基于 MB-RRT* 的无人机航迹规划算法研究

陈晋音 施 晋 杜文耀 吴洋洋
(浙江工业大学信息工程学院 杭州 310023)

摘 要 随着小型无人机的广泛应用,提高无人机的自动巡航能力变得至关重要。无人机航迹规划是指其在已知环境地图信息下展开航迹规划,实现无碰撞的、平滑的、从初始点到达目标点的路径。针对现有算法依然存在收敛速度慢、内存消耗大、航迹规划固定步长和航迹平滑度无法满足实际无人机飞行等问题,提出了 MB-RRT*(Modified B-RRT*)算法,通过懒惰采样方法加快算法收敛速度并减少内存占用;设计自适应步长来解决算法在障碍物附近生长树的局限性问题,从而提高了找到初始可行解的速度和质量;然后利用降采样和3次贝塞尔插值算法实现了曲线拟合的功能,使算法最终生成相对平滑的航迹,为无人机实际飞行提供可行的航迹规划方法。最后在多组不同环境复杂度的实验中,通过与其他算法相比较,验证了所提算法的有效性。

关键词 RRT,无人机,航迹规划,收敛速度,懒惰采样
中图法分类号 TP305 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.08.035

MB-RRT* Based Navigation Planning Algorithm for UAV

CHEN Jin-yin SHI Jin DU Wen-yao WU Yang-yang
(College of Information Engineering, Zhejiang University of Technology, Hangzhou 310023, China)

Abstract With the wide application of unmanned aerial vehicle(UAV), it is important to improve the capacity of automatic navigation. Navigation for UAV is the algorithm that can automatically find out the obstacle-free, smoothing path from start position to target position. Most current navigation algorithms for UAV still have shortcomings including low convergence speed, large memory cost, fixed navigation step setting and smoothing challenges. MB-RRT* algorithm was proposed in this paper which has three outstanding strategies for UAV. Lazy sampling was adopted to improve convergence speed and achieve less memory cost. Self-adaptive step length algorithm was applied to solve navigation limitation near obstacles and improve initial solutions' quality and speed. Down sampling and curve fitting were introduced to improve the convergence rate of the algorithm and the smoothness of the final path. Finally abundant simulations were carried out to testify the high performances of MB-RRT* compared with RRT* and BRRT*.

Keywords RRT, UAV, Navigation planning, Convergence speed, Lazy sampling

1 引言

航迹规划是无人机导航和机器人技术中的一个重要问题,其基本定义为:给定一个初始状态和一个目标状态,寻找一个可行航迹使无人机无碰撞地从初始状态运行到目标状态。航迹规划有广泛的应用场景:GPS导航、无人驾驶汽车、计算机动画、路由问题、制造行业的机械臂运动以及生活和工业领域的很多方面。因此近年来对航迹规划问题的研究成为了一个研究热点。

无人机航迹规划算法根据其感知能力可以分为局部航迹规划和全局航迹规划,其中全局航迹规划就是在已知环境地图的情况下进行规划,预先了解环境的全局信息;而局部航迹规划只需要获得机器人感知范围内的环境信息,主要指障碍物信息,根据局部信息完成规划。全局航迹规划算法有很多,人工势场算法是典型的航迹规划算法[1],该算法在环境中建立人工势场,障碍物和环境边界具有斥力,目标区域具有引力,无人机根据所受力向目标区域靠近。势场法不需要进行复杂的计算,只需要计算出环境的势场即可,但在复杂环境中容易使飞行器陷入局部最小,并不适合在复杂环境以及狭窄的通道中进行规划。针对势场法的不足,学者们提出了Dubins曲线算法以及细胞分裂算法[2]和Delaunay三角算法等离散化搜索空间的算法[3],通过对障碍物或者环境空间进行建模的方法寻找最优航迹。同时也有人将进化计算(如遗传算法[4]、粒子群算法[5])用于解决航迹规划问题,利用算法的进化操作和迭代过程找到最优航迹。然而这类算法的计算开销特别大,算法在复杂的环境以及高维的环境中需要大量的

到稿日期:2016-06-13 返修日期:2016-10-24 本文受浙江省自然科学基金(Y14F020092),国家自然科学青年基金(61502423),浙江省科技厅科研院专项(2016F50047)资助。

陈晋音(1982-),女,副教授,主要研究方向为数据挖掘、智能计算等,E-mail:chenjinyin@zjut.edu.cn(通信作者);施 晋(1994-),男,硕士生,主要研究方向为数据科学;杜文耀(1989-),男,硕士生,主要研究方向为计算机视觉;吴洋洋(1994-),男,硕士生,主要研究方向为数据挖掘及其应用。

计算时间，无法直接应用在无人机的航迹规划中。

基于采样的航迹规划算法已被证明可以高效地解决航迹规划问题[6]，概率路线图算法（Probability Roadmap Method，PRM）[7]和快速扩展随机树算法（Rapidly-exploring Random-Tree，RRT）[8]是目前两种主要的采样算法。PRM 算法随机在空间生成采样点，并对这些点进行连接，最后通过图搜索算法找到从初始状态到目标区域的航迹。与 PRM 算法相比，RRT 算法采用树结构描述碰撞检测的次数，并且树的航迹搜索比图的航迹搜索更容易实现。然而 RRT 算法的收敛率太低，即需要通过大量的迭代才能找到最优航迹，并且随着迭代次数的上升，算法也需要大量的内存。因此，人们提出了很多针对 RRT 算法的变种算法以及改进算法。Nik A 将粒子滤波与 RRT 算法相结合，提出了 PRRT 算法[9]用于局部航迹规划；Stephen R 将泰森多边形（Voronoi）引入树的生长中，提高了 RRT 找到可行解的速度[10]。其中应用最广且效果最好的是 Sertac Karaman 提出的 RRT* 算法[11]。RRT* 算法在每次迭代后对新加入的节点及其邻近节点进行优化，这一优化操作改善了算法的收敛率，保证了算法的渐近最优性，从而使其广泛应用于航迹规划领域并衍生出了一系列变种算法。Qureshi A H 为了加快 RRT* 算法的收敛速度，在生成随机点的同时将随机点以及目标点和初始位置 3 个点构成的三角形的内心作为新的随机点加入树中，使随机点在一定程度上偏向了目标点[12]；Jordan M 提出了利用两棵树生长寻找航迹的方法，其提高了算法的收敛率[13]。这些改进算法在实现无人机的航迹规划时仍然存在以下问题：

