Abstract

Esse espaço será utilizado para o abstract do artigo.

Contents

1 Introduction

GRASP é uma metaheurística multistart de busca local que consiste em duas fases:

1. Construção
2. Busca Local

O C-GRASP é uma derivação do GRASP para resolver problemas no domínio contínuo de otimização global sujeito a restrições de caixa. C-GRASP consiste em uma série de ciclos de melhoria na construção local com sua saída sendo entrada para o melhoramento local,e a saída do melhoramento local sendo a entrada da construção. O parâmetro h controla a discretização do espaço de busca. A variável $NumIterNoImprov$ controla a densidade da busca no grid. A função modificada $Ternary'$ é uma função de mapeamento que converte um número na base 10 para a base 3, substituindo a ocorrência do número 2 por -1. O parâmetro $\alpha \in [0, 1]$ é usado para limitar a lista restrita de candidatos (RCL).

Segue algumas notas e lembrentes

1. Usar Mersenne Twister algorithm para geracao de números aleatórios
   http://create.stephan-brumme.com/mersenne-twister/ ou
   http://fmg-www.cs.ucla.edu/geoff/mtwist.html ou
   http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
2. Para o critério de parada podemos utilizar a definição da solução ser significativamente proxima ao ótimo global, ou depois de completar 20 iterações multistart. Definimos como significativamente perto da solução se: $\mid f^* - f^{\sim} \mid \leq \epsilon_1 \mid f^* \mid + \epsilon_2$ Onde $\epsilon_1 = 10^{-4}$ e $\epsilon_2 = 10^{-6}$
   Para cada iteração usaremos um seed diferente para gerar números aleatórios.
3. Para embarcar o codigo em python em outras aplicaçoes
   https://docs.python.org/3/extending/embedding.html
4. Para tratamento de verificacao do input das bibliotecas seja feito com um codigo python, mais especificamente o pyparsing
   http://pyparsing.wikispaces.com/
5. Utilizaremos varios benchmarks de teste que usaremos para testar e validar esta biblioteca
   http://www.mat.univie.ac.at/~neum/glopt/test.html
6. Porem começaremos utilizando as funcoes descritas neste artigo
   http://arxiv.org/pdf/1308.4008.pdf
7. Segue abaixo as funcoes de uma competicao do CEC para uma possível participação
   http:
   //goanna.cs.rmit.edu.au/~xiaodong/lsgo-competition-cec15/

8. Para analise no tempo de execuçao utilizaremos o $gprof$ https://sourceware.org/binutils/docs/gprof/ e o $Valgrind$ http://valgrind.org/

9. Utilizaremos a biblioteca MPRF para suportar precisão aritmética de pontos flutuantes http://www.mpfr.org/
   Não esquecer de compilar com a opção -lmpfr -lgmp

10. Para geraçao de graficos utilizaremos gnuplot http://gnuplot.info, tttplot http://mauricio.resende.info/tttplots/ e para extrair estatísticas utilizaremos o R https://www.r-project.org

11. Por último afim de paralelizar o programa utilizaremos o OpenMP http://openmp.org/wp/
    Não esquecer de compilar com a opção $-fopenmp$!

2 Continuous Greedy Randomized Adaptive Search Procedure (C-GRASP)

.

.

Estratégia (Bottom-UP)

Line Search:

A ideia fundamental por trás da abordagem de busca é gerar e avaliar de forma iterativa soluções candidatas; no caso de um problema de optimização, que tipicamente envolve a determinação o respectivo valor da função objetivo.

Busca local é um método para buscar em um determinado espaço de soluções candidatas(que começa a partir de uma solução candidata inicial) e, em seguida, de forma iterativa, se move de uma solução candidata a uma solução candidata de sua vizinhança direta, com base em informações locais, até que uma condição de término é satisfeita.

O Line Search é uma técnica que exibe, para algumas extensões, características chamada de cão de guarda, que tem sido proposto em ligação com métodos de programação quadrática recursivas para otimização restrita. Line search garante uma diminuição monotônica da função objetivo.

O Line Search é realizado em cada sentido da coordenarda livre(unfixed) i de x, com o outro $n-1$ coordenadas de x em seus valores atuais. O valor z_i para a iésima($i-th$) coordenada que minimiza a função objetivo do Line Search assim como o valor da função objetivo g_i são salvos.

