Graz University of Technology

Two Exponential Neighbourhoods for the TSP and Related Heuristics

Pyramidal and strongly balanced tours – theory and implementation

Bachelorarbeit

Eingereicht für das Bachelorstudium
Technische Mathematik
der Technischen Universität Graz

Vorgelegt von
Maksym Deineko

Betreut durch
Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Eranda Dragoti-Cela

September 2016

Abstract

Dynamic programming delivers solutions to many problems in combinatorial optimization; to this the traveling salesman problem and its special cases — where dynamic programming can often be applied effectively — are no exception.

Herein we describe a general procedure for constructing, given a description of such a solution, a working implementation thereof as well as of a corresponding iterative heuristic.

We then proceed to provide such implementation for two special cases of TSP — of which the first one (albeit not its extension to the heuristic method) is well known, while the other had yet at all to see a working implementation, which can be seen as a major goal of this effort — and present results thus obtained.

This page would have been intentionally left blank had we not chosen to mention it.

Contents

This page would have been intentionally left blank had we not chosen to mention it.

1. Introduction

> Nothing clears up a case so much as stating it to another person.
>
> *(Sherlock Holmes, «The Memoirs of Sherlock Holmes», Arthur Conan Doyle)*

The *traveling salesman problem (TSP)* can be colloquially stated as follows:

> A traveling merchant (or salesman) wishes to visit each city from a given list exactly once, starting from and returning to any select one of the cities given. Assuming that distances between the cities are known and not subject to change, what would be the shortest route for him to take?

This well studied problem in combinatorial optimization is known to be NP-hard [7]; hence, one is interested in efficiently solvable cases as well as useful heuristics, of which a good number exists.

In this paper we look at two exponential sets of feasible solutions over which the TSP can be solved in polynomial time as well as at extension of these to iterative heuristics. For this, we need to establish some basic definitions first.

1.1. Basic definitions and notation

The TSP, as most mathematical problems, can be modeled in a number of often equivalent or similar ways, depending on what aspects of the problem are relevant to the task at hand — which, for a problem as extensively studied as this one, results in a great number of common terms which from author to author can differ ever so slightly.

In what follows, our goal is to establish notation which can be easily interpreted as a (functional) computer program.

Definition 1.1 (cities, paths and tours)**.** Given $n \in \mathbb{N}\,(n \geq 2)$, let

$$\mathcal{N}_n \coloneqq \{1, \dots, n\}\,.$$

We will refer to elements of $\mathcal{N}_n$ as *cities*. A non-empty tuple $\left(p_1, p_2, \dots, p_m\right)$ of cities is called a *path* (of *walking length* $m-1$ *over* $\mathcal{N}_n$); it is called *simple* if it contains distinct entries only, with possible exception of its *end points* p_1 and p_m, i.e. if

$$\left|\left\{p_1, p_2, \dots, p_{m-1}\right\}\right| = \left|\left\{p_2, p_3, \dots, p_m\right\}\right| = m-1.$$

We shall refer to a path of non-zero walking length with distinct end points as an *open* path, one with equal end points — a *closed* one, or a *cycle*. A path of walking length 1 is also called an *edge*; a simple cycle of walking length n is called a *tour*. We then denote by $\mathcal{P}_n \coloneqq \bigcup_{k\in\mathbb{N}} \mathcal{N}_n^k$ the set of all paths and by $\mathcal{T}_n$ — the set of all tours over $\mathcal{N}_n$.

We extend the common notion of path *concatenation* to perform a single reduction, as well as to allow for convenient notation for images under said map defined as infix operator:

Definition 1.2 (path operations)**.** Define *path concatenation* to be the following map:

$$\oplus \ : \left(\mathcal{P}_n \cup 2^{\mathcal{P}_n}\right)^2 \to \mathcal{P}_n \cup 2^{\mathcal{P}_n} \quad \text{(infix)},$$

$$p \oplus q \coloneqq \begin{cases} \left(p_1, \dots, p_m, q_1, \dots, q_k\right), & p = \left(p_1, \dots, p_m\right) \in \mathcal{P}_n, \\ & q = \left(q_1, \dots, q_k\right) \in \mathcal{P}_n, \\ & p_m \neq q_1; \\ \left(p_1, \dots, p_{m-1}, q_1, \dots, q_k\right), & p = \left(p_1, \dots, p_m\right) \in \mathcal{P}_n, \\ & q = \left(q_1, \dots, q_k\right) \in \mathcal{P}_n, \\ & p_m = q_1; \\ \left\{p' \oplus q \mid p' \in p\right\}, & p \notin \mathcal{P}_n, q \in \mathcal{P}_n; \\ \left\{p \oplus q' \mid q' \in q\right\}, & p \in \mathcal{P}_n, q \notin \mathcal{P}_n; \\ \left\{p' \oplus q' \mid p' \in p, q' \in q\right\}, & p \notin \mathcal{P}_n, q \notin \mathcal{P}_n, \end{cases}$$

path reversal as

$$\mathrm{rev} \ : \ \mathcal{P}_n \to \mathcal{P}_n, \ \left(p_1, p_2, \dots, p_{m-1}, p_m\right) \mapsto \left(p_m, p_{m-1}, \dots, p_2, p_1\right),$$

and *path closure* as

$$\overline{\cdot} \ : \ \mathcal{P}_n \to \mathcal{P}_n, \quad \left(p_1, \dots, p_m\right) \mapsto \left(p_1, \dots, p_m\right) \oplus \left(p_1\right),$$

while also allowing for image notation: for $P \subseteq \mathcal{P}_n$, let

$$\overline{P} \coloneqq \left\{\overline{p} \mid p \in P\right\}.$$

We choose the concatenation operator to take precedence over set union.

Example 1.1. With above definition, following holds for paths over $\mathcal{P}_n$ (for $n \geq 3$):

$$\begin{aligned} (1,2) \oplus (3,2) &= (1,2,3,2), \\ (3,1) \oplus (1,1) &= (3,1,1), \\ \{(2),(1)\} \oplus (1) &= \{(2,1),(1)\}, \\ \{(1)\} \oplus \{(1),(2)\} \cup \{(3)\} &= \{(1),(1,2),(3)\}. \end{aligned}$$

Remark 1.1. Also note that path closure produces a bijection from the set of all simple open paths of walking length $n-1$ over $\mathcal{N}_n$ onto $\mathcal{T}_n$. Further, $\overline{\mathcal{T}_n} = \mathcal{T}_n$ holds true.

Definition 1.3 (costs and distances)**.** Given a $n \in \mathbb{N}\,(n \geq 2)$ and a matrix $C = \left(c_{i,j}\right) \in \mathbb{R}^{n \times n}$, which we shall call *cost* (or *distance) matrix*, we will refer to its elements as *distances* or *weights*. Now let

$$\omega_C \ : \ \mathcal{P}_n \to \mathbb{R}, \quad \left(p_1, p_2, \dots, p_m\right) \mapsto \sum_{k=1}^{m-1} c_{p_k, p_{k+1}}.$$

For a path p over $\mathcal{N}_n$ we then call $\omega_C\left(p\right)$ *cost* of p. An edge's cost is also called its *weight* (which agrees with the definition above).

We would like to stress here that requiring tours to explicitly include a city twice (as end points) allows us to employ same notion of cost for paths and tours, which, in turn, will allow for convenient recursive constructions later on.

Definition 1.4 (TSP)**.** Given a $n \in \mathbb{N}\,(n \geq 2)$, $\boldsymbol{M} \subseteq \mathbb{R}^{n\times n}$ and $\boldsymbol{T} \subseteq \mathcal{T}_n$, $\boldsymbol{T} \neq \emptyset$, as well as a total order $\boldsymbol{L}$ on $\mathcal{T}_n$, we define *traveling salesman problem* (*over* $(\boldsymbol{M}, \boldsymbol{T})$, also *TSP*) to be the map

$$\mathrm{tsp}_{L,M,T} : \boldsymbol{M} \to \boldsymbol{T}, \quad \boldsymbol{C} \mapsto \min_{L} \mathrm{tsp}^*{}_{M,T}\,(\boldsymbol{C})\,,$$

and

$$\mathrm{tsp}^*{}_{M,T} : \boldsymbol{M} \to 2^T, \quad \boldsymbol{C} \mapsto \arg\min_{\tau\in T} \omega_C\,(\tau) = \left(\omega_C \restriction_T\right)^{-1}\left(\left\{\min \omega_C\,(T)\right\}\right).$$

We call n the TSP's *dimension*. If $\boldsymbol{M}$ is a subset of symmetric matrices (over $\mathbb{R}$), the TSP is called *symmetric (sTSP)*, otherwise — *asymmetric (aTSP)*.

Remark 1.2. Both tsp* and tsp are well-defined maps: $1 \leq |\boldsymbol{T}| < \infty$ holds true above, hence ω_C assumes a minimum value over $\boldsymbol{T}$ and $\emptyset \notin \mathrm{tsp}^*{}_{M,T}\,(\boldsymbol{M})$.

Remark 1.3. Frequently, the TSP is defined via a map which provides the sets $\boldsymbol{M}$ and $\boldsymbol{T}$ for every dimension n. We make no such requirement — however, such an extension is easily established from context, and throughout the rest of this paper we will usually assume that some such dimension is given. Further, whenever the sets $\boldsymbol{M}, \boldsymbol{T}$ in above definition are established elsewhere, they are usually omitted from notation. The pair $(\boldsymbol{M}, \boldsymbol{T})$ is called a *class* or *(special) case* of TSP.

Remark 1.4. Also not unusual (and useful in practice) is a variation of TSP which returns the cost of optimal tour along with such a tour — transition between these definitions is usually obvious.

