INTELLIGENT STORAGE SYSTEM MV2ME: PROOF OF CONCEPT SIMULATION
INITIAL STUDY

Kåre Olaussen[1], Ladislav Dobrovsky[2], Radomil Matousek[2]

[2]Brno University of Technology, Faculty of Mechanical Engineering Institute of Automation and Computer Science
Technická 2896/2, 616 69, Brno, Czech Republic
matousek@fme.vutbr.cz
[1]NTNU, Faculty of Natural Sciences and Technology, Department of Physics
Høgskoleringen 5, NO-7491 Trondheim, Norway

Abstract: *In mv2me storage system the storage space consists of regular grids where regular shaped containers move. This proof of concept solves case for rectangular grid and resulting simulation software is used as testbed to study emergent behaviour of random walk, soft-computing and exact algorithms considering how the containers are moved when we require more than one in optimal order.*

Keywords: *orthogonal grid, random walk, intelligent storage*

1 Introduction

Motivation for such storage system is search for low maintenance solutions with inherent scalability. Reducing the problem to moving regular shaped containers in compatible grid.

2 Problem Formulation

Let G be graph (grid) of possible container positions with edges as possible trajectories. Then let's define E as set of empty container positions, we can define a operator for swapping a container with empty container position.

$$G = \{c | c \in C \cup E\}, \tag{1}$$

$$C \cap E = \emptyset \tag{2}$$

By using the swap operator with random walk or an evolutionary algorithm, we can find the series of steps that will eventually move desired container to desired position.

3 Problem Simulation

Testbed simulation application *simshelf* enables random walk and exact algorithm testing for rectangular grid, designed bearing in mind further development of more sophisticated grids and evolutionary algorithms. Used technologies are Python, PySide and numpy.

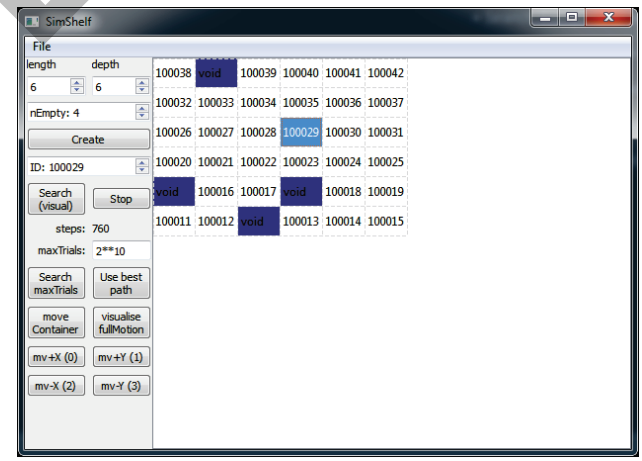

Fig. 1: SimShelf testbed application

```
    def shortestPath(self, start, end=0):
        '''Shortest path from start to end, with maximum number of bends.
        '''
        if start == end:
            return (0, [self.index(start)])
        if self.type == 'Rectangular':
            i0 = self.index(start); i1 = self.index(end)
            di = (i1[0]-i0[0], i1[1]-i0[1])
            path = [i0]
            if di[0] == 0: # Straight line in y-direction
                dy = numpy.sign(di[1])
                for n in range(abs(di[1])):
                    path.append((path[-1][0], path[-1][1] + dy))
                return (len(path)-1, path)
            if di[1] == 0: # Straight line in x-direction
                dx = numpy.sign(di[0])
                for n in range(abs(di[0])):
                    path.append((path[-1][0] + dx, path[-1][1]))
                return (len(path)-1, path)
            dx = numpy.sign(di[0]); dy = numpy.sign(di[1])
            if abs(di[0]) > abs(di[1]): # Most steps in x-direction
                for n in range(abs(di[1])):
                    path.append((path[-1][0] + dx, path[-1][1]))
                    path.append((path[-1][0], path[-1][1] + dy))
                for n in range(abs(di[0])-abs(di[1])):
                    path.append((path[-1][0] + dx, path[-1][1]))
                return (len(path)-1, path)
            else: # Most steps in y-direction
                for n in range(abs(di[0])):
                    path.append((path[-1][0], path[-1][1] + dy))
                    path.append((path[-1][0] + dx, path[-1][1]))
                for n in range(abs(di[1])-abs(di[0])):
                    path.append((path[-1][0], path[-1][1] + dy))
                return (len(path)-1, path)
        else:
            raise RuntimeError('Unimplemented shelf type')
```

Listing 1: exact shortest path algorithm for one container

```
    def makeRandomMotion(self, container, posNum=0):
        '''Move containers randomly until 'container' is at 'posNum'.
        '''
        nEmpties = len(self.emptyPositions)
        nDirs = len(self.directions)
        if self.type == 'Rectangular':
            nsteps = 0; path = []; oldNum = -5
            while True:
                empNum = numpy.random.randint(nEmpties)
                dirNum = numpy.random.randint(nDirs)
                if abs(dirNum-oldNum) == 2:
                    continue # Back-and-forth does nothing!
                if self.mvEmptyPosition(dirNum, empNum=empNum):
                    nsteps +=1
                    path.append((empNum, dirNum))
                    oldNum = dirNum
                if self.contents[posNum] is container:
                    return (nsteps, path)
        else:
            raise RuntimeError('Unimplemented shelf type')
```

