Code Listing

Daniel Joffe

June 2021

Line breaks were added to stop code from going off the page.

Contents

1 Cargo.toml **1**

2 py/requirements.txt **2**

3 src/lib.rs **2**

4 src/metrics.rs **4**

5 src/hill_climb.rs **6**

6 src/bin/hill-solve.rs **12**

7 src/bin/lahc-solve.rs **13**

8 src/bin/data-gen.rs **13**

9 src/bin/mass-data-gen.rs **18**

10 src/bin/score.rs **20**

11 py/src/int-prog.py **24**

1 Cargo.toml

```
[package]
name = "dissertation"
version = "0.1.0"
authors = ["hypen-emdash <joffe.daniel@gmail.com>"]
edition = "2018"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/
```

```
    manifest.html

[dependencies]
rand = "0.8.3"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
structopt = "0.3.21"
anyhow = "1.0.38"
csv = "1.1.6"
ordered-float = "2.1.1"
```

2 py/requirements.txt

```
absl-py==0.11.0
appdirs==1.4.4
black==20.8b1
click==7.1.2
cycler==0.10.0
kiwisolver==1.3.1
matplotlib==3.4.2
mypy-extensions==0.4.3
numpy==1.20.3
ortools==8.2.8710
pandas==1.2.4
pathspec==0.8.1
Pillow==8.2.0
protobuf==3.15.5
pyparsing==2.4.7
python-dateutil==2.8.1
pytz==2021.1
regex==2020.11.13
scipy==1.6.3
six==1.15.0
toml==0.10.2
typed-ast==1.4.2
typing-extensions==3.7.4.3
```

3 src/lib.rs

```
mod hill_climb;

pub use hill_climb::{HillClimbingPlanner, LahcPlanner};

pub mod metrics;
```

```
use serde::{Deserialize, Serialize};

/// A complete, undirected graph that models the relationship between
/// all guests at a wedding.
/// Guests are indexed as ‘usize‘.
#[derive(Debug, Clone, Hash, PartialEq, Eq, Deserialize, Serialize)]
pub struct GuestRelations {
    // A square array, symmetrical (ie ‘relationships[i][j] == relationships[j][i]‘)
    // with zeros along the diagonal.
    relationships: Vec<Vec<i64>>,
}

impl GuestRelations {
    pub fn new(relationships: Vec<Vec<i64>>) -> Self {
        // TODO: check/force symmetry, squareness, and lack of self-loops.
        Self::new_unchecked(relationships)
    }

    pub fn new_unchecked(relationships: Vec<Vec<i64>>) -> Self {
        Self { relationships }
    }

    /// Returns the degree of friendship two guests have.
    /// Positive is good, negative is bad. 0 is either unmet or self.
    /// # Panics
    /// Panics if either guest is unknown (out of bounds).
    pub fn relationship(&self, guest1: usize, guest2: usize) -> i64 {
        self.relationships[guest1][guest2]
    }

    /// Returns the number of guests.
    pub fn len(&self) -> usize {
        self.relationships.len()
    }

    /// Returns an iterator over the relationships.
    /// Should be combined with ‘.enumerate()‘
    /// if you want the indicies of the relevant guests.
    pub fn iter(&self) -> impl Iterator<Item = impl Iterator<Item = i64> + '_> + '_ {
        self.relationships.iter().map(|row| row.iter().copied())
    }
}

/// A seating plan. The outer vector is a list of tables.
/// Each table has a list of people.
/// People are 0-indexed, using the same scheme as ‘GuestRelations‘.
/// We use ‘Vec‘ for simplicity, but order is not for tables
/// or people within tables.
/// Each table should be the same size, and each guest should
/// appear exactly once.
pub type Plan = Vec<Vec<usize>>;

/// A problem-instance, which solvers should read from stdin with json.
#[derive(Debug, Clone, Hash, PartialEq, Eq, Serialize, Deserialize)]
pub struct Problem {
    pub relations: GuestRelations,
    pub n_tables: usize,
}

/// The main trait of the project - implement this for your solver.
pub trait SeatingPlanner {
    fn plan(&mut self, problem: &Problem) -> Plan;
}

/// Once a solver is constructed, pass it to this function to
/// read the problem and write the solution.
pub fn run<T>(mut planner: T) -> anyhow::Result<()>
where
    T: SeatingPlanner,
{
    use std::io;

    let stdin = io::stdin();
    let reader = stdin.lock();

    let stdout = io::stdout();
    let writer = stdout.lock();

    let problem: Problem = serde_json::from_reader(reader)?;
    let plan = planner.plan(&problem);

    serde_json::to_writer(writer, &plan)?;

    Ok(())
}
```