Remark 1.5. Often, the convention is to define TSP to be the symmetric version — we shall make no such assumption without a prior notice. On the other hand, following convention is useful: observe how it follows from definitions 1.3 and 1.4 that $\mathrm{tsp}^*_{M,T}\,(\boldsymbol{C}) = \mathrm{tsp}^*_{M',T}\,(\boldsymbol{C} + \lambda \boldsymbol{J})$ for all $\lambda \in \mathbb{R}$ and $\boldsymbol{C} \in \boldsymbol{M}$, where $\boldsymbol{J}$ is a matrix of ones (of suitable dimension), and we have a bijection from $\boldsymbol{M}$ onto $\boldsymbol{M}'$. Hence we can always assume

$$c_{i,j} \geq 0 \quad \forall i, j \in \mathcal{N}_n$$

and shall do so henceforth.

Remark 1.6. Cost equality naturally induces a partition of $\mathcal{T}_n$ (or its subset $\boldsymbol{T}$ above). We are not always interested in all elements of tsp* $(\boldsymbol{C})$ or its representative given by a particular order $\boldsymbol{L}$ (as produced by tsp$_L$ $(\boldsymbol{C})$) but rather in any such tour. We hence treat $\boldsymbol{L}$ as implicit argument to tsp$_L$, omitting it from notation where possible — in practice, it is often given implicitly by construction, or may depend on even more arguments (such as some probability distribution) — still, this choice of representative is something we need to be aware of throughout, as it propagates to any construct using tsp — such as iterative methods we shall present herein.

1.2. Exponential neighbourhoods and local search

Since the TSP in its general form is NP-hard, one is naturally interested in efficiently solvable cases as well as good heuristics; this leads us to following general definitions.

Definition 1.5 ((exponential) neighbourhood)**.** A *neighbourhood* (*in* $\mathcal{T}_n$ or *over* $\mathcal{N}_n$) is a map

$$F : \mathcal{T}_n \to 2^{\mathcal{T}_n} \quad \text{s.t. } \tau \in F(\tau)\ \forall \tau \in \mathcal{T}_n.$$

$F(\tau)$ is also called *neighbourhood of* τ. Once again, n can be seen either as given dimension or implicit argument to F; a neighbourhood F is called *exponential* if for some $a \in \mathbb{R}, a > 1$,

$$|F(\tau)| = \Omega(a^n) \qquad \forall \tau \in \mathcal{T}_n.$$

Definition 1.6 (local search)**.** Given a neighbourhood F in $\mathcal{T}_n$ as well as $M \subseteq \mathbb{R}^{n\times n}$ and $C \in M$, we define *local search* (*in* or *over* F) to be the map

$$l_C : \mathcal{T}_n \to \mathcal{T}_n, \quad \tau \mapsto \mathrm{tsp}_{M,F(\tau)}(C).$$

F (as well as l_C) is said to be *polynomially solvable* if $l_{C'} \in \mathsf{P}\ \forall C' \in M$ (where time complexity is measured in relation to n).

Note how for any tour τ local search defines a map on M via $C' \mapsto l_{C'}(\tau)$, which can also be seen as approximation of $\mathrm{tsp}_{M,\mathcal{T}_n}$.

Remark 1.7. Not a universally common requirement, our reason to demand in the definition above that $F(\tau)$ include τ is to guarantee that for any tour τ and any cost matrix C,

$$\omega_C\left(l_C(\tau)\right) \leq \omega_C(\tau).$$

This naturally yields an iterative improvement heuristic:

Definition 1.7 (ILS)**.** Given a local search function as above and a *starting tour* τ^*, we define *iterative* or *iterated local search* (*ILS*) to be the map assigning to each $C \in M$ the fixed point of

$$\tau \mapsto \begin{cases} \tau & \text{if } \omega_C\left(l_C(\tau)\right) = \omega_C(\tau), \\ l_C(\tau) & \text{otherwise} \end{cases}$$

which is reached by iteration from τ^*. We shall sometimes refer to such a fixed point as *stale iteration*.

In practice, termination conditions employed in ILS may vary and commonly include such parameters as number of iterations performed, time elapsed or some cost improvement metric.

Remark 1.8. There seems to exist no commonly accepted distinction between local search and iterated local search. Often, the former is not required to solve the problem in $F(\tau)$ but only in some small subset of it, or to consist of multiple such improvement steps, which is referred to as *anytime* heuristic. Thus, depending on the definition of what constitutes a step, one can potentially be seen as special case of the other.

1.3. Tours and permutations

Given the set $\mathcal{N}_n$ of n cities, let $\mathcal{S}_n$ denote the symmetric group (on $\mathcal{N}_n$) from now on.

Definition 1.8 (associated permutation)**.** We define $\sigma_\bullet$ (using argument-in-subscript notation) to be the map

$$\sigma_\bullet : \mathcal{T}_n \to \mathcal{S}_n, \quad \tau = (p_1, p_2, \dots, p_n, p_1) \mapsto \sigma_\tau := \begin{pmatrix} 1 & 2 & \cdots & n \\ p_1 & p_2 & \cdots & p_n \end{pmatrix},$$

and, seeing how this constitutes a bijection, define π to be its inverse, while letting σ_τ^{-1} denote permutation inverse to σ_τ. We say then that τ and σ_τ are *associated* with one another.

Remark 1.9. We note here that function composition is default group operation on $\mathcal{S}_n$ and therefore can be omitted from notation where it is convenient and does not cause unnecessary ambiguity.

The map π, being a bijection, conveniently provides a natural extension of cost to $\mathcal{S}_n$ via $\omega_C \circ \pi$:

Definition 1.9 (cost of permutation)**.** For a cost matrix C of dimension n, we shall extend ω_C to $\mathcal{P}_n \cup \mathcal{S}_n$ by setting, for any $\rho \in \mathcal{S}_n$,

$$\omega_C(\rho) := \omega_C\left(\pi(\rho)\right).$$

Similarly, we can use the map $\sigma_\bullet$ for following convenient notation:

Definition 1.10 (action of $\mathcal{S}_n$)**.** For $\tau \in \mathcal{T}_n, \rho \in \mathcal{S}_n, C \in M \subseteq \mathbb{R}^{n\times n}, T \subseteq \mathcal{T}_n$, we define

$$\begin{aligned} \tau\rho &:= \sigma_\tau \rho, \\ \rho\tau &:= \rho\sigma_\tau, \\ \rho T &:= \pi\left(\left\{\rho\sigma_\eta \mid \eta \in T\right\}\right), \\ \rho C &:= \left(c_{\rho(i),\rho(j)}\right)_{i,j\in\mathcal{N}_n}, \\ \rho M &:= \left\{\rho\widehat{C} \mid \widehat{C} \in M\right\}. \end{aligned}$$

We can see that $\rho\tau$ is associated with the tour τ over cities reordered by ρ, and if we consider $\tau\rho$ to be a reordering of the tour τ, following result becomes intuitively agreeable:

Proposition 1.1. *For any permutation* $\rho \in \langle(1\ 2\ \cdots\ n)\rangle = \left\{(1\ 2\ \cdots\ n)^m \mid m \in \mathbb{Z}\right\}$,

$$\omega_C(\tau) = \omega_C\left(\tau\rho\right)$$

holds true for all $\tau \in \mathcal{T}_n$ *and all* $C \in \mathbb{R}^{n\times n}$.

Proof. Consider following natural isomorphism between $\mathbb{Z}/n\mathbb{Z}$ and $\mathcal{N}_n$: for $i \in \mathbb{Z}$, let

$$d(i) \coloneqq \big((i-1) \bmod n\big) + 1.$$

We have $d(\mathbb{Z}) = \mathcal{N}_n$, and $d \restriction_{\mathcal{N}_n} = \mathrm{id}_{\mathcal{N}_n}$ is trivially a bijection (as is ρ); further,

$$d\left(x+y\right) = d\left(d(x) + d\left(y\right)\right) \quad \forall x \in \mathbb{Z}\ \forall y \in \mathbb{Z}, \tag{1.1}$$

and with $d(n+1) = 1$ we can now write the cost function from definition 1.3 as

$$\omega_C(\tau) = \sum_{i=1}^{n} c_{\sigma_\tau(i),\sigma_\tau(d(i+1))}. \tag{1.2}$$

We also note that $\exists K \in \mathcal{N}_n$ s.t. $\rho(i) = d(i+K)\ \forall i \in \mathcal{N}_n$, and, using eq. (1.1), obtain

$$\begin{aligned} \rho\left(d(i+1)\right) = d\left(d(i+1)+K\right) = d\left(d(i+1)+d(K)\right) = d(i+1+K) = \\ = d\left(d(i+K)+d(1)\right) = d\left(d(i+K)+1\right) = d\left(\rho(i)+1\right) \end{aligned} \tag{1.3}$$

for any city i. It now follows that

$$\begin{aligned} \omega_C\left(\tau\rho\right) = \omega_C\left(\pi\left(\sigma_\tau \circ \rho\right)\right) &\overset{(1.2)}{=} \sum_{i=1}^{n} c_{\sigma_\tau(\rho(i)),\sigma_\tau\left(\rho\left(d(i+1)\right)\right)} = \\ &\overset{(1.3)}{=} \sum_{k=1}^{n} c_{\sigma_\tau(k),\sigma_\tau(d(k+1))} \overset{(1.2)}{=} \omega_C(\tau). \end{aligned}$$

□

This shows, in particular, how restricting the tour set in the TSP to a fixed starting city does not necessarily signify a qualitative restriction of solutions set.

We now want to establish a connection between permutations of cities, reordered tours and solutions to the TSP.

Lemma 1.2. *For any $\rho \in \mathcal{S}_n, C \in \mathbb{R}^{n\times n}, \tau \in \mathcal{T}_n$,*

$$\omega_C\left(\rho\,\tau\right) = \omega_{\rho\,C}(\tau)$$

holds true.

