MIS Quarterly

RESEARCH ARTICLE

OPTIMIZING AND SATISFICING: THE INTERPLAY BETWEEN PLATFORM ARCHITECTURE AND PRODUCERS' DESIGN STRATEGIES FOR PLATFORM PERFORMANCE

Sabine Brunswicker
Research Center for Open Digital Innovation, Purdue University, 516 Northwestern Avenue,
West Lafayette, IN 49706 U.S.A. {sbrunswi@purdue.edu}

Esteve Almirall
ESADE Business School, Universitat Ramon Llull, Av. Toree Blanca 59,
SantCugat-Barcelona, SPAIN {esteve.almirall@esade.edu}

Ann Majchrzak
Marshall School of Business, University of Southern California,
Los Angeles, CA 90089 U.S.A. {amajchrzak@usc.edu}

Appendix A

Summary of Robustness Checks

We performed a series of simulations that were examining the robustness of our model specification (Davis et al. 2007). In particular, we explored whether the results would change if we modified the search heuristic h (see Table 3 in the main document) and the design moves (hill-climbing and long-jump) that define how agents search the design space when combining design elements into apps. We implemented three major robustness analyses: First, we examined the robustness of our binary representation of long-jump and hill-climbing as two dichotomous search moves, modeled in accordance with Levinthal (1997). To do so, we explored the effects of an alternative continuous representation following Billinger et al. (2013). Second, we examined the robustness of our assumption about the average amount of resources for long-jumps (R) that each agent has available when moving through the iterative search process (Billinger et al. 2013; March 1981; Rivkin 2000). Third, we also explored whether a simple categorical function to model failure-induced jumps is appropriate given alternative probabilistic models suggested in the literature on search and decision making (Greve 1998, 2002; Hu et al. 2011; Lant 1992). We will briefly report the results of these three robustness checks.

Robustness Check 1: Alternative Modeling of Local Versus Distant Search

In our simulations reported in the main document, we modeled hill-climbing and long-jumps as dichotomous facets of local versus distant search, following the line of research of Levinthal (1997). Our agents randomly change a design element in their design vector d = <d1,…,d16>. The type of search move (hill-climbing or long-jump) defines how many decision variables they change. If they are hill-climbing, they change only one element, but if they engage in a long-jump, they randomly change several (up to six) design elements in their vector. We labeled this as a "greedy" model in our simulation model, and also in the code itself. As an alternative approach, we implemented and tested Billinger et al.'s (2013) approach to modeling different facets of search. In this alternative approach, the agents gradually adjust their search distance starting with an initial search distance of three that is then adjusted according to their success. We labeled this modeling as
adaptive" (and the code respectively). In essence, this implies that if agents could find a higher position, they became more conservative and

gradually reduced the search distance over time. On the contrary, if agents are unsuccessful, they became more risk-taking by increasing their search distance gradually. Thus, our agents rapidly take many long-jumps at high levels of coupling. The use of what we call adaptive in our code resulted in a higher number of iterations, and slightly less pronounced results. However, the general insights gained from our simulations remain the same. Only minor differences could be detected. We judged our results as robust after completing these robustness checks.

Robustness Check 2: Varying the Level of Resources for Long-Jumps

The second aspect that we explored was resources for long-jumps available to our agents (March and Shapira 1992). Indeed, prior studies extending Levinthal's the NK model highlight that bold long-jumps are limited by the resources available to the agent (Billinger et al. 2013; Rivkin 2000). Further, this theoretical assumption is also consistent with empirical insights. Major design moves are resource intensive, and accrue technological debt (Gilette 2011; Woodard et al. 2013). Developing a radically new app takes time, money, and energy, and such resources deplete. Thus, we explored different scenarios by limiting the number of long-jumps available to each agent from 25, 50,100, to 250. Obviously more resources for long-jumps altered the results significantly, particularly at the lower end of the spectrum: If resources were really low (10 or 25 jumps as average), agents quickly suffered from too little resources to engage in long-jumps even if they aspired to jump because they were below their competitive aspiration. We learned that a minimum of 50 long-jumps is necessary to allow developers to cope with higher levels of coupling. If the amount of resources available is really high (e.g., 500 long-jumps as average), the differences in the effect of producers' design strategies (optimizing versus satisficing) unfold in an even more pronounced way. The downside of optimizing is even more obvious: platforms with optimizing producers perform significantly lower, and the outcome is even more skewed such that only a few stars are clearly separated from the rest of the population. However, general trends and transition points were similar, and we learned that, on platforms where "extra" effort and major design moves are needed (tight coupling), very high levels of resources for risk-taking long-jumps can be very detrimental.

