Part 1: Pseudo-code of operators

[Mutation]

```
SwappedMutation (parent, child)
{
int subLength; // swapped part length
int pos1; // part 1 start index
int pos2; // part 2 start index

for (int i = 0; i < numberOfGenes; i++)
{
    if (i < pos1) // i before part 1
    {
        // Copy parent to child;
    }else if (i >= pos1 && i < pos1 + subLength) // i is in part 1
    {
        // Copy parent's part2 to child;
    }else if (i >= pos1 + subLength && i < pos2) // i is between part 1 and part 2
    {
        // Copy parent to child;
    }else if (i >= pos2 && i < pos2 + subLength) // I is in part 2
    {
        // Copy parent's part1 to child;
    }
    else
    {
        // Copy parent to child;
    }
}
}
```

```
ReciprocalMutation (parent, child)
{
    int pos1; // first element
    int pos2; // second element
    // make sure pos1 is less then pos2
    if (pos1 > pos2)
    {
      // Exchange pos1 & pos2;
    }
for (int i = 0; i < numberOfGenes; i++)
{
    if (i < pos1)
    {
        // copy parent to child
    }else if (i == pos1)
    {
        // copy parent’s pos2 to child
    }else if (i > pos1 && i < pos2)
    {
        // copy parent to child
    }else if (i == pos2)
    {
        // copy parent’s pos1 to child
    }
    else
    {
        // copy parent to child
    }
}
}
```

```
DisplacementMutation ( parent, child)
{
    int pos1; // cut-part start position
    int pos2; // cut-part end position
    // make sure pos1 is less then pos2
    if (pos1 > pos2)
    {
        // exchange pos1 & pos2
    }
    int subLength = pos2 - pos1;
    // determine insert position
    int insertPos = randomizer.Next(numberOfGenes - subLength);
    int minPos = Math.Min(pos1, insertPos);
    for (int i = 0; i < numberOfGenes; i++)
    {
      if (minPos == pos1)
      {
        if(i < pos1)
        {
            // copy parent to child
        }else if (i >= pos1 && i < insertPos)
        {
            // copy parent to child (except cut-part)
        }else if (i >= insertPos && i < insertPos + subLength)
        {
            // copy cut-part to child
        }
        else
        {
             // copy parent to child
        }
      }
      else
      {
        if (i < insertPos)
        {
            // copy parent to child
        }else if (i >= insertPos && i < insertPos + subLength)
```

```
         {
             // copy cut-part to child
         }else if (i >= insertPos + subLength && i < pos2)
         {
             // copy parent to child (except cut-part)
         }
         else
         {
             // copy parent to child
         }
        }
}
}
```

```
InsertionMutation ( parent, child)
{
    int pos1; // select a cut-part
    int pos2; // select a pos to insert cut-part
    for (int i = 0; i < numberOfGenes; i++)
    {
        if (i < pos1)
        {
              // copy parent to child
        }else if (i > pos2)
        {
              // copy parent to child
        }else if (i == pos1)
        {
              // copy parent to child's pos2
        }else
        {
              // copy parent to child orderly
        }
    }
}
```

```
InversionMutation ( parent, child)
{
    int pos1; // inverse-part start position
    int pos2; // inverse-part end position
    // make sure pos1 is less then pos2
    if (pos1 > pos2)
    {
        // exchange pos1 & pos2
    }
    for (int i = 0; i < numberOfGenes; i++)
    {
        // copy parent to child
    }

    for (int i = pos1; i < pos2; i++)
    {
        // fill in number inversely into child (overwrite original one)
    }
}
```