4 src/metrics.rs

```
use crate::{GuestRelations, Plan};
```

```
pub struct Metrics {
    // Index by guest to get a vector of how they feel about
    // everone *else* at the table.
    neighbour_relationships: Vec<Vec<i64>>,
}

impl Metrics {
    pub fn new(plan: &Plan, relationships: &GuestRelations) -> Self {
        let mut inner = vec![Vec::new(); relationships.len()];

        for table in plan {
            for guest in table {
                for neighbour in table {
                    if guest == neighbour {
                        continue;
                    }
                    inner[*guest].push(relationships.relationship(*guest, *neighbour))
                }
            }
        }

        Self {
            neighbour_relationships: inner,
        }
    }

    pub fn n_lonely(&self) -> usize {
        self.neighbour_relationships
            .iter()
            .filter(|v| v.iter().all(|r| *r <= 0))
            .count()
    }

    pub fn happinesses(&self) -> impl Iterator<Item = i64> + '_ {
        self.neighbour_relationships.iter().map(|v| v.iter().sum())
    }

    pub fn total_happiness(&self) -> i64 {
        self.happinesses().sum()
    }

    pub fn mean_happiness(&self) -> f64 {
        self.total_happiness() as f64 / self.neighbour_relationships.len() as f64
    }

    pub fn median_happiness(&self) -> f64 {
        let all: Vec<i64> = self.happinesses().collect();

        let left_i = all.len() - 2;
        let right_i = all.len() - left_i;

        let left_med = all[left_i] as f64;
        let right_med = all[right_i] as f64;

        0.5 * (left_med + right_med)
    }

    pub fn max_happiness(&self) -> i64 {
        self.happinesses()
            .max()
            .expect("Expected a nonempty Metrics object.")
    }

    pub fn min_happiness(&self) -> i64 {
        self.happinesses()
            .min()
            .expect("Expected a nonempty Metrics object.")
    }
}
```

