4，栈和队列-3

- 栈和队列的概念
- 数据的生成，缓存，使用和顺序
- 栈的实现和问题
- 栈应用实例
- 栈与递归，递归和非递归
- 队列的实现和问题
- 队列应用实例
- 迷宫求解，搜索问题，栈与队列
- 相关问题

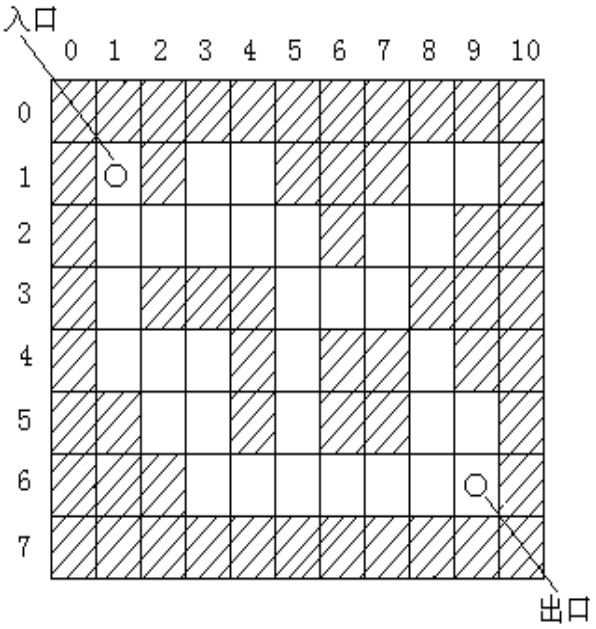
应用：迷宫问题

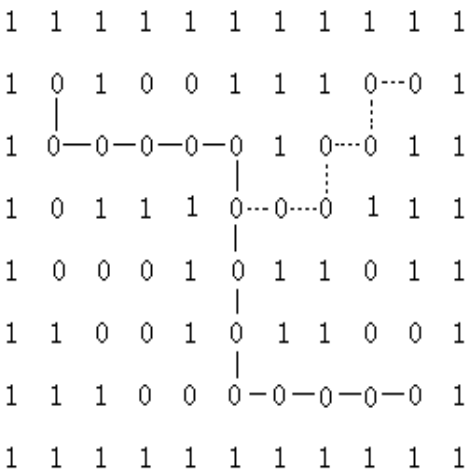
- 走迷宫是一种常见智力游戏，也是许多实际问题的反映和抽象，例如，在公路网或铁路网上找可行或最优路线，电子地图路径检索，计算机网络的路由检索，情况都与此类似。下面的讨论从具体到一般
- 迷宫问题：
  - 给定一个迷宫，包括一个迷宫图，一个入口点和一个出口点
  - 设法找到一条从入口到出口的路径
- 搜索从入口到出口的路径的问题具有递归性质：
  - 如果当前位置就是出口，问题已经解决
  - 如果从当前位置已无路可走，当前的探查失败（下面怎么办？）
  - 取一个可行方向前进一步，从那里继续探查通往出口的路径
- 每个具体迷宫是这个问题的一个实例

  各种具体的迷宫可能具有不同的表面结构，但问题实质都差不多

迷宫问题

- 现在考虑一种抽象迷宫，它是平面的，形式比较规范
  - 在每个可行位置，上/下/左/右四个方向可能有通路，可以移动一步
  - 这种迷宫可以直接映射到二维的 0/1 矩阵，如下图：





迷宫问题

- 迷宫问题的特点：
  - 存在一集可能位置，一些位置相互连通，一步可达
  - 一个位置可能连通若干位置，出现向前探查的多种可能（有分支）
  - 目标是找到一条路径（而不是找所有的能行路径）
  - 为了找到路径，可能需要逐一探查不同的分支
- 工作中需要缓存信息：如果当前位置存在多个可能探查方向，由于下步只能探查一个，需要把不能立刻处理的方向记下来，后面可能使用
- 向前搜索也有一些不同方式，可以比较冒进，也可以稳扎稳打
- 如果搜索还没找到出口，有一些可能做法。例如：
  - 直到无路可走时才考虑其他选择，换一条没走过的路继续
  - 每步都从最早记录的有选择位置向前，检查并保存另一个可达位置

