# 2021 EECBS

演变：

显式估计搜索EES+CBS --->> EECBS新的有界次优CBS变体；

使用在线学习来获得每个高级节点解决方案成本的不可接受估计，并使用 EES 来选择接下来要扩展的高级节点。

ECBS的有界次优性是通过用焦点搜索代替CBS的高和低级别中的最佳第一搜索来实现的，焦点搜索使用一种可接受的启发式方法来界定解决方案成本，使用另一种启发式方法来确定节点到目标节点的距离。

ECBS 和 EECBS 与 CBS 的不同之处仅在于它们在高级和低级搜索中使用的节点选择规则。

采用绕过冲突（Boyarski 等人 2015a）、冲突优先级（Boyarski 等人 2015b）、对称推理（Li 等人 2019c, 2020a），以及使用可接受的启发式来聚焦高级搜索（Li 等人 2019a）等方法对EECBS进行优化，经过改进的 EECBS 运行速度明显快于 ECBS、BCP-7（Lam 和 Le Bodic 2020）和 eMDD-SAT（Surynek 等人 2018），这两种算法也是目前最先进的有界次优 MAPF 算法。

针对ECBS的主要改进：

在外层选择节点时，加入使用在线学习估计节点选择时次优界范围的计算，解决了由于ECBS在FOCAL列表中选择节点时，按照冲突数量排序时，现有分支次优解决方案的节点代价超过次优下界，而不断在冲突数量最少的节点末端寻找解决方案，导致的耗时过大问题。

EECBS在节点选择时FOCAL层通过距离函数排序，可以使EECBS快速的切换搜索节点的分支，从而让EECBS更快的更新次优解下界（w\*LB）

伪代码：

生成CT根节点R;

计算R的WDG启发式;（计算启发值排序，cleanup按此排序，为了更快的提高次优下界）

将R推入节点列表;

当OPEN列表非空时，执行以下操作：

从节点列表中选择一个节点P;

如果P没有冲突

返回P的路径;

如果P是从CLEANUP列表中选择的，且尚未计算WDG启发式;

计算P的WDG启发式;

将P推入节点列表;

continue;

对P的冲突进行优先级排序;（如果是从cleanup中选择的节点，或者agent对中有一个agent的代价等于其下界（依然是最优路径，还未解过冲突）先解决）

对P的冲突进行对称性推理;（两个路径均为最优，且必然冲突）

从P的冲突中选择一个冲突;

解决该冲突，获得新的约束;

生成一个存放子节点的列表;

遍历新的约束；

生成子节点，并添加新的约束;

如果该子节点的成本在次优化界内，且冲突数减少

则用该子节点替换掉当前节点的路径及冲突；（旁路选择，如果冲突减少，且代价总和没有变化，用此子节点替换当前节点）

判断当前节点冲突是否为空；

返回P的路径;

将子节点加入列表;

将所有子节点加入队列中;

更新一步错误(可以看作根节点到此节点解决冲突数量的平均速度);

返回无解;

实验数据：

200 standard MAPF benchmarks (Stern et al. 2019)

参数及地图设置：

每个实例的时间限制为一分钟；

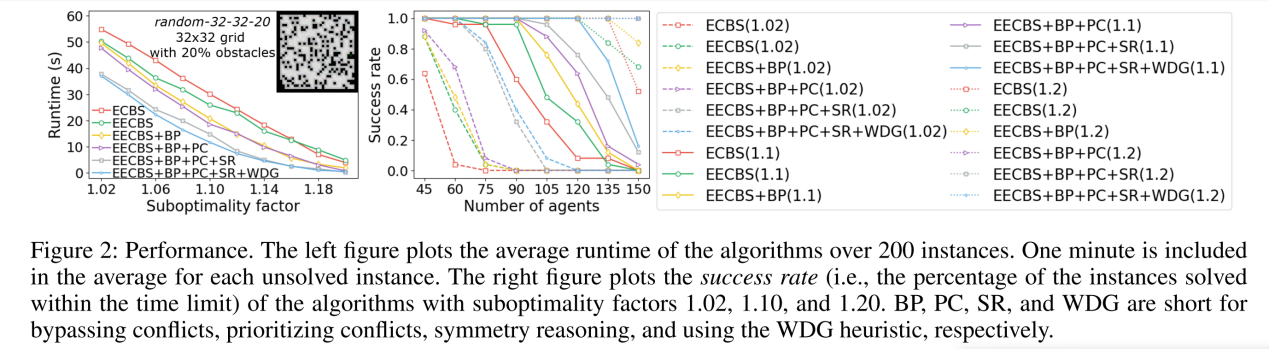
使用map random-32-32-20，一个32×32的4-邻居网格，具有20%的随机阻塞的障碍；

