

THE CHINESE UNIVERSITY OF HONG KONG, SHENZHEN

---

## CSC4120 Project: Pandemic Control

---

*Group Members:*

Derong JIN 120090562

Yutong FENG 120090266

Shijie WANG 120090331

December 18, 2022



# 1 Part 1

## Answers to Therotical questions (Part 1.1, 1.2, 1.4)

1.1. Suppose the root  $r$  of the tree corresponds to level 0 and the level increases by one when going down the tree. Then, in the  $i^{th}$  round, the government should always be to vaccinate a vertex at level  $i$  in the tree. **This argument can be proved in 3 steps:**

1. In the  $i^{th}$  round, vertices at level  $i' < i$  is either saved by one of its ancestors, or is infected by the virus. So vaccinating vertices at level  $i'$  contributes nothing.
2. Vaccinating vertices at level  $i' > i$  always produce a worse solution than that at level  $i$ . Suppose there is a vertex  $v$  at level  $i' > i$  and vaccinating will save it (it has no vaccinated ancestors), then it must have an ancestor, say  $a_i$ , who is at level  $i$  and is not infected in the  $i^{th}$  round (because in the  $i^{th}$  round, infected vertices are at maximum level of  $i - 1$ ). Vaccinating ancestor  $a_i$  is always better than vaccinating  $v$ , because the subtree of  $v$  is totally included into the subtree of  $a_i$ , and the size of subtree of  $a_i$  is strictly greater than that of  $v$ .
3. Vaccinating vertices at level  $i$  is always possible, unless the pandemic stops spreading before the  $i^{th}$  round. The reason is because if the pandemic does not reach the leaf node of a chain (a path from root to a leaf node) and there is no vaccinated vertices on the chain, then the vertex at the level less than  $i$  is infected, and the vertex at the level of  $i$  is not. The vertices at level  $i$  are always able to be vaccinated.

1.2. Consider a tree  $T_L$  described as following:

1. Define a tree structure  $t_m$  who has  $m$  individual vertices linked directly to the root.

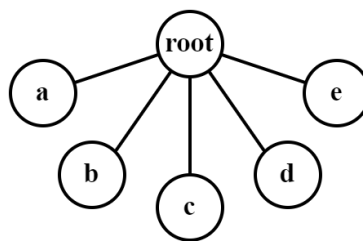
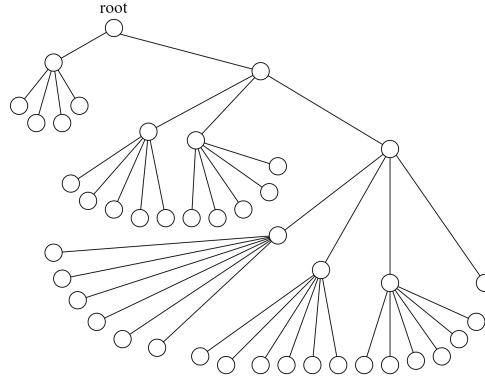


Figure 1: An example of  $t_5$ , where 5 vertices are linked to the root directly and form a tree of size 6.

2. In the tree  $T_L$ , there exists a chain of length  $L$ , which consists of  $L + 1$  vertices with depth ranging from 0 to  $L$  (including the root and the leaf node). For the vertex of depth  $0 \leq i < L$  on the chain, it has  $i + 2$  neighbours, one of them is the vertex with depth  $i + 1$  on the chain, the other  $i + 1$  neighbours each are a root of an instance of  $t_{i+4}$ , where  $t_m$  is defined earlier. For example,  $T_0$  only contains a root it self,  $T_1$  is a  $t_4$  and a independent vertex linked to a root, and  $T_2$  contains extends  $T_1$  by enlarging the length of the chain by 1 and links 2  $t_5$  to the second vertex on the chain.

Figure 2: An example of  $T_3$ 

It is concluded that the number of vertices in  $T_L$ , say  $|T_L|$ , is:

$$\begin{aligned} |T_L| &= 1 + \sum_{i=0}^{L-1} ((i+1) \times |t_{i+4}| + 1) \\ &= 1 + \sum_{i=0}^{L-1} ((i+1)(i+5) + 1) \\ &= \frac{1}{6} (2L^3 + 15L^2 + 19L + 6) \end{aligned} \tag{1}$$

3. Consider the number of saved vertices if applying ‘degree-heavy’ algorithm on  $T_L$ : In the first round, the government will choose to vaccinate the  $t_5$  linked to the root, and then the next round it will choose to save a  $t_6$ , and then  $t_7 \dots$  In the  $i^{th}$  round ( $1 \leq i \leq L$ ) the government saves a  $t_{i+3}$ , and in the  $(L+1)^{th}$  round it saves a single vertex, then the pandemic is over. Hence, the number of vertices saved by this ‘degree-heavy’ greedy algorithm is:

$$\begin{aligned} S_{\text{greedy}} &= 1 + \sum_{i=1}^L |t_{i+3}| \\ &= 1 + \sum_{i=1}^L (i+4) \\ &= 1 + \frac{9L}{2} + \frac{L^2}{2} \end{aligned} \tag{2}$$

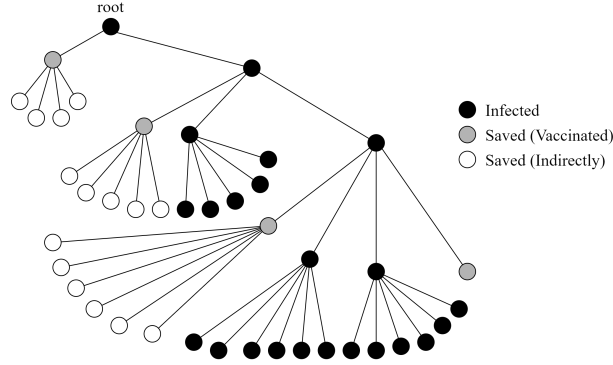


Figure 3: The solution given by ‘degree-heavy’ greedy algorithm on  $T_3$

4. Consider another possible solution, say “**plan B**”: in the first round, the government save the second vertex on the chain, and in the second round, save any leaf node in the  $t_4$ , then the pandemic is over in 2 rounds and only 5 vertices will be infected, and thus  $S_{\text{plan B}} = |T_L| - 5$  vertices are saved.

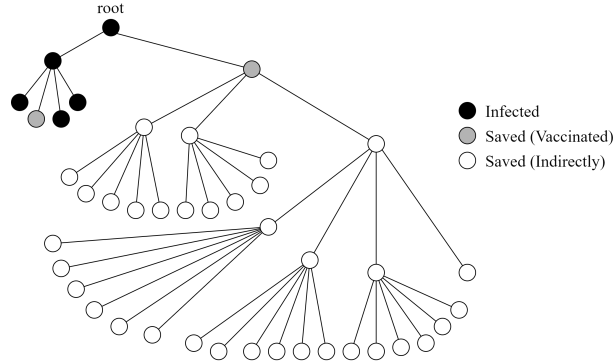


Figure 4: The solution given by “plan B” on  $T_3$

5. “**Plan B**” gives a lowerbound of the optimal solution, i.e.,  $S_{\text{optimal}} \geq S_{\text{plan B}}$ . Consider the value of  $\frac{S_{\text{greedy}}}{S_{\text{optimal}}}$  on such tree  $T_L$ :

$$\begin{aligned}
 \frac{S_{\text{greedy}}}{S_{\text{optimal}}} &= \frac{1 + \frac{9L}{2} + \frac{L^2}{2}}{S_{\text{optimal}}} \\
 &\leq \frac{1 + \frac{9L}{2} + \frac{L^2}{2}}{S_{\text{plan B}}} \\
 &= \frac{6 + 27L + 3L^2}{(2L^3 + 15L^2 + 19L + 6)}
 \end{aligned} \tag{3}$$

Which converges to 0 as  $L$  goes to the infinity. Hence, we can conclude that there does not exist a constant  $c \in (0, 1]$  such that ‘degree-heavy’ greedy algorithm saves a guaranteed fraction  $c$  of the number of vertices saved by the optimal solution.

- 1.4. **optional:** We shall show that the subtree-heavy algorithm heuristic saves at least  $\frac{1}{2}$  of the number of vertices saved by the optimal solution, i.e.,  $S_{\text{greedy}} \geq \frac{1}{2}S_{\text{optimal}}$ .



*Proof.* Denote  $V_{\text{optimal}}$  as the set of vertices saved by the optimal solution, and  $V_{\text{greedy}}$  as the set of vertices saved by subtree-heavy algorithm. Hence,  $S_{\text{optimal}} = |V_{\text{optimal}}|$  and  $S_{\text{greedy}} = |V_{\text{greedy}}|$ . Consider the following division:

$$\begin{cases} V_{\text{optimal}}^A = V_{\text{optimal}} \cap V_{\text{greedy}} \\ V_{\text{optimal}}^B = V_{\text{optimal}} \setminus V_{\text{greedy}} \end{cases} \quad (4)$$

Where  $V_{\text{optimal}}^A$  is the vertices who are saved by both strategies and  $V_{\text{optimal}}^B$  are the vertices who are saved only by optimal solution. This implies:

1.  $V_{\text{optimal}}^A \cap V_{\text{optimal}}^B = \emptyset$
2.  $V_{\text{optimal}}^A \cup V_{\text{optimal}}^B = V_{\text{optimal}}$
3.  $|V_{\text{optimal}}| = |V_{\text{optimal}}^A| + |V_{\text{optimal}}^B|$

Denote  $S_{\text{optimal}}^A = |V_{\text{optimal}}^A|$ ,  $S_{\text{optimal}}^B = |V_{\text{optimal}}^B|$ , and  $S_{\text{optimal}} = |V_{\text{optimal}}|$ . We shall show that  $S_{\text{greedy}} \geq S_{\text{optimal}}^A$  and  $S_{\text{greedy}} \geq S_{\text{optimal}}^B$  both hold, which implies  $S_{\text{greedy}} \geq \frac{1}{2}S_{\text{optimal}}$ , and then we are done.