（1）算法的收敛速度还有很大的提升空间；

（2）算法寻找最优航迹需要进行大量的迭代，因此算法运行时需要较大内存开销；

（3）算法的节点基于固定的步长生长，因此在障碍物附近生长的树具有局限性；

（4）由于算法的航迹由树节点连接生成，最后生成的航迹不够平滑，难以满足无人机实际应用。

针对这些问题，本文提出了一种 MB-RRT*（Modified B-RRT*）算法，通过懒惰采样的方法提升了算法的收敛率并且减少了算法的内存占用；采用自适应步长的方法解决算法在障碍物附近生长树的局限性问题，并且提高了算法找到初始可行解的速度和质量；利用了降采样和三次贝塞尔插值算法实现了曲线拟合的功能，使算法生成相对平滑的航迹，最后通过相关实验验证了算法的有效性，并通过与其他算法比较对 MB-RRT* 算法的性能进行了验证。

2　相关算法研究

2.1　RRT

RRT 算法是由 LaValle S 和 Kuffner J 首先提出，并且被证明具有概率完备性，同时可以高效地解决航迹规划问题[14]。RRT 算法首先在无障碍状态空间中进行随机采样，生成采样点后寻找树中距离采样点最近的节点，对这两点之间的连线进行碰撞检测，判断连线中是否有障碍物，如果没有则将这个新的采样点加入树中，通过不断地迭代生长树直到树生长到目标区域，最后生成最优航迹。

由于 RRT 算法的采样是随机采样，这使得算法产生的树节点随着迭代次数的上升覆盖了整个地图区域，保证了算法的概率完整性，而且算法只需经过采样、碰撞检测、连接这 3 个过程，不需要进行过于复杂的计算。然而 RRT 算法的缺点也很明显，因为算法需要经过多次迭代才能找到最优解，所以收敛率太低，且算法在插入随机采样点之后并没有其他修正操作，导致算法需要进行大量的迭代才能找到相对最优的航迹，这需要消耗大量的时间和内存。

2.2　RRT* 算法

针对 RRT 算法收敛率低等问题，Karaman S 和 Frazzoli E 对 RRT 算法的连接过程进行了改进，提出了 RRT* 算法[15]。与 RRT 算法的直接连接过程不同，RRT* 算法在添加节点的同时对采样点周围的节点进行了检测，判断是否在加入新节点后有更短的航迹，若有则用新节点替换旧节点，从而对航迹进了修正。

RRT* 算法的主要步骤如算法 1 所示。

算法 1　RRT* (x_{start}, x_{goal})

```
1. V←x_start; E←∅; T←(V,E); i←0
2. while i < N do
3.   x_rand←Sample(i)
4.   x_nearest←NearestNode(x_rand, T)
5.   x_new←Steer(x_nearest, x_rand)
6.   if CollisionCheck(x_nearest, x_new) then
7.     T←InsertNode(x_new)
8.     x_near←NearNodes(x_new, T)
9.     x_parent←ChooseBestParent(x_new, x_near, T)
10.    OptimizeVertices(x_parent, x_new, T)
11.  end if
12. end while
```

上述算法中各函数的功能如下。

Sample：该函数在无障碍环境中随机生成一个状态点。

Steer：给定两点 x_1，x_2，函数 *Steer* 返回点 x_{new}，点 x_{new} 满足 $\| x_{new}-x_1 \| <\mu$ 并且使 $\| x_{new}-x_2 \|$ 最小，其中 μ 为树生长的步长。

NearestNode：给定一个状态点 $x\in X_{free}$ 以及一棵树 $T=(V,E)$，函数 *NearestNode*(x,T) 在树 T 中搜索与点 x 欧氏距离相距最近的节点 $x_{nearest}$ 并返回。

NearNodes：给定一个状态点 $x\in X_{free}$ 以及一棵树 $T=(V,E)$，函数 *NearNodes*(x,T) 返回一个与点 x 的欧氏距离小于 γ 的点集 $V_1\in V$。其中 γ 的大小为 $k(\log n/n)^{1/d}$，k 为常数，n 为算法的迭代次数，d 为所在环境的维数。这一过程可以简化为：

$$V_1=\{x_{near}\in V: d(x,x_{near})\leqslant\gamma\}$$

CollisionCheck：给定两个点 x_1，$x_2\in X_{free}$，函数 *CollisionCheck*(x_1,x_2) 对两点之间的连线航迹进行障碍检测，若两点之间无障碍则返回 True，否则返回 False。

函数 ChooseBestParent 的主要步骤如算法 2 所示。

算法 2　ChooseBestParent(x_{new}, X_{near}, T)

```
1. x' = x_nearest
2. for all x_near ∈ X_near do
3.   if CollisionCheck(x_near, x_new) then
4.     c' = c(x_start, x_near) + c(x_near, x_new)
5.     if c' < c(x_new) then
6.       x' = x_near
7.       c(x_new) = c'
8.     end if
9.   end if
```

10. end for
11. return x′

函数 OptimizeVertices 的主要步骤如算法 3 所示。

算法 3 OptimizeVertices(x_{parent}, X_{near}, T)

1. for all $x_{near} \in X_{near}$ not x_{parent} do
2. if CollisionCheck(x_{near}, x_{new}) and c(x_{start}, x_{new})+c(x_{new}, x_{near})< c(x_{near}) then
3. x_p←Parent(x_{near})
4. T←DeleteEdge(x_p, x_{near})
5. T←AddEdge(x_{new}, x_{near})
6. end if
7. end for

RRT* 算法解决了 RRT 算法生成航迹不是最优的问题，保留了 RRT 算法的概率完整性；且相比于 RRT 算法，RRT* 算法具有较快的收敛速度，即可以通过更少的迭代次数找到最优解。RRT* 算法仍然有很多待解决的问题：RRT* 收敛到最优解的速率仍旧不够高，算法寻找最优解仍需进行大量的迭代过程，这也导致 RRT* 算法在迭代求解的过程中需要消耗大量的内存；同时，RRT* 算法在面对复杂的地图（如通道和迷宫）时需要进行大量的迭代才能找到最优解。

2.3 B-RRT* 算法

针对 RRT* 算法的问题，不少学者对其进行了研究并提出了一系列解决方案。Qureshi A H 提出了 TG-RRT*（Triangular Geometerised RRT*）算法[16]，该算法较 RRT* 算法有更高的收敛率，然而在一定程度上损失了算法的概率完整性，使得树的生长具有一定偏向性且容易使算法陷入局部最优。Jordan M 和 Perez A 提出的 B-RRT*（Bidirectional RRT*）算法在提升了算法收敛率的同时保留了算法的概率完整性以及渐近最优性。Carsten J 等人利用这一特性进行三维插值函数的推导，展示了其优势和实时性能[17]。B-RRT* 算法在 RRT* 算法的基础上引入了双向树结构，即从初始点和目标点同时生长两棵树，在加入新的节点之后进行一次连接检测，以判断两棵树能否相连。B-RRT* 算法的主要步骤如算法 4 所示。