agent的数量从45到150不等，增量为15；

使用基准测试的“随机”场景，为每个数量的代理生成25个实例；

将次优因子从1.02变为1.20，增量为0.02；

实验结果



第一个图是次优因子对不同算法算法运行时间的影响；

第二个图是在一分钟之内不同算法在200个测试实例的运行成功率受agent数量、次优因子的影响。

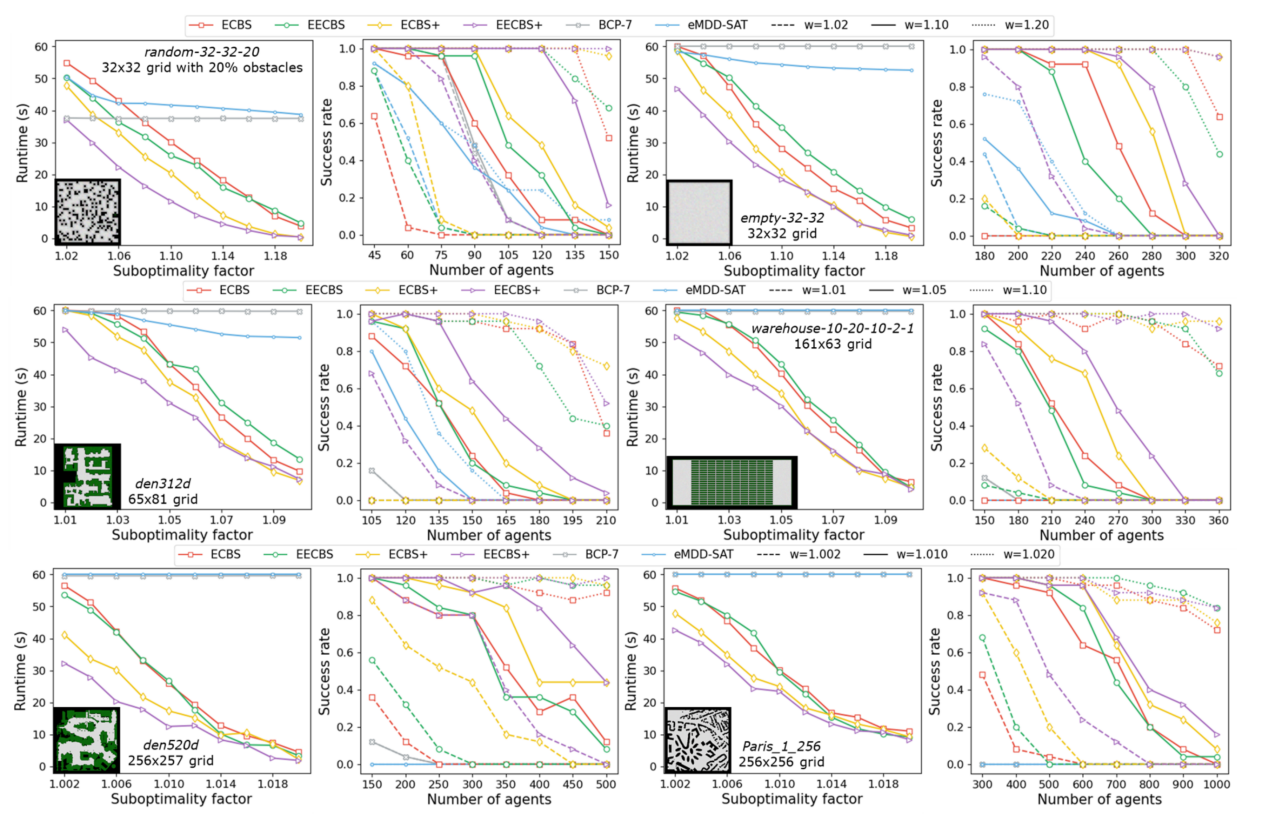
测试环境：

算法在C++中实现，实验在Ubuntu 20.04 LTS上在Intel Xeon 8260 CPU上进行，内存限制为16GB，时间限制为1分钟。

是否开源：

<https://github.com/Jiaoyang-Li/EECBS>.