To show  $S_{\text{greedy}} \geq S_{\text{optimal}}^A$ , we just need to consider their definition.

Since  $(V_{\text{optimal}} \cap V_{\text{greedy}}) \subseteq V_{\text{greedy}}$ , the cardinality of  $V_{\text{optimal}}^A$  is no greater than  $V_{\text{greedy}}$ , which is equivalent to say  $S_{\text{greedy}} \geq S_{\text{optimal}}^A$

To show  $S_{\text{greedy}} \geq S_{\text{optimal}}^B$ , we first briefly discuss about the idea and prove it more formally later.

From Part 1.1 we know that the government saves a vertex at level  $i$  in the  $i^{\text{th}}$  round, and this implies that the set of saved vertices consists of all vertices from some subtrees, and the roots of these subtrees are at different levels. By definition,  $V_{\text{optimal}}^B$  consists of the vertices that are not saved by subtree-heavy algorithm but saved by the optimal solution. So the saved subtrees in  $V_{\text{optimal}}^B$  is not selected by the greedy algorithm and thus must be an option which can be chosen by the greedy algorithm in a certain round. However, the greedy algorithm did not choose them when saving these subtrees is a valid option. This fact implies that the sizes of these subtrees are not greater than those chosen by the greedy algorithm at that round. Hence, every subtree contributes to  $V_{\text{optimal}}^B$  is smaller than one of the subtrees in  $V_{\text{greedy}}$ , and these larger subtrees in  $V_{\text{greedy}}$  have no intersections between each other. The cardinality of  $V_{\text{optimal}}^B$  is smaller the sum of the sizes of saved subtrees, so we conclude that  $|V_{\text{optimal}}^B| \leq |V_{\text{greedy}}|$ , which is equivalent to  $S_{\text{optimal}}^B \leq S_{\text{greedy}}$

Formally speaking, let  $T_v \subseteq V, v \in V$  denote the set of all the vertices in the subtree rooted at  $v$ . And denote a strategy by a sequence of vertices where the  $i^{\text{th}}$  element represent the selected vertex in the  $i^{\text{th}}$  round.

The optimal solution is denoted by  $\{v_i^{\text{Opt}}\}$ , where  $i \in \{1, 2, \dots, \text{MAX}^{\text{Opt}}\}$ ,  $\text{depth}[v_i^{\text{Opt}}] = i$ .

The greedy approximation is  $\{v_i^{\text{G}}\}$ , where  $i \in \{1, 2, \dots, \text{MAX}^{\text{G}}\}$ ,  $\text{depth}[v_i^{\text{G}}] = i$ .

Hence, the set of vertices saved by the optimal solution is:

$$V_{\text{optimal}} = \bigcup_{1 \leq i \leq \text{MAX}^{\text{Opt}}} T_{v_i^{\text{Opt}}}$$



Similarly, the set of vertices saved by greedy algorithm:

$$V_G = \bigcup_{1 \leq i \leq \text{MAX}^G} T_{v_i^G}$$

From analysis, it is shown that  $T_{v_i^{\text{Opt}}} \subseteq V_G$  contributes nothing to  $V_{\text{optimal}}^B$ . So we define a set of index  $I_B$  where  $T_{v_i^{\text{Opt}}}, i \in I_B$  is not a subset of  $V_G$ , i.e.,

$$\begin{aligned} T_{v_i^{\text{Opt}}} \setminus V_G &\neq \emptyset, \quad \forall i \in I_B \\ T_{v_i^{\text{Opt}}} &\subseteq T_{v_{i'}^G} \subseteq V_G, \quad \forall i \notin I_B, \quad \exists i' \in \{1, 2, \dots, \text{MAX}^G\} \end{aligned}$$

It is easy to show  $\max I_B \leq \text{MAX}^G$ :

1. let  $i = \max I_B$
2.  $(T_{v_i^{\text{Opt}}} \setminus V_G \neq \emptyset) \Rightarrow$  The greedy algo. have vertices to save in the  $i^{\text{th}}$  round.
3. Hence,  $\max I_B \leq \text{MAX}^G$

Then, with the help of  $I_B$ ,  $V_{\text{optimal}}^B$  can be written as:

$$\begin{aligned} V_{\text{optimal}}^B &= V_{\text{optimal}} \setminus V_G \\ &= \left( \bigcup_{1 \leq i \leq \text{MAX}^{\text{Opt}}} T_{v_i^{\text{Opt}}} \right) \setminus V_G \\ &= \bigcup_{1 \leq i \leq \text{MAX}^{\text{Opt}}} (T_{v_i^{\text{Opt}}} \setminus V_G) \\ &= \bigcup_{i \in I_B} (T_{v_i^{\text{Opt}}} \setminus V_G) \end{aligned}$$

And if we apply one step of scaling, we obtain the upperbound of  $S_{\text{optimal}}^B$ :

$$V_{\text{optimal}}^B \subseteq \bigcup_{i \in I_B} T_{v_i^{\text{Opt}}} \tag{5}$$

we name the superset on the right hand side  $V_{\text{super}}^B$ . Therefore,  $|V_{\text{super}}^B| \geq |V_{\text{optimal}}^B| = S_{\text{optimal}}^B$

Next consider each  $T_{v_i^{\text{Opt}}}$  such that  $i \in I_B$ . In the  $i^{\text{th}}$  round ( $i \in I_B$ ), the greedy algorithm have at least two choices: to save  $T_{v_i^{\text{Opt}}}$  or to save  $T_{v_i^G}$ , the first choice exists because  $v_i^{\text{Opt}}$  is never protected in the greedy algo., and the second choice exists because  $i \leq \text{MAX}^G$  is shown earlier. The greedy algorithm selected  $T_{v_i^G}$  shows that  $|T_{v_i^{\text{Opt}}}| \leq |T_{v_i^G}|$ . Additionally, from Part 1.1, we know that:

$$\begin{cases} T_{v_i^{\text{Opt}}} \cap T_{v_j^{\text{Opt}}} = \emptyset, & \forall i \neq j \\ T_{v_i^G} \cap T_{v_j^G} = \emptyset, & \forall i \neq j \end{cases}$$



Therefore, the cardinality of  $V_{\text{super}^B}$  is:

$$\begin{aligned} |V_{\text{super}^B}| &= \left| \bigcup_{i \in I_B} T_{v_i^{\text{Opt}}} \right| \\ &= \sum_{i \in I_B} |T_{v_i^{\text{Opt}}}| \end{aligned}$$

Because we shown that  $v_i^G$  exists,  $\forall i \in I_B$ , by applying 2 steps of inequality scaling:

$$\begin{aligned} |V_{\text{super}^B}| &= \sum_{i \in I_B} |T_{v_i^{\text{Opt}}}| \\ &\leq \sum_{i \in I_B} |T_{v_i^G}| \\ &\leq \sum_{1 \leq i \leq \text{MAX}^G} |T_{v_i^G}| \\ &= \left| \bigcup_{1 \leq i \leq \text{MAX}^G} T_{v_i^G} \right| \\ &= |V_G| \end{aligned}$$

Hence, we have shown that:

$$|V_G| \geq |V_{\text{super}^B}| \geq |V_{\text{optimal}^B}|$$

Therefore,  $S_{\text{greedy}} \geq S_{\text{optimal}^B}$  is proved.

In conclusion, it is shown that

$$\begin{cases} S_{\text{greedy}} \geq S_{\text{optimal}^A} \\ S_{\text{greedy}} \geq S_{\text{optimal}^B} \\ S_{\text{optimal}} = S_{\text{optimal}^A} + S_{\text{optimal}^B} \end{cases}$$

finally,  $S_{\text{greedy}} \geq \frac{1}{2} S_{\text{optimal}}$  is proved.

Q. E. D.

### Implementation of both greedy algorithms (Part 1.3)

Because there is only one selection operation at one level, the process will be much like that of a Breadth-first search (BFS), hence we develop a “layer-based BFS”, of which the procedure is shown as below:

**Algorithm 1:** Layered Breadth-first search**Data:**  $r$  : The root of the tree

Select\_Verx: selects a vertex to vaccinate

 $Q \leftarrow \text{Queue}(V[r].\text{children});$ **for**  $i \leftarrow 1$  **to**  $\text{Max\_Depth}$  **do** simulate the pandemic of  $i^{\text{th}}$  round     $\text{chosen\_idx} \leftarrow \text{call Select\_Vertex on } Q;$     **while**  $\neg Q.\text{empty}() \wedge Q.\text{front}().\text{depth} = i$  **do**         $x \leftarrow Q.\text{pop}();$         **if**  $x \neq \text{chosen\_idx}$  **then**             $V[x].\text{set\_infected}();$             **foreach**  $y \in V[x].\text{children}$  **do**                 $Q.\text{add}(y);$             **end**        **end**    **end****end**

Assume the procedure of Select\_Verx takes  $O(|V|)$  in total, this algorithm takes  $O(|V|)$  times, where  $V$  is the set of vertices, which is the same as regular BFSs, since  $|E| = |V| - 1$ .

After this framework is established, the only work to be done is implement the function of Select\_Verx in ‘degree-heavy’ and ‘subtree-heavy’ manners. Before each step of the “for” loop, the elements in the queue are the vertices at a certain level on the tree.

The C++ program is named “part1.cpp” and the content is shown in **Appendix B**. section. To compile the program, use command:

```
g++ part1.cpp -o part1 -std=c++11
```

And the details of implementation of different strategies are shown as below:

1. **‘Degree-heavy’ greedy approach:** the program iterate through the frontier queue and pick the one with the largest degree as the next vaccinated vertex.