Robustness Check 3: Probabilistic Function for Failure-Induced Long-Jumps

Finally, we also explored the impact of a probabilistic model for failure-induced long-jumps as a function of one agent's distance from the performance target associated with his competitive aspiration. Prior research on adaptive aspirations has concluded that both individuals and organizations often follow a simple heuristic when judging their performance as failure, and taking distant moves depending on their relative standing. They encode any value above their aspiration as a success and thus hill-climb (and the opposite for any value below as failure, triggering long-jumps). However, following prior work by Greve (2002) and other recent studies on adaptive aspirations (Hu et al. 2011; Lant 1992), we also pursued a probabilistic representation of the rule. We provided a higher probability for an agent making long-jumps if the agent is farther away from the agent's competitive aspiration (which can be either an optimizing or a satisficing one). In our probabilistic modeling, the ones that are separated from their aspiration by the greatest distance had a probability of 0.9 to engage in a long-jump; the ones that were closer to their aspiration had only 0.1 probability of taking a long-jump. The probability was linearly distributed between 0.1 and 0.9, in accordance with the constant-slope response model proposed by Greve (1998). The results obtained in the experiments with a probabilistic modeling approach were completely consistent with the ones obtained when agents follow a categorical decision rule.

Appendix B

Note on Simulation Length

Our simulation ends when all the agents exhausted their resources available for long-jumps (the maximum number of long-jumps available to them) or when no agent changes the position after a full iteration. The length of the simulations varies depending on K, the tightness of coupling of the elements in the platform, and other treatment conditions. For the reported number of simulations (based on an average maximum number of long-jumps of 100), the number of design iterations ranged from 6 to 500.

In Figure B1, we provide an overview of the length of the simulations for different levels of coupling (K), no constraint (C = 0), and speed of adjustment of S = 1 and S = 10. The length of the simulation increases as K increases. Further, with a higher S, we see that the number of iterations decreases as K increases. If we increase C, the simulations also become shorter. The average number of iterations was 311 across all simulation experiments. Thus, on average the simulations ended before the maximum length of 500 iterations because agents had exploited their resources for long-jumps or had settled on the design with the highest fitness.

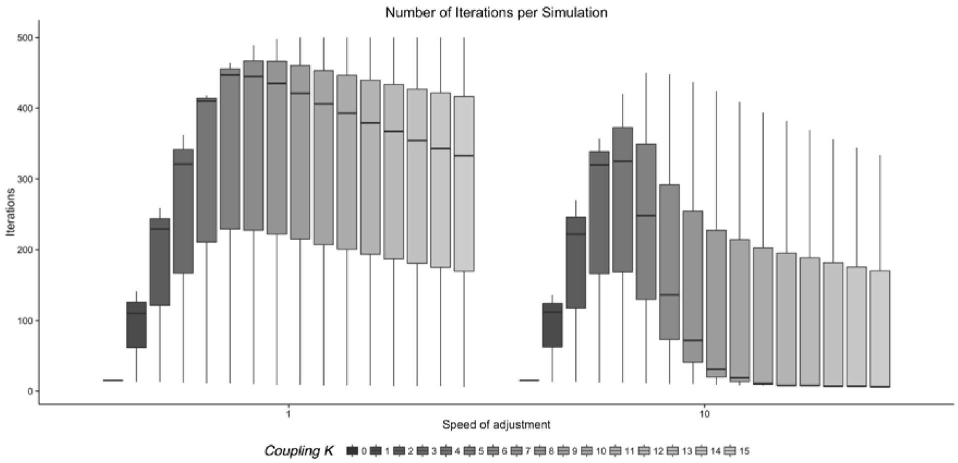

Figure B1. Overview of Length of Simulations

Appendix C

Simulation Code

Summary Information

Our computational model extends the traditional NK Model used by Levinthal (1997). In this pseudocode, we present the main loop of the simulations with variations.