5 src/hill_climb.rs

```
use crate::metrics::Metrics;
use crate::{Plan, Problem, SeatingPlanner};

use std::{collections::VecDeque, num::NonZeroUsize};

use rand::prelude::*;

type Float = ordered_float::NotNan<f64>;

/// The EMA factor is the ratio for exponential-moving-average,
/// which decides how much heavily new information should be weighted
/// over our current estimates.
/// This is used to terminate the hill-climbing algorithms. We use a
/// fairly small algorithm so as not to terminate prematurely.
// SAFETY: the argument to ‘unchecked_new‘ must not be NaN.
// The value is constant, so we can see it is not NaN.
// We use the unsafe version because ‘Float::new‘ is not ‘const‘.
//
// Technically, ‘unchecked_new‘ is deprecated in favour of ‘new_unchecked‘
// but the only difference is the name (a good change in my opinion, since
// it is more consistent with ‘std‘) and the replacement is only in a newer
// version of the library. I would update this code to reflect that, but
// now that the code has run and I’ve used it to gather results, it’s more
// important to keep it as it is.
const EMA_FACTOR: Float = unsafe { Float::unchecked_new(0.01) };

/// Calculates the new exponential moving average, based on our
/// previous estimate of it, and one new value.
/// Formula taken from Operating Systems Concepts, page 209.
/// estimate = a*x + (1-a)*estimate
fn shift_ema(old_ema: Float, new_val: Float) -> Float {
    (EMA_FACTOR * new_val) + ((Float::new(1.0).unwrap() - EMA_FACTOR) * old_ema)
}

/// A swap of two people based on their locations.
#[derive(Debug, Copy, Clone, Hash, PartialEq, Eq)]
struct Swap {
    table1: usize,
    seat1: usize,
    table2: usize,
    seat2: usize,
}

#[derive(Debug, Clone, Hash, PartialEq, Eq)]
pub struct HillClimbingPlanner<R> {
    rng: R,

    /// How infrequent do updates need to get before we give up?
    termination_threshold: Float,
}

impl<R> HillClimbingPlanner<R>
where
    R: Rng,
{
    pub fn new(rng: R, termination_threshold: Float) -> Self {
        Self {
            rng,
            termination_threshold,
        }
    }
}

impl<R> SeatingPlanner for HillClimbingPlanner<R>
where
```

```
    R: Rng,
{
    fn plan(&mut self, problem: &Problem) -> Plan {
        let relationships = &problem.relations;
        let n_tables = problem.n_tables;
        let table_size = relationships.len() / n_tables;

        // Get an initial solution with no regard for quality.
        let mut plan = random_plan(&mut self.rng, relationships.len(), n_tables);

        // A moving average of how often we update our best solution.
        let mut update_ema = Float::new(1.0).unwrap();

        while update_ema >= self.termination_threshold {
            // Propose a small random change.
            let swap = get_random_swap(&mut self.rng, n_tables, table_size);

            // Measure current utility.
            let old_metrics = Metrics::new(&plan, relationships);

            // Make the change and measure new utility.
            // (Changing the solution in-place means that we don't have to
            // allocate a whole new solution with each iteration, but we
            // do have to keep a record of what the change was so we can
            // undo it if necessary.)
            let new_metrics = Metrics::new(&plan, relationships);

            // Check if we made things better or worse.
            let updated: Float;
            if new_metrics.total_happiness() > old_metrics.total_happiness() {
                // Happy case. We found a better solution.
                updated = Float::new(1.0).unwrap();
            } else {
                // Sad case. We need to go back by performing the same swap again.
                make_swap(&mut plan, swap);
                updated = Float::new(0.0).unwrap();
            }

            // Update our estimate of how often we update the solution.
            update_ema = shift_ema(update_ema, updated);
        }
        plan
    }
}

#[derive(Debug, Clone, Hash, PartialEq, Eq)]
```

```rust
pub struct LahcPlanner<R> {
    rng: R,

    /// How far back do we look?
    queue_size: NonZeroUsize,

    /// How infrequent do updates need to get before we give up?
    termination_threshold: Float,
}

impl<R> LahcPlanner<R>
where
    R: Rng,
{
    pub fn new(rng: R, termination_threshold: Float) -> Self {
        Self {
            rng,
            queue_size: NonZeroUsize::new(1000).unwrap(),
            termination_threshold,
        }
    }
}

impl<R> SeatingPlanner for LahcPlanner<R>
where
    R: Rng,
{
    fn plan(&mut self, problem: &Problem) -> Plan {
        let relationships = &problem.relations;
