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.

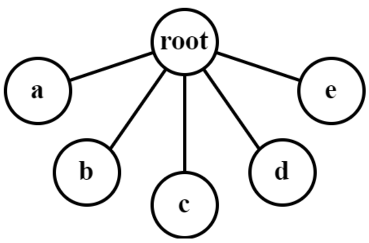


Figure 1: An example of t_5, where 5 vertices are linked to the root directly and form a tree of size 6.

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.

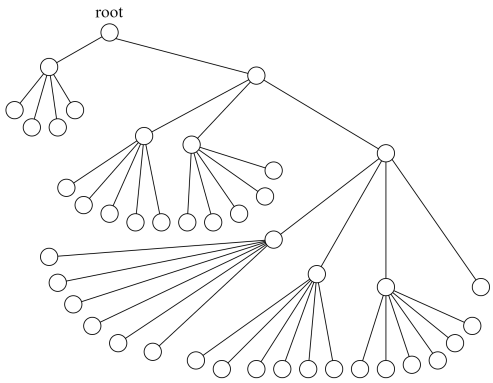


Figure 2: An example of T_3

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

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

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... In the i^{th} round ($1 \le i \le L$) 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:

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

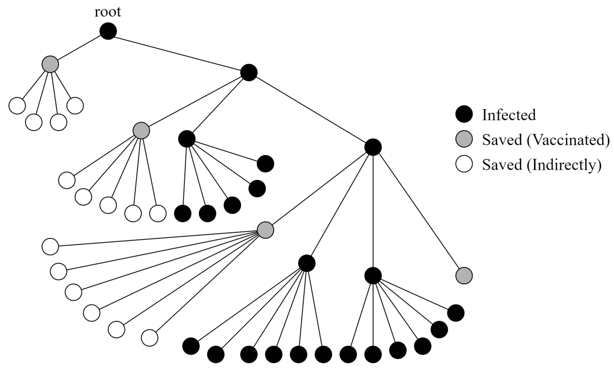


Figure 3: The solution given by ‘degree-heavy’ greedy algorithm on T_3

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.

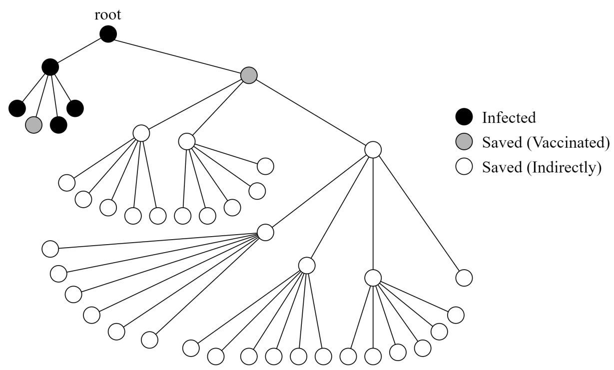


Figure 4: The solution given by “plan B” on T_3

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_{optimal} as the set of vertices saved by the optimal solution, and V_{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:

$$\begin{cases} V^A_{\text{optimal}} = V_{\text{optimal}} \cap V_{\text{greedy}} \\ V^B_{\text{optimal}} = V_{\text{optimal}} \setminus V_{\text{greedy}} \end{cases} \tag{4}$$

Where V^A_{optimal} is the vertices who are saved by both strategies and V^B_{optimal} are the vertices who are saved only by optimal solution. This implies:

1. $V^A_{\text{optimal}} \cap V^B_{\text{optimal}} = \emptyset$
2. $V^A_{\text{optimal}} \cup V^B_{\text{optimal}} = V_{\text{optimal}}$
3. $|V_{\text{optimal}}| = \left|V^A_{\text{optimal}}\right| + \left|V^B_{\text{optimal}}\right|$

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

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

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

To show $S_{\text{greedy}} \geq S^B_{\text{optimal}}$, 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^{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 defination, V^B_{optimal} consists of the vertices that are not saved by subtree-heavy algorithm but saved by the optimal solution. So the saved subtrees in V^B_{optimal} 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^B_{optimal} is smaller than one of the subtrees in V_{greedy}, and these larger subtrees in V_{greedy} have no intersectons between each other. The cardinality of V^B_{optimal} is smaller the sum of the sizes of saved subtrees, so we conclude that $\left|V^B_{\text{optimal}}\right| \leq |V_{\text{greedy}}|$, which is equivalent to $S^B_{\text{optimal}} \leq S_{\text{greedy}}$

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^th element represent the selected vertex in the i^{th} round.

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

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:

$$V_{\mathrm{G}} = \bigcup_{1 \le i \le \mathrm{MAX}^{\mathrm{G}}} T_{v_i^{\mathrm{G}}}$$

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

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

It is easy to show $\max I_B \le \mathrm{MAX}^{\mathrm{G}}$:

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

Then, with the help of I_B, V^B_{optimal} can be written as:

$$\begin{aligned} V^B_{\text{optimal}} &= V_{\text{optimal}} \setminus V_{\mathrm{G}} \\ &= \left(\bigcup_{1 \le i \le \mathrm{MAX}^{\mathrm{Opt}}} T_{v_i^{\mathrm{Opt}}} \right) \setminus V_{\mathrm{G}} \\ &= \bigcup_{1 \le i \le \mathrm{MAX}^{\mathrm{Opt}}} \left(T_{v_i^{\mathrm{Opt}}} \setminus V_{\mathrm{G}} \right) \\ &= \bigcup_{i \in I_B} \left(T_{v_i^{\mathrm{Opt}}} \setminus V_{\mathrm{G}} \right) \end{aligned}$$

And if we apply one step of scaling, we obtain the upperbound of S^B_{optimal}:

$$V^B_{\text{optimal}} \subseteq \bigcup_{i \in I_B} T_{v_i^{\mathrm{Opt}}} \tag{5}$$

we name the superset on the right hand side V^B_{super}. Therefore, $\left|V^B_{\text{super}}\right| \ge \left|V^B_{\text{optimal}}\right| = S^B_{\text{optimal}}$

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

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

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

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

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

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

Hence, we have shown that:

$$|V_{\text{G}}| \geq \left|V_{\text{super}}^B\right| \geq \left|V_{\text{optimal}}^B\right|$$

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_Vertex: selects a vertex to vaccinate

$Q \leftarrow Queue(V[r].children)$;

for $i \leftarrow 1$ **to** Max_Depth **do** simulate the pandemic of i^{th} round

  $chosen\_idx \leftarrow$ call Select_Vertex on Q;

  while $\neg Q.empty() \wedge Q.front().depth = i$ **do**

    $x \leftarrow Q.pop()$;

    if $x \neq chosen\_idx$ **then**

      $V[x].set\_infected()$;

      foreach $y \in V[x].children$ **do**

        $Q.add(y)$;

      end

    end

  end

end

Assume the procedure of Select_Vertex 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_Vertex 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 **Appendex 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.

