高速路网设计

课前知识

1.声明和定义

如何组织好 C 的头文件

变量的声明和定义

- 变量定义：用于为变量 `分配存储空间` ，还可为变量指定初始值。程序中，变量有且仅有一个定义
- 变量声明：用于向程序表明变量的类型和名字。 `不分配存储空间`
- 定义也是声明(定义的同时完成声明)，extern声明不是定义

> extern声明不是定义，也不分配存储空间。事实上它只是说明变量定义在程序的其他地方。程序中变量可以声明多次，但只能定义一次。
>
> 如果声明有初始化式，就被当作定义，即使前面加了extern
>
> ```
> extern double pi=3.141592654;   //定义
> ```

函数的声明和定义

函数的声明和定义区别比较简单，带有{ }的就是定义，否则就是声明。

`头文件只包含声明`

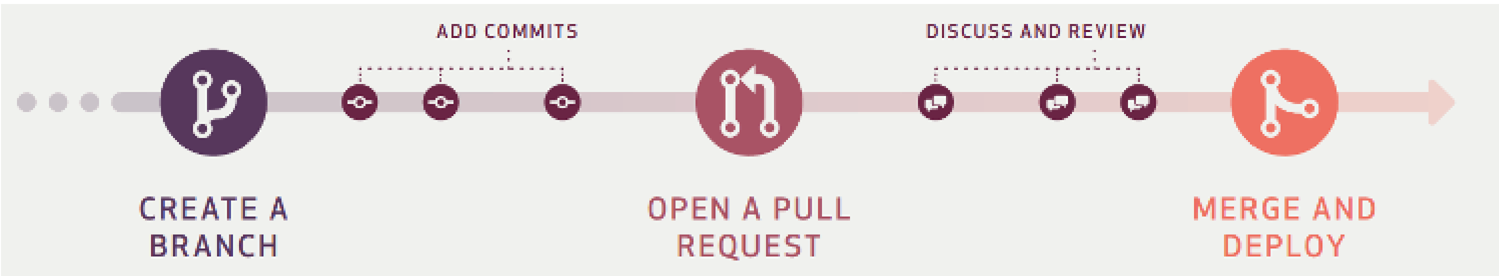
2. Git协作

Git 工作流程

Github flow 是Git flow的简化版，专门配合"持续发布"。它是 Github.com 使用的工作流程。

它只有一个长期分支，就是 `master` ，因此用起来非常简单。

官方推荐的流程如下。



> 第一步：根据需求，从 `master` 拉出新分支，不区分功能分支或补丁分支。
>
> 第二步：新分支开发完成后，或者需要讨论的时候，就向 `master` 发起一个pull request（简称PR）。
>
> 第三步：Pull Request既是一个通知，让别人注意到你的请求，又是一种对话机制，大家一起评审和讨论你的代码。对话过程中，你还可以不断提交代码。
>
> 第四步：你的Pull Request被接受，合并进 `master` ，重新部署后，原来你拉出来的那个分支就被删除。（先部署再合并也可。）

```
# 在github新建仓库
git clone {url}
git checkout -b {yourbranchname}
# coding
git add .
git commit -m
git push
# After a while
git pull
# github上提出pull request
# 仓库的管理负责merge分支到master
```

图的建立和读取

结构体

```c
typedef struct edge_ {
  int v;
  int f;
  int next;
} edge;

typedef struct node_ {
  int industry;
  size_t lx;
  size_t ly;
  int near;
  int nearnum;
} node;

struct State {
  node states[MAXSIZE];
  edge adjvex[MAXESIZE];
  int statenum;
};
```

初始化和删除

```c
void init_State(struct State *s) {
  for (int i = 0; i < MAXSIZE; i++) {
    s->states[i].industry = 0;
    s->states[i].nearnum = 0;
    s->states[i].near = -1;
  }
  s->statenum = 0;
}

void delete_State(struct State *s) { return; }
```

读取图前置工具

```c
// BLACK(255*255*3) -> black
// WHITE(0) -> white
// other -> industry number
int colorpick(struct PXL *p) {
  return (0xff * 0xff * 3 - (int)(p->red * p->red) -
          (int)(p->green * p->green) - (int)(p->blue * p->blue));
}
```

```c
node newnode(size_t lx, size_t ly, int industry) {
  node temp;
  temp.industry = industry;
  temp.lx = lx;
  temp.ly = ly + 6;
  temp.nearnum = 0;
  temp.near = -1;
  return temp;
}
```

```c
bool adjoin(node *state1, node *state2) {
  size_t lx1 = state1->lx;
  size_t ly1 = state1->ly;
  size_t lx2 = state2->lx;
  size_t ly2 = state2->ly;
  if (((lx1 + 8 == lx2) && (ly1 == ly2)) ||
      ((lx1 + 4 == lx2) && (ly1 + 8 == ly2)) ||
      ((lx1 - 4 == lx2) && (ly1 + 8 == ly2))) {
    return true;
  } else {
    return false;
  }
}
```

```c
void insert(int u, int v, struct State *s) {
  edge *adj = s->adjvex;
  edge t = {v, 1, s->states[u].near};
  adj[eid] = t;
  s->states[u].near = eid++;
}
```