Proof. Borrowing definition of d from the proof of proposition 1.1, and using eq. (1.2) along with definition 1.10, we obtain

$$\omega_{\rho\,C}(\tau) = \sum_{i=1}^{n} c_{\rho(\sigma_\tau(i)),\rho\left(\sigma_\tau(d(i+1))\right)} = \omega_C\left(\rho \circ \sigma_\tau\right) = \omega_C\left(\rho\,\tau\right).$$

□

Corollary 1.3 (reordered TSP). *For any* $\rho \in S_n, C \in M \subseteq \mathbb{R}^{n\times n}, \tau \in \mathcal{T}_n$,

$$\begin{aligned}\operatorname{tsp}^*_{M,\rho T}(C) &= \rho\ \operatorname{tsp}^*_{\rho M,T}\left(\rho C\right),\\ \operatorname{tsp}_{M,\rho T}(C) &= \pi\big(\rho\ \operatorname{tsp}_{\rho M,T}\left(\rho C\right)\big)\end{aligned}$$

(where latter holds for any pair of linear orders congruent under ρ*).*

Proof. Using lemma 1.2 and definition 1.4, we obtain

$$\begin{aligned}\operatorname{tsp}^*_{\{\rho C\},T}\left(\rho C\right) &= \operatorname*{arg\,min}_{\tau\in T}\omega_{\rho C}(\tau) = \operatorname*{arg\,min}_{\tau\in T}\omega_C\left(\rho\tau\right) =\\ &= \rho^{-1}\operatorname*{arg\,min}_{\rho\tau\in\rho T}\omega_C\left(\rho\tau\right) = \rho^{-1}\ \operatorname{tsp}^*_{\{C\},\rho T}(C),\end{aligned}$$

from which the rest follows. □

1.4. Tour sets as neighbourhoods

Given a non-empty set of tours T, we can choose some (any) element $\tau^* \in T$ and consider T to be a set of permutations of τ^* — its action then defines a neighbourhood in $\mathcal{T}_n$:

Definition 1.11 (induced neighbourhood)**.** Given a set of tours $T \subseteq \mathcal{T}_n$ and a tour $\tau^* \in T$, define F_T to be the map

$$F_T : \mathcal{T}_n \to 2^{\mathcal{T}_n}, \quad \tau \mapsto \pi\left(\left\{\sigma_\tau \circ \sigma^{-1}_{\tau^*} \circ \sigma_{\tau'} \mid \tau' \in T\right\}\right) = \left(\sigma_\tau \circ \sigma^{-1}_{\tau^*}\right)T.$$

We call F_T *neighbourhood induced by* T (*centred at* or *with centre* τ^*).

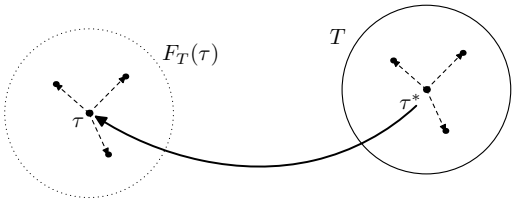


Figure 1.1: A non-empty set of tours T induces a neighbourhood F_T.

Remark 1.10. It is quickly verified that $\tau \in F_T(\tau)\ \ \forall\tau \in \mathcal{T}_n$, i.e. F_T is in fact a neighbourhood according to our definition. We further have $F_T\left(\tau^*\right) = T$. We deliberately omit the choice of centre from notation of induced neighbourhood, treating it as implicit parameter which is usually to be established along with the set T.

If we now can solve the TSP efficiently over a fixed set of tours T (for arbitrary cost matrices), choosing a $\tau^* \in T$ immediately allows us to construct a local search over F_T: applying corollary 1.3 to definition 1.11, we arrive at

$$l_C(\tau) = \pi\left(\rho\ \operatorname{tsp}_{\rho M,T}\left(\rho C\right)\right), \quad \text{where } \rho = \sigma_\tau\sigma^{-1}_{\tau^*}$$

(which is mirrored in our implementation), and choosing τ^* to be the starting tour in definition 1.7 then yields iterated local search.

2. Theory

To iterate is human, to recurse divine.

(L. Peter Deutsch)

Dynamic programming is a fruitful approach to tackling many problems in combinatorial optimization — this specifically includes traveling salesman problem and a number of its special cases. Here we want to present two exponential neighbourhoods which are polynomially solvable via dynamic programming solutions.

While the first one — *pyramidal tours* — can be considered widely known, less so is its extension to a viable local search heuristic which we will present herein. The second — *strongly balanced tours* — is somewhat more involved in its construction and so had yet to see an implementation until now.

Solutions as they are presented herein can be taken to reflect inner workings of our code.

2.1. Pyramidal tours

Pyramidal TSP, as seen in, e.g. [2, 6], yields a classic example of application of dynamic programming and can be defined as follows:

Definition 2.1 (pyramidal TSP)**.** A simple path

$$\big(p_1, \dots, p_k, q_1, \dots, q_m\big) \text{ s.t. } \begin{cases} p_i < p_{i+1} & \forall i \in \mathcal{N}_{k-1}, \\ q_j > q_{j+1} & \forall j \in \mathcal{N}_{m-1}, \end{cases}$$

is called *pyramidal*. We shall denote by Pyr_n the set of all pyramidal tours in $\mathcal{T}_n$, and call TSP restricted to Pyr_n *pyramidal TSP*.

We note here that the set Pyr_n induces a neighbourhood in $\mathcal{T}_n$ with centre $(1, 2, \dots, n, 1)$.

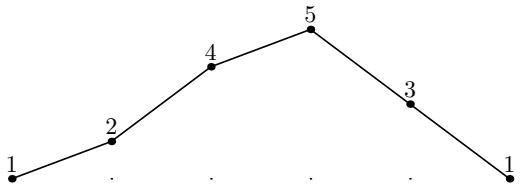


Figure 2.1: For a pyramidal tour τ, connected plot of σ_τ's (function) graph resembles a pyramid (here: $n = 5$, plot repeated at 1 for cosmetic reasons).

This definition of pyramidal tours, while explaining the name, does not yet offer a new solution to the corresponding optimization problem. To achieve that, first we observe that every pyramidal tour necessarily takes the shape

$$\big(1, p_1, \dots, p_k, n, q_1, \dots, q_m, 1\big) \text{ with } \begin{cases} p_i < p_{i+1} & \forall i \in \mathcal{N}_{k-1}, \\ q_j > q_{j+1} & \forall j \in \mathcal{N}_{m-1}, \end{cases}$$

and then notice how for any city between 1 and n, we can choose it to lie either to the left or to the right of n in the above representation and how that choice then uniquely defines the city's position in the tour.

This argument makes $\left|\mathrm{Pyr}_n\right| = \Theta\left(2^n\right)$ evident and following construction transparent:

Proposition 2.1 (recursive structure of Pyr_n)**.** *For cities i, j in $\mathcal{N}_n$, let $V\left(i,j\right)$ denote the set of all pyramidal paths $\left(i, p_1, \ldots, p_m, j\right)$ in $\mathcal{P}_n$ s.t. $\left\{p_1, \ldots, p_m\right\} = \{k, \ldots, n\}$ with $k = \max\left\{i,j\right\} + 1$. Then*

$$V\left(i,j\right) = \begin{cases} \left\{\left(i,j\right)\right\}, & n \in \left\{i,j\right\}, \\ (i) \oplus V\left(k,j\right) \bigcup V(i,k) \oplus \left(j\right) & \text{otherwise (k as above)}, \end{cases} \tag{2.1}$$

and $\mathrm{Pyr}_n = V(1,1)$.

Proof. The second case in eq. (2.1) becomes apparent when, using preceding argument, we choose each city's position in the tour one city at a time in increasing order. The rest follows directly from our definition of V. □

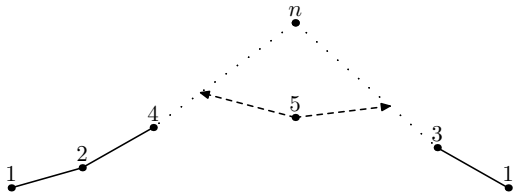


Figure 2.2: A tour in Pyr_n is constructed one city at a time.

This translates directly onto a dynamic programming solution to pyramidal TSP:

Corollary 2.2 (dynamic programming solution to pyramidal TSP)**.** *For a suitable path-valued definition of* $\arg\min'$*, the recurrence relation*

$$\begin{aligned} \Phi_C\left(i,j\right) \coloneqq & \begin{cases} \left(i,j\right), & n \in \left\{i,j\right\}, \\ \underset{T}{\arg\min}' \omega_C & \text{otherwise}, \end{cases} \\ \text{where} \quad T \coloneqq & \left\{(i) \oplus \Phi_C\left(k,j\right), \Phi_C\left(i,k\right) \oplus \left(j\right)\right\}, \\ & k \coloneqq \max\left\{i,j\right\} + 1, \end{aligned} \tag{2.2}$$

produces, for arbitrary cost matrices, a well-defined map Φ_C on $\mathcal{N}_n^2$, and via

$$\phi \;:\; C \;\mapsto\; \Phi_C\left(1,1\right)$$

a map on $\mathbb{R}^{n\times n}$. For latter,

$$\phi \equiv \mathrm{tsp}_{\mathbb{R}^{n\times n},\mathrm{Pyr}_n}$$

holds true.

Proof. Borrowing definition of V from proposition 2.1, we first note that, per induction over (2.1), $V\left(i,j\right)$ is never empty and the two sets under union in eq. (2.1) are always disjoint.

If we now choose arg min$'$ to select from two equal cost tours in T one deterministically and independently from C (say, the first of the two options in definition of T above), then this, per construction, produces a partial order on paths in $V\left(\mathcal{N}_n^2\right)$ (bar (n,n)) and a total order on Pyr_n.