无论如何，路径搜索中必须记录（缓存）有关信息，以供后面使用

迷宫问题

- 实现不同的路径搜索过程，需要用不同缓存结构保存分支点信息
  - 用栈保存和提供信息，实现的是回到之前最后选择点继续的过程
  - 用队列保存信息，就是总从最早遇到的选择点继续搜索
- 用栈保存信息，实际上是尽可能利用已经走过的路（改变最少）

  下面先考虑实现这一过程的算法。算法里用一个栈保存分支点信息

  用队列方式记录信息的可能性后面考虑
- 解决这个问题需要设计一种问题表示形式和一种确定可行方向的方式
- 注意：还要防止出现兜圈子的情况，为此需要记录到过的位置

  如果不记录试探过的位置，程序运行中就可能重复探查某些局部路径，即使出口是可达的，也永远找不到

  记录了探查过的位置，在搜索中检查，保证不重复探查任一位置
- 两种记录方式：1, 专门保存信息；2, 或直接记在“地图” (迷宫图) 上

迷宫问题

- 问题求解中很关键的一件事是选择合适的问题表示。这里采用：
  - 一个整数“矩阵”（两层的 list）表示迷宫
  - 入口和出口都用一对表下标表示（位置）
  - 初始时，通路上的点用元素值为 0 表示，非通路点用 1
  - 为避免无穷循环，在搜索中把检查过的点标记为2（记录方法 2）
  - 这样，能通行的位置里的数值是 0，否则都是不能通行的（或者根本不是通路，或者是已经探查过的位置）
- 每个位置有四个相邻位置。为正确（且方便）地处理可能前进方向，需要确定一种系统的检查方法。我们把一个位置的四邻看作“东南西北”四方位置，按这个顺序逐方向处理
- 对单元 (i, j)，其相邻的四个位置的数组元素下标如右图所示

	N (i-1,j)	
w (i,j-1)	* (i, j)	E (i,j+1)
	S (i+1,j)	

迷宫问题

- 用一个序对的 **list** 表示从任一 **(i, j)** 得到其四邻位置应加的数对：

  dirs = [(0,1), (1,0), (0,-1), (-1,0)]

  对当前位置 **(i, j)**，顺序加 **dirs[0], dirs[1], dirs[2], dirs[3]** 就可以得到在东西南北四个相邻位置

  这使检查当前位置四邻的工作可以通过一个循环完成

- 先定义两个简单的辅助函数（设 **pos** 是形式为 **(i, j)** 的序对）：

  def mark(maze, pos): # 给迷宫 maze 的位置 pos 标 2 表示“到过了”
  maze[pos[0]][pos[1]] = 2

  def passable(maze, pos): # 检查迷宫 maze 的位置 pos 是否可行
  return maze[pos[0]][pos[1]] == 0

- 先考虑用递归方式写出的算法，它比较简单，不需要辅助数据结构

  如前所述，语言系统为支持递归程序，在内部也用了一个运行栈

迷宫的递归求解

- 前面提出了对迷宫求解问题的递归观点，改写如下：
  - 每个时候总有一个当前位置，开始时这个位置是迷宫入口
  - 如果当前位置就是出口，问题已解决
  - 如果从当前位置已无路可走，当前的探查失败
  - 取一个可行相邻位置用同样方式探查，如果从那里可以找到通往出口的路径，那么从当前位置到出口的路径也就找到了
- 递归的迷宫求解算法是上述描述的实现，开始时把入口作为当前位置
  - **mark** 当前位置
  - 检查它是否出口，如果是则成功结束
  - 逐个检查当前位置的四邻是否可以通到出口（递归）
    - 成功时，整个求解也成功
    - 失败时放弃这个可能性