The code is optimized for speed. Therefore, the code is as simple as possible using extremely simple logical structures. NK landscapes are mapped into a vector with a single index. Agents are depicted as a structure and also arranged as a vector of this structure. The program is written in Julia, a very fast dynamic programming language for high-performance numerical analysis.

The resulting algorithm is simple. For each simulation a landscape is created. Then 1,000 agents are randomly placed on it. For each iteration and each agent, a movement is executed. Hill-climbing is first attempted. If hill-climbing is not possible because the agent has reached a local maximum, a long-jump is executed in accordance with the behavioral rules specified for the agents. These movements are continued until the end of the simulation (when no movements are left).

Table C1. Summary of Code Structure (Pseudo-Code)

```
type agent
    position          # position in the NK landscape
    searchdistance    # (initially 3, only in case of adaptive jumps with changing radius
                      #in accordance with Billinger et al. 2013, not used for greedy)
    maxJumps          #each agent has a max number of jumps
    numJumps          #the number of long-jumps that has done the agent so far
end

    for constraints = none, 2 bits, 4 bits, 6 bits
        for aspiration point = none, medianAgent, topAgent
            for K = 0..15
                for experiments = 1..500
                landscape = create a landscape(N = 16, K, constraints)
                Deploy 1000 agents in random locations in the landscape
                while there are still changes AND there are iterations left
                    find aspiration point # either top, median or none if hill-climbing
                    for each agent
                        # hill-climbing
                        Search at distance 1 for the best design with platform constraints
                        If none better found AND fitness(agent)<aspiration point
                        jump by randomly changing between 2..6 bits
                        agent.numJumps++
                        there are changes = TRUE
                    end
                end
            end
        end
    end
end
```

The Implementation in Julia (version v 0.4)

BestStrategy.jl

```
include("ListStrategies.jl")
include("Fitness.jl")

function BestStrategy(strategy,ag,iN)
# BestStrategy - Looks for the best possible strategy of the agents
#Return
#     newStg  ->  New Strategy to implement
#Inputs
#    strategy->  1)Incremental + greedy (max fitness)
#            2)Incremental + fitter (better fitness with fitness' prob.)
#            3)Pattern selection
#     ag  ->  Agent to be considered
#    iN    ->  Range of bits to consider e.g., beginning:  end (depends if some components are fixed ...)

maxFit=Fitness(ag.stg)
newStg=ag.stg
```

```
#if (strategy == 1 || strategy == 2 || strategy==3 || strategy==4 || strategy==5)

                # Incremental + greedy
            lStg=ListStrategies(ag.stg,iN,1)

            for lS=1:size(lStg)[1]
                    if (Fitness(lStg[lS]) > maxFit)
                newStg=lStg[lS]
                maxFit=Fitness(lStg[lS])
            end
        end
#end
return newStg
end
```

BitGet.jl

```
function BitGet(i,nbit)
# BitGet        Returns the value of a certain bit
# Returns:
#     0,1       -> value of the bit
# Inputs:
#     i         -> integer to consider
#                   nbit -> number of bit to consider

i=int32(i)

if (i & int32(2^(nbit-1))) >0
    return 1
else
    return 0
end

end
```

BitSet.jl

```
function BitSet(i,nbit,val)
# BitSet        Returns i with nbit set to val
# Returns:
#     i         ->   i with nbit set to val
# Inputs:
#     i         -> integer to consider
#               nbit      -> number of bit to consider
#               val            -> value to set (0,1)

i=int32(i)

if val==1
    i=i|2^(nbit-1)
else
    i=i&~(2^(nbit-1))
end

return i

end
```

