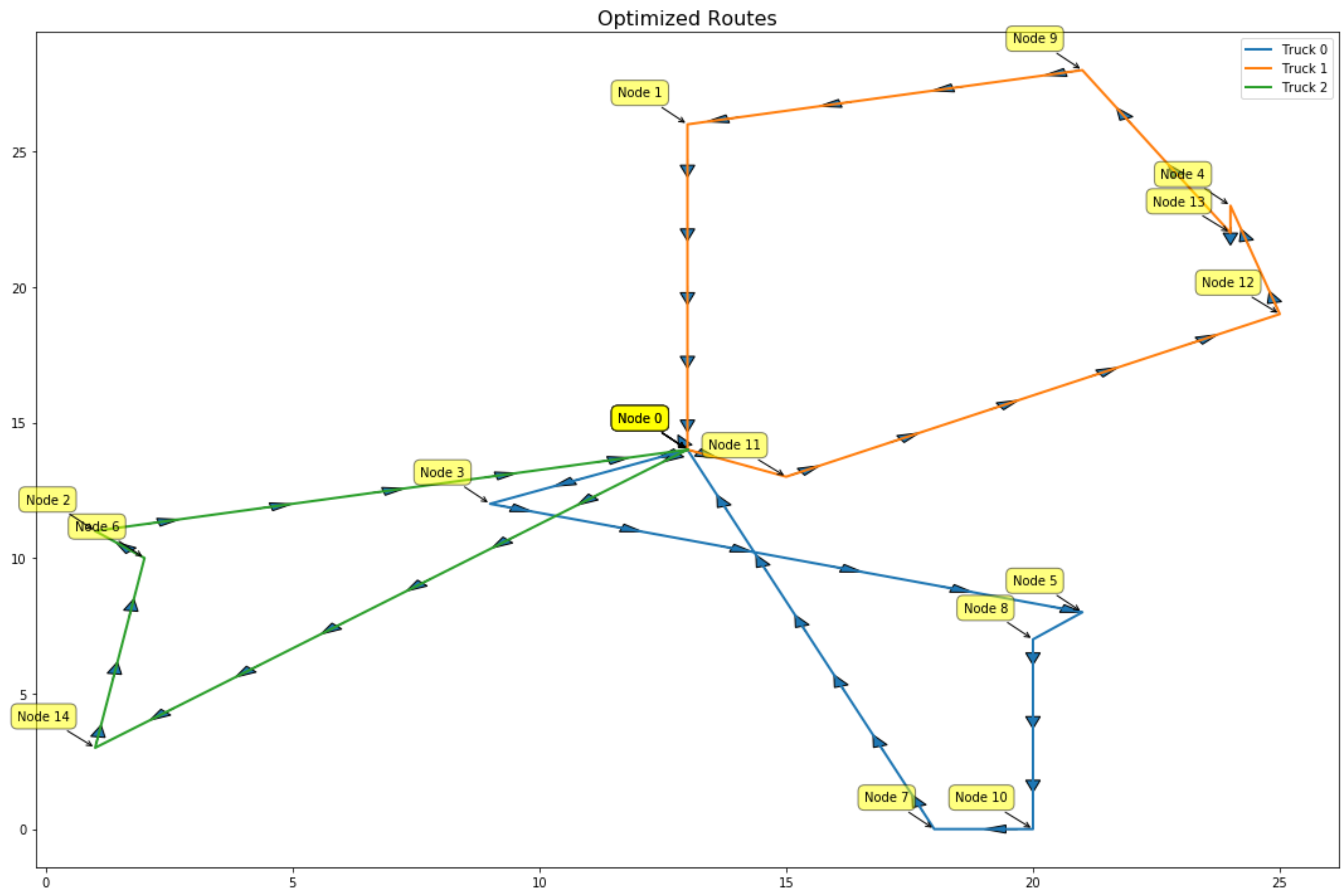
In [19]:

```
def add_truck_route_data(node_idx, num_truck, truck_route_data):
    x, y = locations[node_idx]
    truck_route_data['Truck'].append('Truck %s' % str(num_truck))
    truck_route_data['Location'].append('Node %s' % str(node_idx))
    truck_route_data['X'].append(x)
    truck_route_data['Y'].append(y)
    return truck_route_data

if assignment:
    print('Optimal Routes for each Truck:\n')
    capacity_dimension = routing.GetDimensionOrDie('Capacity')
    time_dimension = routing.GetDimensionOrDie('Time')
    truck_route_data = {'Truck':[], 'Location': [], 'X':[], 'Y':[]}
    for num_truck in range(num_trucks):
        index = routing.Start(int(num_truck))
        plan_output = 'Truck {0} Route:\n'.format(num_truck)
        while not routing.IsEnd(index):
            node_index = routing.IndexToNode(index)
            truck_route_data = add_truck_route_data(node_index, num_truck, truck_route_d
ata)
            load_var = capacity_dimension.CumulVar(index)
            time_var = time_dimension.CumulVar(index)
            plan_output += 'Node({node_index}) Load({load}) Time({tmin} min, {tmax} min)
 ->\n'.format(
                node_index = node_index, load = assignment.Value(load_var),
                tmin = str(int(round(assignment.Min(time_var) / 60, 0))),
                tmax = str(int(round(assignment.Max(time_var) / 60, 0)))
            )
            index = assignment.Value(routing.NextVar(index))
        node_index = routing.IndexToNode(index)
        truck_route_data = add_truck_route_data(node_index, num_truck, truck_route_data)
        load_var = capacity_dimension.CumulVar(index)
        time_var = time_dimension.CumulVar(index)
        plan_output += "Node({node_index}) Load({load}) Time({tmin} min, {tmax} min)".fo
rmat(
                      node_index=node_index,
                      load=assignment.Value(load_var),
                      tmin = str(int(round(assignment.Min(time_var) / 60, 0))),
                      tmax = str(int(round(assignment.Max(time_var) / 60, 0)))
        )
        print(plan_output)
        tot_route_time_str = str(int(round(assignment.Max(time_var) / 60, 0)))
        print('Total Route Time:  %s minutes'  % tot_route_time_str)
        print()

    print("Total distance of all routes: %s\n" % str(assignment.ObjectiveValue()))
else:
    print('No solution found.')