迷宫问题

- 递归实现的主函数如下：

```
def find_path(maze, start, end):
    mark(maze, start);
    if start == end:            # 已到达出口
        print(start, end=" ") # 输出这个位置
        return True             # 成功结束
    for i in range(4): # 否则按四个方向顺序探查
        nextp = start[0]+dirs[i][0], start[1]+dirs[i][1] # 下一个考虑
        if passable(maze, nextp): # 不可行的相邻位置不管
            if find_path(maze, nextp, end): # 如果从 nextp 可达出口
                print(start, end=" ") # 输出这个点
                return True;            # 成功结束
    return False
```

函数按从出口到入口的顺序输出路径上经历的位置（下标序对）

迷宫问题：回溯法和栈

- 不用递归技术求解迷宫问题的一种方法称为回溯法，需要用一个栈：
  - 前进：
    - 如果当前位置存在尚未探查的向前分支，就选定一个这样的分支向前探查。如这里还有其他未探查分支，需记录相关信息
    - 找到出口时成功结束，所有可能都已探查但不能成功时失败结束
  - 后退（回溯）
    - 遇死路（已无向前的未探查分支）时退回最近记录的分支点，检查那里是否还存在未探查分支，如没有就将其删除并继续回溯
    - 找到存在未探查分支的位置时将其作为当前位置继续探查
- 由于分支位置的记录和使用/删除具有后进先出性质，应该用栈保存信息。遇到分支点将相关信息压入栈，删除分支点时将有关信息弹出
- 回溯法也是一种重要的算法设计模式，通常总用一个栈作为辅助结构，保存工作中发现的回溯点，以便后面考虑其他可能性时使用

回溯法和栈

- 可以应用回溯法求解的具体问题可能差别很大，但它们
  - 都是从一个出发点开始，设法找到目标（因此也称为搜索）
  - 都需要使用一个栈，搜索过程的行为分为向前搜索和向后回溯
- 一种典型的实现方法如下：
  - 首先，把出发点放入栈，在栈里保存与搜索有关的信息
  - 反复做下面几个操作（条件：栈不空）
    - 弹出一项以前保存的信息（当前点）
    - 检查从这里出发前进的可能性（下一个探查点）

      如果可以向前，把从当前点出发的其他可能存入栈里

      把下一个探查点也入栈

注意：如果从当前点已无前进可能，算法将直接转到下一次迭代，弹出更早保存的点（这是回溯）。找到下一探查点就是前进

迷宫问题：回溯法

- 对迷宫问题，需要从入口出发搜索
  - 遇到出口时成功结束
  - 遇分支结点时按上面介绍的方式记录信息，继续探查并可能回溯
- 搜索中把哪些位置入栈？存在两种合理的选择：
  - 从入口到当前探查位置，途径的所有位置都入栈
  - 只在栈里保存上述路径中存在未探查方向的那些位置。这一方式要求在入栈操作前检查所考虑位置的情况，有可能节省空间
- 仔细考虑，可以看到两个情况：
  1. 把一个存在未探查方向的位置入栈，后来回溯到这里时也可能不再存在未探查方向了（原有的未探查方向在此期间已经检查过了）
  2. 为在算法最后输出找到的路径，也需要知道路径上所有的位置

下面算法采用记录经过所有位置的方式，主要是为了输出结果路径

迷宫问题：回溯法

- 迷宫问题算法框架（用一个栈记录搜索中需保存的信息）：

  入口 start 相关信息（位置和尚未探索方向）入栈;
  while 栈不空:
    弹出栈顶元素作为当前位置继续搜索
    while 当前位置存在未探查方向:
      求出下一探查位置 nextp
      if nextp 是出口: 输出路径并结束
      if nextp 尚未探查
        将当前位置和 nextp 顺序入栈并退出内层循环