Notas: o Line Search recebe uma solução do problema e cria uma hiperesfera de raio h, $||x^* - x|| == h$. Tomamos um valor de uma das coordenadas livres (unfixed), escolhida de forma aleatoria, e projetamos ela sobre a superfície. Ao final a variável livre sai do conjunto S

Veja : Algorithm 8 Line Search

Algorithm 1 C-GRASP

Input:

n ▷ Dimension
l ▷ Lower Bound
u ▷ Upper Bound
$f(.)$ ▷ Objective function
$MaxInter$ ▷ Times to call construction and local improvement phases
$MaxNumIterNoImprov$ ▷ Maximum times with no improvement
$NumTimesToRun$ ▷ Maximum number of multistart
$MaxDirToTry$ ▷ Maximum distinct random directions
α ▷ $\in [0, 1]$

Output:

x^*

Begin:

$f^* \leftarrow \infty$;
for $j = 1 \rightarrow NumTimesToRun$ **do**
    x $\leftarrow$ UnifRand(l,u);
    h $\leftarrow$ 1;
    NumIterNoImprov $\leftarrow$ 1;
    for Iter=1 $\rightarrow MaxInter$ **do**
        $x \leftarrow$ ConstructGreedRandomized(x,$f(.)$,n,h,l,u,α);
        $x \leftarrow$ LocalImprovement(x,$f(.)$,n,h,l,u,$MaxDirToTry$);
        if $f(x) < f^*$ **then**
            $x^* \leftarrow x$;
            $f^* \leftarrow f(x)$;
            $NumIterNoImprov \leftarrow 0$;
        else
            $NumIterNoImprov \leftarrow NumIterNoImprov + 1$;
        end if
        if $NumIterNoImprov \geq MaxNumIterNoImprov$ **then**
            $h \leftarrow h/2$;
            $NumIterNoImprov \leftarrow 0$;
        end if
    end for
end for
return x^*

Algorithm 2 C-GRASP improved

Input:

n ▷ Dimension
l ▷ Lower Bound
u ▷ Upper Bound
$f(.)$ ▷ Objective function
h_s ▷ Starting grid
h_e ▷ Ending grid
ρ_{lo} ▷ Portion of the neighborhood
$MaxInter$ ▷ Times to call construction and local improvement phases
$MaxNumIterNoImprov$ ▷ Maximum times with no improvement
$NumTimesToRun$ ▷ Maximum number of multistart
$MaxDirToTry$ ▷ Maximum distinct random directions
α ▷ $\in [0, 1]$

Output:

x^*

Begin:

$f^* \leftarrow \infty$;
for $j = 1 \rightarrow NumTimesToRun$ **do**
  x $\leftarrow$ UnifRand(l,u);
  $h \leftarrow h_s$
  while $h \geq h_e$ **do**
    $Improv_c \leftarrow$ False;
    $Improv_l \leftarrow$ False;
    $[x, Improv_c] \leftarrow$ ConstructGreedyRandomized$(x, f(,), n, h, l, u, Improv_c)$;
    $[x, Improv_l] \leftarrow$ LocalImprovement$(x, f(.), n, h, l, u, \rho_{lo}, Improv_l)$;
    if $f(x) < f^*$ **then**
      $x^* \leftarrow x$;
      $f^* \leftarrow f(x)$;
    end if
    if $Improv_c$ =False and $Improv_l$ =False **then**
      $h \leftarrow h/2$
    end if
  end while
end for
return x^*

Algorithm 3 ConstructGreedyRandomized

Input:

x ▷ Solution x
$f(.)$ ▷ Objective function
n ▷ Dimension
h ▷ Grid density
l ▷ Lower Bound
u ▷ Upper Bound
α ▷ $\in [0, 1]$

Output:

x

Begin:

$S \leftarrow \{1, 2, ..n\}$
do
  $min \leftarrow +\infty$;
  $max \leftarrow -\infty$;
  for i=1 $\rightarrow n$ **do**
    if $i \in S$ **then**
      $z_i \leftarrow$ LineSearch$(x, h, i, n, f(.), l, u)$;
      $g_i \leftarrow f(z_i)$
      if $min > g_i$ **then**
        $min \leftarrow g_i$
      end if
      if $max < g_i$ **then**
        $max \leftarrow g_i$
      end if
    end if
  end for
  $RCL \leftarrow \emptyset$
  for i=1 $\rightarrow n$ **do**
    if $i \in S$ and $g_i \leq (1 - \alpha) * min + \alpha * max$ **then**
      $RCL \leftarrow RCL \cup \{i\}$
    end if
  end for
  $j \leftarrow$ RandomlySelectElement(RCL);
  $x_j \leftarrow z_j$
  $S \leftarrow S \setminus \{j\}$
while $S \neq \emptyset$
return x

Algorithm 4 ConstructGreedyRandomized improved

Input:

x ▷ Solution x
$f(.)$ ▷ Objective function
n ▷ Dimension
h ▷ Grid density
l ▷ Lower Bound
u ▷ Upper Bound
$Improv_c$ ▷ Construct improved

Output:

$(x, Improv_c)$

Begin:

$S \leftarrow \{1, 2, ..n\}$
$\alpha \leftarrow$ UnifRand(0,1);
$Reuse \leftarrow$False;
do
  $min \leftarrow +\infty$;
  $max \leftarrow -\infty$;
  for i=1 $\rightarrow n$ **do**
    if $i \in S$ **then**
      if $Reuse$ =False **then**
        $z_i \leftarrow$ LineSearch$(x, h, i, n, f(.), l, u)$;
        $g_i \leftarrow f(z_i)$
      end if
      if $min > g_i$ **then**
        $min \leftarrow g_i$
      end if
      if $max < g_i$ **then**
        $max \leftarrow g_i$
      end if
    end if
  end for
  $RCL \leftarrow \emptyset$
  $Threshold \leftarrow min + \alpha.(max - min)$
  for i=1 $\rightarrow n$ **do**
    if $i \in S$ and $g_i \leq Threshold$ **then**
      $RCL \leftarrow RCL \cup \{i\}$
    end if
  end for
  $j \leftarrow$ RandomlySelectElement(RCL);
  if $x_j = z_j$ **then**
    $Reuse \leftarrow$True;
  else
    $x_j \leftarrow z_j$;
    $Reuse \leftarrow$False;
    $Improv_c \leftarrow$True;
  end if
  $S \leftarrow S \setminus \{j\}$
while $S \neq \emptyset$
return $(x, Improv_c)$

Algorithm 5 LocalImprovement

Input:

x ▷ Solution x
$f(.)$ ▷ Objective function
n ▷ Dimension
h ▷ Grid density
l ▷ Lower Bounf
u ▷ Upper Bound
$MaxDirToTry$ ▷ Maximum distinct random directions

Output:

x^*

Begin:

$improved \leftarrow$ True
$D \leftarrow \emptyset$
$NumDirToTry \leftarrow min\{3^n - 1, MaxDirToTry\}$
while $improved$ **do**
  $improved \leftarrow$ False
  while $\mid D \mid \leq NumDirToTry$ and not $improved$ **do**
    Generate $r \leftarrow \lceil \text{UnifRand}(1, 3^n - 1) \rceil \notin D$
    $D \leftarrow D \cup \{r\}$
    $d \leftarrow$Ternary'(r)
    $x \leftarrow x^* + h * d$
    if $l \leq x \leq u$ **then**
      if $f(x) < f^*$ **then**
        $x^* \leftarrow x$
        $f^* \leftarrow f(x)$
        $D \leftarrow \emptyset$
        $improved \leftarrow$ True
      end if
    end if
  end while
end while
return x^*

Algorithm 6 LocalImprovement improved

Input:

x ▷ Solution x
$f(.)$ ▷ Objective function
n ▷ Dimension
h ▷ Grid density
l ▷ Lower Bounf
u ▷ Upper Bound
ρ_{lo} ▷ Portion of the neighborhood
$Improv_l$ ▷ Local Improvement

Output:

$(x^*, Improv_l)$

Begin:

$x^* \leftarrow x$;
$f^* \leftarrow f(x)$;
$NumGridPoints \leftarrow \prod_{i=1}^{n} \lceil (u_i - l_i)/h \rceil$;
$MaxPointsToExamine \leftarrow \lceil \rho_{lo} \ . \ NumGridPoints \rceil$;
$NumPointsExamined \leftarrow 0$;
while $NumPointsExamined \leq MaxPointsToExamine$ **do**
  $NumPointsExamined \leftarrow NumPointsExamined + 1$;
  $x \leftarrow RandomlySelectElement(B_h(x^*))$;
  if $l \leq x \leq u$ and $f(x) < f^*$ **then**
    $x^* \leftarrow x$;
    $f^* \leftarrow f(x)$;
    $Improv_l \leftarrow$ True;
    $NumPointsExamined \leftarrow 0$;
  end if
end while
  return $(x^*, Improv_l)$

Algorithm 7 LocalSearch

Input:

x ▷ Solution x
$f(.)$ ▷ Objective function
n ▷ Dimension
h ▷ Grid density
l ▷ Lower Bounf
u ▷ Upper Bound
$MaxDirToTry$ ▷ Maximum distinct random directions

Output:

x^*

Begin:

$x^* \leftarrow x$;
$f^* \leftarrow f(x)$;
$NumDirTried \leftarrow 0$;
$S \leftarrow \{x :\| x^* - x \|^2 = h\}$;
while $NumDirTried < MaxDirTry$ **do**
  $NumDirTried \leftarrow NumDirTried+1$;
  $x \leftarrow$ RandomlySelectElement(S);
  if $l \leq x \leq u$ **then**
    if $f(x) < f^*$ **then**
      $x^* \leftarrow x$;
      $f^* \leftarrow f(x)$;
      $S \leftarrow \{x :\| x^* - x \|^2 = h\}$;
      $NumDirTried \leftarrow 0$;
    end if
  end if
end while
return x^*

Algorithm 8 Line Search

Input:

x ▷ Solution x
$f(.)$ ▷ Objective function
n ▷ Dimension
h ▷ Grid density
l ▷ Lower Bounf
u ▷ Upper Bound
k ▷ Index

Output:

z_k

Begin:

$t \leftarrow x$;
$z_k \leftarrow x_k$;
$minF \leftarrow f(x)$;
$t_k \leftarrow l_k$;
while $t_k \leq u_k$ **do**
  if $f(t) < minF$ **then**
    $minF \leftarrow f(t)$;
    $z_k \leftarrow t_k$;
  end if
  $t_k \leftarrow t_k + h$;
end while
$t_k \leftarrow u_k$;
if $f(t) < minF$ **then**
  $minF \leftarrow f(t)$;
  $z_k \leftarrow t_k$;
end if
return z_k

3 GRASP with path-relinking (GRASP+PR)

Algorithm 9 GRASP+PR

Input:

N ▷ N of facilities
M ▷ set M of locations
A ▷ Flow matrix A
B ▷ Flow matrix B
C ▷ Assignment cost matrix C
z ▷ Scaling factor z
q_i ▷ Facility demands q_i , $i \in N$ N
Q_j ▷ Location capacities Q_j , $j \in$ M

Output:

$\pi \in \chi$

Begin:

$P \leftarrow \emptyset$
while *stopping criterion not satisfied* **do**
  $\pi' \leftarrow$GreedyRandomized(.);
  if elite set P has at least ρ elements **then**
    if π' not feasible **then**
      Randomly select a new solution $\pi' \in P$
    end if
    $\pi' \leftarrow$ApproxLocalSearch(π');
    Randomly select a solution $\pi^+ \in P$
    $\pi' \leftarrow$ PathRelinking(π', π^+)
    $\pi' \leftarrow$ ApproxLocalSearch(π')
    if elite set P is full **then**
      if $c(x) \leq max\{c(\pi) \mid \pi \in P\}$ and $\pi' \not\approx P$ **then**
        Replace the element most similar to π' among all elements with cost worst than π';
      end if
    else if $\pi' \not\approx P$ **then**
      $P \leftarrow P \cup \{\pi'\}$
    end if
  else if π' is feasible and $\pi' \not\approx P$ **then**
    $P \leftarrow P \cup \{\pi'\}$
  end if
end while
return $\pi^* = min\{c(\pi) \in P\}$;