测试结果：



测试数据：

6（地图数量）\* 8（种智能体数量）\* 25（个实例）\* 10（个w值，随着地图增大逐渐减小）

# 2022 MAPF-LNS2

实验环境：

语言：C++

硬件环境：AmazonEC2“m4.xlarge”， 16GB内存

运行时间限制：5分钟

算法思路：

初次规划使用PP算法，也可以用于加速EECBS

获取一个agent子集（选中一个agent，然后扩展与它有冲突的agent）

优化agent子集，减少子集内的冲突和子集与其他智能体组成集合的冲突

最后得到无冲突的解决方案

伪代码：

# 硬障碍

Oh = [...]

Os = [...]

initial\_plan = PP() # 调用MAPF算法生成初始规划路径P

# 构建一个冲突图

Gc = {

"agents": {agent1, agent2, agent3, agent4, ...},

"collision": [[agent1, agent2], [agent1, agent3]...]

}

# 初始化各个邻域选择方法的权重

w\_collision = 1

w\_failure = 1

w\_random = 1

# 经验参数，用来调整邻域选择方法的权重

gama = 0.1

# 邻域内的agent数量

N = 8

# 自适应邻域选择，根据权重选择不同的方法获取邻域As

while (存在冲突的路径对):

As = []

if w\_collision 最大, 使用基于冲突的邻域选择方法:

select\_method = "collision"

# 随机选取一个初始的agent

agent\_cen = random(Gc["agents"])

As.append(agent\_cen)

# 从agent\_cen开始递归寻找，把和它冲突的agent都找出来

agents = [...]

if len(agents) >= N - 1:

while len(As) < N:

# 从agent\_cen开始扩散寻找，将找到的agent都加入As中

As.append(agent in agents)

else:

# 从As中随机挑选一个agent，从其路径上随机的一点开始随机移动

while len(As) < N:

agent = random(As)

point = random(agent.path)

point[随机一个索引] += random((1, -1))

if point in Gc["agents"][不在As中的agent].path

As.append(Gc["agents"][any])

break

elif w\_failure 最大:

select\_method = "failure"

# 选择冲突最多的agent

agent\_cen = max(count(agent in Gc["collision"]))

As.append(agent\_cen)

# 将起点在agent\_cen中的agent加入A1

A1 = [aj in A | pj in P visits si]

# 将侵占A1中agent路径终点的agent加入A2中

A2 = [aj in A | p in P visits si]

if len(A1 ∪ A2) = 0:

# agent\_cen没有冲突

return As

elif len(A1 ∪ A2) < N - 1:

while len(As) < N:

As.append(agent(终点在As中随机选出的agent中))

else:

if len(A1) == 0:

As.extend(A2[: N - 1], random)

elif len(A2) > N - 1:

As.extend(A2[: N - 2], random)

else:

As.extend(A2)

As.extend(A1[: N - 1 - len(A2)].sort(by和agent\_cen起点冲突时间，升序))

else:

select\_method = "random"

As.append(agent in Gc["agents"], random(deg(i)与概率成正比))

# 构建安全区间表T，为每个顶点构建安全区间序列T[v]

T\_safe = buildSafeIntervalTable(As, Oh, Os)

# 为As中的代理重新规划路径

for agent in As:

# 初始化根节点root，以起点s的最早安全区间作为root的安全区间，id为1，表示为起点s的第一个安全区间

root = Node(start, T[s][1], 1, false)

use\_time = 0

# 如果终点处有硬障碍，硬障碍物存在的时间为use\_time的下界

if exists t使得(goal, t) in Oh:

use\_time = max{(goal, t) in Oh} + 1

g\_score = root.low

h\_score = 预估到达路径长度

f\_score = g\_score + f\_score

c\_value = 父节点的软碰撞数量(root节点为0) + 当前点的安全区间包含点障碍为1否则为0 + 父节点和此节点的最早到达时间相同为1否则为1

open\_list = queue(root)

close\_list = []

while open\_list:

pos\_node = open\_list.pop()

if pos\_node.is\_goal():

path = 从终点溯源父节点到起点返回完整路径

if pos\_node.v = goal and pos\_node.low > use\_time:

c\_future = 获取终点从最早到达时间之后还有没有软碰撞

if c\_future == 0:

return path

pos\_node\_copy = deepcopy(pos\_node)

pos\_node\_copy.is\_goal = True

pos\_node\_copy.c\_value = pos\_node.c\_value + c\_future

# 将节点pos\_node\_copy添加到open\_list和close\_list中(或移除)

insert\_node(pos\_node\_copy, open\_list, close\_list)