ListStrategies.jl

```
function ListStrategies(stgO,iN,M)
# ListStrategies From a given strategy, lists all strategies that differ in M or less components
# Returns:
#     lStg -> vector with all possible strategies
# Inputs:
#     stgO -> original strategy
#     iN -> elements (bits) to be considered in the set
#     M -> maximum number of components in which strategies can differ=1;
xM=M

lStg=zeros(Int,1)

if (xM>size(iN,1))
    xM=size(iN,1)
end

for i=1:xM
    combi=collect(combinations(iN,i))
    n_combi=size(combi,1)
    s_combi=size(combi[1],2)

    for j=1:n_combi
        n_stg=stgO
        for t=1:s_combi

            if (n_stg & 2^(combi[j,t][1]-1)) == 0
            # if (bitget(n_stg,combi[j,t])==0)
                n_stg=(n_stg | 2^(combi[j,t][1]-1))
                # n_stg=bitset(n_stg,combi[j,t]);
            else
                n_stg=(n_stg $ 2^(combi[j,t][1]-1))
                # n_stg=bitset(n_stg,combi[j,t],0);
            end
        end

        if i==1 && j==1     # first time
            lStg[1]=n_stg
        else
            push!(lStg,n_stg)
        end
    end
end
return lStg
end
```

CreaLandscape.jl

```
#---------------------------------------------------------------------------
function CreaLandscape(N,K)
# CreaLandscape    Creates a landscape N-K (see Kauffman)
# Returns:
#     m_cs->max interactions
#     CS -> global variable that contains vector dependencies
#     CV -> global variable that contains random number used to build the
#     landscape
# Inputs:
#     N -> number of different components of the Strategy
#     K -> number of components of wich every single component depends on


#global landscape
#global cs
#global maxLand

dosaN=2^N
dosaK1=2^(K+1)

landscape=zeros(dosaN,1)

cs=zeros(Int,N,K+1)
cvx=rand(dosaK1,N)

#Random with repetition
for i=[1:N]
      tmp=[1:i-1,i+1:N]
      tmp1=randperm(N-1)
            cs[i,:]=[i tmp[tmp1[1:K]]']
##    cs(i,:)=sort(cs(i,:))
end

maxval=0
minval=9

for i=[0:(dosaN-1)]
      valor=0;
      for j=[1:N]
            ind=0;
            for p=[1:(K+1)]
#                 pm=int32(2^cs[j,p])
#                 println(i," ",pm," ",i&pm," ",ind|pm)
            if (i & int32(2^(cs[j,p]-1))) >0
                  ind=(ind | 2^(p-1))
            end
#                 if (bitget(i,cs(j,p))==1)
#                       ind=bitset(ind,p,1);
#                 end
            end
            valor=valor+cvx[ind+1,j]
      end

      landscape[i+1]=valor/N
      if (landscape[i+1]>maxval)
            maxval=landscape[i+1]
```

```
            maxLand=i+1
        end
        if (landscape[i+1]<minval)
            minval=landscape[i+1]
        end
end

dif=maxval-minval
landscape=(landscape.-minval)/dif

return landscape
end
```

Fitness.jl

```
function Fitness(stg)
# Fitness       Returns the fitness of an strategy
# Returns:
#     fit    -> fitness corresponding to the strategy of the agent
#                              corresponds to the strategy of the agent + 1 into the landscape
# Inputs:
#     stg         -> strategy of the agent

global landscape

fit=landscape[stg+1]

return fit

end
```