读图

```c
  size_t width = p->width;
  size_t height = p->height;
  node *states = s->states;
  for (size_t y = 0; y < height; y++) {
    for (size_t x = 0; x < width; x++) {
      struct PXL pxl = p->image[width * y + x];
      if (x + 1 < width && y + 1 < height && colorpick(&pxl) == BLACK &&
          (colorpick(&p->image[width * (y + 1) + (x - 1)]) == BLACK) &&
          (colorpick(&p->image[width * (y + 1) + (x + 1)]) == BLACK) &&
          (colorpick(&p->image[width * (y + 1) + x]) != WHITE)) {
        struct PXL pxlnode = p->image[width * (y + 1) + x];
        states[s->statenum++] = newnode(x, y, colorpick(&pxlnode));
      }
    }
  }
```

建边

```
for (int i = 0; i < s->statenum; i++) {
    for (int j = i + 1; j < s->statenum; j++) {
      if (adjoin(&states[i], &states[j]) == true) {
        states[i].nearnum++;
        states[j].nearnum++;
        insert(i, j, s);
        insert(j, i, s);
      }
    }
  }
```

parse函数

```
void parse(struct State *s, struct PNG *p) {
  size_t width = p->width;
  size_t height = p->height;
  node *states = s->states;
  for (size_t y = 0; y < height; y++) {
    for (size_t x = 0; x < width; x++) {
      struct PXL pxl = p->image[width * y + x];
      if (x + 1 < width && y + 1 < height && colorpick(&pxl) == BLACK &&
          (colorpick(&p->image[width * (y + 1) + (x - 1)]) == BLACK) &&
          (colorpick(&p->image[width * (y + 1) + (x + 1)]) == BLACK) &&
          (colorpick(&p->image[width * (y + 1) + x]) != WHITE)) {
        struct PXL pxlnode = p->image[width * (y + 1) + x];
        states[s->statenum++] = newnode(x, y, colorpick(&pxlnode));
      }
    }
  }
  for (int i = 0; i < s->statenum; i++) {
    for (int j = i + 1; j < s->statenum; j++) {
      if (adjoin(&states[i], &states[j]) == true) {
        states[i].nearnum++;
        states[j].nearnum++;
        insert(i, j, s);
        insert(j, i, s);
      }
    }
  }
  return;
}
```

最短路算法

数据范围:

$V \le 5000 \ and \ E \le 30000$

稀疏图/稠密图 ¶

若一张图的边数远小于其点数的平方，那么它是一张 **稀疏图 (Sparse graph)**。

若一张图的边数接近其点数的平方，那么它是一张 **稠密图 (Dense graph)**。

这两个概念并没有严格的定义，一般用于讨论 时间复杂度 为 $O(|V|^2)$ 的算法与 $O(|E|)$ 的算法的效率差异（在稠密图上这两种算法效率相当，而在稀疏图上 $O(|E|)$ 的算法效率明显更高）。

1. SPFA

队列优化：SPFA ¶

即 Shortest Path Faster Algorithm。

很多时候我们并不需要那么多无用的松弛操作。

很显然，只有上一次被松弛的结点，所连接的边，才有可能引起下一次的松弛操作。

那么我们用队列来维护“哪些结点可能会引起松弛操作”，就能只访问必要的边了。

```
q = new queue();
q.push(S);
in_queue[S] = true;
while (!q.empty()) {
  u = q.pop();
  in_queue[u] = false;
  for each edge(u, v) {
    if (relax(u, v) && !in_queue[v]) {
      q.push(v);
      in_queue[v] = true;
    }
  }
}
```

虽然在大多数情况下 SPFA 跑得很快，但其最坏情况下的时间复杂度为 $O(NM)$，将其卡到这个复杂度也是不难的，所以考试时要谨慎使用（在没有负权边时最好使用 Dijkstra 算法，在有负权边且题目中的图没有特殊性质时，若 SPFA 是标算的一部分，题目不应当给出 Bellman-Ford 算法无法通过的数据范围）。