        let n_tables = problem.n_tables;
        let table_size = relationships.len() / n_tables;

        // Initialise our queue full of random solutions.
        let mut queue = VecDeque::with_capacity(self.queue_size.get());
        for _ in 0..self.queue_size.get() {
            queue.push_back(random_plan(&mut self.rng, relationships.len(), n_tables))
        }

        let mut update_ema = Float::new(1.0).unwrap();

        while update_ema >= self.termination_threshold {
            // Try a new solution and compare it to the front *and* back of our queue.
            // (We can't avoid allocation as we did with naive hill-climbing as we need
            // our old solution whether we accept the new one or not.)
            let mut new_plan = queue.back().cloned().expect("nonempty queue");
            let swap = get_random_swap(&mut self.rng, n_tables, table_size);
            make_swap(&mut new_plan, swap);

            // Find out how good the new solution is.
            let new_metrics = Metrics::new(&new_plan, relationships);
            let new_happiness = new_metrics.total_happiness();

            // Compare the new solution to the newest and oldest in the queue.
            let compare_to: [&Plan; 2] = [queue.front().unwrap(), queue.back()
                .unwrap()];
            let to_update = compare_to
                .iter()
                .any(|other| new_happiness > Metrics::new(other, relationships)
                    .total_happiness());

            // Keep the new solution if it is good enough, reject if not.
            let updated: Float;
            if to_update {
                queue.pop_front();
                queue.push_back(new_plan);
                updated = Float::new(1.0).unwrap();
            } else {
                updated = Float::new(0.0).unwrap();
            }

            // Update our estimate of how often we update the solution.
            update_ema = shift_ema(update_ema, updated);
        }

        // Select the best solution we have on record.
        queue
            .into_iter()
            .max_by_key(|plan| Metrics::new(plan, &relationships).total_happiness())
            .expect("Queue length is not zero.")
    }
}

/// Picks two locations out of a hat and creates the instruction to swap
/// the people there.
///
/// Technically, could swap two people at the same table, or even pick
/// exactly the same seat twice. This happens seldom enough that we
/// don't worry about it.
fn get_random_swap<R>(mut rng: R, n_tables: usize, table_size: usize) -> Swap
where
    R: Rng,
{
    let table1 = rng.gen_range(0..n_tables);
    let table2 = rng.gen_range(0..n_tables);

    let seat1 = rng.gen_range(0..table_size);
    let seat2 = rng.gen_range(0..table_size);

    Swap {
        table1,
        table2,
        seat1,
        seat2,
    }
}

/// Swaps the people at the locations specified in the ‘Swap‘.
///
/// We can’t use ‘std::mem::swap‘ because the two locations
/// have the same lifetime and the borrow-checker would have
/// our heads. If you don’t know what the borrow-checker is
/// don’t worry about it.
fn make_swap(plan: &mut [Vec<usize>], swap: Swap) {
    let tmp = plan[swap.table1][swap.seat1];
    plan[swap.table1][swap.seat1] = plan[swap.table2][swap.seat2];
    plan[swap.table2][swap.seat2] = tmp;
}

/// Creates a random plan, used to initialise the hill-climbing solutions.
fn random_plan<R>(mut rng: R, n_guests: usize, n_tables: usize) -> Plan
where
    R: Rng,
{
    assert_eq!(n_guests % n_tables, 0);

    // Generate a random permutation of guests.
    let mut permutation = (0..n_guests).collect::<Vec<usize>>();
    permutation.shuffle(&mut rng);

    // Chunk the guests into tables.
    let table_size = n_guests / n_tables;
    permutation
        .chunks_exact(table_size)
        .map(ToOwned::to_owned)
        .collect::<Vec<Vec<usize>>>()
}

#[cfg(test)]
```

```
mod tests {
    use super::*;

    #[test]
    fn plan_random_init() {
        let n_tables = 12;
        let table_size = 5;
        let n_guests = n_tables * table_size;
        let plan = random_plan(thread_rng(), n_guests, n_tables);

        // Correct number of tables.
        assert_eq!(plan.len(), n_tables);

        // Correct number of guests at each table.
        for table in &plan {
            assert_eq!(table.len(), table_size);
        }

        // Check that each guest appears exactly once.
        let mut guest_appearances = vec![0; n_guests];
        for table in &plan {
            for guest in table {
                guest_appearances[*guest] += 1;
            }
        }
        for n in guest_appearances {
            assert_eq!(n, 1);
        }
    }
}
```