- 根据迷宫的表示形式，以及回溯后能继续搜索的需要。需要在栈里保存序对 (pos, nxt)，记录分支点位置和在该位置还需要考虑的下一方向
  - 分支点位置 pos 用行、列坐标表示，可以用一个序对
  - 在 pos 的下一探索方向 nxt 是整数，表示回溯到此时应探查的下一方向。4 个方向编码为 0、1、2、3（dirs 的下标）

迷宫问题：回溯法

```
def maze_solver(maze, start, end):
    if start == end: print(start); return
    st = SStack()
    mark(maze, start)
    st.push((start, 0))     # 入口和方向 0 的序对入栈
    while not st.is_empty(): # 走不通时回退
        pos, nxt = st.pop()     # 取栈顶及其探查方向
        for i in range(nxt, 4): # 依次检查未探查方向
            nextp = (pos[0] + dirs[i][0], pos[1] + dirs[i][1]) # 算出下一点
            if nextp == end:        # 到达出口,打印路径
                print_path(end, pos, st)
                return
            if passable(maze, nextp):# 遇到未探查的新位置
                st.push((pos, i+1))        # 原位置和下一方向入栈
                mark(maze, nextp)
                st.push((nextp, 0))       # 新位置入栈
                break # 退出内层循环，下次迭代将以新栈顶为当前位置继续
    print("No path found.")  # 找不到路径
```

注意，栈里一点之下总是到它的路径上的前一个点。这是搜索中的一个不变性质

迷宫问题和搜索

- 迷宫问题是一大类问题的代表。这类问题的基本特征是：
  - 存在一集可能状态（位置、情况等，状态集合可能很大）

    例：迷宫问题中的所有可能位置
  - 有一个初始状态 **s0**，一个或多个结束状态（或者有结束判断方法）

    例：迷宫问题中的入口
  - 对每个状态 **s**，**neighbor(s)** 表示与 **s** 相邻的一组状态（一步可达）

    例：迷宫中每个位置的相邻位置
  - 有一个判断函数 **valid(s)** 判断 **s** 是否为可行的下一状态

    前面算法定义里的 **possible** 实现这种功能
- 问题是：找出从 **s0** 出发到达某个（或全部）结束状态的路径

  或是从 **s0** 出发，设法找到一个或全部解（一个或全部结束状态）

  这类问题被称为状态空间搜索或者路径搜索

迷宫问题和搜索

- 从前面情况看，这种问题
  - 可以用递归的方法求解，其中用递归的方式向前探查
  - 也可以用一个栈保存中间信息，通过回溯法求解
- 可以通过状态空间搜索解决的问题很多，经典的简单实例：
  - 八皇后问题（在国际象棋棋盘安排八个皇后，使之不能相互攻击）
  - 骑士周游问题（在国际象棋棋盘上为骑士找到一条路径，使之可以经过棋盘的每个格子恰好一次。骑士走法与中国象棋的马一样，走“日”字）；等等
  - 这些可以作为空间搜索问题的练习
- 许多实际应用问题需要通过空间搜索的方式解决，如
  - 许多调度、规划、优化问题（如背包问题）
  - 数学定理证明（有一些事实和推理规则）

状态空间搜索：栈和队列

- 一般性认识：对一个需要用计算的方法处理的问题
  - 如有全局性系统性的认识，就可能做出一个专门的算法解决问题
  - 如果只有对问题空间的局部性认识，无法做出直接求解问题的算法，还是有可能将其转化为一个状态空间搜索问题。搜索法是一种通用问题求解方法（**general problem solving**）
- 搜索过程进展中的情况：
  - 已经探查了从初始状态可达的一些中间状态
  - 已经探查的某些中间状态存在着尚未探查的相邻状态
  - 显然，对任何从初始状态可达的中间状态，与它相邻的状态也是可达的。因此，从一个中间状态向前探查可能得到新的可达状态
  - 若新确定的可达状态是结束状态，就么找到了初始状态到结束状态的路径；否则，新可达状态应加入已探查的中间状态集
- 显然，搜索中需要记录存在未探查邻居的中间状态，以备后面使用