2. Dijkstra

时间复杂度：只用分析集合操作，n 次 `delete-min`，m 次 `decrease-key`。

如果用暴力：$O(n^2+m)=O(n^2)$。

如果用堆 $O(m\log n)$。

如果用 priority_queue：$O(m\log m)$。

（注：如果使用 priority_queue，无法删除某一个旧的结点，只能插入一个权值更小的编号相同结点，这样操作导致堆中元素是 $O(m)$ 的）

如果用线段树（ZKW 线段树）：$O(m\log n+n)=O(m\log n)$

如果用 Fibonacci 堆：$O(n\log n+m)$（这就是为啥优秀了）。

堆

```c
void init_heap(struct Heap *s, int (*compare)(struct Node, struct Node)) {
  memset(s->num, 0, sizeof(s->num));
  s->pos = 0;
  s->cmp = compare;
}
```

```c
int heap_empty(struct Heap *s) { return s->pos == 0; }

int heap_insert(struct Heap *s, struct Node x) {
  if (s->pos + 1 == MAX_HEAP_SIZE) {
    return -1;
  }
  s->num[++s->pos] = x;
  int now = s->pos;
  while (now > 1) {
    if (s->cmp(s->num[now], s->num[now >> 1]) > 0) {
      break;
    }
    struct Node temp = s->num[now];
    s->num[now] = s->num[now >> 1];
    s->num[now >> 1] = temp;
    now >>= 1;
  }
  return 0;
}

struct Node heap_top(struct Heap *s) {
  if (!s->pos) {
    return (struct Node){0, 0};
  }
  return s->num[1];
}

int heap_pop(struct Heap *s) {
  if (!s->pos) {
    return -1;
  }
  // int temp = s->num[1];
  s->num[1] = s->num[s->pos--];
  // s->num[s->pos--] = temp;
  int now = 1, next = now << 1;
  while (next <= s->pos) {
    if (next < s->pos && s->cmp(s->num[next], s->num[next | 1]) > 0) {
      next |= 1;
    }
    if (s->cmp(s->num[now], s->num[next]) < 0) {
      break;
    }
    struct Node temp = s->num[now];
    s->num[now] = s->num[next];
    s->num[next] = temp;
    now = next;
    next = next << 1;
  }
  return 0;
}
```

3. A*

https://oi-wiki.org/graph/kth-path/

问题描述

给定一个有 n 个结点，m 条边的有向图，求从 s 到 t 的所有不同路径中的第 k 短路径的长度。

A*算法

A*算法定义了一个对当前状态 x 的估价函数 $f(x) = g(x) + h(x)$，其中 $g(x)$ 为从初始状态到达当前状态的实际代价，$h(x)$ 为从当前状态到达目标状态的最佳路径的估计代价。每次取出 $f(x)$ 最优的状态 x，扩展其所有子状态，可以用 **优先队列** 来维护这个值。

在求解 k 短路问题时，令 $h(x)$ 为从当前结点到达终点 t 的最短路径长度。可以通过在反向图上对结点 t 跑单源最短路预处理出对每个结点的这个值。

由于设计的距离函数和估价函数，对于每个状态需要记录两个值，为当前到达的结点 x 和已经走过的距离 $g(x)$，将这种状态记为 $(x, g(x))$。

开始我们将初始状态 $(s, 0)$ 加入优先队列。每次我们取出估价函数 $f(x) = g(x) + h(x)$ 最小的一个状态，枚举该状态到达的结点 x 的所有出边，将对应的子状态加入优先队列。当我们访问到一个结点第 k 次时，对应的状态的 $g(x)$ 就是从 x 到该结点的第 k 短路。

优化：由于只需要求出从初始结点到目标结点的第 k 短路，所以已经取出的状态到达一个结点的次数大于 k 次时，可以不扩展其子状态。因为之前 k 次已经形成了 k 条合法路径，当前状态不会影响到最后的答案。

当图的形态是一个 n 元环的时候，该算法最坏是 $O(nk \log n)$ 的。但是这种算法可以在相同的复杂度内求出从起始点 s 到每个结点的前 k 短路。

参考程序

老师的：https://paste.ubuntu.com/p/mkTn6nH3ct/

我的：https://pastebin.skymoon.top/?609a2a3fa5f90424#8LiUK39nsM2Rxx2j6kyQ1u2Pm8WR9BenrnqiLjq6zFt2

刘建妙大佬的：https://pastebin.skymoon.top/?b81a63d5bf141e02#GsMsW1fqB6Ucq9pYtaZi4KoNJVjmEbsKST4MQjMWXDHm