Simula.jl

```
include("BestStrategy.jl")
include("Fitness.jl")
include("BitGet.jl")
include("BitSet.jl")

#--------------------------------------------------------------------------
function Simula(strategy,ag,aex...)
# Simula    Performs a simulation depending on the Strategy
#                       -> strategy=1 - Hill-climbing
#                       -> strategy=2 - Hill-climbing with info about avg Fitness of the Landscape
# Returns:
#      ag                   ->    structure of agents
#           bestCases    ->  final benchmark
# Inputs:
#           strategy ->    0= Hill-climbing - used as a baseline
#                                 1= Hill-climbing with restricted bits
#                                 2= Hill-climbing with explorers using max fitness found w restricted bits
#                                 3= Hill-climbing with explorers using avg fitness found w restricted bits
#                                 4= Hill-climbing using Best Cases from explorers
#                                 5= Hill-climbing from Best Cases extracted from the agents themselves
#           ag              ->    structure of agents
#           aex            ->    structure of the explorers or number of array of agents to consider for Best Cases

#           required global variables
#                 landscape     -> the vector representing the landscape
#           N                -> number of different components of the Strategy
#           K                -> number of components of which every single component depends on


global landscape
global N, K

global nBestCases

global _fixbits, _freebits,_fixval
global _dpivot

dosaN=2^N
dosaK1=2^(K+1)

nagents=size(ag,1)

#counting iterations
_niter=0

if strategy==1 || strategy==0
      # Do Hill-climbing
      canvi=true

      while canvi
            canvi=false
            for i=1:nagents
                  if strategy==0
                        newStg=BestStrategy(strategy, ag[i],[1:N])
                  else
                        newStg=BestStrategy(strategy, ag[i], _freebits)
```

```
            end
            if newStg != ag[i].stg
                ag[i].stg=newStg
                canvi=true
            end
        end
        _niter=_niter+1
    end
    return (ag, _niter)
end

if (strategy==2 || strategy==3 || strategy==4)
    # Do Hill-climbing with Explorers with the max fitness found by the explorers
    ex=aex[1]
    e=size(ex,1)
    if (strategy==2 || strategy==3)
        avgEx=0
        for i=1:e
            if strategy==2
                if Fitness(ex[i].stg)>avgEx
                    avgEx=Fitness(ex[i].stg)
                end
            else
                avgEx=avgEx+Fitness(ex[i].stg)
            end
        end
        if strategy==3
            avgEx=avgEx/e
        end
#       @printf("avgEx %4f\n",avgEx)
    else
        #Select the best cases found by explorers
        bestCases=zeros(e)
        for i=1:e
            bestCases[i]=Fitness(ex[i].stg)
        end
        bestCases=sort(bestCases,rev=true)
    end

    canvi=true

    while canvi
        canvi=false
        if _dpivot>0
            #find minimum fitness
            _minfit=9.0
            for i=1:nagents
                if Fitness(ag[i].stg)<_minfit
                    _minfit=Fitness(ag[i].stg)
                end
            end
        end
        for i=1:nagents
            if ag[i].nPivot<ag[i].mPivot
                newStg=BestStrategy(strategy, ag[i], _freebits)
                if newStg != ag[i].stg
                    ag[i].stg=newStg
                    canvi=true
```

```
                else
#                   @printf("agent %2d tBCase %2d nPivot %2d mPivot %2d \n",i,ag[i].tBCase,ag[i].nPivot,ag[i].mPivot)
                    if ((strategy== 2 || strategy==3) && Fitness(newStg)<avgEx) ||
                        ( strategy==4 && Fitness(newStg)<bestCases[ag[i].tBCase] )
#                       @printf("Old strategy %7f New strategy %7f",ag[i].stg,newStg)
#                       @printf("Aixo no hauria de passar Fitness(newStg) %7f avgEx %7f dif %7f
\n",Fitness(newStg),avgEx,avgEx-Fitness(newStg))
                        # Jump
                        _jump=false
                        if _dpivot==0
                            #greedy
                            _jump=true
                        else
                            #only 1 proportional negative is considered
                            if (strategy==2 || strategy ==3)
                                _p=(Fitness(ag[i].stg)-_minfit)/(avgEx-_minfit)
                            else
                                _p=(Fitness(ag[i].stg)-_minfit)/(bestCases[ag[i].tBCase]-_minfit)
                            end
                            _p=1-_p
                            if rand()<=_p
                                _jump=true