6 src/bin/hill-solve.rs

```
use dissertation::{run, HillClimbingPlanner};

use ordered_float::NotNan;
use rand::prelude::*;

fn main() -> anyhow::Result<()> {
    let solver = HillClimbingPlanner::new(thread_rng(), NotNan::new(0.001).unwrap());
    run(solver)
}
```

7 src/bin/lahc-solve.rs

```
use dissertation::{run, LahcPlanner};

use ordered_float::NotNan;
use rand::prelude::*;

fn main() -> anyhow::Result<()> {
    let solver = LahcPlanner::new(thread_rng(), NotNan::new(0.001).unwrap());
    run(solver)
}
```

8 src/bin/data-gen.rs

```
use std::fs::File;
use std::io::{self, Write};
use std::ops::Range;
use std::path::PathBuf;
use std::str::FromStr;

use dissertation::{GuestRelations, Problem};

use anyhow::anyhow;
use rand::prelude::*;
use structopt::StructOpt;

#[derive(Debug, PartialEq, Eq)]
enum GenerationMethod {
    Random,
    CompleteComponents,
    Rings,
    Tense,
}

impl FromStr for GenerationMethod {
    type Err = anyhow::Error;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        match s.to_lowercase().as_str() {
            "rand" | "random" => Ok(GenerationMethod::Random),
            "comp" | "complete" | "complete-components" =>
                Ok(GenerationMethod::CompleteComponents),
            "ring" | "rings" => Ok(GenerationMethod::Rings),
            "tense" => Ok(GenerationMethod::Tense),
            _ => Err(anyhow!("Unrecognised generation method")),
        }
    }
}

#[derive(Debug, StructOpt)]
struct Opt {
    method: GenerationMethod,
    n_tables: usize,
    table_size: usize,
    output: Option<PathBuf>,
}

fn main() -> anyhow::Result<()> {
    let opt = Opt::from_args();

    if let Err(e) = run(opt) {
        eprintln!("{}", e);
        Err(e)
    } else {
        Ok(())
    }
}

fn run(opt: Opt) -> anyhow::Result<()> {
    let relations = match opt.method {
        GenerationMethod::Random => random_relations(opt.n_tables * opt.table_size),
        GenerationMethod::CompleteComponents =>
            complete_components(opt.n_tables, opt.table_size),
        GenerationMethod::Rings => rings(opt.n_tables, opt.table_size),
        GenerationMethod::Tense => tense(opt.n_tables * opt.table_size),
    };
    let problem = Problem {
        relations,
        n_tables: opt.n_tables,
    };

    let mut out: Box<dyn Write> = match opt.output {
        None => Box::new(io::stdout()),
        Some(path) => Box::new(File::create(path)?),
    };
    serde_json::to_writer(&mut out, &problem)?;
    Ok(())
}

/// Tense weddings where people have strong feelings about other guests,
/// positive or negative.
```

```
fn tense(n_guests: usize) -> GuestRelations {
    let mut rng = thread_rng();
    let mut relations = vec![vec![0; n_guests]; n_guests];

    for i in 0..n_guests {
        for j in 0..i {
            // Assume relations are normally distributed
            let r: f64 = {
                let base_dist = rand::distributions::Uniform::new(0.0, 1.0);
                (0..12).map(|_| base_dist.sample(&mut rng)).sum::<f64>() - 6.0
            };
            relations[i][j] = (r * 30.0) as i64;
            relations[j][i] = (r * 30.0) as i64;
        }
    }

    GuestRelations::new(relations)
}

/// Tables where everyone knows each other.
fn complete_components(n_tables: usize, table_size: usize) -> GuestRelations {
    let n_guests = n_tables * table_size;
    let mut relations = vec![vec![0; n_guests]; n_guests];

    for table in 0..n_tables {
        for i in table * table_size..(table + 1) * table_size {
            for j in table * table_size..(table + 1) * table_size {
                relations[i][j] = 1;
            }
        }
    }

    for i in 0..n_guests {
        relations[i][i] = 0;
    }

    GuestRelations::new(relations)
}

/// Tables where A knows B knows C knows D knows E knows A,
/// or similar patterns.
fn rings(n_tables: usize, table_size: usize) -> GuestRelations {
    let n_guests = n_tables * table_size;
    let mut relations = vec![vec![0; n_guests]; n_guests];

    for table in 0..n_tables {
        for i in 0..table_size {
            let j = (i + 1) % table_size;
            relations[table * table_size + i][table * table_size + j] = 1;
            relations[table * table_size + j][table * table_size + i] = 1;
        }
    }

    GuestRelations::new(relations)
}

/// Assign everyone a small number of friends, and then a smaller number
/// of friends-of-friends.
fn random_relations(n_guests: usize) -> GuestRelations {
    // We use an adjacency-list representation inside the algorithm,
    // and convert it to an adjacency matrix at the end.
    let mut friend_lists = random_friend_lists(n_guests);
    friends_of_friends(&mut friend_lists);

    let mut relationships = vec![vec![0; n_guests]; n_guests];
    fill_adj_matrix(&friend_lists, 1, &mut relationships);
    GuestRelations::new(relationships)
}

fn random_friend_lists(n_guests: usize) -> Vec<Vec<usize>> {
    let mut rng = thread_rng();

    let mut friend_lists = vec![Vec::new(); n_guests];

    // Start by assigning everyone at least one random friend.
    for i in 0..n_guests {
        while rng.gen_range(0..=friend_lists[i].len()) == 0 {
            let j = random_associate(&mut rng, i, 0..n_guests);
            friend_lists[i].push(j);
            friend_lists[j].push(i);
        }
    }

    friend_lists
}

fn friends_of_friends(friend_lists: &mut [Vec<usize>]) {
    let mut rng = thread_rng();

    for i in 0..friend_lists.len() {
        // A random number of times, but with decreasing probability.
        // Not guaranteed to run even once.
        while rng.gen_range(0..=friend_lists[i].len()) == 0 {
            let mutual_friend = friend_lists[i]
                .choose(&mut rng)
                .copied()
                .expect("Everyone should have at least one friend by now.");

            // Find a friend of our mutual friend, and make them friends
            // with the current guest.
            let new_friend = friend_lists[mutual_friend]
                .choose(&mut rng)
                .copied()
                .expect("Everyone should have at least one friend by now.");

            friend_lists[i].push(new_friend);
            friend_lists[new_friend].push(i);
        }
    }
}

/// Pick someone out of the crowd for our ‘person‘ to be friends with.
/// They cannot be friends with themselves.
fn random_associate<R>(mut rng: R, person: usize, choices: Range<usize>) -> usize
where
    R: Rng,
{
    loop {
        let associate = rng.gen_range(choices.clone());
        if associate != person {
            return associate;
        }
    }
}

/// Takes an adjacency list and fills up an adjacency matrix with the same
/// data. Can only fill in one value for the nonzero entries of the matrix.
fn fill_adj_matrix(lists: &[Vec<usize>], val: i64, matrix: &mut [Vec<i64>]) {
    for (guest_id, rel_list) in lists.iter().enumerate() {
        for &rel_id in rel_list {
            if rel_id != guest_id {
                // No-one is considered a relative of themselves.

                // The adjacency list should be symmetrical, so we don’t need to
                // reflect this.
                matrix[guest_id][rel_id] = val;
            }
        }
    }
}
```

