**Вопросы к экзамену по курсу “ЛИПО САПР” 4 семестр (часть 1).**

1. **Шаблоны функций. Определение. Пример. Использование для стандартных и пользовательских типов.**

Параметрический полиморфизм позволяет многократно использовать один и тот же код применительно к разным типам данных – **тип указывается как параметр функции или класса.**

При создании функций иногда возникают ситуации, когда две функции выполняют одинаковую обработку, но работают с разными типами данных. С помощью механизма перегрузки функций можно использовать одно и то же имя для функций, выполняющих разные действия и имеющих разные типы параметров. **Шаблон определяет набор операторов, с помощью которых программы позже могут сами создать несколько функций.**

**Шаблоны –** средство для реализации параметризированных классов функций на языке С++.

Template <class TYPE>

Void swap (TYPE &x, TYPE &y)

{

TYPE temp;

temp = x;

x = y;

y = temp;

}

**Инстанцирование** – генерация функции по шаблону и её аргументу.

Может быть и два типа (TYPE1, TYPE2), или параметр может быть стандартного типа (int)

Определяя шаблоны функций, вы **заставляете компилятор С++ создавать** в случае необходимости функции, которые отличаются типом возвращаемого значения или типами параметров.

1. **Шаблоны классов. Определение, пример.**

Для создания объектов класса с использованием шаблоном программы просто ссылаются на имя класса, за которым внутри угловых скобок следуют типы, например, <int, float>

Если у класса есть конструктор, с помощью которого вы инициализируете элементы данных, вы можете вызвать этот конструктор при создании объекта с использованием шаблона.

Если компилятор С++ встречает объявление объекта, он создаёт класс из шаблона, используя соответствующие типы.

Шаблон класса определяет типонезависимый класс, который в дальнейшем служит для создания объектов требуемых типов. Если компилятор С++ встречает объявление объекта, основанное на шаблоне классе, то **для построения класса требуемого типа он будет использовать типы, указанные при объявлении**.

Template <class T, class T1>

class array

{T \*data;

int size;

int index;

public:

array (int size);

T1 sum (void);

T average\_value (void);

void show\_array (void);

int add\_value (T); };

Сразу после имени класса вы должны указать типы класса, например: array <T, T1>::avg

Создание класса требуемого типа array<int, long> Kl1;

Объявление класса:

Template\_class\_name<type1, type2> object\_name (parameter1, parameter2)

Массив типа char, в котором 100 элементов:

Array<char, int> numbers (100)

**Использование не типизированных параметров**

template <typename T, int n>

array <int, 100> S; создаст класс array с именем S, состоящий из 100 элементов типа int.

**Свойства шаблона класса**

Методы шаблона класса автоматически становятся шаблонами функций.

1. Описание параметров шаблона в заголовке функции должно соответствовать шаблону класса.
2. Локальные класса не могут иметь шаблоны в качестве своих элементов.
3. **Шаблоны методов не могут быть виртуальными**.
4. Шаблоны классов могут содержать статические элементы, дружественные функции и классы.
5. Шаблоны могут быть производными как от шаблонов, так и от обычных классов, а также являться базовыми и для шаблонов, и для обычных классов
6. **Внутри шаблона нельзя определять friend-шаблоны**.

**Свойства параметров шаблона класса**

1. Параметры перечисляются через запятую, слово class (typename) требуется записывать перед каждым параметром. Template <class T1, class T2>
2. Параметрам шаблонного класса можно присваивать значения по умолчанию

Специализация шаблона:

template <>

class List <int>

1. **Шаблоны классов и дружественность. Пример.**

Дружественность может быть установлена:

* Между шаблоном класса и глобальной функцией.
* Методом другого класса (возможно, шаблонного) или даже целым классом (возможно, шаблонным).