算法 4 B-RRT* (x_{start}, x_{goal})

1. E←Ø; T_a← (x_{start}, E); T_b← (x_{goal}, E)
2. i←0; θ_{best}←∞
3. while i < N do
4. x_{rand}←Sample(i)
5. $x_{nearest}$←NearestNode(x_{rand}, T_a)
6. x_{new}←Steer($x_{nearest}$, x_{rand})
7. if CollisionCheck($x_{nearest}$, x_{new}) then
8. T_a←InsertNode(x_{new})
9. X_{near}←NearNodes(x_{new}, T_a)
10. x_{parent}←ChooseBestParent(x_{new}, X_{near}, T_a)
11. OptimizeVertices(x_{parent}, x_{new}, T_a)
12. x_{mid}←NearestNode(x_{new}, T_b)
13. θ'←Connect(x_{new}, x_{mid}, μ)
14. if $\theta' < \theta_{best}$ then
15. $\theta_{best} = \theta'$
16. end if
17. end if
18. SwapTrees(T_a, T_b)
19. end while

可以总结得到：算法采用双向树的结构需要额外的 *Connect* 函数对两棵树进行连接，*Connect* 函数应用了 RRT* 算法寻找新节点时所用的策略，调用了 *NearNodes* 函数在 $k(\log n/n)^{1/d}$ 范围内对另一棵树的节点进行搜索计算，从邻近节点中搜索最优的连接节点并连接。

函数 *Connect* 的主要步骤如算法 5 所示。

算法 5 Connect(x_{new}, x_{mid}, μ)

1. $\theta = \infty$
2. if c(x_{new}, x_{mid}) $< \mu$ then
3. θ=c(x_{start}, x_{new})+c(x_{new}, x_{mid})+c(x_{mid}, x_{goal})
4. end if
5. return θ

3 基于 MB-RRT* 的航迹规划算法设计

3.1 主要思路

虽然 B-RRT* 算法通过利用双向树同时生长提高了算法的收敛率，并且保留了 RRT* 算法的概率完整性，但是 B-RRT* 算法仍然有很多缺点需要改善。首先，B-RRT* 算法保留了 RRT* 算法以固定步长生长树的方式，较大的步长可以较快地生长，然而生长至障碍物附近的节点有较大概率由于无法通过碰撞检测而被抛弃；同时，较小的步长虽然在障碍物附近生长较好，但在空旷的环境地图中生长相对较慢。其次，B-RRT* 算法在找到可行航迹之后仍然对整个环境空间进行采样，这种采样策略容易产生大量无效节点。最后，B-RRT* 算法保留了 RRT* 算法将最后生成的航迹点直接相连的方法，从而生成最终航迹，导致最后产生的航迹过于粗糙，无法应用于实际情况中。针对 B-RRT* 算法的这些问题，本文提出了 MB-RRT* 算法，通过引入自适应步长、懒惰采样以及贝塞尔曲线优化等方法对 B-RRT* 算法进行改进优化。

3.2 算法设计与步骤

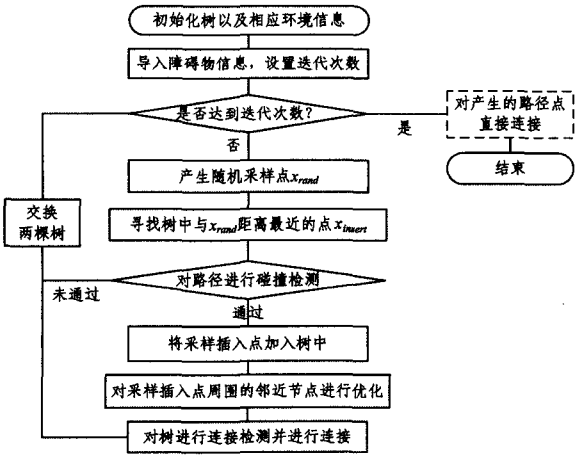
根据对 B-RRT* 的分析，本文设计的 MB-RRT* 从 3 个方面对 B-RRT* 进行改进。改进后的 MB-RRT* 算法的主要步骤如算法 6 所示。

算法 6 MB-RRT* (x_{start}, x_{goal})

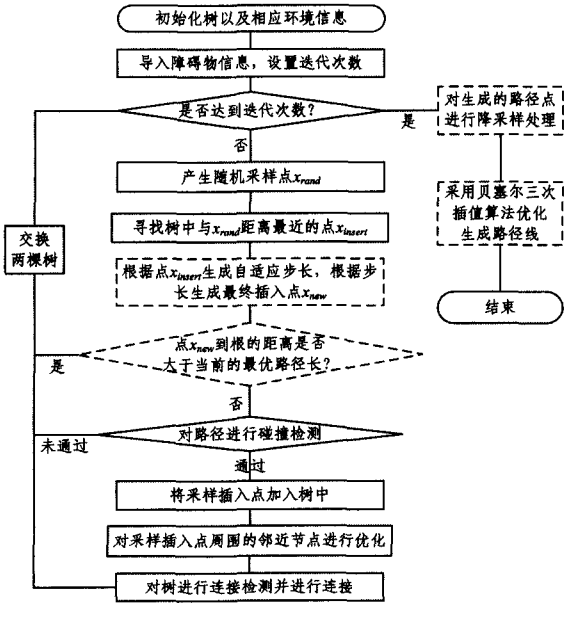
1. E←Ø; T_a← (x_{start}, E); T_b← (x_{goal}, E)
2. i←0; θ_{best}←∞
3. while i<N do
4. x_{rand}←Sample(i)
5. $x_{nearest}$←NearestNode(x_{rand}, T_a)
6. x_{new}←Steer($x_{nearest}$, x_{rand})
7. if c(x_{nearst} + c($x_{nearest}$, x_{new})) >θ_{best} then
8. Continue
9. end if
10. if CollisionCheck($x_{nearest}$, x_{new}) then
11. T_a←InsertNode(x_{new})
12. X_{near}←NearNodes(x_{new}, T_a)
13. x_{parent}←ChooseBestParent(x_{new}, X_{near}, T_a)
14. OptimizeVertices(x_{parent}, x_{new}, T_a)
15. x_{mid}←NearestNode(x_{new}, T_b)
16. θ'←Connect(x_{new}, x_{mid}, μ)
17. if $\theta' < \theta_{best}$ then
18. $\theta_{best} = \theta'$
19. end if

20. end if
21. SwapTrees(T_a, T_b)
22. end while
23. θ_{best}←DownSample(θ_{best})
24. θ_{best}←BezierCurve(θ_{best})