To run the program using this strategy, add `-d` flag and use `-t` to specify the path of test case:

```
./part1 -d -t <path_to_test_file>
```

**Demo Output**

```
(base) PS C:\Users\14591\Desktop\Project\src> g++ part1.cpp -o part1
(base) PS C:\Users\14591\Desktop\Project\src> ./part1 -d -t '..\test data\testCase.txt'
Reading test file: done.
Simulating using degree-heavy strategy
[round 1]: vaccinate 2
[round 2]: vaccinate 8
[round 3]: vaccinate 12
9 people are saved in total.
```

2. **‘Subtree-heavy’ greedy approach:** the program iterate through the frontier queue and pick the one with the heaviest subtree weight as the next vaccinated vertex.





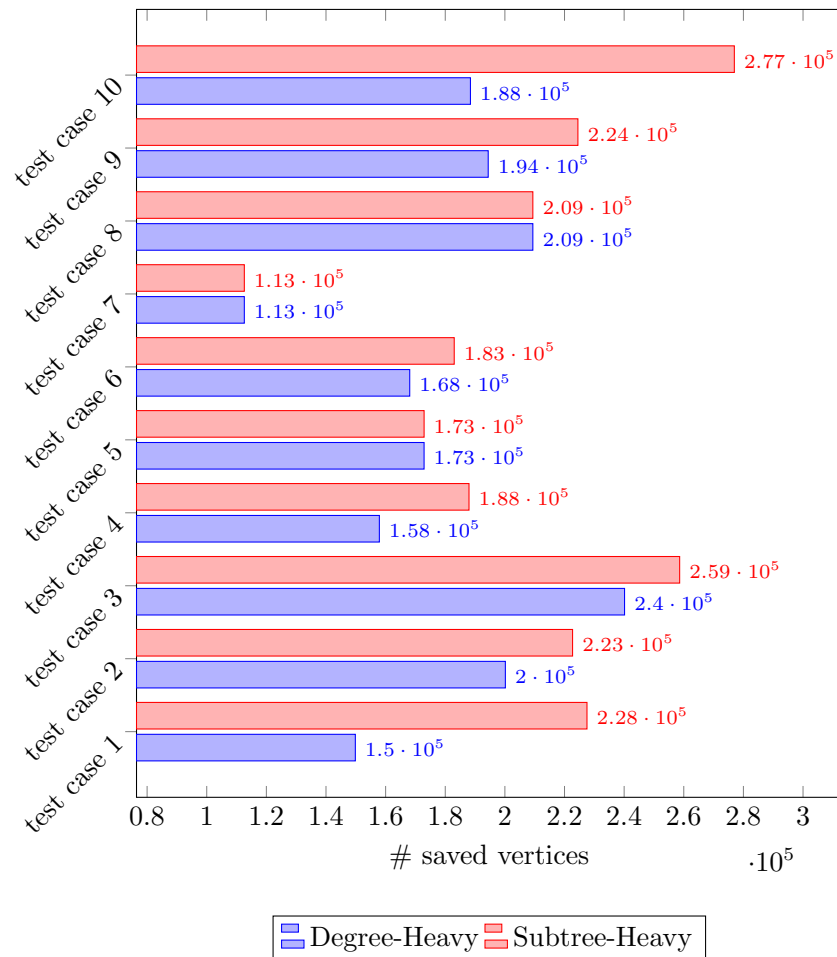
To run the program using this strategy, add `-w` flag and use `-t` to specify the path of test case:

```
./part1 -w -t <path_to_test_file>
```

### Demo Output

```
(base) PS C:\Users\14591\Desktop\Project\src> g++ part1.cpp -o part1
(base) PS C:\Users\14591\Desktop\Project\src> ./part1 -w -t '..\test data\testCase.txt'
Reading test file: done.
Simulating using subtree-heavy strategy
[round 1]: vaccinate 3
[round 2]: vaccinate 4
7 people are saved in total.
```

The performance of these two strategies on the given 10 testcases is illustrated as below:



test case #	1	2	3	4	5	6	7	8	9	10
Degree-heavy	149841	200161	240139	157899	172898	168086	112606	209379	194408	188446
Subtree-heavy	227538	222704	258622	187968	172904	183009	112607	209379	224494	276960

Table 1: Number of saved vertices

It is seen that in most cases when the number of vertices is large, the degree-heavy approach performs worse than the performance by subtree-heavy approach.



## 2 Part 2

### Assumptions

1. The vertex chosen in the  $i^{th}$  round is at the level of  $i$  on the tree.
2. In each round, the player is forced to choose one unvaccinated vertex to save, even if saving this subtree benefits more to its opponent than to itself.
3. Initially, the color of each vertex is randomly generated, and the difference in the number of vertices of both colors does not exceeds 1 in any tree instance.

**NOTE:** The assumption of 1. is to simplify the simulation and thus is unnecessary. The removal version of the game can be seen in section **Appendix A..**

- The ‘selfish’ strategy (SS): choose the node that has the largest difference of (# same color descendants - # different color descendants).
- The ‘subtree-heavy’ strategy (SH) with no color priority: this is same as discussed earlier. A player acts altruistically and is not concerned to save more individuals of the same color.

### Implementation & Program Usage

The implementation is based on that of part 1 and have little change in idea after adding the players into the decision part. And we shuffle the color sequence of vertices in order to generate a random game with equal number on both sides (i.e., red & blue). The program is saved in `part2.cpp` and can be compiled using the command:

```
g++ part2.cpp -o game
```

To simulate the game, using command:

```
./game -s <strategy_combination> -t <path_to_the_test_file>
```

Valid combinations of strategy includes:

SSSS: red selfish, blue selfish

SSSH: red selfish, blue altruistic

SHSS: red altruistic, blue selfish

SHSH: red altruistic, blue altruistic

For example, to simulate on the tree stored in `./testCase1.txt` with SSSS strategies, use command:

```
./game -s SSSS -t ./testCase1.txt
```



## Experiments

In each experiment, the red and the blue players have the same probability to play first. Since the game is based on the analysis of average payoff of the combination of strategies, and we take the average value as the average payoff in the given 10 tree instances, meanwhile reduce the noise as much as possible. Therefore for each strategy combination (i.e.,  $\langle S_r, S_b \rangle \in \{\langle SS, SS \rangle, \langle SS, SH \rangle, \langle SH, SS \rangle, \langle SH, SH \rangle\}$ ), we repeat the experiment for 1,000 times and take the average payoff.

## Result & Analysis

After the experiments designed above are conducted, we obtain the following 2 payoff matrices.

$$\begin{aligned}\Pi^R &= (\pi_{i,j}^R)_{2 \times 2} = \begin{pmatrix} -0.1511 & 292.9151 \\ -293.3994 & 0.1158 \end{pmatrix} \\ \Pi^B &= (\pi_{i,j}^B)_{2 \times 2} = \begin{pmatrix} 0.1511 & -292.9151 \\ 293.3994 & -0.1158 \end{pmatrix}\end{aligned}\tag{6}$$

To analyze the best movement, we represent the game by the payoff matrix. (since it is a zero-sum game, focusing on the red's payoff  $\pi^R$  is enough)

$$G = \begin{array}{c|cc} & \text{Blue} & \\ & SS & SH \\ \hline \text{Red} & SS & -0.1511 & 292.9151 \\ & SH & -293.3994 & 0.1158 \end{array}\tag{7}$$

The representation of payoff matrix sometimes does not give Nash Equilibrium immediately, hence we look at the game in a probability aspect. Meanwhile red and blue governments are allowed to take any of the strategy  $s \in \{SS, SH\}$ . From the view of red government, it does not know but assume the blue one has a probability of  $b_1$  to take the selfish strategy of  $SS$ , and a probability of  $b_2$  to take the strategy of  $SH$ , where  $0 \leq b_1, b_2 \leq 1$  and  $b_1 + b_2 = 1$ . The view of blue government is similar: the red government has a probability of  $r_1$  to take  $SS$  and of  $r_2$  to take  $SH$ . Once this assumption is made by both governments, the best move of on government is to maximize its possible guaranteed payoff (which is equivalent to minimize the possible guaranteed payoff of its opposite).

Specifically, the goal of the blue government is to minimize red's payoff by assuming all the strategy of red government  $\mathbf{r} = (r_1, r_2)$  is possible. And the corresponding minimum payoff of red is guaranteed as:

$$\min\{\pi_{1,1}^R r_1 + \pi_{2,1}^R r_2, \pi_{1,2}^R r_1 + \pi_{2,2}^R r_2\}$$

and the red government will have no choice but take the minimum of these two, hence, the strategy of red government is to maximize this minimum value. i.e., the object of red is :  $\max \min\{\pi_{1,1}^R r_1 + \pi_{2,1}^R r_2, \pi_{1,2}^R r_1 + \pi_{2,2}^R r_2\}$ . And the problem can be formulated as an LP:



$$\begin{aligned}
& \text{maximize} && p_1 \\
& \text{subject to} && \pi_{1,1}^R r_1 + \pi_{2,1}^R r_2 - p_1 \geq 0 \\
& && \pi_{1,2}^R r_1 + \pi_{2,2}^R r_2 - p_1 \geq 0 \\
& && r_1 + r_2 = 1 \\
& && r_1, r_2 \geq 0
\end{aligned} \tag{8}$$

By solving this LP, the optimal solution is given by  $\tilde{\mathbf{r}} = (r_1, r_2) = (1, 0)$ , and the optimal value is  $p_1 = -0.1511$ . This shows that the maximum guaranteed payoff is  $-0.1511$ , and is ensured by choosing *SS* strategy firmly and never consider about *SH*. Since it is a zero-sum game, it is assume that the opposite will always take the best strategy to maximize its own strategy and minimize its opposite's, hence the best strategy of the red is *SS*. We can further implies the best of the blue one is *SS* by symmetric property, and thus announce the *Nash Equilibrium* is established by  $(SS, SS)$  pair. However, we would like to see what is happening when we consider the problem in the blue's view. And we will go back to the discussion of *Nash Equilibrium* later.