Не параметризованная дружественная функция:

friend void f1 (int c) {}

Параметризованная дружественная функция:

friend void f2<T> (Myclass<T> &ob);

Дружественной для класса будет функция только для конкретного типа T (например, float)

Template <class T>

void f2 (Myclass<T> &ob);

Дружественные к шаблону класса методы другого класса

template <class T>

class X {… friend void A<**int**>::f4 (); //из класса А };

1. **Шаблоны классов и наследование. Пример**.
2. Обычный класс наследует шаблон класса

**class** **ItemExt** : **public** Item <**int**>{}

Когда Вы используете ItemExt, то вам не нужно указывать тип:

ItemExt int\_item;

**Класс не позволит использовать гибкость шаблонов**.

1. Шаблон класса наследует другой шаблон класса

**Template** <**typename** T>

**class** **SmartItem** : **public** Item <T> {};

Тогда при объявлении в main надо указать тип

SmartItem <int> c;

1. **Стандартная библиотека шаблонов. Контейнеры и итераторы. Примеры определения последовательных контейнеров. Заполнение контейнеров. Пример**

**Контейнеры** - объекты для хранения однотипных данных: последовательные (vector, deque, list); ассоциативные (set, multiset, map, multimap). В STL определены два основных типа контейнеров: **последовательности** и **ассоциативные** контейнеры. Разница между ними состоит в том, что для последовательностей имеет значение порядок следования элементов (вектор, стек, очередь), а для ассоциативных контейнеров – нет (ассоциативный массив, множество). В каждом классе-контейнере определён набор функций для работы с ними. **Ключевая идея для стандартных контейнеров** заключается в том, что они **должны быть взаимозаменяемыми**, если это разумно для решения задачи.

Хранение элементов контейнера:

* последовательно (непрерывным блоком) или
* разбросаны по всей памяти

Доступ к элементам контейнера

* по индексу (например,vector)
* только последовательным перебором (list)

**Вектор:** хранение элементов единым блоком; **доступ** к любому элементу **по индексу.**

**Очередь**: хранение элементов единым блоком; **доступ** только **к первому элементу**, внесенному в очередь.

**Список**: элементы разбросаны по памяти - каждый элемент содержит указатель на последующий и предыдущий; **доступ** - только перебор.

**Заполнение контейнеров** можно осуществить либо через итераторы, либо через push\_back().

**Итераторы** - объекты, которые используются для универсального доступа к элементам, хранящимся в контейнере любого типа. Они играют роль указателей, позволяют получить доступ к содержимому контейнера и сканировать его элементы.

1. **Стандартная библиотека шаблонов. Итераторы и примеры определения и работы с ними.**

**Итератор** – это обобщённый "указатель" на элемент, хранящийся в контейнере

list<double> ls; // объявлен список

list<double>::iterator it; // объявлен итератор

1) оператор \* – разыменование и получение доступа к значению элемента

(\*it)=5;

2) операторы «++» и «– –» получение указателя на следующий и предыдущий элемент

it++, it—

3) итератор на первый элемент контейнера

it=ls.begin()

4) итератор на **следующий после последнего** элемент контейнера

it=ls.end()

**Существует пять типов итераторов:**

1. Итераторы ввода (input iterator) поддерживают операции равенства, разыменования и инкремента: ==, !=, \*i, ++i, i++, \*i++. Специальным случаем итератора ввода является istream\_iterator.

2. Итераторы вывода (output iterator) поддерживают операции разыменования, допустимые только с левой стороны присваивания, и инкремента: ++i, i++, \*i = t, \*i++ = t. Специальным случаем итератора вывода является ostream\_iterator.

3. Однонаправленные итераторы (forward iterator) поддерживают все операции итераторов ввода/вывода и, кроме того, позволяют без ограничения применять присваивание: ==, !=, =, \*i, ++i, i++, \*i.

4. Двунаправленные итераторы (bidirectional iterator) обладают всеми свойствами forward-итераторов, а также имеют дополнительную операцию декремента (--i, i--, \*i--), что позволяет им проходить контейнер в обоих направлениях.

5. Итераторы произвольного доступа (random access iterator) обладают всеми свойствами bidirectional-итераторов, а также поддерживают операции сравнения и адресной арифметики, то есть непосредственный доступ к элементу по индексу: i += n, i + n, i -= n, i - n, i1 - i2, i[n], i1 < i2, i1 <= i2, i1 > i2, i1 >= i2.