状态空间搜索：栈和队列

- 要记录存在尚未完全探索后继状态的中间状态，需要用缓存结构。原则上说栈和队列都可用，但结构的选择将对搜索进展方式产生重大影响
- 前面的迷宫算法里用的是栈，栈的特点是后进先出
  - “后进”的状态是在搜索过程中较晚遇到的状态，即是与开始状态距离较远的状态。“后进先出”意味着从最后遇到的状态考虑继续向前探索，尽可能向前检查，尽可能向远处探索
  - 只有后来的状态已经无法继续前进时，才会退到前面最近保存的状态，换一种可能性继续，后退并考虑其他可能性的动作就是回溯
- 如果用队列，队列的特点是先进先出
  - “先进”的状态是在搜索过程中较早遇到的状态，即与开始状态距离较近的状态。“先进先出”要求先考虑距离近的状态，从它们那里向外扩展，实际上是一种从各种可能性“齐头并进”式的搜索
  - 在这种搜索过程中没有回溯，是一种逐步扩张的过程。作为示例，下面考虑基于队列的迷宫求解算法

基于队列的迷宫求解算法

- 要实现一个基于队列的迷宫求解算法，可以重用前面的基本设计
  - 迷宫的矩阵表示和矩阵元素的取值方式
  - 标记函数 **mark** 和位置检查函数 **passable**
  - 表示方向的表 **dirs** 和对方向的循环处理方式
- 新算法的基本框架：

将 **start** 标记为已达
start 入队
while 队列里还有为充分探查的位置
　取出一个位置 **pos**
　检查 **pos** 的相邻位置
　　遇到 **end** 成功结束
　　尚未探查的都 **mark** 并入队
队列空，搜索失败

- 很容易基于这一算法框架写出一个函数（这里没有递归）

基于队列的迷宫求解算法

- 基本设计与用栈的算法一样，只是改用队列作为缓存结构：

```
def maze_solver_queue0(maze, start, end):
    qu = SQueue()
    mark(maze, start)
    qu.enqueue(start)           # start position into queue
    while not qu.is_empty(): # have possibility to try
        pos = qu.dequeue()     # take next try position
        for i in range(4):   # chech each direction
            nextp = (pos[0] + dirs[i][0],
                     pos[1] + dirs[i][1])    # next position
            if passable(maze, nextp): # find a new direction
                if nextp == end:            # end position, :-)
                    print("Path find.")     # succeed!
                    return
                mark(maze, nextp)
                qu.enqueue(nextp)        # new position into queue
    print("No path.")  # :-(
```

有问题吗？

如果入口就是出口，怎么样？

很容易修改路在那里？

基于队列的迷宫求解算法

- 这里遇到的最重要问题是如何获得所希望的路径
  - 前面的栈算法里，栈里每个点下面是到达它路径上的前一个点

    找到出口时，当时的栈里正好保存着到从入口到出口一条路径
  - 对于队列算法，栈中保存的位置点的顺序与路径无关

    例如，如果一个点有几个“下一探查点”，它们将顺序进队列
- 结论：在找到出口时，无法基于当时的信息追溯出一条成功路径

  要想在算法结束时得到路径，必须在搜索中另行记录有关信息
- 注意：在从当前点 **a** 找到下一点 **b** 时，如果从 **b** 能到达出口，那么 **a** 就是成功路径上 **b** 的前一个点，这个关系可能是最终路径里的一节
- 为搜索到达出口时能获得相关路径，遇到新位置时就需要记住其前驱
  - 可以用一个字典记录搜索中得到的这方面信息
  - 到达出口后反向追溯，就可以得到从迷宫入口到出口的路径了