图 1 所示为 B-RRT* 算法与 MB-RRT* 算法的流程图对比，使用虚线框图展示两个算法的不同部分。可以看出，与 B-RRT* 算法相比，MB-RRT* 算法在每次迭代过程中增加了两个步骤，并且在算法迭代结束后又加入了两个处理方法。



(a)B-RRT* 算法流程图



(b)MB-RRT* 算法流程图

图 1　B-RRT* 与 MB-RRT* 算法的流程图对比

从图 1 可以看出，针对 2.3 节中提出的 B-RRT* 算法的问题，MB-RRT* 算法加入了一系列操作对这些问题进行处理和优化。针对第 1 节中普遍存在的问题(1)和问题(3)，MB-RRT* 算法引入了自适应步长的方法来优化算法的收敛率；针对问题(1)和问题(2)，MB-RRT* 算法引入了懒惰采样的方法在提高算法收敛率的同时减少了算法内存占用；针对问题(4)，MB-RRT* 算法抛弃了 B-RRT* 算法直接连接航迹点的过程，通过引入降采样和贝塞尔三次插值算法使最终生成的航迹更加平滑。

下面分别对 3 个改进策略展开解释。

3.2.1　自适应步长

传统的 B-RRT* 算法以固定的步长 μ 对树进行生长，μ 过小会导致树生长过慢，找到初始可行解的速度降低；μ 过大会导致生长过程中的采样点很难通过碰撞检测，使采样效率低下，且过大的步长会使得树的生长很难通过类似窄道的环境。

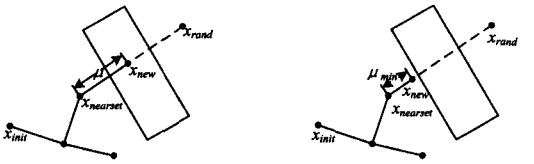
针对固定步长对树的生长造成的问题，MB-RRT* 算法引入了自适应步长的概念，并通过函数 *AutoStepSteer*(x_1, x_2)实现了树生长过程中的自适应步长。

传统的 *Steer*(x_1, x_2)函数以矢量 x_2-x_1 为方向，距离为步长 μ 的位置生成点 x_{new}，而 *AutoStepSteer*(x_1, x_2)函数首先通过 *NearestOb*(x_1, $2\mu_{Max}$)寻找并计算出距离点 x_1 最近的障碍物的距离 D_i，根据式(1)计算出步长。

$$StepSize=\frac{D_i}{\mu_{Max}}\mu_{Min} \tag{1}$$

通过这一方法计算出的步长保持在范围 $\mu_{Min}<\mu<\mu_{Max}$ 内，点 x_1 距离障碍越近，步长越接近 μ_{Min}，当以 x_1 为中心、μ_{Max} 为半径的范围内没有障碍时，步长为 μ_{Max}。

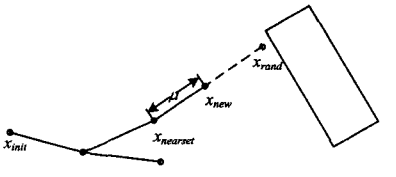
图 2 所示是自适应步长生长与传统固定步长生长过程的区别。图 2(a)为传统的节点靠近障碍物的情况下的生长，可以看出传统固定步长的生长方法易导致生成的新节点 x_{new} 进入障碍物范围而无法通过障碍检测；而图 2(b)所示为通过本文提出的自适应步长生长的节点可以在障碍物附近生成有效的采样点。



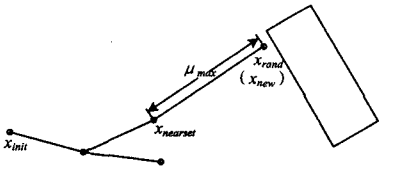
(a)靠近障碍物情况下的固定步长生长　　(b)自适应步长生长

图 2　固定步长与自适应步长生长过程的区别

在远离障碍物的情况下，传统固定步长生长的过程由于受限于步长，其在障碍物空旷的区域生长得较为缓慢(按原有步长生长)。而本文提出的自适应步长的生长方式按照计算的步长的最大值 μ_{max} 生长，加快了空旷区域探索速度，减少了找到初始可行解的时间，后面的实验结果同样证实了这一点。在远离障碍物情况下的固定步长生长与自适应步长生长的区别如图 3 所示。



(a)远离障碍物情况下的固定步长生长



(b)自适应步长生长

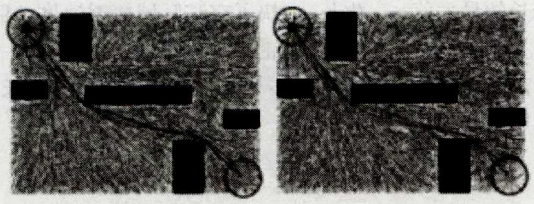
图 3

3.2.2 懒惰采样

传统 B-RRT* 算法的采样过程保留了 RRT* 算法的采样过程，对状态空间进行均匀采样，从而继承了 RRT* 算法的概率完整性。然而 B-RRT* 算法是采用双向树生长的算法，在算法找到可行解并开始收敛到最优解的过程中，双向树到各自树的根的距离大于当前最优解的采样点对最优解的寻找并没有太大意义(算法的最优解不会大于当前最优解)。传统的 B-RRT* 算法对这些采样点进行了大量的操作，并且这些采样点也占用了一定量的内存，这在一定程度上降低了算法收敛的速度。

根据 B-RRT* 算法的这一问题，本文提出的 MB-RRT* 算法采用懒惰采样的方法来减少无效采样点。懒惰采样在采样阶段对采样点进行了筛选，即在每次生成插入点 x_{new} 后计算插入点到根节点的长度 $c(x_{new},x_{init})$，如果这个长度大于当前最优航迹的长度，则抛弃这个插入点。采用懒惰采样可以抛弃一系列对最优航迹搜索没有帮助的新的采样点，因为最优航迹节点距离总是小于或等于当前最优航迹长度，从而避免了对无效节点进行碰撞检测、优化等操作。与此同时，抛弃这些采样点可以使算法新生成的采样点收敛到当前最优航迹长度的范围内，在不影响算法寻找最优航迹的同时减少了树的生长范围。因此，采用懒惰采样这一过程理论上可以在保留算法概率完整性的同时提高算法的收敛速度。

进行 10 次实验并取实验结果的平均值，同样进行 20000 次迭代，未使用懒惰采样的 MB-RRT* 算法的平均运行时间为12.20316s，而使用了懒惰采样的算法的平均运行时间为 10.51662s，加入懒惰采样后的算法的运行效率得到了大幅提升。图 4 所示为常规采样的 MB-RRT* 算法与懒惰采样的 MB-RRT* 算法在 MAP3D 软件上的运行效果对比图，左上圆圈中的黑点部分为初始点以及以初始点为树根生长的树 T_{init}，右下圆圈部分为目标点以及以目标点为树根生长的树 T_{goal}。从图中圆圈圈出的区域可以看出，加入懒惰采样后树 T_{init} 很少在目标点附近采样生长，同时树 T_{goal} 也很少在初始点附近采样生长，这些区域的采样生长对算法收敛到最优解并没有什么帮助，减少对这一区域的采样可以使树的采样生长点收敛在中间区域。