1. **Стандартная библиотека шаблонов. Общие свойства контейнеров. Операции с контейнерами. Примеры (размер контейнера, проверка пустой или нет и др.)**

Ключевая идея для стандартных контейнеров заключается в том, что когда это представляется разумным, они должны быть логически взаимозаменяемыми. Пользователь может выбирать между ними, основываясь на соображениях эффективности и потребности в специализированных операциях. Например, если часто требуется поиск по ключу, можно воспользоваться map (ассоциативным массивом). С другой стороны, если преобладают операции, характерные для списков, можно воспользоваться контейнером list. Если добавление и удаление элементов часто производится в концы контейнера, следует подумать об использовании очереди queue, очереди с двумя концами deque, стека stack. По умолчанию пользователь должен использовать vector ; он реализован, чтобы хорошо работать для самого широкого диапазона задач.

**В STL определены следующие классы-контейнеры** (в угловых скобках указаны заголовочные файлы, где определены эти классы):

* bitset - множество битов <bitset.h>
* vector - динамический массив <vector.h>
* list - линейный список <list.h>
* deque - двусторонняя очередь <deque.h>
* stack - стек <stack.h>
* queue - очередь <queue.h>
* priority\_queue - очередь с приоритетом <queue.h>
* map - ассоциативный список для хранения пар ключ/значение, где с каждым ключом связано одно значение <map.h>
* multimap - с каждым ключом связано два или более значений <map.h>
* set - множество <set.h>
* multiset - множество, в котором каждый элемент не обязательно уникален <set.h>

**Доступ к элементам**:

* front() - ссылка на первый элемент
* back() - ссылка на последний элемент

**Включение элементов**:

* insert(p, x) - добавление х перед элементом, на который указывает р
* insert(p, n, x) - добавление n копий х перед р
* insert(p, first, last) - добавление элементов из [first:last] перед р
* push\_back(x) - добавление х в конец
* push\_front(x) - добавление нового первого элемента (только для списков и очередей с двумя концами)

**Удаление элементов**

* pop\_back() - удаление последнего элемента
* pop\_front() - удаление первого элемента (только для списков и очередей с двумя концами)
* erase(p) - удаление элемента в позиции р
* erase(first, last) - удаление элементов из [first:last]
* clear() - удаление всех элементов

**Другие операции**

* size() - число элементов
* empty() - контейнер пуст?
* capacity() - память, выделенная под вектор (только для векторов)
* reserve(n) - выделяет память для контейнера под n элементов
* resize(n) - изменяет размер контейнера (только для векторов, списков и очередей с двумя концами)
* swap(x) - обмен местами двух контейнеров
* ==, !=, < операции сравнения

1. **Стандартная библиотека шаблонов. Алгоритмы. Примеры работы со стандартными алгоритмами)**

**Алгоритмы** – обобщенные процедуры для обработки элементов любых контейнеров.

#include <algorithm>

Алгоритмы, не модифицирующие контейнер – процедуры поиска и сравнения.

list<string> ls;

list<string>::const\_iterator it;

// поиск значения "К8" в диапазоне от ls.begin() до ls.end()

it=find(ls.begin(), ls.end(), "К8");

Алгоритмы, **модифицирующие** **значение** элементов контейнера, но **не изменяющие порядок** их следования – выполнение действий над каждым элементом контейнера, **поиск и замена**

vector<int> v(100);

// **заполнение** всех элементов от ls.begin() до ls.end() значением 0

fill(v.begin(), v.end(), 0);

// **замена всех элементов** от ls.begin() до ls.end() равных -1 на 1

replace(v.begin(), v.end(), -1, 1);

Алгоритмы, **модифицирующие контейнер** – функции копирования, перестановок, удаления и сортировки, разбиения и слияния последовательностей

vector<int> v(100);

// сортировка массива

sort(v.begin(), v.end());

// перестановка элементов массива в обратном порядке

reverse(v.begin(), v.end());

**Функции-помощники** (перестановки и сравнения)

vector<int> v(100);

vector<int>::iterator it=v.begin();

it++;

swap(\*v.begin(), \*it);

Алгоритмы **min\_element**() и **max\_element**() находят минимальный и максимальный элементы в контейнере:

**for\_each -** выполнение пользовательской функции (в данном случае, вывода в консоль) для каждого элемента вектора.

**replace\_if -** замена с условием.

1. **Стандартная библиотека шаблонов. Последовательные контейнеры. Примеры vector**.