基于队列的迷宫求解算法

- 对原算法做几处修改，让它返回得到的路径：

```
def maze_solver_queue1(maze, start, end):
    if start == end: return [start]
    precedent = dict()
    ... ...
    while not qu.is_empty(): # have possibility to try
        ... ...
            if passable(maze, nextp):     # find new position
                if nextp == end:          # end position, :-)
                    return build_path(start, pos, end, precedent)
                ... ...
                precedent[nextp] = pos    # set precedent of nextp
    ... ...
```

  - 用字典 **precedent** 记录前驱关系，遇到新探查点时记录其前驱
  - 最后用一个过程追溯前驱关系，构造出路径返回

基于队列的迷宫求解算法

- 构造路径的函数很简单：

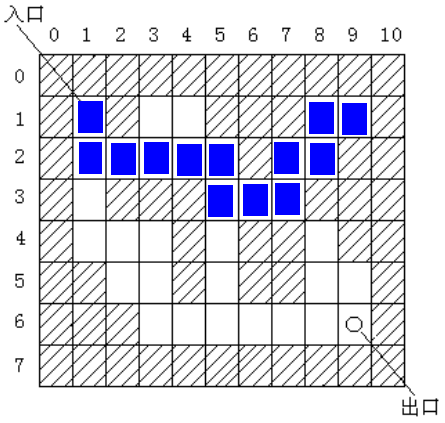
```
def build_path(start, pos, end, precedent):
    path = [end ]
    while pos != start:
        path.append(pos)
        pos = precedent[pos]
    path.append(start)
    path.reverse()
    return path
```

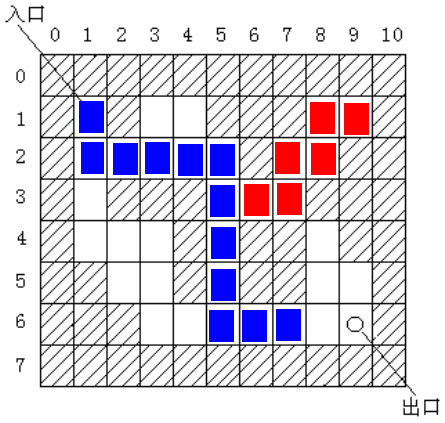
  - 最后调用 reverse 方法，得到从迷宫入口到出口的路径
  - 这个算法具有 O(n) 复杂性，如果用直接在前端插入的方式构造正确顺序的路径，就会是 $O(n^2)$ 算法
- 可见，在需要做搜索时，栈或队列都可以用作缓存结构。什么情况下应该优先考虑某一方式？为此需要理解这两种工作方式，理解其性质