Similarly, the goal of the blue government is to minimize the maximum possible payoff of the red government, which could be formulated as another LP:

$$\begin{aligned}
& \text{minimize} && p_2 \\
& \text{subject to} && \pi_{1,1}^R b_1 + \pi_{1,2}^R b_2 - p_2 \leq 0 \\
& && \pi_{2,1}^R b_1 + \pi_{2,2}^R b_2 - p_2 \leq 0 \\
& && b_1 + b_2 = 1 \\
& && b_1, b_2 \geq 0
\end{aligned} \tag{9}$$

Solving this LP yields the output of optimal solution of  $\tilde{\mathbf{b}} = (b_1, b_2) = (1, 0)$  and the corresponding optimal value is  $\min\{p_2\} = -0.1511$ . This result highly fits our prediction, which means the blue government should no doubt but use the strategy of *SS* to win the game (or win as much as he can).

By observation, LPs of equation 8 and 9 are dual to each other, hence they are ensured to take the same optimum and which represents by applying optimal strategies of both governments, the game enters a state where both cannot have a better result when changing one of their strategies only, which is the *Nash Equilibrium*. And in this game, we found that **the *Nash Equilibrium* is established by both players adopting *SS* strategy**. And the average outcome of 0.1511 should be counted as a experimental noise since the strategy is symmetric and thus should has an expected payoff of 0. And this noise will not affect the overall result since  $0.1511 \ll 29293.3994$ .

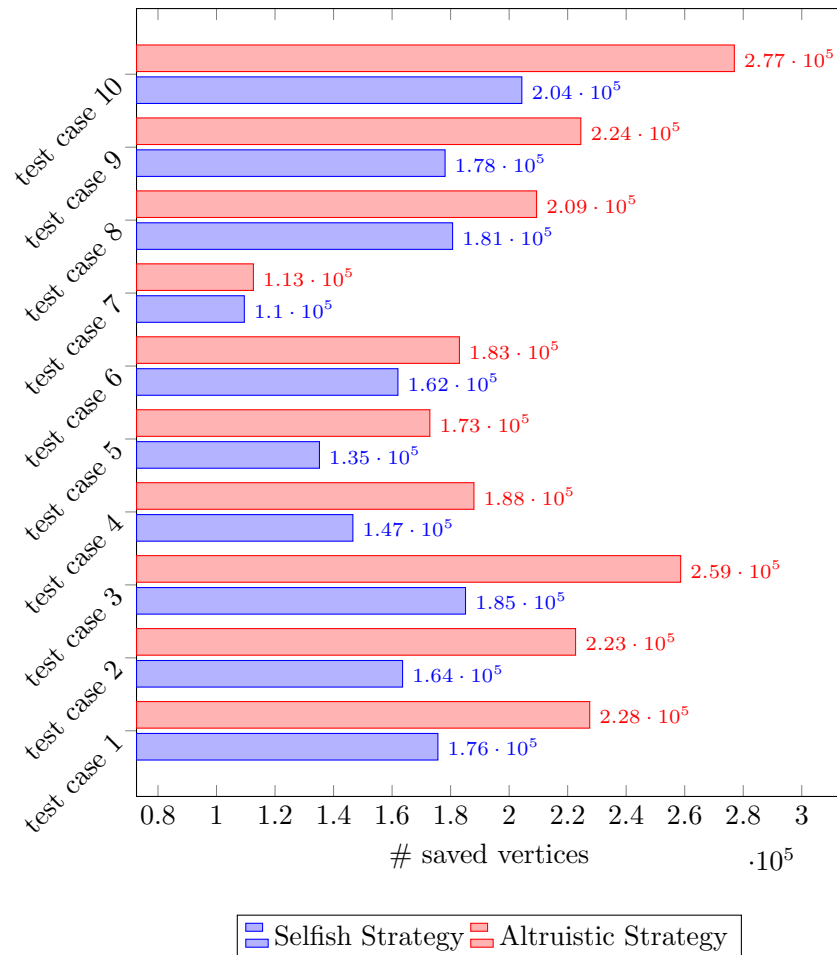
## Nash Equilibrium v.s. Altruism Strategies

We know that in this game, both players are at a equal position to each other, hence any symmetric movement will endup to an excepted draw in the computation. There are two possible movements in this game, so it means that we have an alternative way to reach a tie other than the strategies of *Nash Equilibrium*, and the combination is  $\langle SH, SH \rangle$ , where both players play altruistically and tries to save as much vertices as he can.

The following figure compares the number of saved vertices of both strategy combinations. It is obvious to see that the altruistic strategy ( $\langle SH, SH \rangle$ ) saves much more



vertices than selfish strategy ( $\langle SS, SS \rangle$ ). However, since the goal of the governments is to save as more vertices than its opposite as he can, and this game reaches a unique *NashEquilibrium* at using the selfish strategies. This observation reveals that in this zero-sum game, the competition itself, labels the individuals as different, then block the cooperation, and finally hurt the outcome of the entire society.



test case #	1	2	3	4	5	6	7	8	9	10
Selfish	175656.4	163585.404	185093.0	146598.2	135178.8	161996.1	109501.4	180676.7	178106.6	204340.7
Altruism	227538.0	222704.0	258622.0	187968.0	172904.0	183009.0	112607.0	209379.0	224494.0	276960.0

Table 2: Average number of saved vertices with 1,000 samples



## Appendix A. Game without assumption 1

Previously, we assume the players must select vertex at level  $i$  in the  $i^{th}$  round. This assumption always holds when adopting *SH* strategy (since large subtrees generally assumes lower level). However in *SS* strategy, this assumption not always implies a best step, some unvaccinated subtrees with high level might have large payoffs. And if we want to consider a better choice for *SS* strategy, we define want to remove assumption 1.

This removal leads to new problems in the time complexity of the simulation algorithm. The original time complexity with assumption 1 is  $T = O(|V|)$ . And if this assumption is removed, the time complexity degrades to  $O(|V|^2)$  using brute-force search to find the best payoff subtree. This appendix section adopts a  $O(|V|(\log |V|)^2)$  time simulation method and gives a brief analysis on the average payoff matrix and a comparison with the result we concluded in **Part 2**.

### Method: Segment tree & Heavy path decomposition

From the analysis above, we see that the main cause of the increment in time complexity is that the decision time increases to  $O(|V|)$  from  $O(1)$  on each vertex of each layer. And since there are  $O(|V|)$  decisions to make, the time complexity grows to  $O(|V|^2)$  as a result. To reduce the time complexity as much as possible, the focus should be on the method to reduce the decision cost on one vertex and quickly find out the best subtree of its subtree.

One idea about querying the maximum/minimum value of a certain vertex property in a subtree is to use a segment tree to maintain the dfs preordering sequence of the vertices. That is, by mapping a vertex to its preordering index, we obtain a sequence of vertices, and the subtree of a vertex lies in range  $[\text{pre}_v, \text{post}_v]$  of the preordering sequence. To find the maximum value of the color difference, we just need to maintain a segment tree (storing the information about the weight and subtree payoff in one node) on this preordering sequence and conduct a interval query to find out the optimal answer. To maintain the data structure, we need to update the range  $[\text{pre}_v, \text{post}_v]$  and set them as “saved” vertices, and update every ancestor of  $v$ , where  $v$  is the chosen vertex. This optimization takes  $O(\log |V|)$  times to do the query, and subtree updating, meanwhile takes  $O(\text{depth}_v \times \log |V|)$  to update all the ancestors. So one decision on one vertex overall takes  $O(\text{depth}_v \times \log |V|)$ , this is much more efficient than the brute-force updation.

However, by observation, the bottleneck of this algorithm is at the step of updating the ancestors of the selected vertex. Hence, we adopt **heavy path decomposition** of the tree in this part. This technique allows us to dfs the tree in a ‘heaviest first’ manner, i.e. the heaviest child of the vertex is the first one to dfs. In this way, the heaviest path is a continuous interval of the dfs preordering sequence, meanwhile keeps the subtree continuous as same. This allows us to update a sequence of ancestors at one time.

**Algorithm 2:** Updation of ancestors via heavy path decomposition

---

**Data:**  $v$  : The vertex to vaccinated  
 $chain\_top[\cdot]$  : the top vertex of the chain which  $(\cdot)$  resides  
 $W \leftarrow \text{SEGMENTTREE-QUERY-WEIGHT}(\text{pre}_v)$ ;  
 $C_{sum} \leftarrow \text{SEGMENTTREE-QUERY-COLOR-SUM}(\text{pre}_v)$ ;  
 $\text{SEGMENTTREE-SET-MAINTAINED}(\text{pre}_v, \text{post}_v)$ ;  
 $u \leftarrow v$ ;  
**while**  $u \neq \text{NIL}$  **do**  
     $\text{SEGMENTTREE-ADD-WEIGHT}(\text{pre}_{chain\_top[u]}, \text{pre}_u, -W)$ ;  
     $\text{SEGMENTTREE-ADD-COLOR}(\text{pre}_{chain\_top[u]}, \text{pre}_u, -C_{sum})$ ;  
     $u = \pi[chain\_top[u]]$ ;  
**end**

---

Since there are at most  $O(\log |V|)$  heavy paths, this algorithm finally yields a time complexity of  $O((\log |V|)^2)$  to make one decision.

Hence our solution achieves an overall time complexity of  $O(|V| (\log |V|)^2)$  to simulate the game. The implementation is saved in file `game_complex.cpp` (**Appendix D.**) and the usage is the same as the code of part 2.

## Results

After running on the we obtain the following 2 payoff matrices.