В шаблоне vector **определены операция присваивания** и **метод копирования** (assign).Векторы можно присваивать друг другу так же, как стандартные типы данных или строки. После присваивания размер вектора становится равным новому значению, все старые элементы удаляются.

Память под вектор выделяется динамически, но не под один элемент в каждый момент времени, а сразу под группу элементов, например 256 или 1024. Перераспределение памяти происходит только при превышении количества элементов, при этом объем выделенного пространства удваивается. **После перераспределения** любые **итераторы**, ссылающиеся на элементы вектора, становятся **недействительными**, поскольку вектор может быть перемещен в другой участок памяти и **нельзя ожидать, что связанные с ним ссылки будут обновлены автоматически**.

Пример.

include <vector>

using namespace std;

vector<double> x; // создание вектора

x.resize(10); // изменение размера вектора

x.resize(x.size()+100); // изменение размера вектора

double sum=0.0;

for(int i=0; i<x.size(); i++)

{

sum+=x[i]; //доступ по индексу к элементам вектора

}

**Функции работы с контейнером** – vector:

1. **Для любого объекта, который будет храниться в векторе**, **должен быть определен конструктор по умолчанию, определены операторы < и ==.**
2. Для класса вектор определены операторы сравнения ==, <, <=, !=, >, >=. Кроме этого, для класса vector определяется оператор индекса []. x[i]=5
3. Новые элементы могут включаться с помощью функций insert(), push\_back(), resize(), assign(). x.resize(100); int n=x.size(); x.push\_back(7)
4. Существующие элементы могут удаляться с помощью функций erase(), pop\_back(), resize(), clear(). x.clear(); //разрушает все элементы и освобождает контейнер
5. Доступ к отдельным элементам осуществляется с помощью итераторов begin(), end(), rbegin(), rend().
6. Определить пустой ли контейнер

bool res=x.empty(); // эквивалентно x.size() == 0

7. Выделить дополнительную память для размещения новых элементов

x.reserve(200);

1. **Стандартная библиотека шаблонов. Последовательные контейнеры. Примеры list.**

**Список** – структура данных для организации хранения элементов с эффективной вставкой и удаления в любом месте этой структуры, не требует доступа по индексу

**УЗЕЛ (Node)** – объект, который содержит ссылки на последующий и предыдущий элементы, а также значение данного элемента списка

В STL – простой двухсвязный список

list<int> example; //создание списка

example.push\_back(0); // вставка в конец

example.push\_front(1); // вставка в начало

example.insert(x.begin(), 3); // вставка в любое место списка

**Вопросы к экзамену по курсу ЛИПО САПР (2 часть)**

1. **Технологии разработки печатных плат (однослойные, двухсторонние, МПП).**

**Печатные платы** – это элементы конструкции, которые состоят из плоских проводников в виде **участков металлизированного покрытия**, размещённых **на диэлектрическом основании** и обеспечивающих соединение элементов электрической цепи. Они получили широкое распространение в производстве модулей, ячеек и блоков РЭА.

Достоинства печатных плат

* Увеличение плотности монтажных соединений и возможность микроминиатюризации изделий.
* Гарантированная стабильность и повторяемость электрических характеристик
* Повышенная стойкость к климатическим и механическим воздействиям
* Унификация и стандартизация конструктивных и технологических решений
* Увеличение надёжности
* Возможность комплексной автоматизации монтажно-сборочных и контрольно-регулировочных работ

Элементами печатной платы являются

* Диэлектрическое основание (подложка)
* Металлическое покрытие в виде рисунка печатных проводников и контактных площадок (трассы)
* Монтажные и фиксирующие отверстия
* Контактные площадки-участки металлического покрытия, которые соединяют печатные проводники с металлизацией монтажных отверстий.