Listing 2: random walk algorithm

```
    def shortestBpath(self, start, end, block):
        if start == end:
            return (0, [self.index(start)])
        if self.type == 'Rectangular':
            i0 = self.index(start); i1 = self.index(end)
            ib = self.index(block)
            di = (i1[0]-i0[0], i1[1]-i0[1])
            path = [i0]
            if di[0] == 0: # Maybe straight line in y-direction?
                dy = numpy.sign(di[1])
                if i0[0] == ib[0] and \
                    min(i0[1],i1[1]) < ib[1] < max(i0[1],i1[1]):
                    # Straight line is blocked!
                    # First a straight line
                    for n in range(abs(ib[1]-i0[1])-1):
                         path.append((path[-1][0], path[-1][1] + dy))
                    # Take a detour
                    if i0[0] > 0:
                        path.append((path[-1][0]-1, path[-1][1]))
                        path.append((path[-1][0], path[-1][1] + dy))
                        path.append((path[-1][0], path[-1][1] + dy))
                        path.append((path[-1][0]+1, path[-1][1]))
                    else:
                        path.append((path[-1][0]+1, path[-1][1]))
                        path.append((path[-1][0], path[-1][1] + dy))
                        path.append((path[-1][0], path[-1][1] + dy))
                        path.append((path[-1][0]-1, path[-1][1]))
                    # Finally a straight line
                    for n in range(abs(i1[1]-ib[1])-1):
                         path.append((path[-1][0], path[-1][1] + dy))
                    return (len(path)-1, path)
                else:
                    for n in range(abs(di[1])):
                        path.append((path[-1][0], path[-1][1] + dy))
                    return (len(path)-1, path)
            if di[1] == 0: # Maybe straight line in x-direction?
                dx = numpy.sign(di[0])
                if i0[1] == ib[1] and \
                    min(i0[0],i1[0]) < ib[0] < max(i0[0],i1[0]):
                    # Straight line is blocked!
                    # First a straight line
                    for n in range(abs(ib[0]-i0[0])-1):
                        path.append((path[-1][0] + dx, path[-1][1]))
                    # Take a detour
                    if i0[1] > 0:
                        path.append((path[-1][0], path[-1][1] - 1))
                        path.append((path[-1][0] + dx, path[-1][1]))
                        path.append((path[-1][0] + dx, path[-1][1]))
                        path.append((path[-1][0], path[-1][1] + 1))
                    else:
                        path.append((path[-1][0], path[-1][1] + 1))
                        path.append((path[-1][0] + dx, path[-1][1]))
                        path.append((path[-1][0] + dx, path[-1][1]))
                        path.append((path[-1][0], path[-1][1] - 1))
                    # Finally a straight line
                    for n in range(abs(i1[0]-ib[0])-1):
                        path.append((path[-1][0] + dx, path[-1][1]))
                    return (len(path)-1, path)
                else:
                    for n in range(abs(di[0])):
                        path.append((path[-1][0] + dx, path[-1][1]))
                    return (len(path)-1, path)
            dx = numpy.sign(di[0]); dy = numpy.sign(di[1])
            if i0[1] == ib[1]:
                for n in range(abs(di[1])):
                    path.append((path[-1][0], path[-1][1] + dy))
                for n in range(abs(di[0])):
                    path.append((path[-1][0] + dx, path[-1][1]))
            else:
                for n in range(abs(di[0])):
                    path.append((path[-1][0] + dx, path[-1][1]))
                for n in range(abs(di[1])):
                    path.append((path[-1][0], path[-1][1] + dy))
            return (len(path)-1, path)
```

Listing 3: Shortest path from start to end not passing through block

```
    def moveContainer(self, container, end=0):
        '''Efficient way to move a 'container' to position 'end'.'''
        # Find position of container:
        start = -1
        for n in range(len(self.contents)):
            if self.contents[n] is container:
                start = n
                break
        if start < 0:
            raise RuntimeError('Cannot find container')
        fullMotion = []
        # Find shortest path from start to end
        (steps, cpath) = self.shortestPath(start, end=end)
        # print (cpath)
        for n in range(len(cpath)-1):
            end = self.lindex(cpath[n+1])
            # Find the void closest to 'end', and the path moving it there
            minsteps = 2**32; empNum = -1
            for k in range(len(self.emptyPositions)):
                steps = self.shortestBpathLength(self.emptyPositions[k], end,
                        self.lindex(cpath[n]))
                if steps < minsteps:
                    minsteps = steps; empNum = k
            (length, path) = self.shortestBpath(self.emptyPositions[empNum],
                                        end, self.lindex(cpath[n]))
            # Add motion of void to the position of the container
            length += 1; path.append(cpath[n])
            # Transform the path to a set of moves
            moves = self.path2moves(path)
            # Move the void to 'end' while keeping record
            for k in range(len(moves)):
                dirNum = moves[k]
                fullMotion.append((empNum, dirNum))
                test = self.mvEmptyPosition(dirNum, empNum=empNum)
                if not test:
                    raise RuntimeError('False after %d trials' % trials)
        return fullMotion
```

Listing 4: overall algorithm for iterative best path moving

4 Conclusion

We have demonstrated that exact algorithms for one container usually yield better solutions, however for retrieving multiple containers it is worth to investigate multiple solutions by evolutionary algorithms. In case of the rectangular grid the solution is quite obvious, however we expect that for hexagonal grids or more exotic shapes of containers the leverage of the evolutionary methods will become even more overwhelming.

Acknowledgement: Support for this paper is from Norway grants project NF-CZ07-MOP-3-202-2015.

References

[1] Pereira, André Grahl, Marcus Rolf Peter Ritt, and Luciana Salete Buriol. "Finding optimal solutions to Sokoban using instance dependent pattern databases." Sixth Annual Symposium on Combinatorial Search. 2013.

[2] Matousek, R., Zampachova, E.: Promising GAHC and HC12 algorithms in global optimization tasks. Optimization Methods and Software 26(3), 405–419 (2011). DOI 10.1080/10556788.2011.556826