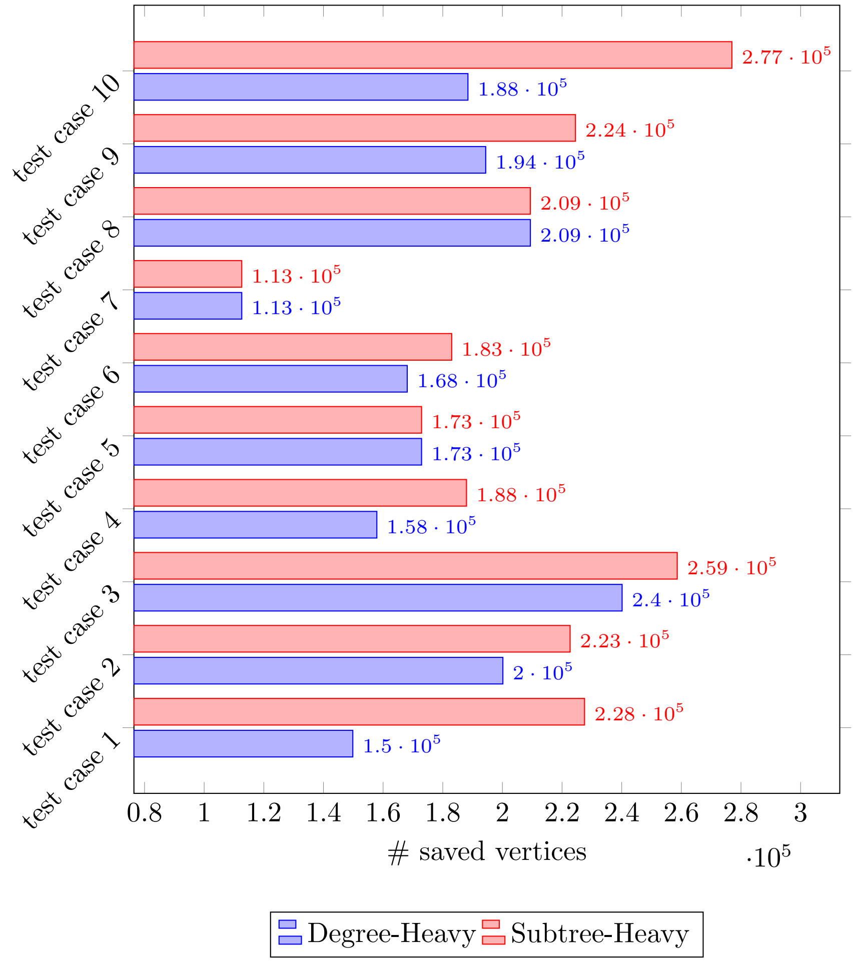
To run the program using this strategy, add `-w` flag and use `-t` to specify the path of test case:

```
./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 unvaccined 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.

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

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 π^R is enough)

$G =$

		Blue	
		SS	SH
Red	SS	-0.1511	292.9151
	SH	-293.3994	0.1158

(7)

The representation of payoff matrix sometimes does not gives 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 $0 \le b_1, b_2 \le 1$ 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 posiible 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:

$$\min\{\pi^R_{1,1} r_1 + \pi^R_{2,1} r_2, \pi^R_{1,2} r_1 + \pi^R_{2,2} r_2\}$$

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 : $\max \min\{\pi^R_{1,1} r_1 + \pi^R_{2,1} r_2, \pi^R_{1,2} r_1 + \pi^R_{2,2} r_2\}$. And the problem can be formulated as an LP:

$$\begin{array}{ll} \text{maximize} & p_1 \\ \text{subject to} & \pi^R_{1,1} r_1 + \pi^R_{2,1} r_2 - p_1 \geq 0 \\ & \pi^R_{1,2} r_1 + \pi^R_{2,2} r_2 - p_1 \geq 0 \\ & r_1 + r_2 = 1 \\ & r_1, r_2 \geq 0 \end{array} \tag{8}$$

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:

$$\begin{array}{ll} \text{minimize} & p_2 \\ \text{subject to} & \pi^R_{1,1} b_1 + \pi^R_{1,2} b_2 - p_2 \leq 0 \\ & \pi^R_{2,1} b_1 + \pi^R_{2,2} b_2 - p_2 \leq 0 \\ & b_1 + b_2 = 1 \\ & b_1, b_2 \geq 0 \end{array} \tag{9}$$

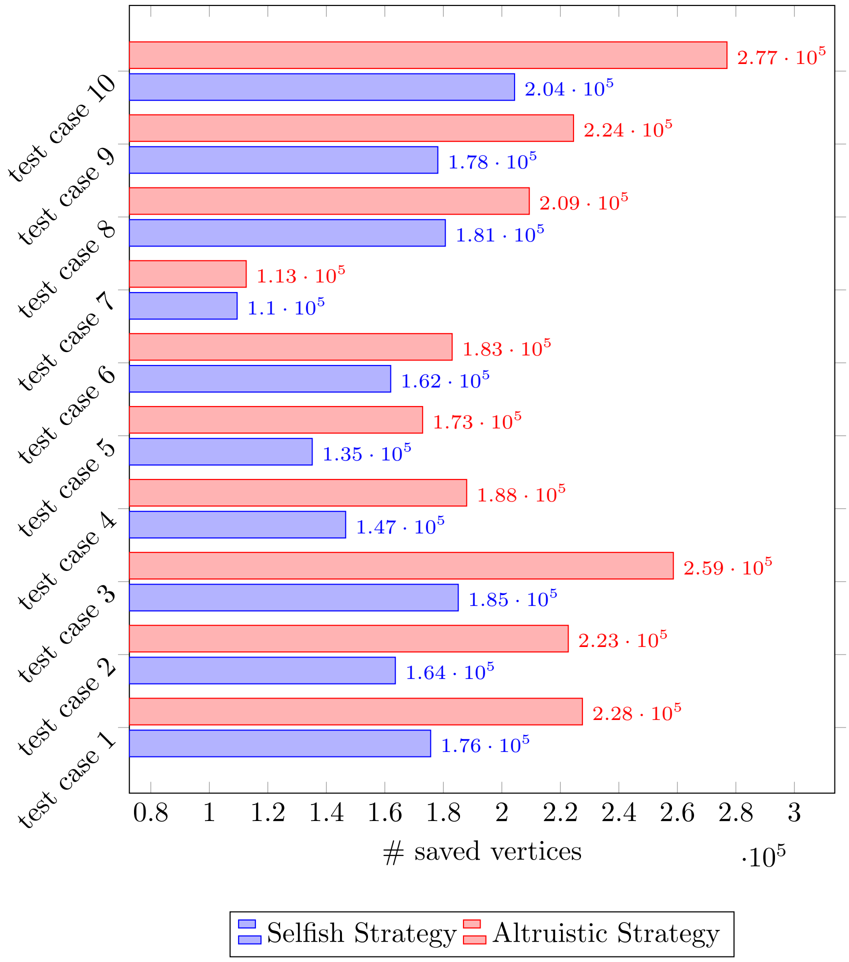
Solving this LP yields the output of optimal solution of $\tilde{\mathbf{b}} = (b_1, b_2) = (1, 0)$ and the corresponding optimal value is $\min\{p_2\} = -0.1511$. This result highly fits our prediction, which means the blue government should no doubt but use the strategy of SS to win the game (or win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

$W \leftarrow$ SegmentTree-Query-Weight(pre$_v$);

$C_{sum} \leftarrow$ SegmentTree-Query-Color-Sum(pre$_v$);

SegmentTree-Set-Saved(pre$_v$, post$_v$);

$u \leftarrow v$;

while $u \neq NIL$ **do**

  SegmentTree-Add-Weight(pre$_{chain_top[u]}$, pre$_u$, $-W$);

  SegmentTree-Add-Color(pre$_{chain_top[u]}$, pre$_u$, $-C_{sum}$);

  $u = \pi[top\_chain[u]]$;

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.

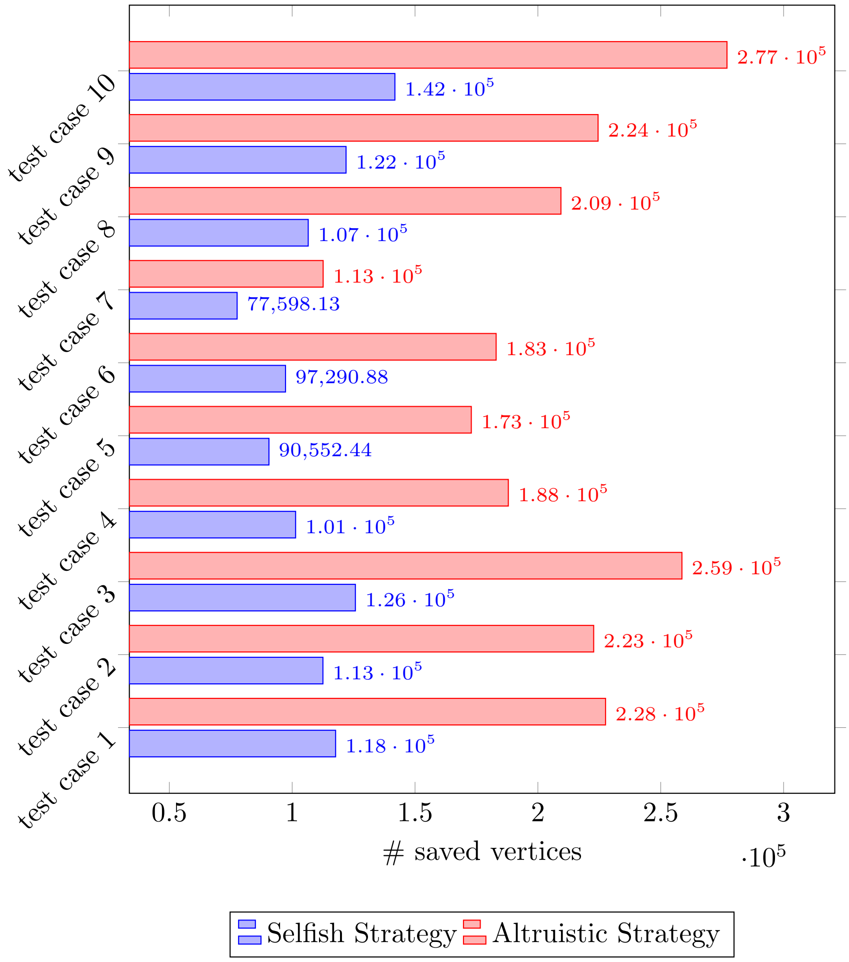
Results

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

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

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 applys selfish strategy earns more, and the one who is altruistic suffers more. This fact further illustrate 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 controvertion 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

