CA4011 - Operations Research Assignment 1

Michael Wall
13522003
michael.wall22@mail.dcu.ie

Part A (Simulation)

Structure of software simulation model

My simulation model operates on a minute by minute basis, tracking the state of the system as each minute passes. A user can specify the following parameters for the simulation:

- Number of replications of the simulation to perform
- Regular or random arrivals
- Mean time between arrivals
- Start time (in minutes) eg. 09:00 is 540 minutes
- End time (also in minutes) eg. 17:30 is 1050 minutes
- Percentage of new customers to arrive
- When to schedule new customers: first, last, or randomly
- How many servers are working, and for each one:
    - How long does it take to provide service for a new customer
    - How long does a regular customer take
    - How many breaks does this server take, and for each one:
        - When is the break due to start
        - What is the duration of the break

Although iterative simulations are usually slow, my software model allows for approximately 10000 replications of the simulation in less than 1 second on an 2GHz machine. The times for arrival are generated before the simulation using the parameters set above. Once the simulation begins, at each minute if a customer arrives they will join the queue. If a server is free, they will take the next person in the queue. The wait time for each customer is tracked and added to a total wait time which is later used to calculate the average wait time. The same thing applies for the total system time. Service times are exponentially distributed from the mean service times specified above. At the end of one replication, new data for arrivals are generated, and the statistics for one round are collected. The average results are displayed at the end of all the repetitions. I found that greater than 1000 replications gave more consistent results.

The statistics displayed by the system are as follows:

- Time to produce the replications
- Average system time per customer(in minutes)
- Average queue time per customer
- Max system time experienced by a customer across all replications
- Max queue time experienced by a customer across all replications

- Proportion of time a server is likely to be idle (decimal between 0.00 and 1.00)
- Average number of customers in the system at any time
- Average number of customers in the queue at any time
- Average number of customers served per day

Task 1 (a) - Scheduled arrivals

Customer appointments are scheduled every X minutes, where X is defined as the mean time between arrivals. There is a level of uncertainty as to when a customer will arrive, and the actual arrival times are independently distributed around the scheduled arrival times.

Assuming the following parameters
Replications: 10000
Servers: 2
Service time: 15 mins
Arrival times: 10 mins

The following results were obtained
10000 replications took 361.993793ms
Average system time: 19.79695098039212
Average queue time: 5.256101960784338
Max system time: 254
Max queue time: 141
Proportion of time idle: 0.3058996078431378
Average customers in system: 1.9844135294117793
Average customers in queue 0.5238452941176441
Average customers served 51

Task 1 (b) - Random arrival pattern

Assuming the following parameters
Replications: 10000
Servers: 2
Service time: 15 mins
Arrival times: 10 mins

10000 replications took 264.040583ms
Average system time: 27.12577330971574
Average queue time: 12.583440161255716
Max system time: 281
Max queue time: 242
Proportion of time idle: 0.3270696078431366
Average customers in system: 2.839137647058823
Average customers in queue 1.3571749019607855
Average customers served 51.7264

Task 1 - Conclusions

I would conclude that a scheduled arrival pattern is better than a random one. Although customers may arrive early or late for their appointments, the average queue time is reduced overall for customers.

I varied the values of the arrival rates and the service times to confirm this conclusion. The results of these tests are as follows:

	Scheduled	Random	Scheduled	Random
Inter arrival rate	5	5	5	5
Service time	15	15	10	10
Ave Sys time	131.77	138.88	26.71	35.82
Ave Queue time	117.25	124.39	17.21	26.35
Max Sys time	598	676	236	331
Max Queue time	545	662	202	311
Proportion Idle	0.03	0.05	0.11	0.14
Ave cust in sys	26.36	28.75	5.36	7.46
Ave cust in queue	23.45	25.81	3.44	5.53
Ave cust served	102	103	102	102

In all cases, the scheduled arrivals perform better. Servers have less idle time, and customers have less wait times. Queue lengths are also shorter.

Task 2 - Is one experienced server better than two novices?

To test this I performed tests with the following parameters and results:

	Scheduled 1 server	Scheduled 2 servers	Scheduled 1 server	Scheduled 2 servers	Random 1 server	Random 2 servers
Inter arrival rate	10	10	5	5	5	5
Service time (per server)	10	20	10	20	10	20
Ave Sys time	36.60	44.74	244.73	257.07	251.12	263.01
Ave Queue time	27.10	25.26	235.23	237.55	241.62	243.51
Max Sys time	342	421	985	974	1123	1181
Max Queue time	324	325	978	945	1111	1113
Proportion Idle	0.12	0.14	0.0074	0.03	0.02	0.04
Ave cust in sys	3.66	4.47	48.96	51.42	51.60	54.05
Ave cust in queue	2.70	2.52	47.04	47.51	49.66	50.10
Ave cust served	51	51	102	102	102.88	102.99

From this I can conclude that 1 experienced server is more effective than 2 novice servers. This will obviously depend on the level of experience of the different servers. In a standard case where 1 worker is expected to be twice as experienced as 2 workers this holds true. This also holds true for random arrivals. The experienced server shows lower average system times, although its performance on queue times is not that much better, and in some cases is slightly worse than two servers. For this reason I would conclude that having 1 experienced server is preferable to 2 novice servers.