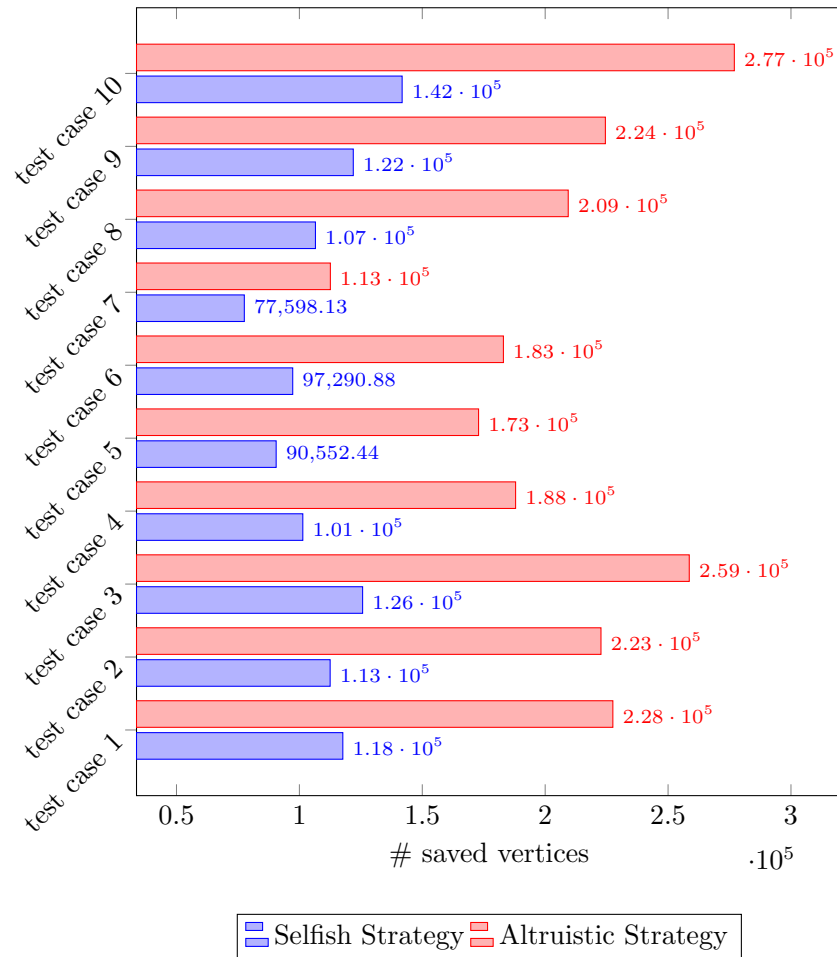
$$\begin{aligned}\Pi^R &= (\pi_{i,j}^R)_{2 \times 2} = \begin{pmatrix} 0.9093 & 412.1404 \\ -409.1442 & 1.4016 \end{pmatrix} \\ \Pi^B &= (\pi_{i,j}^B)_{2 \times 2} = \begin{pmatrix} -0.9093 & -412.1404 \\ 409.1442 & -1.4016 \end{pmatrix}\end{aligned}\tag{10}$$

Compare to the payoff matrices shown in Eq. 6, the absolute payoff by applying asymmetric strategy is much greater, this implies the *Nash Equilibrium* wouldn't change if the first assumption is removed. Nearly all the conclusions are the same as we announced in Part 2. Hence, it is reasonable to make such assumption to simplify the game.

The only different thing to part 2. is that the selfish strategy harms more to the society but enhanced the *Nash Equilibrium* on the other hand. The large difference in payoff infers that allowing players to save less vertices will enhance the imbalance of different strategies. The one who applies selfish strategy earns more, and the one who is altruistic suffers more. This fact further illustrates that the best choice under such zero-sum game is to be selfish, since the penalty of being altruistic is too large. And we can see from the average numbers of saved vertices that, the allowance of saving vertices not at level  $i$  in the  $i^{th}$  round further pushed the selfish player to save less vertices that is not in the same color as the government. And the comparison between the two symmetric acts reveals that although the Equilibrium is further enhanced, it makes more harm to the overall society, because the sum of saved vertices by Equilibrium strategy shrinks to less than one half of that made by SH strategies. Further more, we conclude that such zero-sum game is in essence a controversion to the goal of saving people, since the more



stable the *Nash Equilibrium* is established, the less people both governments save in the end.



test case #	1	2	3	4	5	6	7	8	9	10
Selfish	117675.0	112517.7	125727.0	101395.1	90552.4	97290.9	77598.1	106534.9	121930.1	141806.3
Altruism	227538.0	222704.0	258622.0	187968.0	172904.0	183009.0	112607.0	209379.0	224494.0	276960.0

Table 3: Average number of saved vertices with 1,000 samples

## Appendix B.

Listing 1: Code for Part 1

```

1 #include <queue>
2 #include <vector>
3 #include <fstream>
4 #include <sstream>
5 #include <iostream>
6 #include <algorithm>
7
8 class Vertex
9 {
10 public:
11     int depth = 0; // the depth on the tree
12     int degree_cnt = 0; // number of degree of the node
13     int weight = 0; // number of children nodes in the subtree

```





```
14
15     int infected = 0; // 1: infected, 0: not infected yet
16
17     std::vector<int> children; // children nodes
18 };
19
20 std::vector<Vertex> V;
21
22 // helper function
23 int degree_greedy(std::queue<int> &boundary)
24 {
25     int head = boundary.front(), ans = head;
26
27     do
28     {
29         int x = boundary.front();
30         boundary.pop();
31         boundary.push(x);
32
33         if(V[x].degree_cnt > V[ans].degree_cnt)
34         {
35             ans = x;
36         }
37     } while (head != boundary.front());
38
39     return ans;
40 }
41
42 // helper function
43 int weight_greedy(std::queue<int> &boundary)
44 {
45     int head = boundary.front(), ans = head;
46
47     do
48     {
49         int x = boundary.front();
50         boundary.pop();
51         boundary.push(x);
52
53         if(V[x].weight > V[ans].weight)
54         {
55             ans = x;
56         }
57     } while (head != boundary.front());
58
59     return ans;
60 }
61
62 // build tree
63 void build_tree(int now_idx)
64 {
65     V[now_idx].weight = 1;
66     V[now_idx].degree_cnt = 0;
67
68     for(int child : V[now_idx].children)
69     {
70         V[child].depth = V[now_idx].depth + 1;
71
72         build_tree(child);
73
74         V[now_idx].degree_cnt++;
75         V[now_idx].weight += V[child].weight;
76     }
77 }
78
79 // simulation the vaccine status
80 void simulation_on_tree(int start_idx, int (*select_fn) (std::queue<int> &))
81 {
82     std::queue<int> boundary;
83     V[start_idx].infected = 1;
```



```
86
87     for(int child : V[start_idx].children)
88     {
89         boundary.push(child);
90     }
91
92     for(int depth = 1; !boundary.empty(); depth++)
93     {
94         int choice_idx = select_fn(boundary);
95         std::cout << "[round " << depth << "]: vaccinate " << choice_idx << std::endl;
96         while(!boundary.empty())
97         {
98             int x = boundary.front();
99             if(V[x].depth != depth)
100             {
101                 break;
102             }
103
104             boundary.pop();
105
106             if(x == choice_idx)
107             {
108                 continue;
109             }
110
111             V[x].infected = 1;
112
113             for(int child : V[x].children)
114             {
115                 boundary.push(child);
116             }
117         }
118     }
119 }
120
121 char * getCmdOption(char **first, char **last, const std::string & opt)
122 {
123     char ** iter = std::find(first, last, opt);
124
125     if(iter != last && iter + 1 != last)
126     {
127         return *(iter + 1);
128     }
129
130     return nullptr;
131 }
132
133 bool cmdOptionExists(char **first, char **last, const std::string & opt)
134 {
135     return std::find(first, last, opt) != last;
136 }
137
138 int main(int argc, char **argv)
139 {
140     // parse command lines
141     char * test_file = getCmdOption(argv, argv + argc, "-t");
142     int use_weight_greedy = cmdOptionExists(argv, argv + argc, "-w");
143     int use_degree_greedy = cmdOptionExists(argv, argv + argc, "-d");
144
145     if(use_degree_greedy && use_weight_greedy)
146     {
147         std::cerr << "Please specify with one strategy!" << std::endl;
148         return 1;
149     }
150
151     use_weight_greedy = 1 - use_degree_greedy;
152
153     if(test_file == nullptr)
154     {
155         std::cerr << "Please specify the path of the test file!" << std::endl;
156         return 1;
157     }
158 }
```



```
158
159 // open the test file
160 std::ifstream file_in(test_file);
161 std::string buffer;
162 std::istringstream line;
163
164 if(file_in.fail())
165 {
166     std::cerr << "Error in opening the test case file!" << std::endl;
167     return 1;
168 }
169
170 std::cerr << "Reading test file:";
171 int num_vertices;
172
173 std::getline(file_in, buffer);
174 line.str(buffer);
175 line >> num_vertices;
176
177 V.resize(num_vertices + 1);
178 V[0].infected = 1;
179
180 for(int i = 1; i <= num_vertices && std::getline(file_in, buffer); i++)
181 {
182     int idx, x;
183     line.clear();
184     line.str(buffer);
185     line >> idx;
186
187     while(line >> x)
188     {
189         V[idx].children.push_back(x);
190     }
191 }
192
193 file_in.close();
194
195 std::cerr << " done." << std::endl;
196
197 std::cerr << "Simulating using "
198 << (use_degree_greedy ? "degree-heavy" : "subtree-heavy")
199 << " strategy" << std::endl;
200
201 build_tree(1);
202
203 simulation_on_tree(1, use_degree_greedy ? degree_greedy : weight_greedy);
204
205 int saved_cnt = 0;
206
207 for(const Vertex &v : V)
208 {
209     if(v.infected == 0)
210     {
211         saved_cnt++;
212     }
213 }
214
215 std::cout << saved_cnt << " people are saved in total." << std::endl;
216
217 return 0;
218 }
```