```
#include <queue>
#include <vector>
#include <fstream>
#include <sstream>
#include <iostream>
#include <algorithm>

class Vertex
{
public:
    int depth = 0; // the depth on the tree
    int degree_cnt = 0; // number of degree of the node
    int weight = 0; // number of children nodes in the subtree
```

```
    int infected = 0; // 1: infected, 0: not infected yet

    std::vector<int> children; // children nodes
};

std::vector<Vertex> V;

// helper function
int degree_greedy(std::queue<int> &boundary)
{
    int head = boundary.front(), ans = head;

    do
    {
        int x = boundary.front();
        boundary.pop();
        boundary.push(x);

        if(V[x].degree_cnt > V[ans].degree_cnt)
        {
            ans = x;
        }

    } while (head != boundary.front());

    return ans;
}

// helper function
int weight_greedy(std::queue<int> &boundary)
{
    int head = boundary.front(), ans = head;

    do
    {
        int x = boundary.front();
        boundary.pop();
        boundary.push(x);

        if(V[x].weight > V[ans].weight)
        {
            ans = x;
        }

    } while (head != boundary.front());

    return ans;
}

// build tree
void build_tree(int now_idx)
{
    V[now_idx].weight = 1;
    V[now_idx].degree_cnt = 0;

    for(int child : V[now_idx].children)
    {
        V[child].depth = V[now_idx].depth + 1;

        build_tree(child);

        V[now_idx].degree_cnt++;
        V[now_idx].weight += V[child].weight;
    }
}

// simulation the vaccine status
void simulation_on_tree(int start_idx, int (*select_fn) (std::queue<int> &))
{
    std::queue<int> boundary;
    V[start_idx].infected = 1;
```

```cpp

    for(int child : V[start_idx].children)
    {
        boundary.push(child);
    }

    for(int depth = 1; !boundary.empty(); depth++)
    {
        int choice_idx = select_fn(boundary);
        std::cout << "[round " << depth << "]: vaccinate " << choice_idx << std::endl;
        while(!boundary.empty())
        {
            int x = boundary.front();
            if(V[x].depth != depth)
            {
                break;
            }

            boundary.pop();

            if(x == choice_idx)
            {
                continue;
            }

            V[x].infected = 1;

            for(int child : V[x].children)
            {
                boundary.push(child);
            }
        }
    }
}

char * getCmdOption(char **first, char **last, const std::string & opt)
{
    char ** iter = std::find(first, last, opt);

    if(iter != last && iter + 1 != last)
    {
        return *(iter + 1);
    }

    return nullptr;
}

bool cmdOptionExists(char **first, char **last, const std::string & opt)
{
    return std::find(first, last, opt) != last;
}

int main(int argc, char **argv)
{
    // parse commond lines
    char * test_file = getCmdOption(argv, argv + argc, "-t");
    int use_weight_greedy = cmdOptionExists(argv, argv + argc, "-w");
    int use_degree_greedy = cmdOptionExists(argv, argv + argc, "-d");

    if(use_degree_greedy && use_weight_greedy)
    {
        std::cerr << "Please specify with one strategy!" << std::endl;
        return 1;
    }

    use_weight_greedy = 1 - use_degree_greedy;

    if(test_file == nullptr)
    {
        std::cerr << "Please specify the path of the test file!" << std::endl;
        return 1;
    }
```

```cpp

    // open the test file
    std::ifstream file_in(test_file);
    std::string buffer;
    std::istringstream line;

    if(file_in.fail())
    {
        std::cerr << "Error in opening the test case file!" << std::endl;
        return 1;
    }

    std::cerr << "Reading test file:";
    int num_vertices;

    std::getline(file_in, buffer);
    line.str(buffer);
    line >> num_vertices;

    V.resize(num_vertices + 1);
    V[0].infected = 1;

    for(int i = 1; i <= num_vertices && std::getline(file_in, buffer); i++)
    {
        int idx, x;
        line.clear();
        line.str(buffer);
        line >> idx;

        while(line >> x)
        {
            V[idx].children.push_back(x);
        }
    }

    file_in.close();

    std::cerr << " done." << std::endl;

    std::cerr << "Simulating using "
        << (use_degree_greedy ? "degree-heavy" : "subtree-heavy")
        << " strategy" << std::endl;

    build_tree(1);

    simulation_on_tree(1, use_degree_greedy ? degree_greedy : weight_greedy);

    int saved_cnt = 0;

    for(const Vertex &v : V)
    {
        if(v.infected == 0)
        {
            saved_cnt++;
        }
    }

    std::cout << saved_cnt << " people are saved in total." << std::endl;

    return 0;
}
```

Appendix C.

Listing 2: Code for Part 2

```
#include <queue>
#include <vector>
#include <chrono>
#include <random>
#include <cstring>
#include <fstream>
#include <sstream>
#include <iostream>
#include <algorithm>

#define RED -1
#define BLUE 1


class Vertex
{
public:
    int color = 0; // -1 or 1, red or blue
    int color_diff = 0;
    int depth = 0; // the depth on the tree
    int degree_cnt = 0; // number of degree of the node
    int weight = 0; // number of children nodes in the subtree

    int infected = 0; // 1: infected, 0: not infected yet

    std::vector<int> children; // children nodes
};

std::vector<Vertex> V;

class Player
{
public:
    int color;
    int save_cnt;
    virtual int select_vertex(std::queue<int> &boundary) = 0;

    Player() { }
    Player(int col) : color(col), save_cnt(0) { }
};

class SelfishPlayer
    : public Player
{
public:

    SelfishPlayer() { }
    SelfishPlayer(int col) : Player(col) { }

    int select_vertex(std::queue<int> &boundary) override
    {
        int head = boundary.front(), ans = head;

        do
        {
            int x = boundary.front();
            boundary.pop();
            boundary.push(x);

            if(V[x].color_diff * color > V[ans].color_diff * color)
            {
                ans = x;
            }

        } while (head != boundary.front());

        return ans;
```

```
    }
};

class SubtreeHeavyPlayer
    : public Player
{
public:

    SubtreeHeavyPlayer() { }
    SubtreeHeavyPlayer(int col): Player(col) { }

    int select_vertex(std::queue<int> &boundary) override
    {
        int head = boundary.front(), ans = head;

        do
        {
            int x = boundary.front();
            boundary.pop();
            boundary.push(x);

            if(V[x].weight > V[ans].weight)
            {
                ans = x;
            }

        } while (head != boundary.front());

        return ans;
    }
};