We then see per induction over eqs. (2.1) and (2.2) that for all cities i,j and cost matrices C, $\Phi_C\left(i,j\right)$ is the smallest (according to said order) cost minimizing path in $V\left(i,j\right)$, from which the rest follows. □

A recurrence relation like the one introduced above can be visualized via so called *recursion tree* — a directed graph in which nodes correspond to different arguments to the relation and edges represent recursive dependence. For pyramidal TSP, fig. 2.3 shows such a graph.

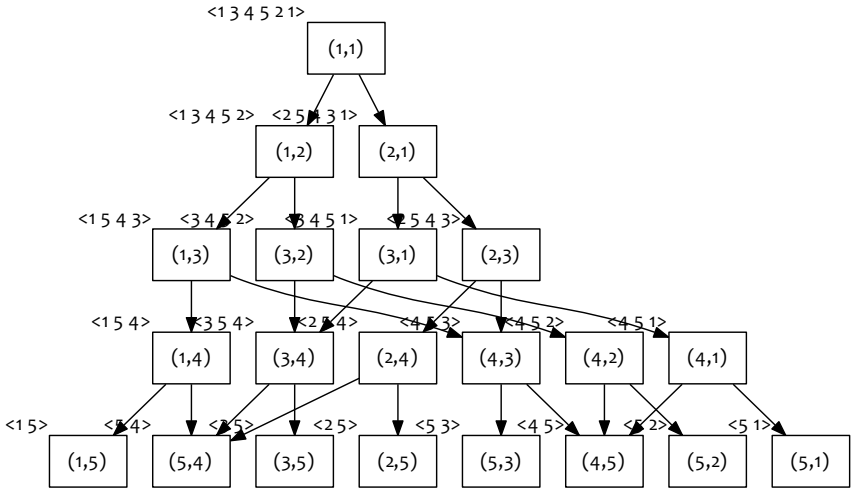


Figure 2.3: Pyramidal (a)TSP recursion tree along with values of Φ_C from corollary 2.2 for a sample cost matrix C ($n = 5$).

2.2. Recursion and complexity

While corollary 2.2 offers a solution to the pyramidal TSP, the function as it is presented in eq. (2.2) does not yet necessarily translate onto an efficient computer program. To this end, we need to perform the additional step of *memoization*.

The term, coined in computing (from *memo*), denotes the technique of evaluating a function no more than once at any point in its domain, storing evaluation results some-

where and retrieving (i.e. substituting) them for subsequent evaluations. Not unnatural to computations by hand, memoization can formally be seen as evaluating, in place of a function over some set, said function's graph over the set. In what now follows, X can thus be seen as some memory holding such evaluation results:

Proposition 2.3 (memoized solution to pyramidal TSP)**.** *Given a set of cities $\mathcal{N}_n$, let, for cities i, j, a cost matrix C and arbitrary function*[1] *X,*

$$\widehat{\Phi}_C\left((i,j),X\right) \coloneqq \begin{cases} \left(v^*, X\right), & \left((i,j),v^*\right) \in X \text{ for some } v^*, \\ \left((i,j), X \cup \left\{\left((i,j),(i,j)\right)\right\}\right) & \text{otherwise, if } n \in \left\{i,j\right\}, \\ \left(v, X'' \cup \left\{\left((i,j),v\right)\right\}\right) & \text{otherwise, where} \end{cases} \tag{2.3}$$

$$\begin{aligned} v &\coloneqq \arg\min{}' \omega_C \quad \text{over } \left\{(i) \oplus v', v'' \oplus (j)\right\}, \\ \left(v'', X''\right) &\coloneqq \widehat{\Phi}_C\left((i,k),X'\right), \\ \left(v', X'\right) &\coloneqq \widehat{\Phi}_C\left((k,j),X\right), \\ k &\coloneqq \max\left\{i,j\right\} + 1. \end{aligned}$$

Then, borrowing definition of Φ_C from corollary 2.2,

$$\widehat{\Phi}_C\left((1,1),\varnothing\right) = \left(\Phi_C(1,1), E\right),$$

where E is the graph of Φ_C over $\mathcal{N}_n^2 \setminus \{(n,n)\}$.

Proof. Comparing the construction of $\widehat{\Phi}_C$ in eq. (2.3) to that of Φ_C in eq. (2.2), we see that whenever X in (2.3) is a subset of Φ_C's graph, so is the second element of the tuple returned by $\widehat{\Phi}_C$, so that evaluation of $\widehat{\Phi}_C$ via (2.3) represents depth-first search over recursion tree of Φ_C — and with this the rest follows. □

Remark 2.1. This gives us a viable way of computing $\Phi_C(1,1)$ and, by extension, $\mathrm{tsp}_{\mathrm{Pyr}_n}$ — in fact, our code, which generated fig. 2.3 as its execution trace, closely mirrors this exact definition. It is also a reasonably efficient implementation, provided that we are allowed following assumptions:

(i) $\arg\min{}' \omega_C$ in definition of Φ_C above is computable in constant time: this is usually achieved by returning tour cost along with the tour[2];

(ii) tour concatenation in said definition is computable in constant time: in this case, an obviously reasonable assumption;

[1] For above construction to yield a well-defined map $\widehat{\Phi}_C$, X must be a set satisfying at least $(x,y) \in X \wedge x \in \mathcal{N}_n^2 \to z = x\ \forall (z,y) \in X$.

[2] It should be noted, without too much detail, that this also spares us the need to construct all but the optimal tour.

(iii) memory access (search for v^* in first case of (2.3), and set union in the other two) can be accomplished in constant time: sometimes overlooked, this is equivalent to existence of efficient hashing function on the domain of the map which is to be memoized — in this case, $(i,j) \mapsto (n+1) \times i + j$ is one such function.

Note that evaluating $\widehat{\Phi}_C$ following its definition corresponds to depth-first search over the recursion tree, size of which is quadratic in number of nodes, as it is in the number of edges — if we now consider memoization to be what is usually called edge marking in that it guarantees that every edge is visited no more than once, we intuitively arrive at following result:

Corollary 2.4. *Under assumptions made in remark 2.1, pyramidal TSP is polynomially solvable.*

Proof. Consider the map $\widehat{\Phi}_C$ as it was defined in proposition 2.3. If, for a bound M, we now rewrite the recurrence relation from said definition of $\widehat{\Phi}_C$ as

$$\begin{aligned} R\left((i,j),X\right) &:= \begin{cases} \left(M, X \cup \left\{(i,j)\right\}\right) & (i,j) \in X \text{ or } n \in \left\{i,j\right\}, \\ \left(M+t, X'' \cup \left\{(i,j)\right\}\right) & \text{otherwise, where} \end{cases} \\ t &:= t' + t'', \\ \left(t'', X''\right) &:= R\left((i,k), X'\right), \\ \left(t', X'\right) &:= R\left((k,j), X\right), \\ k &:= \max\left\{i,j\right\} + 1, \end{aligned} \tag{2.4}$$

and define (over suitable domain) time $:= (x,y) \mapsto x$, $T := \text{time} \circ R$, then for some such M the maximum number of steps in which we can compute $\widehat{\Phi}_C\left((i,j),X\right)$ equals to $T\left((i,j), \text{time}\,(X)\right)$ for any pair of cities. We now want to show that

$$T\left((1,1),\varnothing\right) = \mathcal{O}\left(n^2\right).$$

Let us expand

$$\begin{aligned} T\left((1,1),\varnothing\right) &= M + T\left((1,2), X^{1,1}_{1,2}\right) + T\left((2,1), X^{1,1}_{2,1}\right) = \\ &= M + 2M + T\left((1,3), X^{1,2}_{1,3}\right) + T\left((3,2), X^{1,2}_{3,2}\right) + \\ &\quad + T\left((3,1), X^{2,1}_{3,1}\right) + T\left((2,3), X^{2,1}_{2,3}\right) = \\ &= M + 2M + 2M \cdot 2 + \\ &\quad + T\left((1,4), X^{1,3}_{1,4}\right) + T\left((4,3), X^{1,3}_{4,3}\right) \\ &\quad + T\left((3,4), X^{3,2}_{3,4}\right) + T\left((4,2), X^{3,2}_{4,2}\right) \\ &\quad + T\left((3,4), X^{3,1}_{3,4}\right) + T\left((4,1), X^{3,1}_{4,1}\right) \\ &\quad + T\left((2,4), X^{2,3}_{2,4}\right) + T\left((4,3), X^{2,3}_{4,3}\right) = \ldots \end{aligned}$$

where the sets $X_{i,j}^{k,l}$ correspond to evaluation of R as defined — which still can be seen as depth-first search over the recursion tree of Φ_C. Here we make a critical observation: any two nodes in the recursion tree given by eq. (2.2) have a common ancestor, so at any point in our expansion, given a pair (i, j), it must (by construction of R) lie in all but the very last of $X_{i,j}^{k,l}$, hence $T\left((i,j), X_{i,j}^{k,l}\right) = M$ for all but last such contributor to the sum. We thus arrive at

$$T\left((1,1),\varnothing\right) \leq M + 2M\left(1+2+\cdots+(n-3)\right) + 2M\left(1+2+\cdots+(n-2)\right) + \\ + \sum_{i=1}^{n-1}\left(T\left((i,n), X'_{i,n}\right) + T\left((n,i), X'_{n,i}\right)\right) = \mathcal{O}\left(n^2\right)$$

per $T\left((n,i),.\right) = T\left((i,n),.\right) = M\ \forall i$ via first case in (2.4). □

This illustrates how, given a recurrence relation, under certain circumstances (efficiently computable edges) memoization technique can be used to achieve time complexity which is asymptotically no worse than size of the recursion tree — which we think can be seen as key concept behind dynamic programming.

In computing, memoization can be applied to any function values of which depend only on its arguments — such a function is also said to be *referentially transparent* (in mathematics, every well-defined map is referentially transparent; in computing, depending on programming language, the concept of function may allow for implicit dependency on environment — i.e. some mutable state).