**Односторонние печатные платы**. За один технологический цикл получается вся конструкция с монтажными отверстиями и специальными углублениями для расположения элементов, монтируемых на поверхности платы.

**Двусторонние печатные платы**. Два слоя. Электрическая связь слоёв печатного монтажа осуществляется с помощью расположенных на плате металлизированных отверстий. Двусторонние ПП обладают повышенной плотностью монтажа и надёжностью соединений. Они используются в измерительной технике, системах управления и автоматического регулирования.

**Многослойные печатные платы**. Состоят из чередующихся слоёв изоляционного материала и проводящего рисунка, соединённых клеевыми прокладками в монолитную структуру путём прессования. Электрическая связь между проводящими слоями выполняется специальными объёмными деталями, печатными элементами или химико-гальванической металлизацией. По сравнению с ОПП и ДПП они характеризуются повышенной надёжностью и плотностью монтажа, устойчивостью к механическим и климатическим воздействиям, уменьшением размеров.

**Методы изготовления печатных плат**.

**Субтрактивный** – в качестве основания для печатного монтажа используют фольгированные диэлектрики, на которых формируется проводящий рисунок путём удаления фольги с непроводящих участков.

**Аддитивные методы** – основаны на избирательном осаждении токопроводящего покрытия на диэлектрическое основание, на которое предварительно может наноситься слой клеевой композиции.

1. **Модели схем в конструкторском проектировании. Модели элементов, модели цепей. Примеры.**

Модели схемы традиционно **граф**, но вид графа, ориентированный он или нет, обычный или **мультиграф** (граф, где пара вершин соединяется более чем одним ребром), **полный** граф (граф, где каждая вершина соединена со всеми остальными) или **гиперграф** (граф, в котором одно ребро содержит множество вершин) – всё это связано с решаемой задачей.

Модели схем для разных задач конструирования должны удовлетворять разным требованиям. Универсальная модель утяжеляет алгоритмы автоматизации. Обычно рассматривают модели элементов и модели цепей.

Традиционно в качестве модели элемента рассматривают **вершину графа**, если необходимо учитывать контакты элементов, то для **двухконтактных** элементов модель – **это отрезок, соединяющий две вершины. Для многополюсных элементов**, где контактов много, моделью элемента является **звезда**, в центре звезды – вершина элемент, по периферии от центра к вершинам – контактам элемента проводятся отрезки лучи. Иногда используется более жёсткая конструкция для многополюсных элементов – **многоугольник** – число вершин равно числу контактов элемента, в центре многоугольника сам элемент, который соединён ребром с каждой вершиной – контактом элемента.

**Модели цепей**. Часто – это **ребро**, если есть соединение между элементами ребро есть, ребро отсутствует, когда нет соединения. Такая модель не учитывает, что два элемента могут быть связаны по разным контактам разными электрическими цепями, чтобы это учесть вводили смежные рёбра, то есть моделью схемы являлся **мультиграф**. Однако, мультиграф не учитывал специфику цепи, объединяющей несколько контактов, **здесь вообще контакты не учитывались.**

1. **Задача компоновки, классификация, критерии. Используемые модели, в задаче компоновки.**

**Компоновка** – это назначение элементов в корпуса и назначение корпусов на платы.

1. **Алгоритмы компоновки, классификация. Последовательный алгоритм разрезания мультиграфа.**
2. **Итерационные алгоритмы компоновки. Пример.**

Варианты итерационных алгоритмов:

1. Алгоритмы парных перестановок.
2. Алгоритмы групповых перестановок.

**В первом случае** в графе, разрезанном на подграфы (части), производится парная перестановка элементов из разных подграфов с проверкой на каждом шаге приращения числа рёбер, соединяющих части. Цель итерации – минимизация числа соединяющих рёбер.