## Appendix C.

Listing 2: Code for Part 2

```
1 #include <queue>
2 #include <vector>
3 #include <chrono>
4 #include <random>
5 #include <cstring>
6 #include <fstream>
7 #include <sstream>
8 #include <iostream>
9 #include <algorithm>
10
11 #define RED -1
12 #define BLUE 1
13
14
15 class Vertex
16 {
17 public:
18     int color = 0; // -1 or 1, red or blue
19     int color_diff = 0;
20     int depth = 0; // the depth on the tree
21     int degree_cnt = 0; // number of degree of the node
22     int weight = 0; // number of children nodes in the subtree
23
24     int infected = 0; // 1: infected, 0: not infected yet
25
26     std::vector<int> children; // children nodes
27 };
28
29 std::vector<Vertex> V;
30
31 class Player
32 {
33 public:
34     int color;
35     int save_cnt;
36     virtual int select_vertex(std::queue<int> &boundary) = 0;
37
38     Player() { }
39     Player(int col) : color(col), save_cnt(0) { }
40 };
41
42 class SelfishPlayer
43     : public Player
44 {
45 public:
46
47     SelfishPlayer() { }
48     SelfishPlayer(int col) : Player(col) { }
49
50     int select_vertex(std::queue<int> &boundary) override
51     {
52         int head = boundary.front(), ans = head;
53
54         do
55         {
56             int x = boundary.front();
57             boundary.pop();
58             boundary.push(x);
59
60             if(V[x].color_diff * color > V[ans].color_diff * color)
61             {
62                 ans = x;
63             }
64
65         } while (head != boundary.front());
66
67         return ans;
```



```
68     }
69 };
70
71 class SubtreeHeavyPlayer
72     : public Player
73 {
74 public:
75     SubtreeHeavyPlayer() { }
76     SubtreeHeavyPlayer(int col): Player(col) { }
77
78     int select_vertex(std::queue<int> &boundary) override
79     {
80         int head = boundary.front(), ans = head;
81
82         do
83         {
84             int x = boundary.front();
85             boundary.pop();
86             boundary.push(x);
87
88             if(V[x].weight > V[ans].weight)
89             {
90                 ans = x;
91             }
92         } while (head != boundary.front());
93
94         return ans;
95     }
96 };
97
98
99
100
101 Player * player[2] = { nullptr, nullptr }; // 0 red, 1 blue
102
103
104 // build tree
105 void build_tree(int now_idx)
106 {
107     V[now_idx].weight = 1;
108     V[now_idx].degree_cnt = 0;
109     V[now_idx].color_diff = V[now_idx].color;
110
111     for(int child : V[now_idx].children)
112     {
113         V[child].depth = V[now_idx].depth + 1;
114
115         build_tree(child);
116
117         V[now_idx].degree_cnt++;
118         V[now_idx].weight += V[child].weight;
119         V[now_idx].color_diff += V[child].color_diff;
120     }
121 }
122
123 // simulation the vaccine status
124 void simulation_on_tree(int start_idx)
125 {
126     std::queue<int> boundary;
127     V[start_idx].infected = 1;
128
129     for(int child : V[start_idx].children)
130     {
131         boundary.push(child);
132     }
133
134     for(int depth = 1, now = std::rand() % 2; !boundary.empty(); depth++, now = 1 - now)
135     {
136         int choice_idx = player[now]->select_vertex(boundary);
137         while(!boundary.empty())
138         {
139             int x = boundary.front();
```



```
140         if(V[x].depth != depth)
141         {
142             break;
143         }
144
145         boundary.pop();
146
147         if(x == choice_idx)
148         {
149             continue;
150         }
151
152         V[x].infected = 1;
153
154         for(int child : V[x].children)
155         {
156             boundary.push(child);
157         }
158     }
159 }
160 }
161
162 void init_vertices(int num_vertices)
163 {
164     unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
165     std::srand(seed);
166
167     int * colors = new int[num_vertices + 1];
168     int off = std::rand() % 2;
169
170     for(int i = 1; i <= num_vertices; i++)
171     {
172         colors[i] = (i + off) % 2 ? RED : BLUE;
173     }
174
175     std::shuffle(colors + 1, colors + num_vertices + 1, std::default_random_engine(seed));
176
177     for(int i = 1; i <= num_vertices; i++)
178     {
179         V[i].color = colors[i];
180     }
181
182     delete[] colors;
183 }
184 }
185
186 int game_strategy(char * arg)
187 {
188     if(arg == nullptr)
189     {
190         return -1;
191     }
192
193     int ret_val = -1;
194
195     if(std::strcmp(arg, "SSSS") == 0)
196     {
197         ret_val = 0;
198     }
199     else if(std::strcmp(arg, "SSSH") == 0)
200     {
201         ret_val = 1;
202     }
203     else if(std::strcmp(arg, "SHSS") == 0)
204     {
205         ret_val = 2;
206     }
207     else if(std::strcmp(arg, "SHSH") == 0)
208     {
209         ret_val = 3;
210     }
```



```
211
212     if(ret_val == -1)
213     {
214         return -1;
215     }
216
217     if(ret_val & 2)
218     {
219         player[0] = new SubtreeHeavyPlayer(RED);
220     }
221     else
222     {
223         player[0] = new SelfishPlayer(RED);
224     }
225
226     if(ret_val & 1)
227     {
228         player[1] = new SubtreeHeavyPlayer(BLUE);
229     }
230     else
231     {
232         player[1] = new SelfishPlayer(BLUE);
233     }
234
235     return ret_val;
236 }
237
238 char * getCmdOption(char **first, char **last, const std::string & opt)
239 {
240     char ** iter = std::find(first, last, opt);
241
242     if(iter != last && iter + 1 != last)
243     {
244         return *(iter + 1);
245     }
246
247     return nullptr;
248 }
249
250 bool cmdOptionExists(char **first, char **last, const std::string & opt)
251 {
252     return std::find(first, last, opt) != last;
253 }
254
255 int main(int argc, char **argv)
256 {
257     std::ios::sync_with_stdio(false);
258
259     // parse command lines
260     char * test_file = getCmdOption(argv, argv + argc, "-t");
261     char * strategy = getCmdOption(argv, argv + argc, "-s");
262     int stat = game_strategy(strategy);
263
264     if(stat == -1)
265     {
266         std::cerr << "Please specify a valid strategy!" << std::endl;
267         return 1;
268     }
269
270     if(test_file == nullptr)
271     {
272         std::cerr << "Please specify the path of the test file!" << std::endl;
273         return 1;
274     }
275
276     // open the test file
277     std::ifstream file_in(test_file);
278     std::string buffer;
279     std::istringstream line;
280
281     if(file_in.fail())
282     {
```



```
283         std::cerr << "Error in opening the test case file!" << std::endl;
284         return 1;
285     }
286
287     std::cerr << "Reading test file:";
288     int num_vertices;
289
290     std::getline(file_in, buffer);
291     line.str(buffer);
292     line >> num_vertices;
293
294     V.resize(num_vertices + 1);
295     V[0].infected = 1;
296
297     for(int i = 1; i <= num_vertices && std::getline(file_in, buffer); i++)
298     {
299         int idx, x;
300         line.clear();
301         line.str(buffer);
302         line >> idx;
303
304         while(line >> x)
305         {
306             V[idx].children.push_back(x);
307         }
308     }
309
310     file_in.close();
311
312     std::cerr << " done." << std::endl;
313
314     std::cerr << "Red Player: " << ((stat & 2) ? "SH" : "SS") << std::endl;
315     std::cerr << "Blue Player: " << ((stat & 1) ? "SH" : "SS") << std::endl;
316
317     init_vertices(num_vertices);
318
319     build_tree(1);
320
321     simulation_on_tree(1);
322
323     int saved_cnt = 0;
324
325     for(const Vertex &v : V)
326     {
327         if(v.infected == 0)
328         {
329             saved_cnt++;
330             if(v.color == RED)
331             {
332                 player[0]->save_cnt++;
333             }
334             else if(v.color == BLUE)
335             {
336                 player[1]->save_cnt++;
337             }
338         }
339     }
340
341     std::cout << saved_cnt << " people are saved in total." << std::endl;
342     std::cout << "Saved Reds: " << player[0]->save_cnt << std::endl;
343     std::cout << "Saved Blues: " << player[1]->save_cnt << std::endl;
344     std::cout << ((player[0]->save_cnt > player[1]->save_cnt) ? "Red" : "Blue") << " wins"
345     " << std::endl;
346
347     delete player[0];
348     delete player[1];
349
350     return 0;
351 }
```