Player * player[2] = { nullptr, nullptr }; // 0 red, 1 blue


// build tree
void build_tree(int now_idx)
{
    V[now_idx].weight = 1;
    V[now_idx].degree_cnt = 0;
    V[now_idx].color_diff = V[now_idx].color;

    for(int child : V[now_idx].children)
    {
        V[child].depth = V[now_idx].depth + 1;

        build_tree(child);

        V[now_idx].degree_cnt++;
        V[now_idx].weight += V[child].weight;
        V[now_idx].color_diff += V[child].color_diff;
    }
}

// simulation the vaccine status
void simulation_on_tree(int start_idx)
{
    std::queue<int> boundary;
    V[start_idx].infected = 1;

    for(int child : V[start_idx].children)
    {
        boundary.push(child);
    }

    for(int depth = 1, now = std::rand() % 2; !boundary.empty(); depth++, now = 1 - now)
    {
        int choice_idx = player[now]->select_vertex(boundary);
        while(!boundary.empty())
        {
            int x = boundary.front();
```

```
            if(V[x].depth != depth)
            {
                break;
            }

            boundary.pop();

            if(x == choice_idx)
            {
                continue;
            }

            V[x].infected = 1;

            for(int child : V[x].children)
            {
                boundary.push(child);
            }
        }
    }
}


void init_vertices(int num_vertices)
{
    unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
    std::srand(seed);

    int * colors = new int[num_vertices + 1];
    int off = std::rand() % 2;

    for(int i = 1; i <= num_vertices; i++)
    {
        colors[i] = (i + off) % 2 ? RED : BLUE;
    }

    std::shuffle(colors + 1, colors + num_vertices + 1, std::default_random_engine(seed))
    ;

    for(int i = 1; i <= num_vertices; i++)
    {
        V[i].color = colors[i];
    }

    delete[] colors;
}

int game_strategy(char * arg)
{
    if(arg == nullptr)
    {
        return -1;
    }

    int ret_val = -1;

    if(std::strcmp(arg, "SSSS") == 0)
    {
        ret_val = 0;
    }
    else if(std::strcmp(arg, "SSSH") == 0)
    {
        ret_val = 1;
    }
    else if(std::strcmp(arg, "SHSS") == 0)
    {
        ret_val = 2;
    }
    else if(std::strcmp(arg, "SHSH") == 0)
    {
        ret_val = 3;
    }
```

```
    if(ret_val == -1)
    {
        return -1;
    }

    if(ret_val & 2)
    {
        player[0] = new SubtreeHeavyPlayer(RED);
    }
    else
    {
        player[0] = new SelfishPlayer(RED);
    }

    if(ret_val & 1)
    {
        player[1] = new SubtreeHeavyPlayer(BLUE);
    }
    else
    {
        player[1] = new SelfishPlayer(BLUE);
    }

    return ret_val;
}

char * getCmdOption(char **first, char **last, const std::string & opt)
{
    char ** iter = std::find(first, last, opt);

    if(iter != last && iter + 1 != last)
    {
        return *(iter + 1);
    }

    return nullptr;
}

bool cmdOptionExists(char **first, char **last, const std::string & opt)
{
    return std::find(first, last, opt) != last;
}

int main(int argc, char **argv)
{
    std::ios::sync_with_stdio(false);

    // parse commond lines
    char * test_file = getCmdOption(argv, argv + argc, "-t");
    char * strategy = getCmdOption(argv, argv + argc, "-s");
    int stat = game_strategy(strategy);

    if(stat == -1)
    {
        std::cerr << "Please specify a valid strategy!" << std::endl;
        return 1;
    }

    if(test_file == nullptr)
    {
        std::cerr << "Please specify the path of the test file!" << std::endl;
        return 1;
    }

    // open the test file
    std::ifstream file_in(test_file);
    std::string buffer;
    std::istringstream line;

    if(file_in.fail())
    {
```

```
        std::cerr << "Error in opening the test case file!" << std::endl;
        return 1;
    }

    std::cerr << "Reading test file:";
    int num_vertices;

    std::getline(file_in, buffer);
    line.str(buffer);
    line >> num_vertices;

    V.resize(num_vertices + 1);
    V[0].infected = 1;

    for(int i = 1; i <= num_vertices && std::getline(file_in, buffer); i++)
    {
        int idx, x;
        line.clear();
        line.str(buffer);
        line >> idx;

        while(line >> x)
        {
            V[idx].children.push_back(x);
        }
    }

    file_in.close();

    std::cerr << " done." << std::endl;

    std::cerr << "Red Player: " << ((stat & 2) ? "SH" : "SS") << std::endl;
    std::cerr << "Blue Player: " << ((stat & 1) ? "SH" : "SS") << std::endl;

    init_vertices(num_vertices);

    build_tree(1);

    simulation_on_tree(1);

    int saved_cnt = 0;

    for(const Vertex &v : V)
    {
        if(v.infected == 0)
        {
            saved_cnt++;
            if(v.color == RED)
            {
                player[0]->save_cnt++;
            }
            else if(v.color == BLUE)
            {
                player[1]->save_cnt++;
            }
        }
    }

    std::cout << saved_cnt << " people are saved in total." << std::endl;
    std::cout << "Saved Reds: " << player[0]->save_cnt << std::endl;
    std::cout << "Saved Blues: " << player[1]->save_cnt << std::endl;
    std::cout << ((player[0]->save_cnt > player[1]->save_cnt) ? "Red" : "Blue") << " wins
    " << std::endl;

    delete player[0];
    delete player[1];

    return 0;
}
```

Appendix D.

Listing 3: Code for the game without the first assumption

```
#include <queue>
#include <vector>
#include <chrono>
#include <random>
#include <cstring>
#include <fstream>
#include <sstream>
#include <iostream>
#include <algorithm>

#define RED -1
#define BLUE 1

class SegtreeNode
{
public:
    // data stored
    int max_color_diff;
    int min_color_diff;
    int max_weight;

    int max_color_index;
    int min_color_index;
    int max_weight_index;