Remark 2.2. The solution presented in proposition 2.3 incurs quadratic space overhead. We would like to mention here that linear space complexity is achievable if we rewrite said solution as a so called *tail-recursive* function. We shall leave out the details at this point, only noting that this is equivalent to breadth-first search over the recursion tree.

It should also be noted that the tree size can be halved (as shown in fig. 2.4) if we restrict ourselves to symmetric matrices in pyramidal TSP.

Remark 2.3. While determining if a given TSP instance is pyramidal is NP-hard, a number of polynomially testable classes which possess pyramidal solutions exists [1].

Before we turn our attention to the next neighbourhood, there is one last point that we feel needs to be addressed: while we have discussed depth-first search and mentioned breadth-first search computation over the recursion tree, we have not yet considered the bottom-up approach often used in imperative programming.

While its merits and caveats are manifold, there is one important requirement said technique impedes on the problem: for it to work, terminal nodes (or leafs) of the recursion tree must be feasibly computable beforehand. While this presents no problem for pyramidal TSP, it is much less the case for the neighbourhood we are about to see — which made bottom-up approach ultimately unsuitable for this project.

2.3. Strongly balanced tours

Throughout remainder of this section we shall restrict our considerations, unless noted otherwise, to symmetric TSP only — in particular, we can consider paths to be equivalent

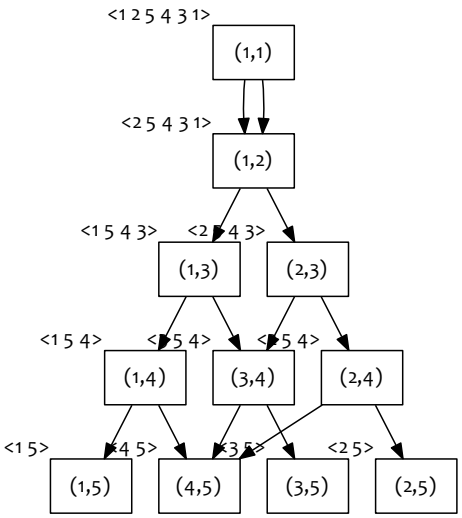


Figure 2.4: A sample trace of our implementation of pyramidal sTSP ($n = 5$).

whenever they are equal under path reversal.

Definition 2.2 (strongly balanced tours)**.** Given a set of cities $\mathcal{N}_n$, we define, inductively, for every $m \in \mathcal{N}_n$,

$$\begin{aligned}\mathcal{B}_1 &:= \left\{\{(1)\}\right\},\\ \mathcal{B}_m &:= \bigcup_{\tau\in\mathcal{B}_{m-1}} \left(\operatorname{add}(\tau, m) \cup \operatorname{append}(\tau, m) \cup \operatorname{merge}(\tau, m)\right),\end{aligned}$$

where for $\tau = \left\{\pi_1, \pi_2, \ldots\right\}$ with $\pi_i = \left(m_i, \ldots, m_i'\right)$, $m_i \le m_i'\ \forall i$, and $m_1 < m_2 < \ldots$, we define the three operations above as

$$\begin{aligned}\operatorname{add}(\tau, m) &:= \tau \cup \{(m)\},\\ \operatorname{append}(\tau, m) &:= \left(\tau \setminus \pi_1\right) \cup \left\{\operatorname{rev}\left(\pi_1\right) \oplus (m)\right\},\\ \operatorname{merge}(\tau, m) &:= \begin{cases}\varnothing, & |\tau| < 2,\\ \left(\tau \setminus \left\{\pi_1, \pi_2\right\}\right) \cup \left\{\operatorname{rev}\left(\pi_1\right) \oplus (m) \oplus \pi_2\right\} & otherwise.\end{cases}\end{aligned}$$

For a $\tau \in \mathcal{B}_n$ with $|\tau| = 1$, we then call closure of its single element a *strongly balanced tour.*

Above construction is best visualized per analogy to argument which led us to proposition 2.1: while in latter, we were appending city at a time to a partially constructed tour which consisted of two (or one, if we joined them at city 1) paths, here such a

partially constructed tour (such as $\tau \in \mathcal{B}_{m-1}$ above) can consist of multiple paths, and for each city m we can choose from two to three placement possibilities (according to add, append, merge) as illustrated in fig. 2.5.

We can also now see that per above definition, every element of $\mathcal{B}_m$ contains simple disjoint paths over $\mathcal{P}_m$ which together contain all cities in $\mathcal{N}_m$ (and $m_i = m_i'$ above is only possible where $\pi_i = (m_i)$), thus making following evident:

Remark 2.4. Strongly balanced tours, as introduced in definition 2.2, are in fact elements of $\mathcal{T}_n$.

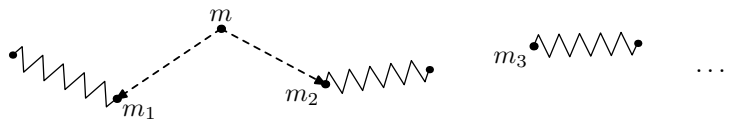


Figure 2.5: Placement options for a city m during construction of a strongly balanced tour: we can choose m to be adjacent to m_1 (via append), both m_1 and m_2 (via merge), or none of the two (add).

Though the above construction may seem somewhat contrived, elements thereby created have roots in combinatorial optimization — they represent a variation on so called *balanced tours*, which contain a solution to (shown to be NP-hard) *Relaxed Supnick TSP* (*RS-TSP*) (see [5, 4]). And while strongly balanced tours may lack a nice visual description comparable to that of pyramidal tours, the neighbourhood they represent is certainly no less interesting.

Definition 2.3. For $M \in \mathbb{N}$, we define SBal_n^M to be the set of all strongly balanced tours obtained by restricting each $\mathcal{B}_m$ in definition 2.2 to contain only elements of size no higher than M. We shall call the TSP restricted to said set *strongly balanced TSP* (with *maximum node size* M).

Remark 2.5. For any $M \geq 2$, at any step of city-at-a-time construction of a strongly balanced tour we have a choice between at least two of the three placement options shown above, which all yield essentially different tours (recall fig. 2.5). This shows that

$$\left|\mathrm{SBal}_n^M\right| = \Omega\left(2^n\right).$$

Also, with $M \geq 1$ the append option is always available, hence for every M SBal_n^M induces a neighbourhood in $\mathcal{T}_n$ with centre $(\ldots, 5, 3, 1, 2, 4, \ldots)$.

Recall how we could uniquely identify a partially constructed pyramidal tour using only a pair of cities (i, j) — which we then used to describe, in proposition 2.1, elements which would complete such partially constructed tours to elements of $\mathcal{T}_n$.

Equivalently, a partially constructed strongly balanced tour (i.e. element of $\mathcal{B}_m$) can be described by the end points of its elements and the set of cities it contains (which is necessarily $\{1, \ldots, m\}$).

Proposition 2.5 (recursive structure of SBal_n^M). *Given $n \in \mathbb{N}, m \in \mathcal{N}_n$, $M \in \mathcal{N}_n$ and*

$$N = \left\{ \left\{a_i, b_i\right\} \mid i = 1, \dots, k \right\}, k \in \mathcal{N}_M \cup \{0\},$$

with $a_1 < a_2 < \dots < a_k$, $a_i \leq b_i \ \forall i = 1, \dots, k$, define $W(m, N)$ to be the set of all sets (none higher than M in size) of simple paths through all of the cities $\{m+1, \dots, n\}$ as well as all cities contained in (elements of) N, which per path concatenation (as defined in definition 1.2) complement the set of non-singleton elements of N seen as paths (a_i, b_i) to a single cycle, therefore also complementing corresponding (to N) element of $\mathcal{B}_m$ to a strongly balanced tour. Then

$$\mathrm{SBal}_n^M = W(0, \varnothing) = W\left(1, \{1\}\right)$$

(as quotients under rotation equivalence shown in proposition 1.1) and

$$W(m, N) = \begin{cases} \left\{(a_1, b_1)\right\}, & m = n, k = 1, \\ \varnothing, & m = n, k \neq 1 \\ W\left(m+1, \mathrm{add}'(N, m+1)\right) \cup & \\ \qquad \cup W\left(m+1, \mathrm{append}'(N, m+1)\right) \cup & \\ \qquad \cup W\left(m+1, \mathrm{merge}'(N, m+1)\right) & \text{otherwise,} \end{cases} \tag{2.5}$$

where

$$\begin{aligned} \mathrm{add}'(N, m) &\coloneqq N \cup \{m\} && (\text{or } \varnothing \text{ for } |N| \geq M), \\ \mathrm{append}'(N, m) &\coloneqq \left(N \setminus \left\{\left\{a_1, b_1\right\}\right\}\right) \cup \left\{\left\{m, b_1\right\}\right\} && (\text{or } \varnothing \text{ for } N = \varnothing), \\ \mathrm{merge}'(N, m) &\coloneqq \left(N \setminus \left\{\left\{a_1, b_1\right\}, \left\{a_2, b_2\right\}\right\}\right) \cup \left\{\left\{b_1, b_2\right\}\right\} && (\text{or } \varnothing \text{ for } |N| < 2). \end{aligned}$$

Proof. Follows from definition 2.2 per construction. □

This defines a recurrence relation to which we can apply same memoized graph traversing technique we demonstrated on pyramidal TSP (in fact, apart from description of recursion tree, counting tour reconstruction rules, our implementation uses same code for both) and compute, for each node (m, N), a minimum total cost element in $W(m, N)$ — and thus a minimum cost tour in SBal_n^M. Sample trace of such computation can be seen in fig. 2.6. Exact tour reconstruction rules can be deduced directly from above relation and are left as exercise for the reader (they amount to inserting into the solution, with possible reversal and joining, paths (m, a_1) or (a_1, m, a_2)).