(a)常规采样 (b)懒惰采样

图 4

3.2.3 降采样与曲线拟合

B-RRT* 算法对最后生成的最优航迹的轨迹点进行直线连接，由于采样的随机性以及树以固定步长的方式生长，算法生成的部分航迹轨迹点可以进行截短操作，从而生成更短的航迹；并且≥直线连接的方式使得算法最终生成一条生硬的折线航迹，并不符合实际的运动状况。针对这一情况，MB-RRT* 算法在算法迭代完成之后对最后的轨迹点分别进行了降采样处理和曲线拟合处理，使算法得出的最终航迹为平滑的曲线。

降采样过程对最终生成的轨迹点之间不断地进行碰撞检测，对最后的轨迹不断地截短直至无法截短，这一过程的主要步骤如算法 7 所示。

算法 7 DownSample(θ_{best})

```
1. span=2
2. for span<θbest. size() do
3.   flag=false
4.   i=0
5.   for span+ i <θbest. size() do
6.        i=i+ 1
7.        if CollisionCheck(θbest[i],θbest[i+ span]) then
8.        t=1
9.        for t< span do
10.            t=t+1
11.            θbest. erase(i+1)
12.          end for
13.          flag= true
14.       end if
15.  end for
16.  if flag==false then
17.       span=span+ 1
18.         end if
19. end for
20. return θbest
```

为了使算法最后生成的航迹更加平滑，本文通过构建三次贝塞尔曲线对轨迹点进行插值生成最后的航迹曲线，三次贝塞尔曲线的方程如式(2)所示：

$$B(t)=P_0(1-t)^3+3P_1t(1-t)^2+3P_2t^2(1-t)+P_3t^3 \tag{2}$$

其中，P_0 为起始点，P_3 为目标点，P_1 和 P_2 为控制曲线方向的控制点，只需要根据轨迹点不断计算两个控制点并将其代入方程，即可生成曲线。

图 5 所示为降采样与曲线拟合示意图，虚线为树生长最后生成的轨迹点，折型实线连接的端点为经过降采样函数处理后的轨迹点，曲型实线为根据降采样之后的端点再进行曲线拟合后生成的最终航迹。

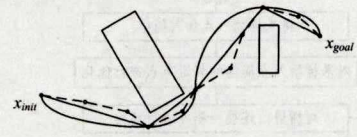


图 5 降采样与曲线拟合示意图

在完成曲线拟合后需要对生成的曲线进行碰撞检测，当发现其中的一段轨迹与障碍物发生碰撞时，需要对这段轨迹进行插值并重新拟合。可直接选取在碰撞部分的轨迹曲线的中点作为插值点，在每次曲线拟合后需要重新进行碰撞检测，从而保证最终生成轨迹的安全性。

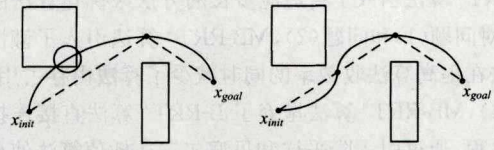


图 6 插值处理示意图

4　MB-RRT* 算法的性能分析

4.1　概率完备性

在任意环境中，若算法在航迹可行解存在的情况下且当运行迭代次数趋于无穷大时能够找到可行解，那么称该算法具有概率完备性。RRT* 算法已经被证明具有概率完备性，同时 RRT* 算法的双向版本 B-RRT* 算法也被证明具有概率完备性。本文提出的 MB-RRT* 算法使用了新的采样策略，即懒惰采样。由于懒惰采样略过了对最优航迹没有影响的无效节点，而且采用双向树结构生长，保证了树的生长能覆盖整个环境空间，并且该算法附加的曲线拟合以及降采样操作在算法完成迭代后进行，并不对算法的概率完备性产生影响，因此算法依旧保留着 RRT* 算法的概率完备性。

4.2　渐近最优性

渐近最优(Asymptotic Optimality)在航迹规划问题当的定义为：令 c^* 为当前环境下航迹规划的最优解，Y_n^{ALG} 为算法 ALG 在迭代到 n 次之后产生的最优航迹解的长度，算法 ALG 满足渐近最优性，当 $P(\{\limsup_{n\to\infty} Y_n^{ALG}=c^*\})=1$ 时。

已经知道 RRT 算法并不具有渐近最优性[18]；而 Sertac Karaman 和 Emilio Frazzoli 证明了 RRT* 算法具有渐近最优性；同时 Matthew Jordan 和 Alejandro Perez 证明了双向树版本的 B-RRT* 算法具有渐近最优性，当其两棵树的连接过程使用类似于 RRT* 算法加入新节点的过程，即上文提到的 *Connect* 函数。本文提出的 MB-RRT* 算法并未对连接过程进行修改，单棵树的生长过程在继承 RRT* 算法的同时引入了自适应步长，而 Sertac Karaman 和 Emilio Frazzoli 提出 RRT* 算法的渐近最优性适用于步长 $\mu>0$ 的任意步长[19]。因此可以得出 MB-RRT* 算法继承了 RRT* 算法的渐近最优性。

4.3　时间复杂度

将 MB-RRT* 算法的计算复杂度与 B-RRT* 算法和 RRT* 算法的计算复杂度进行对比。

首先，*Sample*，*CollisionCheck*，*Steer* 等函数均可在常数时间内完成，而 *AutoStepSteer* 仅仅在传统 *Steer* 函数中多调用了一次 *NearestOb* 方法，该方法和 *CollisionCheck* 类似。对于空间范围内的障碍物搜索，由于障碍物数量是固定的且并不随迭代次数增长，因此 *AutoStepSteer* 函数也可以在常数时间内完成。其次，Sunil Arya 已经证明 *Nearest* 函数需要消耗对数时间进行计算，即 $\Omega(\log n)$[20]，同时与 *Nearest* 函数类似，*Near* 函数作为一个邻近点搜索算法，它的计算时间也为 $\log n$。最后，MB-RRT* 算法的连接过程 *Connect* 函数延续了传统 B-RRT* 算法的连接方法，因此其计算时间和 *Near* 函数相同，即 $\log n$。根据上述条件可以总结出，存在一个常数 ϕ_1，使得：

$$\limsup_{r\to\infty} E\left[\frac{M_r^{MB\text{-}RRT^*}}{M_r^{B-RRT^*}}\right]\leqslant\phi_1 \tag{3}$$