Task 3 - Realistic breaks

For this I gave two breaks to the servers at offset times so that at least one server was always available. The following are the results obtained for scheduled arrivals with breaks:

	Scheduled	Scheduled	Scheduled	Scheduled
Inter arrival rate	10	5	5	10
Service time	15	15	10	10
Ave Sys time	25.50	166.73	47.72	11.28
Ave Queue time	10.98	152.21	38.21	1.79
Max Sys time	252	636	281	138
Max Queue time	195	593	252	92
Proportion Idle	0.21	0.03	0.06	0.41
Ave cust in sys	2.55	33.35	9.56	1.13
Ave cust in queue	1.09	30.33	7.64	0.17
Ave cust served	51	102	102	51

Similar effects are seen by varying the inter arrival rate and service times with breaks as without breaks. However, when breaks are introduced, the average system performance decreases. This is due to the queue building up while each server goes on a break.

Task 4 (a) - Some new customers, intermixed

Below are the results when customers are intermixed:

	Scheduled	Scheduled	Scheduled	Scheduled
Inter arrival rate	10	10	10	10
Service time new	15	15	15	15
Service time norm	20	30	20	30
Proportion New	0.2	0.2	0.4	0.4
Ave Sys time	30.64	44.80	36.87	71.06
Ave Queue time	15.15	27.28	20.37	50.57
Max Sys time	327	508	315	562
Max Queue time	267	406	254	464
Proportion Idle	0.18	0.14	0.15	0.09
Ave cust in sys	3.06	4.48	3.69	7.10
Ave cust in queue	1.51	2.72	2.03	5.05
Ave cust served	51	51	51	51

Adding in new customers to the system takes a toll on the system, resulting in much longer queue times than normal, although it does not have a huge effect on the queue lengths.

Task 4 (b) - Some new customers, scheduled last

Below are the results when new customers are scheduled last:

	Scheduled	Scheduled	Scheduled	Scheduled
Inter arrival rate	10	10	10	10
Service time new	15	15	15	15
Service time norm	20	30	20	30
Proportion New	0.2	0.2	0.4	0.4
Ave Sys time	50.14	122.47	40.63	81.06
Ave Queue time	31.71	96.28	23.15	57.63
Max Sys time	398	779	358	672
Max Queue time	330	615	277	603
Proportion Idle	0.13	0.10	0.15	0.13
Ave cust in sys	5.01	12.25	4.06	8.10
Ave cust in queue	3.16	9.62	2.31	5.76
Ave cust served	51	51	51	51

From these results and the results in Task 4 (a), I can conclude that scheduling new customers has a negative effect on the performance of the system overall, increasing average times in queues and systems. This presumably results in better performance for customers who are regulars, as they will experience queue times similar to those presented in Task 3. The customers who are new will experience poor system times, but this is probably acceptable for the sake of better performance times for regulars. For this reason I would say it is preferable to schedule new customers at the end of the day when possible.