    // auxiliary
    int saved;
    int save_tag;
    int color_tag;
    int weight_tag;

    void to_default()
    {
        max_color_diff = -0x3f3f3f3f;
        min_color_diff = 0x3f3f3f3f;
        max_weight = 0;
        saved = 0;
        color_tag = 0;
        weight_tag = 0;
    }

    void set_saved()
    {
        to_default();
        saved = 1;
        save_tag = 1;
    }

    void apply_color_tag(int tag)
    {
        color_tag += tag;
        max_color_diff += tag;
        min_color_diff += tag;
    }

    void apply_weight_tag(int tag)
    {
        weight_tag += tag;
        max_weight += tag;
    }
};

class Segtree
{
public:
    int length;
    std::vector<SegtreeNode> tree;
```

```

    Segtree(): length(0) { }
    Segtree(int seg_length)
        : length(seg_length)
    {
        tree.resize(length * 4 + 1);
        init(1, length, 1);
    }

    void init(int l, int r, int now)
    {
        tree[now].to_default();
        if(l == r)
        {
            return ;
        }

        int mid = (l + r) >> 1;
        init(l, mid, now << 1);
        init(mid + 1, r, now << 1 | 1);
    }

    void insert(int idx, int v_index, int color_sum, int weight)
    {
        insert(1, this->length, idx, 1, v_index, color_sum, weight);
    }

    void save_range(int first_index, int last_index)
    {
        save_range(1, length, first_index, last_index, 1);
    }

    void update(int first, int last, int col_diff, int weight_diff)
    {
        update_range(1, length, first, last, 1, col_diff, weight_diff);
    }

    int max_weight_index(int first, int last, int &ans, int &max_val)
    {
        ans = -1, max_val = -0x3f3f3f3f;

        query_max_weight(1, length, first, last, 1, max_val, ans);

        return ans;
    }

    int max_color_diff(int first, int last, int &ans, int &max_val)
    {
        ans = -1, max_val = -0x3f3f3f3f;

        query_max_color_diff(1, length, first, last, 1, max_val, ans);

        return ans;
    }

    int min_color_diff(int first, int last, int &ans, int &min_val)
    {
        ans = -1, min_val = 0x3f3f3f3f;

        query_min_color_diff(1, length, first, last, 1, min_val, ans);

        return ans;
    }

    int is_saved(int index)
    {
        return query_saved(1, length, index, 1);
    }

    const SegtreeNode * snapshot(int index)
    {
        return query_everything(1, length, index, 1);
```

```
    }

private:
    void pushup(int now)
    {
        tree[now].max_color_diff = std::max(tree[now << 1].max_color_diff, tree[now << 1 | 1].max_color_diff);
        tree[now].max_color_index = (tree[now << 1].max_color_diff > tree[now << 1 | 1].max_color_diff) ?
            tree[now << 1].max_color_index : tree[now << 1 | 1].max_color_index;

        tree[now].min_color_diff = std::min(tree[now << 1].min_color_diff, tree[now << 1 | 1].min_color_diff);
        tree[now].min_color_index = (tree[now << 1].min_color_diff < tree[now << 1 | 1].min_color_diff) ?
            tree[now << 1].min_color_index : tree[now << 1 | 1].min_color_index;

        tree[now].max_weight = std::max(tree[now << 1].max_weight, tree[now << 1 | 1].max_weight);
        tree[now].max_weight_index = (tree[now << 1].max_weight > tree[now << 1 | 1].max_weight) ?
            tree[now << 1].max_weight_index : tree[now << 1 | 1].max_weight_index;
    }

    void pushdown(int now)
    {
        if(tree[now].save_tag)
        {
            tree[now << 1].set_saved();
            tree[now << 1 | 1].set_saved();
            tree[now].save_tag = 0;
        }

        if(tree[now].color_tag)
        {
            tree[now << 1].apply_color_tag(tree[now].color_tag);
            tree[now << 1 | 1].apply_color_tag(tree[now].color_tag);
            tree[now].color_tag = 0;
        }

        if(tree[now].weight_tag)
        {
            tree[now << 1].apply_weight_tag(tree[now].weight_tag);
            tree[now << 1 | 1].apply_weight_tag(tree[now].weight_tag);
            tree[now].weight_tag = 0;
        }
    }

    void insert(int l, int r, int i, int now, int v_index, int color, int weight)
    {
        if(l == r)
        {
            tree[now].max_color_diff = tree[now].min_color_diff = color;
            tree[now].max_weight = weight;
            tree[now].max_color_index = tree[now].min_color_index = tree[now].max_weight_index = v_index;
            return ;
        }

        pushdown(now);

        int mid = (l + r) >> 1;

        if(i <= mid)
        {
            insert(l, mid, i, now << 1, v_index, color, weight);
        }
        else
        {
            insert(mid + 1, r, i,now << 1 | 1, v_index, color, weight);
        }

```

```
        pushup(now);
    }

    void save_range(int l, int r, int saved_l, int saved_r, int now)
    {
        if(saved_l > r || l > saved_r)
        {
            return ;
        }

        if(saved_l <= l && r <= saved_r)
        {
            tree[now].set_saved();
            return ;
        }

        pushdown(now);

        int mid = (l + r) >> 1;

        save_range(l, mid, saved_l, saved_r, now << 1);
        save_range(mid + 1, r, saved_l, saved_r, now << 1 | 1);

        pushup(now);
    }

    void update_range(int l, int r, int upd_l, int upd_r, int now, int upd_color, int
    upd_weight)
    {
        if(upd_l > r || l > upd_r)
        {
            return ;
        }

        if(upd_l <= l && r <= upd_r)
        {
            tree[now].apply_color_tag(upd_color);
            tree[now].apply_weight_tag(upd_weight);
            return ;
        }

        pushdown(now);

        int mid = (l + r) >> 1;

        update_range(l, mid, upd_l, upd_r, now << 1, upd_color, upd_weight);
        update_range(mid + 1, r, upd_l, upd_r, now << 1 | 1, upd_color, upd_weight);

        pushup(now);
    }

    void query_max_weight(int l, int r, int query_l, int query_r, int now, int &
    max_weight, int & index)
    {
        if(query_l > r || l > query_r)
        {
            return ;
        }