```

```
Optimal Routes for each Truck:

Truck 0 Route:
Node(0) Load(0) Time(0 min, 0 min) ->
Node(3) Load(0) Time(649 min, 712 min) ->
Node(5) Load(10) Time(699 min, 762 min) ->
Node(8) Load(12) Time(709 min, 772 min) ->
Node(10) Load(35) Time(824 min, 887 min) ->
Node(7) Load(54) Time(1218 min, 1218 min) ->
Node(0) Load(75) Time(1323 min, 1323 min)
Total Route Time:  1323 minutes

Truck 1 Route:
Node(0) Load(0) Time(0 min, 0 min) ->
Node(11) Load(0) Time(231 min, 416 min) ->
Node(12) Load(23) Time(346 min, 531 min) ->
Node(4) Load(25) Time(547 min, 547 min) ->
Node(13) Load(51) Time(677 min, 677 min) ->
Node(9) Load(66) Time(752 min, 752 min) ->
Node(1) Load(76) Time(802 min, 802 min) ->
Node(0) Load(100) Time(922 min, 922 min)
Total Route Time:  922 minutes

Truck 2 Route:
Node(0) Load(0) Time(5 min, 5 min) ->
Node(14) Load(0) Time(294 min, 594 min) ->
Node(6) Load(5) Time(780 min, 780 min) ->
Node(2) Load(33) Time(920 min, 920 min) ->
Node(0) Load(41) Time(960 min, 960 min)
Total Route Time:  960 minutes

Total distance of all routes: 156
```

In [20]:

```
truck_route_df = pd.DataFrame(truck_route_data)
groups = truck_route_df.groupby('Truck')
fig, axes = plt.subplots()
fig.set_size_inches(15,10)
for name, group in groups:
    scale = 23.5
    x_vals = group.X.values
    y_vals = group.Y.values
    aspace = .1
    aspace *= scale
    span_points = [0]
    for i in range(1,len(x_vals)):
        dx = x_vals[i] - x_vals[i-1]
        dy = y_vals[i] - y_vals[i-1]
        span_points.append(np.sqrt(dx * dx + dy * dy))
    span_points = np.array(span_points)
    span_cum_sum = []
    for i in range(len(span_points)):
        span_cum_sum.append(span_points[0:i].sum())
    span_cum_sum.append(span_points.sum())
    arrow_data = []
    arrow_pos = 0
    span_count = 1
    while arrow_pos < span_points.sum():
        x1, x2 = x_vals[span_count - 1], x_vals[span_count]
        y1, y2 = y_vals[span_count - 1], y_vals[span_count]
        da = arrow_pos - span_cum_sum[span_count]
        theta = np.arctan2((x2 - x1),(y2 - y1))
        ax = np.sin(theta) * da + x1
        ay = np.cos(theta) * da + y1
        arrow_data.append((ax, ay, theta))
        arrow_pos += aspace
        while arrow_pos > span_cum_sum[span_count+1]:
            span_count += 1
            if arrow_pos > span_cum_sum[-1]:
                break
    for ax,ay,theta in arrow_data:
        axes.arrow(ax, ay, np.sin(theta) * aspace / 10,
                   np.cos(theta) * aspace / 10,
                   head_width = aspace / 8)
    axes.plot(x_vals, y_vals, linestyle='-',
              ms = 12, label = name, linewidth = 2)
    labels = group.Location.values.tolist()
    for label, x, y in zip(labels, x_vals, y_vals):
        plt.annotate(label, xy = (x, y), xytext = (-20, 20),
            textcoords = 'offset points',
            ha = 'right', va = 'bottom',
            bbox = dict(boxstyle = 'round,pad=0.5',
                        fc = 'yellow', alpha=0.5),
            arrowprops=dict(arrowstyle = '->',
                            connectionstyle = 'arc3,rad=0'))
axes.legend()
axes.set_title('Optimized Routes', fontsize=16)
plt.tight_layout()
plt.show()
```


Optimized Routes
Truck 0
Truck 1
Truck 2
Node 0
Node 1
Node 2
Node 3
Node 4
Node 5
Node 6
Node 7
Node 8
Node 9
Node 10
Node 11
Node 12
Node 13
Node 14
0
5
10
15
20
25