                            end
                        end
                        if _jump==true
                            btC=int(rand()*4)+2     #bt 2..6 bits
                            for j=1:btC
                                bC=int(rand()*(length(_freebits)-1))+1
                                if BitGet(ag[i].stg,_freebits[bC])==0  # Flip
                                    ag[i].stg=BitSet(ag[i].stg,_freebits[bC],1)
                                else
                                    ag[i].stg=BitSet(ag[i].stg,_freebits[bC],0)
                                end
                            end
                            canvi=true
                            ag[i].nPivot=ag[i].nPivot+1
                        end
                    end
                end
            end
        end
        _niter=_niter+1
    end
    return (ag, _niter)
end

if (strategy==5)
    # Do Hill-climbing using Best Cases crowdsourced from the agents themselves
    ex=aex[1]
    e=size(ex,1)
    bestCases=zeros(e)
    for i=1:e
        bestCases[i]=Fitness(ag[ex[i]].stg)
    end
    bestCases=sort(bestCases,rev=true)


#   for i=1:length(bestCases)
```

```
#       @printf("Best Case %2d %2.5f \n",i,bestCases[i])
#   end

    avgF=0
    for i=1:nagents
        avgF=avgF+Fitness(ag[i].stg)
    end
    avgF=avgF/nagents
#    @printf("Init %3d Average fitness of Best Cases %2.5f Agents %2.5f\n",e,mean(bestCases[1:5]),avgF)

    canvi=true
    njump=0

    while canvi
        canvi=false
        if _dpivot>0
            #find minimum fitness
            _minfit=9.0
            for i=1:nagents
                if Fitness(ag[i].stg)<_minfit
                    _minfit=Fitness(ag[i].stg)
                end
            end
        end
#       @printf("We have _minfit \n")

        for i=1:nagents
            if ag[i].nPivot<ag[i].mPivot
                newStg=BestStrategy(strategy, ag[i], _freebits)
                if newStg != ag[i].stg
                    ag[i].stg=newStg
                    canvi=true
                else
#                   @printf("Are we going to jump? Fitness(newStg) %5f bestCases[ag[i].tBCase] %5f
\n",Fitness(newStg),bestCases[ag[i].tBCase])
                    if Fitness(newStg)<bestCases[ag[i].tBCase]
#                   @printf("Are we going to jump 2?\n")
#                   @printf("agent %2d tBCase %2d nPivot %2d mPivot %2d \n",i,ag[i].tBCase,ag[i].nPivot,ag[i].mPivot)
#                   @printf("Fitness(newStg) %4f bestCases[ag[i].tBCase] %4f \n",Fitness(newStg),bestCases[ag[i].tBCase])
                        # Jump
                        _jump=false
                        if _dpivot==0
                            #greedy
                            _jump=true
                        else
                            #only 1 proportional negative is considered
                            _p=(Fitness(ag[i].stg)-_minfit)/(bestCases[ag[i].tBCase]-_minfit)
                        end
                        _p=1-_p
                        if rand()<=_p
                            _jump=true
                        end
                        if _jump==true
                            btC=int(rand()*4)+2     #bt 2..6 bits
                            for j=1:btC
                                bC=int(rand()*(length(_freebits)-1))+1
                                if BitGet(ag[i].stg,_freebits[bC])==0  # Flip
                                    ag[i].stg=BitSet(ag[i].stg,_freebits[bC],1)
```

```
                                    else
                                        ag[i].stg=BitSet(ag[i].stg,_freebits[bC],0)
                                    end
                                end
                                canvi=true
                                ag[i].nPivot=ag[i].nPivot+1
                                njump=njump+1
                            end
                        end
                    end
                end
            end

            bestCases=zeros(e)
            for i=1:e
                bestCases[i]=Fitness(ag[ex[i]].stg)
            end
            bestCases=sort(bestCases,rev=true)
            _niter=_niter+1
        end
#       avgF=0
#       for i=1:nagents
#           avgF=avgF+Fitness(ag[i].stg)
#       end
#       avgF=avgF/nagents
        # @printf(" ...      Average fitness of Best Cases %2.5f Agents %2.5f jumps %4d\n",mean(bestCases[1:5]),avgF,njump)
#       for i=1:nagents
#           if Fitness(ag[i].stg)<bestCases[ag[i].tBCase]
                #tobat
#               @printf(">>> agent %3d fitness %2.4f tBCase %2d fitness Best Case %2.4f nPivots %3d maxPivots %3d \n",
#               i,Fitness(ag[i].stg),ag[i].tBCase,bestCases[ag[i].tBCase],ag[i].nPivot,ag[i].mPivot)
#           end
#       end

#       return ag, bestCases[1:nBestCases]
        return (ag, _niter)

end
```