        if(query_l <= l && r <= query_r)
        {
            if(max_weight < tree[now].max_weight)
            {
                max_weight = tree[now].max_weight;
                index = tree[now].max_weight_index;
            }

            return ;
        }

        pushdown(now);

```

```
        int mid = (l + r) >> 1;

        query_max_weight(l, mid, query_l, query_r, now << 1, max_weight, index);
        query_max_weight(mid + 1, r, query_l, query_r, now << 1 | 1, max_weight, index);
    }

    void query_max_color_diff(int l, int r, int query_l, int query_r, int now, int &
    max_color_diff, int & index)
    {
        if(query_l > r || l > query_r)
        {
            return ;
        }

        if(query_l <= l && r <= query_r)
        {
            if(max_color_diff < tree[now].max_color_diff)
            {
                max_color_diff = tree[now].max_color_diff;
                index = tree[now].max_color_index;
            }

            return ;
        }

        pushdown(now);

        int mid = (l + r) >> 1;

        query_max_color_diff(l, mid, query_l, query_r, now << 1, max_color_diff, index);
        query_max_color_diff(mid + 1, r, query_l, query_r, now << 1 | 1, max_color_diff,
    index);
    }

    void query_min_color_diff(int l, int r, int query_l, int query_r, int now, int &
    min_color_diff, int & index)
    {
        if(query_l > r || l > query_r)
        {
            return ;
        }

        if(query_l <= l && r <= query_r)
        {
            if(min_color_diff > tree[now].min_color_diff)
            {
                min_color_diff = tree[now].min_color_diff;
                index = tree[now].min_color_index;
            }

            return ;
        }

        pushdown(now);

        int mid = (l + r) >> 1;

        query_min_color_diff(l, mid, query_l, query_r, now << 1, min_color_diff, index);
        query_min_color_diff(mid + 1, r, query_l, query_r, now << 1 | 1, min_color_diff,
    index);
    }

    int query_saved(int l, int r, int i, int now)
    {
        if(l == r)
        {
            return tree[now].saved;
        }

        pushdown(now);

        int mid = (l + r) >> 1;
```

```
        if(i <= mid)
        {
            return query_saved(l, mid, i, now << 1);
        }

        return query_saved(mid + 1, r, i, now << 1 | 1);
    }

    SegtreeNode * query_everything(int l, int r, int i, int now)
    {
        if(l == r)
        {
            return &tree[now];
        }

        pushdown(now);

        int mid = (l + r) >> 1;

        if(i <= mid)
        {
            return query_everything(l, mid, i, now << 1);
        }

        return query_everything(mid + 1, r, i, now << 1 | 1);
    }
};

Segtree * segtree = nullptr;

class Vertex
{
public:
    int color = 0; // -1 or 1, red or blue
    int color_diff = 0;
    int depth = 0; // the depth on the tree
    int degree_cnt = 0; // number of degree of the node
    int weight = 0; // number of children nodes in the subtree

    int parent = 0;

    int infected = 0; // 1: infected, 0: not infected yet

    int max_child = 0;
    int chain_top;
    int pre, post;

    std::vector<int> children; // children nodes
};

std::vector<Vertex> V;

class Player
{
public:
    int color;
    int save_cnt;
    virtual int select_vertex(std::queue<int> &boundary) = 0;

    Player() { }
    Player(int col) : color(col), save_cnt(0) { }

    void update_situation(int x)
    {
        const SegtreeNode * it = segtree->snapshot(V[x].pre);
        int col_diff = it->max_color_diff;
        int weight = it->max_weight;

        segtree->save_range(V[x].pre, V[x].post);

        int now = x;
```

```
        while(now != 0)
        {
            segtree->update(V[V[now].chain_top].pre, V[now].pre, -col_diff, -weight);
            now = V[V[now].chain_top].parent;
        }
    }
};

class SelfishPlayer
    : public Player
{
public:

    SelfishPlayer() { }
    SelfishPlayer(int col) : Player(col) { }

    int select_vertex(std::queue<int> &boundary) override
    {
        int head = boundary.front(), ans = head, weight_ans = (-color) * 0x3f3f3f3f;

        do
        {
            int x = boundary.front();
            boundary.pop();
            boundary.push(x);

            int ans_x, weight_x;

            if(color == RED)
            {
                segtree->min_color_diff(V[x].pre, V[x].post, ans_x, weight_x);
                if(weight_x < weight_ans)
                {
                    weight_ans = weight_x;
                    ans = ans_x;
                }
            }
            else
            {
                segtree->max_color_diff(V[x].pre, V[x].post, ans_x, weight_x);
                if(weight_x > weight_ans)
                {
                    weight_ans = weight_x;
                    ans = ans_x;
                }
            }


        } while (head != boundary.front());

        update_situation(ans);

        return ans;
    }
};

class SubtreeHeavyPlayer
    : public Player
{
public:

    SubtreeHeavyPlayer() { }
    SubtreeHeavyPlayer(int col): Player(col) { }

    int select_vertex(std::queue<int> &boundary) override
    {
        int head = boundary.front(), ans = head, weight_ans = -0x3f3f3f3f;

        do
        {
            int x = boundary.front();
            boundary.pop();
```

```
            boundary.push(x);

            int ans_x, weight_x;

            segtree->max_weight_index(V[x].pre, V[x].post, ans_x, weight_x);

            if(weight_x > weight_ans)
            {
                weight_ans = weight_x;
                ans = ans_x;
            }


        } while (head != boundary.front());

        update_situation(ans);

        std::cout << ans << " saves " << weight_ans << std::endl;;
        return ans;
    }
};


Player * player[2] = { nullptr, nullptr }; // 0 red, 1 blue

int time_stamp = 0;

// build tree
void build_tree(int now_idx)
{
    V[now_idx].weight = 1;
    V[now_idx].degree_cnt = 0;
    V[now_idx].color_diff = V[now_idx].color;

    int max_weight = -1;

    for(int child : V[now_idx].children)
    {
        V[child].parent = now_idx;
        V[child].depth = V[now_idx].depth + 1;

        build_tree(child);

        if(V[child].weight > max_weight)
        {
            max_weight = V[child].weight;
            V[now_idx].max_child = child;
        }