[Crossover]

```
PartialMappedCrossover (father, mother, child1, child2)
{
    int pos1 = randomizer.Next(numberOfGenes);
    int pos2 = randomizer.Next(numberOfGenes);
    // make sure pos1 is less then pos2
    if (pos1 > pos2)
    {
       // exchange pos1 & pos2
    }
    // prepare proto-child
    for (int i = 0; i < numberOfGenes; i++)
    {
       if (i >= pos1 && i < pos2)
       {
          temp1[i] = chromosomes[motherIdx][i];
          temp2[i] = chromosomes[fatherIdx][i];
       }
       else
       {
           temp1[i] = chromosomes[fatherIdx][i];
           temp2[i] = chromosomes[motherIdx][i];
       }
     }
     // prepare map
     int cutLength = pos2 - pos1;
     int[][] map = new int[cutLength][];
     for (int i = 0; i < cutLength; i++)
     {
          map[i] = new int[2];
          for (int j = 0; j < 2; j++)
          {
              // initialize map member
          }
       }
       for (int i = 0; i < cutLength; i++)
       {
```

```
   // fill in map
}
// rearrange map
for (int i = 0; i < cutLength; i++)
{
   for (int j = i+1; j < cutLength; j++)
   {
     if (map[i][0] == map[j][1])
     {
       map[j][1] = map[i][1];
       map[i][0] = map[i][1] = -1; // erase useless(repeated) numbers
     }
     if (map[i][1] == map[j][0])
     {
       map[j][0] = map[i][0];
       map[i][0] = map[i][1] = -1; // erase useless(repeated) numbers
     }
    }
  }
  // prepare child
  for (int i = 0; i < numberOfGenes; i++)
  {
    if (i >= pos1 && i < pos2)
    {
       // copy cut-part to childs
    }
    else
    {
      // prepare child1
      if (IsContain(temp1[i], map))
      {
        for (int j = 0; j < cutLength; j++)
        {
          if (temp1[i] == map[j][0])
          {
            chromosomes[child1Idx][i] = map[j][1]; // copy from map
            break;
          }
```

```
                  }
                }
                else
                {
                      // copy from father
                }
                // prepare child2
                if (IsContain(temp2[i], map))
                {
                    for (int j = 0; j < cutLength; j++)
                    {
                          if (temp2[i] == map[j][1])
                          {
                              chromosomes[child2Idx][i] = map[j][0]; // copy from map
                              break;
                          }
                      }
                  }
                  else
                  {
                      // copy from mother
                  }
              }
        }
}
```

```
OrderCrossover (father, mother, child1, child2)
{
    int pos1 = randomizer.Next(numberOfGenes);
    int pos2 = randomizer.Next(numberOfGenes);
    // make sure pos1 is less then pos2
    if (pos1 > pos2)
    {
        // exchange pos1 & pos2
    }
    // copy parent to proto-child
    for (int i = 0; i < numberOfGenes; i++)
    {
        temp1[i] = chromosomes[fatherIdx][i];
        temp2[i] = chromosomes[motherIdx][i];
    }
    // tag father's gene position with respect to mother's gene (by setting num = -1)
    for (int i = 0; i < numberOfGenes; i++)
    {
       if (i >= pos1 && i < pos2)
       {
          for (int j = 0; j < numberOfGenes; j++)
          {
              if (temp1[i] == temp2[j]) temp2[j] = -1;
          }
        }
     }
     // prepare child 1
     for (int i = 0; i < numberOfGenes; i++)
     {
        if (i < pos1)
        {
            for (int j = 0; j < numberOfGenes; j++)
            {
                if (temp2[j] != -1)
                {
                   chromosomes[child1Idx][i] = temp2[j]; // copy to child 1
                   temp2[j] = -1; // turn off this number
                   break;
```

```
             }
           }
         }else if (i >= pos1 && i < pos2)
         {
           chromosomes[child1Idx][i] = temp1[i];