同时结合 Yasar Ayaz 的定理二[16]，可以推论出存在一个常数 ϕ_2，使得：

$$\limsup_{r\to\infty} E\left[\frac{M_r^{MB\text{-}RRT^*}}{M_r^{RRT^*}}\right]\leqslant\phi_2 \tag{4}$$

其中，r 为迭代次数，M_r^{ALG} 为算法 ALG 在 r 次迭代后所执行的步骤数量。

5　仿真实验与分析

二维环境下的航迹规划演示与优化在 Ubuntu 系统下进行，利用 Qt 完成算法在二维环境下的可视化演示。本实验使用的硬件配置如表 1 所列。

表 1　实验硬件环境

类型	参数
处理器	Intel(R) Core(TM) i5-3337U 1.80GHz
系统版本	Ubuntu 64-bit
内存	4GB

5.1　实验地图

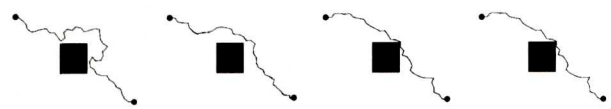
二维环境下的实验环境大小为 800 * 600，通过在环境中放置不同的障碍物生成了 6 幅难度不同的地图，并进行了算法验证，地图的相关参数如表 2 所列。

表 2　实验地图参数

地图	起点坐标	终点坐标	障碍物数量	障碍物占空比
Map1	(50,50)	(750,550)	1	72/1200
Map2	(50,50)	(750,550)	3	96/1200
Map3	(50,50)	(750,550)	5	172/1200
Map4	(50,50)	(750,550)	4	240/1200
Map5	(250,300)	(550,300)	13	197/1200
Map6	(50,300)	(750,300)	2	308/1200

5.2　实验结果与分析

本节介绍 MB-RRT* 算法、B-RRT* 算法、RRT* 算法和 B-RRT 算法在不同地图环境下的运行情况。为了使对比效果更明显，将 MB-RRT* 算法与 B-RRT 算法的运行结果图进行展示对比，其中图 7—图 12 为 MB-RRT* 算法和 B-RRT 算法的运行节点分布图，算法的迭代次数从左到右依次增加。图中的黑色区域为障碍物，左上角黑点为初始节点，右下角黑点为目标点，黑点曲线为经过曲线拟合的最终航迹。



(a)B-RRT 算法运行情况

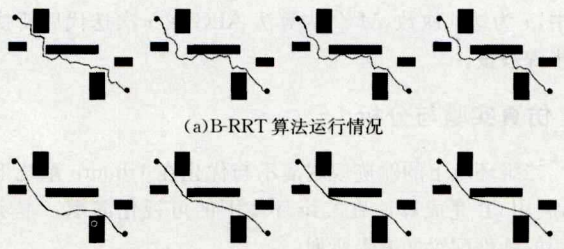
(b)MB-RRT* 算法运行情况

图 7　Map1 环境下两种算法航迹规划结果比较

(a)B-RRT 算法运行情况

(b)MB-RRT* 算法运行情况

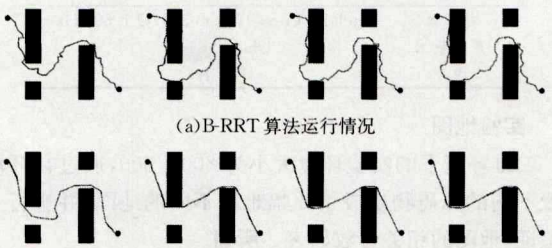
图 8　Map2 环境下两种算法航迹规划结果比较



(a)B-RRT 算法运行情况

(b)MB-RRT* 算法运行情况

图 9 Map3 环境下两种算法航迹规划结果比较



(a)B-RRT 算法运行情况

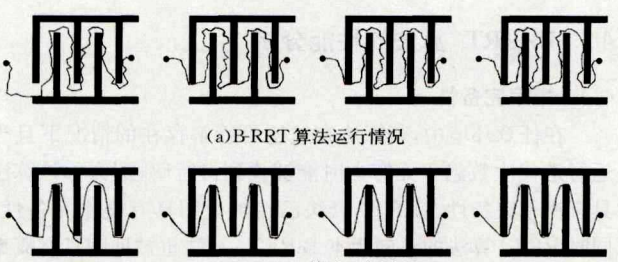
(b)MB-RRT* 算法运行情况

图 10 Map4 环境下两种算法航迹规划结果比较

(a)B-RRT 算法运行情况

(b)MB-RRT* 算法运行情况

图 11 Map5 环境下两种算法航迹规划结果比较



(a)B-RRT 算法运行情况

(b)MB-RRT* 算法运行情况

图 12 Map6 环境下两种算法航迹规划结果比较

通过 MB-RRT* 算法与 B-RRT 算法的实验结果对比可以发现，MB-RRT* 算法拥有更高的收敛率，并且找到的第一个可行解更优，在经历过多次迭代后，MB-RRT* 算法生成的分支和航迹远优于 B-RRT 算法。并且可以观察到，在引入曲线拟合后，MB-RRT* 算法生成的航迹曲线更加平滑，在引入自适应步长之后，MB-RRT* 算法的树在障碍物附近生长的效果更好。同时从 Map3－Map5 中可以明显看出，算法在引入懒惰采样后不仅明显减少了采样点，而且还使采样点往最优航迹处收敛。

表 3 所列为 Map1－Map6 运行的精确数据，数据为每幅地图分别运行 10 次算法所得结果的平均值。通过实验数据可以观测出，与 B-RRT 算法、RRT* 算法和 B-RRT* 算法相比，MB-RRT* 算法显著地提高了算法收敛率，在 30s 的运行时间中，MB-RRT* 算法获得的最优航迹始终优于其他 3 种算法获得的当前最优航迹。而且在环境较为复杂的 Map6 中，相对于其他 3 种算法，MB-RRT* 算法可以较快地找到高质量的初始可行解。