Example 2.1. The construction above is best illustrated by example: shown in table 2.1 is one path taken from the root of the recursion tree, along with the strongly balanced tour corresponding to said path as well as sample optimal solution from $W(m, N)$ as it would be constructed returning along the path.

Regarding time complexity, we need to address concerns similar to those raised in remark 2.1:

$\tau \in \mathcal{B}_m \sim (m, N)$	m	N	optimum in $W(m, N)$	path from parent
$\{(1)\}$	1	$\{\{1\}\}$	$\{(1,4,6,2,5,7,3,1)\}$	
$\{(1),(2)\}$	2	$\{\{1\},\{2\}\}$	$\{(1,4,6,2,5,7,3,1)\}$	add′
$\{(1,3),(2)\}$	3	$\{\{1,3\},\{2\}\}$	$\{(1,4,6,2,5,7,3)\}$	append′
$\{(2),(3,1,4)\}$	4	$\{\{2\},\{3,4\}\}$	$\{(3,7,5,2,6,4)\}$	append′
$\{(2,5),(3,1,4)\}$	5	$\{\{2,5\},\{3,4\}\}$	$\{(3,7,5),(2,6,4)\}$	append′
$\{(3,1,4),(5,2,6)\}$	6	$\{\{3,4\},\{5,6\}\}$	$\{(3,7,5),(4,6)\}$	append′
$\{(4,1,3,7,5,2,6)\}$	7	$\{\{4,6\}\}$	$\{(4,6)\}$	merge′

Table 2.1: Sample path taken in the strongly balanced recursion tree ($n = 7$, $M \geq 2$) along with accordingly computed optimum tour. Elements of $\mathcal{B}_m$ corresponding to the nodes are given for illustration purposes.

(i) ad tour reconstruction: even if we assume $|N| \leq M = \mathcal{O}(1)$, potentially reversing and joining paths in this step cannot be considered a constant time operation, but still can reasonably be assumed to contribute to time complexity a polynomial factor only (note that we only need to reconstruct the optimal tour);

(ii) ad memory access: while enumeration of *node types* (introduced below) yields a polynomial time collision-free solution to this, in practice we were able to achieve best results with polynomial hash functions such as

$$(m, N) \mapsto mX^{2k+1} + \sum_{i=1}^{k} \left((m - a_i)X^{2k-2i+1} + (m - b_i)X^{2k-2i} \right),$$
$$X = \lceil \log_2 n \rceil \quad (N \text{ as above}).$$

Corollary 2.6. *Strongly balanced TSP (with maximum node size M) is polynomially solvable.*

Proof. Note that with $|N| \leq M$, each of add′, append′, merge′ can be considered constant time operation. Seeing how size of recursion tree corresponding to eq. (2.5) necessarily lies in $\mathcal{O}\left(n^{2M+1}\right)$, memoized computation technique as seen in proof of corollary 2.4 then yields the result. □

2.4. Strongly balanced tours and linear time conjecture

It is not impossible that strongly balanced recursion tree as presented here grows in size only linearly with n — which, not counting tour reconstruction, would make time complexity of our solution to strongly balanced TSP linear as well.

To elaborate on this, we consider the map

$$\chi : (m, N) \mapsto \left\{ \{m - a_i, m - b_i\} \mid i = 1, \ldots, k \right\},$$

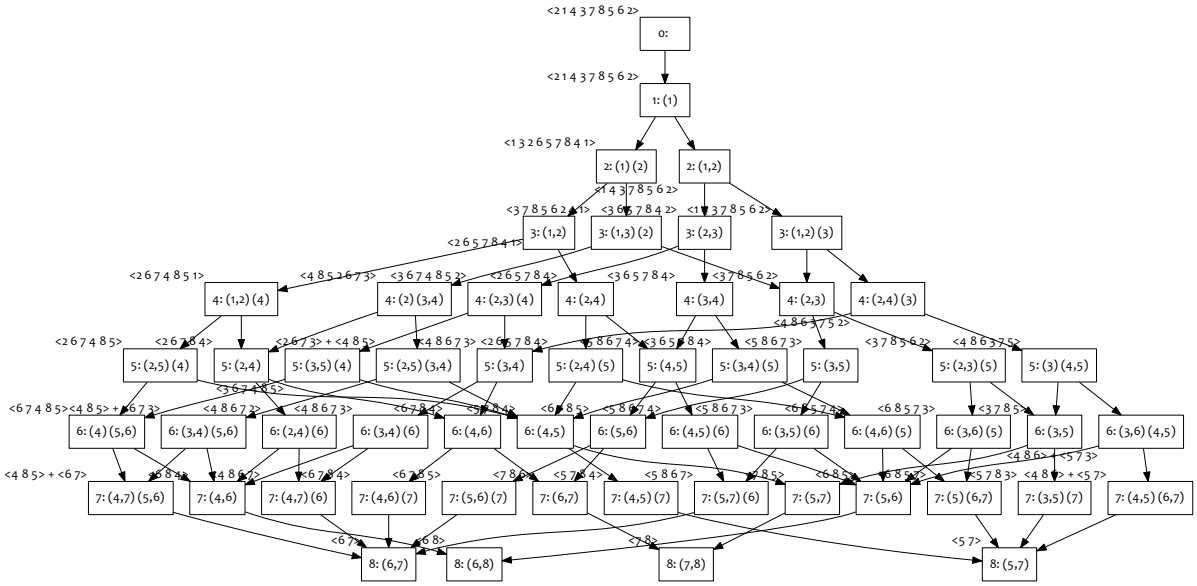


Figure 2.6: Strongly balanced TSP recursion tree ($M = 2, n = 8$).

where (m, N) is a node of the recursion tree as defined in proposition 2.5 (for such a node, we call $\chi(m, N)$ its *type*). We then recognize following crucial points:

- the correspondence between (m, N) and $(m, \chi\left(m, N\right))$ is trivial; hence, if the number of different node types in the above recursion tree happens to be bounded from above by some constant K, then total number of nodes in the tree is bounded by $nK = \mathcal{O}(n)$;

- while descending the recursion tree, we are adding cities in increasing order; therefore (as is quickly verified) each of $\chi \circ \text{add}'$, $\chi \circ \text{append}'$, $\chi \circ \text{merge}'$, evaluated at (N, m), can be written in terms of $\chi(N)$ only and hence does not depend on m;

- ergo, if, for some node size bound M, after adding next city in the construction of the tree in question, the number of different node types did not increase, then the set of node types has reached a fixed point relative to expansion through all of $\text{add}', \text{append}', \text{merge}'$, i.e. above conjecture holds true for given M.

While we are not aware of a general proof thereof, we have been able to confirm said conjecture (having computed the number of different node types, shown in table 2.2) for values of M up to 6 — and while this confirms linear tree size for given values of M, the growth behaviour of the constant factor does seem intimidating, reflecting the runtime behaviour we encountered in practice.

M	number of node types	m
1	3	2
2	16	7
3	121	13
4	1074	20
5	10 387	28
6	107 176	37

Table 2.2: Computed maximum number of node types for values of node size limit M up to 6, along with the number of cities m at which the fixed point was reached.

2.5. Local search and flowers

To conclude the discussion of theory behind our implementation, it should be noted that none of the two neighbourhoods as presented above achieve appreciable results as local search heuristic — to improve on that, we introduce the simple concept of flowers:

Definition 2.4. For $T \subseteq \mathcal{T}_n, S \subseteq \mathcal{S}_n$ define *flower over T by S* to be

$$\mathcal{F}(S,T) \coloneqq \left\{ s\tau \mid s \in S, \tau \in T \right\} \cup T$$

(compare this to definition 1.10).

TSP over such set is then computable via choosing best from no more than $|S| + 1$ local search results. Also note that whenever T induces a neighbourhood, flower over T induces one with same centre.

While, as seen in [3], a good extension of pyramidal local search can be achieved via

$$\mathcal{F}\left(\langle (1\ 2\ \cdots\ n) \rangle, \mathrm{Pyr}_n\right),$$

for strongly balanced tours such an extension is not so easy to find, and is left to be seen as an open question — in our tests, we have not been able to significantly improve on[1]

$$\mathcal{F}\left(\mu \langle (1\ 2\ \cdots\ n) \rangle \mu^{-1}, \mathrm{SBal}_{\mathrm{n}}^{\mathrm{M}}\right) \quad \text{with} \quad \mu = \left(\ldots, 5, 3, 1, 2, 4, \ldots\right)$$

in this regard.

In our implementation we also refer to permutations from the set S above as *rotations* of T. We also define therein a version of iterated local search which we call *adaptive*, wherein we increase the number of considered rotations whenever a stale iteration is reached.

Now we can turn our attention to technical details of our implementation as well as practical test results.

[1] In our tests, we used μ alongside an "interleaved" version thereof — see our code for exact rotations set.

3. Implementation

> You're bound to be unhappy if you optimize everything.
>
> *(Donald Knuth)*

3.1. Key features

Our choice of programming language fell on Standard ML, which we feel strikes the right (for this project) balance between expressiveness and convenience and, while being not too low level a language, can still offer effective execution speed.

The core functionality contained in about 3000 lines of code requires only a Standard ML environment which implements the `smlnj-lib` (as provided by the widely available `MLton` compiler or `SML/NJ`) to work[1] and can be used

- as standalone executable (compiles via `MLton` or `Poly/ML`);
- from a REPL environment (such as `SML/NJ` or `Poly/ML`) as shown in Appendix A;
- via shared library interface (built via `MLton`) as shown in Appendix B.

All code written for this project (along with precompiled `MinGW/x86` build) is comprehensively documented and available, along with latest version of this document, from `https://bitbucket.org/mad_hatter/rstsp/`. Bug reports and pull requests are welcome.