          }else
         {
           for (int j = 0; j < numberOfGenes; j++)
           {
              if (temp2[j] != -1)
              {
                chromosomes[child1Idx][i] = temp2[j];
                temp2[j] = -1;
                break;
              }
            }
          }
        }
        // copy parent to proto-child again
        for (int i = 0; i < numberOfGenes; i++)
        {
           temp1[i] = chromosomes[fatherIdx][i];
           temp2[i] = chromosomes[motherIdx][i];
        }
        // tag mother's gene position with respect to father's gene (by setting num
= -1)
        for (int i = 0; i < numberOfGenes; i++)
        {
          if (i >= pos1 && i < pos2)
          {
             for (int j = 0; j < numberOfGenes; j++)
             {
               if (temp2[i] == temp1[j]) temp1[j] = -1;
             }
           }
         }
          // prepare child 2
          for (int i = 0; i < numberOfGenes; i++)
```

```
    {
        if (i < pos1)
        {
            for (int j = 0; j < numberOfGenes; j++)
            {
                if (temp1[j] != -1)
                {
                    chromosomes[child2Idx][i] = temp1[j]; // copy to child 2
                    temp1[j] = -1; // turn off this number
                    break;
                }
            }
        }
        else if (i >= pos1 && i < pos2)
        {
            chromosomes[child2Idx][i] = temp2[i];
        }
        else
        {
            for (int j = 0; j < numberOfGenes; j++)
            {
                if (temp1[j] != -1)
                {
                chromosomes[child2Idx][i] = temp1[j];
                temp1[j] = -1;
                break;
                }
              }
          }
      }
}
```

```
PositionBasedCrossover (father, mother, child1, child2)
{
      // copy parent to proto-child
      for (int i = 0; i < numberOfGenes; i++)
      {
          temp1[i] = chromosomes[fatherIdx][i];
          temp2[i] = chromosomes[motherIdx][i];
          temp3[i] = chromosomes[fatherIdx][i];
          temp4[i] = chromosomes[motherIdx][i];
        }
        // determine number of selected genes
        int numOfSelectedGenes = randomizer.Next(numberOfGenes/3);
        int[] arrayPos = new int[numOfSelectedGenes];
        for (int i = 0; i < numOfSelectedGenes; i++)
        {
            arrayPos[i] = -1; // initialize array position index to -1
        }
        for (int i = 0; i < numOfSelectedGenes; i++)
        {
            int pos1 = randomizer.Next(numberOfGenes);
            if (!IsContain2(pos1, arrayPos))
            {
                arrayPos[i] = pos1; // add number to arrayPos
            }
            else
            {
                continue;
            }
            for (int j = 0; j < numberOfGenes; j++)
            {
                if (temp1[pos1] == temp2[j]) temp2[j] = -1; // turn off this number
                if (temp4[pos1] == temp3[j]) temp3[j] = -1; // turn off this number
            }
          }
          // prepare child 1
          for (int i = 0; i < numberOfGenes; i++)
          {
              if (IsContain2(i, arrayPos))
```

```
       {
          chromosomes[child1Idx][i] = temp1[i];
       }
       else
       {
         for (int j = 0; j < numberOfGenes; j++)
         {
            if (temp2[j] != -1)
            {
               chromosomes[child1Idx][i] = temp2[j];
               temp2[j] = -1;
               break;
             }
          }
        }
       }
       // prepare child 2
       for (int i = 0; i < numberOfGenes; i++)
       {
          if (IsContain2(i, arrayPos))
          {
            chromosomes[child2Idx][i] = temp4[i];
          }
          else
          {
             for (int j = 0; j < numberOfGenes; j++)
             {
               if (temp3[j] != -1)
               {
                   chromosomes[child2Idx][i] = temp3[j];