表 3 Map1－Map6 实验结果

	算法	Map1	Map2	Map3	Map4	Map5	Map6
初次找到可行解的迭代数量	B-RRT	48.00	56.00	59.00	121.60	510.25	1931.40
	RRT*	642.8889	676.4000	832.1000	582.6000	836.2000	2279.4000
	B-RRT*	38.7	43.0	58.1	77.5	437.3	1273.8
	MB-RRT*	24.1	31.4	31.4	59.9	310.1	1141.8
初次找到可行解的路径长度	B-RRT	1185.252	1101.854	1161.894	1517.506	1491.003	2747.996
	RRT*	974.7834	1002.3490	1005.1390	1343.4120	1031.9770	2092.4640
	B-RRT*	953.4240	978.5331	1021.0440	1360.5550	1098.5760	2038.5970
	MB-RRT*	917.2541	942.1798	952.4027	1238.1790	1051.8820	2001.3810
初次找到可行解的运行时间	B-RRT	0.004253	0.005852	0.005314	0.013311	0.096945	1.041348
	RRT*	0.606613	0.845238	1.135039	1.805126	1.926819	13.933850
	B-RRT*	0.005545	0.006843	0.007908	0.011877	0.080707	0.512396
	MB-RRT*	0.004266	0.004211	0.004516	0.011643	0.069621	0.460113
运行 1s 所得解	B-RRT	1041.938	1058.590	1108.920	1533.778	1159.265	无
	RRT*	986.062	1009.508	1006.722	1314.696	1113.648	无
	B-RRT*	904.9468	919.3655	918.0334	1199.2860	961.3479	1946.0880
	MB-RRT*	893.4950	909.8631	908.7100	1178.9010	942.8484	1921.4260
运行 2s 所得解	B-RRT	1041.938	1058.590	1108.920	1533.778	1159.265	2432.474
	RRT*	948.9941	962.1591	995.1841	1299.7740	1044.6940	无
	B-RRT*	900.2464	914.4207	914.1423	1181.4000	931.9325	1900.0890
	MB-RRT*	892.5797	907.3002	906.5679	1169.8760	917.9452	1880.7750
运行 3s 所得解	B-RRT	1040.944	1046.424	1108.448	1528.304	1097.521	2431.362
	RRT*	944.0964	958.7564	984.7014	1294.5810	1030.1220	无
	B-RRT*	899.1169	913.0011	913.1852	1178.7230	924.8651	1888.3440
	MB-RRT*	891.4023	906.1998	905.0390	1167.4530	914.6325	1869.4980
运行 4s 所得解	B-RRT	1040.944	1046.424	1107.574	1527.894	1097.521	2397.938
	RRT*	935.4510	958.5441	969.5709	1290.8330	1024.1110	无
	B-RRT*	898.7651	912.4743	911.9371	1176.5250	922.0849	1882.7670
	MB-RRT*	891.7371	905.0170	904.2226	1166.6800	910.7767	1862.6820

（续表）

	算法	Map1	Map2	Map3	Map4	Map5	Map6
运行 10s 所得解	B-RRT	1040.9440	1044.7100	1107.574	1527.8940	1097.5210	2397.9380
	RRT*	923.2262	940.2506	951.1838	1236.2610	979.0442	2022.9070
	B-RRT*	896.6633	910.4573	910.1475	1170.3670	910.8276	1870.9970
	MB-RRT*	891.1582	903.6154	902.1375	1160.2750	899.5495	1854.1670
运行 20s 所得解	B-RRT	1038.522	1043.309	1100.438	1527.894	1073.651	2390.934
	RRT*	915.6263	934.7329	938.3407	1228.8240	948.6643	2011.0180
	B-RRT*	895.7608	907.8965	907.1583	1166.3950	901.7587	1861.1430
	MB-RRT*	890.8855	902.8492	900.7393	1156.4680	892.8557	1841.6910
运行 30s 所得解	B-RRT	1038.400	1043.309	1099.760	1521.162	1006.578	2384.430
	RRT*	913.8704	929.8907	928.7140	1216.6980	937.4207	1952.3870
	B-RRT*	894.9959	906.9062	905.8214	1164.5280	898.7005	1852.2480
	MB-RRT*	890.4516	902.2722	900.3799	1155.4290	890.2752	1836.2450

图 13－图 15 为算法找到初始可行解的情况对比柱状图，可以看出，拥有自适应步长的 MB-RRT* 算法能够在较短的时间内以较少的迭代次数找到相对最优的初始可行解。

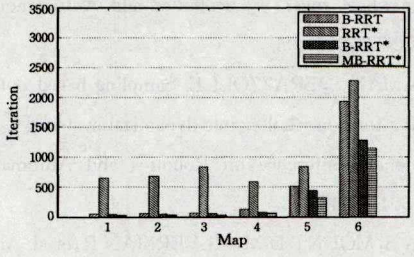


图 13　初始找到可行解的迭代次数对比图

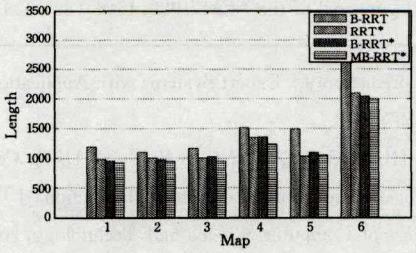


图 14　初始找到可行解的路径长度对比图

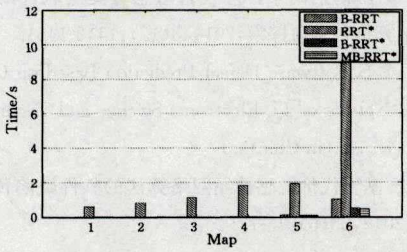


图 15　初始找到可行解的运行时间对比图

图 16 为 RRT* 算法、B-RRT* 算法和 MB-RRT* 算法在 Map5 中路径长度随时间变化的趋势图，由于 B-RRT 算法不具备渐近最优性，因此不进行比较。

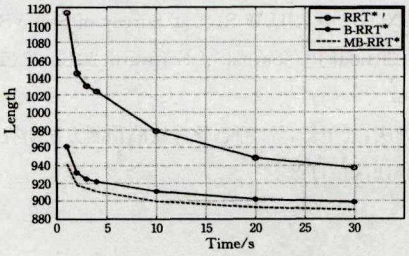


图 16　3 种算法的收敛情况的对比

从图中可以看出 3 种算法求得的路径长度随时间呈指数下降，且 3 种算法的收敛率大小关系为 MB-RRT* ＞B-RRT* ＞RRT* 。

结束语　本文设计了一种无人机航迹规划算法 MB-RRT* ，提出了懒惰采样以提高算法收敛速度并减少内存占用；设计了自适应步长以解决算法在障碍物附近生长树的局限性问题，从而提高了找到初始可行解的速度和质量；最后利用降采样和三次贝塞尔插值算法实现了曲线拟合的功能，使算法最终生成相对平滑的航迹，为无人机实际飞行提供了可行的航迹规划方法。本文仅在二维平面中进行可行航迹规划，对于障碍物的设计也都是静止且边缘平滑的矩形叠加，但是真正的无人机飞行环境都是在三维空间中，而且障碍物形状都是不规则的，甚至会随时间推移而在空间中移动。因此在未来的工作中，为了将无人机航迹自动规划进行实际应用，需要进一步尝试在三维地图环境中的航迹规划，同时深入研究对动态障碍物的避障效果。

参 考 文 献

[1] VALAVANIS K P. Advance in unmanned aerial vehicles[M]. Springer Netherlands, 2007.