**Во втором случае** перестановки идут между группами элементов.

Итерационный алгоритм включает следующие шаги:

1. Из множества подграфов разрезания выделяют два любых G1(X1,U1) и G2(X2,U2). Для каждой пары вершин и оценивается разность между числом новых внешних связей (после перестановки) и старых до перестановки △S = Sпосле – Sдо.
2. Те пары и , для которых △S<0 заносятся в список кандидатов для обмена.
3. Из списка выбирается та пара, для которой |△S| максимально, если таких несколько, то переставляется любая пара.
4. Переставленные пары отмечаются и больше в перестановке не участвуют.
5. Для вновь выбранных подграфов повторяется процесс из п.1.
6. Повторяются п.п. 2, 3, 4 до тех пор, пока в списке замен не будет пар, для которых △S<0 (или по времени, или по числу итераций).
7. Далее устраняются все метки вершин G1 и G2, выбирается другая пара подграфов и повторяются п.п. 1-6, то есть минимизируется число соединений для вновь выбранных подграфов. Процесс продолжается до тез пор, пока не будет сделана перестановка во всех парах подграфов графа G(X, U).
8. **Гиперграф как модель схемы в алгоритмах разрезания. Сравнение с моделью мультиграфа.**
9. **Формализация задачи разрезания гиперграфа.**
10. **Алгоритм последовательного разрезания на блоки (модель схемы гиперграф).**
11. **Задача компоновки. Алгоритмы покрытия. Классификация.**

Задача покрытия.

Первым этапом технического проектирования печатных плат является задача распределения базовых элементов (БЭ) логической схемы по корпусам микросхем (ИС). Её можно решать в такой последовательности:

* Определить оптимальный набор ИС для проектирования схемы из числа разрешённых на данном предприятии.
* Распределить БЭ по корпусам ИС таким образом, чтобы минимизировать число межмодульных связей.

Мы рассмотрим случай, когда оптимальный набор есть. Это задача покрытия.

* Исходная схема представляет собой множество связанных функциональных элементов И, ИЛИ, НЕ, триггеров, усилителей, генераторов и т.д. Каждый функциональный элемент реализует некоторую функцию. Их множество определяет совокупность всех функций, выполняемых в схеме.
* Схема представляется взвешенным графом G(X, U), где Х – множество вершин элементов, U – рёбер, **каждой вершине ставится в соответствие вес, соответствующий выполняемой функции**. Задан ограниченный набор (библиотека) типов модулей N = {n1, n2, … , np}, где p – число типов конструктивных модулей в библиотеке (ячеек, ИС, ЭРЭ ПЛИС, ГИС, БИС). Здесь ni – количество элементов i-го типа. Каждый из группы модулей реализует **одну или несколько функций** (имеет определённый заранее состав функциональных элементов). Эти функции соответствуют по функционалу тем, которые в схеме.
* Каждый конструктивный модуль из библиотеки можно представить подграфом Gi=n (Yi, Ui), в котором вершины отображают функциональный элементы, входящие в модуль. Рёбра Ui – реально могут быть или отсутствовать, если это модули-наборы, как, например, в ПЛИС.
* Требуется разбить исходную схему, то есть граф G(X, U) на подграфы Gi(Xi,Ui) **изоморфные Gi(Yi,Ui)**. При выполнении следующих условий:

1. Каждая подсхема Gi(Xi,Ui) по своему функциональному составу должна соответствовать некоторому модулю Gi(Yi,Ui)
2. Любые две подсхемы G(X,U) не пересекаются.

**Оценка качества покрытия**: число **типов** модулей, используемых для покрытия должно быть минимально, а полнота использования конструктивных **модулей максимальна, общее** число всех модулей для покрытия минимально.

**Эвристический алгоритм задачи покрытия**

Для каждого подграфа Gi(Xi,Ui) – он соответствует некоторой микросхеме, в графе G(X,U) ищут изоморфные подграфы, найденные подграфы удаляются из графа G, если после этого остаются непокрытые части Hj, то задача далее решается для каждой части отдельно. Процесс продолжается до тех пор, пока вся схема не окажется покрытой модулями из заданного набора. Качество покрытия как всегда в последовательных алгоритмах зависит от порядка выбора элементов из библиотеки модулей. Улучшение покрытия возможно итерационным алгоритмом.

1. **Последовательно - приближенный алгоритм покрытия схемы по шагам (связные модули).**

Приближённый алгоритм покрытия связными модулями по шагам.