3.2. Runtime behaviour

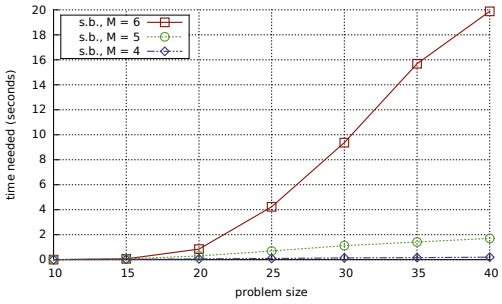
Measured single local search running times, shown in fig. 3.1, further confirm our expectations:

- for a fixed maximum node size, strongly balanced search yields runtime behaviour which looks pronouncedly linear, with a heavy constant factor which increases approximately tenfold along with by-one increase in node size limit (see figs. 3.1a to 3.1c);
- pyramidal search, as can be expected of its quadratic complexity nature, exhibits a decidedly lower growth rate initially, but eventually falls behind strongly balanced local search — as can be seen in figs. 3.1d and 3.1e.

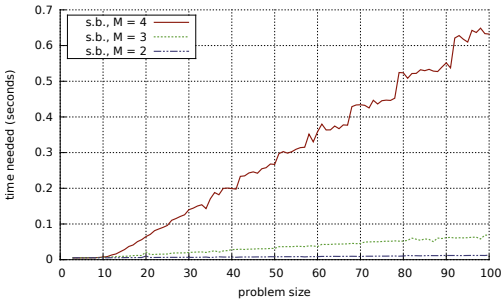
We would expect breadth-first search or dedicated (possibly machine-generated) implementations of above heuristics to bring notable improvement in this regard.

We did not systematically measure memory consumption behaviour, as running time was the bottleneck factor in our tests.

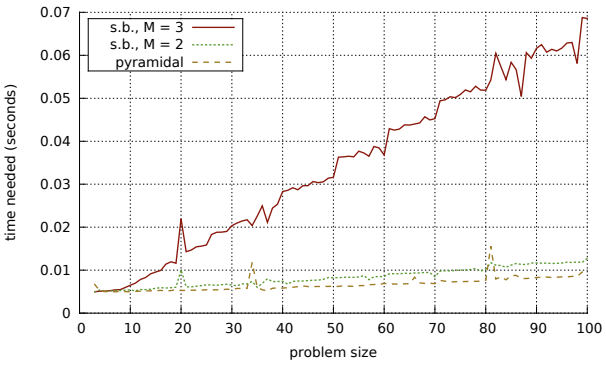
[1] The housekeeping environment – i.e. build, testing, benchmarking and plotting scripts – uses a wider set of tools all of which are easily obtainable in common `GNU/Linux` distributions today.



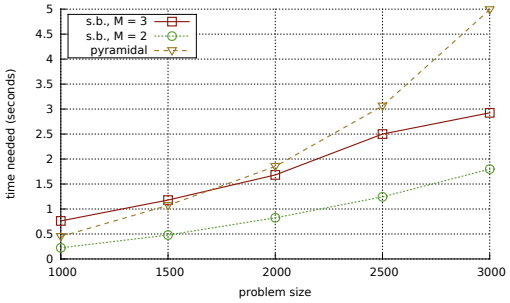
(a) strongly balanced, $M = 4, 5, 6$



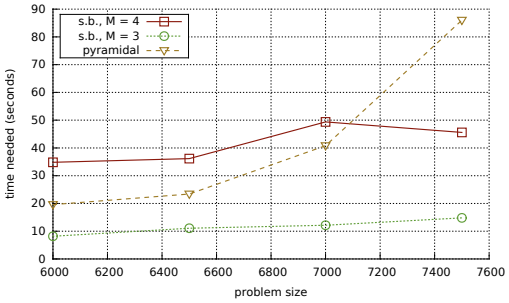
(b) strongly balanced, $M = 2, 3, 4$



(c) pyramidal and str. balanced with $M = 2, 3$



(d) pyramidal falls behind str. balanced ($M = 3$)



(e) pyramidal falls behind str. balanced ($M = 4$)

Figure 3.1: Measured local search running time. Results presented here have been produced on a machine running Linux 3.10.17 (x86_64) @ i5-2520M and 8GB of RAM.

Another point we feel needs to be mentioned is that for now, we do not compute rotations as a set, rather simply iterating given enumerating map — which we have to consider when evaluating running time measurements in ILS.

3.3. Tour benchmarks

Finally, we present, in fig. 3.2,

- computed tour cost over randomly generated symmetric instances where each matrix entry is sampled independently from (pseudorandom) uniform distribution on same fixed interval;
- computed tour cost (as quotient to optimum, named *tour quality* below) and running time (number of iterations as well as real time needed) of ILS over a set of TSPLIB instances (sample size 28, instance size under 130, many of them euclidean).

From what little testing this represents, we can reasonably conclude that

- prior to flower extension, none of the two neighbourhoods deliver interesting results in general case;
- pyramidal tours, thus extended, yield a viable local search heuristic for the TSP;
- strongly balanced tours not so readily do so; it remains to be seen if a good extension (e.g. via a flower) or a special case where this neighbourhood yields better results exist.

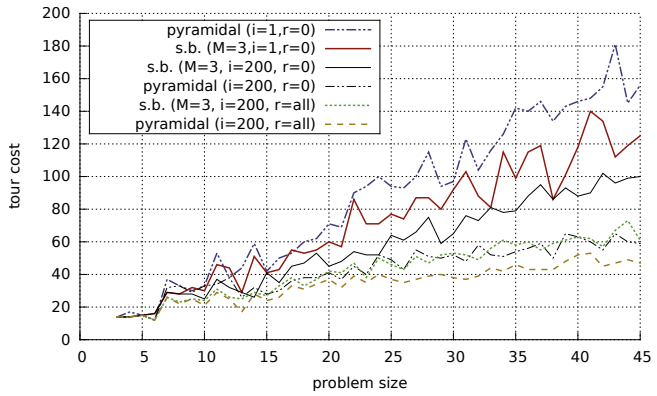
4. Conclusions

> Enough research will tend to support your conclusions.
>
> *(Arthur Bloch)*

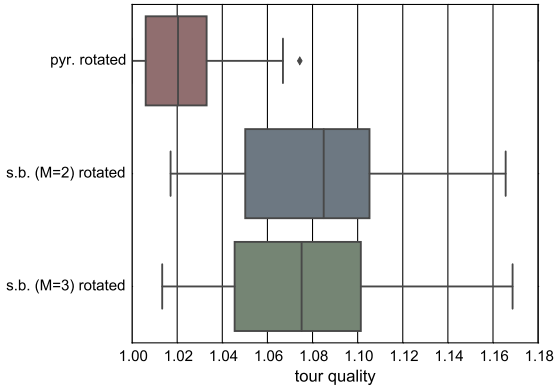
Having described extension of dynamic programming solutions to special cases of traveling salesman problem from neighbourhoods to iterative heuristics, we have provided for two such neighbourhoods a working implementation which is general enough to be readily extendible to other solutions and efficient enough for study of such solutions or combinations[1] of corresponding heuristics.

For the exponential neighbourhood of strongly balanced tours this represents first practical results; in the process of implementing the solution, we have further confirmed the linear time conjecture for select few node size limits.

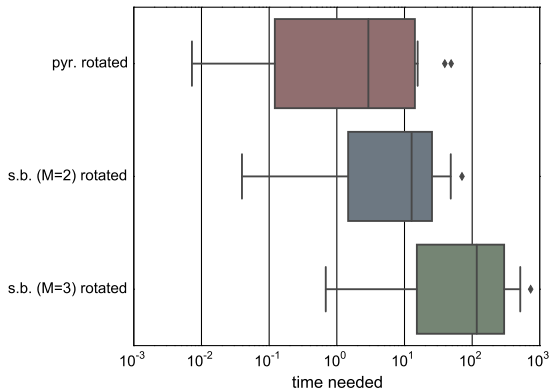
[1] Also possible with the standalone executable, which provides the option of choosing the starting tour (which is automatically mapped to given neighbourhood's centre).



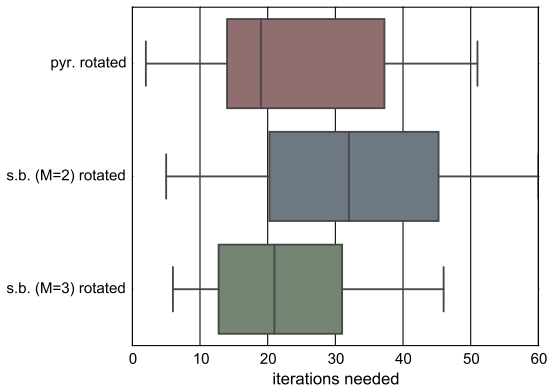
(a) tour cost comparison over random instances: here, r denotes maximum number of rotations considered, i — maximum number of iterations in ILS



(b) ILS over 28 TSPLIB instances: achieved tour quality



(c) TSPLIB instances: measured runtime



(d) TSPLIB instances: convergence speed

Figure 3.2: Benchmark results.

4.1. Areas for further research

We shall wrap up our discussion by identifying here briefly just some of tasks and areas which one could, given time, explore in more detail:

- precompute flowers as sets to guarantee rotations' uniqueness;
- implement breadth-first search over recursion tree;
- study more flowers (e.g. dependent on node size limit) over strongly balanced tours;
- implement the interface to and test our neighbourhoods on further metrics and TSP instances — such as asymmetric ones for pyramidal search;
- generate and benchmark RS-TSP instances;
- benchmark higher node size limits (in strongly balanced search) as well as higher sized instances;
- study in more detail observed convergence behaviour of ILS heuristics;
- provide for possibility of branch cutting in local search;
- design and implement dedicated bottom-up code generation for s.b. search;
- explore concurrent computation models such as *tuple spaces* or lightweight (a)synchronous message passing frameworks and their applicability to local search as seen here — naïve parallelization does not seem to apply well to the recursion trees we presented;
- …

This concludes our report.