[2] DE A, CAVES J. Human-automation collaborative RRT for UAV mission path planning[M]. Massachusetts Institute of Technology, 2010.

[3] TRIHARMINTO H H, PRABUWONO A S. UAV Dynamic Path Planning for Intercepting of a Moving Target: A Review[J]. Communications in Computer and Information Science, 2013, 376: 206-219.

[4] HOLLAND J H. Adaptation in natural and artificial systems [M]. MIT Press, 1992.

[5] ROBERGE V, TARBOUCHI M, LABONTE G. Comparison of Parallel Genetic Algorithm and Particle Swarm Optimization for Real Time UAV Path Planning[J]. IEEE Transactions on Industrial Informatics, 2013, 9(1): 132-141.

[6] KAVRAKI L, SVESTKA P. Probabilistic roadmaps for path planning in high-dimensional configuration spaces[C] // IEEE Transactions on Robotics & Automations, 1996: 566-580.

[7] LAVALLE S M, KUFFNER J J. Randomized Kinodynamic Planning[J]. IEEE International Conference on Robotics & Automa-

tion,1999,1(5):473-479.

[8] MELCHIOR N A,SIMMONS R. Particle RRT for Path Planning with Uncertainty[C]// IEEE International Conference on Robotics & Automation. 2007:1617-1624.

[9] LINDEMANN S R,LAVALLE S M. Incrementally reducing dispersion by increasing voronoi bias in rrts[J]. IEEE International Conference on Robotics & Automation,2004,4(4):3251-3257.

[10] KARAMAN S,FRAZZOLI E. Incremental sampling-based algorithms for optimal motion planning[J]. International Journal of Robotics Research,2010,30(7):2011.

[11] QURESHI A H,MUMTAZ S,IQBAL K F,et al. Triangular geometry based optimal motion planning using RRT*-motion planner[C]// IEEE International Workshop on Advanced Motion Control. 2014:380-385.

[12] KUFFNER J J,LAVALLE S M. RRT-connect:An efficient approach to single-query path planning[C]// IEEE International Conference on Robotics and Automation. 2000:995-1001.

[13] JORDAN M,PEREZ A. Optimal Bidirectional Rapidly-Exploring Random Trees:MIT-CSAIL-TR-2013-021[R]. 2013.

[14] LAVALLE S M,KUFFNER J J. Rapidly-Exploring Random Trees:Progress and Prospects[J]. Algorithmic & Computational Robotics New Directions,2000:293-308.

[15] KARAMAN S,FRAZZOLI E. Sampling-based algorithms for optimal motion planning[J]. International Journal of Robotics Research,2011,30(7):846-894.

[16] QURESHI A H. AYAZ Y. Intelligent bidirectional rapidly-exploring random trees for optimal motion planning in complex cluttered environments[J]. Robotics & Autonomous Systems,2015,68:1-11.

[17] CARSTEN J,FERGUSON D,STENTZ A. 3D field D*:Improved path planning and replanning in three dimensions[C]// IEEE/RSJ International Conference on Intelligent Robots and Systems. 2006:3381-3386.

[18] RAJA R,DUTTA A,VENKATESH K S. New potential field method for rough terrain path planning using genetic algorithm-for a 6-wheel rover[J]. Robotics and Autonomous Systems,2015,72(C):295-306.

[19] KARAMAN S,FRAZZOLI E. Sampling-based optimal motion planning for non—holonomic dynamical systems[C]// IEEE International Conference on Robotics and Automation. IEEE,2013:5041-5047.

[20] ARYA S,MOUNT D M,SILVERMAN R,et al. An optimal algorithm for approximate nearest neighbor search in fixeddimensions[J]. Journal of the ACM,1999,45(6):891-923.

(上接第 180 页)

[9] CAUWENBERGHS G,POGGIO T. Incremental and Decremental Support Vector Machine Learning[M]// Advances in Neural Information Processing Systems 13. 2010:409-415.

[10] WU F J. Understanding Knowledge Sharing Activities in Software Fault-prone Prediction:a Transfer Learning Study[J]. Journal of Chinese Computer Systems,2014,35(11):2416-2421. (in Chinese)
吴方君. 软件缺陷预测经验共享:一种迁移学习方法[J]. 小型微型计算机系统,2014,35(11):2416-2421.

[11] ZHANG Q,LI M,WANG X S,et al. Instance-based Transfer Learning for Multi-source Domains[J]. Acta Automatica Sinica,2014,40(6):1176-1183. (in Chinese)
张倩,李明,王雪松,等. 一种面向多源领域的实例迁移学习[J]. 自动化学报,2014,40(6):1176-1183.

[12] CHAWLA N V,BOWYER K W,HALL L O,et al. SMOTE:synthetic minority over-sampling technique[J]. Journal of Artificial Intelligence Research,2011,16(1):321-357.

[13] RICHELLI A,COMENSOLI S,KOVACS-VAJNA Z M. A DC/DC Boosting Technique and Power Management for Ultralow-Voltage Energy Harvesting Applications[J]. IEEE Transactions on Industrial Electronics,2012,59(6):2701-2708.

[14] ZHENG J. Cost-sensitive boosting neural networks for software defect prediction[J]. Expert Systems with Applications,2010,37(6):4537-4543.

[15] LI Y,HUANG Z Q,FANG B W,et al. Using Cost-Sensitive Classification for Software Defects Prediction[J]. Journal of Frontiers of Computer Science and Technology,2014,8(12):1442-1451. (in Chinese)
李勇,黄志球,房丙午,等. 代价敏感分类的软件缺陷预测方法[J]. 计算机科学与探索,2014,8(12):1442-1451.

[16] MIAO L S. Software Defect Prediction Based on Cost-Sensitive Neural Networks[J]. Electronic Science and Technology,2012,25(6):75-78. (in Chinese)
缪林松. 基于代价敏感神经网络算法的软件缺陷预测[J]. 电子科技,2012,25(6):75-78.

[17] HE L,SONG Q B,SHEN J Y. Boosting-Based k-NN Learning for Software Defect Prediction[J]. Pattern Recognition and Artificial Intelligence,2012,25(5):792-802. (in Chinese)
何亮,宋擒豹,沈钧毅. 基于 Boosting 的集成 k-NN 软件缺陷预测方法[J]. 模式识别与人工智能,2012,25(5):792-802.

[18] CHEN X,GU Q,LIU W S,et al. Survey of Static Software Defect Prediction[J]. Journal of Software,2016,27(1):1-25. (in Chinese)
陈翔,顾庆,刘望舒,等. 静态软件缺陷预测方法研究[J]. 软件学报,2016,27(1):1-25.