## Appendix D.

Listing 3: Code for the game without the first assumption

```
1 #include <queue>
2 #include <vector>
3 #include <chrono>
4 #include <random>
5 #include <cstring>
6 #include <fstream>
7 #include <sstream>
8 #include <iostream>
9 #include <algorithm>
10
11 #define RED -1
12 #define BLUE 1
13
14 class SegtreeNode
15 {
16 public:
17     // data stored
18     int max_color_diff;
19     int min_color_diff;
20     int max_weight;
21
22     int max_color_index;
23     int min_color_index;
24     int max_weight_index;
25
26     // auxiliary
27     int saved;
28     int save_tag;
29     int color_tag;
30     int weight_tag;
31
32     void to_default()
33     {
34         max_color_diff = -0x3f3f3f3f;
35         min_color_diff = 0x3f3f3f3f;
36         max_weight = 0;
37         saved = 0;
38         color_tag = 0;
39         weight_tag = 0;
40     }
41
42     void set_saved()
43     {
44         to_default();
45         saved = 1;
46         save_tag = 1;
47     }
48
49     void apply_color_tag(int tag)
50     {
51         color_tag += tag;
52         max_color_diff += tag;
53         min_color_diff += tag;
54     }
55
56     void apply_weight_tag(int tag)
57     {
58         weight_tag += tag;
59         max_weight += tag;
60     }
61 };
62
63 class Segtree
64 {
65 public:
66     int length;
67     std::vector<SegtreeNode> tree;
```



```
68
69 Segtree(): length(0) { }
70 Segtree(int seg_length)
71     : length(seg_length)
72 {
73     tree.resize(length * 4 + 1);
74     init(1, length, 1);
75 }
76
77 void init(int l, int r, int now)
78 {
79     tree[now].to_default();
80     if(l == r)
81     {
82         return ;
83     }
84
85     int mid = (l + r) >> 1;
86     init(l, mid, now << 1);
87     init(mid + 1, r, now << 1 | 1);
88 }
89
90 void insert(int idx, int v_index, int color_sum, int weight)
91 {
92     insert(1, this->length, idx, 1, v_index, color_sum, weight);
93 }
94
95 void save_range(int first_index, int last_index)
96 {
97     save_range(1, length, first_index, last_index, 1);
98 }
99
100 void update(int first, int last, int col_diff, int weight_diff)
101 {
102     update_range(1, length, first, last, 1, col_diff, weight_diff);
103 }
104
105 int max_weight_index(int first, int last, int &ans, int &max_val)
106 {
107     ans = -1, max_val = -0x3f3f3f3f;
108
109     query_max_weight(1, length, first, last, 1, max_val, ans);
110
111     return ans;
112 }
113
114 int max_color_diff(int first, int last, int &ans, int &max_val)
115 {
116     ans = -1, max_val = -0x3f3f3f3f;
117
118     query_max_color_diff(1, length, first, last, 1, max_val, ans);
119
120     return ans;
121 }
122
123 int min_color_diff(int first, int last, int &ans, int &min_val)
124 {
125     ans = -1, min_val = 0x3f3f3f3f;
126
127     query_min_color_diff(1, length, first, last, 1, min_val, ans);
128
129     return ans;
130 }
131
132 int is_saved(int index)
133 {
134     return query_saved(1, length, index, 1);
135 }
136
137 const SegtreeNode * snapshot(int index)
138 {
139     return query_everything(1, length, index, 1);
```



```
140     }
141
142 private:
143     void pushup(int now)
144     {
145         tree[now].max_color_diff = std::max(tree[now << 1].max_color_diff, tree[now << 1
146 | 1].max_color_diff);
147         tree[now].max_color_index = (tree[now << 1].max_color_diff > tree[now << 1 | 1].
148 max_color_diff) ?
149         tree[now << 1].max_color_index : tree[now << 1 | 1].max_color_index;
150
151         tree[now].min_color_diff = std::min(tree[now << 1].min_color_diff, tree[now << 1
152 | 1].min_color_diff);
153         tree[now].min_color_index = (tree[now << 1].min_color_diff < tree[now << 1 | 1].
154 min_color_diff) ?
155         tree[now << 1].min_color_index : tree[now << 1 | 1].min_color_index;
156
157         tree[now].max_weight = std::max(tree[now << 1].max_weight, tree[now << 1 | 1].
158 max_weight);
159         tree[now].max_weight_index = (tree[now << 1].max_weight > tree[now << 1 | 1].
160 max_weight) ?
161         tree[now << 1].max_weight_index : tree[now << 1 | 1].max_weight_index;
162     }
163
164     void pushdown(int now)
165     {
166         if(tree[now].save_tag)
167         {
168             tree[now << 1].set_saved();
169             tree[now << 1 | 1].set_saved();
170             tree[now].save_tag = 0;
171         }
172
173         if(tree[now].color_tag)
174         {
175             tree[now << 1].apply_color_tag(tree[now].color_tag);
176             tree[now << 1 | 1].apply_color_tag(tree[now].color_tag);
177             tree[now].color_tag = 0;
178         }
179
180         if(tree[now].weight_tag)
181         {
182             tree[now << 1].apply_weight_tag(tree[now].weight_tag);
183             tree[now << 1 | 1].apply_weight_tag(tree[now].weight_tag);
184             tree[now].weight_tag = 0;
185         }
186     }
187
188     void insert(int l, int r, int i, int now, int v_index, int color, int weight)
189     {
190         if(l == r)
191         {
192             tree[now].max_color_diff = tree[now].min_color_diff = color;
193             tree[now].max_weight = weight;
194             tree[now].max_color_index = tree[now].min_color_index = tree[now].
195 max_weight_index = v_index;
196             return ;
197         }
198         pushdown(now);
199
200         int mid = (l + r) >> 1;
201
202         if(i <= mid)
203         {
204             insert(l, mid, i, now << 1, v_index, color, weight);
205         }
206         else
207         {
208             insert(mid + 1, r, i, now << 1 | 1, v_index, color, weight);
209         }
210     }
211 }
```



```
205     pushup(now);
206 }
207
208 void save_range(int l, int r, int saved_l, int saved_r, int now)
209 {
210     if(saved_l > r || l > saved_r)
211     {
212         return ;
213     }
214
215     if(saved_l <= l && r <= saved_r)
216     {
217         tree[now].set_saved();
218         return ;
219     }
220
221     pushdown(now);
222
223     int mid = (l + r) >> 1;
224
225     save_range(l, mid, saved_l, saved_r, now << 1);
226     save_range(mid + 1, r, saved_l, saved_r, now << 1 | 1);
227
228     pushup(now);
229 }
230
231 void update_range(int l, int r, int upd_l, int upd_r, int now, int upd_color, int
upd_weight)
232 {
233     if(upd_l > r || l > upd_r)
234     {
235         return ;
236     }
237
238     if(upd_l <= l && r <= upd_r)
239     {
240         tree[now].apply_color_tag(upd_color);
241         tree[now].apply_weight_tag(upd_weight);
242         return ;
243     }
244
245     pushdown(now);
246
247     int mid = (l + r) >> 1;
248
249     update_range(l, mid, upd_l, upd_r, now << 1, upd_color, upd_weight);
250     update_range(mid + 1, r, upd_l, upd_r, now << 1 | 1, upd_color, upd_weight);
251
252     pushup(now);
253 }
254
255 void query_max_weight(int l, int r, int query_l, int query_r, int now, int &
max_weight, int & index)
256 {
257     if(query_l > r || l > query_r)
258     {
259         return ;
260     }
261
262     if(query_l <= l && r <= query_r)
263     {
264         if(max_weight < tree[now].max_weight)
265         {
266             max_weight = tree[now].max_weight;
267             index = tree[now].max_weight_index;
268         }
269
270         return ;
271     }
272
273     pushdown(now);
274
```



```
275     int mid = (l + r) >> 1;
276
277     query_max_weight(l, mid, query_l, query_r, now << 1, max_weight, index);
278     query_max_weight(mid + 1, r, query_l, query_r, now << 1 | 1, max_weight, index);
279 }
280
281 void query_max_color_diff(int l, int r, int query_l, int query_r, int now, int &
max_color_diff, int & index)
282 {
283     if(query_l > r || l > query_r)
284     {
285         return ;
286     }
287
288     if(query_l <= l && r <= query_r)
289     {
290         if(max_color_diff < tree[now].max_color_diff)
291         {
292             max_color_diff = tree[now].max_color_diff;
293             index = tree[now].max_color_index;
294         }
295
296         return ;
297     }
298
299     pushdown(now);
300
301     int mid = (l + r) >> 1;
302
303     query_max_color_diff(l, mid, query_l, query_r, now << 1, max_color_diff, index);
304     query_max_color_diff(mid + 1, r, query_l, query_r, now << 1 | 1, max_color_diff,
index);
305 }
306
307 void query_min_color_diff(int l, int r, int query_l, int query_r, int now, int &
min_color_diff, int & index)
308 {
309     if(query_l > r || l > query_r)
310     {
311         return ;
312     }
313
314     if(query_l <= l && r <= query_r)
315     {
316         if(min_color_diff > tree[now].min_color_diff)
317         {
318             min_color_diff = tree[now].min_color_diff;
319             index = tree[now].min_color_index;
320         }
321
322         return ;
323     }
324
325     pushdown(now);
326
327     int mid = (l + r) >> 1;
328
329     query_min_color_diff(l, mid, query_l, query_r, now << 1, min_color_diff, index);
330     query_min_color_diff(mid + 1, r, query_l, query_r, now << 1 | 1, min_color_diff,
index);
331 }
332
333 int query_saved(int l, int r, int i, int now)
334 {
335     if(l == r)
336     {
337         return tree[now].saved;
338     }
339
340     pushdown(now);
341
342     int mid = (l + r) >> 1;
```



```
343
344     if(i <= mid)
345     {
346         return query_saved(l, mid, i, now << 1);
347     }
348
349     return query_saved(mid + 1, r, i, now << 1 | 1);
350 }
351
352 SegtreeNode * query_everything(int l, int r, int i, int now)
353 {
354     if(l == r)
355     {
356         return &tree[now];
357     }
358
359     pushdown(now);
360
361     int mid = (l + r) >> 1;
362
363     if(i <= mid)
364     {
365         return query_everything(l, mid, i, now << 1);
366     }
367
368     return query_everything(mid + 1, r, i, now << 1 | 1);
369 }
370 };
371
372 Segtree * segtree = nullptr;
373
374 class Vertex
375 {
376 public:
377     int color = 0; // -1 or 1, red or blue
378     int color_diff = 0;
379     int depth = 0; // the depth on the tree
380     int degree_cnt = 0; // number of degree of the node
381     int weight = 0; // number of children nodes in the subtree
382
383     int parent = 0;
384
385     int infected = 0; // 1: infected, 0: not infected yet
386
387     int max_child = 0;
388     int chain_top;
389     int pre, post;
390
391     std::vector<int> children; // children nodes
392 };
393
394 std::vector<Vertex> V;
395
396 class Player