Appendix - source code - simulator.go

```
package main

import (
	"errors"
	"fmt"
	"math/rand"
	"time"
)

var (
	debug = false
)

type Line struct {
	Customers []Customer
	Size      int
}

type Customer struct {
	Time  int
	IsNew bool
}

type Server struct {
	TimeLeft int
	RateOld  int
	RateNew  int
	Breaks   map[int]int
	OnBreak  bool
}

func (q *Line) Pop() (cust Customer, err error) {
	if q.Size > 0 {
		cust = q.Customers[0]
		q.Customers = q.Customers[1:]
		q.Size--
		return cust, nil
	} else {
		return cust, errors.New("Empty queue, can't pop")
	}
}
```

```
func (q *Line) Peek() (cust Customer, err error) {
	if q.Size > 0 {
		cust = q.Customers[0]
		return cust, nil
	} else {
		return cust, errors.New("Empty queue, can't peek")
	}
}

func (q *Line) Push(cust Customer) {
	q.Customers = append(q.Customers, cust)
	q.Size++
}

type Sim struct {
	Replications     int
	ArrivalDist      int
	InterArrivalTime float64
	Start            int
	End              int
	RateOfNew        float64
	ScheduleNew      int
	Servers          []Server
	Queue            Line
}

func initSim() (s Sim) {

	fmt.Print("How many replications of the system: ")

	fmt.Scanf("%v", &s.Replications)

	fmt.Print("What arrival of distributions; 0) random 1)
scheduled: ")

	fmt.Scanf("%v", &s.ArrivalDist)

	fmt.Print("What is mean time between arrivals: (in minutes)
")

	fmt.Scanf("%v", &s.InterArrivalTime)

	fmt.Print("What start time (in minutes): ")

	fmt.Scanf("%v %v", &s.Start)
```

```
	fmt.Print("What end time (in minutes): ")

	fmt.Scanf("%v %v", &s.End)

	fmt.Print("What percentage of new customers (0.00 - 1.00): ")

	fmt.Scanf("%v", &s.RateOfNew)

	fmt.Print("When will we schedule new customers? 0) first; 1)
last; 2) randomly: ")

	fmt.Scanf("%v", &s.ScheduleNew)

	fmt.Print("How many servers working: ")

	var servs int
	fmt.Scanf("%v", &servs)

	for i := 1; i <= servs; i++ {
		var rateOld int
		var rateNew int
		var breaks int

		fmt.Printf("For worker %v:\n", i)
		fmt.Printf("  How many minutes for service (scheduled
patient): ")

		fmt.Scanf("%v", &rateOld)

		fmt.Printf("  How many minutes for service (new
patient): ")

		fmt.Scanf("%v", &rateNew)

		fmt.Printf("  How many breaks does this worker take: ")

		fmt.Scanf("%v", &breaks)

		br := make(map[int]int, breaks)
		for j := 1; j <= breaks; j++ {
			var start int
			var duration int

			fmt.Printf("  What is the start time and duration
of break %v in minutes: ", j)
```

```
                fmt.Scanf("%v %v", &start, &duration)
                br[start] = duration
            }
            serv := Server{
                RateOld:  rateOld,
                RateNew:  rateNew,
                Breaks:   br,
                TimeLeft: 0,
                OnBreak:  false,
            }
            s.Servers = append(s.Servers, serv)
        }

        if debug {
            fmt.Println()
        }
        return s
}

func main() {

        rand.Seed(time.Now().UnixNano())

        s := initSim()

        ast, awt, mst, mwt, pit, acs, acw, serv := simulate(&s)

        fmt.Printf("Average system time: %v\n", ast)
        fmt.Printf("Average queue time: %v\n", awt)
        fmt.Printf("Max system time: %v\n", mst)
        fmt.Printf("Max queue time: %v\n", mwt)
        fmt.Printf("Proportion of time idle: %v\n", pit)
        fmt.Printf("Average customers in system: %v\n", acs)
        fmt.Printf("Average customers in queue %v\n", acw)
        fmt.Printf("Average customers served %v\n", serv)
}

func bleedArrivals(s *Sim) {
        for cust, err := s.Queue.Pop(); err == nil; cust, err =
s.Queue.Pop() {
            fmt.Println(cust.Time)
        }
}

func printArrivals(s *Sim) {
        fmt.Println("Schedule of arrivals")
```

```
	for i, e := range s.Queue.Customers {
		fmt.Printf("%v: %v\n", i, e.Time)
	}
}

func simulate(s *Sim) (ast, awt, mst, mwt, pit, acs, acw, aveserv
float64) {

	start := time.Now()
	for i := 0; i < s.Replications; i++ {
		var q Line

		sysTime := 0
		maxSys := 0
		totalSys := 0

		//servers := make([]int, s.Servers)

		waitTime := 0
		maxWait := 0
		totalWait := 0

		idleTime := 0
		custTotal := 0

		createArrivals(s)
		//printArrivals(s)

		idling := false

		for i := s.Start; i < s.End || !(q.Size <= 0) ||
!idling; i++ {
			if debug {
				fmt.Printf("Time: %v, Queue size: %v\n", i,
q.Size)
			}
			idling = true

			// do we need to add a new arrival to the queue

			for next, err := s.Queue.Peek(); err == nil &&
next.Time <= i; next, err = s.Queue.Peek() {
				cust, err := s.Queue.Pop()
				if err == nil {
					if debug {
						fmt.Printf("A customer arrived\n")
```

```
                        }
                        q.Push(cust)
                  }
            }

            // can our servers deal with someone
            for j, _ := range s.Servers {
                  passMinute(j, &s.Servers[j], i)
                  if s.Servers[j].TimeLeft == 0 {
                        cust, err := q.Peek()
                        if err != nil {
                              // the server is idling
                              idleTime++