                   temp3[j] = -1;
                   break;
                }
               }
            }
         }
}
```

```
OrderBasedCrossover (father, mother, child1, child2)
{
    // copy parent to proto-child
    for (int i = 0; i < numberOfGenes; i++)
    {
        temp1[i] = chromosomes[fatherIdx][i];
        temp2[i] = chromosomes[motherIdx][i];
        temp3[i] = chromosomes[fatherIdx][i];
        temp4[i] = chromosomes[motherIdx][i];
     }
     // determine number of selected genes
     int numOfSelectedGenes = randomizer.Next(numberOfGenes / 3);
     int[] arrayPos = new int[numOfSelectedGenes];
     int[] arrayPos2 = new int[numOfSelectedGenes];
     for (int i = 0; i < numOfSelectedGenes; i++)
     {
         // initialize arrayPos and arrayPos2
         arrayPos[i] = -1;
         arrayPos2[i] = -1;
     }
     for (int i = 0; i < numOfSelectedGenes; i++)
     {
        int pos1 = randomizer.Next(numberOfGenes);
        if (!IsContain2(pos1, arrayPos))
        {
          arrayPos[i] = pos1;
          for (int j = 0; j < numberOfGenes; j++)
          {
              if (temp1[pos1] == temp2[j])
              {
                 temp2[j] = -1; // turn off this number
                 arrayPos2[i] = j;
              }
              if (temp4[j] == temp3[pos1])
              {
                  temp3[pos1] = -1; // turn off this number
              }
           }
```

```
}
  else
  {
      continue;
  }
}
Array.Sort(arrayPos);
Array.Sort(arrayPos2);

int count = 0;
for (int i = 0; i < numOfSelectedGenes; i++)
{
    if (arrayPos[i] != -1)
    {
        count = i;
        break;
    }
}

int count2 = 0;
for (int i = 0; i < numOfSelectedGenes; i++)
{
      if (arrayPos2[i] != -1)
      {
          count2 = i;
          break;
      }
}
// prepare child 1
for (int i = 0; i < numberOfGenes; i++)
{
    if (IsContain2(i, arrayPos2))
    {
        chromosomes[child1Idx][i] = temp1[arrayPos[count]];
        count++;
    }
    else
    {
     for (int j = 0; j < numberOfGenes; j++)
     {
        if (temp2[j] != -1)
        {
           chromosomes[child1Idx][i] = temp2[j];
           temp2[j] = -1;
           break;
        }
      }
    }
  }
    // prepare child 2
    for (int i = 0; i < numberOfGenes; i++)
    {
        if (IsContain2(i, arrayPos))
        {
            chromosomes[child2Idx][i] = temp4[arrayPos2[count2]];
            count2++;
        }
        else
        {
            for (int j = 0; j < numberOfGenes; j++)
            {
                if (temp3[j] != -1)
                {
                    chromosomes[child2Idx][i] = temp3[j];
                    temp3[j] = -1;
                    break;
                }
            }
        }
    }
}
```

```
CycleCrossover ( father, mother, child1, child2)
{
    // copy parent to proto-child
    for (int i = 0; i < numberOfGenes; i++)
    {
          temp1[i] = chromosomes[fatherIdx][i];
          temp2[i] = chromosomes[motherIdx][i];
          temp5[i] = -1;
          temp6[i] = -1;
    }
    temp5[0] = 0; // cycle1 index container
    temp6[0] = temp1[0]; // cycle1 container
    temp6[1] = temp2[0];

    // find cycle1 in parents
    for (int i = 1; i < numberOfGenes; i++)
    {
        for (int j = 0; j < numberOfGenes; j++)
        {
            if (temp6[i] == temp1[j])
            {
                temp6[i + 1] = temp2[j];
                temp5[i] = j;
                break;
              }
          }
          // check cycle is completed or not
          if (temp6[i + 1] == temp6[0])
          {
                break;
          }
      }
      // prepare child 1
      for (int i = 0; i < numberOfGenes; i++)
      {
          if (IsContain2(i, temp5))
          {
              //Copy father to child1
      }
      else
      {
        // copy mother to child1
      }
     }