Appendices

Appendix A Sample REPL Session

Listing 1: Sample SML code as it could be used in a REPL session.

```
(**
 * Under SML/NJ, issue
 *   use "./rstsp/rstsp-smlnj.sml";
 * first.
 * Under Poly/ML:
 *   use "./rstsp/rstsp-polyml.sml";
 * .
 *)

open Utils

structure NSrch : SEARCHES = DefaultSearches(IntNum)
datatype tsplib_inst = datatype TsplibReader.tsplib_inst

val node_size = SOME 0w3
val iter_limit = SOME (IntInf.fromInt 10)
val stale_thresh = SOME (IntInf.fromInt 2)
val rotations = SOME 0w24
val max_flips = SOME (IntInf.fromInt 20)
val options = (max_flips,
               SOME (IntInf.fromInt 1),
               (iter_limit, stale_thresh, ()),
               (iter_limit, stale_thresh, 0w0, rotations, node_size))
structure Search = NSrch.FlipFlopSearch

structure Dist = NatDist
val inst = TsplibReader.readTSPFile "../../test/data/tsplib/gr17.tsp"
val data = case inst of
             EXPLICIT_INSTANCE v => v
           | EUCLIDEAN_2D_INSTANCE (xs,ys) => raise Fail "not here"
           | EUCLIDEAN_2D_CEIL_INSTANCE (xs,ys) => raise Fail "not here"
structure CostCheck = TSPCostFn(Dist)

fun main () =
let
  val dist = Dist.getDist data
  val size = Dist.getDim data
  val search = Search.search size dist NONE true options
  val (sol', stats) = search ()
  val (sol_len, sol_fn) = valOf sol'
  val sol = sol_fn ()
  val _ = print ("*********************************\n")
  val _ = print ("  Solution:  " ^ (Search.tourToString sol) ^ "\n")
  val _ = case stats of
            NONE => ()
          | SOME (nn, nk, hs) => (
```

```
            print (" Node Types:  " ^ (wordToString nk) ^ "\n");
            print (" Store size:  " ^ (wordToString nn) ^ "\n");
            print ("Node hashes:  " ^ (wordToString hs) ^ "\n")
          )
  val _ =
    let
      val sol_vec = Search.tourToVector sol
      val len_val = Dist.Num.compare(sol_len, CostCheck.tourCost data sol_vec) =
          EQUAL
      val sol_val = TSPUtils.validTour size sol_vec
    in
      print ("  Solution valid:  " ^ (if sol_val andalso len_val then "yes" else
          "NO!") ^ "\n")
    end
  val _ = print ("  Tour cost:  " ^ (Dist.Num.toString sol_len) ^ "\n")
  val _ = print ("*********************************\n")
in
  ()
end

val _ = TextIO.print;
val _ = main ()
```

Appendix B Shared Library Interface

Listing 2: Shown here is code using our implementation via shared library interface — which can be used to access its functionality from any programming environment equipped with a foreign function interface.

```
#include <stdlib.h>
#include <stdio.h>
#include <inttypes.h>
#include "rstsp.h"

#ifndef PRIu32
#  define PRIu32 "u"
#endif

#ifndef PRIi64
#  if __WORDSIZE == 64
#    define PRIi64 "li"
#  else
#    define PRIi64 "lli"
#  endif
#endif

int64_t dst(uint32_t i, uint32_t j) {
  return i+j;
}

void print_tour(uint32_t *tour, uint32_t size) {
```

```
  int i;
  for(i=0; i<size+1; i++) {
    printf("%" PRIu32, tour[i]+1);
    if(i<size) printf(" ");
  }
  printf("\n");
}

int main(int argc, const char **argv) {

  /**
   * Yes, this is important.
   */
  rstsp_open(argc, argv);

  uint32_t prob_size = 10;

  /**
   * We do not want want to deal with structure alignment for different
   * platforms, hence a call to search function shall yield:
   *   – a pointer to tour cost (int64);
   *   – a pointer to tour array (zero–based, closed cycle, word32)
   *   all packed in a pointer array.
   */
  Pointer *result;

  /**
   * We also want to avoid having to manually manage memory, therefore only
   * basic searches —— not their combinators —— are exposed through the library.
   *
   * This computes optimal pyramidal tour, writing a dot trace if so desired.
   */
  char *dotfilename = NULL;
  int64_t *cost;
  uint32_t *tour;
  result = (Pointer *)rstsp_pyr_search(prob_size, *dst, dotfilename);
  if(result) {
    cost = (int64_t *)result[0];
    tour = (uint32_t *)result[1];
    printf("  > Pyramidal tour: ");
    print_tour(tour, prob_size);
    printf("  > Tour cost: %" PRIi64 "\n", *cost);
    free(tour);
    free(cost);
    free(result);
  }

  /**
   * Optimal strongly balanced tour;  max_width : node size limit.
   */
  uint32_t max_width = 3;
  result = (Pointer *)rstsp_sb_search(prob_size, *dst, max_width, dotfilename);
  if(result) {
    cost = (int64_t *)result[0];
```

```
  tour = (uint32_t *)result[1];
  printf("  > SB tour: ");
  print_tour(tour, prob_size);
  printf("  > Tour cost: %" PRIi64 "\n", *cost);
  free(tour);
  free(cost);
  free(result);
}

/**
 * Iterative pyramidal search which considers up to ((n+1) div 2) rotations in each
 * iteration.
 */
uint32_t max_iters = 10;
uint32_t stale_iters = 3;
uint32_t max_rots = prob_size-1;
result = (Pointer *)rstsp_iter_pyr_search(prob_size, *dst, max_iters, stale_iters,
    max_rots);
if(result) {
  cost = (int64_t *)result[0];
  tour = (uint32_t *)result[1];
  printf("  > Pyramidal/iter/rot tour: ");
  print_tour(tour, prob_size);
  printf("  > Tour cost: %" PRIi64 "\n", *cost);
  free(tour);
  free(cost);
  free(result);
}

/**
 * Iterative strongly balanced search, flower size 2*((n+1) div 2).
 */
max_rots = 2*prob_size;
result = (Pointer *)rstsp_iter_sb_search(prob_size, *dst, max_width, max_iters,
    stale_iters, max_rots);
if(result) {
  cost = (int64_t *)result[0];
  tour = (uint32_t *)result[1];
  printf("  > SB/iter/rot tour: ");
  print_tour(tour, prob_size);
  printf("  > Tour cost: %" PRIi64 "\n", *cost);
  free(tour);
  free(cost);
  free(result);
}

/**
 *  A variation of the above where number of considered rotations grows at
 *  stale iterations (which we call adaptive).
 */
uint32_t min_rots = 0;
result = (Pointer *)rstsp_ad_sb_search(prob_size, *dst, max_width, max_iters,
    stale_iters, min_rots, max_rots);
if(result) {
```

```
    cost = (int64_t *)result[0];
    tour = (uint32_t *)result[1];
    printf("  > SB/adaptive tour: ");
    print_tour(tour, prob_size);
    printf("  > Tour cost: %" PRIi64 "\n", *cost);
    free(tour);
    free(cost);
    free(result);
  }

  /**
   *  A variant combining, in alternating, or flipflop, manner, adaptive s.b. &
   *  iterative pyramidal searches.
   */
  uint32_t max_flips = 0;
  result = (Pointer *)rstsp_ff_search(prob_size, *dst, max_width, max_iters,
      stale_iters, min_rots, max_rots, max_flips);
  if(result) {
    cost = (int64_t *)result[0];
    tour = (uint32_t *)result[1];
    printf("  > SB/flipflop tour: ");
    print_tour(tour, prob_size);
    printf("  > Tour cost: %" PRIi64 "\n", *cost);
    free(tour);
    free(cost);
    free(result);
  }

  /**
   * Since SML offers garbage collection, the library should be clean from leaks in
   * userspace.  Please note that valgrind has issues with mlton's memory management
   * and reports lots of invalid memory accesses.  These messages should be harmless
       __
   * we might one day also supply a valgrind whitelist.
   */
  rstsp_close();

  return 0;

}
```

References

[1] Fazle Baki and Santosh N. Kabadi. «Pyramidal traveling salesman problem». In: *Computers & Operations Research* 26.4 (1999), pp. 353–369.

[2] Rainer E. Burkard et al. «Well-Solvable Special Cases of the Traveling Salesman Problem: A Survey». In: *SIAM Rev.* 40.3 (Sept. 1998), pp. 496–546. ISSN: 0036-1445. DOI: 10.1137/S0036144596297514. URL: http://dx.doi.org/10.1137/S0036144596297514.

[3] J. Carlier and P. Villon. «A New Heuristic for the Traveling Salesman Problem». In: *Revue française d'automatique, d'informatique et de recherche opérationnelle* (1990). URL: http://www.numdam.org/item?id=RO_1990__24_3_245_0.

[4] Vladimir G. Deineko, Bettina Klinz, and Gerhard J. Woeginger. «Four Point Conditions and Exponential Neighborhoods for Symmetric TSP». In: *SODA* (2006), pp. 544–553.

[5] Vladimir G. Deineko et al. «Four-point conditions for the TSP: The complete complexity classification». In: *Discrete Optimization* 14 (2014), pp. 147–159.

[6] P. C. Gilmore, Eugene L. Lawler, and D. B. Shmoys. *Well-solved Special Cases of the Traveling Salesman Problem.* Tech. rep. UCB/CSD-84-208. EECS Department, University of California, Berkeley, 1984. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/1984/5922.html.

[7] Richard M. Karp. «Reducibility among combinatorial problems». In: *Complexity of Computer Computations.* Ed. by R. E. Miller and J. W. Thatcher. Plenum Press, 1972, pp. 85–104.