Algorithm 10 ApproxLocalSearch

Input:

π ▷ Local minimum

$MaxCLS$ ▷ Maximum size of the candidate list in the local search

$MaxItr$ ▷ Maximum number of iterations of neighbors sampled in local search

Output:

Approximate local minimum π

Begin:

$count \leftarrow 0$;
$CLS \leftarrow \emptyset$;
while $CLS \neq \emptyset$ **do**
  while $\mid CLS \mid \geq MaxCLS$ or $count \geq MaxItr$ **do**
    $\pi' \leftarrow Move(\pi)$
    if π' is feasible and $cost(\pi') < cost(\pi)$ **then**
      $CLS \leftarrow \cup\{\pi'\}$;
    end if
$count \leftarrow count + 1$;
  end while
  if $CLS \neq \emptyset$ **then**
    Randomly select a solution $\pi \in CLS$
  end if
end while
  return π;

Algorithm 11 PathRelinking

Input:

π_s ▷ Starting solution

π_t ▷ Target solution

η ▷ Candidate size factor

Output:

Best solution π^* in path from π_s to π_t

Begin:

$\pi^* \leftarrow argmin\{f(\pi_s), f(\pi_t)\}$;
$f^* \leftarrow f(\pi^*)$;
$\pi' \leftarrow \pi_s$;
$Fix \leftarrow \emptyset$;
$nonFix \leftarrow N$;
Compute difference $\varphi(\pi', \pi_t)$ between solution π' and π_t;
while $\varphi(\pi', \pi_t) \neq \emptyset$ **do**
$\beta \leftarrow \emptyset$;
for do$\forall v \in \varphi(\pi', \pi_t)$
Move the facility v in π' to the same location l assigned to v in π_t;
$\pi^- \leftarrow$ makeFeasible(π', v);
if π^- is feasible **then**
if $\mid \beta \mid \geq \eta. \mid \varphi(\pi', \pi_t) \mid$ **then**
if $c(\pi^-) \leq max\{c(\pi) \mid \pi \in \beta\}$ and $\pi^- \notin \beta$ **then**
replace the element most similar to π^- among all elements with cost worst than π^-;
end if
else if $\pi^- \notin \beta$ **then**
$\beta \leftarrow \beta \cup \{\pi^-\}$;
end if
end if
end for
if $\beta \neq \emptyset$ **then**
Randomly select a solution $\pi \in \beta$;
Compute difference $\varphi(\pi, \pi_t)$ between solution π and π_t;
Set $I = \varphi(\pi', \pi_t) \setminus (\varphi(\pi', \pi_t) \cap \varphi(\pi, \pi_t))$
Randomly select a facility $i \in I$;
$Fix \leftarrow Fix \cup \{i\}$;
$noFix \leftarrow noFix \setminus \{i\}$;
$\pi' \leftarrow \pi$;
if $f(\pi') < f^*$ **then**
$f^* \leftarrow f(\pi')$;
$\pi^* \leftarrow \pi'$;
end if
else
return assignment π^*;
end if
end while
return assignment π^*;

Algorithm 12 makeFeasible

Input:

π ▷ Solution
f ▷ Facility f
τ^- ▷ Maximum number of tries

Output:

If possible, a feasible solution

Begin:

if $\sigma_l \geq 0$ **then**
  return feasible solution π;
else
  $k \leftarrow 0$;
  while $k < \tau^-$ and $\sigma_l < 0$ **do**
    $\pi' \leftarrow \pi$
    while $T = \emptyset$ or $\sigma_l \geq 0$ **do**
      Set FTL $\subseteq$ nonFix to be all facilities in π' assigned to l;
      Set $T \subseteq$ FTL to be all facilities with demands less than or equal to the maximum slack in M;
      if $T \neq \emptyset$ **then**
        Randomly select a facility $i \in T$;
        Set $R \subseteq M$ to be all locations having slack greater than or equal to demand of facility i;
        Randomly select a location $j \in R$;
        Assign facility i to location $j : \pi'(i) \leftarrow j$;
      end if
    end while
    $k \leftarrow k + 1$;
  end while
  if $\sigma_l < 0$ **then**
    return infeasible solution π';
  else
    return feasible solution π';
  end if
end if