# 扩展结点

I = []

for v\_pair in E:

if pos\_node\_copy in v\_pair:

I = [(n.v, id) for v, safe\_interval\_list in T.items() if v的第id个安全区间和pos\_node\_copy的安全区间交集不为空]

if T[n.v][id].low = n.high:

I.append((n.v, id))

for (v, id) in I:

[low, high) = T[v][id]

# 给low重新赋值

low = 和硬障碍没有边冲突的最早到达时间

if low 不存在:

continue

low\_ = 和软障碍和硬障碍合集没有边冲突的最早到达时间

if low\_ and low\_ > low:

# 将v的安全区间分开插入

n1 = Node(v, [low, low\_), id, False) ????????????????

insert\_node(n1, open\_list, close\_list)

# 具体逻辑

# 计算n1的g\_score、h\_score、f\_score、c\_value

# X = [n\_i for n\_i in open\_list + close\_list if n\_i.v == n1.v and n\_i.id == n1.id and n\_i.is\_goal == # n1.is\_goal]

# for n\_i in X:

# if n\_i.low <= n1.low and n\_i.c\_value <= n1.c\_value:

# return # 不需要添加结点，应为n\_i节点表现优于n1

# elif n1.low <= n\_i.high and n1.c\_value <= n\_i.c\_value:

# delete n\_i from open\_list and close\_list # n1表现更优

# else：

# # 如果n1和n\_i的区间有重合, 去掉重复空间，避免重复搜索

# if n1.low < n\_i.high and n\_i.low < n1.high:

# if n1.low < n\_i.low

# n1.high = n\_i.low

# else:

# n\_i.high = n1.high

n2 = Node(v, [low\_, high), id, False)

insert\_node(n2, open\_list, close\_list)

else:

n3 = Node(v, [low, high), id, False)

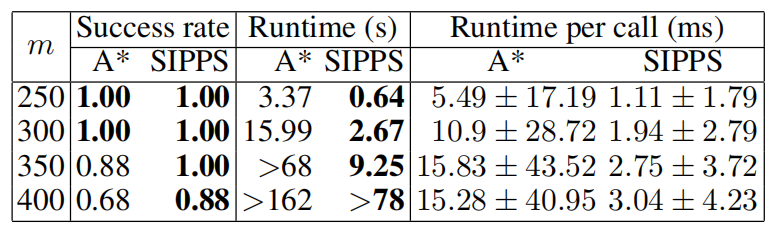
insert\_node(n3, open\_list, close\_list)