NKtransp.jl

```
# NKtransp -------
# command line inputs
# NKtransp.jl <nagents> <nexperiments> <maxTrials> <agentsRisk> <platformBits> <meanPivots> <forceAdopt>
#     nagents              Number of agents to be deployed in the landscape - typically 1000
#     nexperiments       Number of experiments to perform - bt 100..1000
#     maxSearchTrials  Max number of Search Trials - bt 100..1000
#     agentsRisk          0-> conservative.  First they exhaust all incremental opportunities then engage in long-jumps
#                             1-> adaptive.  They engage in adaptive behavior all the time and change their search radius.
#     platformBits   Number of bits fixed devoted to the platform.
#     meanPivots         Mean number of Pivots that agents will do.  Normally distributed around meanPivots, std=1
#     speed                 Speed of the update of the social benchmark 0-> static -1 ->every iteration n-> every n iterations
#

include("../CreaLandscape.jl")
include("../BestStrategy.jl")
include("../Fitness.jl")
include("../Simul.jl")

global landscape, N, K

global _fixbits, _freebits,_fixval


N=16
K=0

#get parameters from command line args
if size(ARGS,1)!=7
      @printf("Incorrent args in command line\n")
      @printf("NKtransp.jl <nagents> <nexperiments> <maxSearchTrials> <agentsRisk> <platformBits> <meanPivots> <speed>
\n")
      exit()
end

_nag=parse(Int,ARGS[1])
_nexp=parse(Int,ARGS[2])
_mST=parse(Int,ARGS[3])        #normally 5* _nexp
_agR=parse(Int,ARGS[4])
_nfixbits=parse(Int,ARGS[5])
_mPivots=parse(Int,ARGS[6])
_speed=parse(Int,ARGS[7])

#Fix bits and assign them a value
_fixbits=randperm(N)
_freebits=_fixbits[1:end-_nfixbits]
_fixbits=_fixbits[end-(_nfixbits-1):end]

_fixval=zeros(_nfixbits)
for i in 1:_nfixbits
     _fixval[i]=round(Int,rand())
end

#File name
fname="NK-""B"string(_agR)"Pl"string(_nfixbits)"Pv"string(_mPivots)"S"string(_speed)Libc.strftime("%Y-%m-%d %H:%M", time())

fOut=open(string(fname,".dat"),"w+")
```

```
fOutCsv=open(string(fname,".csv"),"w+")
fOutCsvD=open(string(fname,"D",".csv"),"w+")

write(fOutCsv,"N.Iter, #Bench, K, #Simu, Mean Fitness, Std Fitness, Search Distance\n")
write(fOutCsvD,"N.Iter, #Bench, K, #Simu, #Agent, Fitness, Search Distance\n")

if _nfixbits!=0
    nB=6
    B=[-1 0 1 2 3 4]
else
    nB=5
    B=[0 1 2 3 4]
end


avgFit=zeros(nB,N,_nexp)
miterF=zeros(nB,N,_mST)
niterF=zeros(nB,N,_mST)

type agent
    stg::Int64
    last::Int64
    sD::Int32
    mPivot::Int32
    nPivot::Int32
end

ag=Array(agent,_nag)
aFitness=zeros(_nag)

cB=1
for b in B
    for K=0:N-1
        @printf("Benchmark %2d NKtransp K=%2d \n",b,K)
        flush(STDOUT)
        siter=0
        for t=1:_nexp

            #Create a landscape
            landscape=CreaLandscape(N,K)

            # Put the agents on the floor
            for i=1:_nag
                init=round(Int,rand()*(2^N-1))
                if b<0
                    #Baseline without restricted bits
                    ag[i]=agent(init,init,0,0,0) # 0..2^N -1
                else
                    _ag=agent(init,init,0,0,0) # 0..2^N -1
                    for j=1:length(_fixbits)
                        _ag.stg=BitSet(_ag.stg,_fixbits[j],_fixval[j])
                    end
                    ag[i]=_ag
                end
                ag[i].sD=3     #initially we set the Search Distance to 3
                ag[i].mPivot=round(Int,randn()+_mPivots)
                ag[i].nPivot=Int(0)
            end
```

```
            ag, _niter, iterF=Simul(ag,b,_agR,_mST,_speed)

            for i=1:_mST
                miterF[cB,K+1,i] += iterF[i]
                if iterF[i] !=0
                    niterF[cB,K+1,i] +=1
                end
            end

            for i=1:_nag
                #    println(Fitness(ag[i].stg))
                aFit=Fitness(ag[i].stg)
                avgFit[cB,K+1,t]=avgFit[cB,K+1,t]+aFit
                aFitness[i]=aFit
                writecsv(fOutCsvD,[_niter b K t i aFit ag[i].sD])
            end
            avgFit[cB,K+1,t]=avgFit[cB,K+1,t]/_nag
            siter=siter+_niter

            writecsv(fOutCsv,[_niter b K t mean(aFitness) std(aFitness) mean(ag[].sD)])

            #        println(avgFit[K+1,t])
        end
        @printf("N. of iterations %3d, Fitness %4f, Search Distance %2d \n",siter/_nexp,mean(avgFit[cB,K+1,:]),mean(ag[].sD))
        flush(STDOUT)
    end
    cB=cB+1

end

for i=1:nB
    for j=1:N
        for k=1:_mST
            if niterF[i,j,k] !=0
                miterF[i,j,k]=miterF[i,j,k]/niterF[i,j,k]
            else
                miterF[i,j,k]=0
            end
        end
    end
end
serialize(fOut,avgFit)
serialize(fOut,miterF)
close(fOut)
close(fOutCsv)
close(fOutCsvD)

#for i=1:2^16
#    @printf("Landscape %5d %7.3f  \n ",i,landscape[i])
#end

#@printf("Max landscape min landscape %7.3f %7.3f %7.3f\n",maximum(landscape),minimum(landscape),mean(landscape))
```