1. Ранжирование схемы.
2. Выбор начального элемента.
3. Выбор покрывающего блока.
4. Выбор элемента в покрывающем блоке.
5. Построение единичной окрестности вокруг покрывающего и покрываемого элементов.
6. Определение пересечения окрестностей. Если пустое, то к п.7, иначе проверка корректности пересечения с точки зрения функционирования схемы, хватит ли выводов, все ли цепи покрываемой схемы могут быть реализованы. Если пересечение некорректно, то к п.7, иначе запоминаем его. Запоминаем пересечение, если модуль не исчерпан, то к п.4, если исчерпан, то к п.8.
7. Если покрывающий блок использован не полностью, то к п.4, иначе к п.9.
8. Выбор наилучшего пересечения из числа запомненных пересечений.
9. Если схема ещё не покрыта, то к п.2, иначе конец.
10. **Покрытие схемы модулями “россыпью”.**

Если есть множество модулей G1, G2, … , Gm. Каждый из этих модулей состоит из несвязанных функциональных частей Gi = {g1, g2, … , gn}, в модулях функциональные элементы могут повторяться. Задача покрытия сводится

1. Определение вспомогательного набора покрывающих элементов из всех модулей {Gi}, то есть подбираются модули, содержащие gi, подходящего типа для покрытия функциональных элементов исходной схемы.
2. Решается задача покрытия схемы блоками из вспомогательного набора b1, b2, … , bn, для чего определяется, сколько каких блоков необходимо взять, чтобы были покрыты соответствующие функциональные схемы. Эта задача решается последовательным алгоритмом. Результатом решения является схема, элементами который являются блоки вспомогательного набора b1, b2, … , bn. Каждое значение b1 вспомогательного набора показывает, сколько надо в схеме иметь блоков типа b1.
3. Так как типы блоков в разных модулях библиотеки могут повторяться, то следующей задачей является определение количества библиотечных модулей из множества библиотечных модулей {Gj} каждого типа, которые надо иметь, чтобы решить с их помощью задачу покрытия схемы. При этом минимизируется суммарная стоимость используемых модулей.
4. **Размещение элементов. Классификация алгоритмов размещения и критерии качества.**

**Размещение** – это определение местоположения элементов на коммутационном поле (КП) конструктивного модуля, такого, при котором создаются наилучшие условия для решения последующей задачи трассировки соединений с учётом конструкторско-технологических требований и ограничений.

Типы алгоритмов размещения:

* **Силовые**. Процесс размещения элементов на плате представляется как движение к состоянию равновесия системы материальных точек (элементов), на каждую из которых действуют силы притяжения и отталкивания, интерпретирующие связи между размещаемыми элементами. Задача оптимального размещения элементов сводится к нахождению такого местоположения точек, при котором равнодействующие всех сил обращаются в ноль.
* **Последовательные**. Они основаны на допущении, что для получения оптимального размещения необходимо в соседних позициях располагать элементы, максимально связанные друг с другом. Сущность этих алгоритмов состоит в последовательном закреплении заданного набора конструктивных элементов на коммутационной плате относительно ранее установленных.
* **Итерационные**. Для улучшения исходного размещения элементов на плате вводят итерационный процесс перестановки местами пар элементов. В итерационном алгоритме парных перестановок каждый элемент меняется с каждым другим и оценивается уменьшается ли при этом средняя длина соединений, фиксируется та перестановка, которая даёт максимальное уменьшение.

**Основной критерий качества**.

Наиболее часто используется критерий минимальной суммарной длины.

Расстояния при этом могут рассчитываться по следующим формулам:

dij = , что соответствует прокладке соединения по кратчайшему пути.

Dij = || + || - когда соединения прокладываются по каналам и магистралям параллельно осям коориднат.

1. **Последовательно-итерационный алгоритм размещения. Последовательный этап. Пример.**

На первом этапе последовательно по известной схеме соединений, заданной мультиграфом и перечню номеров элементов, которые назначены на данную плату в результате решения задачи разрезания, заполняются установочные места на плате. Плата задаётся числом посадочных мест и матрицей расстояний между заданными на плате установочными местами. На каждом шаге последовательного алгоритма выбирается пара, **позиция-элемент** с учётом связей между элементами и оценкой суммарной длины соединений для каждой позиции кандидата для размещения очередного элемента на позицию. Этап заканчивается, когда все элементы размещены.