实验数据来源均为：MAPF benchmark suite

实验1数据及结果：

地图数据：random-32-32-20

Agent数量：400

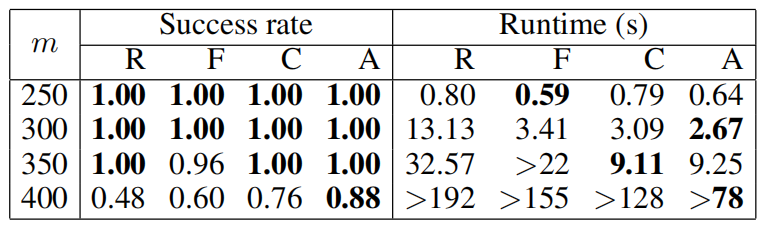


此实验反映了分别以A\*和SIPPS为单agent规划的MAPF-LNS2算法在求解时，规划出最小碰撞路径的成功率，整体的运行时间，单次调用时间的对比

实验2数据及结果：

地图数据：random-32-32-20

Agent数量：400

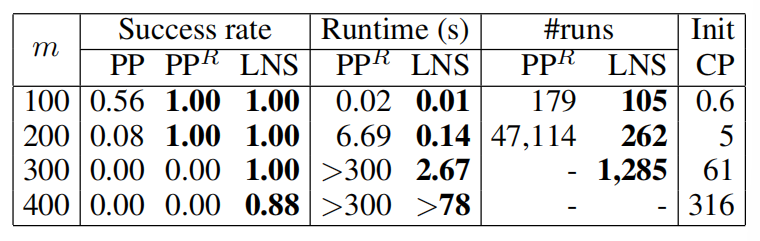


此实验反映了MAPF-LNS2算法在采用不同的邻域选择方法时，求解成功的机率及运行时间的对比

实验3数据及结果：

地图数据：random-32-32-20

Agent数量：400



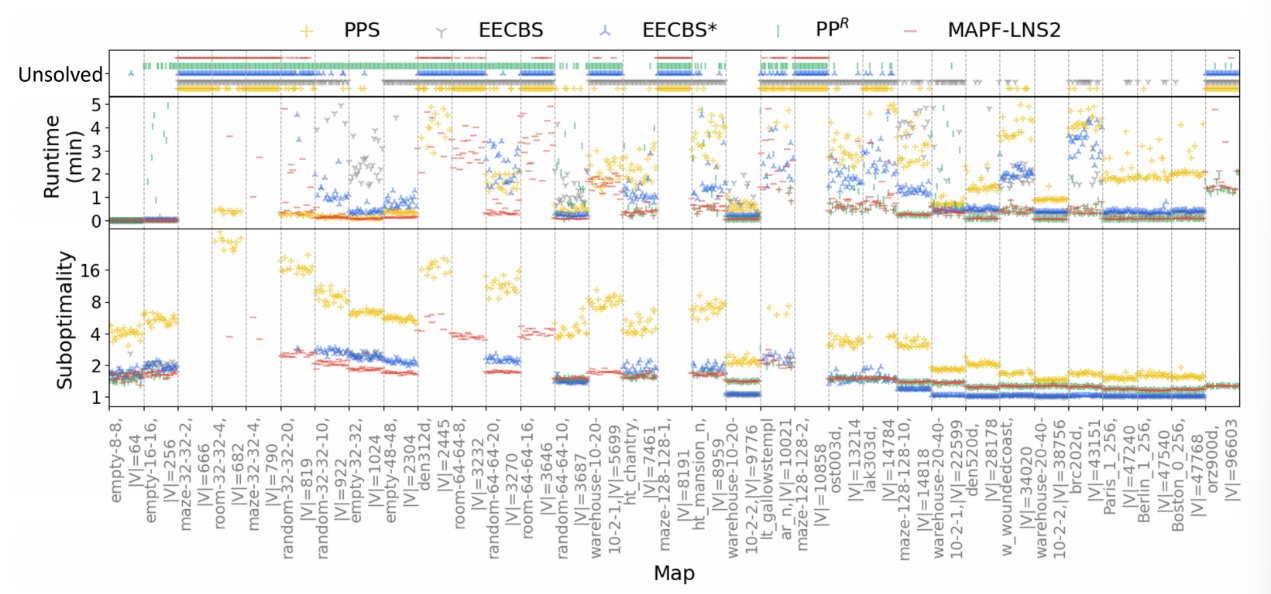
此实验反映了PP，PPR，LNS2三种算法在不同数量agent求解中的成功率、运行时间及SIPP(S)运行次数的对比（init cp为初始cp的平均值）