9 src/bin/mass-data-gen.rs

```
use std::ffi::OsString;
use std::fs::{self, File};
use std::io::{self, BufRead, BufReader};
use std::path::{Path, PathBuf};
use std::process::{Child, Command};

use anyhow::Context;
use rand::prelude::*;
use structopt::StructOpt;

#[derive(StructOpt)]
struct Opt {
    suite: PathBuf,
}

// Take a text file of specifications for weddings and create
// the actual data, placed in a folder located in the same place
// and with a similar name to the spec file.
fn main() -> anyhow::Result<()> {
    let Opt { suite } = Opt::from_args();

    let spec_file = File::open(&suite)?;
    let target_dir_path = create_target_dir(&suite)?;

    mass_generate_data(&spec_file, &target_dir_path)?;

    Ok(())
}

fn create_target_dir(suite: &Path) -> io::Result<PathBuf> {
    let mut target_dir_path = suite.parent().map(ToOwned::to_owned)
        .unwrap_or_default();
    target_dir_path.push(suite.file_stem().expect("Please name your suite suitably."));

    if let Err(e) = fs::create_dir(&target_dir_path) {
        if e.kind() != io::ErrorKind::AlreadyExists {
            return Err(e);
        }
    }
    Ok(target_dir_path)
}

fn mass_generate_data(spec_file: &File, target_dir: &Path) -> anyhow::Result<()> {
    let specs = read_specs(spec_file)?;

    let processes = specs
        .iter()
        .map(|spec| {
            let mut filename = OsString::new();
            let mut command = Command::new("./target/release/data-gen");
            for word in spec.split_whitespace() {
                command.arg(word);
                // Pad numbers in the filenames with 0s so alphabetical
                // ordering coincides with size.
                match word.parse::<usize>() {
                    Ok(n) => filename.push(format!("{:03}", n)),
                    Err(_) => filename.push(word),
                }
                filename.push("_");
            }
            filename.push(thread_rng().next_u32().to_string());
            filename.push(".txt");
            command.arg(target_dir.join(filename));
            command
                .spawn()
                .with_context(|| format!("unable to spawn process for {}", spec))
        })
        .collect::<Result<Vec<Child>, anyhow::Error>>()?;

    for mut proc in processes {
        proc.wait()?;
    }

    Ok(())
}

fn read_specs(file: &File) -> io::Result<Vec<String>> {
    BufReader::new(file)
        .lines()
        .filter(|maybe_line| {
            maybe_line
                .as_ref()
                .map(|line| !line.is_empty())
                .unwrap_or(true)
        })
        .collect()
}
```