        V[now_idx].degree_cnt++;
        V[now_idx].weight += V[child].weight;
        V[now_idx].color_diff += V[child].color_diff;
    }
}

void build_segtree(int now_idx, int top_idx)
{
    if(now_idx == 0)
    {
        return ;
    }

    V[now_idx].pre = ++time_stamp;
    V[now_idx].chain_top = top_idx;

    segtree->insert(V[now_idx].pre, now_idx, V[now_idx].color_diff, V[now_idx].weight);

    build_segtree(V[now_idx].max_child, top_idx);

    for(int child : V[now_idx].children)
    {
        if(child == V[now_idx].max_child)
```

```
        {
            continue;
        }

        build_segtree(child, child);
    }


    V[now_idx].post = time_stamp;
}

// simulation the vaccine status
void simulation_on_tree(int start_idx)
{
    std::queue<int> boundary;
    V[start_idx].infected = 1;

    for(int child : V[start_idx].children)
    {
        boundary.push(child);
    }

    for(int depth = 1, now = std::rand() % 2; !boundary.empty(); depth++, now = 1 - now)
    {
        std::cout << "[round " << depth << "] " << (now ? "RED" : "BLUE") << " saves " ;
        int choice_idx = player[now]->select_vertex(boundary);

        while(!boundary.empty())
        {
            int x = boundary.front();
            if(V[x].depth != depth)
            {
                break;
            }

            boundary.pop();

            if(segtree->is_saved(V[x].pre))
            {
                continue;
            }

            V[x].infected = 1;

            for(int child : V[x].children)
            {
                if(! (segtree->is_saved(V[child].pre)) )
                {
                    boundary.push(child);
                }
            }
        }
    }
}


void init_vertices(int num_vertices)
{
    unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
    std::srand(seed);

    int * colors = new int[num_vertices + 1];
    int off = std::rand() % 2;

    for(int i = 1; i <= num_vertices; i++)
    {
        colors[i] = (i + off) % 2 ? RED : BLUE;
    }

    std::shuffle(colors + 1, colors + num_vertices + 1, std::default_random_engine(seed))
    ;

```

```
    for(int i = 1; i <= num_vertices; i++)
    {
        V[i].color = colors[i];
    }

    delete[] colors;
}

int game_strategy(char * arg)
{
    if(arg == nullptr)
    {
        return -1;
    }

    int ret_val = -1;

    if(std::strcmp(arg, "SSSS") == 0)
    {
        ret_val = 0;
    }
    else if(std::strcmp(arg, "SSSH") == 0)
    {
        ret_val = 1;
    }
    else if(std::strcmp(arg, "SHSS") == 0)
    {
        ret_val = 2;
    }
    else if(std::strcmp(arg, "SHSH") == 0)
    {
        ret_val = 3;
    }

    if(ret_val == -1)
    {
        return -1;
    }

    if(ret_val & 2)
    {
        player[0] = new SubtreeHeavyPlayer(RED);
    }
    else
    {
        player[0] = new SelfishPlayer(RED);
    }

    if(ret_val & 1)
    {
        player[1] = new SubtreeHeavyPlayer(BLUE);
    }
    else
    {
        player[1] = new SelfishPlayer(BLUE);
    }

    return ret_val;
}

char * getCmdOption(char **first, char **last, const std::string & opt)
{
    char ** iter = std::find(first, last, opt);

    if(iter != last && iter + 1 != last)
    {
        return *(iter + 1);
    }

    return nullptr;
}
```

```
bool cmdOptionExists(char **first, char **last, const std::string & opt)
{
    return std::find(first, last, opt) != last;
}

int main(int argc, char **argv)
{
    std::ios::sync_with_stdio(false);

    // parse commond lines
    char * test_file = getCmdOption(argv, argv + argc, "-t");
    char * strategy = getCmdOption(argv, argv + argc, "-s");
    int stat = game_strategy(strategy);

    if(stat == -1)
    {
        std::cerr << "Please specify a valid strategy!" << std::endl;
        return 1;
    }

    if(test_file == nullptr)
    {
        std::cerr << "Please specify the path of the test file!" << std::endl;
        return 1;
    }

    // open the test file
    std::ifstream file_in(test_file);
    std::string buffer;
    std::istringstream line;

    if(file_in.fail())
    {
        std::cerr << "Error in opening the test case file!" << std::endl;
        return 1;
    }

    std::cerr << "Reading test file:";
    int num_vertices;

    std::getline(file_in, buffer);
    line.str(buffer);
    line >> num_vertices;

    V.resize(num_vertices + 1);
    V[0].infected = 1;


    for(int i = 1; i <= num_vertices && std::getline(file_in, buffer); i++)
    {
        int idx, x;
        line.clear();
        line.str(buffer);
        line >> idx;

        while(line >> x)
        {
            V[idx].children.push_back(x);
        }
    }

    file_in.close();

    std::cerr << " done." << std::endl;

    std::cerr << "Red Player: " << ((stat & 2) ? "SH" : "SS") << std::endl;
    std::cerr << "Blue Player: " << ((stat & 1) ? "SH" : "SS") << std::endl;

    init_vertices(num_vertices);
    segtree = new Segtree(num_vertices);

    build_tree(1);
```

```
    build_segtree(1, 1);


    simulation_on_tree(1);

    int saved_cnt = 0;

    for(const Vertex &v : V)
    {
        if(v.infected == 0)
        {
            saved_cnt++;
            if(v.color == RED)
            {
                player[0]->save_cnt++;
            }
            else if(v.color == BLUE)
            {
                player[1]->save_cnt++;
            }
        }
    }

    std::cout << saved_cnt << " people are saved in total." << std::endl;
    std::cout << "Saved Reds: " << player[0]->save_cnt << std::endl;
    std::cout << "Saved Blues: " << player[1]->save_cnt << std::endl;
    std::cout << ((player[0]->save_cnt > player[1]->save_cnt) ? "Red" : "Blue") << " wins
    " << std::endl;

    delete player[0];
    delete player[1];
    delete segtree;

    return 0;
}
```