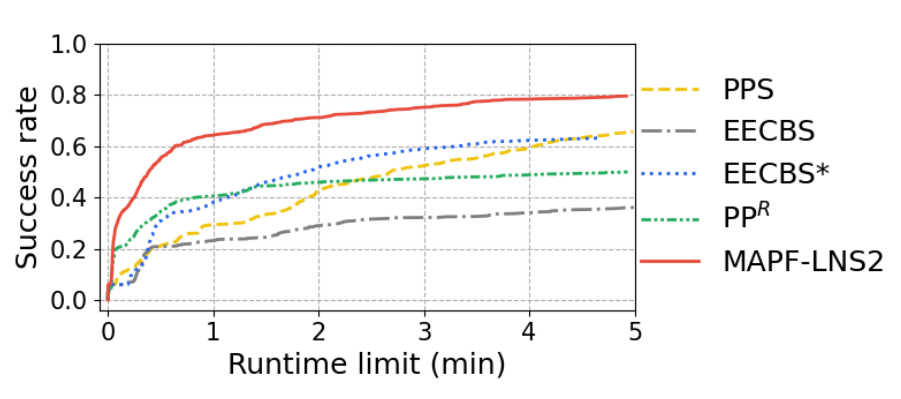
实验4数据及结果：

33（地图数量）\* 25（实例个数）

agent数量：min{0.5|V |, 1000}

运行时间限制：5分钟





算法的配置情况：

MAPF-LNS2：PP（初次规划）+ ALNS（自适应邻域选择N=8）+ SIPPS（单路径规划）

PPR：PP（加入随机重启）+ SIPP

PPS：PP + SIPP

EECBS：EECBS + A\*

EECBS\*：EECBS + SIPPS

此实验反映了5种算法在大量agent场景中的求解效果及运行时间

在成功率上MAPF-LNS明显优于其他算法；

MAPF-LNS2未能解决的一些实例大多在高度拥挤的地图上；

PPS可以解决一部分MAPF-LNS2未解决的实例，但解决方案的质量（代价）总是比使用space-time A\*的MAPF-LNS2算法（及其他算法）差；

某些情况下EECBS\*算法找到的解决方案质量更好，但是运行时间也总是最长的；

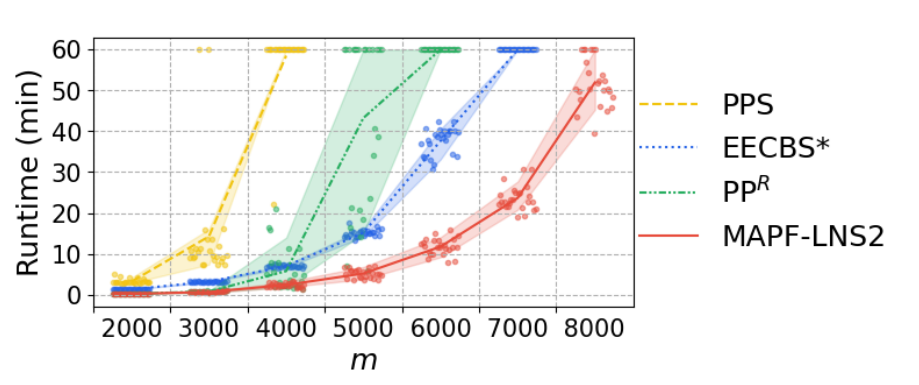
PPS和EECBS（\*）的内存使用量随着时间的推移而快速增加（因为它们生成越来越长的路径或越来越大的搜索边界），PPR和MAPF-LNS2的内存使用率保持稳定；

实验5数据及结果：

实验数据：

地图数据：warehouse-20-40-10-2-2

Agent数量：2000-8000



此实验反映了四种算法在大量代理情况中的求解能力

是否开源：

https://github.com/Jiaoyang-Li/MAPF-LNS2

# 2023 WDG+CHBP

实验环境：

C++，-O3编译

Nectar research cloud with 128GB of RAM running Ubuntu 18.04.4 LTS (Bionic Beaver)

算法伪代码：

CBS+A\*构建初始路径及根CT节点

root\_CT = {

constraints

paths

conflicts

cost

clusters

}

WDG\_sub\_cbs\_node\_num = L

open\_list = [root\_CT] # 使用新的启发值计算排序

while open\_list不为空:

N = open\_list.pop()

if 节点存在冲突:

hp, EAp = 计算WDG启发值(N)

hc, EAc = 从父结点中继承启发值及冲突簇(N) # （Memoization）

EA = EAp + EAc

SG = 构建冲突状态图(N)

while am <- 获取冲突最多的结点（agent）:

C = {am}

PA = []

pm = getpath(am, N) # 获取am当前规划路径

MDDm = getMDD(am, N) # 获取am的MDD（am所有和pm时间步相同的到达路径图）

CA = GETCONFLICTAGENTS(am, pm, N, EA) # 获取所有和am路径存在冲突的agent

for ac in CA:

PA = PA + ac

MDDc = getMDD(ac, N)

M = mutexpropagation(MDDm, MDDc) # 互斥传播，找到所有两个MDD中的所有互斥点

if M is empty:

C = C + {am}

MDDm = MDDm - M # 去掉MDDm中的互斥节点

if MDDm is empty:

R = C # 新的冲突簇

break

if pm 在MDDm中不成立: # 此操作每次通过中心agent寻找冲突簇只执行一次

pm = getminconflictpath(MDDm) # 从去掉互斥结点的MDDm中获取最小冲突路径

CA = GETCONFLICTAGENTS(am, pm, N, EA) # 由于am路径变化重新获取冲突agent

R = 旁路节点(pm)

if R是冲突簇

APPENDCLUSTER(N, C) # 将冲突簇添加到当前节点中

EA = EA + C # 将找到的冲突簇中的所有代理添加到已排除代理中

hc += 1

else R是旁路结点（am的新最小冲突路径）

更新N中的am路径和冲突

更新冲突状态图

N\_new = deepcopy(N)

N\_new.cost = N.cost + hp + hc

open\_list.push(N\_new)