10 src/bin/score.rs

```
use std::fs::{self, DirEntry, File};
use std::io::{Read, Write};
use std::path::{Path, PathBuf};
use std::process::{Command, Stdio};

use dissertation::metrics::Metrics;
use dissertation::{Plan, Problem};

use anyhow::{anyhow, Context};
use serde::Serialize;
use structopt::StructOpt;

#[derive(StructOpt)]
struct Opt {
    solver: PathBuf,
    problem: PathBuf,
}

#[derive(Debug, Clone, Serialize)]
struct Record {
    // Data about the problem
    wedding: PathBuf,
    n_people: usize,
    n_tables: usize,

    // Metrics of solution quality.
    total_happiness: i64,
    mean_happiness: f64,
    median_happiness: f64,
    min_happiness: i64,
    max_happiness: i64,
    n_lonely: usize,

    // Time spent on the problem.
    // Can’t use ‘Duration‘ becuase this is going into a csv.
    seconds: f64,
}

fn main() -> anyhow::Result<()> {
    let opt = Opt::from_args();
    if let Err(e) = run(opt) {
        eprintln!("{}", e);
        Err(e)
    } else {
        Ok(())
    }
}

fn run(opt: Opt) -> anyhow::Result<()> {
    let records = score_path(&opt.solver, &opt.problem)?;

    let out_file = create_out_file(&opt.solver, &opt.problem)?;
    let mut writer = csv::Writer::from_writer(out_file);

    for record in records {
        writer.serialize(record)?;
    }

    Ok(())
}

fn create_out_file(solver: &Path, problem: &Path) -> anyhow::Result<File> {
    let solver_name = solver.file_stem().unwrap();
    let problem_name = problem.file_stem().unwrap();

    let mut csv_name = solver_name.to_owned();
    csv_name.push("_");
    csv_name.push(problem_name);

    let path = problem.with_file_name(csv_name).with_extension("csv");
    File::create(&path).with_context(move || format!("Could not create output file: {:?}
}

fn score_path(solver: &Path, problem: &Path) -> anyhow::Result<Vec<Record>> {
    if problem.is_file() {
        const N_RUNS: usize = 10;

        let records: anyhow::Result<Vec<Record>> = (0..N_RUNS)
            .map(|_| {
                let wedding_file = File::open(problem)
                    .with_context(
                        || format!("Could not open problem file: {:?}", problem))?;
                score_single(solver, &wedding_file, problem.to_owned()).with_context(||
                    format!("Could not run {:?} on wedding {:?}.", solver, problem)
                })
            })
            .collect();

        records
    } else if problem.is_dir() {
        let entries = fs::read_dir(problem)
            .with_context(|| format!("Could not open directory {:?}.", problem))?;

        score_suite(solver, entries)
    } else {
        Err(anyhow!(format!(
            "Could not recognise {:?}. Possibly a broken symlink?",
            problem
        )))
    }
}

fn score_single(
    solver: &Path,
    mut wedding: &File,
    wedding_name: PathBuf,
) -> anyhow::Result<Record> {
    // Create the solver as a child process.
    let mut solver = Command::new(&solver)
        .stdin(Stdio::piped())
        .stdout(Stdio::piped())
        .spawn()
        .with_context(|| "Unable to spawn solver.")?;

    // Read the problem.
    let mut problem_txt = Vec::new();
    wedding
        .read_to_end(&mut problem_txt)
        .with_context(|| "Could not read problem from file.")?;

    // Pipe the problem to the child.
    let child_stdin = solver.stdin.as_mut().expect("We gave the child a stdin.");
    child_stdin
        .write_all(&problem_txt)
        .with_context(|| "Could not pipe problem to solver.")?;
    child_stdin
        .flush()
        .with_context(|| "Could not flush solver's stdin.")?;

    // Take note of when the child started solving
    // AFTER we give them the data. We don't want to time
    // it’s I/O performance.
    let time_begin = std::time::Instant::now();

    // While waiting for the solver, deserialise the problem ourselves
    // so we can evaluate the solver’s performance.
    let problem_data: Problem =
        serde_json::from_slice(&problem_txt).with_context(
            || "Could not deserialise problem.")?;

    // Get the solution from the child.
    let output = solver.wait_with_output()?;

    // Find out how long the child took.
    let duration = time_begin.elapsed();

    if !output.stderr.is_empty() {
        return Err(anyhow!("Solver experienced a problem."));
    }

    let plan: Plan = serde_json::from_slice(&output.stdout)
        .with_context(|| "Could not parse output from solver.")?;

    // Find out how good the solution is and return.
    let metrics = Metrics::new(&plan, &problem_data.relations);
    let score = Record {
        wedding: wedding_name,
        n_people: problem_data.relations.len(),
        n_tables: problem_data.n_tables,
        total_happiness: metrics.total_happiness(),
        mean_happiness: metrics.mean_happiness(),
        median_happiness: metrics.median_happiness(),
        min_happiness: metrics.min_happiness(),
        max_happiness: metrics.max_happiness(),
        n_lonely: metrics.n_lonely(),
        seconds: duration.as_secs_f64(),
    };
    Ok(score)
}

fn score_suite<I, E>(solver: &Path, suite: I) -> anyhow::Result<Vec<Record>>
where
    I: Iterator<Item = Result<DirEntry, E>>,
{
    // Gather the whole suite and go through it alphabetically.
    // The files are named such that alphabetical precedence => smaller problem
    // This means we can find out when things start to get ugly in performance.
    let mut entries: Vec<DirEntry> = suite.filter_map(Result::ok).collect();
    entries.sort_unstable_by_key(|entry| entry.path());

    let mut scores = Vec::with_capacity(entries.len());
    for entry in entries {
        let path_to_solve = &entry.path();
        dbg!(path_to_solve);
        scores.extend(score_path(solver, path_to_solve)?);
        dbg!(&scores);
        eprintln!();
    }

    Ok(scores)
}
```