**Заполнение монтажного пространства**

Размещение производится с помощью оценки коэффициента связность K(Vi), элемента – кандидата на размещение Vi с ранее размещёнными на плате элементами, определяемого по формуле:

K(Vi) = p (vi ∈ Vразм) - p (vi ∈ Vост) = 2 p (vi ∈ Vразм) – P(Vi) = 2 – p(vi)

* – число связей элемента Vi с подмножеством размещённых вершин Vразм
* p (vi ∈ Vост) – число связей элемента Vi с подмножеством оставшихся вершин, оно равно P(Vi) - p (vi ∈ Vразм)
* Jразм – множество размещённых вершин
* p (vi) – степень вершины Vi

В любую свободную позицию размещаются один (первый) конструктивный элемент, далее производится размещение элементов с максимальными значениями коэффициента связности в очередных свободных позициях. Обычно сначала рассматриваются позиции вокруг размещённых элементов.

Например: K1(V2) = 2S21 – P(V2) = …

S21 – количество рёбер между вершинами V1 и V2

P(V2) – степень вершины V2 (общее количество рёбер, входящих в V2)

1. **Последовательно-итерационный алгоритм размещения. Итерационный этап. Пример.**

На втором этапе проводится перестановка элементов с целью минимизации выбранного критерия оптимальности.

* + - 1. Зададим множество конструктивных элементов V и связи между ними графом G(V,R), с помощью матрицы смежности S = {Sij} n × n, где n – число элементов на плате.
      2. Монтажное пространство задаётся множеством установочных мест T в виде матрицы длин L или списка координат позиций. Некоторые элементы могут быть закреплены за отдельными позициями.

Найдём элемент, который имеет максимальную среднюю длину соединений относительно того места, где он находится. Определим среднюю длину связи элемента Vi, расположенного в позиции tk.

Lvi(tk) = – это средняя длина связей между элементами i, j, размещённых соответственно в позициях k и позиции r, тогда для элемента V1, размещённого в позиции t1 средняя длина определяется по формуле:

Lv1(t1) =

l – расстояние между элементами (считается по осям x и y)

Считаем длину связи для всех элементов, выбираем тот, у которого она наибольшая (Vc)

Считаем его координаты его центра тяжести:

Xi(Vk) =

Yi(Vk) =

Где X и Y – заданы на плате

Подставляем элемент в позицию с полученными координатами, если их несколько меняют сначала с одним и считают новую его среднюю длину, потом также с другим, после чего оценивается изменения средней длины от перестановки:

∆L(Vq, Vc) = Lvq(Tq) - Lvq(Tc(c)) + Lvc(Ti) - Lvc(Tk(q))

Перестановки заканчиваются, когда нет уменьшения длин связей.

1. **Алгоритмы трассировки. Общая характеристика задачи, зависимость критериев качества от технологии реализации элементов.**
2. **Алгоритм Прима. Компьютерный алгоритм, пример по шагам.**
3. **Формализация задачи трассировки. Коммутационное поле и размер дискрета, соседство, приоритет координат, функции качества алгоритма. Классификация алгоритмов.**
4. **Волновой алгоритм, исходные данные, основные шаги. Пример. Достоинства и недостатки волнового алгоритма.**
5. **Основные составляющие волнового алгоритма, позволяющие учитывать заданные ограничения разработчика схемы. Соседство, путевые координаты, весовые функции. Примеры. Способы улучшения волнового алгоритма.**
6. **Построение путей волновым алгоритмом по заданным параметрам (мин изгибов, мин пересечений и т.д.)**
7. **Топологическая и гибкая трассировка достоинства и недостатки. Модель схемы в гибкой трассировке.**
8. **Многослойная трассировка. МПП с открытыми контактными площадками. Основные задачи автоматизации. Отличия по сравнению с МПП.**
9. **Графо-теоретический подход к трассировке. Модели элементов. Основные решаемые задачи.**