实验1数据及结果：

地图数据：

random-32-32-20，agent\_num：20，30，…，70

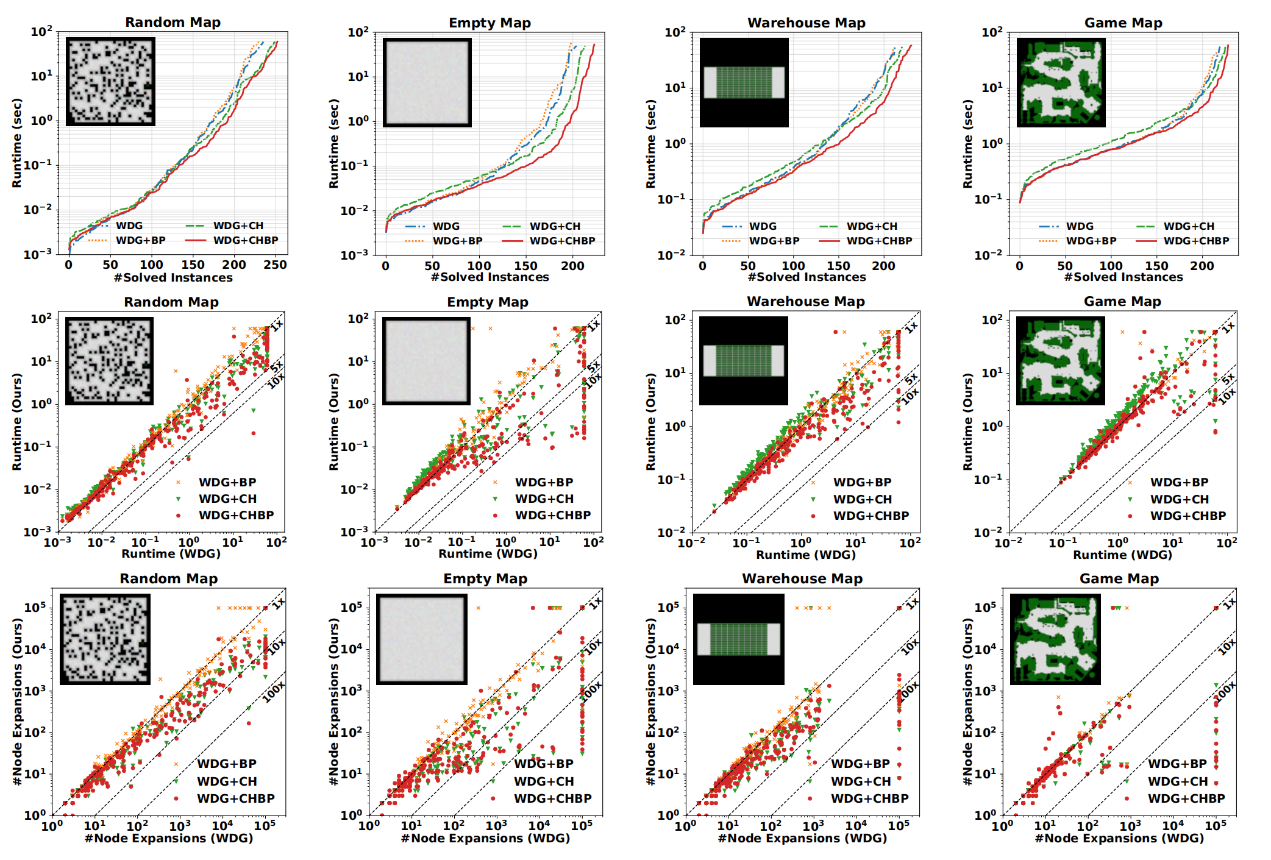
empty 32×32，agent\_num：50，70，…，150

warehouse-10-20-10-2-1，agent\_num：30，50，…，130

10\*20个堆栈，每个堆栈为10\*2

game map 256×257，agent\_num：40，60，…，140

算法限制时间：1分钟



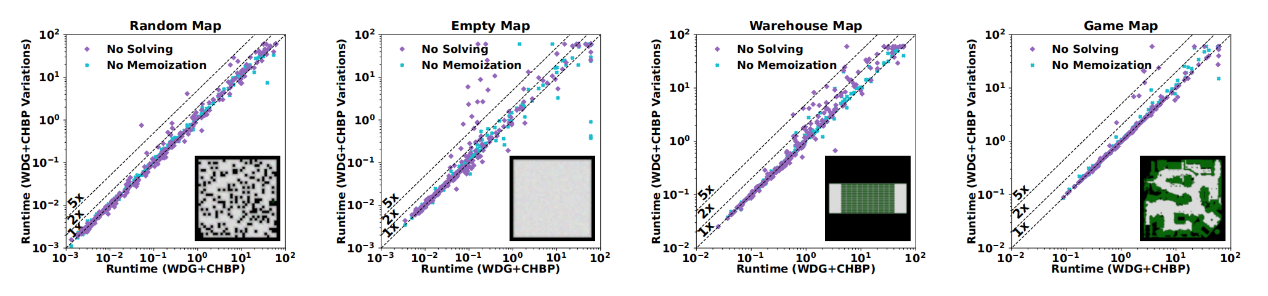
因为WDG+CHBP主要受益于启发值的改进，所以WDG+CH比WDG有所改进，WDG+BP解决困难实例的能力不足；