                        } else {
                              idling = false
                              q.Pop()
                              if debug {
                                    fmt.Printf("Server %v took on
a customer\n", j)
                              }
                              thisWait := i - cust.Time
                              thisSys := thisWait
                              if cust.IsNew {
                                    serviceTime :=
int(float64(s.Servers[j].RateNew) * rand.ExpFloat64())
                                    s.Servers[j].TimeLeft +=
serviceTime
                                    thisSys += serviceTime
                              } else {
                                    serviceTime :=
int(float64(s.Servers[j].RateOld) * rand.ExpFloat64())
                                    s.Servers[j].TimeLeft +=
serviceTime
                                    thisSys += serviceTime
                              }

                              if thisWait > maxWait {
                                    maxWait = thisWait
                              }
                              if thisSys > maxSys {
                                    maxSys = thisSys
                              }

                              waitTime += thisWait
                              sysTime += thisSys
                              custTotal++
```

```
                              totalSys++
                        }
                  } else if !s.Servers[j].OnBreak {
                        // the server is with a patient
                        idling = false
                        totalSys++
                  } else {
                        // the server is on a break
                        if debug {
                              fmt.Printf("Server %v is on a
break\n", j)
                        }
                  }
            }

            totalWait += q.Size
            totalSys += q.Size

      }
      elapsed := float64(s.End - s.Start)
      ast += float64(sysTime) / float64(custTotal)
      awt += float64(waitTime) / float64(custTotal)
      if float64(maxSys) > mst {
            mst = float64(maxSys)
      }
      if float64(maxWait) > mwt {
            mwt = float64(maxWait)
      }
      pit += float64(idleTime) / elapsed
      acs += float64(totalSys) / elapsed
      acw += float64(totalWait) / elapsed
      aveserv += float64(custTotal)
}

ast = ast / float64(s.Replications)
awt = awt / float64(s.Replications)
mst = mst
mwt = mwt
pit = pit / float64(s.Replications) / float64(len(s.Servers))
acs = acs / float64(s.Replications)
acw = acw / float64(s.Replications)
aveserv = aveserv / float64(s.Replications)

elapsed := time.Since(start)
fmt.Printf("%v replications took %s\n", s.Replications,
elapsed)
```

```
	return
}

func passMinute(id int, serv *Server, time int) {
	if serv.TimeLeft > 0 {
		serv.TimeLeft--
	}

	// account for breaks
	if serv.TimeLeft == 0 {
		if serv.OnBreak {
			serv.OnBreak = false
		}
		for j, b := range serv.Breaks {
			if j < time && time < j+b {
				if debug {
					fmt.Printf("Server %v will go on a break
for %v minutes until %v\n", id, b, time+b)
				}
				serv.OnBreak = true
				serv.TimeLeft += b
				break
			}
		}
	}
}

func createArrivals(s *Sim) {
	if s.ArrivalDist == 0 { // random (poisson distributed)

		s.Queue.Customers = make([]Customer, 0)
		s.Queue.Size = 0
		var isNew bool
		duration := s.End - s.Start
		totalCusts := int(float64(duration) * s.RateOfNew /
s.InterArrivalTime)
		sN := s.ScheduleNew
		for lastArrival := float64(s.Start); lastArrival <
float64(s.End); {
			lastArrival = lastArrival + (s.InterArrivalTime *
(rand.ExpFloat64()))
			lenCusts := len(s.Queue.Customers)
			isNew = (rand.Float64() < s.RateOfNew && sN == 2)
|| (sN == 0 && lenCusts < totalCusts) || (sN == 1 && lenCusts >
totalCusts)
			cust := Customer{
```

```
                        Time:  int(lastArrival),
                        IsNew: isNew,
                    }

                    s.Queue.Push(cust)
                }

        } else { // scheduled, == 1

                s.Queue.Customers = make([]Customer, 0)
                s.Queue.Size = 0

                rand.Seed(time.Now().UnixNano())
                var isNew bool
                var arrivalTime float64
                duration := s.End - s.Start
                totalCusts := int(float64(duration) * s.RateOfNew /
s.InterArrivalTime)
                sN := s.ScheduleNew
                for i := s.Start; i < s.End; i +=
int(s.InterArrivalTime) {
                    arrivalTime = float64(i) + (s.InterArrivalTime *
((rand.Float64() - 0.5) * 0.5))
                    lenCusts := len(s.Queue.Customers)
                    isNew = (rand.Float64() < s.RateOfNew && sN == 2)
|| (sN == 0 && lenCusts < totalCusts) || (sN == 1 && lenCusts >
totalCusts)
                    cust := Customer{
                        Time:  int(arrivalTime),
                        IsNew: isNew,
                    }
                    s.Queue.Push(cust)
                }
        }
}
```