References

Billinger, S., Stieglitz, N., and Schumacher, T. R. 2013. "Search on Rugged Landscapes: An Experimental Study," *Organization Science* (25:1), pp. 93-108.

Gilette, F. 2011. "The Rise and Inglorious Fall of Myspace," *Bloomberg.Com* (https://www.bloomberg.com/news/articles/2011-06-22/the-rise-and-inglorious-fall-of-myspace).

Greve, H. R. 1998. "Performance Aspirations and Risky Organizational Change," *Administrative Science Quarterly* (43:1), pp. 58-86.

Greve, H. R. 2002. "Sticky Aspirations: Organization Time Perspective and Competitiveness," *Organization Science* (13:1), pp. 1-17.

Hu, S., Blettner, D., and Bettis, R. A. 2011. "Adaptive Aspirations: Performance Consequences of Risk Preferences at Extremes and Alternative Reference Groups," *Strategic Management Journal* (32:13), pp. 1426-1436.

Lant, T. K. 1992. "Aspiration Level Adaptation: An Empirical Exploration," *Management Science* (38:5), pp. 623-644.

Levinthal, D. A. 1997. "Adaptation on Rugged Landscapes," *Management Science* (43:7), pp. 934-950.

March, J. G. 1981. "Variable Risk Preferences and Adaptive Aspirations," *Journal of Economic Behavior and Organization* (9:1), pp. 5-24.

Rivkin, J. W. 2000. "Imitation of Complex Strategies," *Management Science* (46:4), pp. 824-844.

Woodard, C. J., Ramasubbu, N., Tschang, F. T., and Sambamurthy, V. 2013. "Design Capital and Design Moves: The Logic of Digital Business Strategy," *MIS Quarterly* (37:2), pp. 537-564.