明显的效果改善发生在agent数量增多之后；

总体上WDG、WDG+BP、WDG+CH和WDG+CHBP分别解决875、859、896和937个实例

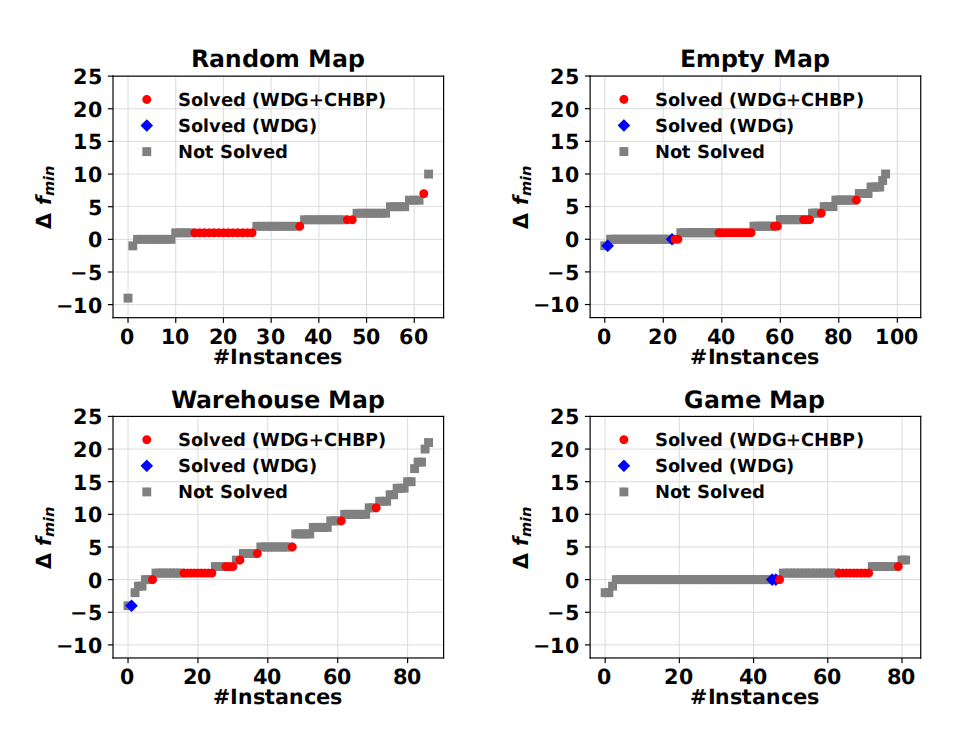
改进后扩展的节点更少。

实验2结果：



此实验反映的是最终版本与之前的未添加优化的版本（无解算，无记忆化）运行时间的对比，证明了算法提出的操作时序的有效性，尤其是聚类进行解算

实验3结果：



此实验反映的是算法终止时，open\_list中的最小f\_value=N.cost+h，图中的结果尤其是在empty\_map和warehouse\_map地图上，表明WDG+CHBP与WDG相比，聚类启发式能够在多大程度上提高搜索进度。

开源状态：

https://github.com/bshen95/CBSH2-RTC-CHBP

现有的MAPF算法包括：

系统优化搜索算法（是指数时间，但保证找到最优或有界次优解）

CBSH2-RTC、Lazy CBS、BCP、ECBS、EECBS

运行时间为agent数量的指数级（适合少量agent规划）

基于规则的算法（通常是多项式时间和完整的）

PPS、PIBT、WSCaS

优先算法（运行速度快，但既不完整也不最优）。

prioritized algorithms、PBS、

对冲突的定义：

原文：A collision happens when two agents occupy the same vertex or traverse the same edge in opposite directions at the same timestep.

译文：当两个代理在同一时间步长占据同一顶点或沿相反方向遍历同一边缘时，就会发生冲突。