     // prepare child 2
     for (int i = 0; i < numberOfGenes; i++)
     {
       if (IsContain2(i, temp5))
       {
          // copy mother to child2
       }
       else
       {
          // copy father to child2
       }
      }
}
```

```
SubtourCrossover ( father , mother, child1, child2)
{
    // determine subtour position
    int pos1 = randomizer.Next(numberOfGenes);
    int pos2 = randomizer.Next(numberOfGenes);
    // make sure pos1 is less then pos2
    if (pos1 > pos2)
    {
        // exchange pos1 & pos2
    }
    // determine sublength
    int subLength = pos2 - pos1;
    // copy parent to proto-child
    for (int i = 0; i < numberOfGenes; i++)
    {
          temp1[i] = chromosomes[fatherIdx][i];
          temp2[i] = chromosomes[motherIdx][i];
          temp3[i] = -1;
    }
    // find father's subtour corresponding position in mother
    for (int i = 0; i < numberOfGenes; i++)
    {
          if (i >= pos1 && i < pos2)
          {
              for (int j = 0; j < numberOfGenes; j++)
              {
                    if (temp1[i] == temp2[j])
                    {
                          temp3[i] = j;
                          break;
                    }
              }
          }
      }

      Array.Sort(temp3);

      // check mother's subtour is reasonable or not
```

```
bool result = true;
for (int i = 0; i < numberOfGenes-1; i++)
{
   if (temp3[i] != -1)
   {
      if (temp3[i+1] != (temp3[i] + 1))
      {
         result = false;
         break;
      }
    }
 }

 int startIndex = -1;
 if (result == true)
 {
    // find startIndex in mother
    for (int i = 0; i < numberOfGenes; i++)
    {
      if (temp3[i] != -1)
      {
        startIndex = temp3[i];
        break;
      }
     }

     // prepare child 1
     for (int i = 0; i < numberOfGenes; i++)
     {
        if (i >= pos1 && i < pos2)
        {
          chromosomes[child1Idx][i] = chromosomes[motherIdx][startIndex];
          startIndex++;
        }
        else
        {
          chromosomes[child1Idx][i] = chromosomes[fatherIdx][i];
        }
        }

        // find startIndex in mother again
        for (int i = 0; i < numberOfGenes; i++)
        {
          if (temp3[i] != -1)
          {
            startIndex = temp3[i];
            break;
          }
         }

         int pos3 = pos1;
         // prepare child 2
         for (int i = 0; i < numberOfGenes; i++)
         {
          if (i >= startIndex && i < startIndex+subLength)
          {
             chromosomes[child2Idx][i] = chromosomes[fatherIdx][pos3];
             pos3++;
           }
           else
           {
              chromosomes[child2Idx][i] = chromosomes[motherIdx][i];
           }
          }
        }
        else
        {
           for (int i = 0; i < numberOfGenes; i++)
           {
             // copy father and mother to child directly
           }
        }
    }
```

Part 2: Comments on Job Assignment Problem Solved by Binary- and Permutation-Encoded GA Solvers

[Mutation comparison by experiments]

PMX + inversion
26 26 28 28 30
PMX + insertion
26 30 27 27 27
PMX + displacement
26 26 26 28 26
PMX + reciprocal exchange
26 26 26 26 26
PMX + swapped
26 26 29 30 26

Conclusion : “Reciprocal exchange” has average best solution (minimum objective value).

[Crossover comparison by experiments]

PMX + reciprocal exchange
26 26 26 26 26
OX + reciprocal exchange
26 26 26 26 26
PBX + reciprocal exchange
26 26 26 26 26
OBX + reciprocal exchange
30 26 28 29 26
CX + reciprocal exchange
26 26 26 26 26
STX + reciprocal exchange
26 26 28 26 26

PMX + inversion
26 26 26 30 26
OX + inversion
26 26 26 26 27
PBX + inversion
28 29 31 30 31

OBX + inversion
30 31 31 30 27
CX + inversion
26 28 32 30 30
STX + inversion
26 26 28 31 27

Conclusion: PMX & OX are relatively good at finding best solution, while OBX is relatively bad.