11 py/src/int-prog.py

```
#!py/venv/bin/python

import json
import sys

from dataclasses import dataclass
from typing import List
from ortools.linear_solver import pywraplp


@dataclass
class Problem:
    guest_relations: List[List[int]]
    n_tables: int


class Solution:
    def __init__(self, solver, status, variables):
        self.solver = solver
        self.status = status
        self.variables = variables


def main():
    problem = get_problem()
    solution = plan_dinner(problem.guest_relations, problem.n_tables)
    display_sol(solution)
```

```
def plan_dinner(guest_relations, n_tables):
    """Create a seating plan to make people happy.

    args:
    * guest_relations: a list of lists signifying relationship between guests i and j.
                       should be square-dimensioned, integer-valued, symmetric, and 0
                       along the diagonal.
    * n_tables: the number of tables. Should evenly divide the number of guests.
    """

    assert (
        len(guest_relations) % n_tables == 0
    ), "Partial filling of tables not yet supported."

    solver = pywraplp.Solver.CreateSolver("SCIP")

    n_guests = len(guest_relations)
    table_size = int(n_guests / n_tables)

    variables = Variables(solver, n_guests, n_tables)
    add_constraints(solver, variables)
    solver.Maximize(get_objective(guest_relations, variables))

    status = solver.Solve()

    return Solution(solver, status, variables)


class Variables:
    def __init__(self, solver, n_guests, n_tables):
        # indexed [table][guest]
        self.at_table = [
            [solver.IntVar(0, 1, f"at_table[{i}{j}]") for j in range(n_guests)]
            for i in range(n_tables)
        ]

        # indexed [table][guest1][guest2]
        self.pair_at_table = [
            [
                [
                    solver.IntVar(0, 1, f"pair_at_table[{i}][{j}][{k}]")
                    for k in range(n_guests)
                ]
                for j in range(n_guests)
            ]
            for i in range(n_tables)
        ]

        self.n_guests = n_guests
        self.n_tables = n_tables


def get_objective(guest_relations, variables):
    obj = 0
    for i in range(variables.n_tables):
        for j in range(variables.n_guests - 1):
            for k in range(j + 1, variables.n_guests):
                obj += guest_relations[j][k] * variables.pair_at_table[i][j][k]
    return obj


def add_constraints(solver, variables):
    max_at_table = int(variables.n_guests / variables.n_tables)

    # original paper | eq 2 - a guest must be seated at exactly one table.
    for j in range(variables.n_guests):
        tables_seated_at = 0
        for i in range(variables.n_tables):
            tables_seated_at += variables.at_table[i][j]
        solver.Add(tables_seated_at == 1)

    # original paper | eq 3 - a table can only fit so many people.
    for i in range(variables.n_tables):
        people_seated = 0
        for j in range(variables.n_guests):
            people_seated += variables.at_table[i][j]
        solver.Add(people_seated <= max_at_table)

    # original paper | eq 5 - join the two types of variables. Scale of eq 3.
    for i in range(variables.n_tables):
        for k in range(variables.n_guests):
            lhs = 0
            for j in range(variables.n_guests):
                lhs += variables.pair_at_table[i][j][k]
            solver.Add(lhs <= max_at_table * variables.at_table[i][k])

    # original paper | eq 6 - mirror of eq 5.
    for i in range(variables.n_tables):
        for j in range(variables.n_guests):
            lhs = 0
            for k in range(variables.n_guests):
                lhs += variables.pair_at_table[i][j][k]
            solver.Add(lhs <= max_at_table * variables.at_table[i][j])


def get_problem():
    problem_json = json.load(sys.stdin)
    problem = Problem(
        problem_json["relations"]["relationships"], problem_json["n_tables"]
    )

    # We require nonnegative weights only for linearisation, otherwise the algorithm
    # can
    # just lie and ignore the fact that two people are sat
    # next to each other if they don't get along.

    worst = min(min(rs) for rs in problem.guest_relations)
    if worst < 0:
        shift = abs(worst)
        for rs in problem.guest_relations:
            for i in range(len(rs)):
                rs[i] += shift

    return problem


def display_sol(solution):

    # The current solution is a list of tables, where each table has a 1 or 0 depending
    # on whether that guest is there or not.
    # We convert that to a list of tables, where each table is a list of integers,
    # representing guest-ids.

    tables_present = [
        [j for j, present in enumerate(table_binary) if present.solution_value() > 0]
        for i, table_binary in enumerate(solution.variables.at_table)
    ]

    print(json.dumps(tables_present))


if __name__ == "__main__":
    main()
```