状态空间搜索：栈和队列

- 基于栈搜索迷宫的两个场景：

  蓝色表示栈里记录的位置，红色表示已搜索过不会再去检查的位置



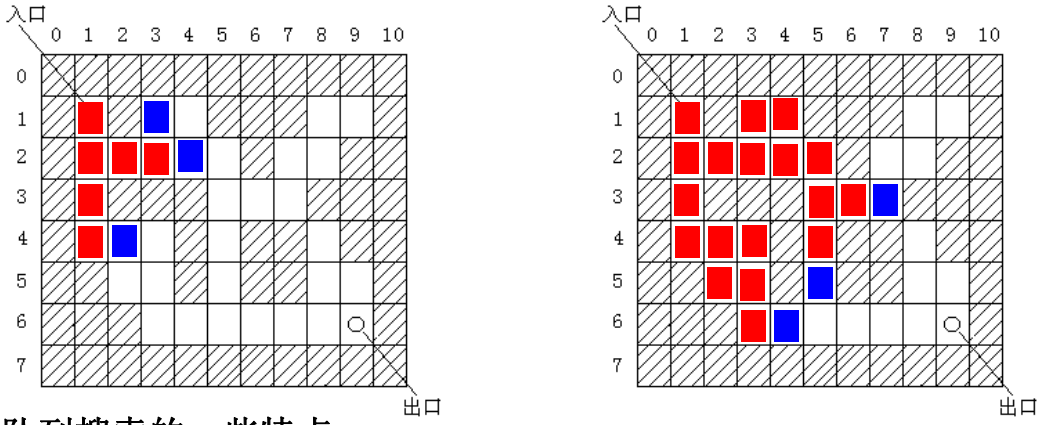


请注意基于栈的搜索过程的一些特点：

- 搜索比较“冒进”，可能在一条路走很远，“勇往直前/不撞南墙不回头”
- 有可能在探查了不多的状态就找到了解；也可能陷入很深的无解区域。如果陷入的无解子区域包含无穷个状态，这种搜索方法就找不到解了

状态空间搜索：栈和队列

- 基于队列搜索迷宫的两个场景（蓝色表示队列里记录的位置）：



基于队列搜索的一些特点：

- 是稳扎稳打的搜索，只有同样距离的状态都检查完后才更多前进一步
- 队列里保存的是已搜索区域的前沿状态
- 如果有解（通过有穷长路径可以到达结束状态），这种搜索过程一定能找到解，而且最先找到的必定是最短的路径（最近的解）

状态空间搜索：栈和队列

- 由于这两种搜索的性质，
  - 基于栈的搜索被称为“深度优先搜索”（**depth-first search**）
  - 基于队列的搜索被称为“宽度优先搜索”（**width-first search**）
- 如果找到了解，如何得到有关的路径：
  - 基于栈的搜索，可以让栈里保存所找到路径上历经的状态序列
  - 基于队列的搜索，需要用其他方法记录经过的路径。一种方式是在到达每个状态时记录其前一状态，最后追溯这种“前一状态”链，就可以确定路径上的状态（相当于对每个状态形成一个栈）
- 搜索所有可能的解和最优解
  - 基于栈搜索，找到一个解之后继续回溯，有可能找到下一个解。遍历完整个状态空间就能找到所有的解，从中可以确定最优解
  - 基于队列搜索，找到一个解之后继续也有可能找到其他解，但是到各个解的路径将越来越长，第一个解就是最优解

状态空间搜索：栈和队列

- 空间开销：搜索状态空间需要保存中间状态，空间开销就是搜索过程中栈或队列里的最大元素项数。两种搜索在这方面的表现也很不同

  基于栈的深度优先搜索，所需的栈空间由找到一个解（或所有解）之前遇到的最长搜索路径确定，这种路径越长，需要的存储量就越大

  基于队列的宽度优先搜索，所需的队列空间由搜索过程中最多的可能路径分支确定。可能分支越多，需要的存储量就越大。此外，为能得到路径还需要另外的存储，其存储量与搜索区域的大小线性相关