397 {
398 public:
399     int color;
400     int save_cnt;
401     virtual int select_vertex(std::queue<int> &boundary) = 0;
402
403     Player() { }
404     Player(int col) : color(col), save_cnt(0) { }
405
406     void update_situation(int x)
407     {
408         const SegtreeNode * it = segtree->snapshot(V[x].pre);
409         int col_diff = it->max_color_diff;
410         int weight = it->max_weight;
411
412         segtree->save_range(V[x].pre, V[x].post);
413
414         int now = x;
```



```
415         while(now != 0)
416         {
417             segtree->update(V[V[now].chain_top].pre, V[now].pre, -col_diff, -weight);
418             now = V[V[now].chain_top].parent;
419         }
420     }
421 };
422
423 class SelfishPlayer
424     : public Player
425 {
426 public:
427
428     SelfishPlayer() { }
429     SelfishPlayer(int col) : Player(col) { }
430
431     int select_vertex(std::queue<int> &boundary) override
432     {
433         int head = boundary.front(), ans = head, weight_ans = (-color) * 0x3f3f3f3f;
434
435         do
436         {
437             int x = boundary.front();
438             boundary.pop();
439             boundary.push(x);
440
441             int ans_x, weight_x;
442
443             if(color == RED)
444             {
445                 segtree->min_color_diff(V[x].pre, V[x].post, ans_x, weight_x);
446                 if(weight_x < weight_ans)
447                 {
448                     weight_ans = weight_x;
449                     ans = ans_x;
450                 }
451             }
452             else
453             {
454                 segtree->max_color_diff(V[x].pre, V[x].post, ans_x, weight_x);
455                 if(weight_x > weight_ans)
456                 {
457                     weight_ans = weight_x;
458                     ans = ans_x;
459                 }
460             }
461
462         } while (head != boundary.front());
463
464         update_situation(ans);
465
466         return ans;
467     }
468 };
469
470
471 class SubtreeHeavyPlayer
472     : public Player
473 {
474 public:
475
476     SubtreeHeavyPlayer() { }
477     SubtreeHeavyPlayer(int col): Player(col) { }
478
479     int select_vertex(std::queue<int> &boundary) override
480     {
481         int head = boundary.front(), ans = head, weight_ans = -0x3f3f3f3f;
482
483         do
484         {
485             int x = boundary.front();
486             boundary.pop();
```



```
487         boundary.push(x);
488
489         int ans_x, weight_x;
490
491         segtree->max_weight_index(V[x].pre, V[x].post, ans_x, weight_x);
492
493         if(weight_x > weight_ans)
494         {
495             weight_ans = weight_x;
496             ans = ans_x;
497         }
498
499
500     } while (head != boundary.front());
501
502     update_situation(ans);
503
504     std::cout << ans << " saves " << weight_ans << std::endl;;
505     return ans;
506 }
507 };
508
509
510 Player * player[2] = { nullptr, nullptr }; // 0 red, 1 blue
511
512 int time_stamp = 0;
513
514 // build tree
515 void build_tree(int now_idx)
516 {
517     V[now_idx].weight = 1;
518     V[now_idx].degree_cnt = 0;
519     V[now_idx].color_diff = V[now_idx].color;
520
521     int max_weight = -1;
522
523     for(int child : V[now_idx].children)
524     {
525         V[child].parent = now_idx;
526         V[child].depth = V[now_idx].depth + 1;
527
528         build_tree(child);
529
530         if(V[child].weight > max_weight)
531         {
532             max_weight = V[child].weight;
533             V[now_idx].max_child = child;
534         }
535
536         V[now_idx].degree_cnt++;
537         V[now_idx].weight += V[child].weight;
538         V[now_idx].color_diff += V[child].color_diff;
539     }
540 }
541
542 void build_segtree(int now_idx, int top_idx)
543 {
544     if(now_idx == 0)
545     {
546         return ;
547     }
548
549     V[now_idx].pre = ++time_stamp;
550     V[now_idx].chain_top = top_idx;
551
552     segtree->insert(V[now_idx].pre, now_idx, V[now_idx].color_diff, V[now_idx].weight);
553
554     build_segtree(V[now_idx].max_child, top_idx);
555
556     for(int child : V[now_idx].children)
557     {
558         if(child == V[now_idx].max_child)
```





```
559     {
560         continue;
561     }
562
563     build_segtree(child, child);
564 }
565
566
567 V[now_idx].post = time_stamp;
568 }
569
570 // simulation the vaccine status
571 void simulation_on_tree(int start_idx)
572 {
573     std::queue<int> boundary;
574     V[start_idx].infected = 1;
575
576     for(int child : V[start_idx].children)
577     {
578         boundary.push(child);
579     }
580
581     for(int depth = 1, now = std::rand() % 2; !boundary.empty(); depth++, now = 1 - now)
582     {
583         std::cout << "[round " << depth << " ] " << (now ? "RED" : "BLUE") << " saves " ;
584         int choice_idx = player[now]->select_vertex(boundary);
585
586         while(!boundary.empty())
587         {
588             int x = boundary.front();
589             if(V[x].depth != depth)
590             {
591                 break;
592             }
593
594             boundary.pop();
595
596             if(segtree->is_saved(V[x].pre))
597             {
598                 continue;
599             }
600
601             V[x].infected = 1;
602
603             for(int child : V[x].children)
604             {
605                 if(! (segtree->is_saved(V[child].pre)) )
606                 {
607                     boundary.push(child);
608                 }
609             }
610         }
611     }
612 }
613
614
615 void init_vertices(int num_vertices)
616 {
617     unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
618     std::srand(seed);
619
620     int * colors = new int[num_vertices + 1];
621     int off = std::rand() % 2;
622
623     for(int i = 1; i <= num_vertices; i++)
624     {
625         colors[i] = (i + off) % 2 ? RED : BLUE;
626     }
627
628     std::shuffle(colors + 1, colors + num_vertices + 1, std::default_random_engine(seed))
629     ;
```



```
630     for(int i = 1; i <= num_vertices; i++)
631     {
632         V[i].color = colors[i];
633     }
634
635     delete[] colors;
636 }
637
638 int game_strategy(char * arg)
639 {
640     if(arg == nullptr)
641     {
642         return -1;
643     }
644
645     int ret_val = -1;
646
647     if(std::strcmp(arg, "SSSS") == 0)
648     {
649         ret_val = 0;
650     }
651     else if(std::strcmp(arg, "SSSH") == 0)
652     {
653         ret_val = 1;
654     }
655     else if(std::strcmp(arg, "SHSS") == 0)
656     {
657         ret_val = 2;
658     }
659     else if(std::strcmp(arg, "SHSH") == 0)
660     {
661         ret_val = 3;
662     }
663
664     if(ret_val == -1)
665     {
666         return -1;
667     }
668
669     if(ret_val & 2)
670     {
671         player[0] = new SubtreeHeavyPlayer(RED);
672     }
673     else
674     {
675         player[0] = new SelfishPlayer(RED);
676     }
677
678     if(ret_val & 1)
679     {
680         player[1] = new SubtreeHeavyPlayer(BLUE);
681     }
682     else
683     {
684         player[1] = new SelfishPlayer(BLUE);
685     }
686
687     return ret_val;
688 }
689
690 char * getCmdOption(char **first, char **last, const std::string & opt)
691 {
692     char ** iter = std::find(first, last, opt);
693
694     if(iter != last && iter + 1 != last)
695     {
696         return *(iter + 1);
697     }
698
699     return nullptr;
700 }
701
```



```
702 bool cmdOptionExists(char **first, char **last, const std::string & opt)
703 {
704     return std::find(first, last, opt) != last;
705 }
706
707 int main(int argc, char **argv)
708 {
709     std::ios::sync_with_stdio(false);
710
711     // parse command lines
712     char * test_file = getCmdOption(argv, argv + argc, "-t");
713     char * strategy = getCmdOption(argv, argv + argc, "-s");
714     int stat = game_strategy(strategy);
715
716     if(stat == -1)
717     {
718         std::cerr << "Please specify a valid strategy!" << std::endl;
719         return 1;
720     }
721
722     if(test_file == nullptr)
723     {
724         std::cerr << "Please specify the path of the test file!" << std::endl;
725         return 1;
726     }
727
728     // open the test file
729     std::ifstream file_in(test_file);
730     std::string buffer;
731     std::istringstream line;
732
733     if(file_in.fail())
734     {
735         std::cerr << "Error in opening the test case file!" << std::endl;
736         return 1;
737     }
738
739     std::cerr << "Reading test file:";
740     int num_vertices;
741
742     std::getline(file_in, buffer);
743     line.str(buffer);
744     line >> num_vertices;
745
746     V.resize(num_vertices + 1);
747     V[0].infected = 1;
748
749     for(int i = 1; i <= num_vertices && std::getline(file_in, buffer); i++)
750     {
751         int idx, x;
752         line.clear();
753         line.str(buffer);
754         line >> idx;
755
756         while(line >> x)
757         {
758             V[idx].children.push_back(x);
759         }
760     }
761
762     file_in.close();
763
764     std::cerr << " done." << std::endl;
765
766     std::cerr << "Red Player: " << ((stat & 2) ? "SH" : "SS") << std::endl;
767     std::cerr << "Blue Player: " << ((stat & 1) ? "SH" : "SS") << std::endl;
768
769     init_vertices(num_vertices);
770     segtree = new Segtree(num_vertices);
771
772     build_tree(1);
773 }
```



```
774     build_segtree(1, 1);
775
776
777     simulation_on_tree(1);
778
779     int saved_cnt = 0;
780
781     for(const Vertex &v : V)
782     {
783         if(v.infected == 0)
784         {
785             saved_cnt++;
786             if(v.color == RED)
787             {
788                 player[0]->save_cnt++;
789             }
790             else if(v.color == BLUE)
791             {
792                 player[1]->save_cnt++;
793             }
794         }
795     }
796
797     std::cout << saved_cnt << " people are saved in total." << std::endl;
798     std::cout << "Saved Reds: " << player[0]->save_cnt << std::endl;
799     std::cout << "Saved Blues: " << player[1]->save_cnt << std::endl;
800     std::cout << ((player[0]->save_cnt > player[1]->save_cnt) ? "Red" : "Blue") << " wins"
801     " << std::endl;
802
803     delete player[0];
804     delete player[1];
805     delete segtree;
806
807     return 0;
808 }
```