- 总结：如果一个问题可以看作空间搜索问题，栈和队列都可以使用，具体用什么要看实际问题的各方面性质

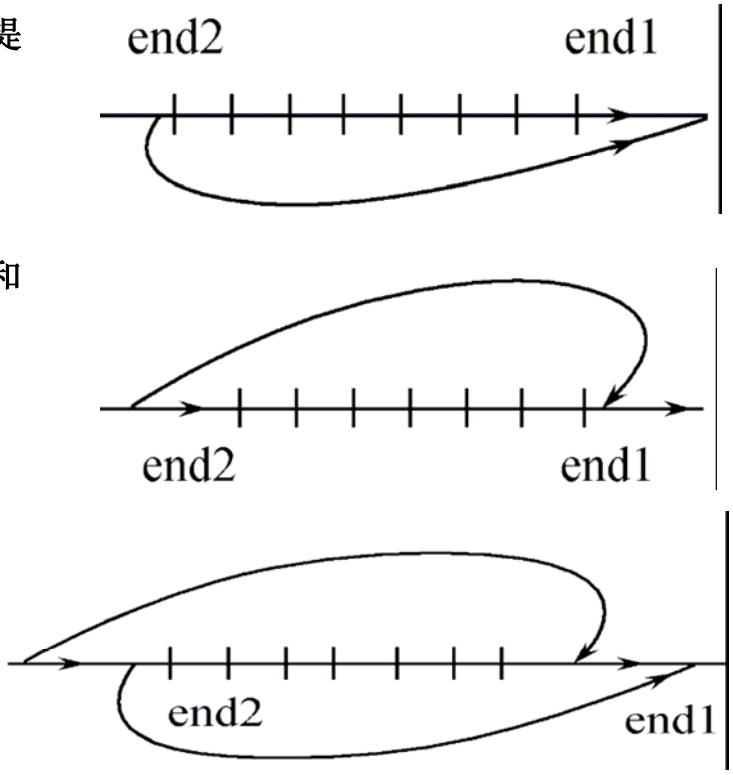
  两种方式遍历状态空间的方式不同。形象地说，基于栈的深度优先搜索像“横扫”，基于队列的宽度优先搜索像“蚕食”

- 遇到搜索问题，应该考虑上面对基于栈和队列的搜索的性质分析

  特别是空间的深度和宽度情况，它们决定搜索的空间开销，还应该考虑是找要求到一个解，全部解，还是最优解等

与栈和队列类似的其他结构

- 数据结构教科书里通常会提出一些相关结构，例如
- 超栈：允许两端弹出
- 超队列：允许两端入队
- 双端队列：允许两端插入和弹出元素



Python 标准库的 deque 类

- Python 标准库包 collections 里定义了一个 deque 类，实现前面说的双端队列，可看作栈和队列的推广，支持两端的高效插入和删除操作
- 使用 deque，应从 collections 导入

  ```
  from collections import deque
  dequ1 = deque()
  ```

- deque 创建可以有两个（带默认值的）参数 deque([iter [,maxlen])
  - iter 可为任何可迭代对象，其值将作为 deque 里元素序列
  - maxlen 为 deque 指定最大元素个数，默认为不限，自动增长
- deque 最重要的操作是两端插入和删除
  - append(.) 和 appendleft(.)
  - pop(.) 和 popleft(.)
  - 其他操作请查阅标准库手册 data types/collections

Python 标准库的 deque 类

- deque 实现保证两端插入和删除的 O(1) 复杂性（不是分期付款式）
- 显然，如果给定 maxlen，这个性质很容易做到

  前面介绍的环形队列就能支持这种要求
- 如果不限 maxlen，就需要采用类似双链表的技术

  如果考虑提高存储效率等问题，可以基于双链表扩充

  实际的双端队列常用这类技术实现
- 另外，标准库的 deque 还保证 thread-safe 性质
  - 也就是说，可以安全地用于多线程的程序
  - 多个线程使用同一个 deque，而且它们都在运行中，各自独立地根据需要去操作和使用这个 deque。线程安全性保证在这种情况下该 deque 不会乱套了，还是按 deque 的性质活动
  - 这方面问题这里不讨论了

本章总结

- 栈和队列都是最常用的缓存数据结构，常用于临时保存中间数据，为实现算法和其他数据结构服务
  - 栈和队列分别支持两种最常用的存储和访问顺序
  - 栈支持后进先出的元素访问，队列支持先进先出
- 栈和队列都可以基于线性表的实现技术实现
  - 栈在一端操作，用连续表或链接表技术实现都非常方便
  - 队列在两端操作，为了操作的效率，用链接表实现时需要尾结点引用，用连续表实现需要用环形表技术
- 栈和队列有非常广泛的实际应用
  - 例如，用栈支持表达式求值，用队列支持各种排队服务
  - 用栈和队列支持状态空间搜索是最典型和重要的应用，基于栈或队列实现的搜索分别称为深度优先搜索和宽度优先搜索